Chapter 1

Basic Concepts

Chapter Overview

- Welcome to Assembly Language
- Virtual Machine Concept
- Data Representation
- Boolean Operations

Welcome to Assembly Language *(cont)*

How does assembly language (AL) 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.

Welcome to Assembly Language *(cont)*

- How do C++ and Java relate to AL?

High-level languages such as Python, C++, and Java have a one-to-many relationship with assembly language and machine language. This relationship implies that a single statement in C++, for example, expands into multiple assembly languages or machine instructions. The following C++ code carries out two arithmetic operations and assigns the result to a variable. Assume X and Y are integers:

```
//Assume x and y have been declared already
x = y + 5
```

Following is the equivalent translation to assembly language. The translation requires multiple statements because each assembly language statement corresponds to a single machine instruction:

```
;Assume x and y have been declared already
mov   bl, y ; move Y to the BL register
add   bl, 5 ; add 4 to the BL register
mov   x, bl ; move BL to X
```

Registers: Small storage areas within a CPU that permit rapid manipulation of data

Welcome to Assembly Language *(cont)*

Is AL portable?

- Assembly language is not portable, because it is designed for a specific processor family. There are a number of different assembly languages widely used today, each based on a processor family. Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370.

Assembly Language Applications

In the early days of programming, most applications were written partially or entirely in assembly language. They had to fit in a small area of memory and run as efficiently as possible on slow processors. As memory became more plentiful and processors dramatically increased in speed, programs became more complex. high-level languages became possible: C, COBOL, C++, Python…

- Some representative types of applications:
  - Business application for single platform
  - Hardware device driver
  - Business application for multiple platforms
  - Embedded systems & computer games

Comparing ASM to High-Level Languages

It takes too much time to write and maintain assembly language programs. Instead, assembly language is used to optimize certain sections of application programs for speed and to access computer hardware.

Type of Application	High-Level Languages	Assembly Language
Business application software, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	Language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Business application written for multiple platforms (different operating systems).	Usually very portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, often using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	Produces too much executable code, and may not run efficiently.	Ideal, because the executable code is small and runs quickly.

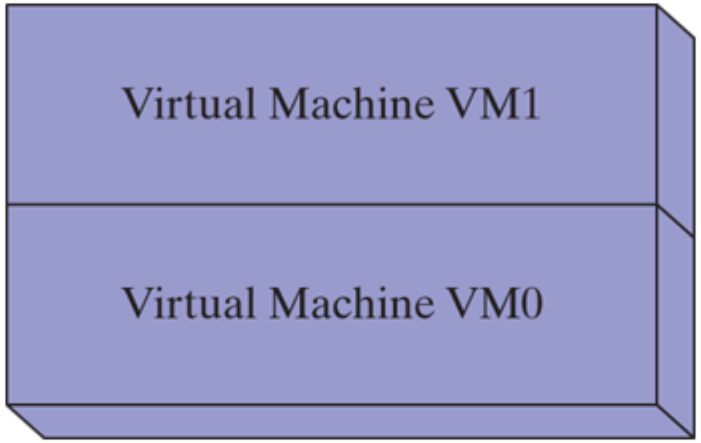
What's Next (1 of 3)

- Welcome to Assembly Language
- **Virtual Machine Concept**
- Data Representation
- Boolean Operations

Virtual Machines (1 of 2)

- Tanenbaum: Virtual machine concept
- Programming Language analogy:
  - Each computer has a native machine language (language L0) that runs directly on its hardware
  - A more human-friendly language is usually constructed above machine language, called Language L1

If the language VM1 supports is still not programmer-friendly enough to be used for useful applications, then another virtual machine, VM2, can be designed that is more easily understood.



Translating Languages

English: Display the sum of A times B plus C.

