Assembly Language

FOR x86 PROCESSORS

Seventh Edition

Kip Irvine

ASCII CONTROL CHARACTERS

The following list shows the ASCII codes generated when a control key combination is pressed. The mnemonics and descriptions refer to ASCII functions used for screen and printer formatting and data communications.

ASCII Code*	Ctrl-	Mnemonic	Description	ASCII Code*	Ctrl-	Mnemonic	Description
00		NUL	Null character	10	Ctrl-P	DLE	Data link escape
01	Ctrl-A	SOH	Start of header	11	Ctrl-Q	DC1	Device control 1
02	Ctrl-B	STX	Start of text	12	Ctrl-R	DC2	Device control 2
03	Ctrl-C	ETX	End of text	13	Ctrl-S	DC3	Device control 3
04	Ctrl-D	EOT	End of transmission	14	Ctrl-T	DC4	Device control 4
05	Ctrl-E	ENQ	Enquiry	15	Ctrl-U	NAK	Negative acknowledge
06	Ctrl-F	ACK	Acknowledge	16	Ctrl-V	SYN	Synchronous idle
07	Ctrl-G	BEL	Bell	17	Ctrl-W	ETB	End transmission block
08	Ctrl-H	BS	Backspace	18	Ctrl-X	CAN	Cancel
09	Ctrl-I	HT	Horizontal tab	19	Ctrl-Y	EM	End of medium
0A	Ctrl-J	LF	Line feed	1A	Ctrl-Z	SUB	Substitute
0B	Ctrl-K	VT	Vertical tab	1B	Ctrl-I	ESC	Escape
0C	Ctrl-L	FF	Form feed	1C	Ctrl-\	FS	File separator
0D	Ctrl-M	CR	Carriage return	1D	Ctrl-]	GS	Group separator
0E	Ctrl-N	SO	Shift out	1E	Ctrl- ^	RS	Record separator
0F	Ctrl-O	SI	Shift in	1F	Ctrl-†	US	Unit separator

* ASCII codes are in hexadecimal.

† ASCII code 1Fh is Ctrl-Hyphen (-).

ALT-KEY COMBINATIONS

The following hexadecimal scan codes are produced by holding down the ALT key and pressing each character:

Key	Scan Code	Key	Scan Code	Key	Scan Code
1	78	A	1E	N	31
2	79	B	30	O	18
3	7A	C	2E	P	19
4	7B	D	20	Q	10
5	7C	E	12	R	13
6	7D	F	21	S	1F
7	7E	G	22	T	14
8	7F	H	23	U	16
9	80	I	17	V	2F
0	81	J	24	W	11
–	82	K	25	X	2D
=	83	L	26	Y	15
		M	32	Z	2C

KEYBOARD SCAN CODES

The following keyboard scan codes may be retrieved either by calling INT 16h or by calling INT 21h for keyboard input a second time (the first keyboard read returns 0). All codes are in hexadecimal:

FUNCTION KEYS

Key	Normal	With Shift	With Ctrl	With Alt
F1	3B	54	5E	68
F2	3C	55	5F	69
F3	3D	56	60	6A
F4	3E	57	61	6B
F5	3F	58	62	6C
F6	40	59	63	6D
F7	41	5A	64	6E
F8	42	5B	65	6F
F9	43	5C	66	70
F10	44	5D	67	71
F11	85	87	89	8B
F12	86	88	8A	8C

Key	Alone	With Ctrl Key
Home	47	77
End	4F	75
PgUp	49	84
PgDn	51	76
PrtSc	37	72
Left arrow	4B	73
Rt arrow	4D	74
Up arrow	48	8D
Dn arrow	50	91
Ins	52	92
Del	53	93
Back tab	0F	94
Gray +	4E	90
Gray –	4A	8E

Assembly Language for x86 Processors

Seventh Edition

KIP R. IRVINE

Florida International University
School of Computing and Information Sciences

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director, ECS: *Marcia Horton*
Executive Editor: *Tracy Johnson*
Executive Marketing Manager: *Tim Galligan*
Marketing Assistant: *Jon Bryant*
Program Management Team Lead: *Scott Disanno*
Program Manager: *Clare Romeo*
Project Manager: *Greg Dulles*
Senior Operations Specialist: *Nick Sklitsis*
Operations Specialist: *Linda Sager*
Permissions Project Manager: *Karen Sanatar*
Full-Service Project Management: *Pavithra Jayapaul, Jouve*
Printer/Binder: *Courier/Westford*
Typeface: *Times*

