Chapter 1

Basic Concepts

Chapter 1

Basic Concepts

1.1 Welcome to Assembly Language 1

- Assembly Language for Intel-Based computers focuses on programming microprocessors compatible with the Intel **IA-32** processor family on the **MS-Windows** platform.
- You can use an **Intel or AMD 32-bit/64-bit processor** to run all program in this book.
- The IA-32 family began with the Intel 80386, continuing to (and including) the Pentium4.
- Microsoft **MASM (Macro Assembler) 8.0** is our assembler of choice, running under MS-Windows.

1.1.1 Good Questions to Ask 2

- What background should I have?
  - You will better understand **high-level** programming constructs such as IF statements, loops, and arrays when implemented in assembly language.
- What are assemblers and Linkers?
  - An **assembler** is a utility program that converts source code programs from assembly language into **machine language**.
  - A **linker** is a utility program that combines individual files created by an assembler into **a single executable** program.
  - A related utility called a debugger; lets you to step through a program while it's running and examine registers and memory.
- What hardware and software do I need?
  - MASM (the assembler) is compatible with all 32-bit versions of MS Windows.
  - Editor: **Microsoft Visual C++ 2005 Express**
  - 32-bit Debugger: The debugger supplied with Visual C++ 2005 Express is excellent.
  - **16-Bit Real-Address Mode**: 16-bit real-address mode programs run under MS-DOS and in the **console window** under MS-Windows. Also known as **real mode** programs, they use a segmented memory model required of programs written for the Intel 8086 and 8088 processors.
  - **32-Bit protected Mode**: 32-bit **protected mode** programs run under all 32-bit versions of MS Windows.
- What do I get with this book?
  - Online Help File
  - Assembly Language Workbook
  - **Irvine32** and **Irvine16** link libraries
  - **Example programs**
  - Corrections
  - Tutorials
  - Articles
  - Discussion Group

- What will I learn?
  - Basic principles of computer architecture as applied to the Intel IA-32 processor family
  - Basic Boolean logic and how it applies to programming and computer hardware
  - How IA-32 processors manage memory, using real mode, protected mode, and virtual mode.
  - How high-level language compilers (such as C++) translate statements from their language into assembly language and native machine code.
  - How high-level languages implement arithmetic expressions, loops, and logical structures at the machine level.
  - Data representation, including signed and unsigned integers, real numbers, and character data.
  - How to debug programs at the machine level. The need for this skill is vital when you work in language such as C and C++, which provide access to low-level data and hardware.
  - How application programs communicate with the computer's operating system via interrupt handlers, system calls, and common memory areas.
  - How to interface assembly language code to C++ programs.
  - How to create assembly language application programs.
- How Does Assembly Language Relate to Machine Language?
  - 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++ and Java have a **one-to-many** relationship with assembly language and machine language.
  - The following C++ statement caries out two arithmetic operations and assigns the result to a variable. Assume X and Y are integer:

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

  - Following is the statement's translation to assembly language. The translation requires multiple statements because assembly language works at a detailed level:

