History of Computing

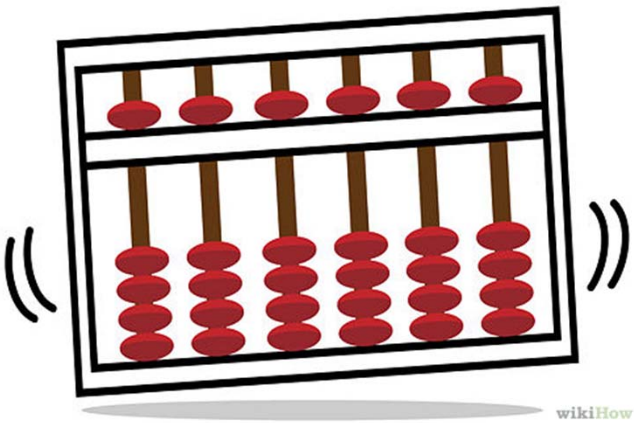
Ahmed Sallam

Outline

- Blast from the past
- Layered Perspective of Computing
- Why Assembly?
- Data Representation
  - Base 2, 8, 10, 16 Number systems
- Boolean operations and algebra

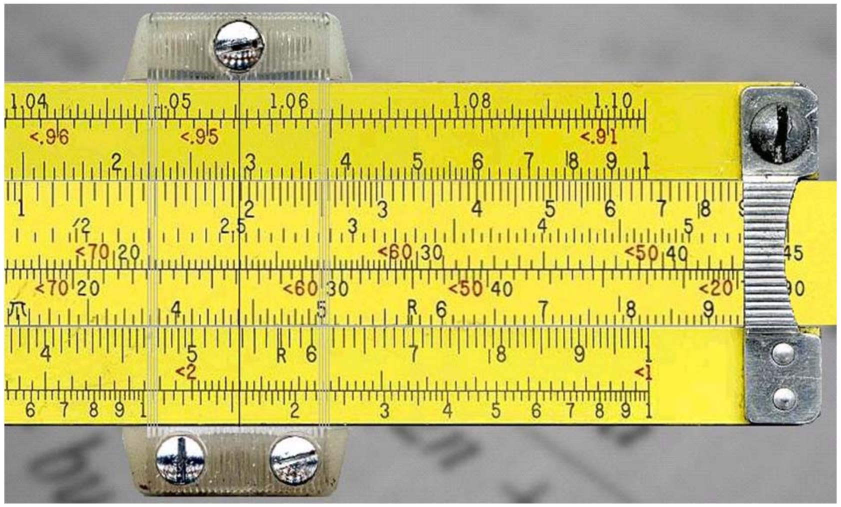
Blast from the past

- Once upon a time



Abacus

Slide rule



Blast from the past cont.1

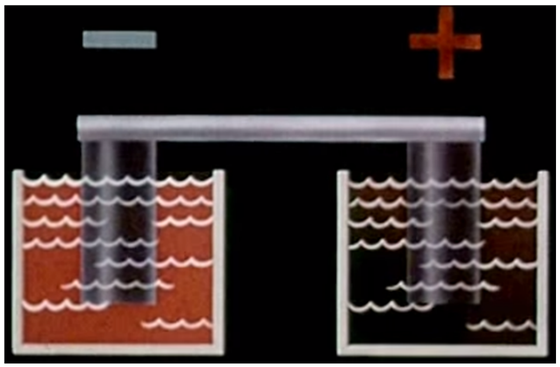
- 17th Century (Gears/Machines)

Pascaline

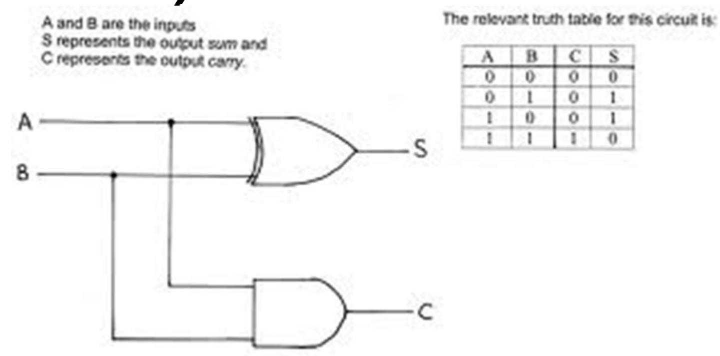
Curta (1948)