Previously published as *Assembly Language for Intel-Based Computers.*

Library of Congress Cataloging-in-Publication Data

Irvine, Kip R., 1951-
Assembly language for x86 processors / Kip R. Irvine, Florida International University, School of Computing and Information Sciences. — Seventh Edition.
pages cm
ISBN-13: 978-0-13-376940-1
ISBN-10: 0-13-376940-2
1. IBM microcomputers–Programming. 2. X86 assembly language (Computer program language) I. Title.
QA76.8.I77 2014
005.265—dc23 2013046432

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN-13: 978-0-13-376940-1
ISBN-10: 0-13-376940-2

To Jack and Candy Irvine

This page intentionally left blank

Contents

This page intentionally left blank

Preface

Assembly Language for x86 Processors, Seventh Edition, teaches assembly language programming and architecture for x86 and Intel64 processors. It is an appropriate text for the following types of college courses:

- Assembly Language Programming
- Fundamentals of Computer Systems
- Fundamentals of Computer Architecture

Students use Intel or AMD processors and program with **Microsoft Macro Assembler (MASM)**, running on recent versions of Microsoft Windows. Although this book was originally designed as a programming textbook for college students, it serves as an effective supplement to computer architecture courses. As a testament to its popularity, previous editions have been translated into numerous languages.

Emphasis of Topics This edition includes topics that lead naturally into subsequent courses in computer architecture, operating systems, and compiler writing:

- Virtual machine concept
- Instruction set architecture
- Elementary Boolean operations
- Instruction execution cycle
- Memory access and handshaking
- Interrupts and polling
- Hardware-based I/O
- Floating-point binary representation

Other topics relate specially to x86 and Intel64 architecture:

- Protected memory and paging
- Memory segmentation in real-address mode
- 16-Bit interrupt handling
- MS-DOS and BIOS system calls (interrupts)
- Floating-point unit architecture and programming
- Instruction encoding

Certain examples presented in the book lend themselves to courses that occur later in a computer science curriculum:

- Searching and sorting algorithms
- High-level language structures

- Finite-state machines
- Code optimization examples

What's New in the Seventh Edition

In this revision, we increased the discussions of program examples early in the book, added more supplemental review questions and key terms, introduced 64-bit programming, and reduced our dependence on the book's subroutine library. To be more specific, here are the details:

- Early chapters now include short sections that feature 64-bit CPU architecture and programming, and we have created a 64-bit version of the book's subroutine library named *Irvine64*.
- Many of the review questions and exercises have been modified, replaced, and moved from the middle of the chapter to the end of chapters, and divided into two sections: (1) Short answer questions, and (2) Algorithm workbench exercises. The latter exercises require the student to write a short amount of code to accomplish a goal.
- Each chapter now has a *Key Terms* section, listing new terms and concepts, as well as new MASM directives and Intel instructions.
- New programming exercises have been added, others removed, and a few existing exercises were modified.
- There is far less dependency on the author's subroutine libraries in this edition. Students are encouraged to call system functions themselves and use the Visual Studio debugger to step through the programs. The Irvine32 and Irvine64 libraries are available to help students handle input/output, but their use is not required.
- New tutorial videos covering essential content topics have been created by the author and added to the Pearson website.

This book is still focused on its primary goal, to teach students how to write and debug programs at the machine level. It will never replace a complete book on computer architecture, but it does give students the first-hand experience of writing software in an environment that teaches them how a computer works. Our premise is that students retain knowledge better when theory is combined with experience. In an engineering course, students construct prototypes; in a computer architecture course, students should write machine-level programs. In both cases, they have a memorable experience that gives them the confidence to work in any OS/machine-oriented environment.

Protected mode programming is entirely the focus of the printed chapters (1 through 13). As such, students will create 32-bit and 64-bit programs that run under the most recent versions of Microsoft Windows. The remaining four chapters cover 16-bit programming, and are supplied in electronic form. These chapters cover BIOS programming, MS-DOS services, keyboard and mouse input, video programming, and graphics. One chapter covers disk storage fundamentals. Another chapter covers advanced DOS programming techniques.

