Essential Principles of Computer Organization and Assembly Language (Working Title)

Patrick Juola

August 19, 2004
Contents

0 Preface
  0.1 Statement of Aims ............................................. 11
    0.1.1 What ...................................................... 11
    0.1.2 How ...................................................... 12
    0.1.3 For whom ............................................... 13
  0.2 Acknowledgements ............................................ 14

1 Computation and Representation ................................... 15
  1.1 Computation .................................................. 15
    1.1.1 Electronic devices ........................................ 15
    1.1.2 Algorithmic machines ...................................... 16
    1.1.3 Functional components ..................................... 17
  1.2 Digital and Numeric Representations ............................ 24
    1.2.1 Digital representations and bits .......................... 24
    1.2.2 Boolean logic ............................................. 26
    1.2.3 Bytes and words .......................................... 27
    1.2.4 Representations ........................................... 28
  1.3 Virtual Machines ............................................... 43
    1.3.1 What is a “virtual machine”? ............................. 43
    1.3.2 Portability concerns ....................................... 45
    1.3.3 Transcending limitations .................................. 45
    1.3.4 Ease of updates .......................................... 46
    1.3.5 Security concerns ......................................... 47
    1.3.6 Disadvantages of a virtual machine ....................... 48
  1.4 Programming the JVM ........................................... 49
    1.4.1 Java : what the JVM isn’t ................................. 49
    1.4.2 Translations of the sample program ....................... 51
    1.4.3 High- and low-level languages ............................ 52
    1.4.4 The sample program as the JVM sees it .................... 53

3
2 Arithmetic Expressions

2.1 Notations .......................... 61
  2.1.1 Instruction sets .................. 61
  2.1.2 Operations, operands, and ordering .................. 62
  2.1.3 Stack-based calculators ................. 63
2.2 Stored-Program Computers .................. 64
  2.2.1 The Fetch-Execute cycle .................. 64
  2.2.2 CISC vs. RISC computers ................. 67
2.3 Arithmetic Calculations on the JVM .................. 69
  2.3.1 General comments .................. 69
  2.3.2 A sample arithmetic instruction set ................. 70
  2.3.3 Logical operations .................. 72
  2.3.4 Stack manipulation operations ................. 74
  2.3.5 Assembly language and machine code .................. 76
  2.3.6 Illegal operations .................. 78
2.4 An Example Program .................. 79
  2.4.1 A annotated example .................. 79
  2.4.2 The final JVM code ................. 81
2.5 JVM Calculation Instructions Summarized .................. 82
2.6 Chapter Review .................. 83
2.7 Exercises .................. 84
2.8 Programming Exercises .................. 85

3 Assembly Language Programming in jasmin 87

3.1 Java, the programming system ................. 87
3.2 Using the Assembler .................. 89
  3.2.1 The assembler .................. 89
  3.2.2 Running a program .................. 90
  3.2.3 Display to the console vs. a window ................. 90
  3.2.4 Using System.out and System.in ................. 94
3.3 Assembly Language Statement Types ................. 95
  3.3.1 Instructions and comments ................. 95
  3.3.2 Assembler directives ................. 97
  3.3.3 Resource directives ................. 98
3.4 Example : Random Number Generation ................. 99
  3.4.1 Generating pseudorandom numbers ................. 99
CONTENTS

6.4 Issues of Interfacing ........................................ 197
6.5 Chapter Review ............................................. 199
6.6 Exercises .................................................. 200

7  The PowerPC ................................................. 201
  7.1 Background ................................................ 201
  7.2 Organization and Architecture ............................ 203
    7.2.1 Central Processing Unit ............................ 203
    7.2.2 Memory ............................................. 204
    7.2.3 Devices and Peripherals ........................... 204
  7.3 Assembly Language ...................................... 205
    7.3.1 Arithmetic ......................................... 205
    7.3.2 Floating point operations ........................... 207
    7.3.3 Comparisons and condition flags .................... 208
    7.3.4 Data movement ..................................... 208
    7.3.5 Branches ............................................ 209
  7.4 Conical mountains revisited .............................. 210
  7.5 Memory Organization and Use ............................ 211
  7.6 Performance Issues ..................................... 213
    7.6.1 Pipelining ......................................... 213
  7.7 Chapter Review ........................................... 216
  7.8 Exercises ............................................... 216

8  The Intel Pentium ............................................ 219
  8.1 Background ................................................ 219
  8.2 Organization and Architecture ............................ 220
    8.2.1 The Central Processing Unit ...................... 220
    8.2.2 Memory ............................................. 221
    8.2.3 Devices and Peripherals ........................... 221
  8.3 Programming .............................................. 222
    8.3.1 Operations and Addressing ......................... 222
    8.3.2 Advanced operations ............................... 222
    8.3.3 Instruction formats ................................ 224
  8.4 Memory Organization and Use ............................ 224
    8.4.1 Memory management ................................ 224
  8.5 Performance Issues ..................................... 225
    8.5.1 Pipelining ......................................... 225
    8.5.2 Parallel operations ............................... 226
    8.5.3 Superscalar architecture ........................... 227
  8.6 RISC vs. CISC revisited ................................ 227
## CONTENTS

8.7 Chapter Review ........................................... 228
8.8 Exercises ............................................. 229

9 Microcontrollers: The Atmel AVR ........................ 231
  9.1 Background ............................................ 231
  9.2 Organization and Architecture ........................ 232
    9.2.1 Central Processing Unit ......................... 232
    9.2.2 Memory ........................................... 233
    9.2.3 Devices and Peripherals ......................... 237
  9.3 Assembly Language ................................... 238
  9.4 Memory Organization and Use ........................ 239
  9.5 Issues of Interfacing ................................. 241
    9.5.1 Interfacing with external devices ............... 241
    9.5.2 Interfacing with timers .......................... 242
  9.6 Designing an AVR Program ............................ 243
  9.7 Chapter Review ....................................... 245
  9.8 Exercises ............................................ 246

10 Advanced Programming Topics on the JVM ............ 247
  10.1 Complex and derived types .......................... 247
    10.1.1 The need for derived types .................... 247
    10.1.2 An example derived type: arrays ............... 248
    10.1.3 Records: classes without methods .............. 255
  10.2 Classes and Inheritance ............................. 257
    10.2.1 Defining classes ................................ 257
    10.2.2 A sample class: String .......................... 258
    10.2.3 Implementing a String ......................... 260
  10.3 Class Operations and Methods ....................... 261
    10.3.1 Introduction to class operations ............... 261
    10.3.2 Field operations ................................ 261
    10.3.3 Methods .......................................... 263
    10.3.4 A taxonomy of classes ......................... 266
  10.4 Objects .............................................. 268
    10.4.1 Creating objects as instances of classes ....... 268
    10.4.2 Destroying objects .............................. 271
    10.4.3 The type Object .................................. 271
  10.5 Class Files and .class File Structure .............. 271
    10.5.1 Class files ...................................... 271
    10.5.2 Starting up classes ............................. 272
  10.6 Class Hierarchy Directives .......................... 273
Chapter 0

Preface

0.1 Statement of Aims

0.1.1 What

This is a book on the organization and architecture of the Java Virtual Machine, the software at the heart of the Java language and is found inside most computers, web browsers, PDAs, and networked accessories. It also covers general principles of machine organization and architecture, with illustrations from other popular (and not-so-popular) computers.

It is not a book on Java, the programming language, although some knowledge of Java or a Java-like language (C, C++, Pascal, Algol, et cetera) may be helpful. Instead, it is a book about how the Java language actually causes things to happen and computations to occur.

This book got its start as an experiment in modern technology. When I started teaching at my present university (1998), the organization and architecture course focused on the 8088 running MS-DOS — essentially a programming environment as old as the sophomores taking the class. (This temporal freezing is unfortunately fairly common; when I took the same class during my undergraduate days, the computer whose architecture I studied was only two years younger than I was.) The fundamental problem is that the modern Pentium 4 chip isn’t a particularly good teaching architecture; it incorporates all the functionality of the twenty-year old 8088, including its limitations, and then provides complex workarounds. Because of this complexity issue, it is difficult to explain the workings of the P4 without detailed reference to long out-dated chip sets, and textbooks have instead focused on the simpler 8088, then described the computers students actually use later, as an extension and afterthought. This is analogous, in my
mind, to learning automotive mechanics on a Ford Model A, and only in
the later chapters discussing such important concepts as catalytic conver-
ters, automatic transmissions, and key-based ignition systems. A course in
architecture should not automatically be forced to be a course in the history
of computing.

Instead, I wanted to teach a course using an easy-to-understand archi-
tecture that incorporated modern principles and could be useful in itself for
students to know. Since every computer that runs a web browser incor-
porates a copy of the JVM as software, almost every machine in existence
today already has a compatible JVM available to it.

This book, then, covers the central aspects of computer organization and
architecture: digital logic and systems, data representation, and machine
organization/architecture. It will also cover the assembly-level language of
one particular architecture, the Java Virtual Machine, with other common
architectures such as the Intel P4 and the PowerPC given as supporting
examples but not as the object of focus. It is designed specifically as a
textbook for a standard second-year course on “the architecture and organi-
zation of computers,” as recommended by the IEEE Computer Society and
the Association for Computing Machinery.¹

0.1.2 How

The book is structured in two broad tracks: the first half (chapters 1-5)
covers general principles of computer organization and architecture and the
art/science of programming in assembly language, using the JVM as an illus-
trative example of those principles in action (How are numbers represented
in a digital computer? What does the “loader” do? What is involved in
format conversion?), as well as the necessary specifics of JVM assembly lan-
guage programming, including detailed discussion of opcodes (what exactly
does the i2c opcode do, and how does it change the stack? What’s the
command to run the assembler?). The second half (chapters 6-10) focuses
on specific architectural details for a variety of different CPUs, including
the Pentium, its archaic and historic cousin the 8088, the PowerPC, and the
Atmel AVR as an example of a typical embedded-systems controller chip.

¹“Computing Curricula 2001.” Dec. 15, 2001 Final Draft; see specifically their recom-
mandation for course CS220.
0.1. STATEMENT OF AIMS

0.1.3 For whom

It is my hope and belief that this framework will permit this textbook to be used by a wide range of people and courses. This book should successfully serve most of the software-centric community. For those primarily interested in assembly language as the basis for abstract study on computer science, the JVM provides a simple and easy to understand introduction to the fundamental operations of computing. As the basis for a compiler theory, programming languages, or operating systems class, the JVM is a convenient and portable platform and target architecture, more widely available than any single chip or operating system. And as the basis for further (platform-specific) study on individual machines, the JVM provides a useful and explanatory teaching architecture that allows for a smooth and principled transition not only to today’s Pentium, but also to other architectures that may replace, supplant, or support the Pentium in the future. To the student interested in learning about how machines work, this textbook will provide them with information on a wide variety of platforms, enhancing their ability to use whatever machines and architectures they find in the work environment.

As alluded to above, the book is mainly intended for a single-semester course for second-year undergraduates. The first four chapters are core material central to the understanding of the principles of computer organization, architecture, and assembly language programming. They assume some knowledge of a high-level imperative language and familiarity with high-school level algebra (but not calculus). After that, professors (and students) have a certain amount of flexibility to pick and choose among the topics, depending upon environment and issues. For Intel/Windows shops, the chapters on the 8088 and Pentium are useful and relevant, while for schools with Apples, the PowerPC chapter is more relevant. The Atmel chapter can lay the groundwork for laboratory work in an embedded systems or microcomputer laboratory, while the advanced JVM topics would be of interest to students planning on implementing JVM-based systems or on writing system software (compilers, interpreters, and so forth) based on the JVM architecture. A fast-paced class might even be able to cover all topics. The appendices are there primarily for reference, since I believe that a good textbook should be useful even after the class is over.
0.2 Acknowledgements

Without the students at Duquesne University, and particularly my guinea pigs from the Computer Organization and Assembly Language class, this textbook couldn’t have happened. Similarly, I am grateful for the support given to me by my department, college, and university, and particularly for the support funding from the Philip H. and Betty L. Wimmer Family Foundation. I would also like to thank my readers, especially Erik Lindsley of the University of Pittsburgh, for their helpful comments on early drafts.

Without a publisher, this book would never have seen daylight; I would therefore like to acknowledge my editor, Kate Hargett, and through her the Prentice-Hall publishing group and the helpful anonymous reviewers whose identities she alone knows, but whose suggestions everyone indirectly appreciates. Similarly, without the software, this book wouldn’t exist – aside from the obvious debt of gratitude to the people at Sun who invented Java, I specifically would like to thank and acknowledge Jon Meyer, the author of jasmin, both for his software and his helpful support.