    ```
    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
    ```

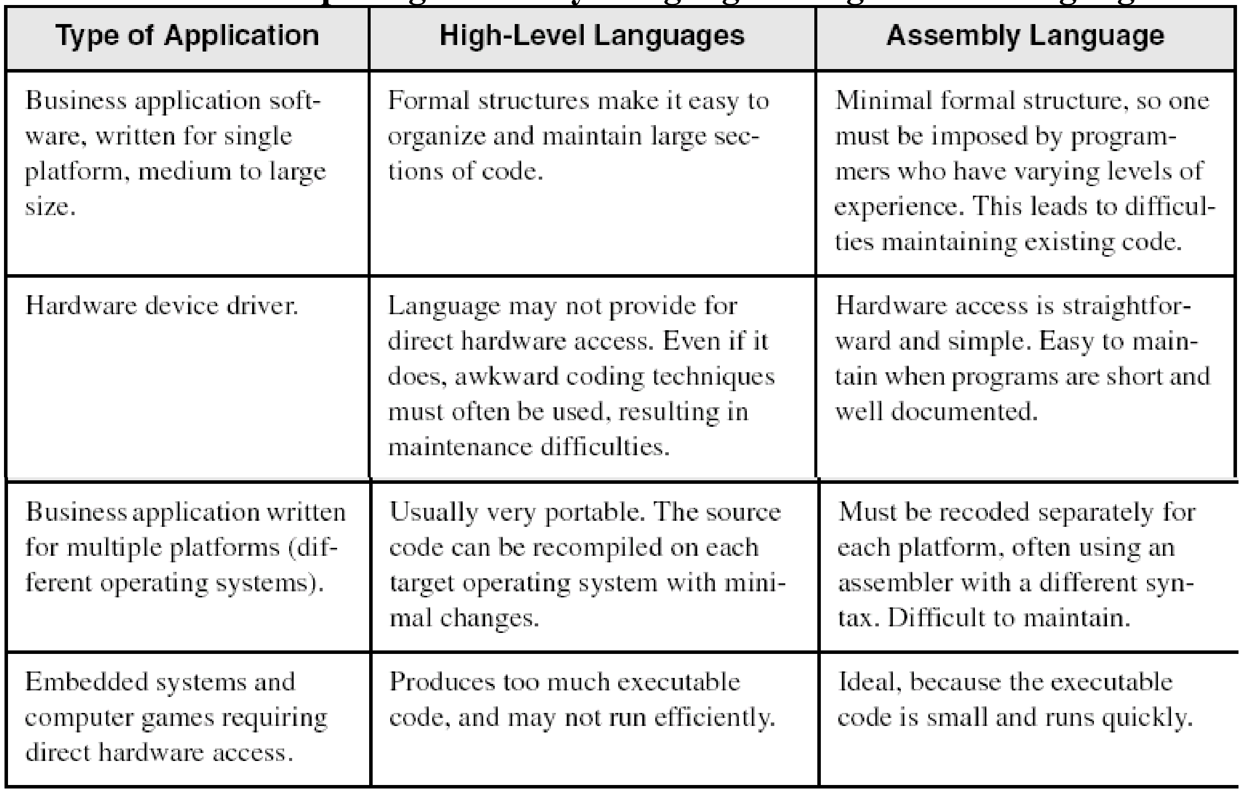
- Is Assembly Language Portable?
  - A language whose **source programs** can be compiled and run on a wide **variety** of computer systems is said to be **portable**. C++ programs, for example.
  - Assembly language is **not portable** because it is designed for a specific processor family.

- Why Learn Assembly Language?
  - If you study computer engineering, you may likely be asked to write **embedded programs**. They are short programs stored in a small amount of memory in single-purposed devices such as telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, data acquisition instruments, video cards, sound cards, hard drives, modems, and printers. Assembly language is ideal for writing embedded programs because of its economical use of memory.
  - **Real-time applications** such as simulations and hardware monitoring require precise **timing and responses**.
  - **Computer game** consoles require their software to be highly **optimized** for small code size and fast execution. It permits **direct** access to computer hardware and code can be hand optimized for speed.
  - Assembly language helps you to gain an **overall understanding** of the interaction between computer hardware, operating system, and application programs.
  - Application programmers occasionally find that **limitations in high-level languages** prevent then from efficiently performing low-level task such as bitwise manipulation and data **encryption**.
  - Hardware manufactures create **device drivers** for the equipment they sell. Device drivers are programs that translate general operating system commands into specific references to hardware details.
- Are There Rules in Assembly Language?
  - Most rules in assembly language are based on physical limitations of the **target processor** and its machine language.
  - Java, for example, **does not** permit access to specific memory address.
  - Assembly language, on the other hand, can access **any** memory address.

1.1.2 Assembly language Applications 5

- Some representative types of applications:
  - Business application for single platform: High-level language is better
  - Hardware device driver: Assembly language is better
  - Business application for multiple platforms: High-level language is better
  - Embedded systems & computer games: Assembly language is better