Subroutine Libraries We supply three versions of the subroutine library that students use for basic input/output, simulations, timing, and other useful tasks. The Irvine32 and Irvine64 libraries run in protected mode. The 16-bit version (Irvine16.lib) runs in real-address mode and is used only by Chapters 14 through 17. Full source code for the libraries is supplied on the companion website. The link libraries are available only for convenience, not to prevent students from learning how to program input–output themselves. Students are encouraged to create their own libraries.

Included Software and Examples All the example programs were tested with Microsoft Macro Assembler Version 11.0, running in Microsoft Visual Studio 2012. In addition, batch files are supplied that permit students to assemble and run applications from the Windows command prompt. The 32-bit C++ applications in Chapter 14 were tested with Microsoft Visual C++ .NET. Information Updates and corrections to this book may be found at the Companion Web site, including additional programming projects for instructors to assign at the ends of chapters.

Overall Goals

The following goals of this book are designed to broaden the student's interest and knowledge in topics related to assembly language:

- Intel and AMD processor architecture and programming
- Real-address mode and protected mode programming
- Assembly language directives, macros, operators, and program structure
- Programming methodology, showing how to use assembly language to create system-level software tools and application programs
- Computer hardware manipulation
- Interaction between assembly language programs, the operating system, and other application programs

One of our goals is to help students approach programming problems with a machine-level mind set. It is important to think of the CPU as an interactive tool, and to learn to monitor its operation as directly as possible. A debugger is a programmer's best friend, not only for catching errors, but as an educational tool that teaches about the CPU and operating system. We encourage students to look beneath the surface of high-level languages and to realize that most programming languages are designed to be portable and, therefore, independent of their host machines. In addition to the short examples, this book contains hundreds of ready-to-run programs that demonstrate instructions or ideas as they are presented in the text. Reference materials, such as guides to MS-DOS interrupts and instruction mnemonics, are available at the end of the book.

Required Background The reader should already be able to program confidently in at least one high-level programming language such as Python, Java, C, or C++. One chapter covers C++ interfacing, so it is very helpful to have a compiler on hand. I have used this book in the classroom with majors in both computer science and management information systems, and it has been used elsewhere in engineering courses.

Features

Complete Program Listings The Companion Web site contains supplemental learning materials, study guides, and all the source code from the book's examples. An extensive link library is supplied with the book, containing more than 30 procedures that simplify user input–output, numeric processing, disk and file handling, and string handling. In the beginning stages of the course, students can use this library to enhance their programs. Later, they can create their own procedures and add them to the library.

Programming Logic Two chapters emphasize Boolean logic and bit-level manipulation. A conscious attempt is made to relate high-level programming logic to the low-level details of the machine. This approach helps students to create more efficient implementations and to better understand how compilers generate object code.

Hardware and Operating System Concepts The first two chapters introduce basic hardware and data representation concepts, including binary numbers, CPU architecture, status flags, and memory mapping. A survey of the computer's hardware and a historical perspective of the Intel processor family helps students to better understand their target computer system.

Structured Programming Approach Beginning with Chapter 5, procedures and functional decomposition are emphasized. Students are given more complex programming exercises, requiring them to focus on design before starting to write code.

Java Bytecodes and the Java Virtual Machine In Chapters 8 and 9, the author explains the basic operation of Java bytecodes with short illustrative examples. Numerous short examples are shown in disassembled bytecode format, followed by detailed step-by-step explanations.

Disk Storage Concepts Students learn the fundamental principles behind the disk storage system on MS-Windows–based systems from hardware and software points of view.

Creating Link Libraries Students are free to add their own procedures to the book's link library and create new libraries. They learn to use a toolbox approach to programming and to write code that is useful in more than one program.

Macros and Structures A chapter is devoted to creating structures, unions, and macros, which are essential in assembly language and systems programming. Conditional macros with advanced operators serve to make the macros more professional.

Interfacing to High-Level Languages A chapter is devoted to interfacing assembly language to C and C++. This is an important job skill for students who are likely to find jobs programming in high-level languages. They can learn to optimize their code and see examples of how C++ compilers optimize code.

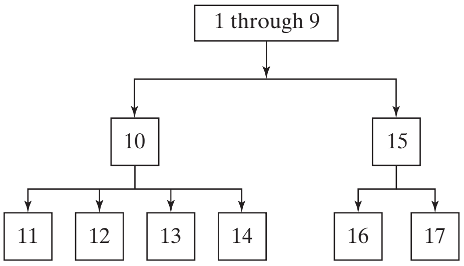
Instructional Aids All the program listings are available on the Web. Instructors are provided a test bank, answers to review questions, solutions to programming exercises, and a Microsoft PowerPoint slide presentation for each chapter.