Blast from the past cont.2

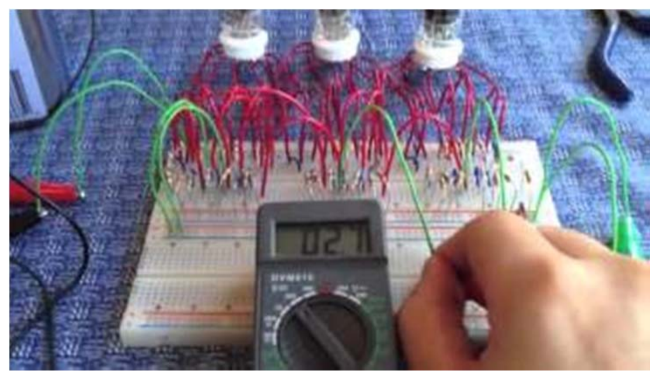
- 20th Century (Electronic)



Vacuum Tube

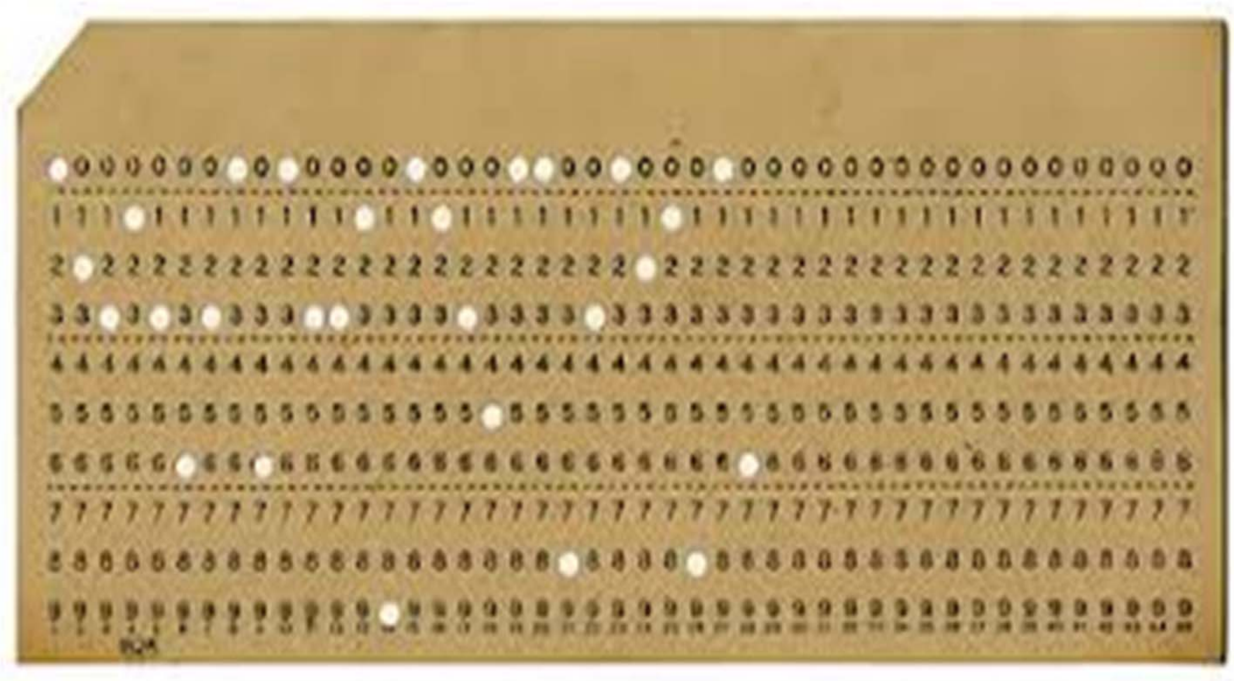


Half Adder



Blast from the past cont.3

- Memory ?!!



Punched Card

Blast from the past

- Everything is there now, let’s start to code ?!!!

