Basic Concepts

COE 205

Computer Organization and Assembly Language

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Overview

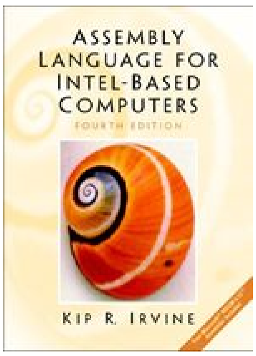
- ❖ Welcome to COE 205
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ Assembly Language Programming Tools
- ❖ Programmer's View of a Computer System
- ❖ Data Representation

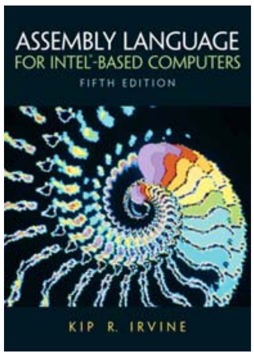
Welcome to COE 205

- Course Web Page:
  - http://www.ccse.kfupm.edu.sa/~mudawar/coe205/index.htm
- Course Lab Page
  - http://www.ccse.kfupm.edu.sa/~mudawar/coe205/lab/index.htm
  - Check with the Lab Instructor for more information about the new lab experiments
- Software Tools
  - Microsoft Macro Assembler (MASM) version 6.15
  - Link Libraries provided by Author (Irvine32.lib and Irivine16.lib)
  - Microsoft Windows debugger
  - ConTEXT Editor

Textbook

- Kip Irvine: Assembly Language for Intel-Based Computers
  - 4th edition (2003) is now available in the bookstore
  - 5th edition (2007) is coming soon but not available this semester
- Read the textbook!
  - Key for learning and obtaining a good grade





Goals and Required Background

- Goals: broaden student's interest and knowledge in …
  - Basic organization of a computer system
  - Intel IA-32 processor architecture
  - How to write assembly language programs
  - How high-level languages translate into assembly language
  - Interaction between the assembly language programs, libraries, the operating system, and the hardware
  - How interrupts, system calls, and handlers work
  - How to debug a program at the machine level
- Required Background
  - The student should already be able to program confidently in at least one high-level programming language, such as Java or C.

Grading Policy

Laboratory	20%
Assignments and Quizzes	25%
Midterm Exam I	15%
Midterm Exam II	20%
Final Exam	20%

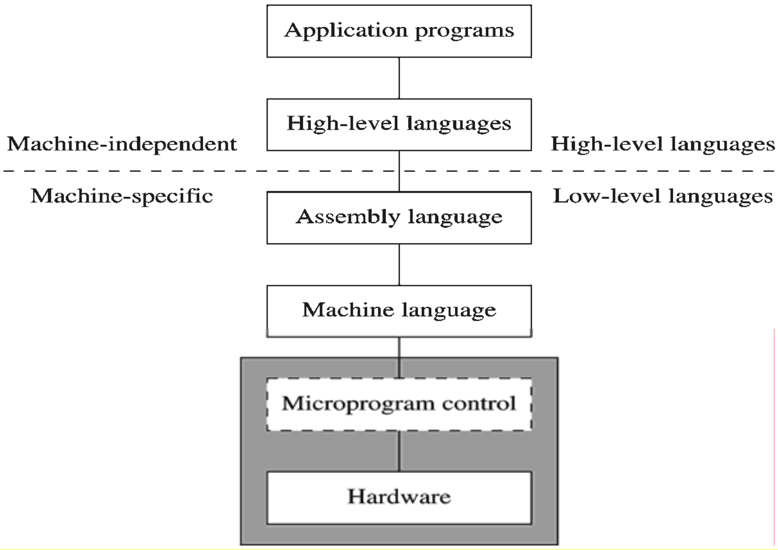
Next …

- ❖ Welcome to COE 205
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ Assembly Language Programming Tools
- ❖ Programmer's View of a Computer System
- ❖ Data Representation

Some Important Questions to Ask

- ❖ What is Assembly Language?
- ❖ Why Learn Assembly Language?
- ❖ What is Machine Language?
- ❖ How is Assembly related to Machine Language?
- ❖ What is an Assembler?
- ❖ How is Assembly related to High-Level Language?
- ❖ Is Assembly Language portable?