VideoNotes VideoNotes are Pearson's new visual tool designed to teach students key programming concepts and techniques. These short step-by-step videos demonstrate basic assembly language concepts. VideoNotes allow for self-paced instruction with easy navigation including the ability to select, play, rewind, fast-forward, and stop within each VideoNote exercise.

VideoNotes are free with the purchase of a new textbook. To *purchase* access to VideoNotes, go to www.pearsonhighered.com/irvine and click on the VideoNotes under *Student Resources*.

Chapter Descriptions

Chapters 1 to 8 contain core concepts of assembly language and should be covered in sequence. After that, you have a fair amount of freedom. The following chapter dependency graph shows how later chapters depend on knowledge gained from other chapters.



1. **Basic Concepts:** Applications of assembly language, basic concepts, machine language, and data representation.
2. **x86 Processor Architecture:** Basic microcomputer design, instruction execution cycle, x86 processor architecture, Intel64 architecture, x86 memory management, components of a microcomputer, and the input–output system.
3. **Assembly Language Fundamentals:** Introduction to assembly language, linking and debugging, and defining constants and variables.
4. **Data Transfers, Addressing, and Arithmetic:** Simple data transfer and arithmetic instructions, assemble-link-execute cycle, operators, directives, expressions, JMP and LOOP instructions, and indirect addressing.
5. **Procedures:** Linking to an external library, description of the book's link library, stack operations, defining and using procedures, flowcharts, and top-down structured design.
6. **Conditional Processing:** Boolean and comparison instructions, conditional jumps and loops, high-level logic structures, and finite-state machines.
7. **Integer Arithmetic:** Shift and rotate instructions with useful applications, multiplication and division, extended addition and subtraction, and ASCII and packed decimal arithmetic.
8. **Advanced Procedures:** Stack parameters, local variables, advanced PROC and INVOKE directives, and recursion.
9. **Strings and Arrays:** String primitives, manipulating arrays of characters and integers, two-dimensional arrays, sorting, and searching.
10. **Structures and Macros:** Structures, macros, conditional assembly directives, and defining repeat blocks.
11. **MS-Windows Programming:** Protected mode memory management concepts, using the Microsoft-Windows API to display text and colors, and dynamic memory allocation.
12. **Floating-Point Processing and Instruction Encoding:** Floating-point binary representation and floating-point arithmetic. Learning to program the IA-32 floating-point unit. Understanding the encoding of IA-32 machine instructions.
13. **High-Level Language Interface:** Parameter passing conventions, inline assembly code, and linking assembly language modules to C and C++ programs.
    - **Appendix A:** MASM Reference
    - **Appendix B:** The x86 Instruction Set
    - **Appendix C:** Answers to Review Questions

The following chapters and appendices are supplied online at the Companion Web site:

14. **16-Bit MS-DOS Programming:** Memory organization, interrupts, function calls, and standard MS-DOS file I/O services.
15. **Disk Fundamentals:** Disk storage systems, sectors, clusters, directories, file allocation tables, handling MS-DOS error codes, and drive and directory manipulation.
16. **BIOS-Level Programming:** Keyboard input, video text, graphics, and mouse programming.
17. **Expert MS-DOS Programming:** Custom-designed segments, runtime program structure, and Interrupt handling. Hardware control using I/O ports.
    - **Appendix D:** BIOS and MS-DOS Interrupts
    - **Appendix E:** Answers to Review Questions (Chapters 14–17)

Instructor and Student Resources

Instructor Resource Materials

The following protected instructor material is available on the Companion Web site:

www.pearsonhighered.com/irvine

For username and password information, please contact your Pearson Representative.

