Credits

Executive Editor
Chris Webb

Development Editor
Adaobi Obi Tulton

Production Editor
William A. Barton

Technical Editor
Paul Carter

Copy Editor
Luann Rouff

Editorial Manager
Kathryn Malm Bourgoine

Vice President & Executive Group Publisher
Richard Swadley

Vice President and Publisher
Joseph B. Wikert

Project Coordinator
Erin Smith

Graphics and Production Specialists
Jonelle Burns
Amanda Carter
Carrie A. Foster
Lauren Goddard
Denny Hager
Joyce Haughey

Quality Control Technicians
David Faust
Susan Moritz
Carl William Pierce

Media Development Specialist
Angie Denny

Proofreading
TECHBOOKS Production Services

Indexing
Richard T. Evans

This book is dedicated to my wife, Barbara, and my daughters, Katie Jane and Jessica. “Trust in the Lord with all your heart and lean not on your own understanding; in all ways acknowledge him, and he will make your paths straight.” Pr 3:5-6 (NIV)

Acknowledgments

First, all honor, glory, and praise go to God, who through His Son makes all things possible and gives us the gift of eternal life.

Many thanks go to the great team of people at John Wiley & Sons Publishing. Thanks to Chris Webb, the acquisitions editor, for offering me the opportunity to write this book. I am forever indebted to Adaobi Obi Tulton, the development editor, for her work in making this book presentable and her overall guidance through the book writing process. Also, many thanks go to Paul Carter, the technical editor of the book. Paul's comments throughout the book were invaluable in presenting the topic in the best way and for pointing out my goofs and blunders. I would also like to thank Carole McClendon at Waterside Productions, Inc., for arranging this opportunity for me, and for helping out in my writing career.

Finally, I would like to thank my parents, Mike and Joyce Blum, for their dedication and support while raising me, and to my wife, Barbara, and daughters, Katie Jane and Jessica, for their love, patience, and understanding, especially while I was writing this book.

Contents

Acknowledgments xi
Contents xiii
Introduction xxiii

Chapter 1: What Is Assembly Language? 1

Processor Instructions 1
Instruction code handling 2
Instruction code format 3
High-Level Languages 6
Types of high-level languages 7
High-level language features 9
Assembly Language 10
Opcode mnemonics 11
Defining data 12
Directives 14
Summary 15

Chapter 2: The IA-32 Platform 17

Core Parts of an IA-32 Processor 17
Control unit 19
Execution unit 24
Registers 25
Flags 29
Advanced IA-32 Features 32
The x87 floating-point unit 32
Multimedia extensions (MMX) 33
Streaming SIMD extensions (SSE) 33
Hyperthreading 34
The IA-32 Processor Family 34
Intel processors 35
Non-Intel processors 36
Summary 37

Chapter 3: The Tools of the Trade **39**

The Development Tools **39**
The Assembler 40
The Linker 42
The Debugger 43
The Compiler 44
The object code disassembler 44
The Profiler 44
The GNU Assembler **45**
Installing the assembler 45
Using the assembler 47
A word about opcode syntax 49
The GNU Linker **50**
The GNU Compiler **53**
Downloading and installing gcc 53
Using gcc 54
The GNU Debugger Program **56**
Downloading and installing gdb 56
Using gdb 57
The KDE Debugger **60**
Downloading and installing kdbg 60
Using kdbg 60
The GNU Objdump Program **62**
Using objdump 63
An objdump example 64
The GNU Profiler Program **65**
Using the profiler 65
A profile example 68
A Complete Assembly Development System **69**
The basics of Linux 69
Downloading and running MEPIS 70
Your new development system 71
Summary **72**

Chapter 4: A Sample Assembly Language Program **73**

The Parts of a Program **73**
Defining sections 74
Defining the starting point 74
Creating a Simple Program **75**
The CPUID instruction 76
The sample program 77

Building the executable 80
Running the executable 80
Assembling using a compiler 80
Debugging the Program 81
Using gdb 81
Using C Library Functions in Assembly 86
Using printf 87
Linking with C library functions 88
Summary 90

Chapter 5: Moving Data 91

Defining Data Elements 91
The data section 91
Defining static symbols 94
The bss section 95
Moving Data Elements 97
The MOV instruction formats 97
Moving immediate data to registers and memory 98
Moving data between registers 99
Moving data between memory and registers 99
Conditional Move Instructions 106
The CMOV instructions 107
Using CMOV instructions 109
Exchanging Data 110
The data exchange instructions 111
Using the data exchange instruction 116
The Stack 119
How the stack works 119
PUSHing and POPing data 120
PUSHing and POPing all the registers 123
Manually using the ESP and EBP registers 123
Optimizing Memory Access 123
Summary 124

Chapter 6: Controlling Execution Flow 127

The Instruction Pointer 127
Unconditional Branches 129
Jumps 129
Calls 132
Interrupts 135

Conditional Branches **136**
Conditional jump instructions 136
The compare instruction 138
Examples of using the flag bits 140
Loops **144**
The loop instructions 144
A loop example 145
Preventing LOOP catastrophes 145
Duplicating High-Level Conditional Branches **146**
if statements 147
for loops 150
Optimizing Branch Instructions **153**
Branch prediction 153
Optimizing tips 155
Summary **158**

Chapter 7: Using Numbers **161**

Numeric Data Types **161**
Integers **162**
Standard integer sizes 162
Unsigned integers 164
Signed integers 166
Using signed integers 168
Extending integers 169
Defining integers in GAS 172
SIMD Integers **173**
MMX integers 173
Moving MMX integers 174
SSE integers 176
Moving SSE integers 177
Binary Coded Decimal **178**
What is BCD? 178
FPU BCD values 179
Moving BCD values 180
Floating-Point Numbers **182**
What are floating-point numbers? 182
Standard floating-point data types 184
IA-32 floating-point values 186
Defining floating-point values in GAS 187
Moving floating-point values 187
Using preset floating-point values 189

SSE floating-point data types 190
Moving SSE floating-point values 191
Conversions 196
Conversion instructions 196
A conversion example 198
Summary 199

Chapter 8: Basic Math Functions 201

Integer Arithmetic 201
Addition 201
Subtraction 210
Incrementing and decrementing 215
Multiplication 216
Division 221
Shift Instructions 223
Multiply by shifting 224
Dividing by shifting 225
Rotating bits 226
Decimal Arithmetic 227
Unpacked BCD arithmetic 227
Packed BCD arithmetic 229
Logical Operations 231
Boolean logic 231
Bit testing 232
Summary 233

Chapter 9: Advanced Math Functions 235

The FPU Environment 235
The FPU register stack 236
The FPU status, control, and tag registers 237
Using the FPU stack 242
Basic Floating-Point Math 245
Advanced Floating-Point Math 249
Floating-point functions 249
Partial remainders 252
Trigonometric functions 254
Logarithmic functions 257
Floating-Point Conditional Branches 259
The FCOM instruction family 260
The FCOMI instruction family 262
The FCMOV instruction family 263

Saving and Restoring the FPU State **265**
Saving and restoring the FPU environment 265
Saving and restoring the FPU state 266
Waiting versus Nonwaiting Instructions **269**
Optimizing Floating-Point Calculations **270**
Summary **270**

Chapter 10: Working with Strings **273**

Moving Strings **273**
The MOVS instruction 274
The REP prefix 278
Other REP instructions 283
Storing and Loading Strings **283**
The LODS instruction 283
The STOS instruction 284
Building your own string functions 285
Comparing Strings **286**
The CMPS instruction 286
Using REP with CMPS 288
String inequality 289
Scanning Strings **291**
The SCAS instruction 292
Scanning for multiple characters 293
Finding a string length 295
Summary **296**

Chapter 11: Using Functions **297**

Defining Functions **297**
Assembly Functions **299**
Writing functions 299
Accessing functions 302
Function placement 304
Using registers 304
Using global data 304
Passing Data Values in C Style **306**
Revisiting the stack 306
Passing function parameters on the stack 306
Function prologue and epilogue 308
Defining local function data 309

Cleaning out the stack 312
An example 312
Watching the stack in action 314
Using Separate Function Files 317
Creating a separate function file 317
Creating the executable file 318
Debugging separate function files 319
Using Command-Line Parameters 320
The anatomy of a program 320
Analyzing the stack 321
Viewing command-line parameters 323
Viewing environment variables 325
An example using command-line parameters 326
Summary 328

Chapter 12: Using Linux System Calls 329

The Linux Kernel 329
Parts of the kernel 330
Linux kernel version 336
System Calls 337
Finding system calls 337
Finding system call definitions 338
Common system calls 339
Using System Calls 341
The system call format 341
Advanced System Call Return Values 346
The sysinfo system call 346
Using the return structure 347
Viewing the results 348
Tracing System Calls 349
The strace program 349
Advanced strace parameters 350
Watching program system calls 351
Attaching to a running program 353
System Calls versus C Libraries 355
The C libraries 356
Tracing C functions 357
Comparing system calls and C libraries 358
Summary 359

Chapter 13: Using Inline Assembly 361

What Is Inline Assembly? 361
Basic Inline Assembly Code 365
The asm format 365
Using global C variables 367
Using the volatile modifier 369
Using an alternate keyword 369
Extended ASM 370
Extended ASM format 370
Specifying input and output values 370
Using registers 372
Using placeholders 373
Referencing placeholders 376
Alternative placeholders 377
Changed registers list 377
Using memory locations 379
Using floating-point values 380
Handling jumps 382
Using Inline Assembly Code 384
What are macros? 384
C macro functions 384
Creating inline assembly macro functions 386
Summary 387

Chapter 14: Calling Assembly Libraries 389

Creating Assembly Functions 389
Compiling the C and Assembly Programs 391
Compiling assembly source code files 392
Using assembly object code files 392
The executable file 393
Using Assembly Functions in C Programs 395
Using integer return values 396
Using string return values 397
Using floating-point return values 400
Using multiple input values 401
Using mixed data type input values 403
Using Assembly Functions in C++ Programs 407
Creating Static Libraries 408
What is a static library? 408
The ar command 409

Creating a static library file 410
Compiling with static libraries 412
Using Shared Libraries 412
What are shared libraries? 412
Creating a shared library 414
Compiling with a shared library 414
Running programs that use shared libraries 415
Debugging Assembly Functions 417
Debugging C programs 417
Debugging assembly functions 418
Summary 420

Chapter 15: Optimizing Routines 421

Optimized Compiler Code 421
Compiler optimization level 1 422
Compiler optimization level 2 423
Compiler optimization level 3 425
Creating Optimized Code 425
Generating the assembly language code 425
Viewing optimized code 429
Recompiling the optimized code 429
Optimization Tricks 430
Optimizing calculations 430
Optimizing variables 433
Optimizing loops 437
Optimizing conditional branches 442
Common subexpression elimination 447
Summary 450

Chapter 16: Using Files 453

The File-Handling Sequence 453
Opening and Closing Files 454
Access types 455
UNIX permissions 456
Open file code 458
Open error return codes 459
Closing files 460
Writing to Files 460
A simple write example 460
Changing file access modes 462
Handling file errors 462

Reading Files **463**
A simple read example 464
A more complicated read example 465
Reading, Processing, and Writing Data **467**
Memory-Mapped Files **470**
What are memory-mapped files? 470
The mmap system call 471
mmap assembly language format 473
An mmap example 475
Summary **479**

Chapter 17: Using Advanced IA-32 Features **481**

A Brief Review of SIMD **481**
MMX 482
SSE 483
SSE2 483
Detecting Supported SIMD Operations **483**
Detecting support 484
SIMD feature program 485
Using MMX Instructions **487**
Loading and retrieving packed integer values 487
Performing MMX operations 488
Using SSE Instructions **497**
Moving data 498
Processing data 499
Using SSE2 Instructions **504**
Moving data 505
Processing data 505
SSE3 Instructions **508**
Summary **508**

Index **511**

Introduction

Assembly language is one of the most misunderstood programming languages in use. When the term assembly language is used, it often invokes the idea of low-level bit shuffling and poring over thousand-page instruction manuals looking for the proper instruction format. With the proliferation of fancy high-level language development tools, it is not uncommon to see the phrase "assembly language programming is dead" pop up among various programming newsgroups.

However, assembly language programming is far from dead. Every high-level language program must be compiled into assembly language before it can be linked into an executable program. For the high-level language programmer, understanding how the compiler generates the assembly language code can be a great benefit, both for directly writing routines in assembly language and for understanding how the high-level language routines are converted to assembly language by the compiler.

Who This Book Is For

The primary purpose of this book is to teach high-level language programmers how their programs are converted to assembly language, and how the generated assembly language code can be tweaked. That said, the main audience for this book is programmers already familiar with a high-level language, such as C, C++, or even Java. This book does not spend much time teaching basic programming principles. It assumes that you are already familiar with the basics of computer programming, and are interested in learning assembly language to understand what is happening underneath the hood.

However, if you are new to programming and are looking at assembly language programming as a place to start, this book does not totally ignore you. It is possible to follow along in the chapters from the start to the finish and obtain a basic knowledge of how assembly language programming (and programming in general) works. Each of the topics presented includes example code that demonstrates how the assembly language instructions work. If you are completely new to programming, I recommend that you also obtain a good introductory text to programming to round out your education on the topic.

What This Book Covers

The main purpose of this book is to familiarize C and C++ programmers with assembly language, show how compilers create assembly language routines from C and C++ programs, and show how the generated assembly language routines can be spruced up to increase the performance of an application.

All high-level language programs (such as C and C++) are converted to assembly language by the compiler before being linked into an executable program. The compiler uses specific rules defined by the creator of the compiler to determine exactly how the high-level language statements are converted. Many programmers just write their high-level language programs and assume the compiler is creating the proper executable code to implement the program.

However, this is not always the case. When the compiler converts the high-level language code statements into assembly language code, quirks and oddities often pop up. In addition, the compiler is often written to follow the conversion rules so specifically that it does not recognize time-saving shortcuts that can be made in the final assembly language code, or it is unable to compensate for poorly written high-level routines. This is where knowledge of assembly language code can come in handy.

This book shows that by examining the assembly language code generated by the compiler before linking it into an executable program, you can often find places where the code can be modified to increase performance or provide additional functionality. The book also helps you understand how your high-level language routines are affected by the compiler's conversion process.

How This Book Is Structured

The book is divided into three sections. The first section covers the basics of the assembly language programming environment. Because assembly language programming differs among processors and assemblers, a common platform had to be chosen. This book uses the popular Linux operating system, running on the Intel family of processors. The Linux environment provides a wealth of program developer tools, such as an optimizing compiler, an assembler, a linker, and a debugger, all at little or no charge. This wealth of development tools in the Linux environment makes it the perfect setting for dissecting C programs into assembly language code.

The chapters in the first section are as follows:

Chapter 1, "What Is Assembly Language?" starts the section off by ensuring that you understand exactly what assembly language is and how it fits into the programming model. It debunks some of the myths of assembly language, and provides a basis for understanding how to use assembly language with high-level languages.

Chapter 2, "The IA-32 Platform," provides a brief introduction to the Intel Pentium family of processors. When working with assembly language, it is important that you understand the underlying processor and how it handles programs. While this chapter is not intended to be an in-depth analysis of the operation of the IA-32 platform, it does present the hardware and operations involved with programming for that platform.

Chapter 3, "The Tools of the Trade," presents the Linux open-source development tools that are used throughout the book. The GNU compiler, assembler, linker, and debugger are used in the book for compiling, assembling, linking, and debugging the programs.

Chapter 4, "A Sample Assembly Language Program," demonstrates how to use the GNU tools on a Linux system to create, assemble, link, and debug a simple assembly language program. It also shows how to use C library functions within assembly language programs on Linux systems to add extra features to your assembly language applications.

The second section of the book dives into the basics of assembly language programming. Before you can start to analyze the assembly language code generated by the compiler, you must understand the assembly language instructions. The chapters in this section are as follows:

Chapter 5, "Moving Data," shows how data elements are moved in assembly language programs. The concepts of registers, memory locations, and the stack are presented, and examples are shown for moving data between them.

Chapter 6, "Controlling Execution Flow," describes the branching instructions used in assembly language programs. Possibly one of the most important features of programs, the ability to recognize branches and optimize branches is crucial to increasing the performance of an application.

Chapter 7, "Using Numbers," discusses how different number data types are used in assembly language. Being able to properly handle integers and floating-point values is important within the assembly language program.

Chapter 8, "Basic Math Functions," shows how assembly language instructions are used to perform the basic math functions such as addition, subtraction, multiplication, and division. While these are generally straightforward functions, subtle tricks can often be used to increase performance in this area.

Chapter 9, "Advanced Math Functions," discusses the IA-32 Floating Point Unit (FPU), and how it is used to handle complex floating-point arithmetic. Floating-point arithmetic is often a crucial element to data processing programs, and knowing how it works greatly benefits high-level language programmers.

Chapter 10, "Working with Strings," presents the various assembly language string-handling instructions. Character data is another important facet of high-level language programming. Understanding how the assembly language level handles strings can provide insights when working with strings in high-level languages.

Chapter 11, "Using Functions," begins the journey into the depths of assembly language programming. Creating assembly language functions to perform routines is at the core of assembly language optimization. It is good to know the basics of assembly language functions, as they are often used by the compiler when generating the assembly language code from high-level language code.

Chapter 12, "Using Linux System Calls," completes this section by showing how many high-level functions can be performed in assembly language using already created functions. The Linux system provides many high-level functions, such as writing to the display. Often, you can utilize these functions within your assembly language program.

The last section of the book presents more advanced assembly language topics. Because the main topic of this book is how to incorporate assembly language routines in your C or C++ code, the first few chapters show just how this is done. The remaining chapters present some more advanced topics to round out your education on assembly language programming. The chapters in this section include the following:

Chapter 13, "Using Inline Assembly," shows how to incorporate assembly language routines directly in your C or C++ language programs. Inline assembly language is often used for "hard-coding" quick routines in the C program to ensure that the compiler generates the appropriate assembly language code for the routine.

Chapter 14, "Calling Assembly Libraries," demonstrates how assembly language functions can be combined into libraries that can be used in multiple applications (both assembly language and high-level language). It is a great time-saving feature to be able to combine frequently used functions into a single library that can be called by C or C++ programs.

