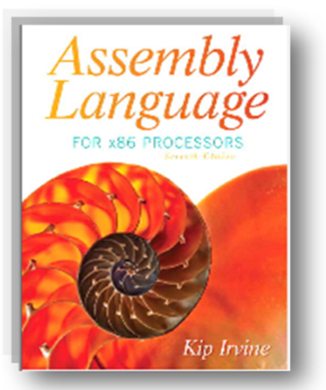
Basic Concepts

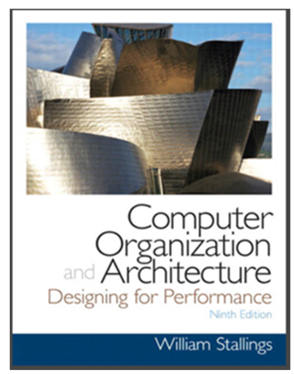
Contents of Lecture:

- Architecture & Organization
- Introduction to Assembly Language
- Virtual Machine Concept
- Data Representation
- Boolean Expressions

References for course:

- ✓ *KIP R. IRVINE, Assembly Language for x86 Processors, 7th Edition*
- ✓ *William Stallings, Computer Organization and Architecture Designing For Performance, 9th Edition*





Organization and Architecture:

- Architecture is those attributes visible to the programmer
  - ✓ Instruction set, number of bits used for data representation, I/O mechanisms, addressing techniques.

- Organization is how features are implemented
  - ✓ Control signals, interfaces, and techniques for addressing memory.

- All Intel x86 family share the same basic architecture
- The IBM System/370 family share the same basic architecture
- This gives code compatibility
- Organization differs between different versions

Computer Components:-

1) **A processor** to interpret and execute programs.
2) **A memory** to store both data and programs.
3) **A mechanism** for transferring data to and from the outside world.