- Lecture PowerPoint Slides
- Instructor Solutions Manual

Student Resource Materials

The student resource materials can be accessed through the publisher's Web site located at *www.pearsonhighered.com/irvine*. These resources include:

- VideoNotes
- Online Chapters and Appendices
    - Chapter 14: *16-Bit MS-DOS Programming*
    - Chapter 15: *Disk Fundamentals*
    - Chapter 16: *BIOS-Level Programming*
    - Chapter 17: *Expert MS-DOS Programming*
    - Appendix D: *BIOS and MS-DOS Interrupts*
    - Appendix E: *Answers to Review Questions (Chapters 14–17)*

Students must use the access card located in the front of the book to register and access the online chapters and VideoNotes. If there is no access card in the front of this textbook, students can purchase access by going to *www.pearsonhighered.com/irvine* and selecting "*Video Notes and Web Chapters*." Instructors must also register on the site to access this material. Students will also find a link to the author's Web site. An access card is not required for the following materials, located at *www.asmirvine.com*:

- *Getting Started*, a comprehensive step-by-step tutorial that helps students customize Visual Studio for assembly language programming.
- Supplementary articles on assembly language programming topics.
- Complete source code for all example programs in the book, as well as the source code for the author's supplementary library.

- *Assembly Language Workbook*, an interactive workbook covering number conversions, addressing modes, register usage, debug programming, and floating-point binary numbers. Content pages are HTML documents to allow for customization. Help File in Windows Help Format.
- Debugging Tools: Tutorials on using the Microsoft Visual Studio debugger.

Acknowledgments

Many thanks are due to Tracy Johnson, Executive Editor for Computer Science at Pearson Education, who has provided friendly, helpful guidance over the past few years. Pavithra Jayapaul of Jouve did an excellent job on the book production, along with Greg Dulles as the production editor at Pearson.

Previous Editions

I offer my special thanks to the following individuals who were most helpful during the development of earlier editions of this book:

- William Barrett, San Jose State University
- Scott Blackledge
- James Brink, Pacific Lutheran University
- Gerald Cahill, Antelope Valley College
- John Taylor

About the Author

Kip Irvine has written five computer programming textbooks, for Intel Assembly Language, C++, Visual Basic (beginning and advanced), and COBOL. His book *Assembly Language for Intel-Based Computers* has been translated into six languages. His first college degrees (B.M., M.M., and doctorate) were in Music Composition, at University of Hawaii and University of Miami. He began programming computers for music synthesis around 1982 and taught programming at Miami-Dade Community College for 17 years. Kip earned an M.S. degree in Computer Science from the University of Miami, and he has been a full-time member of the faculty in the School of Computing and Information Sciences at Florida International University since 2000.

This page intentionally left blank

1

Basic Concepts

This chapter establishes some core concepts relating to assembly language programming. For example, it shows how assembly language fits into the wide spectrum of languages and applications. We introduce the virtual machine concept, which is so important in understanding the relationship between software and hardware layers. A large part of the chapter is devoted to the binary and hexadecimal numbering systems, showing how to perform conversions and do basic arithmetic. Finally, this chapter introduces fundamental boolean operations (AND, OR, NOT, XOR), which will prove to be essential in later chapters.

1.1 Welcome to Assembly Language

Assembly Language for x86 Processors focuses on programming microprocessors compatible with Intel and AMD processors running under 32-bit and 64-bit versions of Microsoft Windows.

The latest version of *Microsoft Macro Assembler* (known as *MASM*) should be used with this book. MASM is included with most versions of Microsoft Visual Studio (Pro, Ultimate, Express, . . .). Please check our web site (*asmirvine.com*) for the latest details about support for MASM in Visual Studio. We also include lots of helpful information about how to set up your software and get started.

Some other well-known assemblers for x86 systems running under Microsoft Windows include TASM (Turbo Assembler), NASM (Netwide Assembler), and MASM32 (a variant of MASM). Two popular Linux-based assemblers are GAS (GNU assembler) and NASM. Of these, NASM's syntax is most similar to that of MASM.

Assembly language is the oldest programming language, and of all languages, bears the closest resemblance to native machine language. It provides direct access to computer hardware, requiring you to understand much about your computer's architecture and operating system.

Educational Value Why read this book? Perhaps you're taking a college course whose title is similar to one of the following courses that often use our book:

- Microcomputer Assembly Language
- Assembly Language Programming
- Introduction to Computer Architecture
- Fundamentals of Computer Systems
- Embedded Systems Programming

This book will help you learn basic principles about computer architecture, machine language, and low-level programming. You will learn enough assembly language to test your knowledge on today's most widely used microprocessor family. You won't be learning to program a "toy" computer using a simulated assembler; MASM is an industrial-strength assembler, used by practicing professionals. You will learn the architecture of the Intel processor family from a programmer's point of view.