Intel Machine Language

A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000

=

Assembly Language

```
mov eax, A
mul B
add eax, C
call WriteInt
```

C++ language

```
cout<<(A*B+C)
```

Outline

- *Blast from the past*
- Why Assembly?
- Layered Perspective of Computing
- Data Representation
  - Base 2, 8, 10, 16 Number systems
- Boolean operations and algebra

Outline

- *Blast from the past*
- Layered Perspective of Computing
- Why Assembly?
- Data Representation
  - Base 2, 8, 10, 16 Number systems
- Boolean operations and algebra

Layered Architecture

- Computers are complicated
  - Layers → abstraction (Hiding the complexity of layers below)
- We also layer programming languages!
- Program execution:
  - Interpretation
  - Compilation (Translation)
  - Every CPU has a built-in interpreter for its own "instruction set" (ISA, Instruction Set Architecture; the binary language it is programmed in)

Machine Levels

Level 4	**High Level Language**
Level 3	**Assembly Language**
Level 2	**Instruction Set Architecture (ISA)**
Level 1	**Digital Logic**

C++ Concepts

Visual Studio

- Programmer (with an editor)
- Produces a <u>C Program</u>

Microsoft C Compiler

- C Compiler (translator)
- Produces assembly language

- Microsoft Assembler "MASM" (translator)
- Produces Intel Binary code (object file)

x86

- Intel x86 CPU (e.g., Intel Core i5)
- Executes (interprets) Intel Binary Instructions

Java – Different Concepts

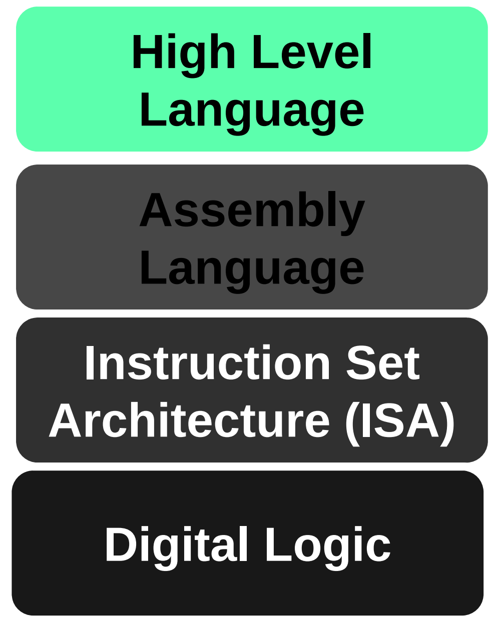
JEdit
- Programmer
- Produces a <u>Java Program</u>

Javacc
- Java Compiler (translator)
- Produces <u>Java Byte Code</u> (class file)

Java
- JVM (Java Virtual Machine – Interpreter)
- Runs the byte code to produce output

The Key Concepts

1. A High-Level Language (C, C++, Fortran, Cobol) is compiled (translated) into Assembly Language
2. The Assembly Language (for a specific CPU) is assembled into binary machine language
3. The binary machine language is interpreted by one of the CPUs in the computer
4. The CPU (Intel, AMD, etc.) uses digital logic circuits to do the interpretation and generate the results



Linking and Loading

- Assembling (running MASM) does not actually create a program that can be executed …

- There are (at least) 4 basic steps that need to be performed:
  - Assembling – translate code into binary
  - Linking – join all the parts together and resolve names
  - Loading – move the program into memory
  - Execution – run the program

Outline

- *Blast from the past*
- *Layered Perspective of Computing*
- Why Assembly?
- Data Representation
  - Base 2, 8, 10, 16 Number systems
- Boolean operations and algebra

Assembly Language

- Designed for a specific family of CPUs (i.e., Intel x86)
- Consists of a mnemonic (simplified command word) followed by the needed data
  - Example: mov eax, A
  - Move into register eax the contents of the location called A
- Generally each mnemonic (instruction) is equivalent to a single binary CPU instruction

CPU Instruction Set

- Appendix B: (Intel IA-32) we will not cover all
- Varies for each CPU
- Intel machines use an approach known as CISC
  - CISC = Complex Instruction Set Computing
  - Lots of powerful and complex (but slow) instructions
- Opposite is RISC (Reduced) with only a few very simple instructions that run fast

Why Assembly

- Communicate with hardware (drivers, embedded systems)
- Games, Graphics
- Some thing High level programming can't do (context switch)
- Better understanding of programming (reverse engineering)