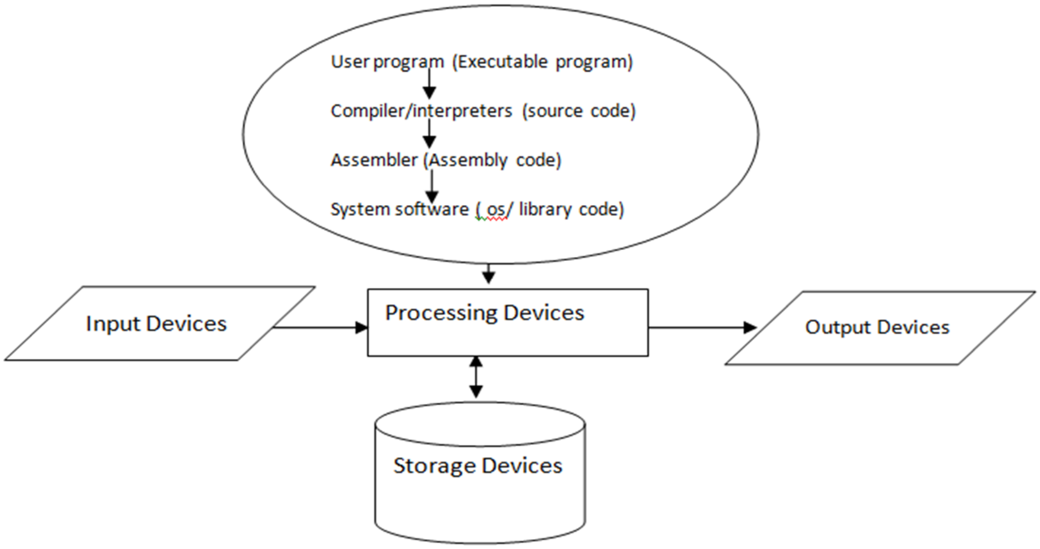


Figure (1) shows computer components and their interrelationships

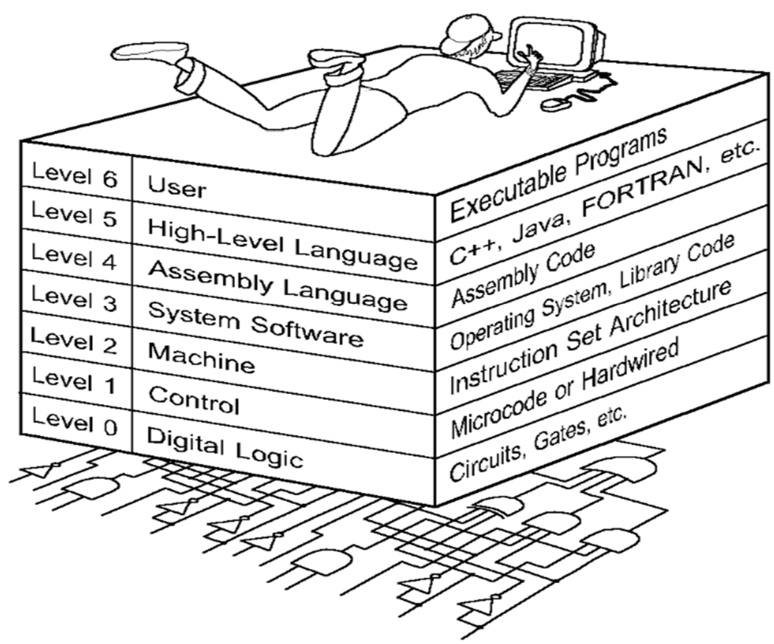


Figure (2) The computer level Hierarchy

- **Level 6: The User Level**
  - ✓ Program execution and user interface level.

- **Level 5: High-Level Language Level**
  - ✓ The level with which we interact when we write programs in languages such as C, Pascal, Lisp, and Java.

- **Level 4: Assembly Language Level**
  - ✓ Acts upon assembly language produced from Level 5, as well as instructions programmed directly at this level.

- **Level 3: System Software Level**
  - ✓ Controls executing processes on the system.
  - ✓ Protects system resources.
  - ✓ Assembly language instructions often pass through Level 3 without modification.

- **Level 2: Machine Level**
  - ✓ Also known as the **Instruction Set Architecture** (**ISA**) Level.
  - ✓ Consists of instructions that are particular to the architecture of the machine.
  - ✓ Programs written in machine language need no compilers, interpreters, or assemblers.

- **Level 1: Control Level**
  - ✓ A ***control unit*** decodes and executes instructions and moves data through the system.
  - ✓ Control units can be ***microprogrammed*** or ***hardwired***.
  - ✓ A **microprogram** is a program written in a low-level language that is implemented by the hardware.
  - ✓ **Hardwired** control units consist of hardware that directly executes machine instructions.

- **Level 0: Digital Logic Level**
  - ✓ This level is where we find digital circuits (the chips).
  - ✓ Digital circuits consist of gates and wires.
  - ✓ These components implement the mathematical logic of all other levels.

Introduction to Assembly Language:

- **Assembly language** is the oldest programming language, and of all languages, bears the closest resemblance to native machine language.

  - ✓ Assembly language is a low-level programming language for a computer.
  - ✓ It provides direct access to computer hardware, requiring you to understand much about your computer's architecture and operating system.

Questions You Might Ask:

- **What background should I have?**
  - ✓ Before learning Assembly language, you should have programmed in at least one structured high-level language, such as Java, C, or C++.
  - ✓ You should know how to use IF statements, arrays, and functions to solve programming problems.

- **What hardware and software do I need?**
  - ✓ Need a computer that runs a 32-bit or 64-bit version of Microsoft Windows, along with one of the recent versions of Microsoft Visual Studio.

- **What types of programs will I create?**
  - ✓ **32-Bit Protected Mode:** 32-bit protected mode programs run under all 32-bit versions of Microsoft Windows. They are usually easier to write and understand than real-mode programs.
  - ✓ **64-Bit Mode:** 64-bit programs run under all 64-bit versions of Microsoft Windows.
  - ✓ **16-Bit Real-Address Mode:** 16-bit programs run under 32-bit versions of Windows and on embedded systems.

