Course Syllabus [1/2]

- Instructor
  - 逄愛君, acpang@csie.ntu.edu.tw
  - Office Number: 417, Office Hour: 15:00~17:00 (Thursday)
- Textbook
  - "Assembly Language for Intel-Based Computers," Kip R. Irvine, Pearson Education, 4th Edition, 2002.
- Requirements
  - Homework x 3 45%
  - Mid-term exam 20%
  - Final exam 20%
  - Term project 15%
- TAs (Office Hour: 13:00~15:00, Wednesday)
  - 林俊仁, jrlin@voip.csie.ntu.edu.tw, Office Number: 438
  - 黃文彬, jason@voip.csie.ntu.edu.tw, Office Number: 438
- Website & BBS
  - http://www.csie.ntu.edu.tw/~acpang
  - ptt.csie.ntu.edu.tw

Course Syllabus [2/2]

- Basic Concepts
- IA-32 Processor Architecture
- Assembly Language Fundamentals
- Data Transfers, Addressing and Arithmetic
- Procedures
- Conditional Processing
- Integer Arithmetic
- Advanced Procedure (Stack, Recursion, …)
- Strings and Arrays
- Structures and Macros
- 32-Bit Window Programming
- High-Level Language Interface
- 16-Bit MS-DOS Programming (MS-DOS Function Calls)
- BIOS-Level Programming

Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine

Chapter 1: Basic Concepts

Chapter Overview

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

Welcome to Assembly Language

- Assembly language is the oldest programming language.
- Of all languages, it bears the closest resemblance to the native language of a computer.
  - Direct access to a computer's hardware
  - To understand a great deal about your computer's architecture and operating system

Some Good Questions to Ask [1/4]

- What background should I have?
  - Computer programming (C++, C#, JAVA, VB…)
- What is an assembler?
  - A program that converts source-code programs from assembly language into machine language
  - MASM (Microsoft Assembler), TASM (Borland Turbo Assembler)
  - Linker (a companion program of Assembler) combines individual files created by an assembler into a single executable program.
  - Debugger provides a way for a programmer to trace the execution of a program and examine the contents of memory.

Some Good Questions to Ask [2/4]

- What hardware/software do I need?
  - A computer with an Intel386, Intel486 or one of the Pentium processors (IA-32 processor family)
  - OS: Microsoft Windows, MS-DOS, LINUX running a DOS emulator
  - Editor, Assembler, Linker (Microsoft 16-bit linker: LINK.EXE, 32-bit linker: LINK32.EXE), Debugger (16-bit MS-DOS programs: MASM CodeView, TASM Turbo Debugger. 32-bit Windows console programs: Microsoft Visual Studio – msdev.exe)
- What types of programs will I create?
  - 16-Bit Real-Address Mode: MS-DOS, DOS emulator
  - 32-Bit Protected Mode: Microsoft Windows
- How does assembly language (AL) relate to machine language?
  - One-to-one relationship

Some Good Questions to Ask [3/4]

- What will I learn?
  - Basic principles of computer architecture
  - Basic Boolean logic
  - How IA-32 processors manage memory, using real mode, protected mode and virtual mode
  - How high-level language compilers (such as C++) translate statements into assembly language and native machine code
  - Improvement of the machine-level debugging skills (e.g., errors due to memory allocation)
  - How application programs communicate with the computer's operating system via interrupt handlers, system calls, and common memory areas
- How do C++ and Java relate to AL? E.g., X=(Y+4) *3
- Is AL portable?
  - A language whose source program can be compiled and run on a wide variety of computer systems is said to be portable.
  - AL makes no attempt to be portable.
    - It is tied to a specific processor family.

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

Some Good Questions to Ask [4/4]

- Why learn AL?
  - Embedded system programs
  - Programs to be highly optimized for both space and runtime speed
  - To gain an overall understanding of the interaction between the hardware, OS and application programs
  - Device driver: programs that translate general operating system commands into specific references to hardware details
- Are there any rules in AL?
  - Yes, there are a few rules, mainly due to the physical limitations of the processor and its native instruction set

Assembly Language Applications

- It is rare to see large application programs written completely in assembly language because they would take too much time to write and maintain.
- AL is used to optimize certain sections of application programs for speed and to access computer hardware.
- 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

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.