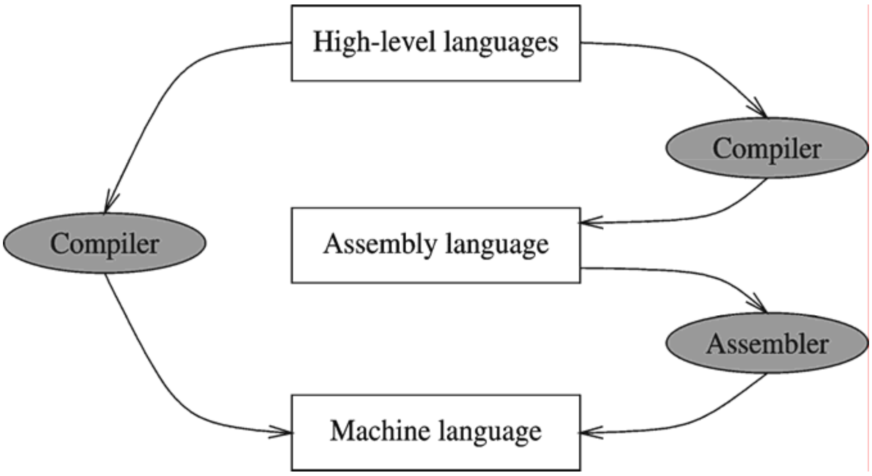
A Hierarchy of Languages



Assembly and Machine Language

- Machine language
  - Native to a processor: executed directly by hardware
  - Instructions consist of binary code: 1s and 0s
- Assembly language
  - Slightly higher-level language
  - Readability of instructions is better than machine language
  - One-to-one correspondence with machine language instructions
- Assemblers translate assembly to machine code
- Compilers translate high-level programs to machine code
  - Either directly, or
  - Indirectly via an assembler

Compiler and Assembler



Translating Languages

English: D is assigned the sum of A times B plus 10.

High-Level Language: D = A * B + 10

A statement in a high-level language is translated typically into several machine-level instructions

Intel Assembly Language:

```
mov   eax, A
mul   B
add   eax, 10
mov   D, eax
```

Intel Machine Language:

```
A1 00404000
F7 25 00404004
83 C0 0A
A3 00404008
```

Advantages of High-Level Languages

- Program development is faster
  - High-level statements: fewer instructions to code
- Program maintenance is easier
  - For the same above reasons
- Programs are portable
  - Contain few machine-dependent details
    - Can be used with little or no modifications on different machines
  - Compiler translates to the target machine language
  - However, Assembly language programs are not portable

Why Learn Assembly Language?

- Two main reasons:
  - Accessibility to system hardware
  - Space and time efficiency
- Accessibility to system hardware
  - Assembly Language is useful for implementing system software
  - Also useful for small embedded system applications
- Space and Time efficiency
  - Understanding sources of program inefficiency
  - Tuning program performance
  - Writing compact code

Assembly vs High-Level Languages

❖Some representative types of applications:

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.

Next ...

- ❖ Welcome to COE 205
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ Assembly Language Programming Tools
- ❖ Programmer's View of a Computer System
- ❖ Data Representation