- **How does assembly language relate to machine language?**
  - ✓ Machine language is a numeric language specifically understood by a computer's processor (the CPU). All x86 processors understand a common machine language.
  - ✓ Assembly language consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL.
  - ✓ Assembly language has a **one-to-one relationship** with machine language: Each assembly language instruction corresponds to a single machine-language instruction.

- **How do C++ and Java relate to assembly language?**
  - ✓ High-level languages such as C, C++, and Java have a **one-to-many relationship** with assembly language and machine language.
  - ✓ A single statement in C++, for example, expands into multiple assembly language or machine instructions.

- **Is assembly language portable?**
  - ✓ A language whose source programs can be compiled and run on a wide variety of computer systems is said to be portable.
  - ✓ Assembly language is **not portable**, because it is designed for a specific processor family.

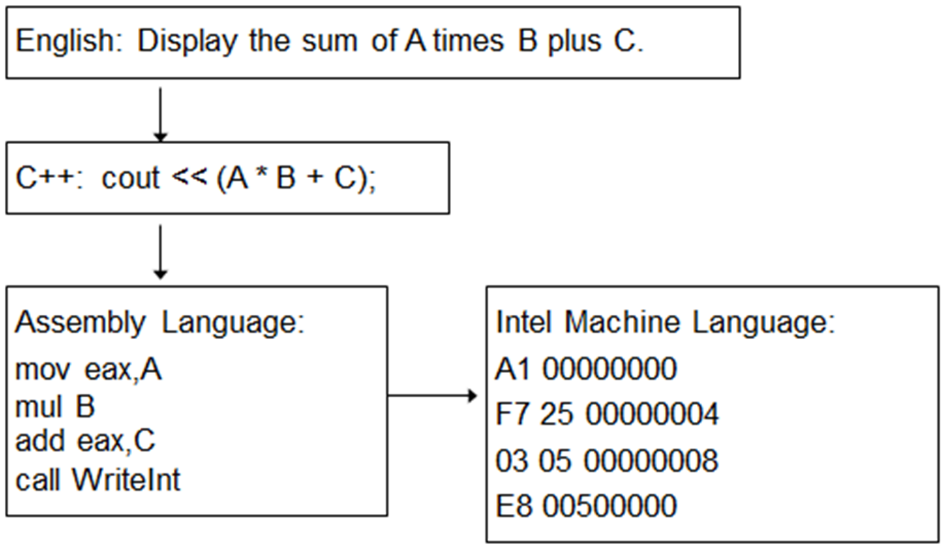
- **Why learn assembly language?**
  - ✓ To learn how high-level language code gets translated into machine language (learn the details hidden in HLL code).
  - ✓ To learn the computer's hardware; by direct access to memory, video controller, sound card, keyboard…
  - ✓ To speed up applications; provide direct access to hardware (ex: writing directly to I/O ports instead of doing a system call)
  - ✓ Speed. Assembly language programs are generally the fastest programs around. Good ASM code is faster and smaller.
  - ✓ Space. Assembly language programs are often the smallest.

<u>Virtual Machine Concept:</u>

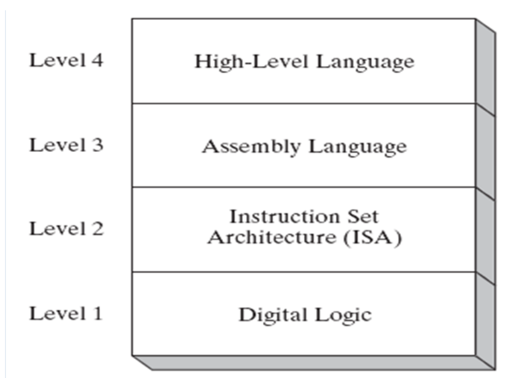
- A computer can usually execute programs written in its native machine language. Each instruction in this language is simple enough to be executed using a relatively small number of electronic circuits. For simplicity, we will call this language **L0**.

- Programmers would have a difficult time writing programs in **L0** because it is enormously detailed and consists purely of numbers. If a new language, **L1**, could be constructed that was easier to use, programs could be written in **L1**. There are two ways to achieve this:
  - ✓ **Interpretation: L0** program interprets and executes **L1** instructions one by one.
  - ✓ **Translation: L1** program is completely translated into an **L0** program, which then runs on the computer.