Virtual Machine Concept

- Virtual Machines
- Specific Machine Levels

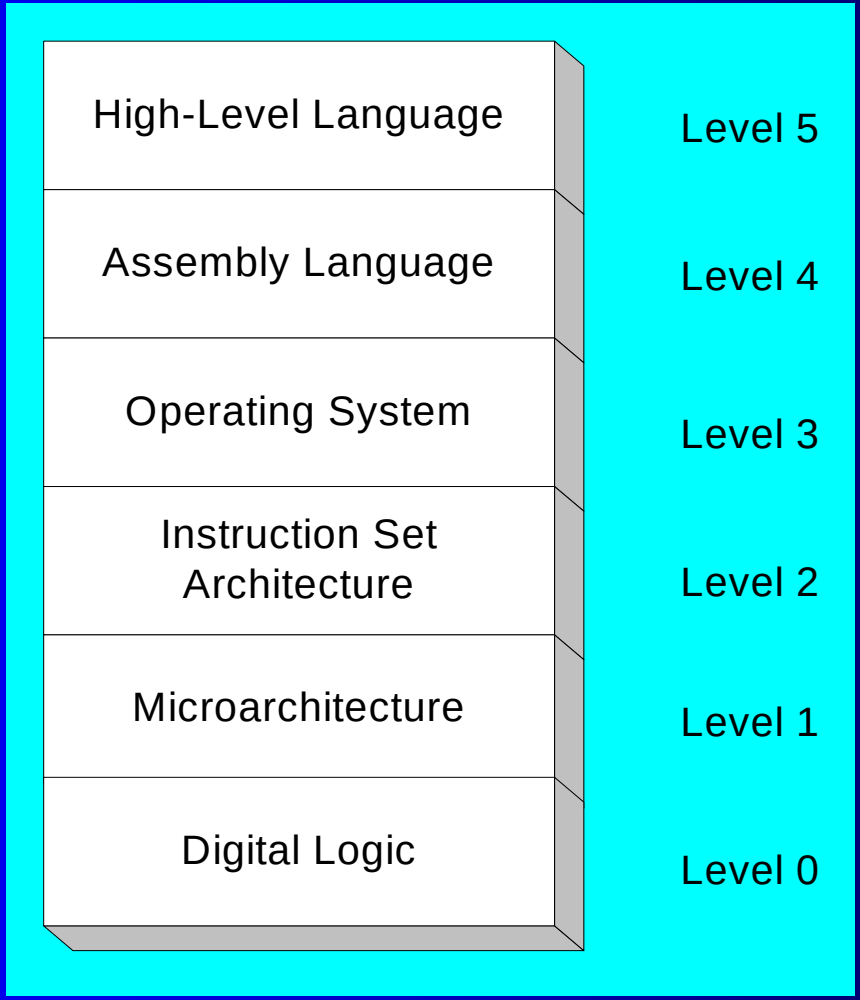
Virtual Machines [1/2]

- Virtual machine concept
  - A most effective way to explain how a computer's hardware and software are related
- In terms of programming languages
  - 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
- Programs written in L1 can run two different ways:
  - 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 hardware

Virtual Machines [2/2]

- In terms of a hypothetical computer
  - VM1 can execute commands written in language L1.
  - VM2 can execute commands written in language L2.
  - The process can repeat until a virtual machine VMn can be designed that supports a powerful, easy-to-use language.
- The Java programming language is based on the virtual machine concept.
  - A program written in the Java language is translated by a Java compiler into Java byte code.
  - Java byte code: a low-level language that is quickly executed at run time by Java virtual machine (JVM).
  - The JVM has been implemented on many different computer systems, making Java programs relatively system-independent.

Specific Machine Levels



Digital Logic

- Level 0
- CPU, constructed from digital logic gates
- System bus
- Memory

Microarchitecture

- Level 1
- Interprets conventional machine instructions (Level 2)
- Executed by digital hardware (Level 0)
- A proprietary secret
  - Computer Chip manufacturers do not generally make it possible for average users to write microinstructions.