Finally, I would like to thank my wife Jodi, who has managed to put up with me through the book’s long gestation and is still willing to live in the same house.
Chapter 1

Computation and Representation

1.1 Computation

1.1.1 Electronic devices

How many people really know what a computer is? If you asked most people what a computer was, they would point you at a set of boxes on someone’s desk (or perhaps in someone’s briefcase) — probably a set of dull-looking rectangular boxes, encased in grey or beige plastic, and surrounded by a tangle of wires and perhaps a TV-looking thing. If pressed for detail, they would point at one particular box as being “the computer.” But, of course, there are also computers hidden in all sorts of everyday electronic gadgets, to make sure that your car’s fuel efficiency stays high enough, to
interpret the signals coming off a DVD player, and possibly even to make sure your morning toast is the right shade of brown. To most people, though, a computer is still the box you buy at an electronics shop, with bits and bytes and gigahertzes that are often compared, but rarely understood.

In functional terms, a computer is simply a high-speed calculator, capable of performing thousands, millions, or even billions of simple arithmetic operations per second from a stored program. Every thousandth of a second or so, the computer in your car reads a few key performance indications from various sensors in the engine, and adjusts the machine slightly to insure proper functioning. The key to being of any use is at least partially in the sensors. The computer itself processes only electronic signals. The sensors are responsible for determining what’s really going on under the hood and converting that into a set of electronic signals that describe, or represent, the current state of the engine. Similarly, the adjustments that the computer makes are stored as electronic signals and converted back into physical changes in the engine’s working.

How can electronic signals “represent” information? And how exactly does a computer process these signals to get the sort of fine control, without any physical moving parts or representation? Questions of representation such as these are, ultimately, the key to understanding both how computers work and how they can be deployed in the physical world.

1.1.2 Algorithmic machines

The single most important concept to the operation of a computer is the concept of an algorithm; an unambiguous, step-by-step process for solving a problem or achieving a desired end. The ultimate definition of a computer does not rely on its physical properties, or even on its electrical properties (such as its transistors), but upon its abilities to represent and carry out algorithms from a stored program. Within the computer are millions of tiny circuits, each of which will perform a specific, well-defined task (such as adding two integers together, or causing an individual wire or set of wires to become energized) when called upon. Most people who use or program computers are not aware of the detailed workings of these circuits.

In particular, we can describe several basic types of operations that a typical computer can perform. As computers are, fundamentally, merely calculating machines, almost all of the functions they can perform are related to numbers (and concepts representable by numbers). A computer can usually perform basic mathematical operations such as addition and division. It can also perform basic comparisons — is one number equal to
another number? Is the first number less that the second? It can store
millions or billions of pieces of information and retrieve them individually.
Finally, it can adjust its actions based on the information retrieved and the
comparisons performed. If the retrieved value is greater than the previous
value, then (for example) our engine is running too hot, and a signal should
be sent to adjust the engine performance.

1.1.3 Functional components

System-level description

Almost any college bulletin board has a few ads that read something like :
“GREAT MACHINE! 1.2GHz Intel Celeron, 128MB, 40GB hard drive, 15-
inch monitor, must sell to make car payment!” Like most ads, there’s a fair
bit of information in there that requires extensive unpacking to understand
fully. For example, what part of a 15-inch monitor is actually fifteen inches?
(The length of the diagonal of the visible screen, oddly enough.) In order to
understand the detailed workings of a computer, we first must understand
the major components and their relations to each other.

Central Processing Unit

The heart of any computer is the so-called Central Processing Unit, or CPU.
This is usually a single piece of high-density circuitry built onto a single inte-
grated circuit (IC) silicon chip. Physically, it usually looks like a small piece
of silicon, mounted on a plastic slab a few centimeters square, surrounded
by metal pins. The plastic slab itself is mounted on the motherboard, an
electronic circuit board, a piece of plastic and metal tens of centimeters on a
side, containing the CPU and a few other components that need to be placed
near the CPU for speed and convenience. Electronically, the CPU is the ul-
timate controller of the computer as well as where all actual calculations are
performed. And, of course, linguistically, it’s the part of the computer that
everyone talks and writes about — a 3.60 GHz Pentium 4 computer, like the
Hewlett-Packard HP xw4200, is simply a computer whose CPU is a Pentium
4 chip, and that runs at a speed of 3.60 gigahertz (GHz), or 3,600,000,000
machine cycles per second. Most of the basic operations a computer can per-
form take one machine cycle each, so another way of describing this is that
a 3.60 GHz computer can perform just over three and a half billion basic
operations per second. At the time of writing, 3.60 GHz is a fast machine,
but this changes very quickly with technological developments. For exam-
ple, in 2000, a 1.0 GHz Pentium was the state-of-the-art, and, in keeping
Figure 1.1: Block architecture of a simple computer
1.1. COMPUTATION

with a long-standing rule of thumb (Moore’s law\textsuperscript{1}) that computers double in computing power every eighteen months, one can confidently predict the wide availability of 8GHz CPUs by 2005 or 2006.

CPUs can usually be described in families of technological progress; the Pentium 4, for example, is a further development of the Pentium, the Pentium II, and the Pentium III, all manufactured by the Intel corporation. Before that, the Pentium itself derived from a long line of numbered Intel chips; starting with the Intel 8088 and progressing through the 80286, 80386, and 80486. The so-called X86 series became the basis for the wide selling IBM PC’s (and their clones) and is probably the most widely used CPU chip. Modern Apple computers use a different family of chips, the PowerPC G3 and G4, manufactured by a consortium between Apple, IBM, and Motorola (AIM). Older Apples and Sun workstations used chips from the Motorola-designed 68000 family.

The CPU itself can be divided into two or three main functional com-

\textsuperscript{1}SIDEBAR: MOORE’S LAW. Gordon Moore, the cofounder of Intel, observed in 1965 that the number of transistors that could be put on a chip was doubling every year. In the 1970s, that pace slowed down slightly, to a doubling every eighteen months, but has been remarkably uniform since then, to the surprise of almost everyone, including Dr. Moore himself.

The implications of smaller transistors (and increasing transistor density) are profound. First, the cost per square inch of a silicon chip itself has been relatively steady by comparison, so doubling the density will approximately halve the cost of a chip. Second, smaller transistors react faster, and components can be placed closer together, meaning that they can communicate with each other faster, vastly increasing the speed of the chip. Smaller transistors also consume less power, meaning longer battery life and lower cooling requirements, avoiding the need for climate-controlled rooms and bulky fans. Because more transistors can be placed on a chip, less soldering is needed to connect chips together, with accordingly reduced chance of solder breakage and correspondingly greater overall reliability. Finally, the fact that the chips are smaller means that computers themselves can be made smaller, enough to make things like embedded controller chips and/or PDA’s practical. It is hard to overestimate the effect that Moore’s Law has had on the development of the modern computer. Moore’s law by now is generally taken to read, more simply, that the power of an available computer doubles every eighteen months (for whatever reason, not just transistor density). A standard, even low-end, computer available off-the-shelf at the local store is faster, more reliable, and has more memory than the original Cray-1 supercomputer of 1973.

The problem with Moore’s law is that it can’t be followed forever. Eventually, the laws of physics are likely to dictate that a transistor can’t be any smaller than an atom (or something like that). More worrisome is what’s sometimes called “Moore’s second law,” that fabrication costs double every three years. As long as fabrication costs grow more slowly than computer power, the performance/cost ratio should remain reasonable. But the cost of investing in new chip technologies may make it difficult for manufacturers such as Intel to continue investing in new capital.
components. The Control Unit is responsible for moving data around within the machine. For example, the Control Unit takes care of loading individual program instructions from memory, identifying individual instructions, and passing the instructions to other appropriate parts of the computer to be performed. The Arithmetic and Logical Unit (ALU) performs all necessary arithmetic for the computer; it typically contains special-purpose hardware for addition, multiplication, division, and so forth. It also, as the name implies, performs all the logical operations, telling whether a given number is bigger or smaller than another number, or checking whether two numbers are equal. Some computers, particularly older computers, have special-purpose hardware, sometimes on a separate chip from the CPU itself, to handle operations involving fractions and decimals. This special hardware is often called the Floating Point Unit or FPU (also called the Floating Point Processor or FPP). Other computers fold the FPU hardware onto the same CPU chip as the ALU and the Control Unit, but the FPU can still be thought of as a different module within the same set of circuitry.

Memory

Both the program to be executed and its data are stored in memory. Conceptually, memory can be regarded as a very long array or row of electromagnetic storage devices. These array locations are numbered, from 0 to a CPU-defined maximum, and can be addressed individually by the Control Unit to put data into memory or to retrieve data from memory. In addition, most modern machines support the ability of high-speed devices such as disk drives to copy large blocks of data without needing the intervention of the Control Unit for each signal. Memory can be broadly divided into two types: Read-Only Memory (ROM), which is permanent, unalterable, and remains even after the power is switched off, and Random Access Memory (RAM), the contents of which can be changed by the CPU for temporary storage, but usually goes away when the power does. Many machines have both kinds of memory; the ROM holds standardized data and a basic version of the operating system that can be used to start the machine up. More extensive programs are stored in RAM and loaded as needed from long-term storage such as disk drives and CDs.

This simplified description deliberately hides some tricky aspects of memory that the hardware and operating system usually take care of for the user. (These issues also tend to be hardware-specific, so they will be dealt with in more detail in later chapters.) For example, different computers, even with identical CPUs, often have different amounts of memory. The amount
of physical memory installed on a computer may be less than the maximum number of locations the CPU can address, or in odd cases, may even be more. Furthermore, memory located on the CPU chip itself is typically much faster to access than memory located on separate chips, so a clever system can try to make sure that data is moved or copied as necessary to be available in the fastest memory when needed.

Input/Output (I/O) peripherals

In addition to the CPU and memory, a computer usually contains other devices to read, display or store data, or more generally to interact with the outside world. These devices vary from commonplace keyboards and hard drives through more unusual devices like facsimile (FAX) boards, speakers, and musical keyboards to downright weird gadgets like chemical sensors, robotic arms, and security deadbolts. The general term for these is peripherals. For the most part, these devices have little direct effect on the architecture and organization of the computer itself — they are just sources and sinks for information. A keyboard, for instance, is simply a device for gathering information. From the point of view of the CPU designer, data is data, whether it came from the Internet, from the keyboard, or from a fancy chemical spectrum analyzer.

In many cases, a peripheral can be physically divided into two or more parts. For example, computers usually display their information to the user via some form of video monitor. The monitor itself is a separate device, connected via a cable to a video adapter board located inside the computer’s casing. The CPU can draw pictures by sending command signals to the video board, which in turn will generate the picture and send appropriate visual signals over the video cable to the monitor itself. A similar process describes how the computer can load a file from many different kinds of hard drive via a SCI (Small Computer System Interface) controller card, or interact via an Ethernet card with the millions of miles of wire that comprise the Ethernet. Conceptually, engineers will draw a distinction between the device itself, the device cable (which is usually just a wire) and the device controller, which is usually a board inside the computer — but to the programmer, they’re usually all one device. Using this kind of logic, the entire Internet, with all of its millions of wire, is just “a device.” With a suitably well-designed system, there’s not much difference between downloading a file off the Internet or loading it from a hard drive.
Interconnections and buses

In order for the data to move between the CPU, memory, and the peripherals, there must be connections. These connections, especially between separate boards, are usually groups of wires to allow for multiple individual signals to be sent in a block. The original IBM-PC, for example, had eight wires to allow data to pass between the CPU and peripherals. A more modern computer’s PCI (Peripheral Component Interconnect) bus has 64 data wires, allowing data to pass eight times as fast, even before increased computer speed is taken into account. These wires are usually grouped into what is called a bus, a single wire-set connecting several different devices. Because it is shared (like an antique-style party line telephone), only one device can transmit data at a time, but the data is available to any connected device. Additional wires are used to determine which device should be listening to the data, and what exactly it should do when it gets it.

In general, the more devices attached to a single bus, the slower it runs. This is for two main reasons — first, the more devices, the greater possibility that two devices will have data to transmit at the same time, and thus that one device will have to wait its turn. Second, “more devices” usually means longer wires in the bus, which reduces the speed of the bus due to propagation delays — the length of time it takes a signal to get from one end of the wire to the other. For this reason, many computers have gone to a multiple-bus design, where, for example, the local bus connects the CPU with high speed memory stored on the CPU’s motherboard. The system bus connects the memory board, the CPU motherboard, and an “expansion bus” interface board. The expansion bus, in turn, is a second bus that connects to other devices such as the network, the disk drives, the keyboard, and the mouse.

On particularly high-performance computers (such as the figure), there may be four or five separate buses, with one reserved for high-speed, data-intensive devices such as the network and video cards, while lower-speed devices such as the keyboard are relegated to a separate and slower bus.
Figure 1.2: Block architecture of a high-speed, multibus computer
Support units

In addition to the devices mentioned already, a typical computer will have a number of crucial components that are important to the physical aspects of the computer itself. For example, inside the case (itself crucial for the physical protection of the delicate circuit boards) will be a power supply that converts the AC line voltage into an appropriately conditioned DC voltage for the circuit boards. There may also be a battery, particularly in laptops, to provide power when wall current is unavailable and to maintain memory settings. There is usually a fan to circulate air inside the case and to prevent components from overheating. There may also be other devices such as heat sensors (to control fan speed), security devices to prevent unauthorized use or removal, and often several wholly internal peripherals such as internal disk drives and CD readers.