If you are planning to be a C or C++ developer, you need to develop an understanding of how memory, address, and instructions work at a low level. A lot of programming errors are not easily recognized at the high-level language level. You will often find it necessary to "drill down" into your program's internals to find out why it isn't working.

If you doubt the value of low-level programming and studying details of computer software and hardware, take note of the following quote from a leading computer scientist, Donald Knuth, in discussing his famous book series, *The Art of Computer Programming*:

> Some people [say] that having machine language, at all, was the great mistake that I made. I really don't think you can write a book for serious computer programmers unless you are able to discuss low-level detail.[1]

Visit this book's web site to get lots of supplemental information, tutorials, and exercises at **www.asmirvine.com**

1.1.1 Questions You Might Ask

What Background Should I Have? Before reading this book, you should have programmed in at least one structured high-level language, such as Java, C, Python, or C++. You should know how to use IF statements, arrays, and functions to solve programming problems.

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? You need a computer that runs a 32-bit or 64-bit version of Microsoft Windows, along with one of the recent versions of Microsoft Visual Studio.

What Types of Programs Can Be Created Using MASM?

- *32-Bit Protected Mode:* 32-bit protected mode programs run under all 32-bit versions of Microsoft Windows. They are usually easier to write and understand than real-mode programs. From now on, we will simply call this *32-bit mode*.
- *64-Bit Mode:* 64-bit programs run under all 64-bit versions of Microsoft Windows.
- *16-Bit Real-Address Mode:* 16-bit programs run under 32-bit versions of Windows and on embedded systems. Because they are not supported by 64-bit Windows, we will restrict discussions of this mode to Chapters 14 through 17. These chapters are in electronic form, available from the publisher's web site.

What Supplements Are Supplied with This Book? The book's web site (*www.asmirvine.com*) has the following:

- ***Assembly Language Workbook***, a collection of tutorials
- ***Irvine32, Irvine64, and Irvine16 subroutine libraries*** for 64-bit, 32-bit, and 16-bit programming, with complete source code
- ***Example programs*** with all source code from the book
- ***Corrections*** to the book
- ***Getting Started***, a detailed tutorial designed to help you set up Visual Studio to use the Microsoft assembler
- ***Articles*** on advanced topics not included in the printed book for lack of space
- ***A link to an online discussion forum***, where you can get help from other experts who use the book

What Will I Learn? This book should make you better informed about data representation, debugging, programming, and hardware manipulation. Here's what you will learn:

- Basic principles of computer architecture as applied to x86 processors
- Basic boolean logic and how it applies to programming and computer hardware
- How x86 processors manage memory, using 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 languages such as C and C++, which generate native machine code
- How application programs communicate with the computer's operating system via interrupt handlers and system calls
- How to interface assembly language code to C++ programs
- How to create assembly language application programs

How Does Assembly Language Relate to Machine Language? *Machine language* is a numeric language specifically understood by a computer's processor (the CPU). All x86 processors understand a common machine language. *Assembly language* consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL. 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 Python, C++, and Java have a *one-to-many* relationship with assembly language and machine language. A single statement in C++, for example, expands into multiple assembly language or machine instructions. Most people cannot read raw machine code, so in this book, we examine its closest relative, assembly language. For example, the following C++ code carries out two arithmetic operations and assigns the result to a variable. Assume X and Y are integers:

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

Following is the equivalent translation to assembly language. The translation requires multiple statements because each assembly language statement corresponds to a single machine instruction:

```
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
```

(*Registers* are named storage locations in the CPU that hold intermediate results of operations.) The point of this example is not to claim that C++ is superior to assembly language or vice versa, but to show their relationship.

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*. A C++ program, for example, will compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system.

Assembly language is not portable, because it is designed for a specific processor family. There are a number of different assembly languages widely used today, each based on a processor family.

Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370. The instructions in assembly language may directly match the computer's architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter*.