Instruction Set Architecture

- Level 2
- Also known as conventional machine language
- Executed by Level 1 program (microarchitecture, Level 1)
  - Each machine-language instruction is executed by several microinstructions.

Operating System

- Level 3
- A Level 3 machine understands interactive commands by users to load and execute programs, display directories, …
- Provides services to Level 4 programs
- Programs translated and run at the instruction set architecture level (Level 2)

Assembly Language

- Level 4
- Instruction mnemonics such as ADD, SUB and MOV that are easily translated to the instruction set architecture level (Level 2)
- Interrupt calls are executed directly by the operating system (Level 3)
- Assembly language programs are usually translated (assembled) in their entirety into machine language before they begin to execute.

High-Level Language

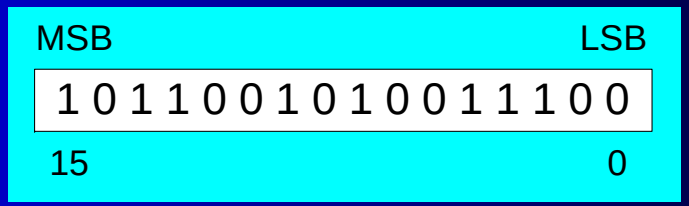
- Level 5
- Application-oriented languages (C++, C#, Virtual Basic, …)
- Programs compiled into assembly language (Level 4)
- Built-in assembly language

Data Representation

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

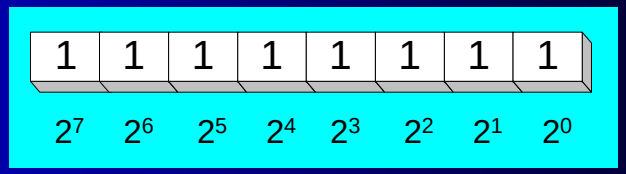
Binary Numbers

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



Binary Numbers

- 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

Translating Binary to Decimal

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

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

D = binary digit

binary 00001001 = decimal 9:

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

Translating Unsigned Decimal to Binary

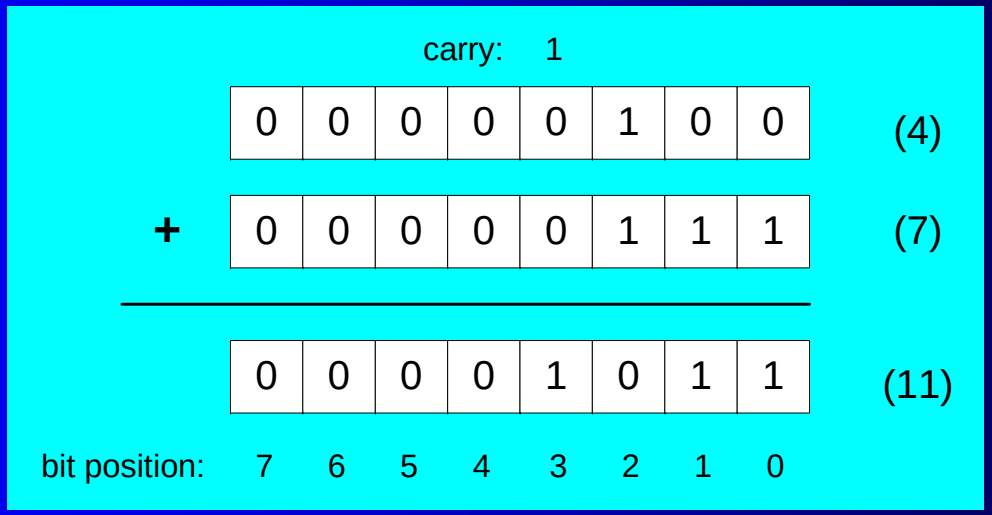
- Repeatedly divide the decimal integer by 2. Each remainder is a binary digit for the translated value:

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

Standard sizes:

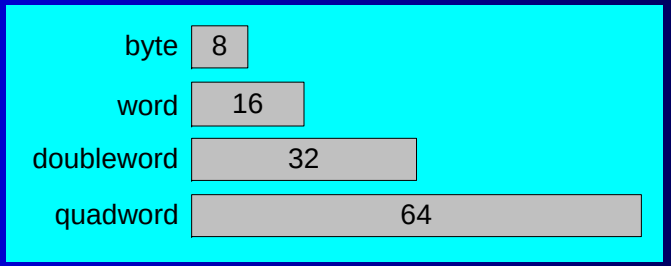


Table 1-4 Ranges of Unsigned Integers.