1.2 Digital and Numeric Representations

1.2.1 Digital representations and bits

![Diagram of diode structure and symbol](image)

Figure 1.3: Diodes

At a fundamental level, computer components, like so many other electronic components, come in two stable states. Lights are on or off, switches are open or closed, and wires are either carrying current or they aren’t. In the case of computer hardware, individual components such as transistors²

²SIDEBAR: HOW TRANSISTORS WORK. The single most important electrical component behind the modern computer is the transistor, first invented by Bardeen, Brattain, and Shockley in 1947 at Bell Telephone Labs. (These men received the Nobel
Physics Prize in 1956 for this invention.) The fundamental idea involves some fairly high-powered (well, yes, Nobel-caliber) quantum physics, but it can be understood in terms of electron transport, as long as you don’t need the actual equations. A transistor is mostly made of a type of material called a semiconductor, which occupies an uneasy middle ground between good conductors (like copper) and bad conductors/good insulators (like glass). A key aspect of semiconductors is that their ability to transmit electricity can change dramatically with impurities (dopants) in the semiconductor.

For example, the element phosphorus, when added to “pure” silicon (a semiconductor) will donate electrons to the silicon. Since electrons have negative charges, phosphorus is termed an n-type dopant, and phosphorus-doped silicon is sometimes called an n-type semiconductor. Aluminum, by contrast, is a p-type dopant and will actually remove — really, lock up — electrons from the silicon matrix. The spots where these electrons have been removed are sometimes called “holes” in the p-type semiconductor.

When you put a piece of n-type next to a piece of p-type semiconductor (the resulting widget is called a diode, there is an interesting electrical effect. An electrical current will not typically be able to pass through such a diode; the electrons carrying the current will encounter and “fall into” the holes. If you apply a bias voltage to this gadget, however, the extra electrons will fill the holes, allowing current to pass. This means that electricity can only pass in one direction through a diode, which makes it useful as an electrical rectifier.

A modern transistor is made like a semiconductor sandwich; a thin layer of p-type semiconductor between two slices of n-type semiconductor, or sometimes the other way around. (Yes, this is just two diodes back-to-back. See the diagram.) Under normal circumstances, current can’t pass from the emitter to the collector as electrons get fall into the holes. Applying a bias voltage to the base (the middle wire) will fill the holes so that electricity can pass. You can think of the base like a gate that can open or shut to allow electricity to flow or not — alternatively, you can think of it like a valve in a hosepipe to control the amount of water it lets through. Turn it one way, the electrical signal drops to a trickle. Turn it the other way, and it flows without hindrance.

The overall effect of the transistor is that a small change in voltage (at the base) will result in a very large change in the amount of current that flows from the emitter to the collector. This makes a transistor extremely useful for amplifying small signals. It also
and resistors are either at zero volts relative to ground, or at some other voltage (typically five volts above ground). These two states are usually held to represent the numbers one and zero, respectively. In the early stages of computer development, these values were hand-encoded by the flipping of mechanical switches. Today, high-speed transistors serve much the same purpose, but the representation of data in terms of these two values remains unchanged since the 1940s. Every such 1 or 0 is usually called a bit, an abbreviation for “binary digit.” (Of course, the word “bit” is itself a normal English word, meaning “a very small amount” — which also describes a “bit” of information.)

1.2.2 Boolean logic

A bit is the smallest unit that can be said to carry information, as in the children’s game of Twenty Questions, where each yes or no question yields an answer that could be encoded with a single bit (for example, 1 represents a “yes” and 0 a “no”). It is also the smallest unit that can be operated upon logically. The conventional way of performing logic on bit quantities is called Boolean logic, after the 19th-century mathematician George Boole. He identified three basic operations: AND, OR, and NOT, and defined their meaning in terms of simple changes upon bits. For example, the expression X AND Y is true (a “yes,” or a 1) if and only if, independently, X is a “yes” and Y is a “yes.” The expression X OR Y, conversely, is a “yes” if either X is a “yes” or Y is a “yes.” An equivalent way of stating this is that X OR Y is false (a “no,” or a 0) if and only if X is a “no” and Y is a “no.” The expression NOT X is the exact opposite of X: “yes” if X is a “no” and “no” if X is a “yes.” Because a bit can be in only one of two states, there are no other possibilities to consider. These operations (AND, OR, and NOT) can be nested or combined as needed. For example, NOT (NOT X) is the exact opposite of the exact opposite of X, which works out to be the same as X itself. These three operations parallel their English logical equivalents fairly well: if I want a cup of coffee “with milk and sugar,” logically what I am asking for is a cup of coffee where “with milk” is true AND “with sugar” is true. Similarly, a cup of coffee “without milk or sugar” is the same as a can function as a binary switch, with the key advantage that it has no moving parts, and thus nothing to break. (It’s also much, much faster to throw an electrical switch than a mechanical one.) With the invention of the integrated circuit (IC), for which Jack Kilby also won the Nobel prize, engineers gained the ability to create thousands, millions or billions of tiny transistors by doping very small areas of a larger piece of silicon. To this day, this remains the primary way that computers are manufactured.
1.2. DIGITAL AND NUMERIC REPRESENTATIONS

cup “with no milk and no sugar.” (Think about it for a bit.)

In addition to these three basic operations, there are a number of other operations that can be defined from them. For example, NAND is an abbreviation for NOT-AND. The expression X NAND Y refers to NOT (X AND Y). Similarly, X NOR Y refers to NOT (X OR Y). Another common expression is the exclusive-OR operation, written XOR. The expression X OR Y is true if X is true, Y is true, or both. By contrast, X XOR Y is true if X is true or Y is true, but not both. This difference is not captured cleanly in English, but is implicit in several different uses: for example, if I am asked if I want milk or sugar in my coffee, I’m allowed to say “yes, please,” meaning that I want both. This is the normal (inclusive) OR. By contrast, if I am offered coffee or tea, it wouldn’t make much sense for me to say “yes,” meaning both. This is an exclusive XOR, where I can have either coffee XOR tea, but not both at the same time.

From a strictly theoretical point of view, it doesn’t matter much whether 1/“yes”/true" is encoded as ground voltage or as five volts above ground as long as the two states are different and consistently applied. From the point of view of a computer engineer or system designer, there may be particular reasons to choose one representation over another. The choice of representation can have profound implications for the design of the chips themselves. The Boolean operations described above are usually implemented in hardware at a very low level on the chip itself. For example, one can build a simple circuit with a pair of switches (or transistors) that will allow current to flow only if both switches are closed. Such a circuit is called an AND gate, because it implements the AND function on the two bits represented by the switch state. This tiny circuit and others like it (OR gates, NAND gates, and so forth), copied millions or billions of times across the computer chip, are the fundamental building blocks of a computer. (See appendix A for more on how these blocks work.)

1.2.3 Bytes and words

For convenience, eight bits are usually grouped into a single block, conventionally called a byte. There are two main advantages to doing this. First, writing and reading a long sequence of zeros and ones is, for humans, tedious and error prone. Second, most interesting computations require more data than a single bit. If multiple wires are available, as in standard buses, then electrical signals can be moved around in groups, resulting in faster computation.

The next-largest named block of bits is a word. The definition and size
of a word is not absolute, but varies from computer to computer. A word is the size of the most convenient block of data for the computer to deal with. (Usually it’s, but not always, it’s the size of the bus — but see the Intel 8088, discussed later, for a counterexample.) For example, the Zilog Z-80 microprocessor (the chip underling the Radio Shack TRS-80, popular in the mid 1970s) had a word size of eight bits, or one byte. The CPU, memory storage, and buses had all been optimized to handle eight bits at a time (for example, there were eight data wires in the system bus). In the event that the computer had to process sixteen bits of data, it would be handled in two separate halves, while if the computer had only four bits of data to process, the CPU would work as though it had eight bits of data, then throw away the extra four (useless) bits. The original IBM-PC, based on the Intel 8088 chip, had a word size of 16 bits. More modern computers such as the Intel Pentium 4 or the PowerPC G4 have word sizes of 32 bits, and computers with word sizes of 64 bits or even larger, such as the Intel Itanium series or AMD Opteron series, are available. Especially for high-end scientific computation or graphics, such as in home video game consoles, a large word size can be key to delivering data fast enough to allow smoothly animated, high-detail graphics.

Formally speaking, the word size of a machine is defined as the size (in bits) of the machine’s registers. A register is the memory location inside the CPU where the actual computations, such as addition, subtraction, and comparisons, take place. The number, type, and organization of registers varies widely from chip to chip and may even change significantly within chips of the same family. The Intel 8088, for example, had four 16-bit general purpose registers, while the Intel 80386, designed seven years later, used 32-bit registers instead. Efficient use of registers is key to writing fast, well-optimized programs. Unfortunately, because of the differences between different computers, this can be one of the more difficult aspects of writing such programs for various computers.

1.2.4 Representations

Bit patterns are arbitrary

Consider, for a moment, one of the registers in an old-fashioned 8-bit microcomputer chip. How many different patterns can it hold? Other ways of asking the same question is to ask how many different ways you can arrange a sequence of ten pennies in terms of heads and tails, or how many strings can be made up of only 8 letters, each of which is a zero or a one.
Perhaps obviously, there are two possibilities for the first bit/coin/letter/digit, two for the second, and so on until we reach the eighth and final digit. There are thus \(2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2\) possibilities. This works out to \(2^8\) or 256 different storable patterns. A similar reasoning shows that there are \(2^{32}\) or just over four billion storable patterns in a 32-bit register. (All right, for the pedants, \(2^{32} = 4,294,967,296\). A handy rule of thumb for dealing with large binary powers is that \(2^{10}\), really \(2^{10} = 1024\), is “close to” 1000. Remember that to multiply numbers, you add the exponents: \(a^b \cdot a^c = a^{b+c}\). Thus, \(2^{32}\) is \(2^{2+10+10+10}\), or \(2^2 \cdot 2^{10} \cdot 2^{10} \cdot 2^{10}\), or about \(4 \cdot 1000 \cdot 1000 \cdot 1000\).)

Yes, but what do these patterns mean? The practical answer is: whatever you as the programmer want them to mean. As you’ll see in the next subsection, it’s fairly easy to read the bit pattern 0001101 as the number 13. It’s also possible to read it as a record of the answers to eight different yes/no questions. (“Are you married?” – No. “Are you older than 25?” – No. “Are you male?” – No. And so forth.) It could also represent a key being pressed on the keyboard. The interpretation of bit patterns is arbitrary, and computers can typically use the same patterns in many ways. Part of the programmer’s task is to make sure that the computer interprets these arbitrary and ambiguous patterns correctly at all times.

**Natural numbers**

A common way to interpret bit patterns is using binary arithmetic (in base 2). In conventional, or decimal (base 10) arithmetic, there are only ten different number symbols, 0 through 9. Larger numbers are expressed as individual digits times a power of the base value. The number four hundred eighty-one (481), for instance, is really \(4 \cdot 10^2 + 8 \cdot 10^1 + 1\). Using this notation, we can express all natural numbers up to nine hundred ninety-nine in only three decimal digits. Using a similar notation, but by adding up powers of two, we can express any number in binary using only zeros and ones.

Taking as an example the (decimal) number 85, simple arithmetic shows that it is equivalent to \(64 + 16 + 4 + 1\), or (in more detail) to \(1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1\). In binary, then, the number would be written as 1010101. In an 8-bit register, this could be stored as 01010101, while in a 32-bit register, this would be 00000000000000000000000000000000001010101.
CHAPTER 1. COMPUTATION AND REPRESENTATION

Figure 1.5: Addition and multiplication tables for binary (base 2) arithmetic

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>·</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

remember that, when adding in binary (base 2), every time the result is a two, it generates a carry (just like every ten generates a carry in base 10). So the result of adding 1 + 1 in base 2 is not 2, but 0, carry the 1 or 10.

Inspection of the tables reveals the fundamental connection between binary arithmetic and Boolean algebra. The multiplication table is identical to the AND of the two factors. Addition, of course, can potentially generate two numbers: a one-digit sum and a possible carry. There is a carry if and only if the first number is a 1 and the second number is a 1, or in other words, the carry is simply the AND of the two addends, while the sum (excluding the carry) is one if the first number is one or the second number is one, but not both: the XOR of the two addends. By building an appropriate collection of AND and XOR gates, the computer can add or multiply any numbers within the expressive power of the registers.

How large a number, then, can be stored in an 8-bit register? The smallest possible value is obviously 00000000, representing the number 0. The largest possible value, then, would be 11111111, the number 255. Any integer in this range can be represented easily as an 8-bit quantity. For 32-bit registers, the smallest value is still 0, but the largest value is just over 4.2 billion.