Why Learn Assembly Language? If you're still not convinced that you should learn assembly language, consider the following points:

- 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-purpose 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 an ideal tool for writing embedded programs because of its economical use of memory.
- Real-time applications dealing with simulation and hardware monitoring require precise timing and responses. High-level languages do not give programmers exact control over machine code generated by compilers. Assembly language permits you to precisely specify a program's executable code.
- Computer game consoles require their software to be highly optimized for small code size and fast execution. Game programmers are experts at writing code that takes full advantage of hardware features in a target system. They often use assembly language as their tool of choice because 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 systems, and application programs. Using assembly language, you can apply and test theoretical information you are given in computer architecture and operating systems courses.
- Some high-level languages abstract their data representation to the point that it becomes awkward to perform low-level tasks such as bit manipulation. In such an environment, programmers will often call subroutines written in assembly language to accomplish their goal.
- Hardware manufacturers create device drivers for the equipment they sell. *Device drivers* are programs that translate general operating system commands into specific references to hardware details. Printer manufacturers, for example, create a different MS-Windows device driver for each model they sell. Often these device drivers contain significant amounts of assembly language code.

Are There Rules in Assembly Language? Most rules in assembly language are based on physical limitations of the target processor and its machine language. The CPU, for example, requires two instruction operands to be the same size. Assembly language has fewer rules than C++ or Java because the latter use syntax rules to reduce unintended logic errors at the expense of low-level data access. Assembly language programmers can easily bypass restrictions characteristic of high-level languages. Java, for example, does not permit access to specific memory addresses. One can work around the restriction by calling a C function using JNI (*Java Native Interface*) classes, but the resulting program can be awkward to maintain. Assembly language, on the other hand, can access any memory address. The price for such freedom is high: Assembly language programmers spend a lot of time debugging!

1.1.2 Assembly Language Applications

In the early days of programming, most applications were written partially or entirely in assembly language. They had to fit in a small area of memory and run as efficiently as possible on slow processors. As memory became more plentiful and processors dramatically increased in speed, programs became more complex. Programmers switched to high-level languages such as C, FORTRAN, and COBOL that contained a certain amount of structuring capability. More recently, object-oriented languages such as Python, C++, C#, and Java have made it possible to write complex programs containing millions of lines of code.

It is rare to see large application programs coded completely in assembly language because they would take too much time to write and maintain. Instead, assembly language is used to optimize certain sections of application programs for speed and to access computer hardware. Table 1-1 compares the adaptability of assembly language to high-level languages in relation to various types of applications.

Table 1-1 Comparison of Assembly Language to High-Level Languages.

Type of Application	High-Level Languages	Assembly Language
Commercial or scientific application, 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.	The language may not provide for direct hardware access. Even if it does, awkward coding techniques may be required, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Commercial or scientific application written for multiple platforms (different operating systems).	Usually portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	May produce large executable files that exceed the memory capacity of the device.	Ideal, because the executable code is small and runs quickly.

The C and C++ languages have the unique quality of offering a compromise between high-level structure and low-level details. Direct hardware access is possible but completely nonportable. Most C and C++ compilers allow you to embed assembly language statements in their code, providing access to hardware details.

1.1.3 Section Review

1. How do assemblers and linkers work together?
2. How will studying assembly language enhance your understanding of operating systems?

3. What is meant by a *one-to-many relationship* when comparing a high-level language to machine language?
4. Explain the concept of *portability* as it applies to programming languages.
5. Is the assembly language for x86 processors the same as those for computer systems such as the Vax or Motorola 68x00?
6. Give an example of an embedded systems application.
7. What is a device driver?
8. Do you suppose type checking on pointer variables is stronger (stricter) in assembly language, or in C and C++?
9. Name two types of applications that would be better suited to assembly language than a high-level language.
10. Why would a high-level language not be an ideal tool for writing a program that directly accesses a printer port?
11. Why is assembly language not usually used when writing large application programs?
12. *Challenge:* Translate the following C++ expression to assembly language, using the example presented earlier in this chapter as a guide: X = (Y * 4) + 3.

1.2 Virtual Machine Concept

An effective way to explain how a computer's hardware and software are related is called the *virtual machine concept*. A well-known explanation of this model can be found in Andrew Tanenbaum's book, *Structured Computer Organization.* To explain this concept, let us begin with the most basic function of a computer, executing programs.

A computer can usually execute programs written in its native *machine language*. Each instruction in this language is simple enough to be executed using a relatively small number of electronic circuits. For simplicity, we will call this language **L0**.