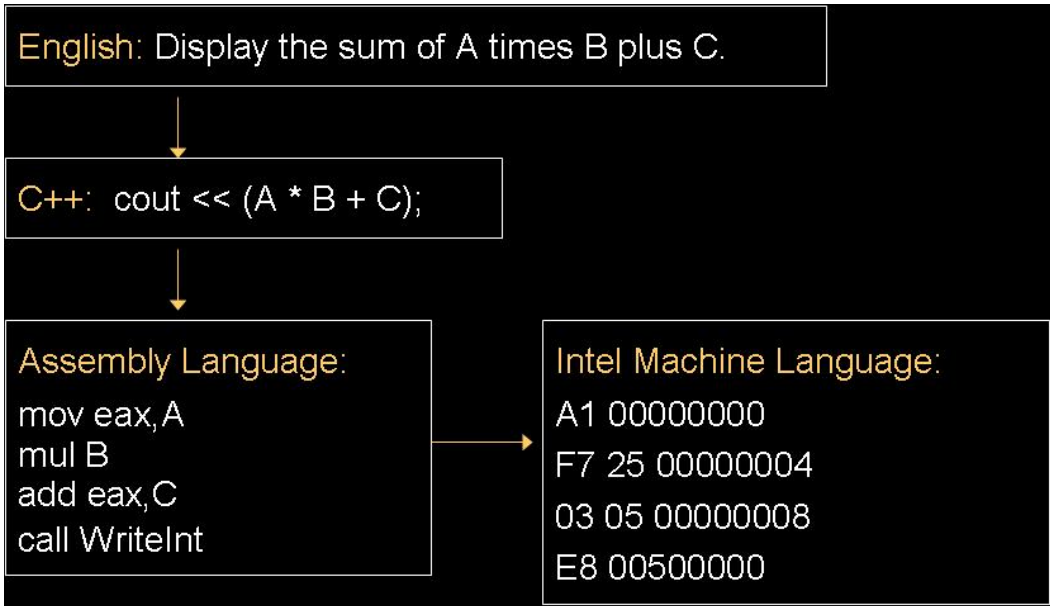
Table 1-1 Comparing Assembly Language to High-Level Languages



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.

1.2 Virtual Machine Concept 7

- 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
- 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
- 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**.
  - The latter is a low-level language **interprets and executed** at run time by a program known as a **Java Virtual Machine (JVM)**.
  - The JVM has been implemented on **many different** computer systems, making Java programs relatively system independent.
- 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
  - High-Level Language (Level 5):
    - Application-oriented languages such as C++, Java, Pascal, Visual Basic.
    - Programs in these languages contain powerful statements that translate into **multiple** instructions at assembly language (Level 4)
  - Assembly Language (Level 4):
    - Instruction **mnemonics** such as ADD, SUB, and MOV, which are easily translated to machine language (Level 2). Instruction mnemonics that have a **one-to-one** correspondence to machine language.
    - Calls functions written at the operating system level (Level 3)
    - Programs are translated into machine language (Level 2) before execution
  - Operating System (Level 3):
    - Provides services to Level 4 programs
    - Translated and run at the instruction set architecture level (Level 2)
    - Operating System understands interactive commands by users to load and execute programs, display directories, and so forth.
  - Instruction Set Architecture (Level 2):
    - Also known as conventional machine language
    - Executed by microarchitecture (Level 1) program
    - Computer chip manufactures design into the processor an instruction set to carry out basic operations, such as move, and, or multiply. Each machine-language instruction is executed by **several** microinstructions.
  - Microarchitecture (Level 1):
    - Interprets conventional machine instructions (Level 2)
    - Executed by digital hardware (Level 0)
    - The specific microarchitecture commands are often a proprietary **secret**.
  - Digital Logic (Level 0):
    - CPU, Memory: Constructed from digital logic gates

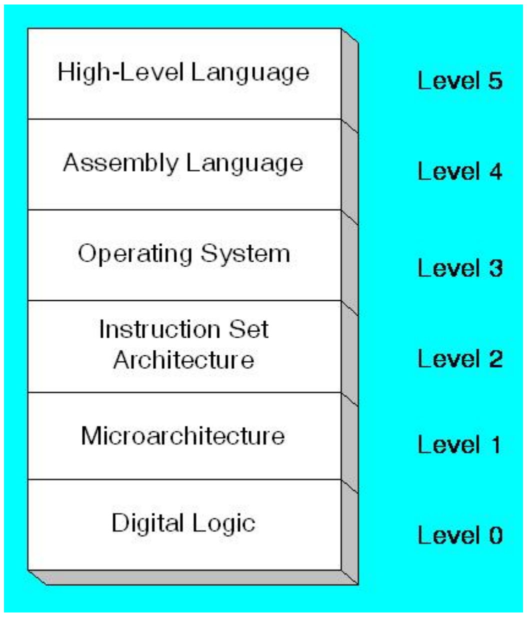


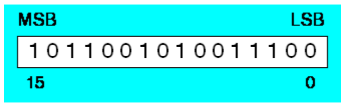
Figure 1-1 Virtual Mahine Levels 0 through 5