Chapter 15, "Optimizing Routines," discusses the heart of this book: modifying compiler-generated assembly language code to your taste. This chapter shows exactly how different types of C routines (such as if-then statements and for-next loops) are produced in assembly language code. Once you understand what the assembly language code is doing, you can add your own touches to it to customize the code for your specific environment.

Chapter 16, "Using Files," covers one of the most overlooked functions of assembly language programming. Almost every application requires some type of file access on the system. Assembly language programs are no different. This chapter shows how to use the Linux file-handling system calls to read, write, and modify data in files on the system.

Chapter 17, "Using Advanced IA-32 Features," completes the book with a look at the advanced Intel Single Instruction Multiple Data (SIMD) technology. This technology provides a platform for programmers to perform multiple arithmetic operations in a single instruction. This technology has become crucial in the world of audio and video data processing.

What You Need to Use This Book

All of the examples in this book are coded to be assembled and run on the Linux operating system, running on an Intel processor platform. The Open Source GNU compiler (`gcc`), assembler (`gas`), linker (`ld`), and debugger (`gdb`) are used extensively throughout the book to demonstrate the assembly language features. Chapter 4, "A Sample Assembly Language Program," discusses specifically how to use these tools on a Linux platform to create, assemble, link, and debug an assembly language program. If you do not have an installed Linux platform available, Chapter 4 demonstrates how to use a Linux distribution that can be booted directly from CD, without modifying the workstation hard drive. All of the GNU development tools used in this book are available without installing Linux on the workstation.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

Tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

As for styles in the text:

- We *highlight* important words when we introduce them.
- We show filenames, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

```
In code examples we highlight new and important code with a gray background.

The gray highlighting is not used for code that's less important in the present
context, or has been shown before.
```

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at `www.wrox.com`. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

> *Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 0-764-57901-0.*

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at `www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to `www.wrox.com` and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page, you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list, including links to each book's errata, is also available at `www.wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form there to send us the error you have found. We'll check the information, and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a Web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com` you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

What Is Assembly Language?

One of the first hurdles to learning assembly language programming is understanding just what assembly language is. Unlike other programming languages, there is no one standard format that all assemblers use. Different assemblers use different syntax for writing program statements. Many beginning assembly language programmers get caught up in trying to figure out the myriad of different possibilities in assembly language programming.

The first step in learning assembly language programming is defining just what type of assembly language programming you want to (or need to) use in your environment. Once you define your flavor of assembly language, it is easy to get started learning and using assembly language in both standalone and high-level language programs.

This chapter begins the journey by showing where assembly language comes from, and defining why assembly language programming is used. To understand assembly language programming, you must first understand the basics of its underlying purpose — programming in processor instruction code. Next, the chapter shows how high-level languages are converted to raw instruction code by compilers and linkers. After having that information, it will be easier for you to understand how assembly language programs and high-level language programs differ, and how they can both be used to complement one another.

Processor Instructions

At the lowest layer of operation, all computer processors (microcomputers, minicomputers, and mainframe computers) manipulate data based on binary codes defined internally in the processor chip by the manufacturer. These codes define what functions the processor should perform, utilizing the data provided by the programmer. These preset codes are referred to as *instruction codes*. Different types of processors contain different types of instruction codes. Processor chips are often categorized by the quantity and type of instruction codes they support.

While the different types of processors can contain different types of instruction codes, they all handle instruction code programs similarly. This section describes how processors handle instructions and what the instruction codes look like for a sample processor chip.

Instruction code handling

As a computer processor chip runs, it reads instruction codes that are stored in memory. Each instruction code set can contain one or more bytes of information that instruct the processor to perform a specific task. As each instruction code is read from memory, any data required for the instruction code is also stored and read in memory. The memory bytes that contain the instruction codes are no different than the bytes that contain the data used by the processor.

To differentiate between data and instruction codes, special *pointers* are used to help the processor keep track of where in memory the data and instruction codes are stored. This is shown in Figure 1-1.

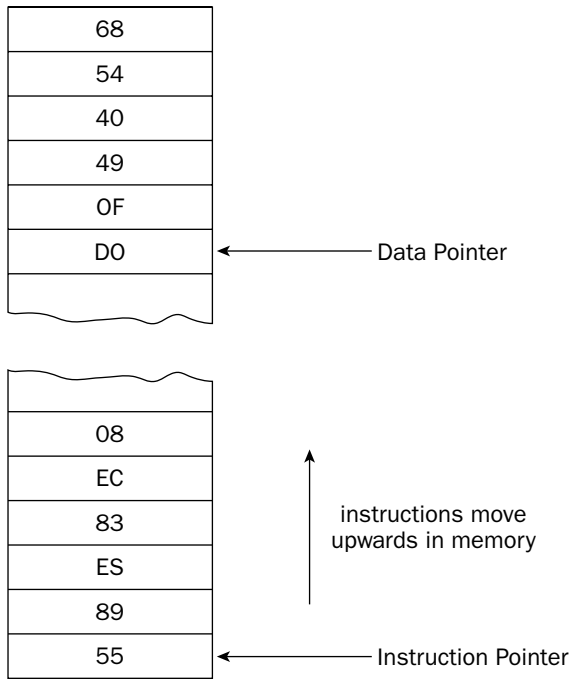

Figure 1-1

The *instruction pointer* is used to help the processor keep track of which instruction codes have already been processed and what code is next in line to be processed. Of course, there are special instruction codes that can change the location of the instruction pointer, such as jumping to a specific location in the program.

Similarly, a *data pointer* is used to help the processor keep track of where the data area in memory starts. This area is called the *stack*. As new data elements are placed in the stack, the pointer moves "down" in memory. As data is read from the stack, the pointer moves "up" in memory.

Each instruction code can contain one or more bytes of information for the processor to handle. For example, the instruction code bytes (in hexadecimal format)

```
C7 45 FC 01 00 00 00
```

tell an Intel IA-32 series processor to load the decimal value 1 into a memory offset location defined by a processor register. The instruction code contains several pieces of information (defined later in the "Opcode" section) that clearly define what function is to be performed by the processor. After the processor completes processing one instruction code set, it reads the next one in memory (as pointed to by the instruction pointer). The instructions must be placed in memory in the proper format and order for the processor to properly step through the program code.

Every instruction must contain at least 1 byte called the *operation code* (or *opcode* for short). The opcode defines what function the processor should perform. Each processor family has its own predefined opcodes that define all of the functions available. The next section shows how the opcodes used in the Intel IA-32 family of microprocessors are structured. These are the types of processor opcodes that are used in all of the examples in this book.

Instruction code format

The Intel IA-32 family of microprocessors includes all of the current types of microprocessors used in modern IBM-platform microcomputers (see Chapter 2, "The IA-32 Platform"), including the popular Pentium line of microprocessors. A specific format for instruction codes is used in the IA-32 family of microprocessors, and understanding the format of these instructions will help you in your assembly language programming. The IA-32 instruction code format consists of four main parts:

- Optional instruction prefix
- Operational code (opcode)
- Optional modifier
- Optional data element

Figure 1-2 shows the layout of the IA-32 instruction code format.

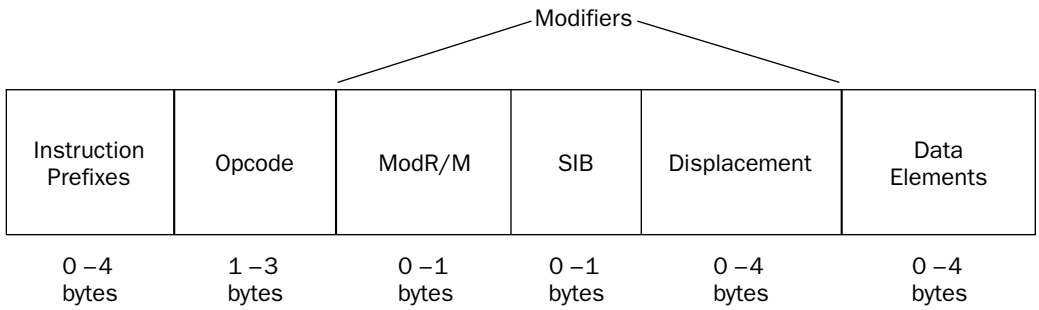

Figure 1-2

Each of the parts is used to completely define a specific instruction for the processor to perform. The following sections describe each of the four parts of the instruction code and how they define the instruction performed by the processor.

> *The Intel Pentium processor family is not the only set of processor chips to utilize the IA-32 instruction code format. The AMD corporation also produces a line of chips that are fully compatible with the Intel IA-32 instruction code format.*

Opcode

As shown in Figure 1-2, the only required part of the IA-32 instruction code format is the opcode. Each instruction code must include an opcode that defines the basic function or task to be performed by the processor.

The opcode is between 1 and 3 bytes in length, and uniquely defines the function that is performed. For example, the 2-byte opcode `OF A2` defines the IA-32 CPUID instruction. When the processor executes this instruction code, it returns specific information about the microprocessor in different registers. The programmer can then use additional instruction codes to extract the information from the processor registers to determine the type and model of microprocessor on which the program is running.

> *Registers are components within the processor chip that are used to temporarily store data while being handled by the processor. They are covered in more detail in Chapter 2, "The IA-32 Platform."*

Instruction prefix

The instruction prefix can contain between one and four 1-byte prefixes that modify the opcode behavior. These prefixes are categorized into four different groups, based on the prefix function. Only one prefix from each group can be used at one time to modify the opcode (thus the maximum of four prefix bytes). The four prefix groups are as follows:

- Lock and repeat prefixes
- Segment override and branch hint prefixes
- Operand size override prefix
- Address size override prefix

The lock prefix indicates that any shared memory areas will be used exclusively by the instruction. This is important for multiprocessor and hyperthreaded systems. The repeat prefixes are used to indicate a repeating function (usually used when handling strings).

The segment override prefixes define instructions that can override the defined segment register value (described in more detail in Chapter 2). The branch hint prefixes attempt to give the processor a clue as to the most likely path the program will take in a conditional jump statement (this is used with predictive branching hardware).

The operand size override prefix informs the processor that the program will switch between 16-bit and 32-bit operand sizes within the instruction code. This enables the program to warn the processor when it uses larger-sized operands, helping to speed up the assignment of data to registers.

The address size override prefix informs the processor that the program will switch between 16-bit and 32-bit memory addresses. Either size can be declared as the default size for the program, and this prefix informs the processor that the program is switching to the other.

Modifiers

Some opcodes require additional modifiers to define what registers or memory locations are involved in the function. The modifiers are contained in three separate values:

- addressing-form specifier (ModR/M) byte
- Scale-Index-Base (SIB) byte
- One, two, or four address displacement bytes

The ModR/M byte

The ModR/M byte consists of three fields of information, as shown in Figure 1-3.

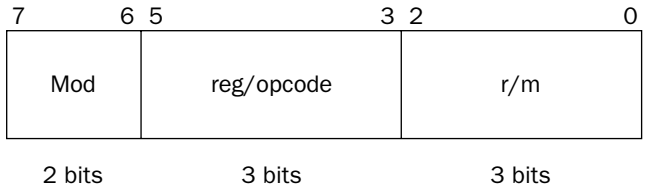

Figure 1-3

The mod field is used with the r/m field to define the register or addressing mode used in the instruction. There are 24 possible addressing modes, along with eight possible general-purpose registers that can be used in the instruction, making 32 possible values.

The reg/opcode field is used to enable three more bits to further define the opcode function (such as opcode subfunctions), or it can be used to define a register value.

The r/m field is used to define another register to use as the operand of the function, or it can be combined with the mod field to define the addressing mode for the instruction.

The SIB byte

The SIB byte also consists of three fields of information, as shown in Figure 1-4.

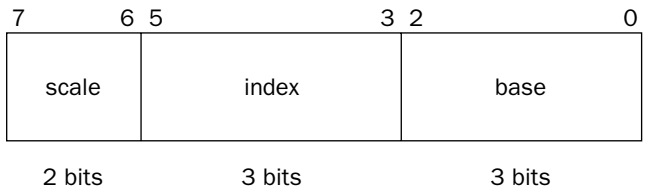

Figure 1-4

The scale field specifies the scale factor for the operation. The index field specifies the register that is used as the index register for memory access. The base field specifies the register that is used as the base register for memory access.

The combination of the ModR/M and SIB bytes creates a table that can define many possible combinations of registers and memory modes for accessing data. The Intel specification sheets for the Pentium processor define all of the possible combinations that are used with the ModR/M and SIB bytes.

The address displacement byte

The address displacement byte is used to indicate an offset to the memory location defined in the ModR/M and SIB bytes. This can be used as an index to a base memory location to either store or access data within memory.

Data element

The final part of the instruction code is the data element that is used by the function. While some instruction codes read data from memory locations or processor registers, some include data within the instruction code itself. Often this value is used to represent a static numeric value, such as a number to be added, or a memory location. This value can contain 1, 2, or 4 bytes of information, depending on the data size.

For example, the following sample instruction code shown earlier:

```
C7 45 FC 01 00 00 00
```

defines the opcode C7, which is the instruction to move a value to a memory location. The memory location is defined by the 45 FC modifier (which defines –4 bytes (the FC value) from the memory location pointed to by the value in the EBP register (the 45 value). The final 4 bytes define the integer value that is placed in that memory location (in this case, the value 1).

As you can see from this example, the value 1 was written as the 4-byte hexadecimal value 01 00 00 00. The order of the bytes in the data stream depends on the type of processor used. The IA-32 platform processors use "little-endian" notation, whereby the lower-value bytes appear first in order (when reading left to right). Other processors use "big-endian" order, whereby the higher-value bytes appear first in order. This concept is extremely important when specifying data and memory location values in your assembly language programs.

High-Level Languages

If it looks like programming in pure processor instruction code is difficult, it is. Even the simplest of programs require the programmer to specify a lot of opcodes and data bytes. Trying to manage a huge program full of just instruction codes would be a daunting task. To help save the sanity of programmers, high-level languages (HLLs) were created.

HLLs enable programmers to create functions using simpler terms, rather than raw processor instruction codes. Special reserved keywords are used to define variables (memory locations for data), create loops (jump over instruction codes), and handle input and output from the program. However, the processor does not have any knowledge about how to handle the HLL code. The code must be converted by some mechanism to simple instruction code format for the processor to handle. This section defines the

different types of HLLs and then shows how the HLL code is converted to the instruction code for the processor to execute.

Types of high-level languages

While programmers can choose from many different HLLs available, they all can be classified into two different categories, based on how they are run on the computer:

- Compiled languages
- Interpreted languages

While it is possible for different implementations of the same programming language to be either compiled or interpreted, these categories are used to show how a particular HLL implementation defines how the programs are run on the processor. The following sections describe the methods used to run programs and show how they affect how the processor operates with them.

Compiled languages

Most production applications are created using compiled HLLs. The programmer creates a program using common statements for the language which carry out the logic of the application. The text program statements are then converted into a set of instruction codes that can be run on the processor. Usually, what is commonly called *compiling* a program is actually a two-step process:

- Compiling the HLL statements into raw instruction codes
- Linking the raw instruction codes to produce an executable program

Figure 1-5 demonstrates this process.

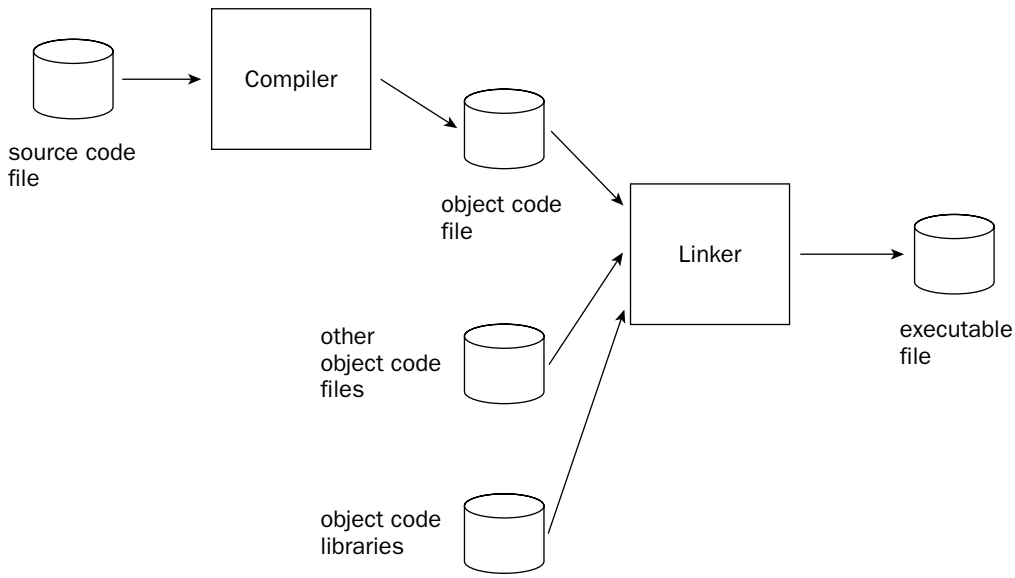

Figure 1-5

The compiling step converts the text programming language statements into the instruction codes required to carry out the application function. Each of the HLL lines of code are matched up with one or more instruction codes pertaining to the specific processor on which the application will run. For example, the simple HLL code

```
int main()
{
    int i = 1;
    exit(0);
}
```

is compiled into the following IA-32 instruction codes:

```
55
89 E5
83 EC 08
C7 45 FC 01 00 00 00
83 EC 0C
6A 00
E8 D1 FE FF FF
```

This step produces an intermediate file, called an *object code file*. The object code file contains the instruction codes that represent the core of the application functions, as shown above. The object code file itself cannot be run by the operating system. Often the host operating system requires special file formats for executable files (program files that can be run on the system), and the HLL program may require program functions from other object files. Another step is required to add these components.

After the code is compiled into an object file, a *linker* is used to link the application object code file with any additional object files required by the application and to create the final executable output file. The output of the linker is an executable file that can only be run on the operating system for which the program is written. Unfortunately, each operating system uses a different format for executable files, so an application compiled on a Microsoft Windows workstation will not work as is on a Linux workstation, and vice versa.

