CSC 3210

Computer Organization and Programming

Chapter 1 Basic Concepts

Dr. Zulkar Nine

mnine@gsu.edu

Georgia State University

Spring 2021

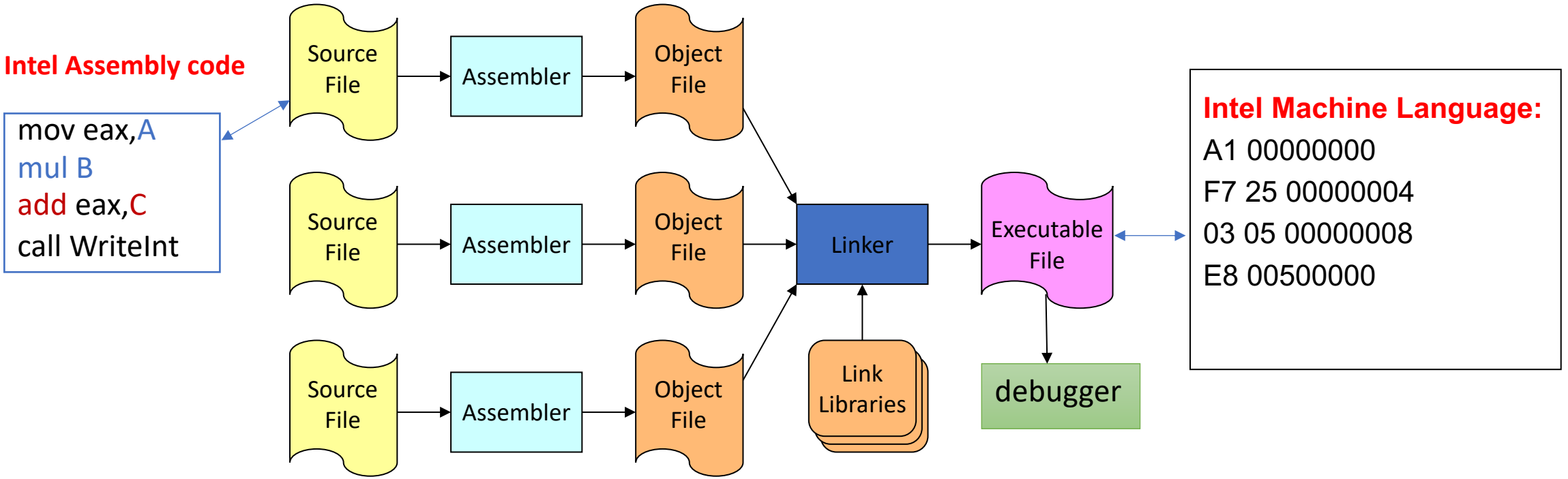
Outline

- Welcome to Assembly Language
  - Some concerns
  - Applications
- Virtual Machine Concepts
  - Virtual Machines
  - Machine Levels
  - Translating Language
- Data Representation
- Boolean Operations

Welcome to Assembly language!

Programming is fun!

What is an **assembler**, a **linker**, a debugger?



Some issues

- What **hardware/software** do I need?
  - Computer 32/64 bit
  - Windows + Microsoft Visual Studio
  - Mac + Virtualization software (Virtualbox) + Windows + Microsoft Visual Studio
- What **types of programs** will I create?
  - Mainly **32-Bit,** Some 64 bit
- What do I get with this book?
  - http://www.asmirvine.com/

More issues

- How does **assembly language** (AL) **relate** to **machine language**?
  - One-to-one relationship
- How do **C/C++** relates to **AL**?

C/C++

```
int Y;
int X = (Y + 4) * 3;
```

Assembly

```
mov eax,Y ;     - move Y to the EAX register
add eax,4 ;     - add 4 to the EAX register
mov ebx,3 ;     - move 3 to the EBX register
imul ebx ;      - multiply EAX by EBX
mov X,eax ;     - move EAX to X
```

- Is **AL** portable? No. Why?

Assembly Language Applications

- Some representative types of applications:
  - **Hardware device driver**
  - **Embedded systems** & **computer games**
  - Business application for single platform (No)
  - Business application for multiple platforms (No)

Think again!

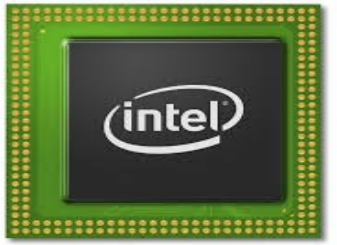
Is the **assembly language for x86** processors **the same as those** for computer systems such as the **Vax, Motorola 68x00, or SPARC**?

SPARC

```
mov 5,r1
mov 6,r2
add r1,r2,r3
```

Intel

```
mov eax,5
add eax,6
```



What's Next

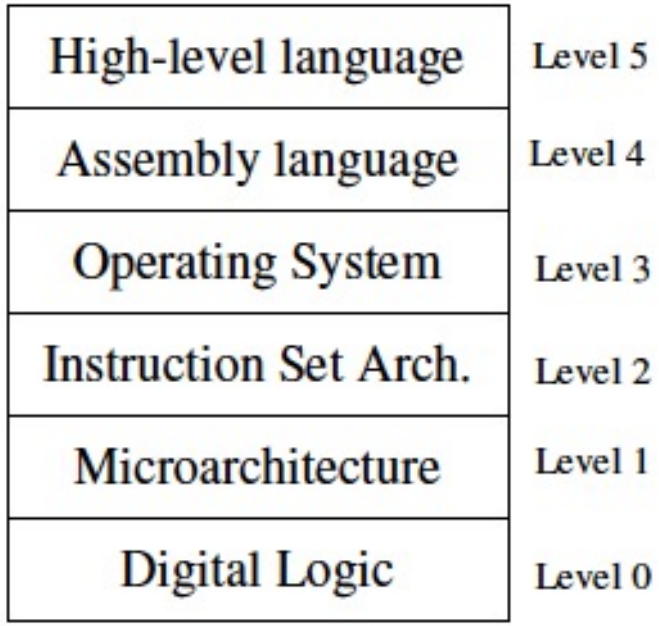
- - Welcome to Assembly Language
  - Some Good Questions to Ask
  - Assembly Language Applications
- **- Virtual Machine Concept**
  - Virtual Machines
  - Specific Machine Levels
  - Translating Languages
- - Data Representation
- - Boolean Operations

Virtual Machine Concepts

Abstraction that hides all the low level complexities

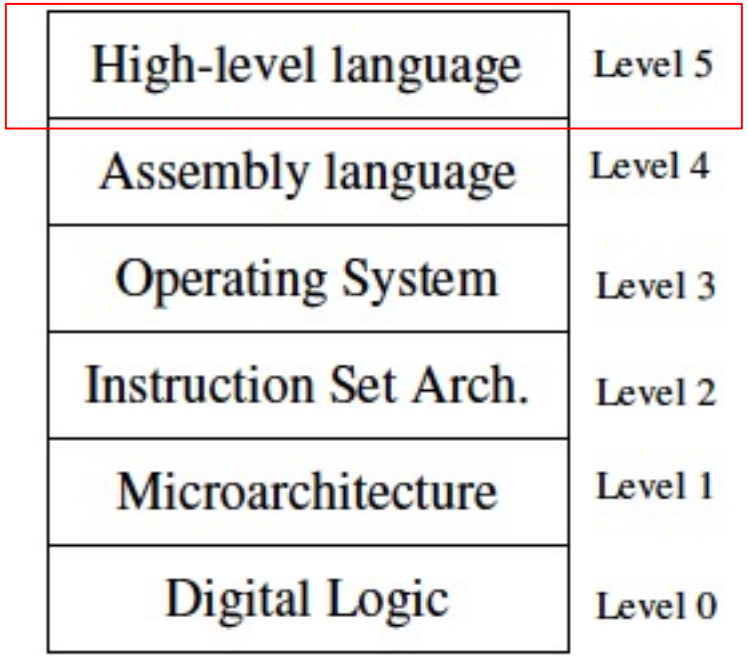
Specific Machine Levels

- **Virtual Machine Concept**
  - Explain how a computer's **hardware** and **software** are related.
  - The main thing a computer does is **executing programs**
  - A computer execute programs written in its **native machine language (ML).**
    - difficult to write programs in ML
      - ➢ What to do?
    - programs could be written in another language (Interpretation vs. Translation)



Specific Machine Levels: High-Level Language

- **Level 5**
- Application-oriented languages
  - **C++,** Java, Visual Basic . . .
- Programs **compile into assembly language** (Level 4)



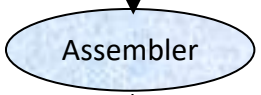
Specific Machine Levels: High-Level Language

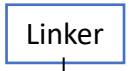
High-level language program (in C)

```
void main()
        {
        C = ( Y + 4 ) * 3 ;
        }
```

Assembly language program

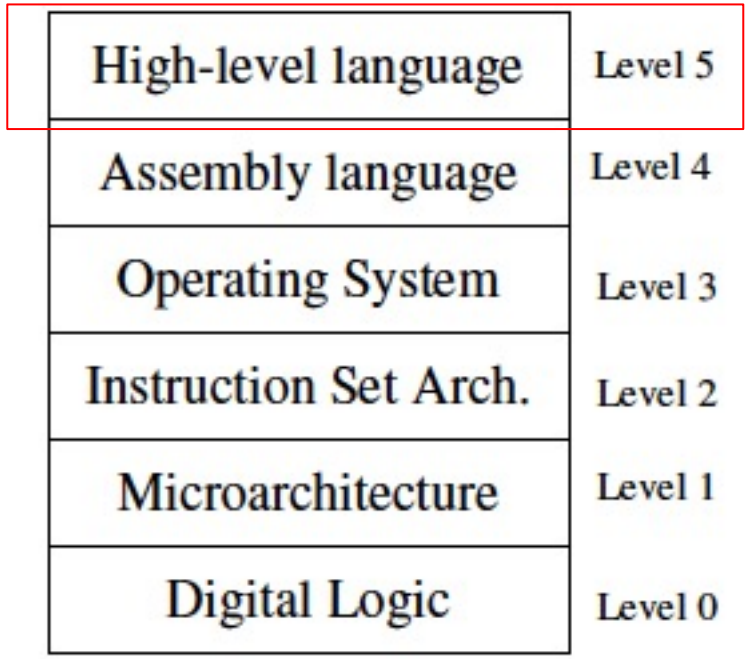
```
mov eax, Y
add eax, 4
mov ebx, 3
imul ebx
mov C, eax
```





Binary machine language program

```
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
00000011111000000000000000001000
```



What's Next

- - Welcome to Assembly Language
  - Some Good Questions to Ask
  - Assembly Language Applications
- **- Virtual Machine Concept**
  - Virtual Machines
  - Specific Machine Levels
  - Translating Languages
- **- Data Representation**
- - Boolean Operations

Data Representation

Lots of math!

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**

Data Representation: Numbering System

- Assembly language deals with **Data at the physical level**
  - so you need to examine **registers** and **memory**
- **Binary** and **hexadecimal** numbers are commonly used to describe those contents (other systems used as well)
- **Need to learn how to translate from one format to another**

TABLE 1-2 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

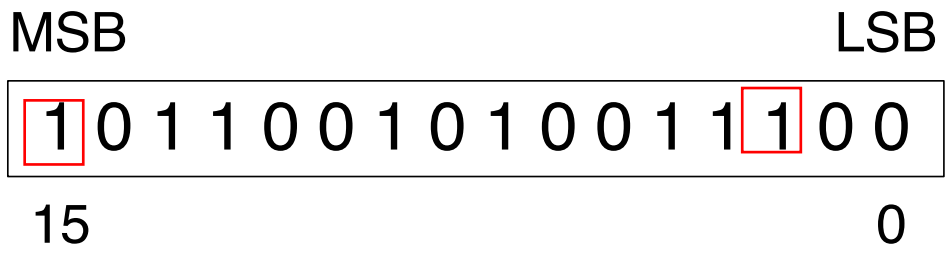
Base: maximum number of **symbols** assigned to every digit

Data Representation: Binary Numbers **(Integers)**

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

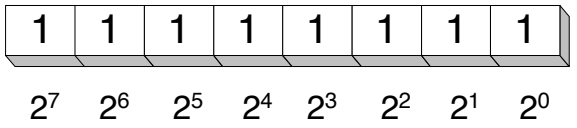
Data Representation: Binary Numbers **(Integers)**

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



Data Representation: Binary Numbers **(Integers)**

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



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

Data Representation: Translating **Binary to Decimal**

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

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

D = binary digit

binary 00001001 = decimal 9:

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

Data Representation: Translating **Binary to Decimal**

Data Representation: Translating **Unsigned Decimal to Binary**

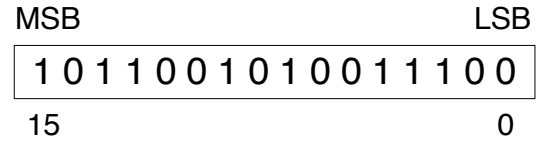
- Repeatedly divide the decimal integer by 2.
- Each remainder is a binary digit in the translated value:

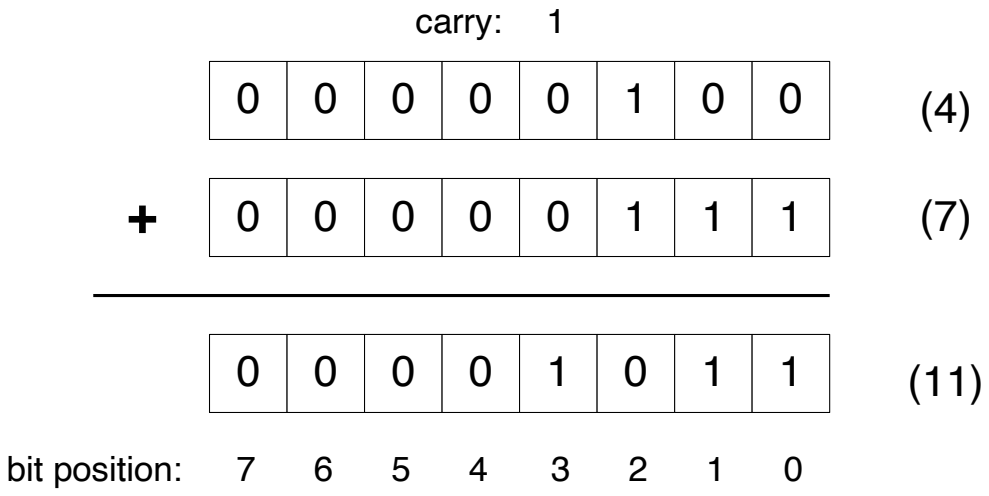
37 = 100101

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

Data Representation: Binary **Addition**

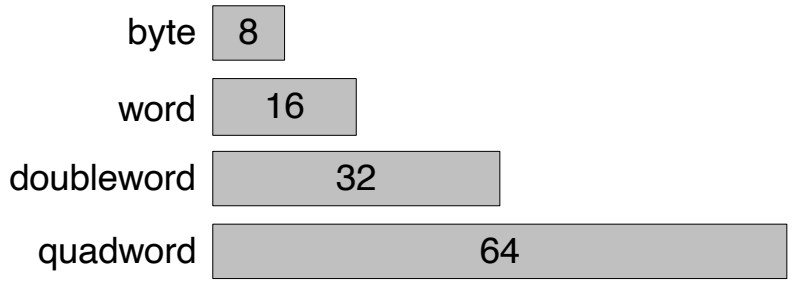
- Starting with the **LSB**, add each pair of digits, **include the carry if present**.





Data Representation: Integer **Storage Sizes**

- Standard sizes (**x86**):



What is the largest unsigned integer that may be stored in 20 bits?

Table 1-4 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

Data Representation: **Hexadecimal** Integers

- Binary values are represented in hexadecimal (Why).
  - **Large binary numbers are hard to read**

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

Data Representation: Translating **Binary** to **Hexadecimal**

- Each hexadecimal digit corresponds to **4 binary bits** (Why?).
- **Example**: Translate the binary integer

  000101101010011110010100

  to hexadecimal:

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

Data Representation: 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 $(3 \times 16^3) + (11 * 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal 15,268.

Data Representation: Converting **Hexadecimal to Decimal**

-

Data Representation: Converting **Decimal to Hexadecimal**

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

decimal 422 = 1A6 hexadecimal

Data Representation: **Hexadecimal** Addition

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

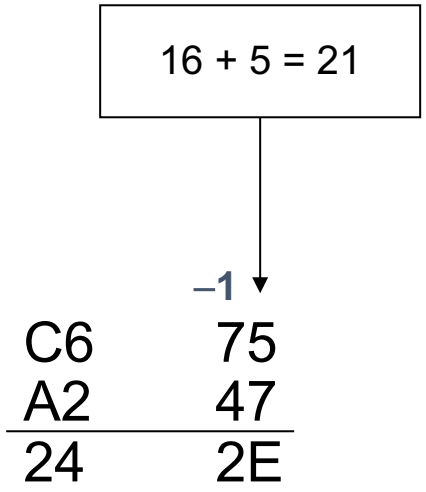
		1	1
36	28	28	6A
42	45	58	4B
78	6D	80	B5

21 / 16 = 1, rem 5

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

Data Representation: **Hexadecimal** **Subtraction**

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

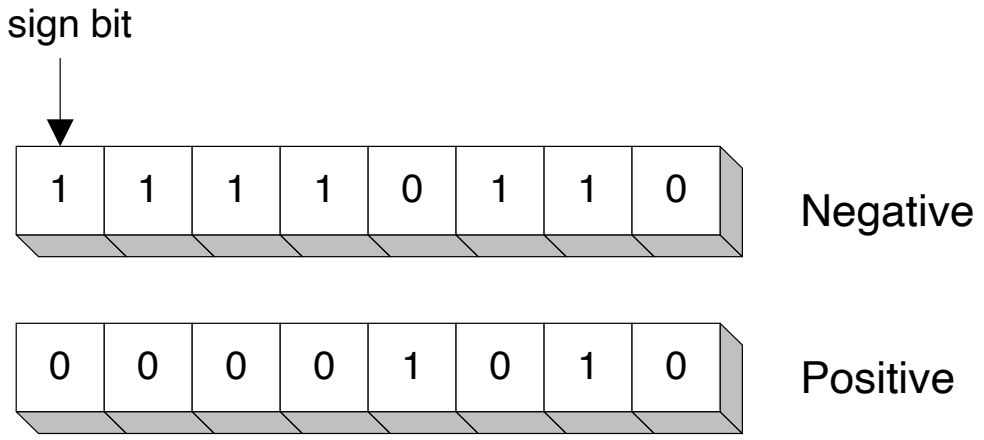


Data Representation: **Signed** Integers

- Challenges -
  - Dealing with both positive and negative numbers
  - Simplify the Circuit design

Data Representation: **Signed** Integers

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



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

Data Representation: Forming the **Two's Complement**

- Negative integer numbers are stored in **two's complement notation, (Why?)**
  - Removes the need to **separate digital circuits** for **addition** and **subtraction**
- When subtracting A – B, processor will **convert it to an addition** expression

A + (-B)

Data Representation: Forming the **Two's Complement**

- Negative integer 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

Algorithm

- **First:** represent the negative number in binary
- **Second:** do One's complement:

  Replace every 0 with a 1,

  and replace every 1 with a 0
- **Third**: do Two's complement: add 1
- **Fourth:** represent the second number in binary
- **Fifth:** add the two numbers

Data Representation: Binary **Subtraction**

- When subtracting A – B, convert B to its two's complement

A + (-B)

A: 0 0 0 0 1 1 0 0

B: (-) 0 0 0 0 0 0 1 1

(where is the carry?)

2's complement of B:

A: 0 0 0 0 1 1 0 0

(+) 1 1 1 1 1 1 0 1

0 0 0 0 1 0 0 1

Data Representation: Binary **Subtraction**

- What if the result is a negative number?

Consider 2 – 4:

$$
\begin{aligned}
2 &= 0010 \\
4 &= 0100 \\
\text{one's complement } 1\text{'s} &= 1011 \\
\text{two's complement } 2\text{'s} &= 1100
\end{aligned}
$$

$$
\begin{array}{r}
0010 \\
1100 \quad + \\
\hline
1110
\end{array}
$$

1110 must be a negative number, so let us complement it:

- convert the negative number to its positive equivalent

$$
\begin{array}{lll}
 & 1110 & \\
1\text{'s} & 0001 & \\
2\text{'s} & 0010 & -2
\end{array}
$$

Data Representation: Learn How To Do the Following:

- Form the two's complement of a hexadecimal integer (page 16)
- Convert signed binary to decimal (page 17)
- Convert signed decimal to binary (page 17)
- Convert signed decimal to hexadecimal (page 17)
- Convert signed hexadecimal to decimal (page 17)

You must know these conversions!

Data Representation: 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?

Data Representation: **Signed** Integers

- List all the **Four-bit** two's complement numbers:

- Total numbers can be stored : $2^4 - 1 = 15$

Binary	Unsigned value	Signed value
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Decimal	Bit Pattern
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Data Representation: Character Storage

- Computers only store binary data, **how do they represent characters?**
  - **Character sets (a mapping of characters to integers)**
    - 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**
  - ASCII (American Standard Code for Information Interchange) **maps each character to a specific number (decimal, hexadecimal, and bit pattern**)

ASCII

 ASCII | Unicode Character | Engine Codes | ASCII How To

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

Data Representation: Numeric Data Representation

- **Pure binary**
  - can be calculated directly
- **ASCII binary**
  - string of digits: **"0 1 1 0 0 0 0 1"** , character "a"
- **ASCII decimal**
  - string of digits: "97" a
- **ASCII hexadecimal**
  - string of digits: "61" a

What's Next

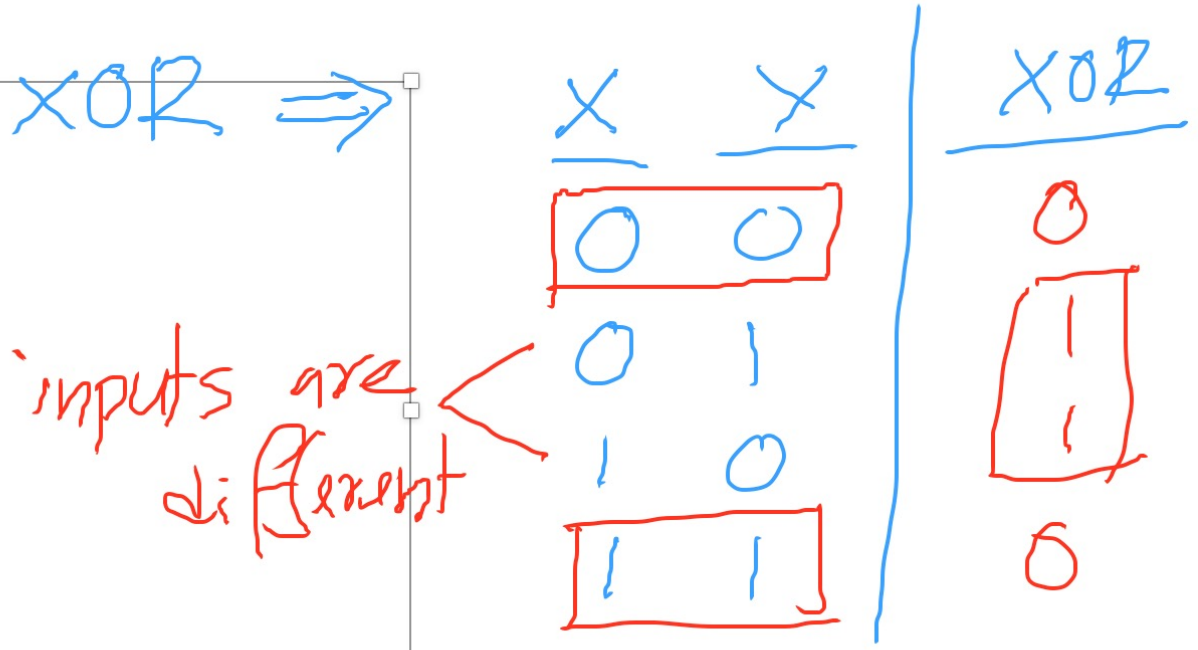
- **- Welcome to Assembly Language**
  - Some Good Questions to Ask
  - Assembly Language Applications
- **- Virtual Machine Concept**
  - Virtual Machines
  - Specific Machine Levels
  - Translating Languages
- **- Data Representation**
- **- Boolean Operations**

Boolean Operations

The world of logic

Boolean Operations

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



Boolean Operations: Boolean Algebra

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

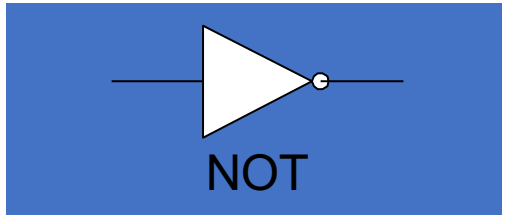
Expression	Description
¬X	NOT X
X ∧ Y	X AND Y
X ∨ Y	X OR Y
¬X ∨ Y	(NOT X) OR Y
¬(X ∧ Y)	NOT (X AND Y)
X ∧ ¬Y	X AND (NOT Y)

Boolean Operations: NOT

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

X	¬X
F	T
T	F

Digital gate diagram for NOT:



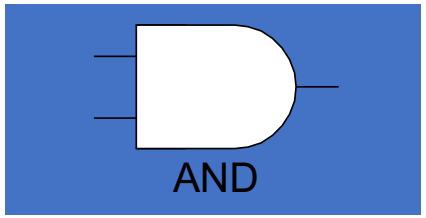
- The **NOT** operator is a **UNARY** operator (i.e. it acts on one variable or expressions)

Boolean Operations: 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:



- **AND** operator is a **BINARY** operator (i.e. it acts on two variables or expressions – "binary" in this context does not mean "base-2")

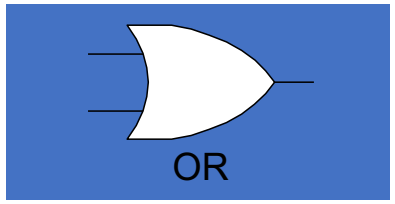
Attendance!

Boolean Operations: 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:



- The **OR** operator is a **BINARY** operator (i.e. it acts on two variables or expressions)

Boolean Operations: Operator Precedence

- Examples showing the **order of operations**:

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

Boolean Operations: Truth Tables

- 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

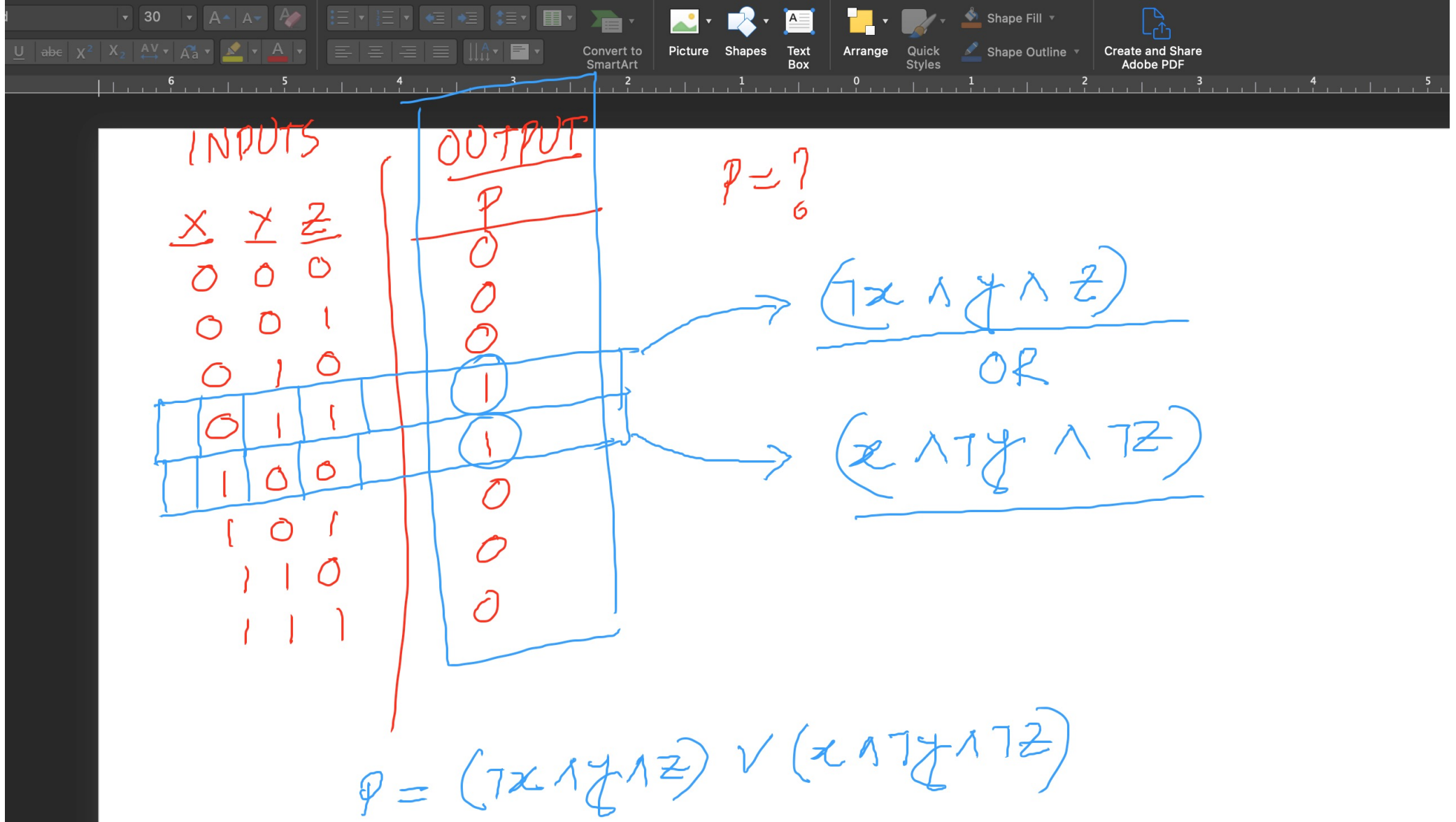
Example: $\neg X \vee Y$

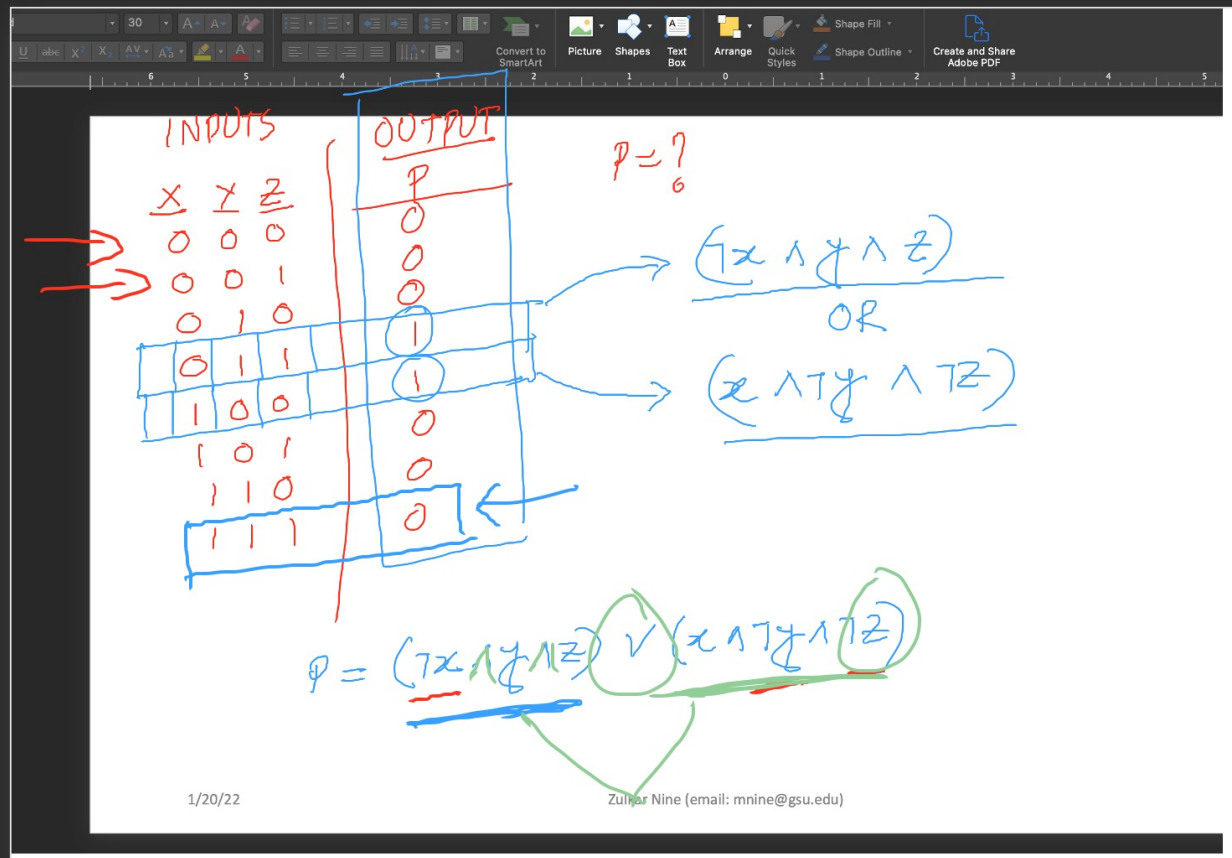
X	¬X	Y	¬X ∨ Y
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

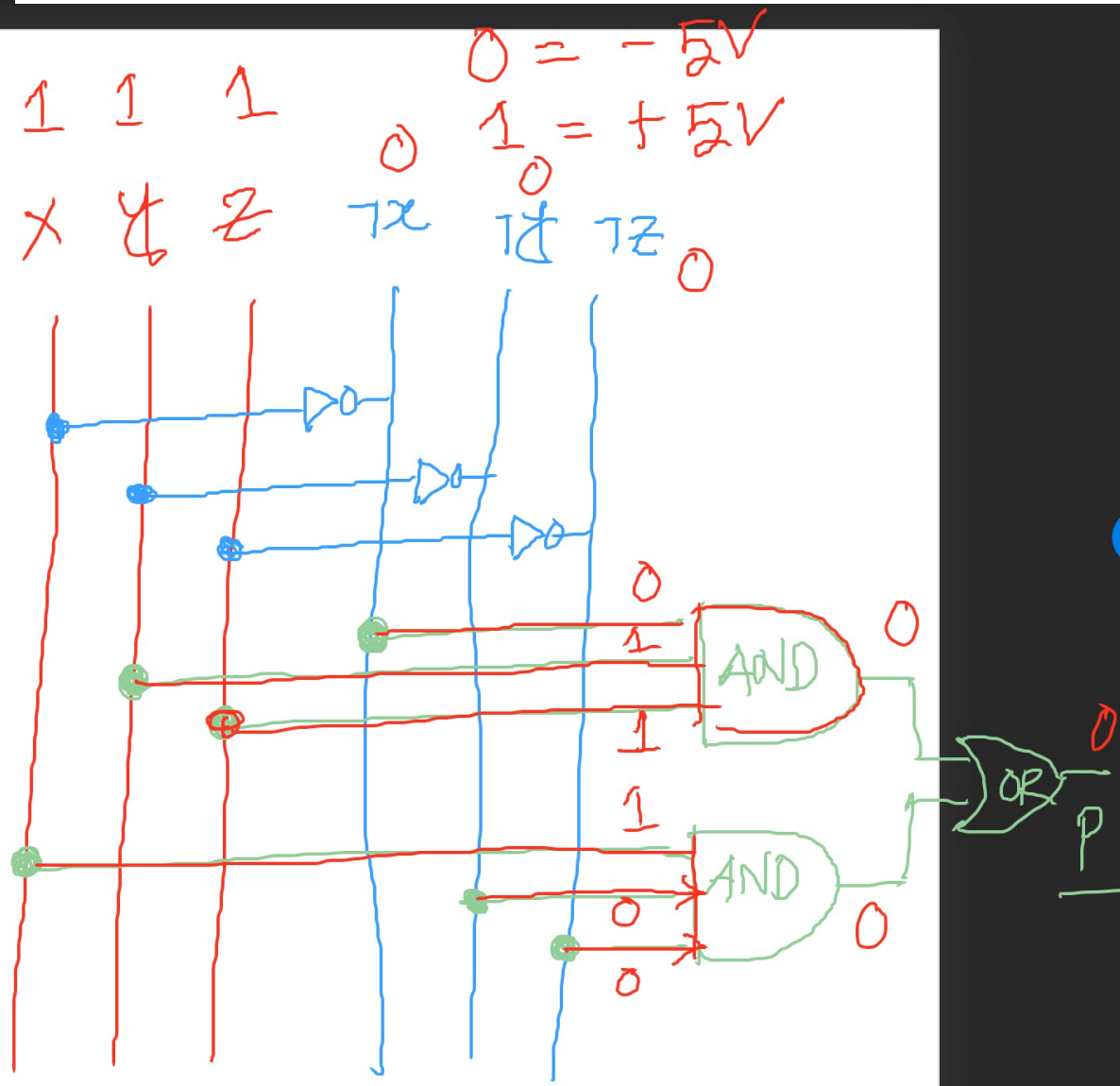
Boolean Operations: Truth Tables

- Example: Z = ?

X	Y	¬Y	Z
F	F	T	F
F	T	F	F
T	F	T	T
T	T	F	F


INPUTS
OUTPUT
X Y Z
P
P = 1
6
OR
P = (¬x ∧ ¬y ∧ z) ∨ (x ∧ ¬y ∧ ¬z)

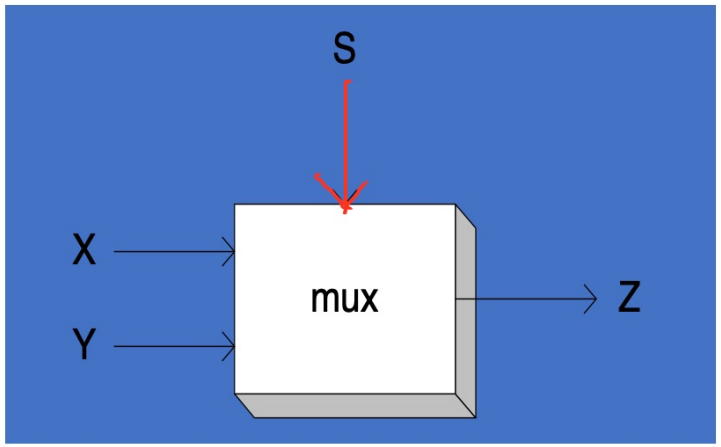

INPUTS
OUTPUT
P = ?
x y z
P
(x ∧ y ∧ z)
OR
(x ∧ ¬y ∧ ¬z)
P = (¬x ∧ y ∧ z) ∨ (x ∧ ¬y ∧ ¬z)
1/20/22
Zulkar Nine (email: mnine@gsu.edu)


0 = −5V
1 = +5V
AND
AND
OR
P

Boolean Operations: Multiplexer

Multiplexer:

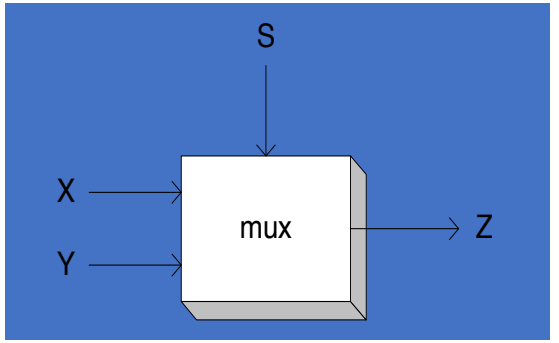
- A **data selector**. Multiplexers can also be used to implement Boolean functions of multiple variables.
- Multiple-input, single-output
- A multiplexer (or **mux**) is a common digital circuit
- Two-input multiplexer

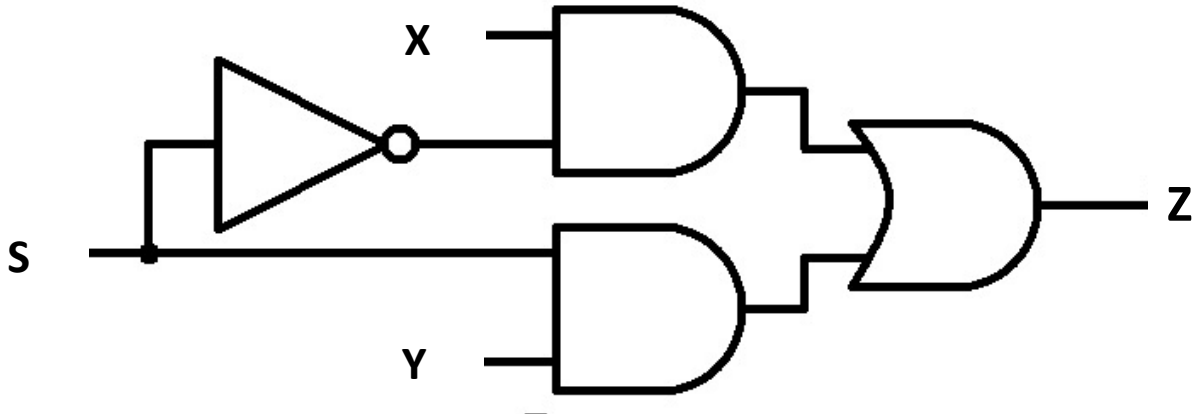


Two-input multiplexer

Boolean Operations: Truth Tables (**Multiplexer**)

S	X	Y	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



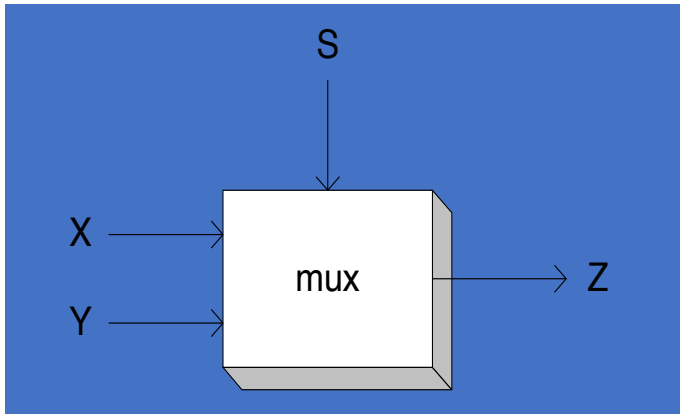


Z = (X and $\overline{S}$) OR (Y and S)

Boolean Operations: Truth Tables (**Multiplexers**)

- Example: $(Y \wedge S) \vee (X \wedge \neg S)$

X	Y	S	$Y \wedge S$	$\neg S$	$X \wedge \neg S$	$(Y \wedge S) \vee (X \wedge \neg S)$
F	F	F	F	T	F	F
F	T	F	F	T	F	F
T	F	F	F	T	T	T
T	T	F	F	T	T	T
F	F	T	F	F	F	F
F	T	T	T	F	F	T
T	F	T	F	F	F	F
T	T	T	T	F	F	T

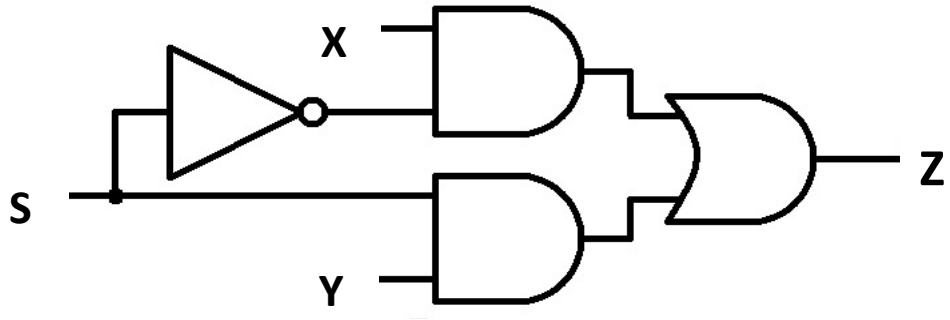


Two-input multiplexer

Boolean Operations: Truth Tables (**Multiplexer**)

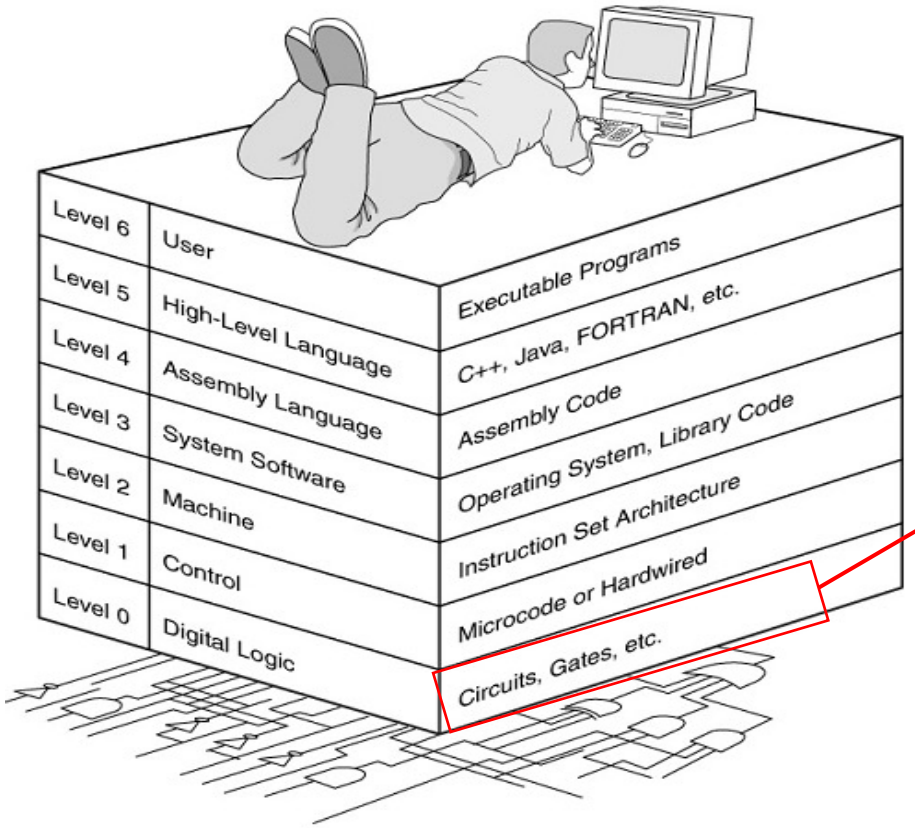
- For combinational logic, **if/else** is implemented as a 2:1 multiplexer.
- If (S)
  Z= X;
  else
  Z= Y;
- In Boolean algebra, this would be:

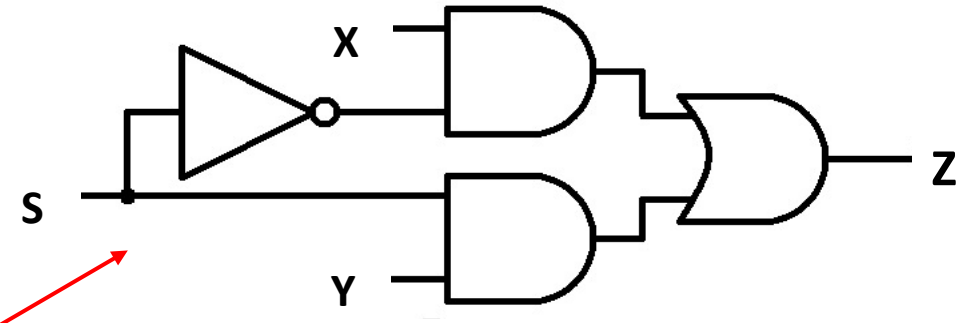
  Z = (X and $\overline{S}$) OR (Y and S) where:
  - **S** is the input fed by the **if** condition,
  - **X** is the input fed by the **then** subexpression,
  - **Y** is the input fed by the **else** subexpression, and
  - **Z** is the output of the **expression**.



Z = (X and $\overline{S}$) OR (Y and S)

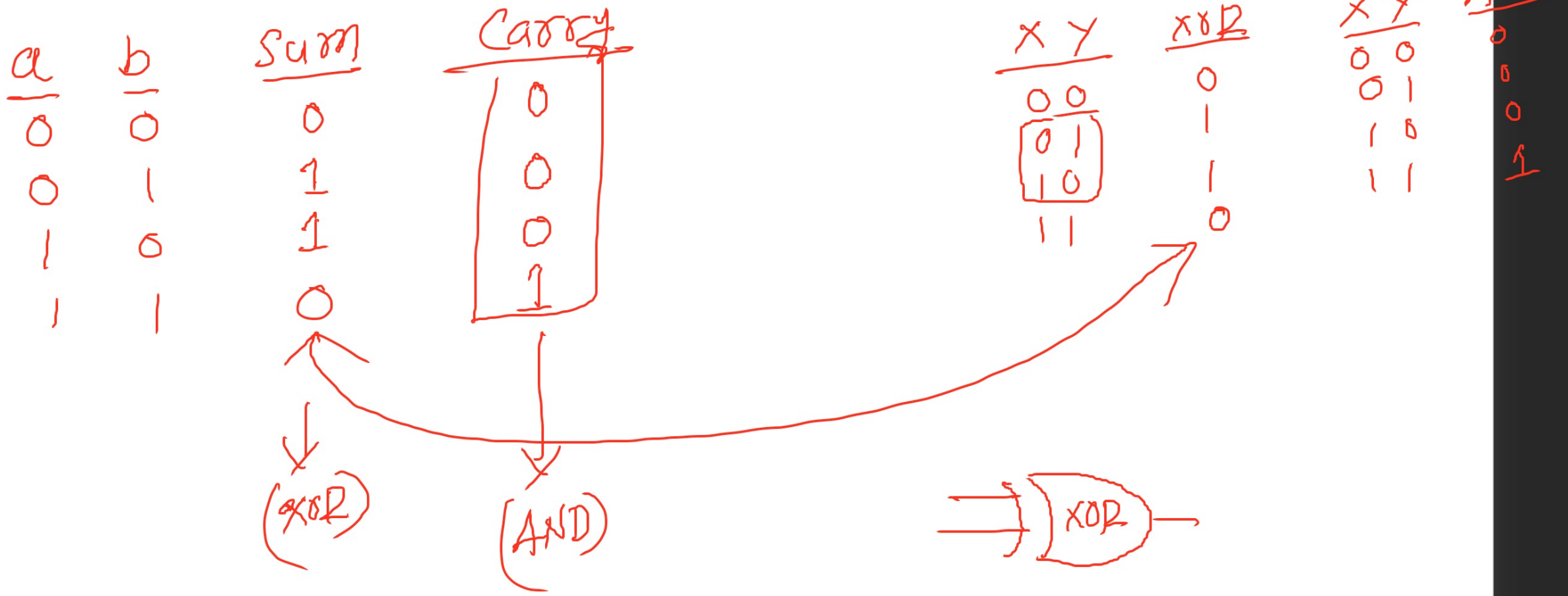
Boolean Operations: Truth Tables (**Multiplexer**)



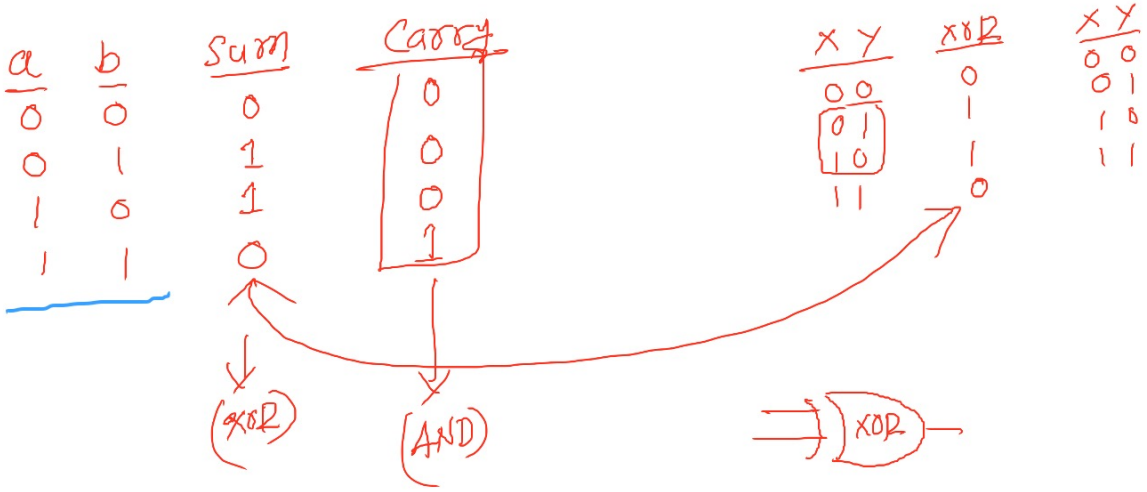


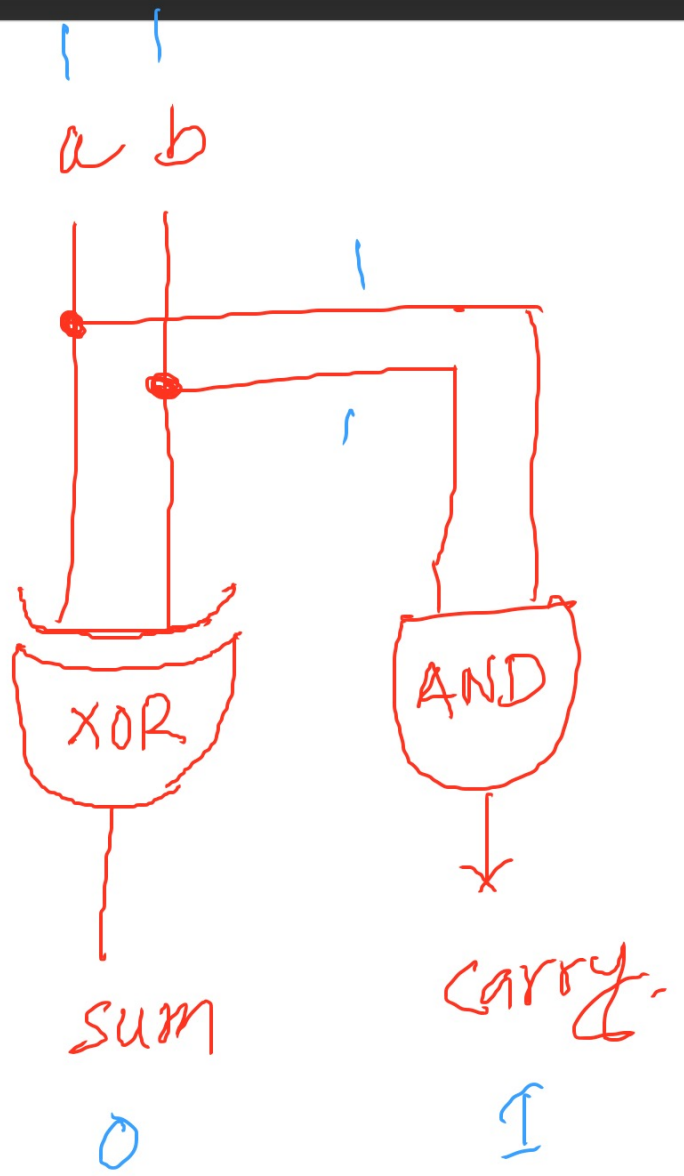
Z = (X and $\overline{S}$) OR (Y and S)

Boolean Operations: Half and Full Adders



Boolean Operations: Half and Full Adders





Boolean Operations: Half and Full Adders

- **Half Adder:**
  - The generation of the sum and carry bits may be performed by very simple **electronic circuits**.
  - The generation of the sum and carry requires an and gate and an xor gate.
  - These may be obtained together in a circuit called a half adder

a	b	Sum
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive or

a	b	Carry
0	0	0
0	1	0
1	0	0
1	1	1

and

a	b	sum	carry
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

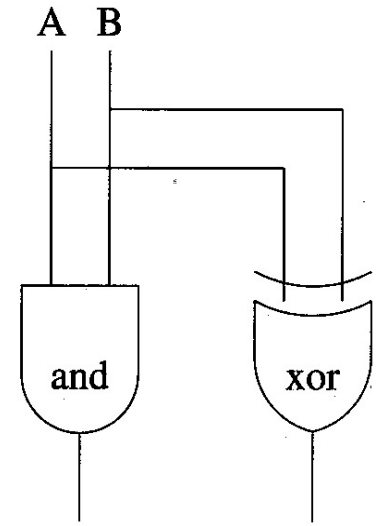


Figure 4.1: Half Adder

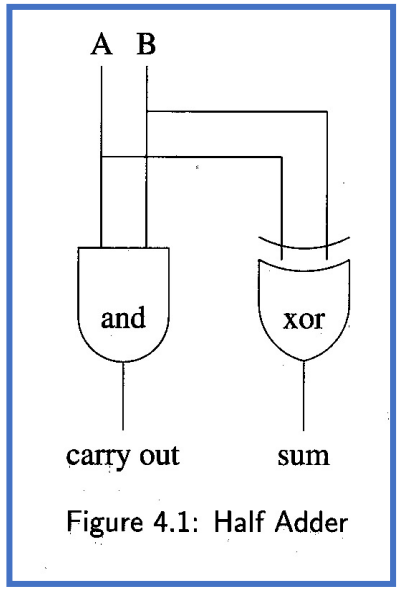


Figure 4.1: Half Adder

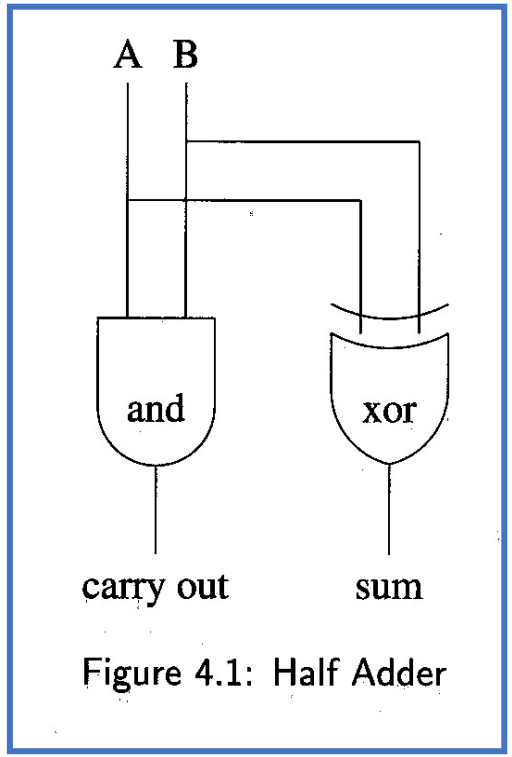


Figure 4.1: Half Adder

Boolean Operations: Half and Full Adders

- **Full Adder:**
  - **Two half adders may then be combined** together with an or gate to add two inputs, A and B, with a carry in, to produce a sum and a carry out.
  - Such a **circuit** is called a full adder

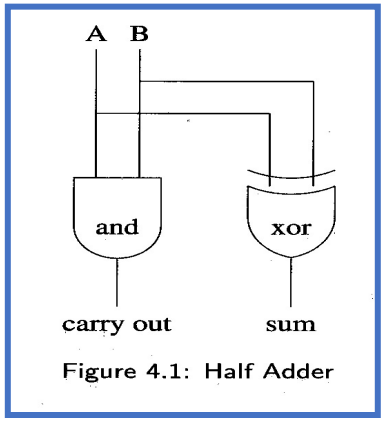


Figure 4.1: Half Adder

1 + 1 + 1 = 1 and carry 1

Half adder cannot handle this addition?

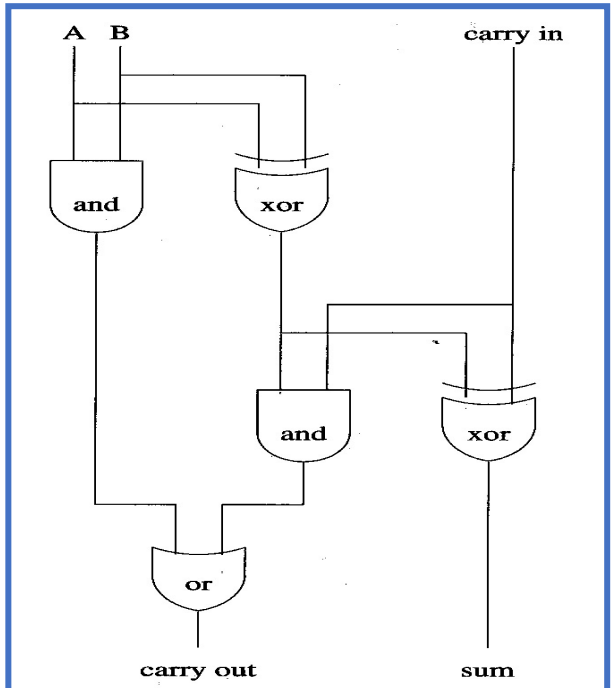


Figure 4.2: Full Adder

Boolean Operations: Half and Full Adders

- **Full Adder:**

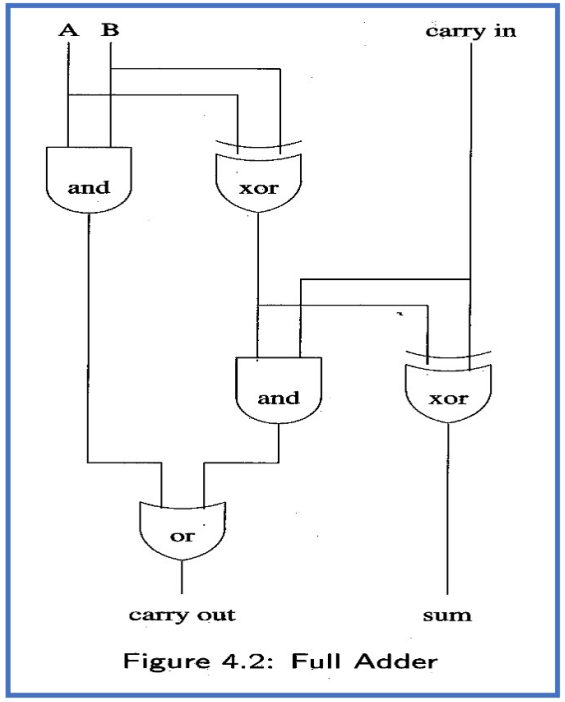
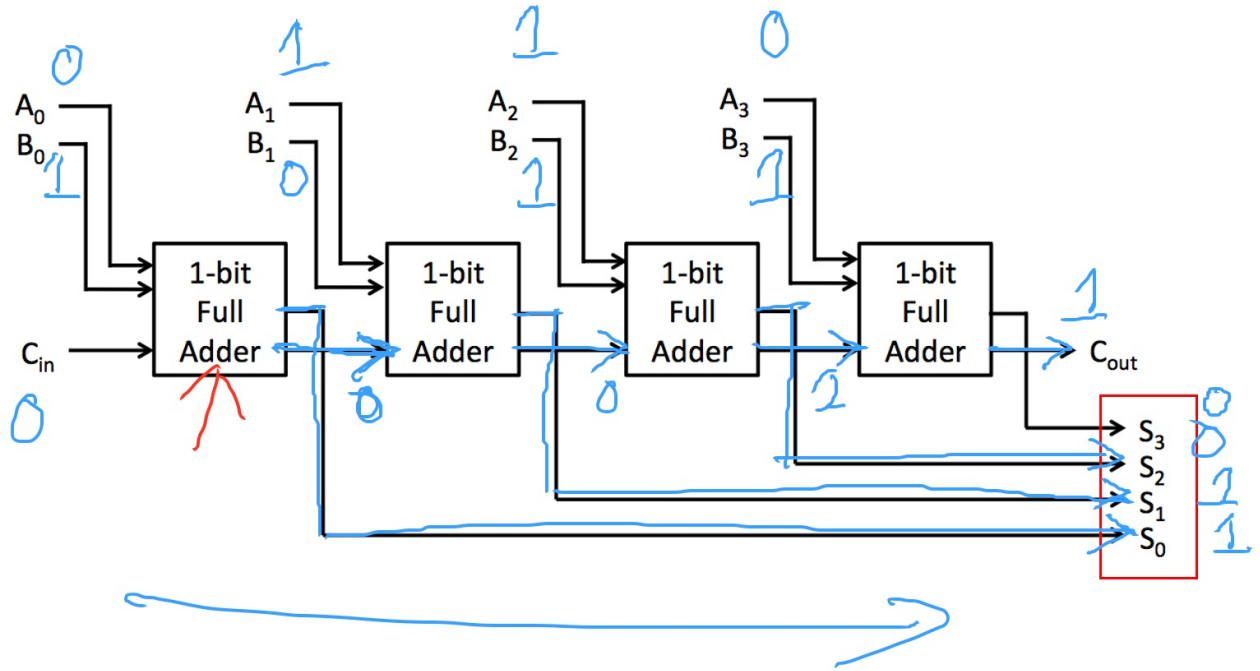


Figure 4.2: Full Adder



Summary

- Assembly language helps you learn **how software is constructed at the lowest levels**
- Assembly language has **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**