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 $(2^{64} - 1)$

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

Hexadecimal Integers

All values in memory are stored in binary. Because long binary numbers are hard to read, we use hexadecimal representation.

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

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

Powers of 16

Used when calculating hexadecimal values up to 8 digits long:

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 Decimal to Hexadecimal

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

decimal 422 = 1A6 hexadecimal

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.

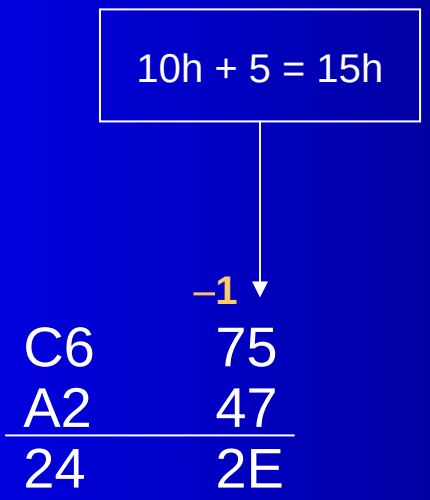
		1	1
36	28	28	6A
42	45	58	4B
78	6D	80	B5

21 / 16 = 1, rem 5

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

Hexadecimal Subtraction

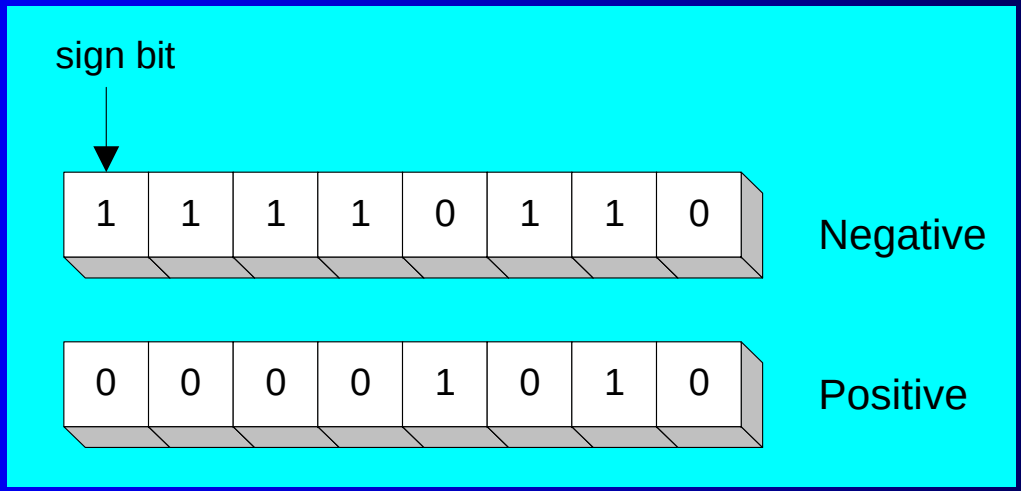
- When a borrow is required from the digit to the left, add 10h to the current digit's value:



Practice: The address of **var1** is 00400020. The address of the next variable after var1 is 0040006A. How many bytes are used by var1?