Translating Languages:



Specific Machine Levels:



- **High-Level Language:**
  - Level 4
  - Application-oriented languages: C++, Java, Pascal, Visual Basic . . .
  - Programs compile into assembly language (Level 3)

- **Assembly Language:**
  - Level 3
  - Instruction mnemonics that have a one-to-one correspondence to machine language
  - Programs are translated into Instruction Set Architecture Level - machine language (Level 2)

- **Instruction Set Architecture (ISA):**
  - ✓ Level 2
  - ✓ Also known as conventional machine language
  - ✓ Executed by Level 1 (Digital Logic)

- **Digital Logic:**
  - ✓ Level 1
  - ✓ CPU, constructed from digital logic gates
  - ✓ System bus
  - ✓ Memory
  - ✓ Implemented using bipolar transistors

Data Representation:

- Assembly language programmers deal with data at the physical level, so they must be adept at examining memory and registers. Often, binary numbers are used to describe the contents of computer memory; at other times, decimal and hexadecimal numbers are used. You must develop a certain number formats, so you can quickly translate numbers from one format to another.

- Each numbering format, or system, has a base, or maximum number of symbols that can be assigned to a single digit.

Binary, Octal, Decimal, and Hexadecimal Digits.

System	Base	Possible Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Binary Numbers:

- Digits are 1 and 0
  - ✓ 1 = true
  - ✓ 0 = false
- The bit on the **left** is called the ***most significant bit (MSB)***, and the bit on the **right** is the ***least significant bit (LSB)***.
- The MSB and LSB bit numbers of a 16-bit binary number are shown in the following figure:

- Binary integers can be:
  - A **signed**: A signed integer is **positive** or **negative**
  - An **unsigned**: An unsigned integer is by default **positive**.
  - **Zero** is considered positive.

- When writing down large binary numbers, many people like to insert a dot every 4 bits or 8 bits to make the numbers easier to read.
  - Examples are 1101.1110.0011.1000.0000 and 11001010.10101100.

Unsigned Binary Integers:

- Starting with the LSB, each bit in an unsigned binary integer represents an increasing power of 2.
- The following figure contains an 8-bit binary number, showing how powers of two increase from right to left:

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

Binary Bit Position Values.

2^n	Decimal Value	2^n	Decimal Value
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

Translating Unsigned Binary Integers to Decimal:

- Weighted positional notation represents a convenient way to calculate the decimal value of an unsigned binary integer having n digits:

$$dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + ... + (D_1 \times 2^1) + (D_0 \times 2^0)$$

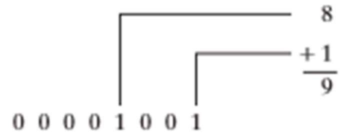
Where:

D = binary digit

- **For example**:
  - Binary 00001001 is equal to 9. We calculate this value by leaving out terms equal to zero:

$$(1 \times 2^3) + (1 \times 2^0) = 9$$

✓ The same calculation is shown by the following figure:



Translating Decimal Integers to Binary:

❖ Repeatedly divide the decimal integer by 2. Each remainder is a binary digit in the translated value:

Division	Quotient	Remainder
37 / 2	18	1
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1

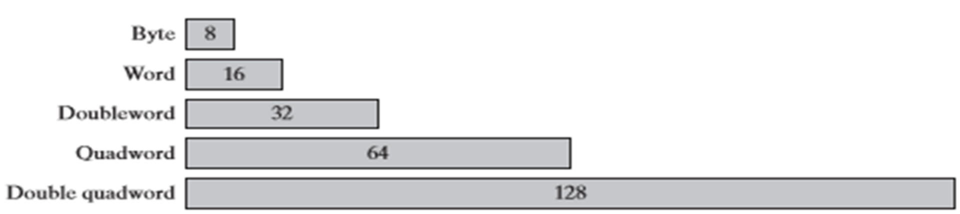
37 = 100101

Binary Addition:

❖ Starting with the LSB, add each pair of digits, include the carry if present.