Assembler

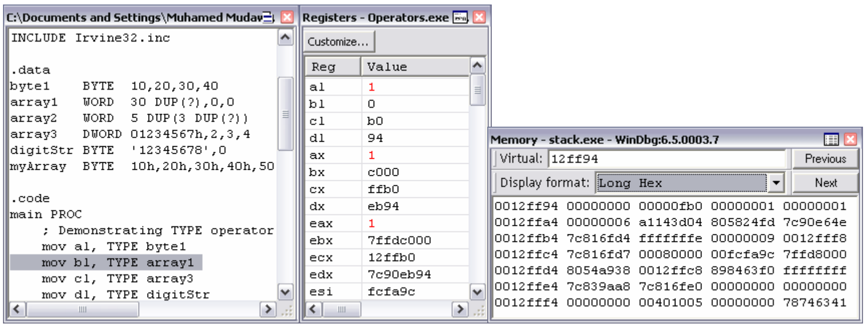
- Software tools are needed for editing, assembling, linking, and debugging assembly language programs
- An assembler is a program that converts source-code programs written in assembly language into object files in machine language
- Popular assemblers have emerged over the years for the Intel family of processors. These include …
  - TASM (Turbo Assembler from Borland)
  - NASM (Netwide Assembler for both Windows and Linux), and
  - GNU assembler distributed by the free software foundation
- You will use MASM (Macro Assembler from Microsoft)

Linker and Link Libraries

- You need a linker program to produce executable files
- It combines your program's object file created by the assembler with other object files and link libraries, and produces a single executable program
- LINK32.EXE is the linker program provided with the MASM distribution for linking 32-bit programs
- We will also use a link library for input and output
- Called Irvine32.lib developed by Kip Irvine
  - Works in Win32 console mode under MS-Windows

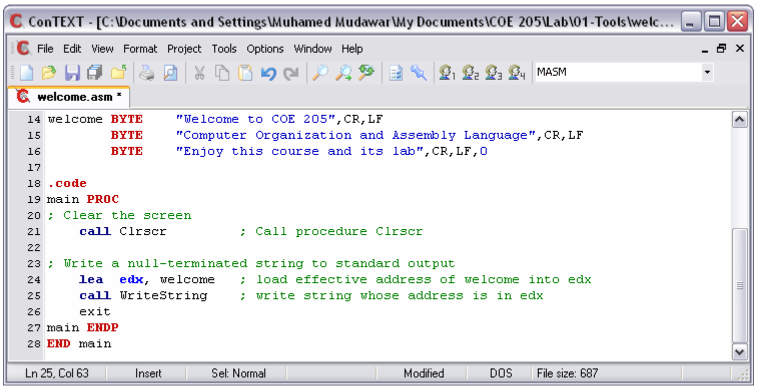
Debugger

- ❖ Allows you to trace the execution of a program
- ❖ Allows you to view code, memory, registers, etc.
- ❖ You will use the 32-bit Windows debugger



Editor

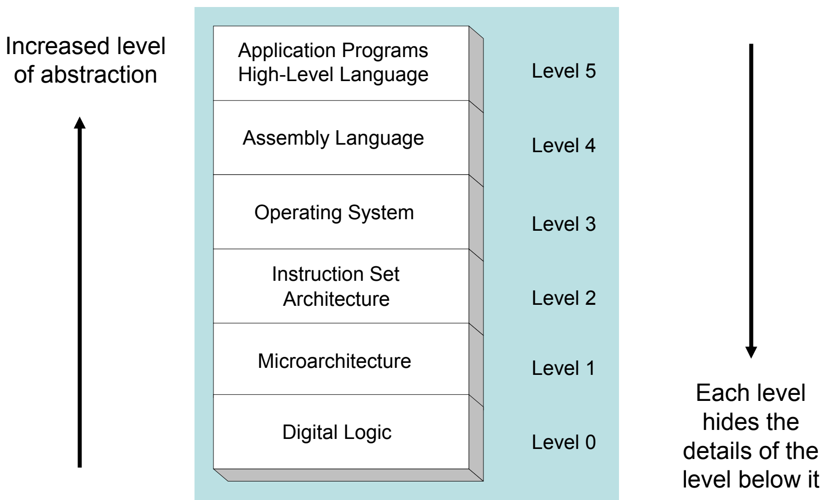
- ❖ Allows you to create assembly language source files
- ❖ Some editors provide syntax highlighting features and can be customized as a programming environment



Next ...

- ❖ Welcome to COE 205
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ Assembly Language Programming Tools
- ❖ Programmer's View of a Computer System
- ❖ Data Representation

Programmer's View of a Computer System



Programmer's View - 2

- Application Programs (Level 5)
  - Written in high-level programming languages
  - Such as Java, C++, Pascal, Visual Basic . . .
  - Programs compile into assembly language level (Level 4)
- Assembly Language (Level 4)
  - Instruction mnemonics are used
  - Have 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)
- Operating System (Level 3)
  - Provides services to level 4 and 5 programs
  - Translated to run at the machine instruction level (Level 2)

Programmer's View - 3

- Instruction Set Architecture (Level 2)
  - Specifies how a processor functions
  - Machine instructions, registers, and memory are exposed
  - Machine language is executed by Level 1 (microarchitecture)
- Microarchitecture (Level 1)
  - Controls the execution of machine instructions (Level 2)
  - Implemented by digital logic (Level 0)
- Digital Logic (Level 0)
  - Implements the microarchitecture
  - Uses digital logic gates
  - Logic gates are implemented using transistors

Next …

- Welcome to COE 205
- Assembly-, Machine-, and High-Level Languages
- Assembly Language Programming Tools
- Programmer's View of a Computer System
- Data Representation

Data Representation

- Binary Numbers
- Hexadecimal Numbers
- Base Conversions
- Integer Storage Sizes
- Binary and Hexadecimal Addition
- Signed Integers and 2's Complement Notation
- Binary and Hexadecimal subtraction
- Carry and Overflow
- Character Storage

Binary Numbers

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

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

Binary Numbers

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

1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

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

Converting Binary to Decimal

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

$$Decimal = (d_{n-1} \times 2^{n-1}) + (d_{n-2} \times 2^{n-2}) + ... + (d_1 \times 2^1) + (d_0 \times 2^0)$$

d = binary digit

binary 00001001 = decimal 9:

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

Convert 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 ← least significant bit
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1 ← most significant bit

stop when quotient is zero

37 = 100101

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

Converting 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:

  $Decimal = (d_3 \times 16^3) + (d_2 \times 16^2) + (d_1 \times 16^1) + (d_0 \times 16^0)$

  d = hexadecimal digit

- Examples:
  - Hex 1234 = $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$ =

    Decimal 4,660
  - Hex 3BA4 = $(3 \times 16^3) + (11 * 16^2) + (10 \times 16^1) + (4 \times 16^0)$ =

    Decimal 15,268

Converting Decimal to Hexadecimal

- Repeatedly divide the decimal integer by 16. Each remainder is a hex digit in the translated value:

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

least significant digit

most significant digit

stop when quotient is zero

Decimal 422 = 1A6 hexadecimal

Integer Storage Sizes

Standard sizes:

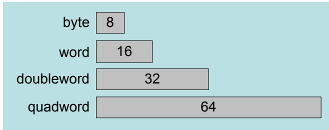


Table 1-4 Ranges of Unsigned Integers.

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 $(2^{64} - 1)$

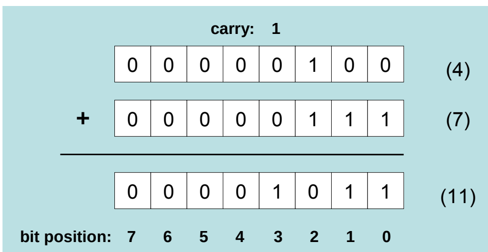
What is the largest unsigned integer that may be stored in 20 bits?

Binary Addition

❖ Start with the least significant bit (rightmost bit)

❖ Add each pair of bits

❖ Include the carry in the addition, if present



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, remainder 5

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

Signed Integers

❖ Several ways to represent a signed number
- ✧ Sign-Magnitude
- ✧ Biased
- ✧ 1's complement
- ✧ 2's complement

❖ Divide the range of values into 2 equal parts
- ✧ First part corresponds to the positive numbers (≥ 0)
- ✧ Second part correspond to the negative numbers (< 0)

❖ Focus will be on the 2's complement representation
- ✧ Has many advantages over other representations
- ✧ Used widely in processors to represent signed integers

Two's Complement Representation

- Positive numbers
  - Signed value = Unsigned value
- Negative numbers
  - Signed value = Unsigned value – 2^n
  - n = number of bits
- Negative weight for MSB
  - Another way to obtain the signed value is to assign a negative weight to most-significant bit

1	0	1	1	0	1	0	0
-128	64	32	16	8	4	2	1

= -128 + 32 + 16 + 4 = -76

8-bit Binary value	Unsigned value	Signed value
00000000	0	0
00000001	1	+1
00000010	2	+2
. . .	. . .	. . .
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
. . .	. . .	. . .
11111110	254	-2
11111111	255	-1

Forming the Two's Complement

starting value	**00100100 = +36**
step1: reverse the bits (1's complement)	**11011011**
step 2: add 1 to the value from step 1	**+ 1**
sum = 2's complement representation	**11011100 = -36**

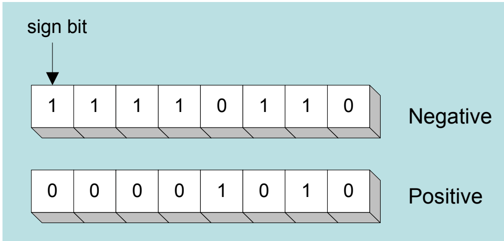
Sum of an integer and its 2's complement must be zero:

00100100 + 11011100 = 00000000 (8-bit sum) ⇒ Ignore Carry

The easiest way to obtain the 2's complement of a binary number is by starting at the LSB, leaving all the 0s unchanged, look for the first occurrence of a 1. Leave this 1 unchanged and complement all the bits after it.

Sign Bit

Highest bit indicates the sign. 1 = negative, 0 = positive



If highest digit of a hexadecimal is > 7, the value is negative

Examples: 8A and C5 are negative bytes

A21F and 9D03 are negative words

B1C42A00 is a negative double-word

Sign Extension

Step 1: Move the number into the lower-significant bits

Step 2: Fill all the remaining higher bits with the sign bit

- This will ensure that both magnitude and sign are correct
- Examples
  - Sign-Extend 10110011 to 16 bits

    10110011 = -77 ⇒ **11111111 10110011 = -77**
  - Sign-Extend 01100010 to 16 bits

    01100010 = +98 ⇒ **00000000 01100010 = +98**
- Infinite 0s can be added to the left of a positive number
- Infinite 1s can be added to the left of a negative number

Two's Complement of a Hexadecimal

- To form the two's complement of a hexadecimal
  - Subtract each hexadecimal digit from 15
  - Add 1
- Examples:

  2's complement of 6A3D = 95C2 + 1 = 95C3

  2's complement of 92F0 = 6D0F + 1 = 6D10

  2's complement of FFFF = 0000 + 1 = 0001
- No need to convert hexadecimal to binary

Binary Subtraction

- When subtracting A – B, convert B to its 2'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 0   ⇒    +  1 1 1 1 1 1 1 0  (2's complement)
  ---------------           ---------------
  0 0 0 0 1 0 1 0            0 0 0 0 1 0 1 0  (same result)
```

- Carry is ignored, because
  - Negative number is sign-extended with 1's
  - You can imagine infinite 1's to the left of a negative number
  - Adding the carry to the extended 1's produces extended zeros

Practice: Subtract 00100101 from 01101001.

Hexadecimal Subtraction

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

```
 16 + 5 = 21
     -1                     1 1
   C675                     C675
 - A247       ==>         + 5DB9   (2's complement)
   ----                     ----
   242E                     242E   (same result)
```

❖ Last Carry is ignored

Practice: The address of **var1** is 00400B20. The address of the next variable after var1 is 0040A06C. How many bytes are used by var1?

Ranges of Signed Integers

The unsigned range is divided into two signed ranges for positive and negative numbers

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 range of signed values that may be stored in 20 bits?

Carry and Overflow

- Carry is important when …
  - Adding or subtracting unsigned integers
  - Indicates that the unsigned sum is out of range
  - Either < 0 or >maximum unsigned *n*-bit value
- Overflow is important when …
  - Adding or subtracting signed integers
  - Indicates that the signed sum is out of range
- Overflow occurs when
  - Adding two positive numbers and the sum is negative
  - Adding two negative numbers and the sum is positive
  - Can happen because of the fixed number of sum bits

Carry and Overflow Examples

- We can have carry without overflow and vice-versa
- Four cases are possible

				1					
	0	0	0	0	1	1	1	1	15
+	0	0	0	0	1	0	0	0	8
	0	0	0	1	0	1	1	1	23

Carry = 0 Overflow = 0

1	1	1	1	1					
	0	0	0	0	1	1	1	1	15
+	1	1	1	1	1	0	0	0	245 (-8)
	0	0	0	0	0	1	1	1	7

Carry = 1 Overflow = 0

	1								
	0	1	0	0	1	1	1	1	79
+	0	1	0	0	0	0	0	0	64
	1	0	0	0	1	1	1	1	143 (-113)

Carry = 0 Overflow = 1

1			1	1					
	1	1	0	1	1	0	1	0	218 (-38)
+	1	0	0	1	1	1	0	1	157 (-99)
	0	1	1	1	0	1	1	1	119

Carry = 1 Overflow = 1

Character Storage

❖ Character sets

- ✧ Standard ASCII: 7-bit character codes (0 – 127)
- ✧ Extended ASCII: 8-bit character codes (0 – 255)
- ✧ Unicode: 16-bit character codes (0 – 65,535)
- ✧ Unicode standard represents a universal character set
  - Defines codes for characters used in all major languages
  - Used in Windows-XP: each character is encoded as 16 bits
- ✧ UTF-8: variable-length encoding used in HTML
  - Encodes all Unicode characters
  - Uses 1 byte for ASCII, but multiple bytes for other characters

❖ Null-terminated String

- ✧ Array of characters followed by a NULL character

Printable ASCII Codes

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td>2</td><td>space</td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>'</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>3</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>4</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>5</td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>\</td><td>]</td><td>^</td><td>_</td></tr>
<tr><td>6</td><td>`</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td></tr>
<tr><td>7</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td><td>x</td><td>y</td><td>z</td><td>{</td><td>|</td><td>}</td><td>~</td><td>DEL</td></tr>
</table>

❖ Examples:

- ✧ ASCII code for space character = 20 (hex) = 32 (decimal)
- ✧ ASCII code for 'L' = 4C (hex) = 76 (decimal)
- ✧ ASCII code for 'a' = 61 (hex) = 97 (decimal)

Control Characters

- The first 32 characters of ASCII table are used for control
- Control character codes = 00 to 1F (hex)
  - Not shown in previous slide
- Examples of Control Characters
  - Character 0 is the NULL character ⇒ used to terminate a string
  - Character 9 is the Horizontal Tab (HT) character
  - Character 0A (hex) = 10 (decimal) is the Line Feed (LF)
  - Character 0D (hex) = 13 (decimal) is the Carriage Return (CR)
  - The LF and CR characters are used together
    - They advance the cursor to the beginning of next line
- One control character appears at end of ASCII table
  - Character 7F (hex) is the Delete (DEL) character

Terminology for Data Representation

- Binary Integer
  - Integer stored in memory in its binary format
  - Ready to be used in binary calculations
- ASCII Digit String
  - A string of ASCII digits, such as "123"
- ASCII binary
  - String of binary digits: "01010101"
- ASCII decimal
  - String of decimal digits: "6517"
- ASCII hexadecimal
  - String of hexadecimal digits: "9C7B"

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
- An assembler is a program that converts assembly language programs into machine language
- A linker combines individual files created by an assembler into a single executable file
- A debugger provides a way for a programmer to trace the execution of a program and examine the contents of memory and registers
- A computer system can be viewed as consisting of layers. Programs at one layer are translated or interpreted by the next lower-level layer
- Binary and Hexadecimal numbers are essential for programmers working at the machine level.