> *Object files that contain commonly used functions can be combined into a single file, called a library file. The library file can then be linked into multiple applications either at compile time (called static libraries), or at the time the application is run on the system (called dynamic libraries).*

Interpreted languages

As opposed to compiled programs, which run by themselves on a processor, an interpreted language program is read and run by a separate program. The separate program is a host for the application program, reading and interpreting the program as it is processed. It is the job of the host program to convert the interpreted program code into the proper instruction codes for the processor as the program is running.

Obviously, the downside to using interpreted languages is speed. Instead of the program being compiled directly to instruction codes that are run on the processor, an intermediary program reads each line of program code and processes the required functions. The amount of time the host program takes to read the code and execute it adds additional delays to the execution of the application.

With the resulting reduction in speed when using interpreted languages, you may be wondering why anyone still uses them. One answer is convenience. With compiled programs, every time a change is made to the program, the program must be recompiled and relinked with the proper code libraries. With interpreted programs, changes can be quickly made to the source code file and the program rerun to check for errors. In addition, with interpreted languages, the interpreter application automatically determines what functions need to be included with the core code to support functions.

> *Today's programming language environment muddies the waters between compiled and interpreted languages. No one specific language can be classified in either category. Instead, individual implementations of different HLLs are categorized. For example, while many BASIC programming implementations require interpreters to interpret the BASIC code into an executable program, there are many BASIC implementations that enable the programmer to compile the BASIC programs into executable instruction code.*

Hybrid languages

Hybrid languages are a recent trend in programming that combine the features of a compiled program with the versatility and ease of an interpreted program. A perfect example is the popular Java programming language.

The Java programming language is compiled into what is called *byte code*. The byte code is similar to the instruction code you would see on a processor, but is itself not compatible with any current processor family (although there have been plans to create a processor that can run Java byte code as instruction sets).

Instead, the Java byte code must be interpreted by a Java Virtual Machine (JVM), running separately on the host computer. The Java byte code is portable, in that it can be run by any JVM on any type of host computer. The advantage is that different platforms can have their own specific JVMs, which are used to interpret the same Java byte code without it having to be recompiled from the original source code.

High-level language features

If you are a professional programmer, most likely you do most (if not all) of your coding using a high-level language. You may or may not have had the luxury of choosing which HLL you use for your projects, but either way, there is no doubt that it makes your job easier. This section describes two of the most useful features of HLLs, portability and standardization, which help set HLLs apart from assembly language programming.

Portability

As described earlier in the "Processor Instructions" section, instruction code programming is highly dependent on the processor used in the computer. Each of the different families of processors utilize different instruction code formats, as well as different methods for storing data (big endian vs. little endian). Instruction codes written for an IA-32 platform will not work on a MIPS processor platform.

Imagine writing a 10,000-line instruction code program for your new application, which runs on a Sun Sparc workstation, and then being asked to port it to a Linux workstation running on a Pentium computer. Because the microprocessor used for the Sun Sparc workstation does not use the same instruction codes as the Pentium, all of your code would need to be redone for the new instruction codes — ouch.

HLLs have the capability to be ported to other operating systems and other processor platforms by simply recompiling the program on the new platform. When the program is recompiled, it is automatically rewritten using the instruction codes for the destination processor.

> *However, in practice, nontrivial programs use operating system APIs that make it difficult to simply recompile the source code for another platform. For example, a program directly using the MS Windows API will not compile under Linux.*

Standardization

Another useful feature of HLLs is the abundance of standards available for the languages. Both the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) have created standard specifications for many different HLLs.

This means that you are guaranteed to obtain the same results from source code compiled with a standard compiler on one type of operating system and processor as you would compiling on a different type of operating system and processor. Each compiler is created to interpret the standard language constructs into instruction code for the destination processor to produce the same functionality across the processor platforms.

Assembly Language

While creating large applications using an HLL is often simpler than using raw instruction codes, it doesn't necessarily mean that the resulting program will be efficient. Unfortunately, in order to increase portability and comply with standards, many compilers code to the "least common denominator." This means that compilers creating instruction codes for advanced processor chips may not utilize special instruction codes unique to those processors to help create faster applications.

One feature that many of the new processors on the market offer is advanced mathematics handling instruction codes. These instruction codes help speed up complex mathematical expression processing by using larger-than-normal byte sizes to represent numbers (either 64 or 128 bits). Unfortunately, many compilers don't take advantage of these advanced instruction codes. Fortunately, there is a simple solution for the programmer. In environments where execution speed is critical, assembly language programming can come to the rescue. Of course, the first step to improving execution speed is to ensure that the best algorithm is used in the first place. Optimizing a poor algorithm does not compensate for using a fast algorithm in the first place.

Assembly language enables programmers to directly create instruction code programs without having to worry about the many different instruction code set combinations on the processor. Instead, an assembly language program uses *mnemonics* to represent instruction codes. The mnemonics enables the programmer to use English-style words to represent individual instruction codes. The assembly language mnemonics are easily converted to the raw instruction codes by an assembler.

This section describes the assembly language mnemonic system, and how it is used to create raw instruction code programs that can be run on the processor.

An assembly language program consists of three components that are used to define the program operations:

- Opcode mnemonics
- Data sections
- Directives

The following sections describe each of these components and show how they are used within the assembly language program to create the resulting instruction code program.

Opcode mnemonics

The core of an assembly language program is the instruction codes used to create the program. To help facilitate writing the instruction codes, assemblers equate mnemonic words with instruction code functions, such as moving or adding data elements. For example, the instruction code sample

```
55
89 E5
83 EC 08
C7 45 FC 01 00 00 00
83 EC 0C
6A 00
E8 D1 FE FF FF
```

can be written in assembly language as follows:

```
push %ebp
mov  %esp, %ebp
sub  $0x8, %esp
movl $0x1, -4(%ebp)
sub  $0xc, %esp
push $0x0
call 8048348
```

Instead of having to know what each byte of instruction code represents, the assembly language programmer can use easier-to-remember mnemonic codes, such as `push`, `mov`, `sub`, and `call`, to represent the instruction codes.

Different assemblers use different mnemonics to represent instruction codes. While trends have emerged to standardize assembler mnemonics, there is still quite a vast variety of mnemonic codes, not only between processor families but even between assemblers used for the same processor instruction code sets.

> *Each processor manufacturer publishes developer manuals detailing all of the instruction codes implemented by a specific chip set. The Intel IA-32 developer manuals are freely available at the Intel Web site (www.intel.com). These developer manuals take over 1,000 pages just to enumerate and describe all of the instruction codes for the Pentium family of processors.*

Defining data

Besides the instruction codes, most programs also require data elements to be used to hold variable and constant data values that are used throughout the program. HLLs use variables to define sections of memory to hold data. For example, it is not uncommon to see the following in an HLL program:

```
long testvalue = 150;
char message[22] = {"This is a test message"};
float pi = 3.14159;
```

Each of these statements is interpreted by the HLL compiler to reserve memory locations of a specific number of bytes to store values that may or may not change during the course of the program. Each time the program references the variable name (such as `testvalue`), the compiler knows to access the specified location in memory to read or change the byte values.

Assembly language also enables the programmer to define data items that will be stored in memory. One of the advantages of programming in assembly language is that it provides you with greater control over where and how your data is stored in memory. The following sections describe two methods used to store and retrieve data in assembly language.

Using memory locations

Similar to the HLL method of defining data, assembly language enables you to declare a variable that points to a specific location in memory. Defining variables in assembly language consists of two parts:

1. A label that points to a memory location
2. A data type and default value for the memory bytes

The data type determines how many bytes are reserved for the variable. In an assembly language program, this would look like the following:

```
testvalue:
   .long 150
message:
    .ascii "This is a test message"
pi:
   .float 3.14159
```

As you can see from the data types, assembly language allows you to declare the type of data stored in the memory location, along with the default values placed in the memory location, similar to most HLL methods. Each data type occupies a specific number of bytes, starting at the memory location reserved for the label. This is shown in Figure 1-6.

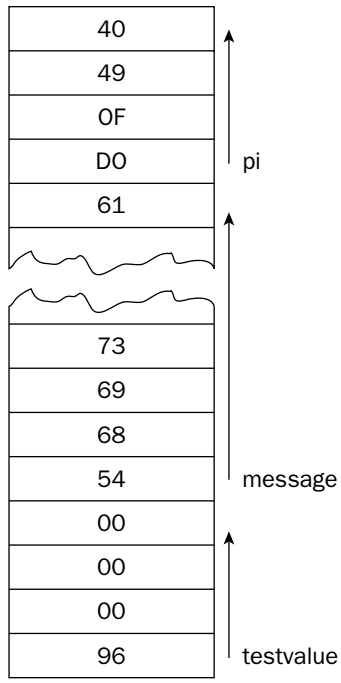

Figure 1-6

The first data element declared, `testvalue`, is placed in memory as a 4-byte hexadecimal value in little-endian order (96 00 00 00). The next data element, `message`, is placed immediately after the last byte of the `testvalue` data element. Because the `message` data element is a text value, it is placed in memory in the order the text characters appear in the string. Finally, the last data element, `pi`, is placed in memory immediately after the last byte of the `message` data element (the floating point is discussed in great detail in Chapter 7, "Using Numbers."

The memory locations are referenced within the assembly language program based on the label used to define the starting location. A sample assembly language program would look like the following:

```
movl testvalue, %ebx
addl $10, %ebx
movl %ebx, testvalue
```

The first instruction loads the EBX register with the 4-byte value located at the memory location pointed to by the `testvalue` label (which was defined with a value of 150). The next instruction adds 10 (in decimal) to the value stored in the EBX register and puts the result back in the EBX register. Finally, the register value is stored in the memory location referenced by the `testvalue` label. This new value can then be referenced again in the program using the `testvalue` label, and it will have the value of 160 (this process is explained in detail in Chapter 5, "Moving Data," and Chapter 8, "Basic Math Functions").

Using the stack

Another method used to store and retrieve data in assembly language is called the *stack*. The stack is a special memory area usually reserved for passing data elements between functions in the program. It can also be used for temporarily storing and retrieving data elements.

The stack is a region of memory reserved at the end of the memory range that the computer reserves for the application. A pointer (called the *stack pointer*) is used to point to the next memory location in the stack to put or take data. Much like a stack of papers, when a data element is placed in the stack, it becomes the first item that can be removed from the stack (assuming you can only take papers off of the top of the paper stack).

When calling functions in an assembly language program, you usually place any data elements that you want passed to the function on the top of the stack. When the function is called, it can retrieve the data elements from the stack.

> *The different methods of storing and retrieving data are discussed in greater detail in Chapter 5, "Moving Data."*

Directives

Instructions and data are not the only elements that make up an assembly language program. Assemblers reserve special keywords for instructing the assembler how to perform special functions as the mnemonics are converted to instruction codes.

You saw an example of directives in the previous section when the data elements were defined. The data types were declared using assembler directives used in the GNU assembler. The `.long`, `.ascii`, and `.float` directives are used to alert the assembler that a specific type of data is being declared. As shown in the example, directives are preceded by a period to set them apart from labels.

Directives are another area in which the different assemblers vary. Many different directives are used to help make the programmer's job of creating instruction codes easier. Some modern assemblers have lists of directives that can rival many HLL features, such as while loops, and if-then statements! The older, more traditional assemblers, however, keep the directives to a minimum, forcing the assembly language programmer to use the mnemonic codes to create the program logic.

One of the most important directives used in the assembly language program is the `.section` directive. This directive defines the section of memory in which the assembly language program is defining elements. All assembly language programs have at least three sections that must be declared:

- A data section
- A bss section
- A text section

The data section is used to declare the memory region where data elements are stored for the program. This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

The bss section is also a static memory section. It contains buffers for data to be declared later in the program. What makes this section special is that the buffer memory area is zero-filled.

The text section is the area in memory where the instruction code is stored. Again, this area is fixed, in that it contains only the instruction codes that are declared in the assembly language program.

These directives used in an assembly language program are demonstrated in Chapter 4, "A Sample Assembly Language Program."

Summary

While assembly language programming is often referred to as a single programming language category, in reality there are a wide variety of different types of assembly language assemblers. Each assembler uses slightly different formats to represent instruction codes, data, and special directives for assembling the final program. The first step to programming in assembly language is deciding which assembler you need to use, and what format it uses.

The purpose of using assembly language is to code as closely to raw processor code as possible. The code recognized by the processor is called instruction code. Each processor family has its own set of instruction codes that define the functions the processor can perform. Each processor family also uses specific formats for the instruction code. The Intel IA-32 family of processors uses a format that consists of four parts. An opcode is used to define which processor instruction should be used. An optional prefix may be used to modify the behavior of the instruction. An optional modifier may also be used to define what registers or memory locations are used in the instruction. Finally, an optional data element may be included, which defines specific data values used in the instruction.

Trying to create large-scale programs using raw instruction codes is not an easy task. Each instruction code must be programmed byte by byte in the proper order for the application to run. Instead of forcing programmers to learn all of the instruction codes, developers have created high-level languages, which enable programmers to create programs in a shorthand method, which is then converted into the proper instruction codes by a compiler. High-level languages use simple keywords and terms to define one or more instruction codes. This enables programmers to concentrate on the logic of the application program, rather than worry about the details of the underlying processor instruction codes.

The downside of using high-level languages is that the programmer is dependant on the compiler creator to convert programming logic to the instruction code run by the processor. There is no guarantee that the created instruction codes will be the most efficient method of programming the logic. For programmers who want maximum efficiency, or the capability to have greater control over how the program is handled by the processor, assembly language programming offers an alternative.

Assembly language programming enables the programmer to program with instruction codes, but by using simple mnemonic terms to refer to those instruction codes. This provides programmers with both the ease of a high-level language and the control offered by using instruction codes.

Unfortunately, assembly language assemblers are not standardized, and there are many different forms of assembly language. All assemblers contain three elements: opcode mnemonics, data elements, and

directives. The opcode mnemonics are used to code the programming logic, and data elements are used to define memory locations to hold both constant and variable data elements. Directives are one of the most controversial elements of assemblers. Directives help the programmer define specific functions, such as declaring data types, and define memory regions within the program. Some assemblers take directives to a higher level, providing directives that support many high-level language functions, such as advanced data management and logic programming.

The next chapter discusses the specific layout of the Intel IA-32 processor family. Before you can start programming for the Pentium family of processors, it is important to understand how the hardware is laid out. Knowing how the processor handles data will enable you to program more efficiently, increasing the speed of your applications.

2

The IA-32 Platform

One key to successful assembly language programming is knowing the environment you are programming for. The biggest part of that environment is the processor. Knowing the hardware platform your program will run on is crucial to being able to exploit both basic and advanced functions of the processor. Often, the whole point of using assembly language is to exploit low-level features of the processor within your application program. Knowing what elements can be used to assist your programs in gaining the most speed possible can mean the difference between a fast application and a slow application.

At the time of this writing, the most popular processor platform by far used in workstations and servers is the Intel Pentium family of processors. The hardware and instruction code set designed for the Pentium processors is commonly referred to as the IA-32 platform.

This chapter describes the hardware elements that make up the Intel IA-32 platform. The first part of the chapter describes the basic components found in the IA-32 processor platforms. Then the chapter describes the advanced features found in the newer Pentium 4 processor chips in the IA-32 family. Finally, the different processors that are contained within the IA-32 platform are discussed, showing what features to watch out for with the different types of processors, both from Intel and from other manufacturers.

Core Parts of an IA-32 Processor

While different processor families incorporate different instruction sets and capabilities, there is a core set of components that can be found on most processors. Most introductory computer science classes teach four basic components of a computer. Figure 2-1 shows a basic block diagram of these core components.

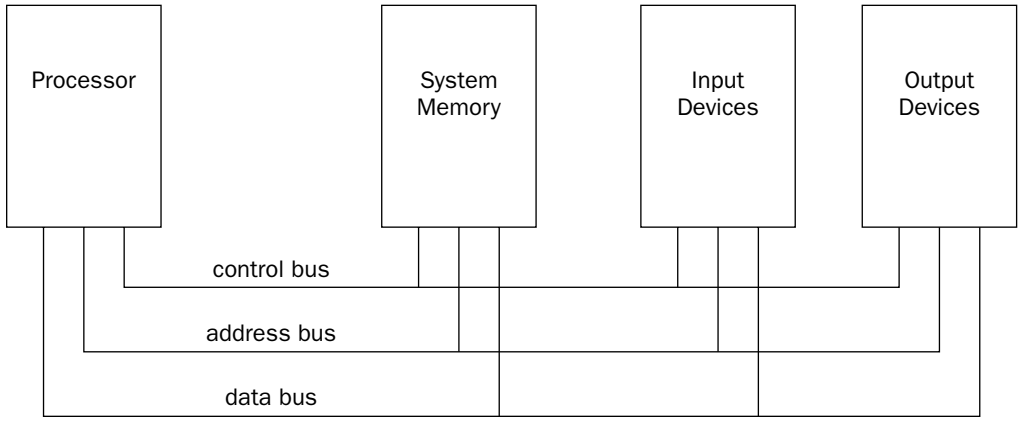

Figure 2-1

The processor contains the hardware and instruction codes that control the operation of the computer. It is connected to the other elements of the computer (the memory storage unit, input devices, and output devices) using three separate buses: a control bus, an address bus, and a data bus.

The control bus is used to synchronize the functions between the processor and the individual system elements. The data bus is used to move data between the processor and the external system elements. An example of this would be reading data from a memory location. The processor places the memory address to read on the address bus, and the memory storage unit responds by placing the value stored in that memory location on the data bus for the processor to access.

The processor itself consists of many components. Each component has a separate function in the processor's ability to process data. Assembly language programs have the ability to access and control each of these elements, so it is important to know what they are. The main components in the processor are as follows:

- Control unit
- Execution unit
- Registers
- Flags

Figure 2-2 shows these components and how they interact within the processor.

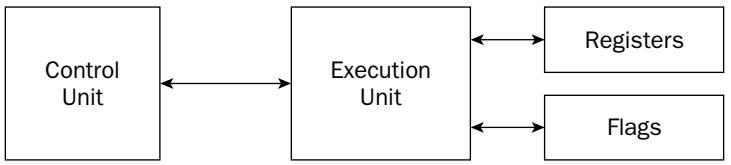

Figure 2-2

The following sections describe each of the core components, and how they are implemented in the IA-32 platform.

Control unit

At the heart of the processor is the control unit. The main purpose of the control unit is to control what is happening at any time within the processor. While the processor is running, instructions must be retrieved from memory and loaded for the processor to handle. The job of the control unit is to perform four basic functions:

1. Retrieve instructions from memory.
2. Decode instructions for operation.
3. Retrieve data from memory as needed.
4. Store the results as necessary.

The instruction counter retrieves the next instruction code from memory and prepares it to be processed. The instruction decoder is used to decode the retrieved instruction code into a micro-operation. The micro-operation is the code that controls the specific signals within the processor chip to perform the function of the instruction code.

When the prepared micro-operation is ready, the control unit passes it along to the execution unit for processing, and retrieves any results to store in an appropriate location.

The control unit is the most hotly researched part of the processor. Many advances in microprocessor technology fall within the control unit section. Intel has made numerous advancements in speeding up the operations within the control unit. One of the most beneficial advancements is the manner in which instructions are retrieved and processed by the control unit.

At the time of this writing, the latest Intel processor (the Pentium 4) uses a control unit technology called NetBurst. The NetBurst technology incorporates four separate techniques to help speed up processing in the control unit. Knowing how these techniques operate can help you optimize your assembly language programs. The NetBurst features are as follows:

- Instruction prefetch and decoding
- Branch prediction
- Out-of-order execution
- Retirement

These techniques work together to make the control unit of the Pentium 4 processor. Figure 2-3 shows how these elements interact.

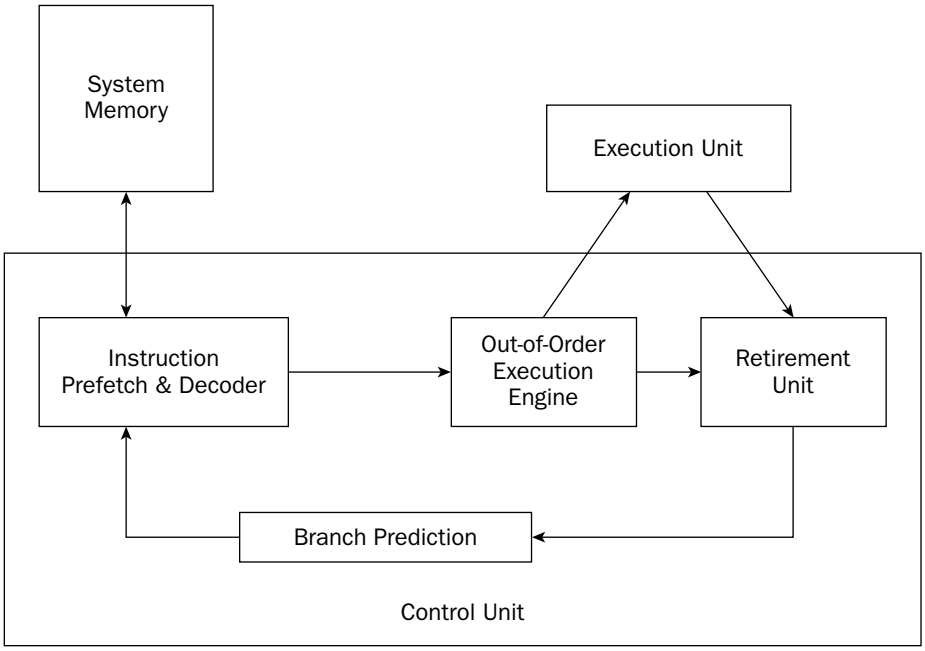

Figure 2-3

The following sections describe each of these techniques as implemented in the Pentium 4 processor.

Instruction prefetch and decoding pipeline

Older processors in the IA-32 family fetched instructions and data directly from system memory as they were needed by the execution unit. Because it takes considerably longer to retrieve data from memory than to process it, a backlog occurs, whereby the processor is continually waiting for instructions and data to be retrieved from memory. To solve this problem, the concept of *prefetching* was created.

Although the name sounds odd, prefetching involves attempting to retrieve (fetch) instructions and/or data before they are actually needed by the execution unit. To incorporate prefetching, a special storage area is needed on the processor chip itself — one that can be easily accessed by the processor, quicker than normal memory access. This was solved using *pipelining*.

Pipelining involves creating a memory cache on the processor chip from which both instructions and data elements can be retrieved and stored ahead of the time that they are required for processing. When the execution unit is ready for the next instruction, that instruction is already available in the cache and can be quickly processed. This is demonstrated in Figure 2-4.

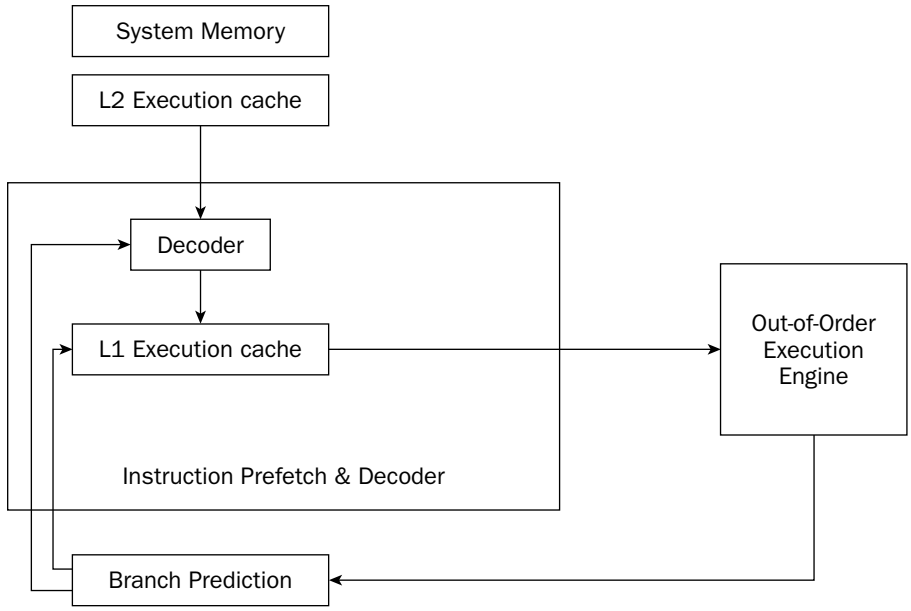

Figure 2-4

The IA-32 platform implements pipelining by utilizing two (or more) layers of cache. The first cache layer (called L1) attempts to prefetch both instruction code and data from memory as it thinks it will be needed by the processor. As the instruction pointer moves along in memory, the prefetch algorithm determines which instruction codes should be read and placed in the cache. In a similar manner, if data is being processed from memory, the prefetch algorithm attempts to determine what data elements may be accessed next and also reads them from memory and places them in cache.

Of course, one pitfall to caching instructions and data is that there is no guarantee that the program will execute instructions in a sequential order. If the program takes a logic branch that moves the instruction pointer to a completely different location in memory, the entire cache is useless and must be cleared and repopulated with instructions from the new location.

To help alleviate this problem, a second cache layer was created. The second cache layer (called L2) can also hold instruction code and data elements, separate from the first cache layer. When the program logic jumps to a completely different area in memory to execute instructions, the second layer cache can still hold instructions from the previous instruction location. If the program logic jumps back to the area, those instructions are still being cached and can be processed almost as quickly as instructions stored in the first layer cache.

While assembly language programs cannot access the instruction and data caches, it is good to know how these elements work. By minimizing branches in programs, you can help speed up the execution of the instruction codes in your program.

Branch prediction unit

While implementing multiple layers of cache is one way to help speed up processing of program logic, it still does not solve the problem of "jumpy" programs. If a program takes many different logic branches, it may well be impossible for the different layers of cache to keep up, resulting in more last-minute memory access for both instruction code and data elements.

To help solve this problem, the IA-32 platform processors also incorporate *branch prediction*. Branch prediction uses specialized algorithms to attempt to predict which instruction codes will be needed next within a program branch.

Special statistical algorithms and analysis are incorporated to determine the most likely path traveled through the instruction code. Instruction codes along that path are prefetched and loaded into the cache layers.

The Pentium 4 processor utilizes three techniques to implement branch prediction:

- Deep branch prediction
- Dynamic data flow analysis
- Speculative execution

Deep branch prediction enables the processor to attempt to decode instructions beyond multiple branches in the program. Again, statistical algorithms are implemented to predict the most likely path the program will take throughout the branches. While this technique is helpful, it is not totally foolproof.

Dynamic data flow analysis performs statistical real-time analysis of the data flow throughout the processor. Instructions that are predicted to be necessary for the flow of the program but not reached yet by the instruction pointer are passed to the out-of-order execution core (described next). In addition, any instructions that can be executed while the processor is waiting for data related to another instruction are processed.

Speculative execution enables the processor to determine what distant instruction codes not immediately in the instruction code branch are likely to be required, and attempt to process those instructions, again using the out-of-order execution engine.

Out-of-order execution engine

The out-of-order execution engine is one of the greatest improvements to the Pentium 4 processor in terms of speed. This is where instructions are prepared for processing by the execution unit. It contains several buffers to change the order of instructions within the pipeline to increase the performance of the control unit. This is demonstrated in Figure 2-5.

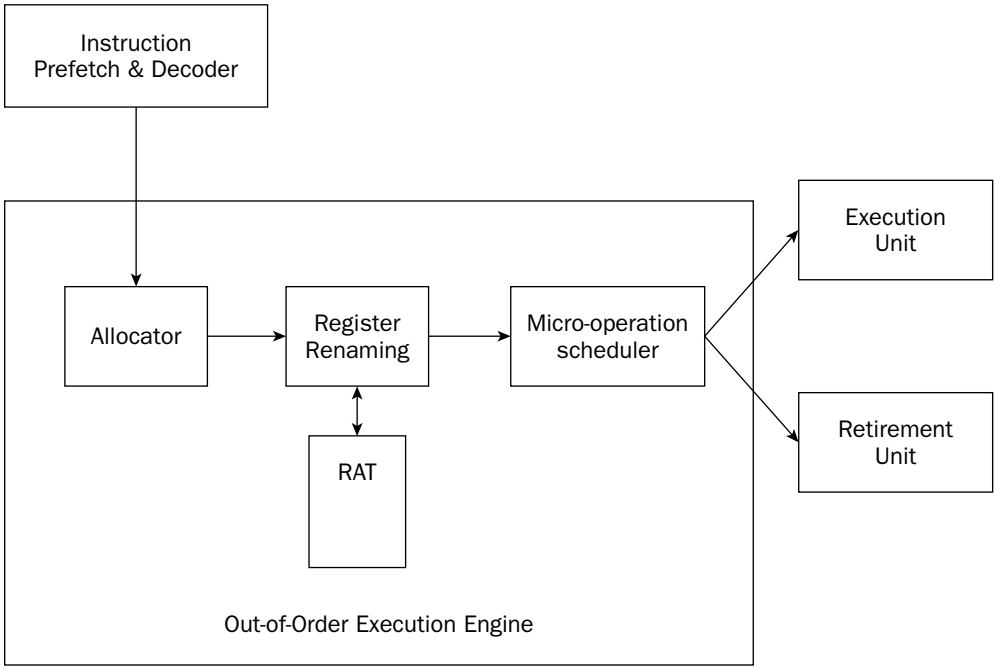

Figure 2-5

Instructions retrieved from the prefetch and decoding pipeline are analyzed and reordered, enabling them to be executed as quickly as possible. By analyzing a large number of instructions, the out-of-order execution engine can find independent instructions that can be executed (and their results saved) until required by the rest of the program. The Pentium 4 processor can have up to 126 instructions in the out-of-order execution engine at any one time.

There are three sections within the out-of-order execution engine:

- The allocator
- Register renaming
- The micro-operation scheduler

The allocator is the traffic cop for the out-of-order execution engine. Its job is to ensure that buffer space is allocated properly for each instruction that the out-of-order execution engine is processing. If a needed resource is not available, the allocator will stall the processing of the instruction and allocate resources for another instruction that can complete its processing.

The register renaming section allocates logical registers to process instructions that require register access. Instead of the eight general-purpose registers available on the IA-32 processor (described later in the "Registers" section), the register renaming section contains 128 logical registers. It maps register requests made by instructions into one of the logical registers, to allow simultaneous access to the same register by multiple instructions. The register mapping is done using the register allocation table (RAT). This helps speed up processing instructions that require access to the same register sets.

The micro-operation scheduler determines when a micro-operation is ready for processing by examining the input elements that it requires. Its job is to send micro-operations that are ready to be processed to the retirement unit, while still maintaining program dependencies. The micro-operation scheduler uses two queues to place micro-operations in — one for micro-operations that require memory access and one for micro-operations that do not. The queues are tied to dispatch ports. Different types of Pentium processors may contain a different number of dispatch ports. The dispatch ports send the micro-operations to the retirement unit.

Retirement unit

The retirement unit receives all of the micro-operations from the pipeline decoders and the out-of-order execution engine and attempts to reassemble the micro-operations into the proper order for the program to properly execute.

The retirement unit passes micro-operations to the execution unit for processing in the order that the out-of-order execution engine sends them, but then monitors the results, reassembling the results into the proper order for the program to execute.

This is accomplished using a large buffer area to hold micro-operation results and place them in the proper order as they are required.

When a micro-operation is completed and the results placed in the proper order, the micro-operation is considered retired and is removed from the retirement unit. The retirement unit also updates information in the branch prediction unit to ensure that it knows which branches have been taken, and which instruction codes have been processed.

Execution unit

The main function of the processor is to execute instructions. This function is performed in the execution unit. A single processor can actually contain multiple execution units, capable of processing multiple instruction codes simultaneously.

The execution unit consists of one or more Arithmetic Logic Units (ALUs) The ALUs are specifically designed to handle mathematical operations on different types of data. The Pentium 4 execution unit includes separate ALUs for the following functions:

- ❑ Simple-integer operations
- ❑ Complex-integer operations
- ❑ Floating-point operations

Low-latency integer execution unit

The low-latency integer execution unit is designed to quickly perform simple integer mathematical operations, such as additions, subtractions, and Boolean operations. Pentium 4 processors are capable of performing two low-latency integer operations per clock cycle, effectively doubling the processing speed.

Complex-integer execution unit

The complex-integer execution unit handles more involved integer mathematical operations. The complex-integer execution unit handles most shift and rotate instructions in four clock cycles. Multiplication and division operations involve long calculation times, and often take 14 to 60 clock cycles.

Floating-point execution unit

The floating-point execution unit differs between the different processors in the IA-32 family. All Pentium processors can process floating-point mathematical operations using the standard floating-point execution unit. Pentium processors that contain MMX and SSE support also perform these calculations in the floating-point execution unit.

The floating-point execution unit contains registers to handle data elements that contain 64-bit to 128-bit lengths. This enables larger floating-point values to be used in calculations, which can speed up complex floating-point calculations, such as digital signal processing and video compression.

Registers

Most of the operations of the processor require processing data. Unfortunately, the slowest operations a processor can undertake are trying to read or store data in memory. As shown in Figure 2-1, when the processor accesses a data element, the request must travel outside of the processor, across the control bus, and into the memory storage unit. This process is not only complicated, but also forces the processor to wait while the memory access is being performed. This downtime could be spent processing other instructions.

To help solve this problem, the processor includes internal memory locations, called *registers*. The registers are capable of storing data elements for processing without having to access the memory storage unit. The downside to registers is that a limited number of them are built into the processor chip.

The IA-32 platform processors have multiple groups of registers of different sizes. Different processors within the IA-32 platform include specialized registers. The core groups of registers available to all processors in the IA-32 family are shown in the following table.

Register	Description
General purpose	Eight 32-bit registers used for storing working data
Segment	Six 16-bit registers used for handling memory access
Instruction pointer	A single 32-bit register pointing to the next instruction code to execute

Table continued on following page

Register	Description
Floating-point data	Eight 80-bit registers used for floating-point arithmetic data
Control	Five 32-bit registers used to determine the operating mode of the processor
Debug	Eight 32-bit registers used to contain information when debugging the processor

The following sections describe the more common registers in greater detail.

General-purpose registers

The general-purpose registers are used to temporarily store data as it is processed on the processor. The general-purpose registers have evolved from the old 8-bit 8080 processor days to 32-bit registers available in the Pentium processors. Each new version of general-purpose registers is created to be completely backwardly compatible with previous processors. Thus, code that uses 8-bit registers on the 8080 chips is still valid on 32-bit Pentium chips.

While most general-purpose registers can be used for holding any type of data, some have acquired special uses, which are consistently used in assembly language programs. The following table shows the general-purpose registers available on the Pentium platform, and what they are most often used for.

Register	Description
EAX	Accumulator for operands and results data
EBX	Pointer to data in the data memory segment
ECX	Counter for string and loop operations
EDX	I/O pointer
EDI	Data pointer for destination of string operations
ESI	Data pointer for source of string operations
ESP	Stack pointer
EBP	Stack data pointer

The 32-bit EAX, EBX, ECX, and EDX registers can also be referenced by 16-bit and 8-bit names to represent the older versions of the registers. Figure 2-6 shows how the registers can be referenced.

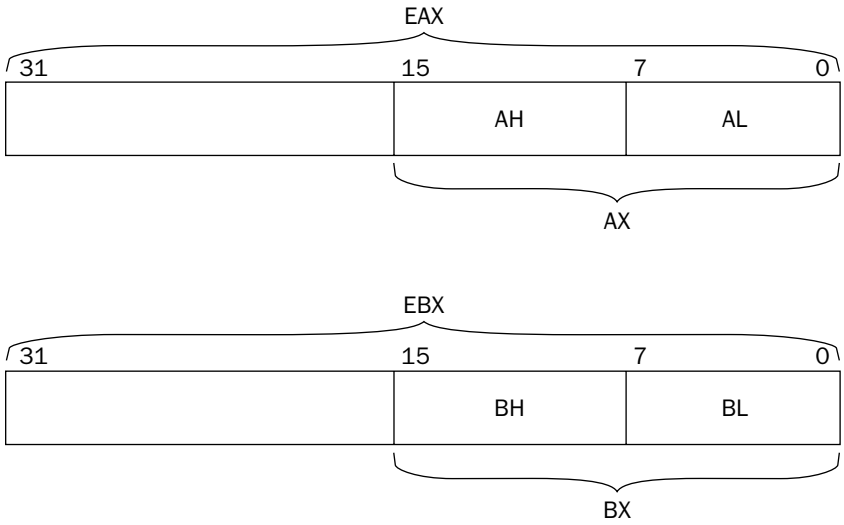

Figure 2-6

By using the reference AX, the lower 16 bits of the EAX register are used. By using the reference AL, the lower 8 bits of the EAX register are used. AH references the next 8 higher bits after AL.

Segment registers

The segment registers are used specifically for referencing memory locations. The IA-32 processor platform allows three different methods of accessing system memory:

- ❑ Flat memory model
- ❑ Segmented memory model
- ❑ Real-address mode

The flat memory model presents all system memory as a contiguous address space. All instructions, data, and the stack are contained in the same address space. Each memory location is accessed by a specific address, called a *linear address*.

The segmented memory model divides the system memory into groups of independent segments, referenced by pointers located in the segment registers. Each segment is used to contain a specific type of data. One segment is used to contain instruction codes, another data elements, and a third the program stack.

Memory locations in segments are defined by logical addresses. A logical address consists of a segment address and an offset address. The processor translates a logical address to a corresponding linear address location to access the byte of memory.

The segment registers are used to contain the segment address for specific data access. The following table describes the available segment addresses.

Segment Register	Description
CS	Code segment
DS	Data segment
SS	Stack segment
ES	Extra segment pointer
FS	Extra segment pointer
GS	Extra segment pointer

Each segment register is 16 bits and contains the pointer to the start of the memory-specific segment. The CS register contains the pointer to the code segment in memory. The code segment is where the instruction codes are stored in memory. The processor retrieves instruction codes from memory based on the CS register value, and an offset value contained in the EIP instruction pointer register. A program cannot explicitly load or change the CS register. The processor assigns its value as the program is assigned a memory space.

The DS, ES, FS, and GS segment registers are all used to point to data segments. By having four separate data segments, the program can help separate data elements, ensuring that they do not overlap. The program must load the data segment registers with the appropriate pointer value for the segments, and reference individual memory locations using an offset value.

The SS segment register is used to point to the stack segment. The stack contains data values passed to functions and procedures within the program.

If a program is using the real address mode, all of the segment registers point to the zero linear address, and are not changed by the program. All instruction codes, data elements, and stack elements are accessed directly by their linear address.

Instruction pointer register

The instruction pointer register (or EIP register), sometimes called the *program counter,* keeps track of the next instruction code to execute. While this sounds like a simple process, with the implementation of the instruction prefetch cache it is not. The instruction pointer points to the next instruction to execute.

An application program cannot directly modify the instruction pointer per se. You cannot specify a memory address and place it in the EIP register. Instead, you must use normal program control instructions, such as jumps, to alter the next instruction to be read into the prefetch cache.

In a flat memory model, the instruction pointer contains the linear address of the memory location for the next instruction code. If the application is using a segmented memory model, the instruction pointer points to a logical memory address, referenced by the contents of the CS register.

Control registers

The five control registers are used to determine the operating mode of the processor, and the characteristics of the currently executing task. The individual control registers are described in the following table.

Control Register	Description
CR0	System flags that control the operating mode and states of the processor
CR1	Not currently used
CR2	Memory page fault information
CR3	Memory page directory information
CR4	Flags that enable processor features and indicate feature capabilities of the processor

The values in the control registers cannot be directly accessed, but the data contained in the control register can be moved to a general-purpose register. Once the data is in a general-purpose register, an application program can examine the bit flags in the register to determine the operating status of the processor and/or currently running task.

If a change is required to a control register flag value, the change can be made to the data in the general-purpose register, and the register moved to the control register. Systems programmers usually modify the values in the control registers. Normal user application programs do not usually modify control registers entries, although they might query flag values to determine the capabilities of the host processor chip on which the application is running.

Flags

For each operation that is performed in the processor, there must be a mechanism to determine whether the operation was successful or not. The processor flags are used to perform this function.

Flags are important to assembly language programs, as they are the only means available to determine whether a program's function succeeded or not. For example, if an application performed a subtraction operation that resulted in a negative value, a special flag within the processor would be set. Without checking the flag, the assembly language program would not have any way to know that something went wrong.

The IA-32 platform uses a single 32-bit register to contain a group of status, control, and system flags. The EFLAGS register contains 32 bits of information that are mapped to represent specific flags of information. Some bits are reserved for future use, to allow additional flags to be defined in future processors. At the time of this writing, 17 bits are used for flags.

The flags are divided into three groups based on function:

- Status flags
- Control flags
- System flags

The following sections describe the flags found in each group.

Status flags

The status flags are used to indicate the results of a mathematical operation by the processor. The current status flags are shown in the following table.

Flag	Bit	Name
CF	0	Carry flag
PF	2	Parity flag
AF	4	Adjust flag
ZF	6	Zero flag
SF	7	Sign flag
OF	11	Overflow flag

The carry flag is set if a mathematical operation on an unsigned integer value generates a carry or a borrow for the most significant bit. This represents an overflow condition for the register involved in the mathematical operation. When an overflow occurs, the data remaining in the register is not the correct answer to the mathematical operation.

The parity flag is used to indicate whether the result register in a mathematical operation contains corrupt data. As a simple check for validity, the parity flag is set if the total number of 1 bits in the result is even, and is cleared if the total number of 1 bits in the result is odd. By checking the parity flag, an application can determine whether the register has been corrupted since the operation.

The adjust flag is used in Binary Coded Decimal (BCD) mathematical operations (see Chapter 7, "Using Numbers"). The adjust flag is set if a carry or borrow operation occurs from bit 3 of the register used for the calculation.

The zero flag is set if the result of an operation is zero. This is most often used as an easy way to determine whether a mathematical operation results in a zero value.

The sign flag is set to the most significant bit of the result, which is the sign bit. This indicates whether the result is positive or negative.

The overflow flag is used in signed integer arithmetic when a positive value is too large, or a negative value is too small, to be properly represented in the register.

Control flags

Control flags are used to control specific behavior in the processor. Currently, only one control flag is defined, the DF flag, or direction flag. It is used to control the way strings are handled by the processor.

When the DF flag is set (set to one), string instructions automatically decrement memory addresses to get the next byte in the string. When the DF flag is cleared (set to zero), string instructions automatically increment memory addresses to get the next byte in the string.

System flags

The system flags are used to control operating system–level operations. Application programs should never attempt to modify the system flags. The system flags are listed in the following table.

Flag	Bit	Name
TF	8	Trap flag
IF	9	Interrupt enable flag
IOPL	12 and 13	I/O privilege level flag
NT	14	Nested task flag
RF	16	Resume flag
VM	17	Virtual-8086 mode flag
AC	18	Alignment check flag
VIF	19	Virtual interrupt flag
VIP	20	Virtual interrupt pending flag
ID	21	Identification flag

The trap flag is set to enable single-step mode. In single-step mode, the processor performs only one instruction code at a time, waiting for a signal to perform the next instruction. This feature is extremely useful when debugging assembly language applications.

The interrupt enable flag controls how the processor responds to signals received from external sources.

The I/O privilege field indicates the I/O privilege level of the currently running task. This defines access levels for the I/O address space. The privilege field value must be less than or equal to the access level required to access the I/O address space; otherwise, any request to access the address space will be denied.

The nested task flag controls whether the currently running task is linked to the previously executed task. This is used for chaining interrupted and called tasks.

The resume flag controls how the processor responds to exceptions when in debugging mode.

The virtual-8086 flag indicates that the processor is operating in virtual-8086 mode instead of protected or real mode.

The alignment check flag is used (along with the AM bit in the CR0 control register) to enable alignment checking of memory references.

The virtual interrupt flag replicates the IF flag when the processor is operating in virtual mode.

The virtual interrupt pending flag is used when the processor is operating in virtual mode to indicate that an interrupt is pending.

The ID flag is interesting in that it indicates whether the processor supports the CPUID instruction. If the processor is able to set or clear this flag, it supports the CPUID instruction. If not, then the CPUID instruction is not available.

Advanced IA-32 Features

The core features of the IA-32 platform mentioned so far are available on all of the processors in the family, starting with the 80386 processor. This section describes some advanced features that the assembly language programmer can utilize when creating programs specifically designed for the Pentium processors.

The x87 floating-point unit

Early processors in the IA-32 family required a separate processor chip to perform floating-point mathematical operations. The 80287 and 80387 processors specialized in providing floating-point arithmetic operations for the computer chips. Programmers who needed fast processing of floating-point operations were forced to turn to additional hardware to support their needs.

Starting with the 80486 processor, the advanced arithmetic functions found in the 80287 and 80387 chips were incorporated into the main processor. To support these functions, additional instruction codes as well as additional registers and execution units were required. Together these elements are referred to as the x87 floating-point unit (FPU).

The x87 FPU incorporates the following additional registers:

FPU Register	Description
Data registers	Eight 80-bit registers for floating-point data
Status register	16-bit register to report the status of the FPU
Control register	16-bit register to control the precision of the FPU
Tag register	16-bit register to describe the contents of the eight data registers

FPU Register	Description
FIP register	48-bit FPU instruction pointer (FIP) points to the next FPU instruction
FDP register	48-bit FPU data pointer (FDP) points to the data in memory
Opcode register	11-bit register to hold the last instruction processed by the FPU

The FPU registers and instruction codes enable assembly language programs to quickly process complex floating-point mathematical functions, such as those required for graphics processing, digital signal processing, and complex business applications. The FPU can process floating-point arithmetic considerably faster than the software simulation used in the standard processor without the FPU. Whenever possible, the assembly language programmer should utilize the FPU for floating-point arithmetic.

Multimedia extensions (MMX)

The Pentium II processor introduced another method for programmers to perform complex integer arithmetic operations. MMX was the first technology to support the Intel Single Instruction, Multiple Data (SIMD) execution model.

The SIMD model was developed to process larger numbers, commonly found in multimedia applications. The SIMD model uses expanded register sizes and new number formats to speed up the complex number crunching required for real-time multimedia presentations.

The MMX environment includes three new floating-point data types that can be handled by the processor:

- 64-bit packed byte integers
- 64-bit packed word integers
- 64-bit packed doubleword integers

These data types are described in detail in Chapter 7, "Using Numbers." To handle the new data formats, MMX technology incorporates the eight FPU registers as special-purpose registers. The MMX registers are named MM0 through MM7, and are used to perform integer arithmetic on the 64-bit packed integers.

While the MMX technology improved processing speeds for complex integer arithmetic, it did nothing for programs that require complex floating-point arithmetic. That problem was solved with the SSE environment.

Streaming SIMD extensions (SSE)

The next generation of SIMD technology was implemented starting with the Pentium III processor. SSE enhances performance for complex floating-point arithmetic, often used in 3-D graphics, motion video, and video conferencing.

The first implementation of SSE in the Pentium III processor incorporated eight new 128-bit registers (called XMM0 through XMM7) and a new data type — a 128-bit packed single-precision floating point. The SSE technology also incorporated additional new instruction codes for processing up to four 128-bit packed single-precision floating-point numbers in a single instruction.

The second implementation of SSE (SSE2) in the Pentium 4 processors incorporates the same XMM registers that SSE uses, and also introduces five new data types:

- 128-bit packed double-precision floating point
- 128-bit packed byte integers
- 128-bit packed word integers
- 128-bit packed doubleword integers
- 128-bit packed quadword integers

These data types are also described in detail in Chapter 7. The new data types and the corresponding instruction codes enable programmers to utilize even more complex mathematical operations within their programs. The 128-bit double-precision floating-point data types allow for advanced 3-D geometry techniques, such as ray tracing, to be performed with minimal processor time.

A third implementation of SSE (SSE3) does not create any new data types, but provides several new instructions for processing both integer and floating-point values in the XMM registers.

Hyperthreading

One of the most exciting features added to the Pentium 4 processor line is *hyperthreading*. Hyperthreading enables a single IA-32 processor to handle multiple program execution threads simultaneously.

The hyperthreading technology consists of two or more logical processors located on a single physical processor. Each logical processor contains a complete set of general-purpose, segment, control, and debug registers. All of the logical processors share the same execution unit. The out-of-order execution core is responsible for handling the separate threads of instruction codes provided by the different logical processors.

Most of the advantages of hyperthreading appear at the operating system level. Multitasking operating systems, such as Microsoft Windows and the various UNIX implementations, can assign application threads to the individual logical processors. To the application programmer, hyperthreading may not appear to be that much of a benefit.

The IA-32 Processor Family

At the time of this writing, the IA-32 family of processors is the most popular computing platform used in desktop workstations and many server environments. The most popular operating system that utilizes the IA-32 platform is Microsoft Windows, although other popular operating systems run on the IA-32 platform, such as Novell file servers, and UNIX-based OSs such as Linux and the BSD derivatives.

While many advances in the IA-32 processor platform have been made throughout the years, many features are common to all IA-32 processors. The features mentioned in this chapter form the core of all assembly language programs written for the IA-32 platform. However, knowing the special features available on a particular processor can help speed your assembly language program along very nicely. This section describes the different processors available in the IA-32 family, and how their features must be taken into consideration when programming for the platform.

Intel processors

Of course, Intel is the main supplier of processors in the IA-32 platform. In today's computing environment, the most commonly used processor platform is the Pentium processor. It is extremely unusual to encounter hardware from the earlier IA-32 processors, such as the 80486 processor.

Unfortunately, several different types of Pentium processors are still active in businesses, schools, and homes. Creating assembly language programs that utilize advanced IA-32 features found only on the latest processors may limit your application's marketability. Conversely, if you know that your programming environment consists of a specific type of processor, utilizing available features may help give your application the performance boost needed to smoke the competition.

This section describes the different types of Pentium processors commonly available in workstations and servers, highlighting the features that are available with each processor.

The Pentium processor family

The core of the Pentium processor line is, of course, the base Pentium processor. The Pentium processor was introduced in 1993 as a replacement for the 80486 processor. The Pentium processor was the first processor to incorporate the dual execution pipeline, and was the first processor to use a full 32-bit address bus and 64-bit internal data path.

While the performance benefits of the Pentium processor were obvious, from a programming point of view, the Pentium processor did not provide any new features beyond the 80486 architecture. All of the core registers and instruction codes from the 80486 processor were supported, including the internal FPU support.

The P6 processor family

The P6 processor family was introduced in 1995 with the Pentium Pro processor. The Pentium Pro processor incorporated a completely new architecture from the original Pentium processor. The P6 family of processors were the first to utilize a superscalar microarchitecture, which greatly increased performance by enabling multiple execution units and instruction prefetch pipelines.

The Pentium MMX and Pentium II processors, part of the P6 family, were the first processors to incorporate MMX technology, and also introduced new low-power states that enabled the processor to be put in sleep mode when idling. This feature helped conserve power, and became the ideal platform for laptop computing devices.

The Pentium III processor was the first processor to incorporate SSE technology, enabling programmers to perform complex floating-point arithmetic operations both quickly and easily.

The Pentium 4 processor family

The Pentium 4 processor was introduced in 2000, and again started a new trend in microprocessor design. The Pentium 4 utilizes the Intel NetBurst architecture, which provides extremely fast processing speeds by incorporating the instruction pipelines, the out-of-order execution core, and execution units.

The Pentium 4 processor supports SSE3, a more advanced SSE technology that implements additional floating-point operations to support high-speed multimedia computations.

The Pentium Xeon processor family

In 2001, Intel introduced the Pentium Xeon processor. It is primarily intended for multi-processor server operations. It supports the MMX, SSE, SSE2, and SSE3 technologies.

Non-Intel processors

While the IA-32 platform is often considered to be an Intel thing, there are many other non-Intel processors available on the market that also implement the IA-32 features. It is possible that your assembly language application may be run on a non-Intel platform, so it is important that you understand some differences between the platforms.

AMD processors

Today, Intel's biggest competitor is AMD. AMD has released a competing processor chip for every release of Intel's Pentium processor. It is not uncommon to run across Microsoft Windows workstations using AMD processors. The following table shows the AMD processor's history.

AMD Processor	Equivalent to	Notes
K5	Pentium	100% software compatible
K6	Pentium MMX	Pentium with full MMX support
K6-2	Pentium II	Uses 3D Now technology
K6-III	Pentium III	
Athlon	Pentium 4	
Athlon XP	Pentium 4 w/SSE	

For the assembly language programmer, the most important difference between AMD and Intel processors is apparent when using SIMD technology. While AMD has duplicated the MMX technology, it has not fully duplicated the newer SSE technology. When Intel introduced SSE in the Pentium II processor, AMD took a different route. The AMD K6-2 processor uses a different SIMD technology called 3D Now. The 3D Now technology uses similar registers and data types as SSE, but it is not software compatible. This has caused high-speed programmers considerable difficulty when programming for SSE functions.

With the release of the Athlon XP processor in 2001, AMD supported SSE integer arithmetic. At the time of this writing, the newest AMD processor chips now fully support SSE technology.

Cyrix processors

While the Cyrix corporation has not been in business for a few years, their IA-32 platform processors still live on in many workstations and low-end servers. It is still possible to run across a Cyrix processor in various environments.

The evolution of the Cyrix processor family mirrored the Intel processors for many versions. The first Pentium-grade processor produced by Cyrix was originally called the 6x86 processor. It is 100 percent software compatible with the Pentium processor. When Intel introduced MMX technology, Cyrix produced the 6x86MX processor (they didn't have a license to call it MMX, but MX was close enough).

When Cyrix was sold to the VIA chipset company, the original Cyrix processor line was renamed. The 6x86 processor was called the M1, and the 6x86MX processor was called M2. Again, these processors retained their compatibility with their Pentium counterparts.

Before the demise of the Cyrix processor, one final version made it to market. Dubbed the Cyrix III, it was also compatible with the Pentium III processor. Unfortunately, similar to AMD, it too had to support SSE using the 3D Now technology, which made it incompatible with assembly language programs written for SSE.

Summary

Before writing an assembly language program, you must know the target processor used when the program is executed. The most popular processor platform in use today is the Intel IA-32 platform. This platform includes the Pentium family of processors from Intel, as well as the Athlon processors from AMD.

The flagship of the IA-32 platform is the Intel Pentium 4 processor. It incorporates the NetBurst architecture to quickly and easily process instructions and data. The core of the NetBurst architecture includes a control unit, an execution unit, registers, and flags.

The control unit controls how the execution unit processes instructions and data. Speed is accomplished by prefetching and decoding instructions from memory long before the execution unit processes them. Instructions can also be processed out of order and the results stored until they are needed in the application.

The execution unit in a Pentium 4 processor has the capability of processing multiple instructions concurrently. Simple integer processes are performed quickly and stored in the out-of-order area in the control unit until needed. Complex integer and floating-point processes are also streamlined to increase performance.

Registers are used as local data storage within the processor to prevent costly memory access for data. The IA-32 platform processors provide several general-purpose registers for holding data and pointers as the program is executed. Instructions are retrieved from memory based on the value of the instruction pointer register. The control register controls the processor's behavior.

A special register containing several flags determines the status and operation of the processor. Each flag represents a different operation within the processor. Status flags indicate the result of operations performed by the processor. Control flags control how the processor behaves for specific operations. System flags determine operating system behavior, and should not be touched by application programmers.

Innovation in the IA-32 platform is alive and well. Many new features have been introduced in recent processor releases. The floating-point unit (FPU) has been incorporated into Pentium processors to assist in floating-point mathematical operations.

To further support complex mathematical processing, the Single Instruction, Multiple Data (SIMD) technology enables the processing of large numerical values in both integer and floating-point form. The Multimedia Extensions (MMX) enable programmers to use 64-bit integers in high-precision integer calculations. Following that, the Streaming SIMD Extensions (SSE) technology enables programmers to use 128-bit single-precision floating-point values, and subsequently, the SSE2 technology enables the use of 128-bit double-precision floating-point data values. These new data types greatly speed up the processing of mathematically intensive programs, such as those used for multimedia processing and digital signal processing.

When programming for the IA-32 platform, you should be aware of the different processors available, and know what functions each processor type supports. The core of the IA-32 platform is the original Pentium processor. It supports the core IA-32 registers and instruction sets, along with simple built-in FPU support. Similar to the Pentium processor, AMD produced the K5 processor, and Cyrix produced the 6x86 processor. Each of these processors is 100 percent software compatible with the IA-32 instruction code set.

Intel introduced MMX functionality in the Pentium II processor line. Following suit, AMD incorporated MMX features in the K6 processor, and Cyrix with the 6x86MX processor. All of these processors include the MMX registers, and the additional MMX instruction codes.

The SSE technology is where things get complicated in the IA-32 world. The Pentium II processor introduced SSE registers and instruction sets, but unfortunately, other processor manufacturers were not able to directly incorporate these features. Instead, AMD and Cyrix implemented the 3D Now technology in their K6-2 (AMD) and Cyrix III (Cyrix) processors. The 3D Now technology provided the same 64-bit integer data types as SSE, but the instruction codes were different.

At the time of this writing, the Pentium 4 processor is the flagship processor for Intel. It supports SSE3 technology, as well as the NetBurst architecture. The AMD rival is the Athlon XP, which now incorporates SSE registers and instruction sets, making it software compatible with the Pentium 4 processor.

Now that you have an understanding of the hardware platform used in this book, it's time to examine the software development environment. The next chapter discusses the assembly language tools that are available in the Linux operating system environment. By using Linux, you can leverage the GNU development tools to create a professional software development environment with minimal cost.

3

The Tools of the Trade

Now that you are familiar with the IA-32 hardware platform, it's time to dig into the tools necessary to create assembly language programs for it. To create assembly language programs, you must have some type of development environment. Many different assembly language development tools are available, both commercially and for free. You must decide which development environment works best for you.

This chapter first examines what development tools you should have to create assembly language programs. Next, the programming development tools produced by the GNU project are discussed. Each tool is described, including downloading and installation.

The Development Tools

Just like any other profession, programming requires the proper tools to perform the job. To create a good assembly language development environment, you must have the proper tools at your disposal. Unlike a high-level language environment in which you can purchase a complete development environment, you often have to piece together an assembly language development environment. At a minimum you should have the following:

- An assembler
- A linker
- A debugger

Additionally, to create assembly language routines for other high-level language programs, you should also have these tools:

- A compiler for the high-level language
- An object code disassembler
- A profiling tool for optimization

The following sections describe each of these tools, and how they are used in the assembly language development environment.

The assembler

To create assembly language programs, obviously you need some tool to convert the assembly language source code to instruction code for the processor. This is where the assembler comes in.

As mentioned in Chapter 1, "What Is Assembly Language?," assemblers are specific to the underlying hardware platform for which you are programming. Each processor family has its own instruction code set. The assembler you select must be capable of producing instruction codes for the processor family on your system (or the system you are developing for).

The assembler produces the instruction codes from source code created by the programmer. If you remember from Chapter 1, there are three components to an assembly language source code program:

- Opcode mnemonics
- Data sections
- Directives

Unfortunately, each assembler uses different formats for each of these components. Programming using one assembler may be totally different from programming using another assembler. While the basics are the same, the way they are implemented can be vastly different.

The biggest difference between assemblers is the assembler directives. While opcode mnemonics are closely related to processor instruction codes, the assembler directives are unique to the individual assembler. The directives instruct the assembler how to construct the instruction code program. While some assemblers have a limited number of directives, some have an extensive number of directives. Directives do everything from defining program sections to implementing if-then statements or while loops.

You may also have to take into consideration how you will write your assembly language programs. Some assemblers come complete with built-in editors that help recognize improper syntax while you are typing the code, while others are simply command-line programs that can only assemble an existing code text file. If the assembler you choose does not contain an editor, you must select a good editor for your environment. While using the UNIX `vi` editor can work for simple programs, you probably wouldn't want to code a 10,000-line assembly program using it.

The bottom line for choosing an assembler is its ability to make creating an instruction code program for your target environment as simple as possible. The next sections describe some common assemblers that are available for the Intel IA-32 platform.

MASM

The granddaddy of all assemblers for the Intel platform, the Microsoft Assembler (MASM) is the product of the Microsoft Corporation. It has been available since the beginning of the IBM-compatible PC, enabling programmers to produce assembly language programs in both the DOS and Windows environments.

Because MASM has been around for so long, numerous tutorials, books, and example programs are floating around, many of which are free or low-cost. While Microsoft no longer sells MASM as a standalone product, it is still bundled with the Microsoft's Visual Studio product line of compilers. The benefit of using Visual Studio is its all-encompassing Integrated Development Environment (IDE). Microsoft has also allowed various companies and organizations to distribute just the MASM 6.0 files free of charge, enabling you to assemble your programs from a command prompt. Doing a Web search for MASM 6.0 will produce a list of sites where it can be downloaded free of charge.

Besides MASM, an independent developer, Steve Hutchessen, has created the MASM32 development environment. MASM32 incorporates the original MASM assembler and the popular Windows Win32 Application Programming Interface (API), used mainly in C and C++ applications. This enables assembly language programmers to create full-blown Windows programs entirely in assembly language programs. The MASM32 Web site is located at `www.masm32.com`.

NASM

The Netwide Assembler (NASM) was developed originally as a commercial assembler package for the UNIX environment. Recently, the developers have released NASM as open-source software for both the UNIX and Microsoft environments. It is fully compatible with all of the Intel instruction code set and can produce executable files in UNIX, 16-bit MS-DOS, and 32-bit Microsoft Windows formats.

Similar to MASM, quite a few books and tutorials are available for NASM. The NASM download page is located at `http://nasm.sourceforge.net`.

GAS

The Free Software Foundation's GNU project has produced many freely available software packages that run in the UNIX operating system environment. The GNU assembler, called **gas,** is the most popular cross-platform assembler available for UNIX.

That is correct, I did say cross-platform. While earlier I mentioned that assemblers are specific to individual processor families, gas is an exception. It was developed to operate on many different processor platforms. Obviously, it must know which platform it is being used on, and creates instruction code programs depending on the underlying platform. Usually gas is capable of automatically detecting the underlying hardware platform and creates appropriate instruction codes for the platform with no operator intervention.

One unique feature of gas is its ability to create instruction codes for a platform other than the one you are programming on. This enables a programmer working on an Intel-based computer to create assembly language programs for a system that is MIPS-based. Of course, the downside is that the programmer can't test the produced program code on the host system.

> *This book uses the GNU assembler to assemble all of the examples. Not only is it a good standalone assembler, it is also what the GNU C compiler uses to convert the compiled C and C++ programs to instruction codes. By knowing how to program assembly language with gas, you can also easily incorporate assembly language functions in your existing C and C++ applications, which is one of the main points of this book.*

HLA

The High Level Assembler (HLA) is the creation of Professor Randall Hyde. It creates Intel instruction code applications on DOS, Windows, and Linux operating systems.

The primary purpose of HLA was to teach assembly language to beginning programmers. It incorporates many advanced directives to help programmers make the leap from a high-level language to assembly language (thus its name). It also has the ability to use normal assembly code statements, providing programmers with a robust platform for easily migrating from high-level languages such as C or C++ to assembly language.

The HLA Web site is located at `http://webster.cs.ucr.edu`. Professor Hyde uses this Web site as a clearinghouse for various assembler information. Not only is a lot of information for HLA located there, it also includes links to many other assembler packages.

The linker

If you are familiar with a high-level language environment, it is possible that you have never had to directly use a **linker.** Many high-level languages such as C and C++ perform both the compile and link steps with a single command.

The process of linking objects involves resolving all defined functions and memory address labels declared in the program code. To do this, any external functions, such as the C language `printf` function, must be included with the object code (or a reference made to an external dynamic library). For this to work automatically, the linker must know where the common object code libraries are located on the computer, or the locations must be manually specified with compiler command-line parameters.

However, most assemblers do not automatically link the object code to produce the executable program file. Instead, a second manual step is required to link the assembly language object code with other libraries and produce an executable program file that can be run on the host operating system. This is the job of the linker.

When the linker is invoked manually, the developer must know which libraries are required to completely resolve any functions used by the application. The linker must be told where to find function libraries and which object code files to link together to produce the resulting file.

Every assembler package includes its own linker. You should always use the appropriate linker for the assembler package you are developing with. This helps ensure that the library files used to link functions together are compatible with each other, and that the format of the output file is correct for the target platform.

The debugger

If you are a perfect programmer, you will never need to use a debugger. However, it is more likely that somewhere in your assembly language programming future you will make a mistake — either a small typo in your 10,000-line program, or a logic mistake in your mathematical algorithm functions. When this happens, it is handy to have a good debugger available in your toolkit.

Similar to assemblers, debuggers are specific to the operating system and hardware platform for which the program was written. The debugger must know the instruction code set of the hardware platform, and understand the registers and memory handling methods of the operating system.

Most debuggers provide four basic functions to the programmer:

- Running the program in a controlled environment, specifying any runtime parameters required
- Stopping the program at any point within the program
- Examining data elements, such as memory locations and registers
- Changing elements in the program while it is running, to facilitate bug removal

The debugger runs the program within its own controlled "sandbox." The sandbox enables the program to access memory areas, registers, and I/O devices, but all under the control of the debugger. The debugger is able to control how the program accesses items and can display information about how and when the program accesses the items.

At any point during the execution of the program, the debugger is able to stop the program and indicate where in the source code the execution was stopped. To accomplish this, the debugger must know the original source code and what instruction codes were generated from which lines of source code. The debugger requires additional information to be compiled into the executable file to identify these elements. Using a specific command-line parameter when the program is compiled or assembled usually accomplishes this task.

When the program is stopped during execution, the debugger is able to display any memory area or register value associated with the program. Again, this is accomplished by running the program within the debugger's sandbox, enabling the debugger to peek inside the program as it is executing. By being able to see how individual source code statements affect the values of memory locations and registers, the programmer can often see where an error in the program occurs. This feature is invaluable to the programmer.

Finally, the debugger provides a means for the programmer to change data values within the program as it is executing. This enables the programmer to make changes to the program as it is running and see how the changes affect the outcome of the program. This is another invaluable feature, saving the time

of having to change values in source code, recompiling the source code, and rerunning the executable program file.

The compiler

If all you plan to do is program in assembly language, a high-level language compiler is not necessary. However, as a professional programmer, you probably realize that creating full-blown applications using only assembly language, although possible, would be a massive undertaking.

Instead, most professional programmers attempt to write as much of the application as possible in a high-level language, such as C or C++, and concentrate on optimizing the trouble spots using assembly language programming. To do this, you must have the proper compiler for your high-level language.

The compiler's job is to convert the high-level language into instruction code for the processor to execute. Most compilers, though, produce an intermediate step. Instead of directly converting the source code to instruction code, the compiler converts the source code to assembly language code. The assembly language code is then converted to instruction code, using an assembler. Many compilers include the assembler process within the compiler, although not all do.

After converting the C or C++ source code to assembly language, the GNU compiler uses the GNU assembler to produce the instruction codes for the linker. You can stop the process between these steps and examine the assembly language code that is generated from the C or C++ source code. If you think something can be optimized, the generated assembly language code can be modified, and the code assembled into new instruction codes.

The object code disassembler

When trying to optimize a high-level language, it usually helps to see how that code is being run on the processor. To do that you need a tool to view the instruction code that is generated by the compiler from the high-level language source code. The GNU compiler enables you to view the generated assembly language code before it is assembled, but what about after the object file is already created?

A **disassembler program** takes either a full executable program or an object code file and displays the instruction codes that will be run by the processor. Some disassemblers even take the process one step further by converting the instruction codes into easily readable assembly language syntax.

After viewing the instruction codes generated by the compiler, you can determine if the compiler produced sufficiently optimized instruction codes or not. If not, you might be able to create your own instruction code functions to replace the compiler-generated functions to improve the performance of your application.

The profiler

If you are working in a C or C++ programming environment, you often need to determine which functions your program is spending most of its time performing. By finding the process-intensive functions, you can narrow down which functions are worth your time trying to optimize. Spending days optimizing a function that only takes 5 percent of the program's processing time would be a waste of your time.

To determine how much processing time each function is taking, you must have a **profiler** in your toolkit. The profiler is able to track how much processor time is spent in each function as it is used during the course of the program execution.

In order to optimize a program, once you narrow down which functions are causing the most time drain, you can use the disassembler to see what instruction codes are being generated. After analyzing the algorithms used to ensure they are optimized, it is possible that you can manually generate the instruction codes using advanced processor instructions that the compiler did not use to optimize the function.

The GNU Assembler

The GNU assembler program (called gas) is the most popular assembler for the UNIX environment. It has the ability to assemble instruction codes from several different hardware platforms, including the following:

- VAX
- AMD 29K
- Hitachi H8/300
- Intel 80960
- M680x0
- SPARC
- Intel 80x86
- Z8000
- MIPS

All of the assembly language examples in this book are written for gas. Many UNIX systems include gas in the installed operating system programs. Most Linux distributions include it by default in the development kit implementations.

This section describes how you can download and install gas, as well as how to create and assemble assembly language programs using it.

Installing the assembler

Unlike most other development packages, the GNU assembler is not distributed in a package by itself. Instead, it is bundled together with other development software in the GNU `binutils` package.

You may or may not need all of the subpackages included with the `binutils` package, but it is not a bad idea to have them installed on your system. The following table shows all of the programs installed by the current `binutils` package (version 2.15):

Package	Description
addr2line	Converts addresses into filenames and line numbers
ar	Creates, modifies, and extracts file archives
as	Assembles assembly language code into object code
c++filt	Filter to demangle C++ symbols
gprof	Program to display program profiling information
ld	Linker to convert object code files into executable files
nlmconv	Converts object code into Netware Loadable Module format
nm	Lists symbols from object files
objcopy	Copies and translates object files
objdump	Displays information from object files
ranlib	Generates an index to the contents of an archive file
readelf	Displays information from an object file in ELF format
size	Lists the section sizes of an object or archive file
strings	Displays printable strings found in object files
strip	Discards symbols
windres	Compiles Microsoft Windows resource files

Most Linux distributions that support software development already include the `binutils` package (especially when the distribution includes the GNU C compiler). You can check for the `binutils` package using your particular Linux distribution package manager. On my Mandrake Linux system, which uses RedHat Package Management (RPM) to install packages, I checked for `binutils` using the following command:

```
$ rpm -qa | grep binutils
libbinutils2-2.10.1.0.2-4mdk
binutils-2.10.1.0.2-4mdk
$
```

The output from the `rpm` query command shows that two RPM packages are installed for `binutils`. The first package, `libbinutils2`, installs the low-level libraries required by the `binutils` packages. The second package, `binutils`, installs the actual packages. The package available on this system is version 2.10.

If you have a Linux distribution based on the Debian package installer, you can query the installed packages using the `dpkg` command:

```
$ dpkg -l | grep binutil
ii  binutils       2.14.90.0.7-3  The GNU assembler, linker and binary utilities
ii  binutils-doc   2.14.90.0.7-3  Documentation for the GNU assembler, linker
$
```

The output shows that the `binutils` version 2.14 package is installed on this Linux system.

> *It is often recommended not to change the **`binutils`** package on your Linux distribution if one is already installed and being used. The **`binutils`** package contains many low-level library files that are used to compile operating system components. If those library files change or are moved, bad things can happen to your system, very bad things.*

If your system does not include the `binutils` package, you can download the package from the `binutils` Web site, located at `http://sources.redhat.com/binutils`. This Web page contains a link to the `binutils` download page, `ftp://ftp.gnu.org/gnu/binutils/`. From there you can download the source code for the current version of `binutils`. At the time of this writing, the current version is 2.15, and the download file is called `binutils-2.15.tar.gz`.

After the installation package is downloaded, it can be extracted into a working directory with the following command:

```
tar -zxvf binutils-2.15.tar.gz
```

This command creates a working directory called `binutils-2.15` under the current directory. To compile the `binutils` packages, change to the working directory, and use the following commands:

```
./configure
make
```

The configure command examines the host system to ensure that all of the packages and utilities required to compile the packages are available on the system. Once the software packages have been compiled, you can use the `make install` command to install the software into common areas for others to use.