Integer Storage Sizes:

❖ The basic storage unit for all data in an x86 computer is a byte, containing 8 bits. Other storage sizes are in the following figure:



Ranges and Sizes of Unsigned Integer Types.

Type	Range	Storage Size in Bits
Unsigned byte	0 to $2^8 - 1$	8
Unsigned word	0 to $2^{16} - 1$	16
Unsigned doubleword	0 to $2^{32} - 1$	32
Unsigned quadword	0 to $2^{64} - 1$	64
Unsigned double quadword	0 to $2^{128} - 1$	128

❖ **Large storage size:**

- One *kilobyte* is equal to 2^{10}, or 1024 bytes.
- One *megabyte* (1 MByte) is equal to 2^{20}, or 1,048,576 bytes.
- One *gigabyte* (1 GByte) is equal to 2^{30}, or 1024^3, or 1,073,741,824 bytes.
- One *terabyte* (1 TByte) is equal to 2^{40}, or 1024^4, or 1,099,511,627,776 bytes.
- One *petabyte* is equal to 2^{50}, or 1,125,899,906,842,624 bytes.
- One *exabyte* is equal to 2^{60}, or 1,152,921,504,606,846,976 bytes.
- One *zettabyte* is equal to 2^{70} bytes.
- One *yottabyte* is equal to 2^{80} bytes.

Hexadecimal Integers:

❖ The following table shows how each sequence of four binary bits translates into a decimal or hexadecimal value.

Binary, Decimal, and Hexadecimal Equivalents.

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Translating Binary to Hexadecimal:

- Each hexadecimal digit corresponds to 4 binary bits.
- **Example**: Translate the binary integer 000101101010011110010100 to hexadecimal:

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

Converting Hexadecimal to Decimal:

- Multiply each digit by its corresponding power of 16:

$$dec = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$$

- Hex 1234 equals $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$, or decimal 4,660.
- Hex 3BA4 equals $(3 \times 16^3) + (11 \times 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal 15,268.

lists the powers of 16 from 16^0 to 16^7.

16^n	Decimal Value	16^n	Decimal Value
16^0	1	16^4	65,536
16^1	16	16^5	1,048,576
16^2	256	16^6	16,777,216
16^3	4096	16^7	268,435,456

Converting Unsigned Decimal to Hexadecimal:

- Repeatedly divide the decimal value by 16 and retain each remainder as a hexadecimal digit.
- For example, the following table lists the steps when converting decimal 422 to hexadecimal:

Division	Quotient	Remainder
422 / 16	26	6
26 / 16	1	A
1 / 16	0	1

Decimal 422 = 1A6 hexadecimal

Character Storage:

❖ **ASCII** Code (7-bit) **American Standard Code for Information Interchange**.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Boolean Operations:

- A boolean expression involves a boolean operator and one or more operands. Each boolean expression implies a value of ***true*** or ***false***.

- Boolean expressions created from the set of operators includes the following:
  - ✓ NOT: notated as ¬ or ~ or ’
  - ✓ AND: notated as ∧ or *
  - ✓ OR: notated as ∨ or +

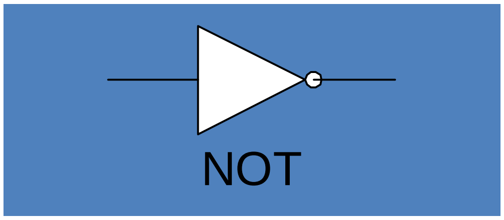
Expression	Description
$\neg X$	NOT X
$X \wedge Y$	X AND Y
$X \vee Y$	X OR Y
$\neg X \vee Y$	(NOT X) OR Y
$\neg(X \wedge Y)$	NOT (X AND Y)
$X \wedge \neg Y$	X AND (NOT Y)

NOT:

- Inverts (reverses) a boolean value
- One operand
- Truth table for Boolean NOT operator:

X	¬X
F	T
T	F

- Digital gate diagram for NOT:

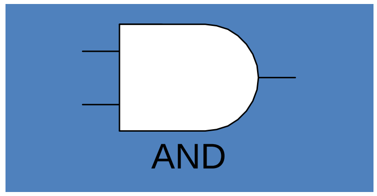


AND:

- Two operands
- Both must be T for T, otherwise F
- Truth table for Boolean AND operator:

X	Y	$X \wedge Y$
F	F	F
F	T	F
T	F	F
T	T	T

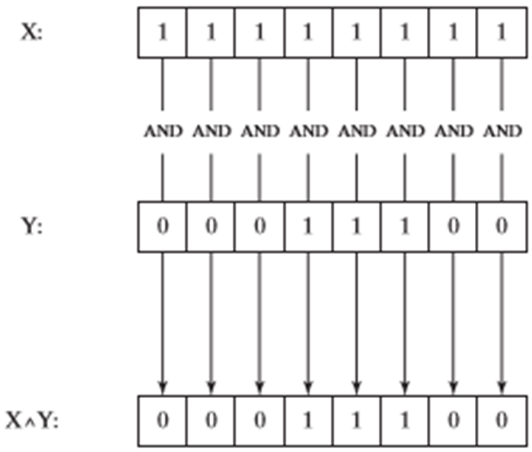
- Digital gate diagram for AND:



- Example:
  - ✓ ANDing the bits of **X ∧ Y** if **X** = 11111111 and **Y** = 00011100

X:	11111111
Y:	00011100
X ∧ Y:	00011100

- ANDing the bits of two binary integers:

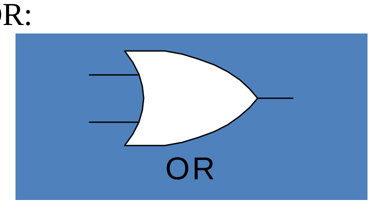


OR:

- Two operands
- Both must be F for F, otherwise T
- Truth table for Boolean OR operator:

X	Y	X ∨ Y
F	F	F
F	T	T
T	F	T
T	T	T

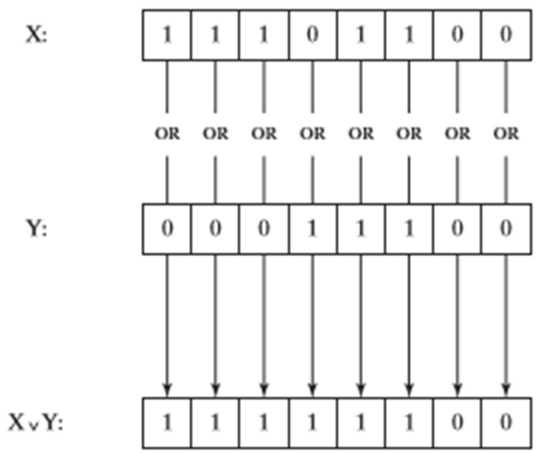
- Digital gate diagram for OR:



- Example:
  - ✓ ORing the bits of **X ∨ Y**if **X** = 11101100 and **Y** = 00011100

    X: 11101100
    Y: 00011100
    X ∨ Y: 11111100

- ORing the bits of two binary integers:

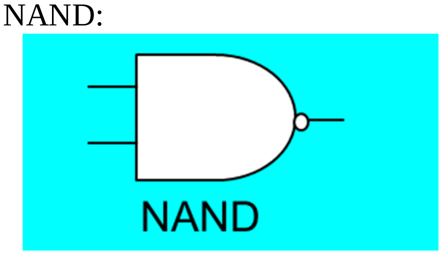


NAND:

- Two operands
- Both T = F, otherwise T
- Truth table for Boolean NAND operator:

X	Y	X NAND Y
F	F	T
F	T	T
T	F	T
T	T	F

- Digital gate diagram for NAND:



NOR:

- Two operands
- Any T = F, otherwise T
- Truth table for Boolean NOR operator:

X	Y	X NOR Y
F	F	T
F	T	F
T	F	F
T	T	F

- Digital gate diagram for NOR: