Contents

How to Use This Book

The examples in this book demonstrate features of the Intel/AMD x64 Assembly programming language, and the screenshots illustrate the actual results produced by the listed code examples. The examples are created for the Microsoft Macro Assembler (MASM) that is included with the free Visual Studio Community Edition IDE. Certain colorization conventions are used to clarify the code listed in the book's easy steps...

Assembly directives and instructions are colored blue, register names are purple, labels and function names are red, literal text and numeric values are black, and code comments are green:

```
INCLUDELIB kernel32.lib ; Import a library.
ExitProcess PROTO       ; Define an imported function.

.CODE                   ; Start of the code section.
main PROC               ; Start of the main procedure.
XOR RCX, RCX            ; Clear a register.
MOV RCX, 10             ; Initialize a counter.
CALL ExitProcess        ; Return control to the system.
main ENDP               ; End of the main procedure.
END                     ; End of the program.
```

To identify the source code for the example programs described in the steps, an icon and project name appear in the margin alongside the steps:

ASM

SIMD

Grab the Source Code

For convenience, the source code files from all examples featured in this book are available in a single ZIP archive. You can obtain this archive by following these easy steps:

1. Browse to **www.ineasysteps.com** then navigate to Free Resources and choose the Downloads section

2. Next, find Assembly x64 Programming in easy steps in the list, then click on the hyperlink entitled All Code Examples to download the ZIP archive file

3. Now, extract the archive contents to any convenient location on your computer

4. Each **Source.asm file** can be added to a Visual Studio project to run the code

If you don't achieve the result illustrated in any example, simply compare your code to that in the original example files you have downloaded to discover where you went wrong.

1 Beginning Basics

Welcome to the exciting world of Assembly programming. This chapter describes the Assembly language, computer architecture, and data representation.

Introducing Assembly

Assembly (ASM) is a low-level programming language for a computer or other programmable device. Unlike high-level programming languages such as C++, which are typically portable across different systems, the Assembly language targets a specific system architecture. This book demonstrates Assembly programming for the x86-64 computer system architecture – also known as "x64", "Intel64" and "AMD64".

Assembly language code is converted into machine instructions by an "assembler" utility program. There are several assemblers available but this book uses only the Microsoft Macro Assembler (MASM) on the Windows operating system.

The Assembly example programs in this book are created and tested in the free Community Edition of the Microsoft Visual Studio Integrated Development Environment (IDE).

Hot tip

Assembly is the only programming language that speaks directly to the computer's CPU.

At the heart of every computer is a microprocessor chip called the Central Processing Unit (CPU) that handles system operations, such as receiving input from the keyboard and displaying output on the screen. The CPU only understands "machine language instructions". These are binary strings of ones and zeros, which are in themselves too obscure for program development. The Assembly language overcomes this difficulty by providing symbols that represent machine language instructions in a useful format. Assembly is, therefore, also known as symbolic machine code.

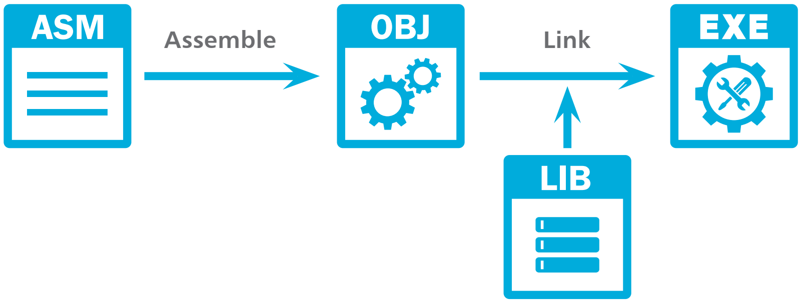
Why learn Assembly language?

By learning Assembly you will discover...

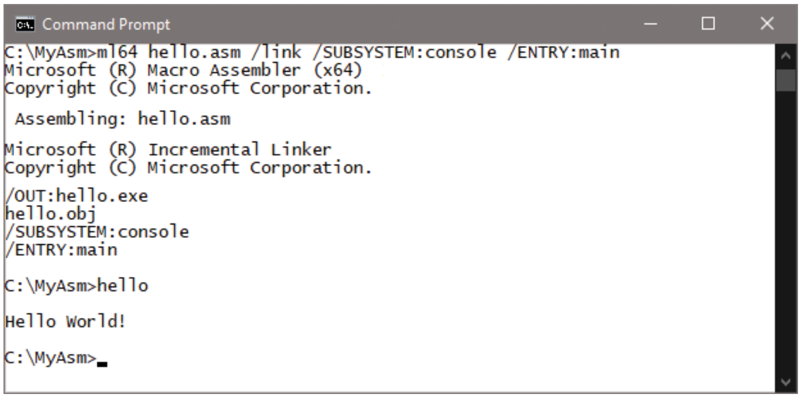
- How the CPU accesses and executes instructions.
- How data is represented in computer memory.
- How instructions access and process data.
- How programs interface with the operating system.
- How to debug a high-level program with disassembly.
- How to handle memory addresses and pointers.
- How to create fast programs that require less memory.

Assembling and Linking

Creation of an executable program from Assembly source code is a two-stage process that requires the assembler to first create an object file, containing actual machine code, then a "linker" to incorporate any required library code and produce the executable.



The 64-bit Microsoft Macro Assembler is a utility program named **ml64.exe** and the Microsoft Incremental Linker is a utility program named **link.exe**. Assembling and linking can be performed on the command line in a directory folder containing the Assembly source code file and both utility programs. The command must specify an Assembly source code file name to the **ml64** program, then call the linker with **/link**. Additionally, it must specify a **/subsystem**: execution mode (**console** or **windows**) and an **/entry**: point – the name of the first procedure to be executed in the Assembly code. The program can then be executed by name. For example, from an Assembly source code file named **hello.asm** with a **main** procedure:



```
C:\MyAsm>ml64 hello.asm /link /SUBSYSTEM:console /ENTRY:main
Microsoft (R) Macro Assembler (x64)
Copyright (C) Microsoft Corporation.

 Assembling: hello.asm

Microsoft (R) Incremental Linker
Copyright (C) Microsoft Corporation.

/OUT:hello.exe
hello.obj
/SUBSYSTEM:console
/ENTRY:main

C:\MyAsm>hello

Hello World!

C:\MyAsm>_
```

Don't forget

Assembling and linking is performed automatically when using the Visual Studio IDE, but the command line option is described here only to demonstrate the process. All examples in this book will be created and executed in the Visual Studio IDE.

Hot tip

You can find the source code of this program listed on page 154.

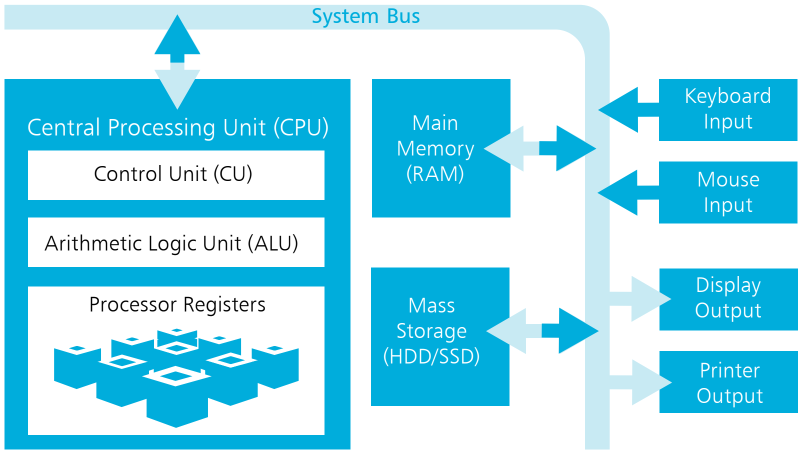
Inspecting Architecture

The fundamental design of nearly all computer systems is based upon a 1945 description by the eminent mathematician John von Neumann. The "von Neumann architecture" describes a computer design containing these components:

- **Processing unit** – containing an arithmetic logic unit and processor registers.
- **Control unit** – containing an instruction register and program counter.
- **Memory** – in which to store data and instructions.
- **External mass storage** – hard disk drive/solid state drive.
- **Input and Output mechanisms** – keyboard, mouse, etc.

Today's computers have a CPU (combining the control unit, arithmetic logic unit, and processor registers), main memory (Random Access Memory – "RAM"), external mass storage (Hard Disk Drive "HDD" or Solid State Drive "SSD"), and various I/O devices for input and output. These components are all connected by the "system bus", which allows data, address, and control signals to transfer between the components.

Computer system architecture defined by the Hungarian-American mathematician John von Neumann (1903-1957).



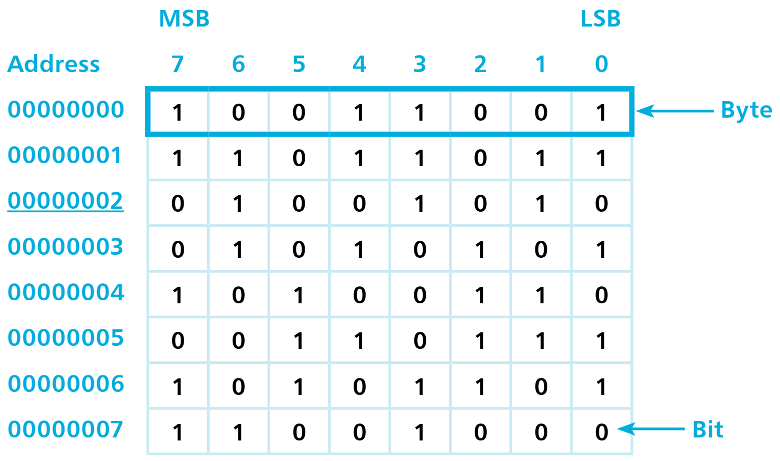
The CPU is the "brain" of the computer. A program is loaded into memory, then the CPU gets instructions from memory and executes them. Accessing memory is slow though, so the CPU contains registers in which to store data that it can access quickly.

The control unit can decode and execute instructions fetched from memory, and direct the operations of the CPU. The arithmetic logic unit performs arithmetic operations, such as addition and subtraction, plus logical operations, such as **AND** and **OR**. This means that all data processing is done within the CPU.

Hot tip

Logical operations are described and demonstrated later – see page 18.

The conceptual view of main memory shown below consists of rows of blocks. Each block is called a "bit" and can store a 0 or 1. The right-most bit (**0**) is called the "Least Significant Bit" (LSB) and the left-most bit (**7**) is called the "Most Significant Bit" (MSB).



	MSB							LSB
Address	7	6	5	4	3	2	1	0
00000000	1	0	0	1	1	0	0	1
00000001	1	1	0	1	1	0	1	1
00000002	0	1	0	0	1	0	1	0
00000003	0	1	0	1	0	1	0	1
00000004	1	0	1	0	0	1	1	0
00000005	0	0	1	1	0	1	1	1
00000006	1	0	1	0	1	1	0	1
00000007	1	1	0	0	1	0	0	0

As a bit can only store 0 or 1, a group of bits can be used to represent larger numbers. For example, two bits can represent numbers 0-3, three bits can represent numbers 0-7, and so on.

Data allocation directives defines these groups of bits in Assembly:

- **BYTE** – Byte (8-bits) range 0-255, or -128 to 127
- **WORD** – Word (16-bits) range 0-65,535, or -32,768 to 32,767
- **DWORD** – Double word (32-bits) range 0-2^{32}, or -2^{31} to 2^{31}-1
- **QWORD** – Quad word (64-bits) range 0-2^{64}, or -2^{63} to 2^{63}-1

Hot tip

Consecutive addresses are allocated for groups of bits that contain multiple bytes, and their content is accessed via the memory address of the first byte.

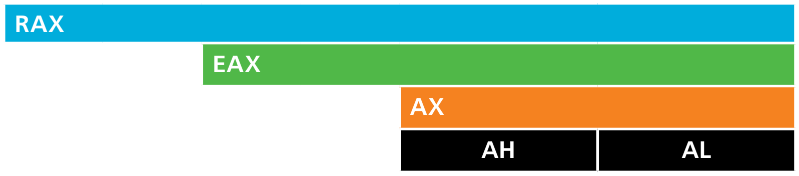
Each row in the view above is a byte – usually the smallest addressable unit of memory. Each byte has a unique address by which you can access its content – for example, to access the content of the third row here via address **00000002**.

Addressing Registers

Although an x86-64 CPU has many registers, it includes 16 general-purpose 64-bit user-accessible registers that are of special significance in Assembly programming. The first eight of these are extensions of eight registers from the earlier Intel 8086 microprocessor, and are addressed in Assembly programming by their historic names **RAX**, **RBX**, **RCX**, **RDX**, **RSI**, **RDI**, **RBP** and **RSP**. The rest are addressed as **R8**, **R9**, **R10**, **R11**, **R12**, **R13**, **R14** and **R15**.

Low-order (right-side) byte, word, and double word fractions of the 64-bit registers can be addressed individually. For example, **AL** (low byte) and **AH** (high byte), **AX** (word) and **EAX** (double word) are all fractions of the **RAX** register.

64-bit	32-bit	16-bit	8-bit
RAX	EAX	AX	AH AL
RBX	EBX	BX	BH BL
RCX	ECX	CX	CH CL
RDX	EDX	DX	DH DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B



Don't forget

Storing a value in the smaller fractional parts of a 64-bit register does not affect the higher bits, but storing a value in a 32-bit register will fill the top of the 64-bit register with zeros.

The **RSP** and **RBP** 64-bit registers are used for stack operations and stack frame operations, but all the other registers can be used for computation in your Assembly programs.

Hot tip

Stack (RSP) and stack frame (RBP) operations are described later – see Chapter 7.

Some 64-bit registers serve an additional purpose – either directly on the CPU hardware or in the x64 Windows calling convention. When a function is called within an Assembly program, the caller may pass it a number of argument values that get assigned, in sequential order, to the **RCX**, **RDX**, **R8**, and **R9** registers.

A procedure will only save those values within non-volatile registers – those values within volatile registers may be lost.

Register	Hardware	Calling Convention	Volatility
RAX	Accumulator	Function return value	Volatile
RBX	Base		Non-volatile
RCX	Counter	1st Function argument	Volatile
RDX	Data	2nd Function argument	Volatile
RSI	Source Index		Non-volatile
RDI	Destination Index		Non-volatile
RBP	Base Pointer		Non-volatile
RSP	Stack Pointer		Non-volatile
R8	General-purpose	3rd Function argument	Volatile
R9	General-purpose	4th Function argument	Volatile
R10	General-purpose		Volatile
R11	General-purpose		Volatile
R12	General-purpose		Non-volatile
R13	General-purpose		Non-volatile
R14	General-purpose		Non-volatile
R15	General-purpose		Non-volatile

One more 64-bit register to be aware of is the **RIP** instruction pointer register. This should not be used directly in your Assembly programs as it stores the address of the next instruction to execute. The CPU will execute the instruction at the address in the **RIP** register then increment the register to point to the next instruction.

Hot tip

Some Assembly instructions can modify the **RIP** instruction pointer address to make the program jump to a different location – see page 62.

Numbering Systems

64
40h
0010 0000b

Numeric values in Assembly program code may appear in the familiar decimal format that is used in everyday life, but computers use the binary format to store the values. Binary numbers are lengthy strings of zeros and ones, which are difficult to read so the hexadecimal format is often used to represent binary data. Comparison of these three numbering systems is useful to understand how numeric values are represented in each system.

Decimal (base 10) – uses numbers 0-9.

Columns from right to left are the value 10 raised to the power of an incrementing number, starting at zero:

10^2	10^1	10^0
1	2	8

$8 \times 10^0 = 8$
$2 \times 10^1 = 20$ (2x10)
$1 \times 10^2 = \underline{100}$ (1x10x10)
$\underline{128}$

1	0	1	1	0	0	1	0
128	64	32	16	8	4	2	1
MSB							LSB

Binary (base 2) – uses numbers 0 and 1.

Columns from right to left are the value 2 raised to the power of an incrementing number, starting at zero:

2^3	2^2	2^1	2^0
1	0	0	1

$1 \times 2^0 = 1$
$0 \times 2^1 = 0$
$0 \times 2^2 = 0$
$1 \times 2^3 = \underline{8}$ (1x2x2x2)
$\underline{9}$

1001 binary = **9** decimal

Hexadecimal (base 16) – uses numbers 0-9 & letters A-F.

Columns from right to left are the value 16 raised to the power of an incrementing number, starting at zero:

16^2	16^1	16^0
2	3	F

$15 \times 16^0 = 15$
$3 \times 16^1 = 48$ (3x16)
$2 \times 16^2 = \underline{512}$ (2x16x16)
$\underline{575}$

23F hexadecimal = **575** decimal

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Converting Binary to Hexadecimal

The table on the left can be used to easily convert a value between binary and hexadecimal numbering systems.

Notice that each hexadecimal digit represents four binary digits. This means you can separate any binary number into groups of 4-bits from right to left, then substitute the appropriate hexadecimal digit for each group. If the left-most group has less than 4-bits just add leading zeros. For example:

11 1101 0101

becomes

$$\underbrace{0011}_{3}\ \underbrace{1101}_{D}\ \underbrace{0101}_{5}$$

1111010101 binary = **3D5** hexadecimal

A 4-bit group is called a "nibble" (sometimes spelled as "nybble").

Similarly, you can use the table to convert each hexadecimal digit to the equivalent group of 4 binary digits to easily convert a value between hexadecimal and binary numbering systems.
For example:

$$\overbrace{0110}^{6}\ \overbrace{1100}^{C}\ \overbrace{0100}^{4}$$

6C4 hexadecimal = **011011000100** binary

Denoting the Numbering System

When using numbers in your Assembly programs you must denote which numbering system they are using if not the decimal system. Add a **b** suffix to denote binary, or add an **h** suffix for hexadecimal.

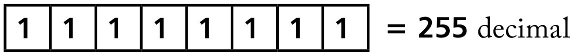
Decimal:	**21**
Binary:	**00010101b**
Hexadecimal:	**15h**

Signing Numbers

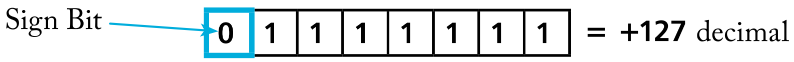
Binary numbers can represent unsigned positive numbers (zero is also considered positive) or signed positive and negative numbers. Page 15 describes the binary representation of unsigned numbers and their conversion to hexadecimal and decimal.

In representing signed numbers, the left-most Most Significant Bit (MSB) is used to denote whether the number is positive or negative. For positive numbers, this "sign bit" will contain a 0, whereas for negative numbers, the sign bit will contain a 1. This reduces the numeric range capacity of a bit group by one bit.

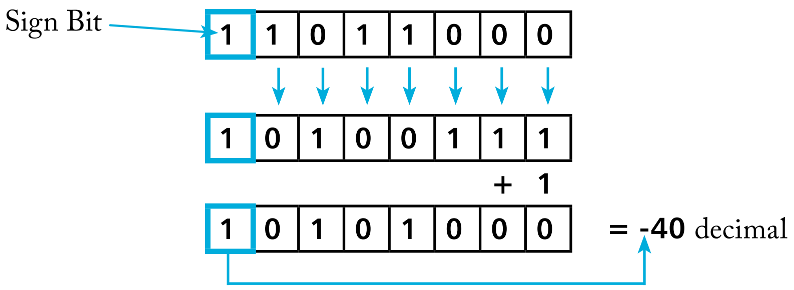
For any group of **N** number of bits, the maximum unsigned number is calculated as $\mathbf{2^N -1}$. For example, with a byte group (8-bits), the capacity is $\mathbf{2^8 -1}$, or **256 -1**, so the range is **0-255**.



But when the MSB is used as a sign bit, the maximum signed number is calculated as $\mathbf{2^{N-1} -1}$. For example, with a byte group (8-bits), the capacity is $\mathbf{2^{8-1} -1}$, which is $\mathbf{2^7 -1}$ or **128 -1**, so the range is **0-127**.



Negative signed numbers are stored in binary as a "Two's Complement" representation. To convert this to a decimal number, the value in each bit (except the sign bit) must be inverted – so that **0**s become **1**s, and **1**s become **0**s. Then, add **1** to the Least Significant Bit (LSB) using binary arithmetic. Finally, observe the sign value denoted by the sign bit. For example:



Hot tip

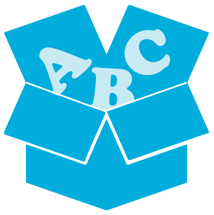
Signed groups of bits in Assembly programming are defined as **SBYTE**, **SWORD**, and **SDWORD**.

Hot tip

You can use the Calculator app's Scientific options in Windows to calculate the result of raising to power values.

Storing Characters

Just as signed and unsigned decimal numbers can be stored in binary format, so too can alphanumeric characters. The American Standard Code for Information Interchange (ASCII) provides a unique code for individual characters. The basic ASCII standard supplies individual codes for 128 characters and these differentiate between uppercase and lowercase characters. Each code is a unique 7-bit number, so a byte is used to store each character. For example, the uppercase letter **A** has a decimal code of **65** (**41** hexadecimal) and is stored in a byte as the binary number **01000001**. The alphanumeric ASCII codes are listed below:



Decimal	Hex	Character	Decimal	Hex	Character
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z

Don't forget

There are ASCII codes for the numeral characters 0-9. The character 5, for example, is not the same as the number 5.

Decimal	Hex	Character
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9

Hot tip

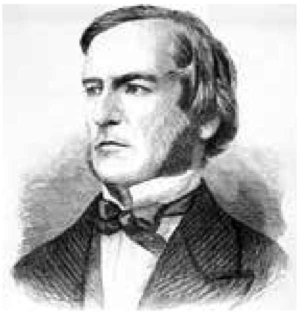
ASCII was later expanded to represent more characters in ANSI character code.

Using Boolean Logic

The CPU recognizes **AND**, **OR**, **XOR**, **TEST** and **NOT** instructions to perform boolean logic operations, which can be used in Assembly programming to set, clear, and test bit values. The syntax of these instructions looks like this:

AND	*Operand1* **,** *Operand2*
OR	*Operand1* **,** *Operand2*
XOR	*Operand1* **,** *Operand2*
TEST	*Operand1* **,** *Operand2*
NOT	*Operand1*

In all cases, the first operand can be either the name of a register or system memory, whereas the second operand can be the name of a register, system memory, or an immediate numeric value.


The term "boolean" refers to a system of logical thought developed by the English mathematician George Boole (1815-1864).

AND Operation

The **AND** operation compares two bits and returns a **1** only if both bits contain a value of **1** – otherwise it returns **0**. For example:

```
               Operand1:   0101
               Operand2:   0011
After AND...   Operand1:   0001
```

The **AND** operation can be used to check whether a number is odd or even by comparing the Least Significant Bit in the first operand to **0001**. If the LSB contains **1** the number is odd, otherwise the number is even.

OR Operation

The **OR** operation compares two bits and returns a **1** if either or both bits contain a **1** – if both are **0** it returns **0**. For example:

```
               Operand1:   0101
               Operand2:   0011
After OR...    Operand1:   0111
```

The **OR** operation can be used to set one or more bits by comparing the bit values in the first operand to selective bits containing a value of **1** in the second operand. This ensures that the selective bits will each return **1** in the result.

...cont'd

XOR Operation

The **XOR** (eXclusive OR) operation compares two bits and returns a **1** only if the bits contain different values – otherwise, if both are **1** or both are **0**, it returns **0**. For example:

	Operand1:	**0101**
	Operand2:	**0011**
After **XOR**...	Operand1:	**0110**

The **XOR** operation can be used to clear an entire register to zero by comparing all its bits to itself. In this case, all bits will match so the **XOR** operation returns a **0** in each bit of the register.

TEST Operation

The **TEST** operation works just like the **AND** operation, except it does not change the value in the first operand. Instead, the **TEST** operation sets a "zero flag" according to the result. For example:

	Operand1:	**0101**
	Operand2:	**0011**
After **TEST**...	Operand1:	**0101** (unchanged)

The **TEST** operation can be used to check whether a number is odd or even by comparing the Least Significant Bit in the first operand to **0001**. If the number is even, the zero flag is **1**, but if the number is odd, the **TEST** operation sets the zero flag to **0**.

Hot tip

A comprehensive description and demonstration of flags is given on page 60.

NOT Operation

The **NOT** operation inverts the value in each bit of a single operand – **1**s become **0**s, and **0**s become **1**s. For example:

	Operand1:	**0011**
After **NOT**...	Operand1:	**1100**

The **NOT** operation can be used to negate a signed binary number to a Two's Complement. The **NOT** operation will invert each bit value, then **1** can be added to the result to find the binary number's Two's Complement.

Summary

- Assembly is a low-level programming language that targets a specific computer system architecture.
- The assembler creates an object file containing machine code, and the linker incorporates any required library code.
- An executable program can be created from Assembly source code on the command line or in the Visual Studio IDE.
- Most computer systems are based on the von Neumann architecture with CPU, memory, storage and I/O devices.
- The CPU contains a control unit, arithmetic logic unit, and processor registers in which to store data for fast access.
- A byte has eight bits that can each store a 0 or a 1, and each byte has a unique memory address to access its content.
- A word consists of two bytes, a double word consists of four bytes, and a quad word consists of eight bytes (64-bits)
- An x86-64 CPU has 16 user-accessible 64-bit registers that are used in Assembly programming.
- Low-order byte, word, and double word fractions of the 64-bit registers can be addressed individually.
- Some 64-bit registers have a special purpose, either on the CPU hardware or in the x64 Windows calling convention.
- On completion of a procedure, values in non-volatile registers are saved but values in volatile registers may not be saved.
- Numeric values in Assembly programming may appear in the decimal, hexadecimal, or binary numbering system.
- Conversion between binary and hexadecimal is performed by separating a binary number into groups of 4-bits.
- Signed numbers are stored in binary as a Two's Complement representation.
- Characters are stored in binary as their ASCII code value.
- The CPU provides **AND**, **OR**, **XOR**, **TEST** and **NOT** instructions to perform boolean logic operations.