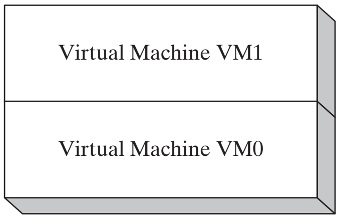
Programmers would have a difficult time writing programs in L0 because it is enormously detailed and consists purely of numbers. If a new language, **L1**, could be constructed that was easier to use, programs could be written in L1. There are two ways to achieve this:

- *Interpretation:* As the L1 program is running, each of its instructions could be decoded and executed by a program written in language L0. The L1 program begins running immediately, but each instruction has to be decoded before it can execute.
- *Translation:* The entire L1 program could be converted into an L0 program by an L0 program specifically designed for this purpose. Then the resulting L0 program could be executed directly on the computer hardware.

Virtual Machines

Rather than using only languages, it is easier to think in terms of a hypothetical computer, or *virtual machine*, at each level. Informally, we can define a virtual machine as a software program that emulates the functions of some other physical or virtual computer. The virtual machine

VM1, as we will call it, can execute commands written in language L1. The virtual machine **VM0** can execute commands written in language L0:



Each virtual machine can be constructed of either hardware or software. People can write programs for virtual machine VM1, and if it is practical to implement VM1 as an actual computer, programs can be executed directly on the hardware. Or programs written in VM1 can be interpreted/translated and executed on machine VM0.

Machine VM1 cannot be radically different from VM0 because the translation or interpretation would be too time-consuming. What if the language VM1 supports is still not programmer-friendly enough to be used for useful applications? Then another virtual machine, VM2, can be designed that is more easily understood. This process can be repeated until a virtual machine VM*n* can be designed to support 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*. The latter is a low-level language quickly executed at runtime 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.

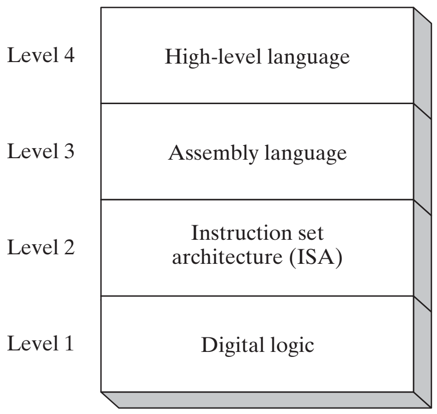
Specific Machines

Let us relate this to actual computers and languages, using names such as **Level 2** for VM2 and **Level 1** for VM1, shown in Figure 1-1. A computer's digital logic hardware represents machine Level 1. Above this is Level 2, called the *instruction set Architecture (ISA).* This is the first level at which users can typically write programs, although the programs consist of binary values called *machine language.*

Instruction Set Architecture (Level 2) Computer chip manufacturers design into the processor an instruction set to carry out basic operations, such as move, add, or multiply. This set of instructions is also referred to as *machine language*. Each machine-language instruction is executed either directly by the computer's hardware or by a program embedded in the microprocessor chip called a *microprogram*. A discussion of microprograms is beyond the scope of this book, but you can refer to Tanenbaum for more details.

Assembly Language (Level 3) Above the ISA level, programming languages provide translation layers to make large-scale software development practical. Assembly language, which appears at Level 3, uses short mnemonics such as ADD, SUB, and MOV, which are easily translated to the ISA level. Assembly language programs are translated (assembled) in their entirety into machine language before they begin to execute.

Figure 1–1 Virtual machine levels.



High-Level Languages (Level 4) At Level 4 are high-level programming languages such as C, C++, and Java. Programs in these languages contain powerful statements that translate into multiple assembly language instructions. You can see such a translation, for example, by examining the listing file output created by a C++ compiler. The assembly language code is automatically assembled by the compiler into machine language.

1.2.1 Section Review

1. In your own words, describe the *virtual machine* concept.
2. Why do you suppose translated programs often execute more quickly than interpreted ones?
3. *(True/False):* When an interpreted program written in language L1 runs, each of its instructions is decoded and executed by a program written in language L0.
4. Explain the importance of translation when dealing with languages at different virtual machine levels.
5. At which level does assembly language appear in the virtual machine example shown in this section?
6. What software utility permits compiled Java programs to run on almost any computer?
7. Name the four virtual machine levels named in this section, from lowest to highest.
8. Why don't programmers write applications in machine language?
9. Machine language is used at which level of the virtual machine shown in Figure 1-1?
10. Statements at the assembly language level of a virtual machine are translated into statements at which other level?

1.3 Data Representation

Assembly language programmers deal with data at the physical level, so they must be adept at examining memory and registers. Often, binary numbers are used to describe the contents of computer memory; at other times, decimal and hexadecimal numbers are used. You must develop