C++: cout << (A * B + C);

Assembly Language:

mov eax,A

mul B
add eax,C
call WriteInt

Intel Machine Language:

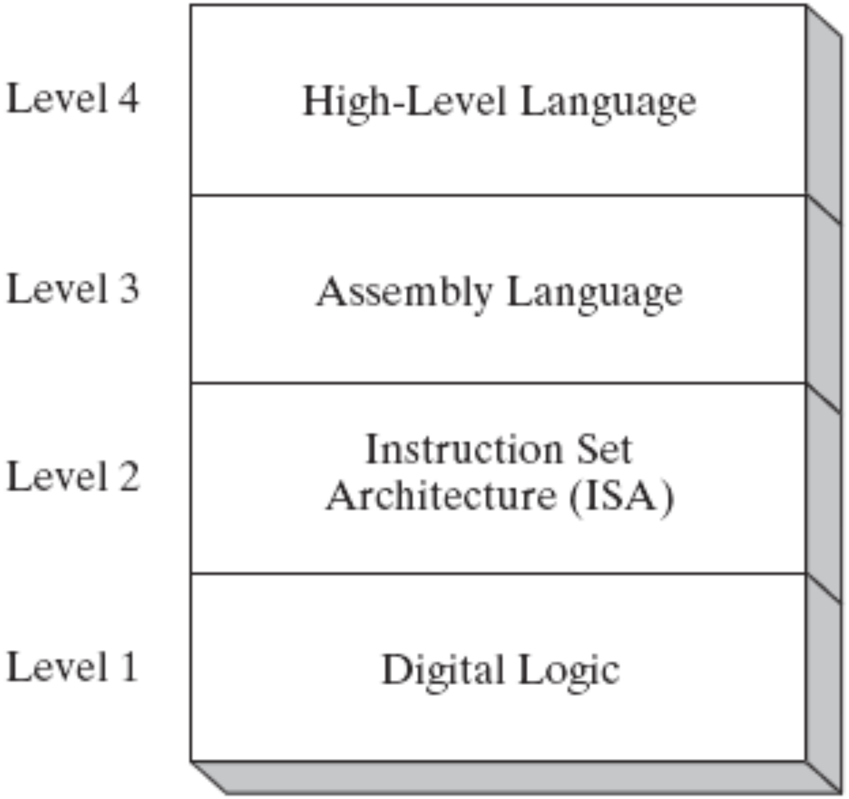
A1 00000000

F7 25 00000004

03 05 00000008

E8 00500000

Specific Machine Levels



(descriptions of individual levels follow . . .)

High-Level Language

- Level 4
- At Level 4 are high-level programming languages such as C, C++, and Java.
- Application-oriented languages
  - C++, Java, Pascal, Visual Basic . . .
- Programs compile into assembly language

(Level 4)

Assembly Language

- Level 3
- Instruction mnemonics(ADD, SUB, MOV) that have a one-to-one correspondence to machine language
- Assembly language programs are translated (assembled) in their entirety into machine language before they begin to execute.
- 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)
- This is the first level at which users can typically write programs, although the programs consist of binary values called machine language.

Digital Logic

- Level 1
- CPU, constructed from digital logic gates
- System bus
- Memory
- Implemented using bipolar transistors

What's Next (2 of 3)

- Welcome to Assembly Language
- Virtual Machine Concept
- **Data Representation**
- Boolean Operations

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 fluency with number formats, so you can quickly translate numbers from one format to another.

Data Representation

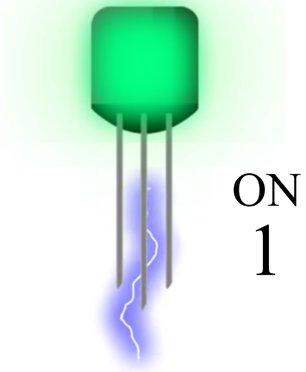
- Binary Numbers
  - Translating between binary and decimal
- Binary Addition
- Integer Storage Sizes
- Hexadecimal Integers
  - Translating between decimal and hexadecimal
  - Hexadecimal subtraction
- Signed Integers
  - Binary subtraction
- Character Storage

Binary Numbers (1 of 2)

- Digits are 1 and 0
  - 1 = true
  - 0 = false
- MSB – most significant bit
- LSB – least significant bit
- Bit numbering:

Micro Transistors:

OFF
0



ON
1

Base 10 number:

Difference is 5000000

Difference is 0000009

5064089

Most significant

Least significant

Base 2 number:

Binary Numbers (2 of 2)

- Each digit (bit) is either 1 or 0
- Each bit represents a power of 2:

128 64 32 16 8 4 2 1

Every binary number is a sum of powers of 2

Table 1-3 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 Binary to Decimal

Weighted positional notation shows how to calculate the decimal value of each binary bit:

$$dec = (\mathbf{D}_{n\text{-}1} \times \mathbf{2}^{n\text{-}1}) + (\mathbf{D}_{n\text{-}2} \times \mathbf{2}^{n\text{-}2}) + \ldots + (\mathbf{D}_1 \times \mathbf{2}^1) + (\mathbf{D}_0 \times \mathbf{2}^0)$$

D = binary digit

binary 00001001 = decimal 9:

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

0	0	0	0	1	0	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	$\boxed{8}$	4	2	$\boxed{1}$

8 + 1

9

Translating Unsigned Decimal 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

142 to binary = 10001110

142 / 2	=	71	R0
71 / 2	=	35	R1
35 / 2	=	17	R1
17 / 2	=	8	R1
8 / 2	=	4	R0
4 / 2	=	2	R0
2 / 2	=	1	R0
1 / 2	=	0	R1

Binary Addition

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

Binary addition facts:

$$
\begin{array}{r} 0 \\ +\,0 \\ \hline 0 \end{array}
\qquad
\begin{array}{r} 1 \\ +\,0 \\ \hline 1 \end{array}
\qquad
\begin{array}{r} 0 \\ +\,1 \\ \hline 1 \end{array}
\qquad
\begin{array}{r} {\scriptstyle 1} \\ 1 \\ +\,1 \\ \hline 10 \end{array}
\qquad
\begin{array}{r} {\scriptstyle 1} \\ 1 \\ 1 \\ +\,1 \\ \hline 11 \end{array}
$$

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

Integer Storage Sizes

1 byte = 8 bits

Standard sizes:

The basic storage unit for all data in an x86 computer is a byte, containing 8 bits. Other storage sizes are word (2 bytes), doubleword (4 bytes), and quadword (8 bytes). In the following figure, the number of bits is shown for each size:

Table 1-4 Ranges of Unsigned Integers.

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

Hexadecimal Integers

Binary values are represented in hexadecimal.

Table 1-5 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

Large binary numbers are cumbersome to read, so hexadecimal digits offer a convenient way to represent binary data. Each digit in a hexadecimal integer represents four binary bits, and two hexadecimal digits together represent a byte.

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

1	0	1	1	0	1	1	1
2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
8	4	2	1	8	4	2	1

8 + 2 + 1   4 + 2 + 1

11   7

$B7_{16}$

0	0	1	0	1	1	1	0	1	0	1	0
8	4	2	1	8	4	2	1	8	4	2	1

2   8 + 4 + 2   8 + 2

14   10

$2EA_{16}$

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.

3	B	A	4
3	11	10	4
16^3	16^2	16^1	16^0

$$3*16^3 + 11*16^2 + 10*16^1 + 4*16^0 = 15268$$

Converting Decimal to Hexadecimal

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

decimal 422 = 1A6 hexadecimal

479_{10} Convert to hexadecimal =

479 / 16 = 29.9375   R15

x16

29 / 16 = 1.8125   R13

x16

1 / 16 = 0.0625   R1

x16

1 13 15

$1\ D\ F_{16}$

Hexadecimal Addition

Base 10 addition:

```
  1
  45        5
+ 17       +7
----       --
  62       12
```

val ≥ 10
val % 10
12 % 10   rem 2

- Divide the sum of two digits by the number base (16). The quotient becomes the carry value, and the remainder is the sum digit.

```
                 1     1
36     28       28    6A
42     45       58    4B
78     6D       80    B5
```

21 / 16 = 1, rem 5

```
    1
    9   C   A
+   2   E   5
-------------
   12  26  15
    C   A   F
```

val ≥ 16
val % 16
26 % 16 rem 10

Important skill: Programmers frequently add and subtract the addresses of variables and instructions.

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

Hexadecimal Subtraction

Base 10 subtraction:

$$\begin{array}{r} ^{-1} \\ 45 \\ -\,17 \\ \hline 28 \end{array} \qquad \begin{array}{r} 10 + 5 \\ -7 \\ \hline 8 \end{array}$$

- When a borrow is required from the digit to the left, add 16 (decimal) to the current digit's value:

16 + 5 = 21

$$\begin{array}{r} C6 \\ -\,A2 \\ \hline 24 \end{array} \qquad \begin{array}{r} ^{-1} \\ 75 \\ -\,47 \\ \hline 2E \end{array}$$

$$\begin{array}{r} 16 + 12 = 28 \\ -14 \\ \hline 14 \end{array}$$

$$\begin{array}{rccc} & ^{-1} & & \\ & 9 & C & A \\ - & 2 & E & 5 \\ \hline & 6 & E & 5 \end{array}$$

Practice: The address of **var1** is 00400066. The address of the next variable after var1 is 0040006A. How many bytes are used by var1?

$$\begin{array}{r} 0040006A \\ -00400066 \\ \hline 4 \end{array}$$

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

Signed Integers

0 1 1 1
8 4 2 1
4+2+1
= 7

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

The highest bit indicates the sign.
1 = negative, 0 = positive

For x86 processors, the (most significant bit) MSB indicates the sign: 0 is positive and 1 is negative.

If the highest digit of a hexadecimal integer is > 7, the value is negative. Examples: 8A, C5, A2, 9D

8A = 10001010 A2 = 10100010
C5 = 11000101 9D = 10011101

Positive:
7A = 01111010
6D = 01101101

Forming the Two's Complement

- Negative numbers are stored in two's complement notation
- Represents the additive Inverse

Starting value	00000001
Step 1: reverse the bits	11111110
Step 2: add 1 to the value from Step 1	11111110 +00000001
Sum: two's complement representation	11111111

Note that 00000001 + 11111111 = 00000000

Negative integers use two's-complement representation, using the mathematical principle that the two's complement of an integer is its additive inverse. (If you add a number to its additive inverse, the sum is zero.)

Binary Subtraction

- When subtracting A – B, convert B to its two's complement. Add A to (–B). This removes the need for separate digital circuits to handle both addition and subtraction.

$$\begin{array}{r} 0\,0\,0\,0\,1\,1\,0\,0 \\ -\,0\,0\,0\,0\,0\,0\,1\,1 \\ \hline \end{array} \longrightarrow \begin{array}{r} 0\,0\,0\,0\,1\,1\,0\,0 \\ +\,1\,1\,1\,1\,1\,1\,0\,1 \\ \hline 0\,0\,0\,0\,1\,0\,0\,1 \end{array}$$

Practice: Subtract 5 - 9.

$$\begin{array}{r} 5 \\ -\,9 \\ \hline -\,4 \end{array} \qquad \begin{array}{r} \_\ 2^3\ 2^2\ 2^1\ 2^0 \\ 0\,0\,1\,0\,1 \\ -\,0\,1\,0\,0\,1 \\ \hline \\ 0\,1\,0\,0\,1 \\ \downarrow \\ 1\,0\,1\,1\,1 \end{array} \qquad \begin{array}{rl} 1\ \ 1\ \ 1\ \ \ \ & \\ 0\,0\,1\,0\,1 & \\ +\,1\,0\,1\,1\,1 & \\ \hline 1\,1\,1\,0\,0 & = -4 \\ \downarrow\ \downarrow\ \downarrow & \\ 0\,0\,1\,0\,0 & = +4 \\ \_\ 8\ 4\ 2\ 1 & \end{array}$$

Optional: The following video will help you refresh your memory on two's complement: https://youtu.be/vbHgvaPSVyc

Hexadecimal Two's Complement

- To create the two's complement of a hexadecimal integer, reverse all bits and add 1. An easy way to reverse the bits of a hexadecimal digit is to subtract the digit from 15. Here are examples of hexadecimal integers converted to their two's complements:

$9C_{16}$ positive or negative? Negative. Since the highest digit is > 7

$$\begin{array}{rr} 9 & C \\ -\,15 & 15 \\ \hline 6 & 3 \\ + & 1 \\ \hline 6 & 4 \end{array}$$

$$\begin{array}{c} 9 \quad\quad C \\ 1\;0\;0\;1\;1\;1\;0\;0 \\ \boxed{0\;1\;1\;0}\;\boxed{0\;1\;0\;0} \\ 8\;4\;2\;1\quad 8\;4\;2\;1 \\ 6 \quad\quad 4 \end{array}$$

$$\begin{array}{rrr} A & 2 & F \\ -\,15 & 15 & 15 \\ \hline 5 & 13 & 0 \\ & + & 1 \\ \hline 5 & D & 0 \end{array}$$

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

Learn How To Do the Following:

- Form the two's complement of a hexadecimal integer
- Convert signed binary to decimal
- Convert signed decimal to binary
- Convert signed decimal to hexadecimal
- Convert signed hexadecimal to decimal

Ranges of Signed Integers

The highest bit is reserved for the sign. This limits the range:

Storage Type	Range (low–high)	Powers of 2
Signed byte	–128 to +127	-2^7 to $(2^7 - 1)$
Signed word	–32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	–2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Practice: What is the largest positive value that may be stored in 20 bits?

$2^{20} - 1 = 1048575$

Character Storage

- Character sets
  - Standard ASCII (0 – 127)
  - Extended ASCII (0 – 255)
  - ANSI (0 – 255)
  - Unicode(0 – 65,535)
- Null-terminated String
  - Array of characters followed by a *null byte*
- Using the ASCII table
  - back inside cover of book

Control Characters				Graphic Symbols											
Name	Dec	Binary	Hex	Symbol	Dec	Binary	Hex	Symbol	Dec	Binary	Hex	Symbol	Dec	Binary	Hex
NUL	0	0000000	00	space	32	0100000	20	@	64	1000000	40	`	96	1100000	60
SOH	1	0000001	01	!	33	0100001	21	A	65	1000001	41	a	97	1100001	61
STX	2	0000010	02	"	34	0100010	22	B	66	1000010	42	b	98	1100010	62
ETX	3	0000011	03	#	35	0100011	23	C	67	1000011	43	c	99	1100011	63
EOT	4	0000100	04	$	36	0100100	24	D	68	1000100	44	d	100	1100100	64
ENQ	5	0000101	05	%	37	0100101	25	E	69	1000101	45	e	101	1100101	65
ACK	6	0000110	06	&	38	0100110	26	F	70	1000110	46	f	102	1100110	66
BEL	7	0000111	07	'	39	0100111	27	G	71	1000111	47	g	103	1100111	67
BS	8	0001000	08	(	40	0101000	28	H	72	1001000	48	h	104	1101000	68
HT	9	0001001	09	)	41	0101001	29	I	73	1001001	49	i	105	1101001	69
LF	10	0001010	0A	*	42	0101010	2A	J	74	1001010	4A	j	106	1101010	6A
VT	11	0001011	0B	+	43	0101011	2B	K	75	1001011	4B	k	107	1101011	6B
FF	12	0001100	0C	,	44	0101100	2C	L	76	1001100	4C	l	108	1101100	6C
CR	13	0001101	0D	−	45	0101101	2D	M	77	1001101	4D	m	109	1101101	6D
SO	14	0001110	0E	.	46	0101110	2E	N	78	1001110	4E	n	110	1101110	6E
SI	15	0001111	0F	/	47	0101111	2F	O	79	1001111	4F	o	111	1101111	6F
DLE	16	0010000	10	0	48	0110000	30	P	80	1010000	50	p	112	1110000	70
DC1	17	0010001	11	1	49	0110001	31	Q	81	1010001	51	q	113	1110001	71
DC2	18	0010010	12	2	50	0110010	32	R	82	1010010	52	r	114	1110010	72
DC3	19	0010011	13	3	51	0110011	33	S	83	1010011	53	s	115	1110011	73
DC4	20	0010100	14	4	52	0110100	34	T	84	1010100	54	t	116	1110100	74
NAK	21	0010101	15	5	53	0110101	35	U	85	1010101	55	u	117	1110101	75
SYN	22	0010110	16	6	54	0110110	36	V	86	1010110	56	v	118	1110110	76
ETB	23	0010111	17	7	55	0110111	37	W	87	1010111	57	w	119	1110111	77
CAN	24	0011000	18	8	56	0111000	38	X	88	1011000	58	x	120	1111000	78
EM	25	0011001	19	9	57	0111001	39	Y	89	1011001	59	y	121	1111001	79
SUB	26	0011010	1A	:	58	0111010	3A	Z	90	1011010	5A	z	122	1111010	7A
ESC	27	0011011	1B	;	59	0111011	3B	[	91	1011011	5B	{	123	1111011	7B
FS	28	0011100	1C	<	60	0111100	3C	\	92	1011100	5C	|	124	1111100	7C
GS	29	0011101	1D	=	61	0111101	3D	]	93	1011101	5D	}	125	1111101	7D
RS	30	0011110	1E	>	62	0111110	3E	^	94	1011110	5E	~	126	1111110	7E
US	31	0011111	1F	?	63	0111111	3F	_	95	1011111	5F	Del	127	1111111	7F

Storage range

- What is the minimum number of **bits** needed to represent 257 different characters?

$2^8 = 256$

$2^9 = 512$

Therefore, we will need at least 9 bits to represent 257 different characters.

- What is the minimum number of **bits** needed to represent the unsigned decimal integer 4095?

$2^{11} = 2048$

$2^{12} = 4096$

With 12 bits, we have the capacity to represent numbers within the range of 0 to 4095.

What's Next (3 of 3)

- Welcome to Assembly Language
- Virtual Machine Concept
- Data Representation
- **Boolean Operations**

Boolean Operations

- NOT
- AND
- OR
- Operator Precedence
- Truth Tables

Boolean Algebra

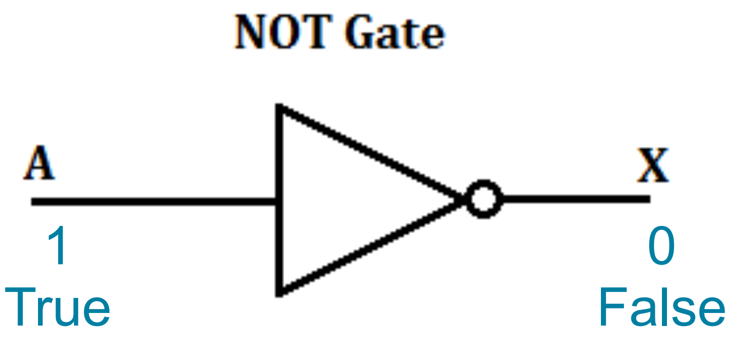
- Based on symbolic logic, designed by George Boole
- Boolean expressions created from:
  - NOT, AND, OR

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
- Truth table for Boolean NOT operator:

X	¬X
F	T
T	F



AND

- Truth table for Boolean AND operator:

X	Y	X ∧ Y
F	F	F
F	T	F
T	F	F
T	T	T

Digital gate diagram for AND:

x: 0 1 1 0 1
y: 1 0 0 1 1

x ∧ y: 0 0 0 0 1

OR

- 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:

x: 0 1 1 0 1
y: 1 0 0 1 1

x ∨ y: 1 1 1 1 1

Operator Precedence

- Examples showing the order of operations:

NOT operator has the highest precedence, followed by AND and OR.
You can use parentheses to force the initial evaluation of an expression:

Expression	Order of Operations
$\neg X \vee Y$	NOT, then OR
$\neg(X \vee Y)$	OR, then NOT
$X \vee (Y \wedge Z)$	AND, then OR

Truth Tables (1 of 3)

- A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
- A truth table shows all the inputs and outputs of a Boolean function

Example: ¬X ∨ Y

intermediate step

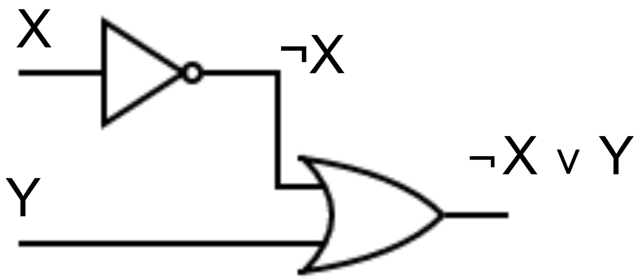
result

inputs

X	¬X	Y	¬X ∨ Y
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

=

X	Y
T	T
T	F
F	T
F	F



Truth Tables (2 of 3)

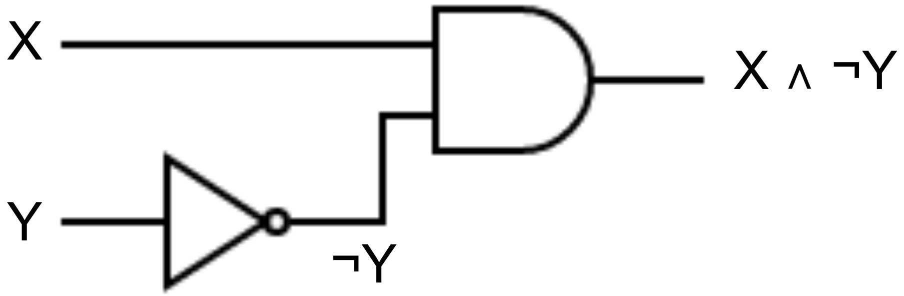
- Example: X ∧ ¬ Y

intermediate step

result

inputs

X	Y	¬Y	X ∧ ¬Y
T	T	F	F
T	F	T	F
F	T	F	F
F	F	T	F



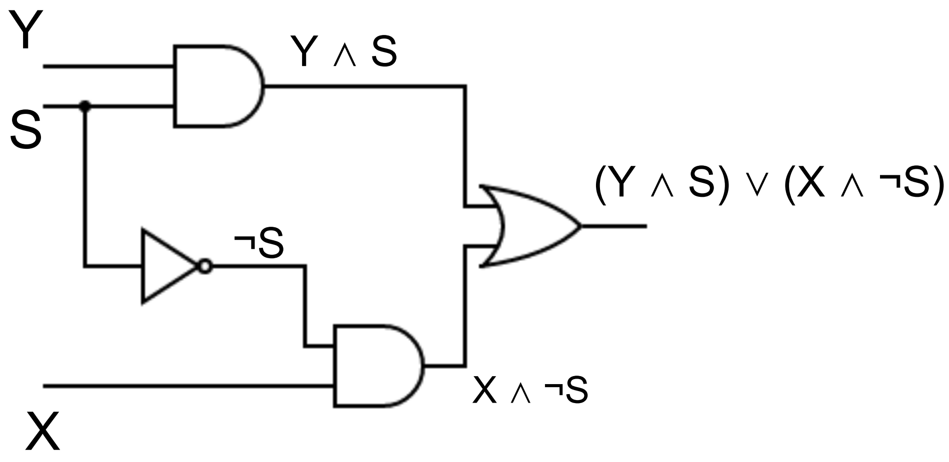
inputs

$2^{3} = 8$

Truth Tables (3 of 3)

- Example: (Y∧S) ∨ (X∧ ¬S)

X	Y	S	Y∧S	¬S	X∧¬S	(Y∧S) ∨ (X∧ ¬S)
	inputs			intermediate step		result
T	T	T	T	F	F	T
T	T	F	F	T	T	T
T	F	T	F	F	F	F
T	F	F	F	T	T	T
F	T	T	T	F	F	T
F	T	F	F	T	F	F
F	F	T	F	F	F	F
F	F	F	F	T	F	F



Summary

- Assembly language helps you learn how software is constructed at the lowest levels
- Assembly language has a one-to-one relationship with machine language
- Each layer in a computer's architecture is an abstraction of a machine
  - layers can be hardware or software
- Boolean expressions are essential to the design of computer hardware and software