Outline

- *Blast from the past*
- *Layered Perspective of Computing*
- *Why Assembly?*
- Data Representation
  - Base 2, 8, 10, 16 Number systems
- Boolean operations and algebra

Data Representation

- Computers work with binary data (sometimes represented in octal – base 8, or hexadecimal – base 16)
- You should know how to translate between these formats – THERE ARE NO CALCULATORS ON AN EXAM!
- I expect you to be able to do simple operations in these bases (you can mostly ignore octal)

Binary Numbers (Base 2)

- Digits are 1 and 0
  - 1 = true, current flowing/a charge present
  - 0 = false, no current flowing/no charge present
- MSB – most significant bit
- LSB – least significant bit
- *Bits numbered from LSB to MSB, starting from 0*

MSB	LSB
1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 0	
15	0

Binary → Decimal

1	0	1	1	0	0	1	0
2^7=128	2^6=64	2^5=32	2^4=16	2^3=8	2^2=4	2^1 = 2	2^0=1

- Simple! Don't memorize formulas from book (makes it harder)
- Learn the powers of 2:
  - 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096,…
- Then, just add up the appropriate powers
  - 10110010 = 128 + 32 + 16 + 2 = 178
- Real programmers use a calculator! We'll just have simple values in exams so you don't need a calculator and practice the basics

Decimal → 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

- Same as normal addition, from right to left
  - 0 + 0 = 0
  - 0 + 1 = 1, 1 + 0 = 1
  - 1 + 1 = 0 with a carry of 1

carry:						1			
	0	0	0	0	0	1	0	0	(4)
+	0	0	0	0	0	1	1	1	(7)
	0	0	0	0	1	0	1	1	(11)
bit position:	7	6	5	4	3	2	1	0	

Hexadecimal Numbers (Base 16)

- Binary values are represented in hexadecimal
- Not that hard: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

YOU WILL NEED THIS! Programmers work frequently in Hex

Binary	Decimal	Hex	Binary	Decimal	Hex
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

Binary → Hexadecimal

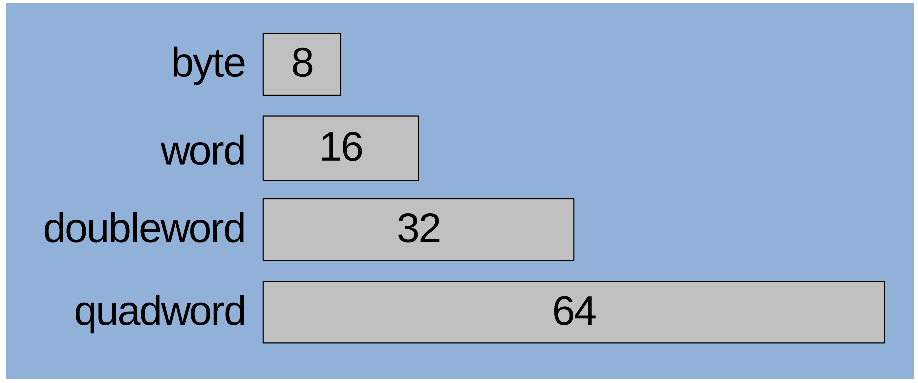
- Each hexadecimal digit corresponds to 4 binary bits.
  - Example: 000101101010011110010100
  - Group binary into groups of 4 digits (starting from the RIGHT)
  - Translate the binary into decimal by adding the powers of 1,2,4, and 8
  - E.g., 0100 = 4, 1001 = 8 + 1 = 9, 0110 = 4 + 2 + 1 = 7, 1010 = 8 + 2 = 10, 0110 = 4 + 2 = 6, 0001 = 1
  - Translate the decimal into hex: 1 6 10 7 9 4 = 16A794

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

Hexadecimal → Decimal

- *Need to know the powers of 16: 1,16,256, 4096, …*
- TOO HARD! Just use a calculator for this!
- WHAT IS IMPORTANT is to know that, FROM the RIGHT, the digits represent: 160 , 161, 162, …
- ALSO REMEMBER: x0 = 1 for all x
- The rightmost digit in a binary, octal, decimal, or hexadecimal number is the base to the power of 0