Using the assembler

The GNU assembler is a command-line-oriented program. It should be run from a command-line prompt, with the appropriate command-line parameters. One oddity about the assembler is that although it is called gas, the command-line executable program is called `as`.

The command-line parameters available for `as` vary depending on what hardware platform is used for the operating system. The command-line parameters common to all hardware platforms are as follows:

```
as [-a[cdhlns][=file]] [-D] [--defsym sym=val]
   [-f] [--gstabs] [--gstabs+] [--gdwarf2] [--help]
   [-I dir] [-J] [-K] [-L]
   [--listing-lhs-width=NUM] [--listing-lhs-width2=NUM]
   [--listing-rhs-width=NUM] [--listing-cont-lines=NUM]
   [--keep-locals] [-o objfile] [-R] [--statistics] [-v]
   [-version] [--version] [-W] [--warn] [--fatal-warnings]
   [-w] [-x] [-Z] [--target-help] [target-options]
   [--|files ...]
```

These command-line parameters are explained in the following table:

Parameter	Description
-a	Specifies which listings to include in the output
-D	Included for backward compatibility, but ignored
--defsym	Define symbol and value before assembling source code
-f	Fast assemble, skips comments and white space
--gstabs	Includes debugging information for each source code line
--gstabs+	Includes special gdb debugging information
-I	Specify directories to search for include files
-J	Do not warn about signed overflows
-K	Included for backward compatibility, but ignored
-L	Keep local symbols in the symbol table
--listing-lhs-width	Set the maximum width of the output data column
--listing-lhs-width2	Set the maximum width of the output data column for continual lines
--listing-rhs-width	Set the maximum width of input source lines
--listing-cont-lines	Set the maximum number of lines printed in a listing for a single line of input
-o	Specify name of the output object file
-R	Fold the data section into the text section
--statistics	Display the maximum space and total time used by the assembly
-v	Display the version number of `as`
-W	Do not display warning messages
--	Use standard input for source files

An example of converting the assembly language program `test.s` to the object file `test.o` would be as follows:

```
as –o test.o test.s
```

This creates an object file `test.o` containing the instruction codes for the assembly language program. If anything is wrong in the program, the assembler will let you know and indicates where the problem is in the source code:

```
$ as -o test.o test.s
test.s: Assembler messages:
test.s:16: Error: no such instruction: `mpvl $4,%eax'
$
```

The preceding error message specifically points out that the error occurred in line 16 and displays the text for that line. Oops, looks like a typo in line 16.

A word about opcode syntax

One of the more confusing parts of the GNU assembler is the syntax it uses for representing assembly language code in the source code file. The original developers of gas chose to implement AT&T opcode syntax for the assembler.

The AT&T opcode syntax originated from AT&T Bell Labs, where the UNIX operating system was created. It was formed based on the opcode syntax of the more popular processor chips used to implement UNIX operating systems at the time. While many processor manufacturers used this format, unfortunately Intel chose to use a different opcode syntax.

Because of this, using gas to create assembly language programs for the Intel platform can be tricky. Most documentation for Intel assembly language programming uses the Intel syntax, while most documentation written for older UNIX systems uses AT&T syntax. This can cause confusion and extra work for the gas programmer.

Most of the differences appear in specific instruction formats, which will be covered as the instructions are discussed in the chapters. The main differences between Intel and AT&T syntax are as follows:

- AT&T immediate operands use a `$` to denote them, whereas Intel immediate operands are undelimited. Thus, when referencing the decimal value 4 in AT&T syntax, you would use `$4`, and in Intel syntax you would just use `4`.
- AT&T prefaces register names with a `%`, while Intel does not. Thus, referencing the EAX register in AT&T syntax, you would use `%eax`.
- AT&T syntax uses the opposite order for source and destination operands. To move the decimal value 4 to the EAX register, AT&T syntax would be movl `$4, %eax`, whereas for Intel it would be `mov eax, 4`.
- AT&T syntax uses a separate character at the end of mnemonics to reference the data size used in the operation, whereas in Intel syntax the size is declared as a separate operand. The AT&T instruction `movl $test, %eax` is equivalent to `mov eax, dword ptr test` in Intel syntax.
- Long calls and jumps use a different syntax to define the segment and offset values. AT&T syntax uses `ljmp $section, $offset`, whereas Intel syntax uses `jmp section:offset`.

While the differences can make it difficult to switch between the two formats, if you stick to one or the other you should be OK. If you learn assembly language coding using the AT&T syntax, you will be comfortable creating assembly language programs on most any UNIX system available, on most any hardware platform. If you plan on doing cross-platform work between UNIX and Microsoft Windows systems, you may want to consider using Intel syntax for your applications.

The GNU assembler does provide a method for using Intel syntax instead of AT&T syntax, but at the time of this writing it is somewhat clunky and mostly undocumented. The `.intel_syntax` directive in an assembly language program tells `as` to assemble the instruction code mnemonics using Intel syntax instead of AT&T syntax. Unfortunately, there are still numerous limitations to this method. For example, even though the source and destination orders switch to Intel syntax, you must still prefix register names with the percent sign (as in AT&T syntax). It is hoped that some future version of `as` will support full Intel syntax assembly code.

All of the assembly language programs presented in this book use the AT&T syntax.

The GNU Linker

The GNU linker, `ld`, is used to link object code files into either executable program files or library files. The `ld` program is also part of the GNU `binutils` package, so if you already have the GNU assembler installed, the linker is likely to be installed.

The format of the `ld` command is as follows:

```
ld [-o output] objfile...
   [-Aarchitecture] [-b input-format] [-Bstatic]
   [-Bdynamic] [-Bsymbolic] [-c commandfile] [--cref]
   [-d|-dc|-dp]
   [-defsym symbol=expression] [--demangle]
   [--no-demangle] [-e entry] [-embedded-relocs] [-E]
   [-export-dynamic] [-f name] [--auxiliary name]
   [-F name] [--filter name] [-format input-format]
   [-g] [-G size] [-h name] [-soname name] [--help]
   [-i] [-lar] [-Lsearchdir] [-M] [-Map mapfile]
   [-m emulation] [-n|-N] [-noinhibit-exec]
   [-no-keep-memory] [-no-warn-mismatch] [-Olevel]
   [-oformat output-format] [-R filename] [-relax]
   [-r|-Ur] [-rpath directory] [-rpath-link directory]
   [-S] [-s] [-shared] [-sort-common]
   [-split-by-reloc count] [-split-by-file]
   [-T commandfile]
   [--section-start sectionname=sectionorg]
   [-Ttext textorg] [-Tdata dataorg] [-Tbss bssorg]
   [-t] [-u sym] [-V] [-v] [--verbose] [--version]
   [-warn-common] [-warn-constructors]
   [-warn-multiple-gp] [-warn-once]
   [-warn-section-align] [--whole-archive]
   [--no-whole-archive] [--wrap symbol] [-X] [-x]
```

While that looks like a lot of command-line parameters, in reality you should not have to use very many of them at any one time. This just shows that the GNU linker is an extremely versatile program and has many different capabilities. The following table describes the command-line parameters that are used for the Intel platform.

Parameter	Description
-b	Specifies the format of the object code input files.
-Bstatic	Use only static libraries.
-Bdynamic	Use only dynamic libraries.
-Bsymbolic	Bind references to global symbols in shared libraries.
-c	Read commands from the specified command file.
--cref	Create a cross-reference table.
-d	Assign space to common symbols even if relocatable output is specified.
-defsym	Create the specified global symbol in the output file.
--demangle	Demangle symbol names in error messages.
-e	Use the specified symbol as the beginning execution point of the program.
-E	For ELF format files, add all symbols to the dynamic symbol table.
-f	For ELF format shared objects, set the DT_AUXILIARY name.
-F	For ELF format shared objects, set the DT_FILTER name.
-format	Specify the format of the object code input files (same as -b).
-g	Ignored. Used for compatibility with other tools.
-h	For ELF format shared objects, set the DT_SONAME name.
-i	Perform an incremental link.
-l	Add the specified archive file to the list of files to link.
-L	Add the specified path to the list of directories to search for libraries.
-M	Display a link map for diagnostic purposes.
-Map	Create the specified file to contain the link map.
-m	Emulate the specified linker.
-N	Specifies read/write text and data sections.
-n	Sets the text section to be read only.
-noinhibit-exec	Produce an output file even if non-fatal link errors appear.
-no-keep-memory	Optimize link for memory usage.
-no-warn-mismatch	Allow linking mismatched object files.

Table continued on following page

Parameter	Description
-O	Generate optimized output files.
-o	Specify the name of the output file.
-oformat	Specify the binary format of the output file.
-R	Read symbol names and addresses from the specified filename.
-r	Generates relocatable output (called partial linking).
-rpath	Add the specified directory to the runtime library search path.
-rpath-link	Specify a directory to search for runtime shared libraries.
-S	Omits debugger symbol information from the output file.
-s	Omits all symbol information from the output file.
-shared	Create a shared library.
-sort-common	Do not sort symbols by size in output file.
-split-by-reloc	Creates extra sections in the output file based on the specified size.
-split-by-file	Creates extra sections in the output file for each object file.
--section-start	Locates the specified section in the output file at the specified address.
-T	Specifies a command file (same as -c).
-Ttext	Use the specified address as the starting point for the text section.
-Tdata	Use the specified address as the starting point for the data section.
-Tbss	Use the specified address as the starting point for the bss section.
-t	Displays the names of the input files as they are being processed.
-u	Forces the specified symbol to be in the output file as an undefined symbol.
-warn-common	Warn when a common symbol is combined with another common symbol.
-warn-constructors	Warn if any global constructors are not used.
-warn-once	Warn only once for each undefined symbol.
-warn-section-align	Warn if the output section address is changed due to alignment.
--whole-archive	For the specified archive files, include all of the files in the archive.
-X	Delete all local temporary symbols.
-x	Delete all local symbols.

For the simplest case, to create an executable file from an object file generated from the assembler, you would use the following command:

```
ld -o mytest mytest.o
```

This command creates the executable file `test` from the object code file `test.o`. The executable file is created with the proper permissions so it can be run from the command line in a UNIX console. Here's an example of the process:

```
$ld -o test test.o
$ ls -al test
-rwxr-xr-x    1 rich     rich          787 Jul  6 11:53 test
$ ./test
Hello world!
$
```

The linker automatically created the executable file with UNIX 755 mode access, allowing anyone on the system to execute it but only the owner to modify it.

The GNU Compiler

The GNU Compiler Collection (`gcc`) is the most popular development system for UNIX systems. Not only is it the default compiler for Linux and most open-source BSD-based systems (such as FreeBSD and NetBSD), it is also popular on many commercial UNIX distributions as well.

`gcc` is capable of compiling many different high-level languages. At the time of this writing, `gcc` could compile the following high-level languages:

- C
- C++
- Objective-C
- Fortran
- Java
- Ada

Not only does `gcc` provide a means for compiling C and C++ applications, it also provides the libraries necessary to run C and C++ applications on the system. The following sections describe how to install `gcc` on your system and how to use it to compile high-level language programs.

Downloading and installing gcc

Many UNIX systems already include the C development environment, installed by default. The `gcc` package is required to compile C and C++ programs. For systems using RPM, you can check for `gcc` using the following:

```
$ rpm -qa | grep gcc
gcc-cpp-2.96-0.48mdk
gcc-2.96-0.48mdk
gcc-c++-2.96-0.48mdk
$
```

This shows that the `gcc` C and C++ compilers version 2.96 are installed. If your system does not have the `gcc` package installed, the first place to look should be your Linux distribution CDs. If a version of `gcc` came bundled with your Linux distribution, the easiest thing to do is to install it from there. As with the `binutils` package, the `gcc` package includes many libraries that must be compatible with the programs running on the system, or problems will occur.

If you are using a UNIX system that does not have a `gcc` package, you can download the `gcc` binaries (remember, you can't compile source code if you don't have a compiler) from the `gcc` Web site. The `gcc` home page is located at `http://gcc.gnu.org`.

At the time of this writing, the current version of `gcc` available is version 3.4.0. If the current `gcc` package is not available as a binary distribution for your platform, you can download an older version to get started, and then download the complete source code distribution for the latest version.

Using gcc

The GNU compiler can be invoked using several different command-line formats, depending on the source code to compile and the underlying hardware of the operating system. The generic command line format is as follows:

```
gcc [-c|-S|-E] [-std=standard]
    [-g] [-pg] [-Olevel]
    [-Wwarn...] [-pedantic]
    [-Idir...] [-Ldir...]
    [-Dmacro[=defn]...] [-Umacro]
    [-foption...] [-mmachine-option...]
    [-o outfile] infile...
```

The generic parameters are described in the following table.

Parameter	Description
-c	Compile or assemble code, but do not link
-S	Stop after compiling, but do not assemble
-E	Stop after preprocessing, but do not compile
-o	Specifies the output filename to use
-v	Display the commands used at each stage of compilation
-std	Specifies the language standard to use
-g	Produce debugging information

Parameter	Description
-pg	Produce extra code used by gprof for profiling
-O	Optimize executable code
-W	Sets compiler warning message level
-pedantic	Issue mandatory diagnostics listing in the C standard
-I	Specify directories for include files
-L	Specify directories for library files
-D	Predefine macros used in the source code
-U	Cancel any defined macros
-f	Specify options used to control the behavior of the compiler
-m	Specify hardware-dependant options

Again, many command-line parameters can be used to control the behavior of `gcc`. In most cases, you will only need to use a couple of parameters. If you plan on using the debugger to watch the program, the `-g` parameter must be used. For Linux systems, the `-gstabs` parameter provides additional debugging information in the program for the GNU debugger (discussed later in the "Using GDB" section).

To test the compiler, you can create a simple C language program to compile:

```
#include <stdio.h>
int main()
{
   printf("Hello, world!\n");
   exit(0);
}
```

This simple C program can be compiled and run using the following commands:

```
$ gcc -o ctest ctest.c
$ ls -al ctest
-rwxr-xr-x    1 rich     rich        13769 Jul  6 12:02 ctest*
$ ./ctest
Hello, world!
$
```

As expected, the `gcc` compiler creates an executable program file, called `ctest`, and assigns it the proper permissions to be executed (note that this format does not create the intermediate object file). When the program is run, it produces the expected output on the console.

One extremely useful command-line parameter in `gcc` is the `-S` parameter. This creates the intermediate assembly language file created by the compiler, before the assembler assembles it. Here's a sample output using the `-S` parameter:

```
$ gcc -S ctest.c
$ cat ctest.s
        .file   "ctest.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "Hello, world!\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        subl    $12, %esp
        pushl   $0
        call    exit
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
$
```

The `ctest.s` file shows how the compiler created the assembly language instructions to implement the C source code program. This is useful when trying to optimize C applications to determine how the compiler is implementing various C language functions in instruction code. You may also notice that the generated assembly language program uses two C functions — `printf` and `exit`. In Chapter 4, "A Sample Assembly Language Program," you will see how assembly language programs can easily use the C library functions already installed on your system.

The GNU Debugger Program

Many professional programmers use the GNU debugger program (`gdb`) to debug and troubleshoot C and C++ applications. What you may not know is that it can also be used to debug assembly language programs as well. This section describes the `gdb` package, including how to download, install, and use its basic features. It is used throughout the book as the debugger tool for assembly language applications.

Downloading and installing gdb

The `gdb` package is often a standard part of Linux and BSD development systems. You can use the appropriate package manager to determine whether it is installed on your system:

```
$ rpm -qa | grep gdb
libgdbm1-1.8.0-14mdk
libgdbm1-devel-1.8.0-14mdk
gdb-5.0-11mdk
$
```

This Mandrake Linux system has version 5.0 of the `gdb` package installed, along with two library packages used by `gdb`.

If your system does not have `gdb` installed, you can download it from its Web site at `www.gnu.org/software/gdb/gdb.html`. At the time of this writing, the current version of `gdb` is 6.1.1 and is available for download from `ftp://sources.redhat.com/pub/gdb/releases` in file `gdb-6.1.tar.gz`.

After downloading the distribution file, it can be unpacked into a working directory using the following command:

```
tar -zxvf gdb-6.1.tar.gz
```

This creates the directory `gdb-6.1`, with all of the source code files. To compile the package, go to the working directory, and use the following commands:

```
./configure
make
```

This compiles the source code files into the necessary library files and the `gdb` executable file. These can be installed using the command **make install**.

Using gdb

The GNU debugger command-line program is called `gdb`. It can be run with several different parameters to modify its behavior. The command-line format for `gdb` is as follows:

```
gdb [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev]
    [-s symfile] [-e prog] [-se prog] [-c core] [-x cmds] [-d dir]
    [prog[core|procID]]
```

The command-line parameters are described in the following table.

Parameter	Description
-b	Set the line speed of the serial interface for remote debugging
-batch	Run in batch mode
-c	Specify the core dump file to analyze
-cd	Specify the working directory
-d	Specify a directory to search for source files

Table continued on following page

Parameter	Description
-e	Specify the file to execute
-f	Output filename and line numbers in standard format when debugging
-nx	Do not execute commands from .gdbinit file
-q	Quiet mode — don't print introduction
-s	Specify the filename for symbols
-se	Specify the filename for both symbols and to execute
-tty	Set device for standard input and output
-x	Execute gdb commands from the specified file

To use the debugger, the executable file must have been compiled or assembled with the `-gstabs` option, which includes the necessary information in the executable file for the debugger to know where in the source code file the instruction codes relate. Once `gdb` starts, it uses a command-line interface to accept debugging commands:

```
$ gcc -gstabs -o ctest ctest.c
$ gdb ctest
GNU gdb 5.0mdk-11mdk Linux-Mandrake 8.0
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-mandrake-linux"...
(gdb)
```

At the `gdb` command prompt, you can enter debugging commands. A huge list of commands can be used. Some of the more useful ones are described in the following table.