Although computers have no difficulty in interpreting long binary numbers, humans often do. For example, is the 32-bit number 00010000000000000001000000000000 the same as the number (deep breath here) 00010000000000000001000000000000? (No, they are different. There are sixteen zeros between the ones in the first number, and only fifteen in the second.) For this reasons, when it is necessary (rarely, one hopes) to deal with binary numbers, most programmers prefer to use hexadecimal (base 16) numbers instead. Since \(16 = 2^4\), every block of four bits (sometimes called a nybble) can be represented as a single base 16 “digit.” Ten of the 16 hexadecimal digits are familiar to us as the numbers 0 through 9, representing the patterns 0000 through 1001. Since our normal base 10 only uses ten digits, computer scientists have co-opted the letters A–F to represent the remaining patterns (1010, 1011, 1100, 1101, 1110, 1111; see table 1.1 for the complete conversion list). The two numbers
1.2. DIGITAL AND NUMERIC REPRESENTATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Hex</th>
<th>Binary</th>
<th>Hex</th>
<th>Binary</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
<td>1000</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>A</td>
<td>1010</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>B</td>
<td>1011</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

Table 1.1: Hexadecimal↔binary digit conversions

above are clearly different when converted to base 16:

\[
\begin{array}{cccccccc}
0001 & 0000 & 0000 & 0000 & 0000 & 1000 & 0000 & 0000 & = 0x10000800 \\
1 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & \\
0001 & 0000 & 0000 & 0000 & 0001 & 0000 & 0000 & 0000 & = 0x10001000 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \\
\end{array}
\]

By convention in many computer languages (including Java, C, and C++), hexadecimal numbers are written with an initial “0x” or “0X.” We follow that convention here, so the number 1001 refers to the decimal value one thousand one. The value 0x1001 would refer to \(16^3+1\), the decimal value four thousand ninety-seven. (Binary quantities will be clearly identified as such in the text. Also, on rare occasions, some patterns will be written as octal, or base 8, numbers. These numbers are written with a leading 0, so the number 01001 would be an octal value equivalent to 513. Note that 0 is still 0 (and 1 is still 1) in any base.

**Base conversions**

Converting from a representation in one base to another can be a tedious, but necessary, task. Fortunately, the mathematics involved is fairly simple. Converting from any other base into base 10, for example, is simple if you understand the notation. The binary number 110110, for instance, is defined to represent \(2^5 + 2^4 + 2^2 + 2^1, 32 + 16 + 4 + 2,\) or 54. Similarly, \(0x481\) is \(4 \cdot 16^2 + 8 \cdot 16^1 + 1, 1024 + 128 + 1,\) or (decimal) 1153.

An easier way to perform the calculation involves alternating multiplication and addition. The binary number 110110 is, perhaps obviously, twice the binary value of 11011. (If this isn’t obvious, notice that the base ten number 5280 is ten times the value of 528.) 11011 is, in turn, twice 1101 plus 1. Thus, one can simply alternate multiplying by the base value and adding the new digit. Using this system, binary 110110 becomes
CHAPTER 1. COMPUTATION AND REPRESENTATION

\[(((((1 \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 0)\]

which simple arithmetic will confirm is 54. Similarly 0x481 is

\[((4 \cdot 16) + 8) \cdot 16 + 1)\]

which can be shown to be 1153.

If alternating multiplication and addition will convert to base 10, then it stands to reason that alternating division and subtraction can be used to convert from base 10 to binary. The subtraction is actually implicit in the way we will be using division. When dividing integers by integers, it’s rather rare that the number comes out exact, and normally there’s a remainder that must be implicitly subtracted from the dividend. These remainders are exactly the base digits. Using 54 again as our example, the remainders generated when we repeatedly divide by two will generate the necessary binary digits.

\[
\begin{align*}
54 \div 2 &= 27r0 \\
27 \div 2 &= 13r1 \\
13 \div 2 &= 6r1 \\
6 \div 2 &= 3r0 \\
3 \div 2 &= 1r1 \\
1 \div 2 &= 0r1
\end{align*}
\]

The boldface numbers are, of course, the bits for the binary digits of 54 (binary 110110). The only tricky thing to remember is that, in the multiplication procedure, the digits are entered in the normal order (left-to-right), so in the division procedure, unsurprisingly, the digits come out in right-to-left order, backwards. The same procedure works for base 16 (or indeed for base 8, base 4, or any other base):

\[
\begin{align*}
1153 \div 16 &= 72r1 \\
72 \div 16 &= 4r8 \\
4 \div 16 &= 0r4
\end{align*}
\]

Finally, the most often used and perhaps the most important conversion is direct (and quick) conversion between base 2 and base 16, in either
1.2. DIGITAL AND NUMERIC REPRESENTATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Hex</th>
<th>Binary</th>
<th>Hex</th>
<th>Binary</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
<td>1000</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>A</td>
<td>1010</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>B</td>
<td>1011</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

Table 1.2: Hexadecimal→binary digit conversions (copy of table 1.1)

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

Table 1.3: Octal→binary digit conversions

direction. Fortunately, this is also the easiest. Because 16 is the fourth power of 2, multiplying by 16 is really just multiplying by $2^4$. Thus, every hexadecimal digit corresponds directly to a group of four binary digits. To convert from hexadecimal to binary, as discussed earlier, simply replace each digit with its four-bit equivalent. To convert binary to hex, break the binary number into groups of four bits (starting at the right) and perform the replacement in the other direction. The complete conversion chart is presented as table 1.1.

So, for example, the binary number 100101101100101 would be broken up into 4-bit nybbles, starting from the right, as 100 1011 0110 0101. (Please notice that I cheated: the number I gave you only has fifteen bits, so one group of 4 isn’t complete. This group will always be on the far left, and will be padded out with zeros, so the “real” value you will need to convert will be 0100 1011 0110 0101.) Looking these four values up in the table, they correspond to the values 4, B, 6, and 5. Therefore, the corresponding hexadecimal number is 0x4B65.

Going the other way, the hexadecimal number 0x18C3 would be converted to the four binary groups 0001 (1), 0100 (8), 1100 (C) and 0101 (3), which are put together to give the binary quantity 0001010011000101.

A similar technique would work for octal (base 8) with only the first two columns of table 1.1 and using only the last three (instead of four) bits of the binary entries, as shown in table 1.3. Using these notations and techniques, you can represent any nonnegative integer in a sufficiently large register, and interpret it in any base you like.
CHAPTER 1. COMPUTATION AND REPRESENTATION

Signed representations

In the real world, there is often a use for negative numbers. If the smallest possible value stored in a register is 0, how can a computer store negative values? The question, oddly enough, is not one of storage, but of interpretation. Although the maximum number of storable patterns is fixed (for a given register size), the programmer can opt instead to interpret some patterns as meaning negative values. The usual method for doing this is to use an interpretation known as two’s-complement notation. Numbers in which the first bit is a zero are interpreted as positive numbers (or zero), while numbers in which the first bit is a one are interpreted as negative. For example, the number 13 would be written in (8-bit) binary as 00001101 (hexadecimal 0x0D), while the number -13 would be 11110011 (0xF3). These patterns are called signed numbers, as opposed to the previously defined unsigned numbers.

How do we get from 0xF3 to -13? Beyond the leading one, there appears to be no similarity between the two representations. The connection is a rather subtle one, based on the definition of negative numbers as the inverses of positive numbers. In particular, 13 + -13 should equal 0. Using the binary representations above, we note that

\[
\begin{align*}
00001101 & \quad 13 \\
+ \quad 11110011 & \quad -13 \text{ in two’s-complement notation} \\
\hline
100000000 & \quad 0 \text{ plus an overflow/carry}
\end{align*}
\]

However, the nine-bit quantity 100000000 (0x100) cannot be stored in only an 8-bit register! Just like a car odometer that rolls over when the mileage becomes too great, an 8-bit register will overflow and lose the information contained in the ninth bit. The resulting stored pattern is therefore 00000000 or 0x00, which is the binary (and hex) equivalent of 0. Using this method, we can see that the range of values stored in an 8-bit register will vary from -128 (0x80) to +127 (0x7F). Approximately half the values are positive, and half the values are negative, which in practical terms is about what people typically want.

This demonstration relies critically on the use of an 8-bit register. In a 32-bit register, a much larger value is necessary to produce overflow and wrap around to zero. The 32-bit two’s-complement representation of -13 would not be 0xF3, but 0xFFF3. In fact, viewed as a 32-bit number, 0xF3 would normally be interpreted as 0x00F3, which isn’t even negative at all, since the first bit isn’t a one.
Calculating the two’s-complement representation of a number (for a
given fixed register size) by hand is not difficult. Notice first that the rep-
resentation of -1, for any register size, is always going to be a register con-
taining all ones. Adding one to this number will produce overflow and a
register full of zeros. For any given bit pattern, if you reverse every individ-
ual bit (each one becomes a zero, each zero becomes a one, while preserving
the original order — this operation is sometimes called the bitwise NOT,
because it’s applying a NOT to every individual bit), and add the resulting
number to the original, the result will always give you a register full of ones.
(Why?) This reversed pattern (sometimes called the one’s-complement or
just the complement), added to the original pattern, will yield a sum of -1.
This reversed pattern plus one, then, will give you the two’s-complement of
the original number.

\[
\begin{array}{c}
00001101 \quad (= 13) \\
11110010 \quad \text{(reversed)} \\
+ \quad 1 \\
\hline
11110011 \quad (= -13)
\end{array}
\]

Note that repeating the process a second time will reverse the reversal.

\[
\begin{array}{c}
11110011 \quad (= -13) \\
00001100 \quad \text{(reversed)} \\
+ \quad 1 \\
\hline
00001101 \quad (= 13)
\end{array}
\]

This process will generalize to any numbers and any (positive) register
size. Subtraction will happen in the exact same way, since subtracting a
number is exactly the same as adding its negative.

**Floating point representation**

In addition to representing signed integers, computers are often called upon
to represent fractions or quantities with decimal points. To do this, they
use a modification of standard “scientific notation” based on powers of two
instead of powers of ten. These numbers are often called floating point num-
bers, because they contain a decimal point that can float around, depending
upon the representation.
Starting with the basics, it’s readily apparent that any integer can be converted into a number with a decimal point, just by adding a decimal point and a lot of zeros. For example, the integer 5 is also 5.0000..., the number -22 is also -22.0000..., and so forth. This is also true for numbers in other bases (except that technically the “decimal” point refers to base 10; in other bases, it would be called a “radix” point). So the (binary) number 1010 is also 1010.0000..., while the (hexidecimal) number 0x357 is also 0x357.0....

Any radix point number can also be written in “scientific notation” by shifting the point around and multiplying by the base a certain number of times. For example, Avogadro’s number is usually approximated as $6.023 \cdot 10^{23}$. Even students who have forgotten its significance in chemistry should be able to interpret the notation — Avogadro’s number is a 24-digit (23+1) number whose first four digits are 6, 0, 2, 3, or about 602,300,000,000,000,000,000. Scientific notation as used here has three parts: the base (in this case, 10), the exponent (23), and the mantissa (6.023). To interpret the number, one raises the base to the power of the exponent, then multiplies by the mantissa. Perhaps obviously, there are lots of different mantissa/exponent sets that would produce the same number; Avogadro’s number could also be written as $6023 \cdot 10^{20}$, $60.23 \cdot 10^{22}$, or even $0.6023 \cdot 10^{24}$.

Computers can use the same idea, but using binary representations. In particular, note the patterns in table 1.4. Multiplying the decimal number by two shifts the representation one bit to the left, while dividing by two shifts the pattern one bit to the right. Alternatively, a form of scientific notation applies where the same bit pattern can be shifted.

We extend this to represent non-integer floating point numbers in binary the usual way, as expressed in table 1.5. The number 2.5, for example, being exactly half of 5, could be represented as the binary quantity 10.1, or the binary quantity 1.01 times $2^1$. 1.25 would be (binary) 1.01 or 1.01 times $2^0$. Using this sort of notation, any decimal floating point quantity has an equivalent binary representation.
Decimal  |  Binary   | Binary real  | Scientific notation  
--- | --- | --- | ---  
5      | 101    | 00101.000... | $1.0100_{(binary)} \cdot 2^2$  
2.5    | 0010.100... | $1.0100_{(binary)} \cdot 2^1$  
1.25   | 00001.010... | $1.0100_{(binary)} \cdot 2^0$  
0.625  | 00000.101... | $1.0100_{(binary)} \cdot 2^{-1}$  

Table 1.5: Exponential notation with binary fractions

The Institute for Electrical and Electronic Engineers (IEEE) has issued a series of specification documents describing standardized ways to represent floating point numbers in binary. One such standard, IEEE 754-1985, describes a method of storing floating point numbers into 32 bit words as follows:

Number the 32 bits of the word starting from bit 31 at the left down to bit 0 at the right. The first bit, (bit 31) is the sign bit, which (as before) tells whether the number is positive or negative.

The next eight bits (bits 30–23) are a “biased” exponent. It would make sense to use the bit pattern 0000000 to represent $2^0$, 0000001 to represent $2^1$, but then one couldn’t represent small values like $2^{-3}$. It would also make sense to use 7-bit two’s complement notation, but that’s not what the IEEE chose. Instead, the IEEE specified the use of the unsigned numbers 0..255, but the number stored in the exponent bits (the representational exponent) is actually 127 higher than the “real” exponent. In other words, a real exponent of zero will be stored as a representational exponent of 127 (binary 01111111). A real exponent of one would be stored as 128, (binary 10000000), and a stored exponent of 00000000 would actually represent the tiny quantity $2^{-127}$. The remaining 23 bits (bits 22–0) are the mantissa, with the decimal point — technically called the radix point, since we’re no longer dealing with decimal — placed conventionally immediately after the first binary digit.

Thus, for normal numbers, the value stored in the register is the value

$$(-1)^{\text{signbit}} \cdot \text{mantissa} \cdot 2^{\text{real exponent}+127}$$

An representational exponent of 127, then, would mean that the mantissa is multiplied by 1 (a corresponding real exponent of 0, hence a multiplier of $2^0$), while an exponent of 126 would mean the fraction is multiplied by $2^{-1}$ or 0.5, and so forth.

Actually, there is a micro-lie in the above equation. Because the numbers are in binary, the first non-zero digit has to be a one (there aren’t any other
choices that aren’t zero!). Since we know that the first digit is a one, we can leave it out and use the space freed up to store another digit whose value we didn’t know beforehand. So the real equation would be

\[ (-1)^{\text{signbit}} \cdot 1.\text{mantissa} \cdot 2^{\text{real exponent} + 127} \]

As a simple example, the number 2.0, in binary, is 1.0\cdot2^1. Representing this as an IEEE floating point number, the sign bit would be 0 (a positive number), the mantissa would be all zeros (and an implicit leading one), while the exponent would be 127+1, or 128, or binary 10000000. This would be stored in the 32-bit quantity

\[
\begin{array}{cccc}
0 & 10000000 & 00000000000000000000000000000000 \\
\text{sign exponent mantissa}& & & \\
\end{array}
\]

This same bit pattern could be written as the hexadecimal number 0x40000000.

The number -1.0, on the other hand, would have a sign bit of 1, an exponent of 127 + 0, or binary 01111111, and a mantissa of all zeros (plus an implicit leading one). This would yield the following 32-bit quantity:

\[
\begin{array}{cccc}
1 & 01111111 & 00000000000000000000000000000000 \\
\text{sign exponent mantissa}& & & \\
\end{array}
\]

Another way of writing this bit pattern would be 0xBF800000.

Of course, if the number to be stored is zero exactly (0.000...), then there is no implicit leading one at any exponent. The IEEE defined as a special case that a bit pattern of all zeros (sign, exponent, and mantissa) would represent the number 0.0. There are also a number of other special cases, including representations of both positive and negative infinity, and the so-called NaN (not a number, the number that results when you try an illegal operation such as taking the logarithm of a negative number). The IEEE has also defined standard methods for storing numbers more accurately in 64-bit and larger registers. These additional cases are similar in spirit, if not in detail, to the 32-bit standard described above, but the details are rather dry and technical and not necessarily of interest to the average programmer.

One problem that comes up in any kind of radix-based representation is the issue of numbers that don’t represent exactly. Even without worrying about irrational numbers (like \(\pi\)), some simple fractions can’t be represented exactly. In base 10, for example, the fraction 1/7 has the approximate value 0.14285714285714... but never comes to an end. The fraction 1/3 is similarly 0.33333... In base 2, the fraction 1/3 is 0.010101010101010101... But
there’s no way to fit an infinite sequence into only 23 mantissa bits. So the solution is: we don’t.

Instead, the number would be represented as closely as possible. Converting to radix point, we see that \(1/3\) is about equal to \(1.010101010101010101010\cdot2^{-2}\), represented as

\[
\begin{array}{ccc}
\text{sign} & \text{exponent} & \text{mantissa} \\
0 & 01111101 & 01010101010101010101010 \\
\end{array}
\]

This isn’t a perfect representation, but it’s close. But “close” may not be good enough in all contexts; small errors (called roundoff error) will inevitably creep into calculations involving floating point numbers. If I multiplied this number by the value 3.0 (\(1.1\cdot2^1\)), I’d be very unlikely to get the exact value 1.0 back. Programmers, especially the ones that do big numerical problems like matrix inversion, spend lots of time trying to minimize this sort of error. For now, all we can do is be aware of it.

Performing operations on floating point numbers is tricky (and often slow). Intuitively, the algorithms are understandable and very similar to the algorithms you already know for manipulating numbers in scientific notation. Multiplication is relatively easy, since we know that \(2^x\) times \(2^y\) is just \(2^{x+y}\). Therefore, the product of two floating point numbers has as its sign bit the XOR of the two signs, as its mantissa the product of the two mantissae, and as its exponent the sum of the two exponents\(^3\). Thus we have the following:

\[
\begin{align*}
3.0 &= (0) \cdot (10000000) \cdot (100000000000000000000000) \ [0x40400000] \\
2.5 &= (0) \cdot (10000000) \cdot (010000000000000000000000) \ [0x40200000] \\
\end{align*}
\]

The sign bit of the result of 3.0 \(\cdot\) 2.5 would be 0. The mantissa would be 1.100…\(\cdot\)1.010…, or 1.111… (do you see why?). Finally, the exponent would be \(1 + 1\), or 2, represented as 1000001. Thus we see that the product would be

\[
(0) \cdot (10000000) \cdot (111000000000000000000000) \ [0x40F00000]
\]

as expected, which converts to 7.5.

Addition is considerably more difficult, as addition can only happen when the exponents of the two quantities are the same. If the two exponent values are the same, then adding the mantissae is (almost) enough:

\[
\begin{align*}
3.0 &= (0) \cdot (10000000) \cdot (100000000000000000000000) \ [0x40400000] \\
2.5 &= (0) \cdot (10000000) \cdot (010000000000000000000000) \ [0x40200000] \\
\end{align*}
\]

\(^3\)But remember to account both for the exponent bias, and the unexpressed 1 bit at the head of the mantissa!
The binary quantities 1.01 and 1.1 add up to 10.11, so the answer would be 10.11 times the common exponent: $10.11 \cdot 2^1$. Of course, this isn’t legal, but it’s easy enough to convert by shifting to a legal equivalent: $1.011 \cdot 2^2$. This yields

$$(0) \ (100000000000000000000000) \ [0x40B00000]$$

which converts as expected to 5.5.

However, when the two exponents are not the same (for example, adding 2.5 and 7.5), one of the addends must be converted to an equivalent form, but with a compatible exponent.

$$2.5 = (0) \ (10000000) \ (010000000000000000000000) \ [0x40200000]$$
$$7.5 = (0) \ (10000001) \ (111000000000000000000000) \ [0x40F00000]$$

Let’s convert 7.5: $1.111 \cdot 2^2$ is the same as $11.11 \cdot 2^1$. Adding 11.11 + 1.01 yields 101.00. $101.00 \cdot 2^1$ is the same as $1.01 \cdot 2^3$. The final answer is thus

$$(0) \ (10000010) \ (010000000000000000000000) \ [0x41200000]$$

which, again, is the expected value, in this case 10.0.

**String representations**

Nonnumeric data such as characters and strings are also treated as binary quantities and differ only in interpretation by the programmer/user. The most common standard for storing characters is the ASCII code, formally the American Standard Code for Information Interchange. ASCII code assigns a particular character value to every number between 0 and 127.

This is a slight oversimplification. Technically speaking, ASCII provides an interpretation for every seven-bit binary pattern between (and including) 0000000 and 1111111. Many of these patterns are interpreted as characters; for example, the pattern 1000001 is an upper case ‘A.’ Some binary strings, especially those between 0000000 and 0011111, are interpreted as “control characters,” such as a carriage return, or a command to a peripheral such as “start of header” or “end of transmission.” As almost all computers are byte-oriented, most store ASCII characters not as 7-bit patterns but as 8-bit patterns, with the leading bit being a zero. The letter ’A’ would be, for example, (binary) 01000001, or (hexadecimal) 0x41, or (decimal) 65. Using 8-bit storage allows computers to use the additional (high-bit) patterns for character set extensions; in Microsoft Windows, for example, almost every character set has different display values in the range 128–255. These display
values may include graphics characters, suits (of cards), foreign characters with diacritical marks, and so forth.

The chief advantage of the ASCII encoding is that every character will fit comfortably into a single byte. The chief disadvantage of ASCII is that, as the American standard code, it does not well reflect the variety of alphabets and letters in use worldwide. As the Internet continues to connect people of different nationalities and languages together, it became obvious that some method of encoding non-English (or at least, non-US) characters was necessary. The result was the UTF-16 encoding, promulgated by the Unicode consortium.

UTF-16 uses two bytes (16 bits) to store each character. The first 128 patterns are almost identical to the ASCII code. With 16 bits available, however, there are over 65,000 (technically, 65536) different patterns, each of which can be assigned a separate (printable) character. This huge set of characters allows uniform and portable treatment of documents written in a variety of alphabets, including (US) English, unusual characters such as ligatures (e.g., æ) and currency symbols, variations on the Latin alphabet such as French and German, and “unusual” (from the point of view of American computer scientists) alphabets such as Greek, Hebrew, Cyrillic (used for Russian), Thai, Cherokee and Tibetan. The Greek capital psi, (Ψ) for example, is represented by 0x803A, or in binary 1000000000111010. Even the Chinese/Japanese/Korean ideograph set (containing over 40,000 characters) can be represented.

Machine operation representations

In addition to storing data of many different types, computers also need to store executable program code. Like all other patterns discussed, at its most basic level, the computer can only interpret binary activation patterns. These patterns are usually called machine language. One of the major roles of a register in the CPU is to fetch and hold an individual bit pattern, decode that statement into a machine instruction, and then execute that instruction.

Interpretation of machine language is difficult in general and varies greatly from computer to computer. For any given computer, the instruction set defines which operations are possible for the computer to execute. The Java Virtual Machine, for example, has a relatively small instruction set, with only 256 possible operations. Every byte, then, can be interpreted as a possible action to take. The value 89 (0x59) corresponds to the dup instruction, causing the machine to duplicate a particular piece of information stored in
the CPU. The value 146 (0x92) corresponds to the \texttt{i2c} instruction, which converts a 32-bit quantity (usually an integer) to a 16-bit quantity (usually a Unicode character). These number-to-instruction correspondences are specific to the JVM and would not work on a Pentium 4 or a PowerPC, which have their own idiosyncratic instruction sets.

The task of generating machine code to do a particular task is often extremely demanding. Computers usually provide a large degree of programmed support for this task. Programmers usually write their programs in some form of human-readable language. This human-readable language is then converted into machine code by a program such as a compiler, in essence a program that converts human-readable descriptions of programs into machine code.

\textbf{Interpretation}

In light of the preceding several sections, any given bit-pattern can almost certainly be interpreted in several different ways. A given 32-bit pattern might be a floating point number, two UTF-8 characters, a few machine instructions, two 16-bit signed integers, an unsigned 32-bit integer, or many other possibilities. How does the computer distinguish between two otherwise identical bitstrings?

The short and misleading answer is that it can’t. A longer and more useful answer is that the answer is provided by the context of the program instructions. As will be discussed in the following chapter, most computers (including the primary machine discussed, the Java Virtual machine) have several different kinds of instructions that do, broadly speaking, the same thing. The JVM, for example, has separate instructions to add 32-bit integers, 64-bit integers, 32-bit floating point numbers, and 64-bit floating point numbers. Implicit in these instructions is that the bit patterns to be added will be treated as though they were the appropriate type. If you, as the programmer, load two integers into registers and then tell the computer to add two floating point numbers, the computer will naively and innocently treat the (integer) bit patterns as though they were floating point numbers, add them, and get a meaningless and error-ridden result. Similarly, if you tell the computer to execute a floating point number as though it were machine code, the computer will attempt, to the best of its ability, whatever silly instruction(s) that number corresponds to. If you’re lucky, this will merely crash your program. If you’re not lucky…well, that’s one of the major ways that hackers can get into a computer, by overflowing a buffer and overwriting executable code with their own instructions.
It’s almost always an error to try to use something as though it were a different data type. Unfortunately, this is an error that the computer can only partially compensate for. Ultimately, it is the responsibility of the programmer (and the compiler writer) to make sure that data is correctly stored and that bit patterns are correctly interpreted. One of the major advantages of the Java Virtual Machine is that it can catch some of these errors.

1.3 Virtual Machines

1.3.1 What is a “virtual machine”?

Because of differences between instruction sets, the chances are very good that a program written for one particular computer will not run on a different one. This is why software vendors sell different versions of programs for Linux, Windows, and Macintosh computers, and also why many programs have “required configurations,” stating that a particular computer must have certain amounts of memory or certain specific video cards to work properly. In the extreme case, this would require that computer programmers write related programs (such as the Mac and Windows version of the same game) independently from scratch, a process that would essentially double the time, effort, and cost of such programs. Fortunately, this is rarely necessary. Most programming is done in so-called high-level languages such as C, C++, or Java, and then the (human-readable) program source code is converted to executable machine code by another program such as a compiler. Only small parts of programs — for example, embedded systems, graphics requiring direct hardware access, or device drivers controlling unusual peripherals — need be written in a machine-specific language.