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

Forming the Two's Complement

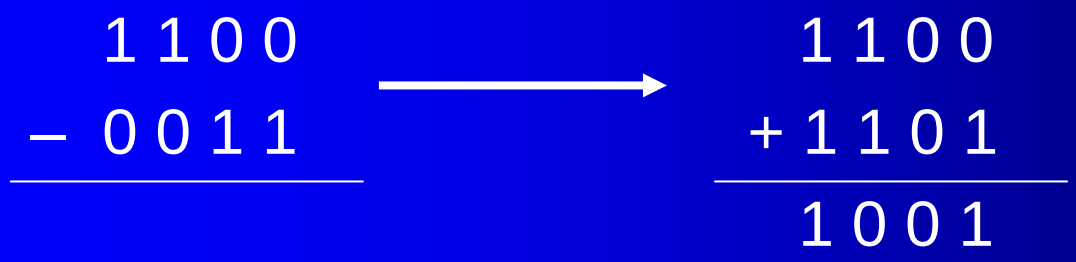
- Negative numbers are stored in two's complement notation.
- Additive Inverse of any binary integer (when a number's additive inverse is added to the number, their sum is zero).
- Steps:
  - Complement (reverse) each bit
  - Add 1

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

Binary Subtraction

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



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?

Character Storage

- Character sets
  - Standard ASCII (0 – 127): 7-bit integer
    - “ABC123” → 41h, 42h, 43h, 31h, 32h, 33h
  - Extended ASCII (0 – 255)
    - Graphics symbols and Greek characters
- Null-terminated String
  - Array of characters followed by a *null byte*
- Using the ASCII table
  - back inside cover of book

Numeric Data Representation

- Pure binary
  - Be a number stored in memory in its raw format
  - Can be calculated directly
  - Stored in multiples of 8 bits
- ASCII digit string
  - A string of ASCII characters

Format	Value
ASCII binary	**“01000001”**
ASCII decimal	**“65”**
ASCII hexadecimal	**“41”**
ASCII octal	**“101”**

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

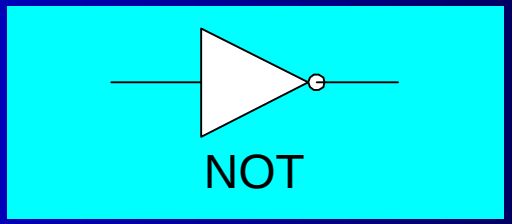
Expression	Description
¬X	NOT X
X ∧ Y	X AND Y
X ∨ Y	X OR Y
¬X ∨ Y	(NOT X) OR Y
¬(X ∧ Y)	NOT (X AND Y)
X ∧ ¬Y	X AND (NOT Y)

NOT

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

X	¬X
F	T
T	F

Digital gate diagram for NOT:

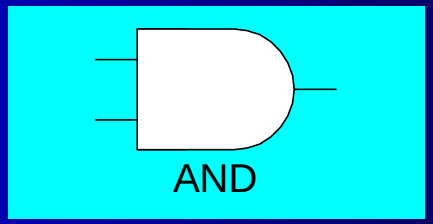


AND

- Truth table for Boolean AND operator:

X	Y	X ∧ Y
F	F	F
F	T	F
T	F	F
T	T	T

Digital gate diagram for AND:

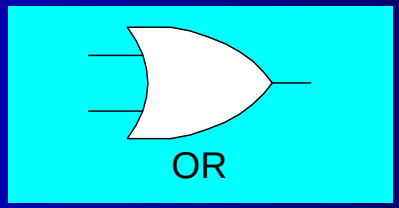


OR

- 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

- 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

Truth Tables [1/3]

- 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: $\neg X \vee Y$

X	$\neg X$	Y	$\neg X \vee Y$
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

Truth Tables [2/3]

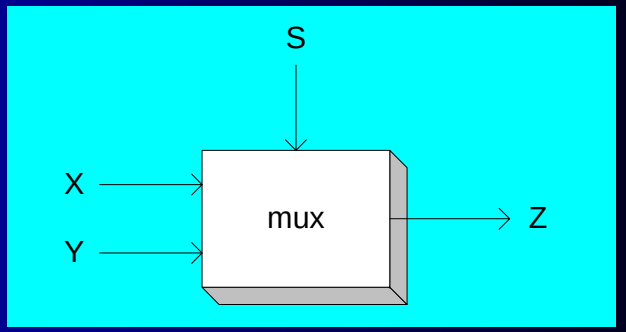
- Example: X ∧ ¬Y

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

Truth Tables [3/3]

- 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