Command	Description
break	Set a breakpoint in the source code to stop execution
watch	Set a watchpoint to stop execution when a variable reaches a specific value
info	Observe system elements, such as registers, the stack, and memory
x	Examine memory location
print	Display variable values

Command	Description
run	Start execution of the program within the debugger
list	List specified functions or lines
next	Step to the next instruction in the program
step	Step to the next instruction in the program
cont	Continue executing the program from the stopped point
until	Run the program until it reaches the specified source code line (or greater)

Here's a short example of a `gdb` session:

```
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5               printf("Hello, world!\n");
6               exit(0);
7       }
(gdb) break main
Breakpoint 1 at 0x8048496: file ctest.c, line 5.
(gdb) run
Starting program: /home/rich/palp/ctest

Breakpoint 1, main () at ctest.c:5
5               printf("Hello, world!\n");
(gdb) next
Hello, world!
6               exit(0);
(gdb) next

Program exited normally.
(gdb)quit
$
```

First, the `list` command is used to show the source code line numbers. Next, a breakpoint is created at the main label using the `break` command, and the program is started with the `run` command. Because the breakpoint was set to main, the program immediately stops running before the first source code statement after `main`. The `next` command is used to step to the next line of source code, which executes the `printf` statement. Another `next` command is used to execute the `exit` statement, which terminates the application. Although the application terminated, you are still in the debugger, and can choose to run the program again.

The KDE Debugger

The GNU debugger is an extremely versatile tool, but its user interface leaves quite a bit to be desired. Often, trying to debug large applications with `gdb` can be difficult. To fix this, several different graphical front-end programs have been created for `gdb`. One of the more popular ones is the KDE debugger (`kdbg`), created by Johannes Sixt.

The `kdbg` package uses the K Desktop Environment (KDE) platform, an X-windows graphical environment used mainly on open-source UNIX systems such as Linux, but also available on other UNIX platforms. It was developed using the Qt graphical libraries, so the Qt runtime libraries must also be installed on the system.

Downloading and installing kdbg

Many Linux distributions include the `kdbg` package as an extra package that is not installed by default. You can check the distribution package manager on your Linux system to see if it is already installed, or if it can be installed from the Linux distribution disks. On my Mandrake system, it is included in the supplemental programs disk:

```
$ ls kdbg*
kdbg-1.2.0-0.6mdk.i586.rpm
$
```

If the package is not included with your Linux system, you can download the source code for the `kdbg` package from the `kdbg` Web site, `http://members.nextra.at/johsixt/kdbg.html`. At the time of this writing, the current stable release of `kdbg` is version 1.2.10. The current beta release is 1.9.5.

> *The **`kdbg`** source code installation requires that the KDE development header files be available. These are usually included in the KDE development package with the Linux distribution and may already be installed.*

Using kdbg

After installing `kdbg`, you can use it by opening a command prompt window from the KDE desktop and typing the **kdbg** command. After `kdbg` starts, you must select both the executable file to debug, as well as the original source code file using either the File menu items or the toolbar icons.

Once the executable and source code files are loaded, you can begin the debugging session. Because `kdbg` is a graphical interface for `gdb`, the same commands are available, but in a graphical form. You can set breakpoints within the application by highlighting the appropriate line of source code and clicking the stop sign icon on the toolbar (see Figure 3-1).

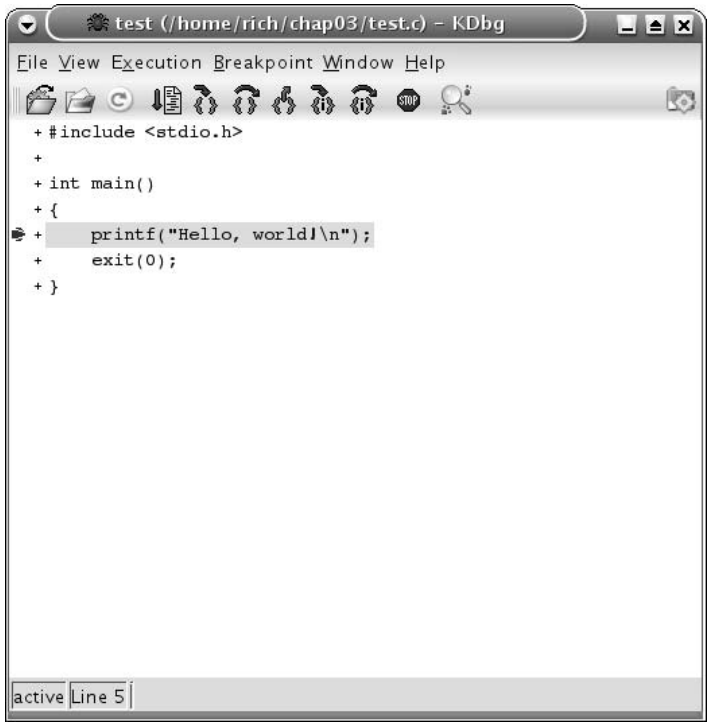

Figure 3-1

If you want to watch memory or register values during the execution of the program, you can select them by clicking the View menu item, and selecting which windows you want to view. You can also open an output window to view the program output as it executes. Figure 3-2 shows an example of the Registers window.

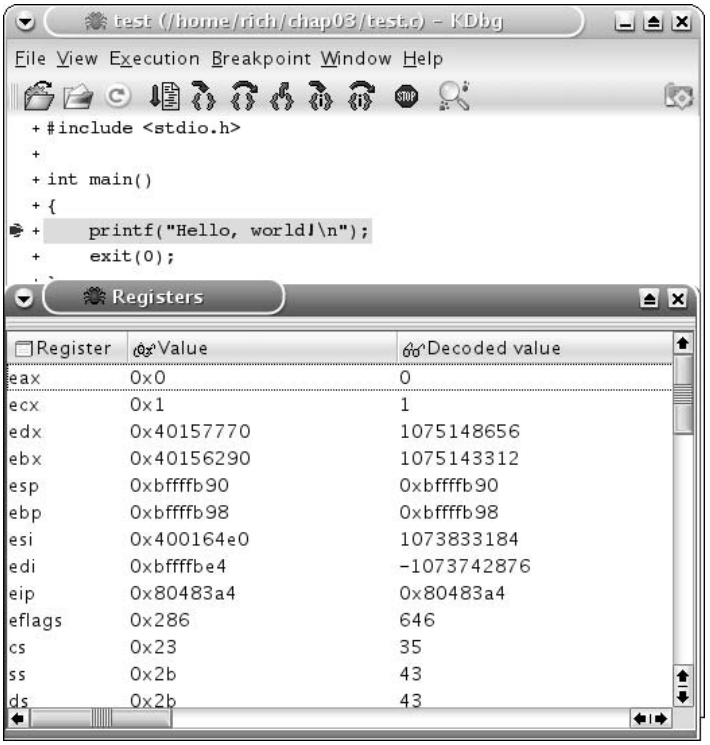

Figure 3-2

After the program files are loaded, and the desired view windows are set, you can start the program execution by clicking the run icon button. Just as in `gdb`, the program will execute until it reaches the first breakpoint. When it reaches the breakpoint, you can step through the program using the step icon button until the program finishes.

The GNU Objdump Program

The GNU `objdump` program is another utility found in the `binutils` package that can be of great use to programmers. Often it is necessary to view the instruction codes generated by the compiler in the object code files. The `objdump` program will display not only the assembly language code, but the raw instruction codes generated as well.

This section describes the `objdump` program, and how you can use it to view the underlying instruction codes contained within a high-level language program.

Using objdump

The `objdump` command-line parameters specify what functions the program will perform on the object code files, and how it will display the information it retrieves. The command-line format of `objdump` is as follows:

```
objdump [-a|--archive-headers] [-b bfdname|--target=bfdname]
        [-C|--demangle[=style] ] [-d|--disassemble]
        [-D|--disassemble-all] [-z|--disassemble-zeroes]
        [-EB|-EL|--endian={big | little }] [-f|--file-headers]
        [--file-start-context] [-g|--debugging]
        [-e|--debugging-tags] [-h|--section-headers|--headers]
        [-i|--info] [-j section|--section=section]
        [-l|--line-numbers] [-S|--source]
        [-m machine|--architecture=machine]
        [-M options|--disassembler-options=options]
        [-p|--private-headers] [-r|--reloc]
        [-R|--dynamic-reloc] [-s|--full-contents]
        [-G|--stabs] [-t|--syms] [-T|--dynamic-syms]
        [-x|--all-headers] [-w|--wide]
        [--start-address=address] [--stop-address=address]
        [--prefix-addresses] [--[no-]show-raw-insn]
        [--adjust-vma=offset] [-V|--version] [-H|--help]
        objfile...
```

The command-line parameters are described in the following table.

Parameter	Description
-a	If any files are archives, display the archive header information
-b	Specify the object code format of the object code files
-C	Demangle low-level symbols into user-level names
-d	Disassemble the object code into instruction code
-D	Disassemble all sections into instruction code, including data
-EB	Specify big-endian object files
-EL	Specify little-endian object files
-f	Display summary information from the header of each file
-G	Display the contents of the debug sections
-h	Display summary information from the section headers of each file
-i	Display lists showing all architectures and object formats
-j	Display information only for the specified section

Parameter	Description
-l	Label the output with source code line numbers
-m	Specify the architecture to use when disassembling
-p	Display information specific to the object file format
-r	Display the relocation entries in the file
-R	Display the dynamic relocation entries in the file
-s	Display the full contents of the specified sections
-S	Display source code intermixes with disassembled code
-t	Display the symbol table entries of the files
-T	Display the dynamic symbol table entries of the files
-x	Display all available header information of the files
--start-address	Start displaying data at the specified address
--stop-address	Stop displaying data at the specified address

The `objdump` program is an extremely versatile tool to have available. It can decode many different types of binary files besides just object code files. For the assembly language programmer, the `-d` parameter is the most interesting, as it displays the disassembled object code file.

An objdump example

Using the sample C program, you can create an object file to dump by compiling the program with the `-c` option:

```
$ gcc -c ctest.c
$ objdump -d ctest.o

ctest.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 ec 0c                sub    $0xc,%esp
   9:   68 00 00 00 00          push   $0x0
   e:   e8 fc ff ff ff          call   f <main+0xf>
  13:   83 c4 10                add    $0x10,%esp
  16:   83 ec 0c                sub    $0xc,%esp
  19:   6a 00                   push   $0x0
  1b:   e8 fc ff ff ff          call   1c <main+0x1c>
$
```

The disassembled object code file created by your system may differ from this example depending on the specific compiler and compiler version used. This example shows both the assembly language mnemonics created by the compiler and the corresponding instruction codes. You may notice, however, that the memory addresses referenced in the program are zeroed out. These values will not be determined until the linker links the application and prepares it for execution on the system. In this step of the process, however, you can easily see what instructions are used to perform the functions.

The GNU Profiler Program

The GNU profiler (`gprof`) is another program included in the `binutils` package. This program is used to analyze program execution and determine where "hot spots" are in the application.

The application hot spots are functions that require the most amount of processing time as the program runs. Often, they are the most mathematically intensive functions, but that is not always the case. Functions that are I/O intensive can also increase processing time.

This section describes the GNU profiler, and provides a simple demonstration that illustrates how it is used in a C program to view how much time different functions consume in an application.

Using the profiler

As with all the other tools, `gprof` is a command-line program that uses multiple parameters to control its behavior. The command-line format for `gprof` is as follows:

```
gprof [ -[abcDhilLsTvwxyz] ] [ -[ACeEfFJnNOpPqQZ][name] ]
      [ -I dirs ] [ -d[num] ] [ -k from/to ]
      [ -m min-count ] [ -t table-length ]
      [ --[no-]annotated-source[=name] ]
      [ --[no-]exec-counts[=name] ]
      [ --[no-]flat-profile[=name] ] [ --[no-]graph[=name] ]
      [ --[no-]time=name] [ --all-lines ] [ --brief ]
      [ --debug[=level] ] [ --function-ordering ]
      [ --file-ordering ] [ --directory-path=dirs ]
      [ --display-unused-functions ] [ --file-format=name ]
      [ --file-info ] [ --help ] [ --line ] [ --min-count=n ]
      [ --no-static ] [ --print-path ] [ --separate-files ]
      [ --static-call-graph ] [ --sum ] [ --table-length=len ]
      [ --traditional ] [ --version ] [ --width=n ]
      [ --ignore-non-functions ] [ --demangle[=STYLE] ]
      [ --no-demangle ] [ image-file ] [ profile-file ... ]
```

This alphabet soup of parameters is split into three groups:

- Output format parameters
- Analysis parameters
- Miscellaneous parameters

The output format options, described in the following table, enable you to modify the output produced by `gprof`.

Parameter	Description
-A	Display source code for all functions, or just the functions specified
-b	Don't display verbose output explaining the analysis fields
-C	Display a total tally of all functions, or only the functions specified
-i	Display summary information about the profile data file
-I	Specifies a list of search directories to find source files
-J	Do not display annotated source code
-L	Display full pathnames of source filenames
-p	Display a flat profile for all functions, or only the functions specified
-P	Do not print a flat profile for all functions, or only the functions specified
-q	Display the call graph analysis
-Q	Do not display the call graph analysis
-y	Generate annotated source code in separate output files
-Z	Do not display a total tally of functions and number of times called
--function-reordering	Display suggested reordering of functions based on analysis
--file-ordering	Display suggested object file reordering based on analysis
-T	Display output in traditional BSD style
-w	Set the width of output lines
-x	Every line in annotated source code is displayed within a function
--demangle	C++ symbols are demangled when displaying output

The analysis parameters, described in the following table, modify the way `gprof` analyzes the data contained in the analysis file.

Parameter	Description
-a	Does not analyze information about statistically declared (private) functions
-c	Analyze information on child functions that were never called in the program

Parameter	Description
-D	Ignore symbols that are not known to be functions (only on Solaris and HP OSs)
-k	Don't analyze functions matching a beginning and ending symspec
-l	Analyze the program by line instead of function
-m	Analyze only functions called more than a specified number of times
-n	Analyze only times for specified functions
-N	Don't analyze times for the specified functions
-z	Analyze all functions, even those that were never called

Finally, the miscellaneous parameters, described in the following table, are parameters that modify the behavior of `gprof`, but don't fit into either the output or analysis groups.

Parameter	Description
-d	Put `gprof` in debug mode, specifying a numerical debug level
-O	Specify the format of the profile data file
-s	Force `gprof` to just summarize the data in the profile data file
-v	Print the version of `gprof`

In order to use `gprof` on an application, you must ensure that the functions you want to monitor are compiled using the `-pg` parameter. This parameter compiles the source code, inserting a call to the `mcount` subroutine for each function in the program. When the application is run, the `mcount` subroutine creates a call graph profile file, called `gmon.out`, which contains timing information for each function in the application.

> *Be careful when running the application, as each run will overwrite the* ***`gmon.out`*** *file. If you want to take multiple samples, you must include the name of the output file on the* ***`gprof`*** *command line and use different filenames for each sample.*

After the program to test finishes, the `gprof` program is used to examine the call graph profile file to analyze the time spent in each function. The `gprof` output contains three reports:

- A flat profile report, which lists total execution times and call counts for all functions
- A listing of functions sorted by the time spent in each function and its children
- A listing of cycles, showing the members of the cycles and their call counts

By default, the `gprof` output is directed to the standard output of the console. You must redirect it to a file if you want to save it.

A profile example

To demonstrate the `gprof` program, you must have a high-level language program that uses functions to perform actions. I created the following simple demonstration program in C, called `demo.c`, to demonstrate the basics of `gprof`:

```
#include <stdio.h>

void function1()
{
        int i, j;
        for(i=0; i <100000; i++)
                j += i;
}
void function2()
{
        int i, j;
        function1();
        for(i=0; i < 200000; i++)
                j = i;
}

int main()
{
        int i, j;
        for (i = 0; i <100; i++)
                function1();

        for(i = 0; i<50000; i++)
                function2();
        return 0;
}
```

This is about as simple as it gets. The main program has two loops: one that calls `function1()` 100 times, and one that calls `function2()` 50,000 times. Each of the functions just performs simple loops, although `function2()` also calls `function1()` every time it is called.

The next step is to compile the program using the `-pg` parameter for `gprof`. After that the program can be run:

```
$ gcc -o demo demo.c -pg
$ ./demo
$
```

When the program finishes, the `gmon.out` call graph profile file is created in the same directory. You can then run the `gprof` program against the demo program, and save the output to a file:

```
$ ls -al gmon.out
-rw-r--r--    1 rich     rich          426 Jul  7 12:39 gmon.out
$ gprof demo > gprof.txt
$
```

Notice that the `gmon.out` file was not referenced in the command line, just the name of the executable program. `gprof` automatically uses the `gmon.out` file located in the same directory. This example redirected the `gprof` output to a file named `gprof.txt`. The resulting file contains the complete `gprof` report for the program. Here's what the flat profile section looked like on my system:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 67.17    168.81   168.81    50000  3376.20  5023.11  function2
 32.83    251.32    82.51    50100  1646.91  1646.91  function1
```

This report shows the total processor time and times called for each individual function that was called by `main`. As expected, `function2` took the majority of the processing time.

The next report is the call graph, which shows the breakdown of time by individual functions, and how the functions were called:

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00  251.32                 main [1]
               168.81   82.35   50000/50000       function2 [2]
                 0.16    0.00     100/50100       function1 [3]
-----------------------------------------------
               168.81   82.35   50000/50000       main [1]
[2]     99.9  168.81   82.35   50000          function2 [2]
                82.35    0.00   50000/50100       function1 [3]
-----------------------------------------------
                 0.16    0.00     100/50100       main [1]
                82.35    0.00   50000/50100       function2 [2]
[3]     32.8   82.51    0.00   50100          function1 [3]
-----------------------------------------------
```

Each section of the call graph shows the function analyzed (the one on the line with the index number), the functions that called it, and its child functions. This output is used to track the flow of time throughout the program.

A Complete Assembly Development System

Now that you know all the pieces needed for an assembly language development environment, it's time to put them all together. One of the best environments for using GNU utilities is the Linux operating system. Many freely available Linux distributions contain all of the GNU utilities presented in this chapter already installed. This section describes some of the basics of the Linux system, along with the GNU utilities necessary for creating an assembly language development environment.

The basics of Linux

If you are new to the Linux environment, you may need some background information before trying out a Linux distribution. When people talk about the Linux operating system, they are really talking about an entire suite of programs, not all of them necessarily related to Linux. The key to building a Linux