The designers of Java, recognizing the popularity of the Web and the need for program-enabled Web pages to run anywhere, have taken a different approach. Java itself is a high-level language. Java programs are typically compiled into class files, with each file corresponding to the machine language for a program or part of a program. Unlike normal executables compiled from C, Pascal, or C++, the class files do not necessarily correspond to the physical computer upon which the program is written or running. Instead, the class file is written using the machine language and instruction set of the Java Virtual Machine (JVM), a machine that exists only as a software emulation, a computer program pretending to be a chip. This “machine” has structure and computing power typical of — in some cases, even greater than — a normal, physical, computer such as an Intel Pentium 4, but is freed
from many of the problems and limitations of a physical chip.

The JVM is usually a program running on the host machine. Like any other executable program, it runs on a physical chip using the instruction set of the local machine. This program, though, has a special purpose in that the program’s primary function is to interpret and execute class files written in the machine language of the Java Virtual Machine. By running a specific program, then, the physical chip installed in the computer can pretend to be a JVM chip, thereby being able to run programs written using the JVM instruction set and machine code.

The idea of a “virtual machine” is not new. In 1964, IBM began work on what would be known as VM/CMS, an operating system for the System/360 that provided time-sharing service to a number of users simultaneously. In order to provide the full services of the computer to every user, the IBM engineers decided to build a software system and user interface that, to the user, looked like he (or she) was alone and had the entire system to him/herself. Every person or program could have an entire virtual S/360 at their disposal as needed, without worrying about whether their program would crash another person’s. This also allowed engineers to upgrade and improve the hardware significantly without forcing users to re-learn or re-write programs. More than twenty years later, VM/CMS was still in use on large IBM mainframes, running programs designed and written for a virtual S/360 on hardware almost a million times faster. Since then, virtual machines and emulators have become a standard part of many programming tools and languages (including Smalltalk, an early example of an object-oriented language).

SIDEBAR: The .NET FRAMEWORK. Another example of a common virtual machine is the .NET Framework, developed by Microsoft and released in 2002. The .NET Framework underlies the current version of Visual Studio and provides a unified programming model for many network-based technologies such as ASP.NET, ADO.NET, SOAP, and .NET Enterprise Servers. Following Sun’s example of the Java Virtual Machine, Visual Studio incorporates a Common Language Runtime (CLR), a virtual machine to manage and execute code developed in any of several languages. The underlying machine uses a virtual instruction set called Microsoft Intermediate Language (MSIL) that is very similar in spirit to JVM bytecode. Even the detailed architecture of the CLR’s execution engine is similar to the JVM; for example, they’re both stack-based architectures with instruction set support for object-oriented, class-based environments. Like the JVM, the CLR was designed to get most of the advantages of a virtual machine: abstract, portable code that can be widely distributed and run without compromising local machine security. Microsoft has also followed Sun’s example in the development and distribution of a huge library of pre-defined classes to support programmers using the .NET Framework who don’t wish to have to reinvent wheels. Both systems were designed with web services and mobile applications in mind. Unlike the JVM, MSIL was designed more or less from the start.
1.3. VIRTUAL MACHINES

1.3.2 Portability concerns

A primary advantage of a virtual machine, and the JVM in particular, then, is that it will run anywhere that a suitable interpreter is available. Unlike a Mac program that requires a PowerPC G4 chip to run (G4 emulators exist, but they can be hard to find and expensive), the JVM is widely available for almost all computers, and even many equipment such as personal digital assistants (PDAs). Every major web browser such as Internet Explorer, Netscape, or Konqueror has a JVM (sometimes called a Java runtime system) built in to allow Java programs to run properly.

Furthermore, the JVM will probably continue to run anywhere, as the program itself is relatively unaffected by changes in the underlying hardware. A Java program (or JVM class file) should behave identically, except for speed, on a JVM emulator written for an old Pentium computer as for a top-end PowerPC G7 — a machine so new that it doesn’t even exist yet, but if/when Motorola makes one, they will almost certainly write a JVM client for it.

1.3.3 Transcending limitations

Another advantage that virtual machines (and the JVM in particular) can provide is the ability to transcend, ignore, or mask limitations imposed by the underlying hardware. The JVM has imaginary parts corresponding to the real computer components discussed above, but because they consist only of software, there’s little or no cost to making them. As a result, they can be as large or as numerous as the programmer needs. Every (physical) register on a chip takes up space, consumes power, and costs significant to support a wide variety of programming languages, including J#, C#, Managed C++, and Visual Basic. Microsoft has also established a standard assembler llasm. In practical terms, the CLR is not as common or as widely-distributed a computing environment as the JVM, but the software market is an extremely dynamic environment and the market’s final verdict has not yet been returned.

One key issue that will probably have a significant impact is the degree of multi-platform support that Microsoft provides. In theory, MSIL is as platform-independent as the JVM, but substantial parts of the .NET Framework libraries are based on earlier Microsoft technologies and will not run successfully on UNIX systems, or even older Windows systems (such as Windows 98). Microsoft’s track record of supporting non-Microsoft (or even older Microsoft) operating systems does not encourage third-party developers to develop cross-platform MSIL software. Java, by comparison, has developed a strong critical mass of developers on all platforms who both rely on and support this portability. If Microsoft can follow through on its promise of multi-platform support and develop a sufficient customer base, .NET might be able to replace Java as the system of choice for developing and deploying portable, Web-based applications.
amounts of money; as a result, registers are often in somewhat short supply. On the JVM, registers are essentially free, and programmers can have and use as many as they like. The system bus, connecting the CPU to memory, can be as large as a programmer wants or needs.

A historical example may make the significance of this point clear. The original IBM-PC was based on the Intel 8088 chip. The 8088 was, in turn, based on another (earlier) chip by Intel, the 8086, almost identical in design, but with a 16-bit bus instead of an 8-bit bus. This implicitly limits the data transfer speed between the CPU and memory (of an 8088) by about fifty percent relative to an 8086, but IBM chose the 8088 to keep its manufacturing costs and sales prices down. Unfortunately, this decision limited PC performance for the next fifteen years or so, as IBM, Intel, and Microsoft were required to maintain backwards compatibility with every succeeding generation of chips in the Intel 80x86 family. Only with the development of Windows was Microsoft finally able to take full advantage of the cheaper manufacturing and wider buses. Similar problems still occur with most major software and hardware manufacturers struggling to support several different chips and chipsets in their products, and a feature available on a high-end chipset may not be available at the lower ends. Appropriately written Java runtime systems can take advantage of the feature where available (on a JVM written specifically for the new architecture) and make it useful to all Java programs or JVM class files.

The JVM also has the advantage of being, fundamentally, a better and cleaner design than most physical computer chips. Part of this is the result of design from scratch in 1995, instead of inheriting several generations of engineering compromises from earlier versions. But the simplicity that results from not having to worry about physical limitations such as chip size, power consumption, or cost, meant that the designers were able to focus their attention on producing a mathematically tractable and elegant design that allows the addition of useful high-level properties. In particular, the JVM design allows for a high degree of security enhancements, something discussed briefly below and in greater detail in a later chapter.

\subsection{1.3.4 Ease of updates}

Another advantage to a virtual machine is the ease of updating or changing the virtual machine, relative to the ease of upgrading hardware. A well-documented error with the release of the Pentium chip in 1994 showed that the FPU in the Pentium P54C didn’t work properly. Unfortunately for the consumers, fixing this flaw required a physical replacement of the chip,
sending the old one back to Intel and receiving/installing a new and updated copy. By contrast, a bug in a JVM implementation can be repaired in software by the writers, or even possibly by a third party with source-code access, and distributed via normal channels, including simply making it available on the Internet.

Similarly, a new-and-improved version of the JVM, perhaps with updated algorithms or a significant speed increase, can be distributed as easily as any other updated program such as a new video card driver or a security upgrade to a standard program. Since the JVM is software-only, a particularly paranoid user can even keep several successive versions around, so that in case a new version has some subtle and undiscovered bug, she or he can revert to the old version and still run programs. (Of course, you only think this user is paranoid until you find out she’s right. Then you realize she’s just careful and responsible.)

1.3.5 Security concerns

A final advantage of virtual machines is that, with the cooperation of the underlying hardware, they can be configured to run in a more secure environment. The Java language and the Java Virtual Machine were designed specifically with this sort of security enhancement in mind. For instance, most Java applets don’t require access to the host computer’s hard drive, and providing them with such access, especially with the ability to write on the hard drive, might lay the computer open to infection by a computer virus, theft of data, or simply having crucial system files deleted or corrupted. The virtual machine, being in software, is in a position to vet attempted disk accesses and to enforce a more sophisticated security policy than the operating system itself may be willing or able to enforce.

The JVM goes even further than that, being designed not only for security, but for a certain degree of verifiable security. Many security flaws in programs are created accidentally, for example, by a programmer attempting to read in a string without making sure that there is enough space allocated to hold it, or even by attempting to perform an illegal operation (perhaps dividing a number by an ASCII string), with unpredictable and perhaps harmful results. JVM bytecode is designed to be verifiable and verified via a computer program that checks for this sort of accidental error. Not only does this reduce the possibility of a harmful security flaw, but it also improves the overall quality and reliability of the software. Software errors, after all, don’t necessarily crash the system or allow unauthorized access. Many errors will, instead, quietly and happily — and almost un-
detectably — produce wrong answers. By catching these errors before the answers are produced, sometimes even before the program is run, this source of wrong answers is substantially reduced and program reliability is significantly increased. The JVM security policy will be discussed extensively in a chapter 10.

1.3.6 Disadvantages of a virtual machine

So if virtual machines are so wonderful, why aren’t they more common? The primary answer, in one word, is “speed.” It usually takes about 1000 times longer to do a particular operation in software instead of hardware, so performing hard-core computations (matrix multiplications, DNA sequencing, and so forth) in Java on a Java virtual machine may be significantly slower than performing the same computations in (chip-specific) machine language, compiled from C++.

The practical gap, fortunately, does not appear to be anything close to 1000 times, mainly because there have been some very smart JVM implementers. A properly-written JVM will take advantage of the available hardware where practical, and will do as many operations as possible using the available hardware. Thus, the program will (for example) use the native machine’s circuitry for addition instead of emulating it entirely in software. Improvements in compiler technology have made Java run almost as fast as natively compiled code, often within a factor of two, and sometimes almost imperceptibly slower. A February 1998 study by Java-World (“Performance tests show Java as fast as C++”) found that, for a set of benchmark tasks, high-performance JVMs with high-efficiency compilers would typically produce programs that ran, at worst, only slightly (about 5.6%) slower than their C++ equivalents, and were mostly identical to the limits of measurement. Of course, comparing computers and languages for speed is a notoriously difficult process, much like the standard apples and oranges comparison, and other researchers have found other values.

In some regards, speed comparisons can be a non-issue; for most purposes other than hard-core number crunching, Java or any other reasonable language is fast enough for most people’s purposes. Java, and by extension the JVM, provides a powerful set of computation primitives including extensive security measures that are not found in many other languages such as C++. It’s not clear that the ability to crash a computer quickly is a tremendous advantage over being able to run a program to completion, but somewhat more slowly.

A more significant disadvantage is the interposition of the JVM inter-
1.4. PROGRAMMING THE JVM

1.4.1 Java: what the JVM isn’t

Java, it must be stressed, is not the same as the Java Virtual Machine, although they were designed together and often are used together. The Java Programming Language is a high-level programming language designed to support secure, platform-independent applications in a distributed networking environment such as the Internet. Perhaps the most common use of Java is to create “applets,” to be downloaded as part of a Web page and to interact with the browser without cluttering up the server’s network connection. The Java Virtual Machine, on the other hand, is a shared virtual computer that provides the basis for Java applications to run.

Much of the design of Java strongly influenced the JVM. For example, it is a virtual machine precisely because Java should be a platform-independent language, and so cannot make any assumption about what kind of computer the viewer uses. The JVM was designed around the notion of a security verifier so that Java programs could run in a secure environment. Networking support is built into the JVM’s standard libraries at a relatively low level to ensure that Java programs will have access to a standardized and useful set of networking operations. There is, however, no necessary connection between the two products. A Java compiler could in theory be written to

prenter between the programmer and the actual physical hardware. For many applications demanding assembly language programming (games, high-speed network interfacing, or attaching new peripherals), the key reason that assembly language is used is to allow direct control of the hardware (especially of peripherals) at a very low level, often bit-by-bit and wire-by-wire. A badly written JVM prevents this sort of direct control. This is becoming less and less of an issue with the development of silicon implementations of the JVM such as the aJile Systems’ aJ-100 microcontroller or the Zucotto Systems’ Xpresso family of processors. Both of these companies are producing controller chips suitable for hand-held Internet devices that use a chip-based implementation of JVM bytecode as machine code. In other words, these chips do not require any software support to translate JVM bytecode into native instruction sets, and therefore can run at full hardware speeds, with full control over the hardware at a bit-by-bit level. With the development of Java chips, the JVM has come full circle from a clever mathematical abstraction to allow portable and secure access to a wide variety of processors to a physical chip of inherent interest in its own right.
public class JavaExample {
    public static void main(String[] args) {
        System.out.println("This is a sample program.");
    }
}

Figure 1.6: Sample program in Java

#include <stdio.h>

int main()
{
    (void) printf("This is a sample program.\n");
}

Figure 1.7: Sample program in C

compile to native code for the PowerPC (the hardware underlying the Macintosh) instead of JVM bytecode, and a compiler for any other language could be written to compile to JVM code.