1.3 Data Representation 9

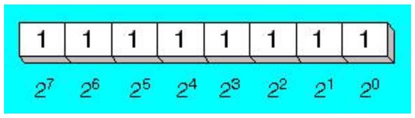
- Binary Numbers
  - Translating between binary and decimal
- Binary Addition
- Integer Storage Sizes
- Hexadecimal Integers
  - Translating between decimal and hexadecimal
  - Hexadecimal subtraction
- Signed Integers
  - Binary subtraction
- Character Storage

1.3.1 Binary Numbers 10

- Digits are 1 and 0: 1 = true; 0 = false
- MSB: Most Significant Bit
- LSB: Least Significant Bit
- Bit numbering:



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

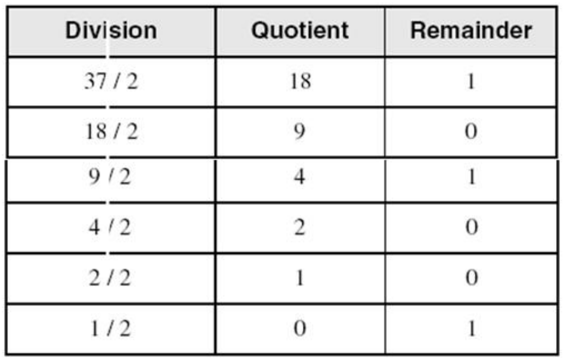
      $$\boldsymbol{dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0)}$$

      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 in the translated value:

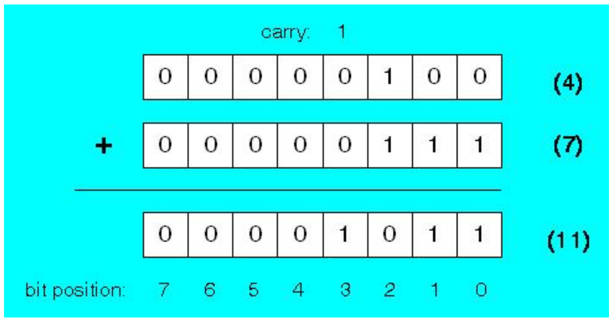


Division	Quotient	Remainder
37 / 2	18	1
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1

37 = 100101

1.3.2 Binary Addition 11

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



1.3.3 Integer Storage Sizes 12

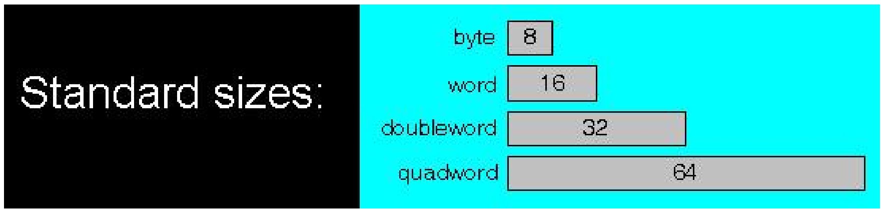


Table 1-4 Ranges of Unsigned Integers.

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 $(2^{64} - 1)$

- Large Measurements when referring to both memory and disk space

Prefix	Symbol	Power of 10	Power of 2
Kilo	K	1 thousand = 10^{3}	2^{10} = 1024
Mega	M	1 million = 10^{6}	2^{20}
Giga	G	1 billion = 10^{9}	2^{30}
Tera	T	1 trillion = 10^{12}	2^{40}
Peta	P	1 quadrillion = 10^{15}	2^{50}
Exa	E	1 quintillion = 10^{18}	2^{60}
Zetta	Z	1 sextillion = 10^{21}	2^{70}
Yotta	Y	1 septillion = 10^{24}	2^{80}

1.3.4 Hexadecimal Integers 13

- 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

- 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:

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

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

- The power of 16 from 16^0 to 16^7

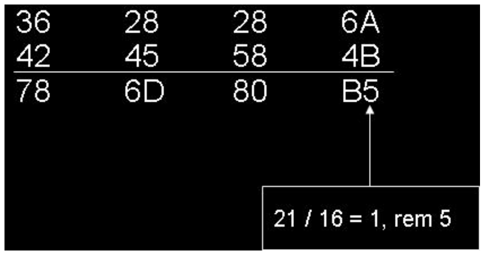
16^n	Decimal Value	16^n	Decimal Value
16^0	1	16^4	65,536
16^1	16	16^5	1,048,576
16^2	256	16^6	16,777,216
16^3	4096	16^7	268,435,456

- Converting Decimal to Hexadecimal

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

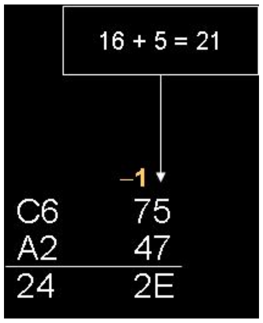
decimal 422 = 1A6 hexadecimal

- 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.



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

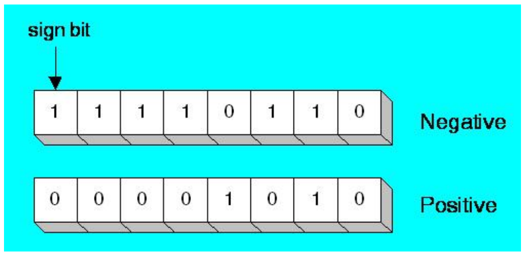
- Hexadecimal Subtraction
  - When a borrow is required from the digit to the left, add 16 (decimal) to the current digit's value:



```
      −1
C6    75
A2    47
24    2E
```

1.3.5 Signed Integers 14

- 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
    - Represents the additive Inverse
    - Note that 00000001 + 11111111 = 00000000

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

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

```
   0 0 0 0 1 1 0 0              0 0 0 0 1 1 0 0
 – 0 0 0 0 0 0 1 1   ──────►  + 1 1 1 1 1 1 0 1
                                0 0 0 0 1 0 0 1
```

- 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 $(2^{63} - 1)$

1.3.6 Character Storage 16

- 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
- Numeric Data Representation

pure binary	can be calculated directly
ASCII binary	string of digits: "01010101"
ASCII decimal	string of digits: "65"
ASCII hexadecimal	string of digits: "9C"

1.4 Boolean Operations 20

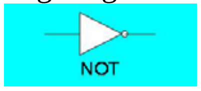
- Boolean algebra defines a set of operations on the values **true** and **false**.
- Boolean expression involves a Boolean operator and one or more operands.
- Boolean Algebra
  - Based on symbolic logic, designed by George Boole
  - Boolean expressions created from: NOT, AND, OR

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

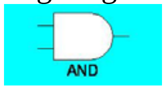
Digital gate diagram for NOT:



- AND
  - Truth table for Boolean AND operator:

X	Y	X ∧ Y
F	F	F
F	T	F
T	F	F
T	T	T

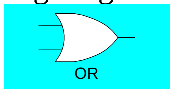
Digital gate diagram for AND:



- 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
  - The NOT operator has the highest precedence followed by AND and OR.
  - To avoid ambiguity, use parentheses to force the initial evaluation of an expression.
  - 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

1.4.1 Truth Tables for Boolean Functions 22

- 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: $\neg X \lor Y$

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

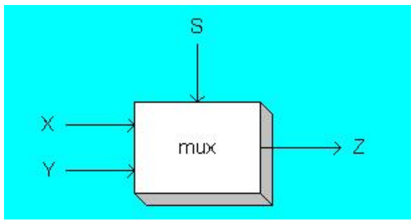
  - Example: $X \lor \neg Y$

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

  - Example: $(Y \land S) \lor (X \land \neg S)$

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

This Boolean function describes a **multiplexer**: two-input multiplexer



1.5 Chapter Summary   23

- This book focuses on programming microprocessors compatible with the Intel **IA-32** processor family, using the MS-Windows platform.
- Assembly language helps you learn how software is constructed at the **lowest** levels
- Assembly language has a **one-to-one** relationship with machine language.
- Assembly language is **not portable** because it is tied to a specific processor family.
- Virtual machine concept can be related to real-world computer layers, including **digital logic, microarchitecture, instruction set architecture, operating system, assembly language, and high-level languages**.
- 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
- A **truth table** is an effective way to show **all** possible inputs and output of a Boolean function.