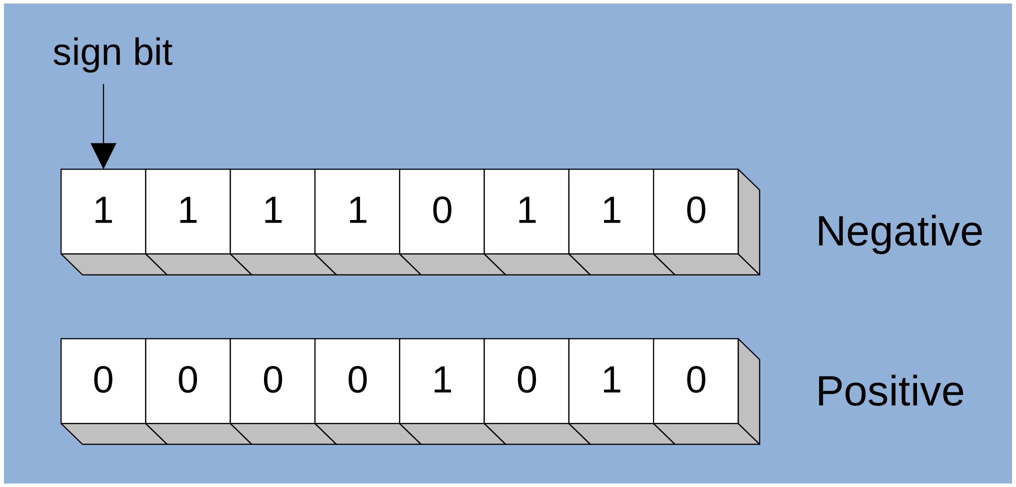
Integer Storage Sizes (Types)



- *Byte = 8 Bits*
- *Word = 2 Bytes*
- *Doubleword = 2 Words = 4 Bytes*
- *Quadword = 4 Words = 8 Bytes = 64 Bits = Max value for a 64 bit CPU*

Storage Type	Max Value	Power of 2
Unsigned byte	**255**	**2^8-1**
Unsigned word	**65,535**	**2^{16}-1**
Unsigned doubleword	**4,294,967,295**	**?**

Singed 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

Two's Complement

- *Negative numbers are stored in two's complement notation*

- *Represents the **additive Inverse***
  - **If you add the number to its additive inverse, the sum is zero.**

Starting value:	00000001
Step1: reverse the bits	11111110
Step 2: add 1 to value from step 1	11111110 + 00000001
Sum: two's complement representation	11111111

Note that 00000001 + 11111111 = 00000000

- *Hexadecimal examples:*
  - **6A3D → 95C2 + 0001 → 95C3**
  - **21F0 → DE0F + 0001 → DE10**

Singed Binary ←→Decimal

- *If the highest bit is a 0, convert it directly as unsigned binary*
- *If the highest bit is 1, the number is stored in two's complement, form its two's complement a second time to get its positive equivalent:*

Starting value:	11110000
Step1: reverse the bits	00001111
Step 2: add 1 to value from step 1	00010000
Convert to decimal and add (-) sign	-16

- *Converting signed decimal to binary:*
  1. Convert the absolute value into binary
  2. If the original decimal is negative, form the two's complement

Max & Min Values

Storage Type	Range(Min-Max)	Power of 2
Unsigned byte	0 to 255	0 to (2^8-1)
Singed byte	-128 to +127	-2^7 to (2^7-1)
Unsigned word	0 to 65,535	0 to $(2^{16}-1)$
Signed word	-32,768 to +32,767	-2^{15} to $(2^{15}-1)$

Character Storage

- *Character sets (Variations of the same thing)*
  - 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
  - Null means zero/0

Using the ASCII Table

- *Back inside cover of book (Need to know this)*
- *To find hexadecimal code of a character:*
  - ASCII Code of a is 61 hexadecimal
- *Character codes 0 to 31 → ASCII control characters*

Code (Decimal)	Description
8	Backspace
9	Horizontal tab
10	Line feed (move to next line)
13	Carriage return (leftmost output column)
27	Escape

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Endianism

- *Intel CPUs are "Little Endian"*
- *For Words, Doublewords, and Quadwords (i.e., types with more than one byte), Least Significant Bytes Come First*
- *Quadword (8 Bytes):*