In particular, consider the program written in figure 1.6. This is a simple example of a Java program, a program that outputs a given fixed string to the default system output, usually the currently-active window. This, or a very similar program, is often the standard first example in any elementary programming class, although sometimes one instead is shown a program that opens a new window and displays a message. Examined in more detail, however, the Java program itself does nothing of the sort. The Java program, by itself, will do nothing. In order to be executed, it must first be run through a translation program of some sort (the compiler) in order to produce an executable class file. Only this can be run to produce the desired output.

Java, or any programming language, is better viewed as a structure for specifying computations to be performed by the computer. A program in any language is a specification of a particular computation. In order for the computer to do its thing, the specification (program) must be translated into the machine code that the computer ultimately recognizes as a sequence of program steps.
Program PascalExample(output);

begin
  writeln ('This is a sample program.');
end.

Figure 1.8: Sample program in Pascal

#include <iostream.h>
#include <stdlib.h>

int main()
{
  cout << "This is a sample program." << endl;
}

Figure 1.9: Sample program in C++

1.4.2 Translations of the sample program

There are usually many ways of specifying a given computation. A very similar program can be written in many other languages. For example, figures 1.7, 1.8, and 1.9 show programs with identical behavior written in C, Pascal, and C++. These programs also show tremendous similarity in overall structure. The core of the program is written in a single line that somehow accesses a fixed string (of ASCII characters) and calls a standard library function on it to pass it to a default output device. The differences are subtle and in the details; for example, the Pascal program has a name (PascalExample), while the C and C++ versions do not. In both C and C++, the program must explicitly represent the end-of-line carriage return, while Java and Pascal have a function that will automatically put a return at the end of the line. These differences, however, are relatively small when compared with the tremendous amount of similarity, especially when one considers the differences between this and machine-level code.

In light of the preceding discussion of the architecture and organization of a typical computer, consider the following: Few, if any, CPUs have enough registers within the ALU or control unit to allow storing an entire string. (This is especially true given that there is no necessary upper limit to the length of a string, in the abstract. Instead of printing a mere sentence, the programs could have printed an entire textbook.) No CPU has a single
instruction to print a string to the screen. Instead, the compiler must break
the program down into sufficiently small steps that are within the computer’s
instruction set.

1. The string itself must be stored somewhere in main memory, and
2. the CPU must determine where that storage location is.
3. The CPU must also determine which output peripheral should print
   the message, and possibly
4. what type it is. Finally,
5. it must pass the appropriate instructions to the peripheral, telling it
6. where the string is stored,
7. that the string is a string (and not an integer or a floating point num-
   ber), and
8. that the appropriate action to take is to print it (possibly while auto-
   matically appending a return).

From a single line of code can be extracted up to eight or more individual
operations that the computer must perform. In fact, there’s no limit on the
number of operations that might be performed in a single line of code; a line
in Java like
\[
i = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 9 + 9 + 10;
\]
requires an operation for every addition, in this case nine separate calcu-
lations. Since there’s no limit to the theoretical complexity of a mathematical
formula, there’s no limit to the complexity of a single Java statement.

1.4.3 High- and low-level languages

This encapsulation of many many machine-language instructions into a sin-
gle line of code is typical of what are called high-level languages. Java, Pascal,
and so forth are typical examples of one style of such languages. The task
of a Java compiler is to take a complicated statement or expression and
and produce an appropriate collection of individual machine instructions to
perform the task.

By contrast, a low-level language is characterized by a very close re-
lation ship between operations in machine language and statements in the
program code. Using this definition, machine language is of course a very
low-level language, since there is always a 1:1 relationship between a machine language program and itself. **Assembly language** is a slightly more human-readable, but still low-level, language, designed to promote total control over the machine and the machine code instructions, but still be readable and understandable by the programmer. Assembly language is also characterized by a 1:1 relationship between assembly language instructions and machine code instructions.

In machine language, every element of the instruction (also called **opcode**, an abbreviation for “operation code”) is, like everything else in the computer, a number. An earlier section mentioned in passing, for instance, that the opcode 89 (0x59) is meaningful to the JVM and causes it to duplicate a particular piece of information. In assembly language, every opcode is also given a **mnemonic** (pronounced “num-ON-ik,” from the Greek word for memory) that explains or summarizes exactly what it does. The corresponding mnemonic for opcode 89 is **dup**, short for duplicate. Similarly, the mnemonic **iadd**, short for “integer add,” corresponds to the machine opcode to perform an integer addition. The task of the translation program, in this case called an **assembler**, is to translate each mnemonic into its appropriate binary opcode and to gather the appropriate pieces of data that the computer needs.

### 1.4.4 The sample program as the JVM sees it

The JVM machine code version of the sample program(s) above would thus be a long and mostly unreadable binary string. A version of the program that corresponds to the machine code is presented as figure 1.11. This program was written (by hand) in a (low-level) JVM assembly language called **jasmin**, but could also have been written by a compiler starting from one of the original sample programs. Notice that the program is much longer — almost thirty lines instead of only three or four — and much more difficult to understand. This is because a lot of the things that you take for granted in writing high-level programs must be specified exactly and explicitly. For example, in Java, every class is implicitly a sub-class of type Object unless specified otherwise. The JVM requires, instead, that every class defines its relationship to other classes explicitly. The comments (beginning with semicolons) in figure 1.11 present a more detailed, line-by-line description of exactly how the (implicit) operations are defined and carried out.

Notice, though, that although the notions of class, subclass, and so forth are explicitly supported and used in Java, there is nothing especially Java-specific about the JVM low-language program presented in this section.
Figure 1.10: The programming process
1.4. PROGRAMMING THE JVM

; defines the class file associated with this as jasminExample.class
.class public jasminExample
; defines jasminExample as a subclass of Object
.super java/lang/Object

; boilerplate needed for object creation
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V
  return
.end method

.method public static main([Ljava/lang/String;)V
  ; we need two stack elements, for System.out and the string
  .limit stack 2
  ; find System.out (an object of type PrintStream)
  ; and put it on the stack
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ; find the string (characters) to be printed
  ; and put it on the stack
  ldc "This is a sample program."
  ; invoke the PrintStream/println method
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  ; ... and that's it!
  return
.end method

Figure 1.11: Sample program in JVM assembly language
In fact, just as it is the task of a Java compiler to convert the high-level Java code to something akin to JVM machine code, so is it the task of a C++ or Pascal compiler to do the same with its program code for a specific platform. There is no reason that a compiler couldn’t be written that produces JVM code as its final (compiled) output instead of PowerPC or Pentium machine code; the program would then run on a JVM (for instance, on a JVM emulator, inside a web browser, or on a special-purpose chip like those mentioned earlier) instead of on the specific chip hardware.

1.5 Chapter Review

- Computers are simply high-speed algorithmic devices; the details of their construction as electronic devices is less important than the details of how they work as information processing devices.

- The main part of a computer is the Central Processing Unit (CPU), which in turn contains the Control Unit, Arithmetic/Logical Unit (ALU) and Floating Point Unit (FPU). This is where all computations actually happen and programs actually run.

- Memory and peripherals are connected to the CPU through a collection of electrical buses that carry information to and from the CPU.

- All values stored in a conventional computer are stored as binary digits or bits. These are grouped for convenience into larger units such as bytes and words. A particular bit pattern may have several interpretations as an integer, a floating point number, a character or sequence of characters, or even as a set of instructions to the computer itself. Deciding what types of data a given bit patterns represents is done on the basis of context, and is largely the programmer’s responsibility.

- Different CPU chips have different instruction sets, representing different things that they can do and ways that they do things. Every CPU, then, needs a different executable program written in a different kind of machine language, even to run the same program.

- A virtual machine is a program that runs atop a real CPU and interprets machine instructions as though it itself were a CPU chip. The Java Virtual Machine (JVM) is a very common example of such a program and can be found on almost every computer and in every Web browser world-wide.
• Virtual machines can provide a lot of advantages over conventional silicon-based CPUs, such as portability, fewer hardware limitations, ease of updates, and security. Virtual machines can have a big disadvantage in speed.

• Java is an example of a high-level language, a language that may combine many machine-code instructions into a single statement. C, Pascal, and C++ are similar high-level languages. These languages must be compiled to convert their code into a machine-executable format.

• Low-level languages like assembly language have a tight, typically 1:1 relationship between program statement and machine instructions.

• There is no necessary connection between Java and the JVM; most Java compilers compile to JVM executable code, but a C, Pascal, or C++ compiler could, as well. Similarly, a Java compiler could compile to Pentium 4 native code, but the resulting program wouldn’t run on a Macintosh computer.

1.6 Exercises

1. What is an algorithm?

2. Is a recipe as given in a cookbook an example of an algorithm?

3. Name the part of the computer described by each phrase:
   • The heart and ultimate controller of the computer, the place where all calculations are performed.
   • The part of the computer responsible for moving data around within the machine.
   • The part of the computer responsible for all computations such as addition, subtraction, multiplication, and division
   • A set of wires to interconnect different devices for data connectivity.
   • A device for reading, displaying, or storing data.
   • Short-term storage for data and executing programs.
4. How many different patterns could be stored in a 16-bit register? What is the largest value that could be stored as a signed integer in such a register? What is the smallest value? How about the largest and smallest values that could be stored as unsigned integers?

5. Convert the following 16-bit binary numbers into hexadecimal and signed decimal numbers (no, you don’t get to use a calculator!):
   - 1001110011101110
   - 1111111111111111
   - 0000000011111111
   - 0100100010000100
   - 1111111100000000
   - 1100101011111110

6. Convert the following 32-bit IEEE floating point numbers from hex into standard decimal notation.
   - 0x40200000
   - 0x41020000
   - 0xC1060000
   - 0xBD800000
   - 0x3EAAAAAB
   - 0x3F000000
   - 0x42FA8000
   - 0x42896666
   - 0x47C35000
   - 0x4B189680

7. Convert the following decimal numbers into 32-bit IEEE floating point notation.
   - 2.0
   - 45.0
   - 61.01
   - -18.375
   - -6.68
8. Are there any numbers that can be represented exactly as a 32-bit integer, but not as a 32-bit IEEE floating point number? Why or why not?

9. Using a standard ASCII table (check the Internet or appendix E) what four hexadecimal bytes would represent the string “Fred”?

10. What ASCII character string would correspond to the hexadecimal number 0x45617379?

11. True or false: the more 1’s in a binary number, the larger it is. Why or why not?

12. Why won’t executables created for a Windows Pentium IV run on a Macintosh (without special software support)?

13. What is the most important advantage of a virtual machine over a chip-based architecture?

14. What is the most important disadvantage?

15. What languages can be used to write programs for the Java Virtual Machine (JVM)?

16. How many low-level machine instructions would correspond to the statement:
   \[ x = a + (b \times c) + (d \times e); \]

### 1.7 Programming Exercises

1. Write a program (in any language approved by the instructor) to read in a 32-bit signed binary integer and to output its decimal equivalent.

2. Write a program (…) to read in a 32-bit binary floating point number and to output its decimal equivalent.

3. Write a program (…) to read in a decimal floating point number and output its 64-bit hexadecimal equivalent. (Note: this may require additional reading.)
4. Write a program (…) to read two 32-bit floating point numbers $A$ and $B$ (in hex) and to output their sum $A + B$ in hex. Do not convert to decimal.

5. Write a program (…) to read two 32-bit floating point numbers $A$ and $B$ (in hex) and to output their product $A \cdot B$ in hex. Again, do not convert to decimal.

6. Write a program (…) to read a 32-bit floating point number $A$ (in hex) and to output its reciprocal $\frac{1}{A}$ in both hex and decimal. Can you use this in conjunction with the previous problem to perform floating point division? How?
Chapter 2

Arithmetic Expressions

2.1 Notations

2.1.1 Instruction sets

A central problem—indeed, perhaps the central problem—with computers is their lack of imagination and limited set of things that they can do. Consider trying to work the following story problem out on a typical pocket calculator:

![Figure 2.1: A conical mountain](image)

What is the volume of a circular mountain 450m in diameter at the base and 150m high?
A few minutes of searching through geometry textbooks will give you the formulas you need: the volume of a cone is one-third the product of the area of the base and the height. The area of the (circular) base is $\pi$ times the square of the radius. The radius is half of the diameter. The value of $\pi$, of course, is 3.14 and a bit. Putting it all together, the answer would be

$$\frac{1}{3} \cdot \pi \cdot \left(\frac{450}{2}\right)^2 \cdot 150$$

So how do you work with this mess?