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0

Memory

Address	Byte
x	B0
x+1	B1
x+2	B2
x+3	B3
x+4	B4
x+5	B5
x+6	B6
x+7	B7

Outline

- *Blast from the past*
- *Layered Perspective of Computing*
- *Why Assembly?*
- *Data Representation*
  - *Base 2, 8, 10, 16 Number systems*
- Boolean operations and algebra

Digital Logic

- CPUs are constructed from digital logic gates such as NAND, OR, XOR, etc.
- Implemented using transistors and various families of silicon devices
- Super complicated – Many millions of transistors on a single CPU

Logic is the

fundamental language of computing

Boolean Algebra

- *The fundamental model by which digital circuits are designed and, as a consequence, in which CPUs operate*
- *Basic assembly language instructions thus perform Boolean operations (so we need to know them)*
- *Based on **symbolic logic**, designed by George Boole*
  - Boolean expressions created from: NOT, AND, OR

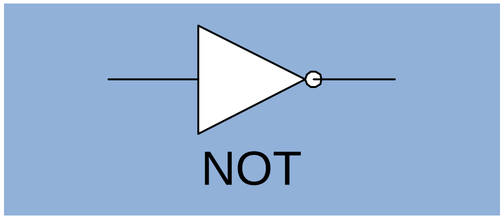
Expression	Description
$\neg X$	NOT X
$X \wedge Y$	X AND Y
$X \vee Y$	X OR Y
$\neg X \vee Y$	(NOT X) OR Y
$\neg(X \wedge Y)$	NOT (X AND Y)
$X \wedge \neg Y$	X AND (NOT Y)

NOT

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

X	¬X
F	T
T	F

Digital gate diagram for NOT:

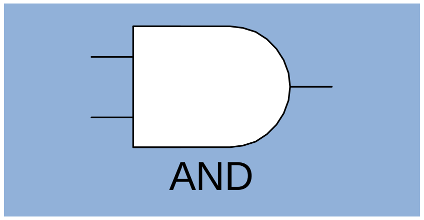


AND

- *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:

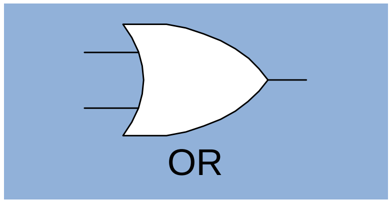


OR

- *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:



Operator Precedence

1. *Parentheses*
2. *NOT*
3. *AND*
4. *OR*

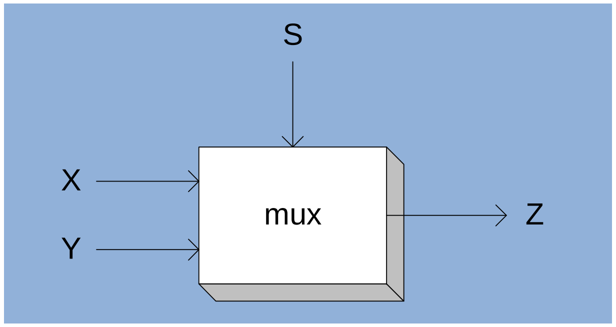
Expression	Order of Operations
$\neg X \vee Y$	NOT, then OR
$\neg(X \vee Y)$	OR, then NOT
$X \vee (Y \wedge Z)$	AND, then OR

Truth Tables

- *You won't formally have to create these, but you should remember how to trace out a complex logical operation*
- *Highly complex logical expressions are often a sign of poor program structure and design!*
  - Example: (Y ^ S) ∨ (X ^ ¬S)

X	Y	S	**Y ∧ S**	**¬S**	**X ∧ ¬S**	**(Y ∧ S) ∨ (X ∧ ¬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



Thoughts…

- *Assembly language is how software is constructed at the lowest levels*
- *Assembly language has a one-to-one relationship with binary machine language*
- *Many programmers never see more than a HLL (e.g., C++) inside and IDE (e.g., Visual Studio) but really, there is a LOT more going on*

And…

- *Nobody uses octal anymore*
- *Hex is nothing more than a useful way to manipulate binary*
- *CPUs do 3 things – Assembly programming is just using these concepts to do larger and more complicated tasks*
  - Add (basic integer math)
  - Compare (Boolean algebra)
  - Move things around