Here is where the issue of the computer’s — or in this case, the calculator’s — instruction set rears its head. This formula cannot be entered as-is into a typical calculator. It will need to be broken down into bite-size pieces that the calculator, or computer, can work with. Only a sequence of such pieces will allow us to extract the final answer. This is no different from traditional computer programming, except that the pieces used by such a calculator are much smaller than the pieces used in higher level languages such as Java.

Depending upon the exact calculator used, there are several sequences of keystrokes that would work to solve this problem. On a typical high-end calculator, the following sequence would calculate $\left(\frac{450}{2}\right)^2$:

$$(450 \div 2)^2$$

or, alternately,

$$450 \div 2 = x^2$$

The entire calculation can be performed by

$$1 \div 3 \pi \cdot \left(450 \div 2\right)^2 \cdot 150$$

2.1.2 Operations, operands, and ordering

Implicit in this are a few subtle points. First, notice that the “instruction set” of this calculator includes an $x^2$ button, so that a number can be squared with a single press. It also has a $\pi$ button. Without these conveniences, there would need to be a lot more keystrokes and greater complexity. In fact, few people know what an appropriate sequence of keystrokes would be to replace the $\sqrt{x}$ button.

Second, notice that ordering is important in this sequence. Just as there is a difference between 450 and 540, there is also a difference between $450 \div 2$ and $450 \cdot 2 \div$. The first yields the value 225, while the second does not even make sense (under conventional notation). In general, most of the mathematical operations people think of are binary, which is to say that
they take two numbers (arguments, formally called operands)) and produce a third. Add 3 and 4, get 7. A few advanced mathematical operations, such as \( \sin x \), \( \cos x \), and \( \log x \) are unary, meaning they take only one argument. In order to process an operation, a computer needs both the operator, defining which operation is to be performed, and the values of all necessary operands, in a very strict format.

In conventional chalkboard mathematics, for example, binary operations are written in so-called infix notation, meaning that the operator is written in the middle, between the two operands (as in \( 3 + 4 \)). Trig functions such as \( \sin \) are written in prefix notation, where the operation precedes the operand(s) (as in \( \sin \frac{\pi}{2} \)). Some high-end calculators such as the Hewlett-Packard HP 49G also support postfix notation (also called reverse Polish notation), where the operation comes last, after the operands. On such a calculator, the sequence of buttons to press to divide 450 by 2 would be

\[
\text{4} \quad \text{5} \quad \text{0} \quad \text{ENTER} \quad \text{2} \quad \div
\]

Although this notation may appear confusing at first, many people come to prefer it with a little bit of experience, for reasons we shall explore later.

### 2.1.3 Stack-based calculators

This kind of calculator can be easily modelled by use of a data structure called a stack. The name derives from the image of a stack of trays in a cafeteria, or of styrofoam cups at a fast-food chain. These are typically stored in spring-loaded containers, where the weight of the trays pushes the entire pile down so that the top tray is always at a uniform height. At any point, only the top tray can be removed (causing the rest of the stack to pop up slightly and expose a new tray), or an additional tray can be placed on the top, causing the additional weight to push the trays down slightly.

A stack is a collection of number or data object with similar properties. Only the “top” element is available for processing, but it can be removed (“popped”) at any time, or another object can be added (“pushed”) to the stack, displacing the previous top (to the second position in the stack, and so forth). Stack-based calculations work particularly well in conjunction with postfix notation. Items are pushed onto the stack in order. Unary operations, such as \( \sin \) and \( \log \), can simply pop the top item off the stack for an operand, perform the operation, and push the answer. For binary operations such as addition, the top two stack items are popped, the operation performed, and the answer pushed back.
The following sequence of operations, then, would perform the calculation described above.

\[ 13 \div \pi \cdot 450 \div 450 \div 150. \]

Unpacking the above, first the numbers 1 and 3 are pushed onto the stack, then they are popped and the quotient \( \frac{1}{3} \) or 0.3333 is calculated. The value \( \pi \) is pushed, and then both 0.3333 and \( \pi \) are multiplied, yielding about 1.047, and so forth. Calculations can be performed quickly, in sequence, without the need for extensive parentheses and structures. In particular, note that there is no possible ambiguity about the order of operations as there would be with an infix expression such as \( 1 + 3 \cdot 4 \). The two possibly equivalent expressions \( 1 3 4 \cdot + \) and \( 1 3 + 4 \cdot \) are clearly and obviously distinct.

### 2.2 Stored-Program Computers

#### 2.2.1 The Fetch-Execute cycle

Computers, of course, do not require direct interaction with the operator (via button presses) in order to do their calculations. Instead, every possible operation is stored (as described in the previous chapter) as a bit-pattern; the sequence of operations is stored as a program containing the sequence of bit-patterns. The computer gets its instructions by reading them from memory. The first computers were outgrowths of ballistic calculators and codebreaking machines built for the Second World War. It’s almost incorrect to call them computers, as they were really just complicated, special purpose electrical machinery. Most importantly, to change what they did required that the physical circuitry of the machinery be changed; the program, if you will, was “hard-wired” into the computer. If you needed to solve a different problem, you needed to re-wire the machine and build new circuits.

The great mathematician John Von Neumann recognized this as a major limitation and proposed (“Preliminary Discussion of the Logical Design of an Electronic Computing Instrument”, written with Arthur Burks and Hermann Goldstine in 1946) a revolutionary concept for producing “general-purpose” computing machines. He identified four major “organs” involved in computing, related respectively to arithmetic, memory, control, and connection to the outside world (and human operator). The key, in his view, to building a general-purpose computer was that it should be able to store not only the intermediate results of arithmetic calculations, but also the orders (instructions) that created those calculations. In other words, the “control organ” should be able to read patterns from memory and act upon them. Furthermore, the storage of the control patterns should be as flexible as the storage of numeric data. Why not, therefore, store instructions as binary patterns, just like numeric data?
memory (into the control unit) and interpreting each bit pattern as an operation to be performed. This is usually called the fetch-execute cycle, as it is performed cyclically and endlessly, millions or billions of times per second within the computer.

The design of the “control organ” thus becomes a kind of selector — when this pattern is read from memory, energize that circuit. Von Neumann further pointed out that control patterns and data patterns could even reside in the same memory if there were some way of telling them apart. Alternatively (the approach taken by modern computers) is that they are distinguished not by patterns, but by use. Any patterns loaded into the control unit is automatically treated as an instruction. (A competing proposal, the Harvard architecture, uses separate storage for code and for data. We’ll see this architecture a little later in the discussion of the Atmel microcontroller.) This also implies that instructions can be used as data or even overwritten, allowing the possibility of self-modifying code. This allows the computer to reprogram itself, for example, by copying a program (as data) from storage into main memory, and then executing the program (as code).

Von Neumann’s computer operates by repeatedly performing the following operations:

1. get an instruction pattern from the memory organ
2. determine and get any data required for this instruction from memory
3. process the data in the arithmetic organ
4. store arithmetic results into the memory organ
5. go back to step 1.

Von Neumann thus laid most of the theoretical foundations for today’s computers. His four organs, for example, can easily be seen to correspond to the ALU, system memory, control unit, and peripherals defined earlier. His method of operation is the fetch-execute cycle. Researchers have been exploring the implications of Von Neumann’s architecture for decades, and in some ways have been able to generalize beyond the limitations of his model. For example, multiprocessor systems, in general, replace the single “control organ” with several CPUs, each able to operate independently. A more radically non-VN architecture can be seen in the various proposals for neural networks and connectionist systems, where “memory” is distributed among dozens or thousands of interlocking “units” and there is no control organ. Today, however, the term “Von Neumann computer” is rather rare, for the same reason that fish don’t often talk about water. This kind of computer is so omnipresent, it’s usually assumed that any given machine follows the Von Neumann/stored program architecture.
Inside the control unit are at least two important storage locations. The first, the **instruction register** or IR, holds the bit pattern that has just been fetched from memory in order that it can be interpreted. The second, the **program counter** or PC, holds the location from which the next instruction will be fetched. Under normal circumstances, every time an instruction is fetched, the PC is incremented so that it points to the next word in memory. If, for example, the PC contains the pattern 0x3000, the next instruction will be fetched from that location (interpreted as a memory address). This means that the bit pattern stored at location 0x3000 (and not 0x3000 itself) will be placed into the IR. The PC will then be updated to contain the number 0x3001. On successive cycles, the PC will contain 0x3002, 0x3003, 0x3004, and so forth. At each of these cycles, the IR will contain some instruction which the computer will dutifully follow. Typical examples of instructions include data transfers (where data is moved between the CPU and either memory or an I/O peripheral), data processing (where data already in the CPU will have some arithmetic or logical operation performed upon it), or control operations that affect the control unit itself.

Interpreting the instructions is itself a task of moderate complexity, in part because some “instructions” are actually instruction groups that need further specification. For example, an instruction to store information in memory is incomplete. What information needs to be stored, and where in memory should it be placed? In addition, many machines have additional details that they need, such as the “addressing mode” (does this bit pattern refer to a memory location, or just a number?), the size (number of words) of the data to be stored, and so forth. For this reason, many machine language instructions are actually complexes of related bits (just as was seen with the detailed structure of floating point numbers in the previous chapter).

An example of this for the IBM-PC (Intel 8086 and later machines) is the simple ADD instruction. This can be encoded in two bytes, where the first 8 bits hold the number 0x04, and the second hold an (unsigned) 8-bit number from 0–255. This number will be added, implicitly, to the number already stored in a particular location (the AL register, a specific 8-bit register in the CPU). If you need to add a number larger than 255, there is a different machine instruction, with the first byte 0x05, and the next 16 bits defining a number from 0–65525. If you want to add to the number stored somewhere other than the AL register, then the machine instruction would begin with the opcode byte 0x81, define where exactly the number is stored in a second byte, and then the number to be added.

Because of the design chosen for the JVM architecture (in particular, all addition is done in one place, the stack), the corresponding interpretation
2.2. STORED-PROGRAM COMPUTERS

As will be discussed in the following section, this reflects as much on the fundamental design philosophies of the builders as it does on the power of the JVM itself. But, for example, all addition is done on the stack (as in an RPN calculator). This means that the computer doesn’t need to worry about where the addends come from (since they always come from the stack) or where the sum goes (since it always goes back to the stack). The instruction to add two integers is thus a single byte of value 0x60 — no mess or kerfluffle.

2.2.2 CISC vs. RISC computers

Perhaps obviously, the more different things the CPU is to do, the more different opcodes and machine instructions there will be. Less obviously, the more different opcodes there are, the more unique bit patterns will be needed and the longer they will need to be (on average) in order to make sure the computer can tell them apart. (Imagine the disaster that would occur if the number 0x05 meant not only “add two numbers” but also “halt the computer”! How could the computer tell which was meant?) Longer opcodes, however, will mean a larger IR, a larger bus connecting the computer to program memory, and a more complicated (and expensive) set of circuitry to interpret the instructions. Such a complex instruction set will also require lots of circuitry to actually perform the various operations, since every individual operation might need a different set of wiring and transistors. This means that such a complex CPU chip is likely to be expensive, require expensive support, and to perform more slowly on each instruction than a more streamlined design. (It will also require a bigger chip, run hotter [and therefore need better cooling], and burn more power, reducing battery life.)

Does this mean that a smaller instruction set is better? Not necessarily, for while a CPU with a reduced instruction set may be able to perform certain tasks faster (every CPU, for example, has to have the ability to add two numbers together), there will be lots of tasks that the smaller CPU can only do by combining several steps. For example, a complex instruction set computing (CISC) chip may be able to move a large block of data, perhaps a several thousand byte string, from one memory location to another, without using any of the CPU’s internal storage. By contrast a reduced instruction set computing (RISC) chip might have to move each byte or word into the CPU (from memory) and then back into memory. More importantly, at every step the instruction to move a particular byte would have to be fetched and interpreted. So although the overall fetch-execute cycle may run faster
(and usually does), the particular program or application may need to use lots of instructions to do (slowly) what the CISC computer could do in a single, although long and complex, machine instruction. Another advantage claimed by RISC proponents is a resistance to “creeping featurism,” the tendency of equipment and programs to add complexity in the form of new features. A RISC chip will typically have a small, clean, simple design (by definition) that will remain small, clean, and simple in future versions, while newer versions of CISC chips will typically add more instructions, more features, and more complexity. Among other effects, this will hinder backwards compatibility, because a program written for a later CISC chip will often take advantage of features and instructions that didn’t exist six months earlier. On the other hand, of course, the new features added to the chip are probably features that the engineers found useful.

The two major CPU chips on the market today provide a good example of this difference. The Pentium 4 is a CISC chip, with a huge instruction set (34 different ways alone of expressing ADD, even before taking into account which register or memory location contains the data), while the PowerPC is a RISC chip, tuned to perform common operations quickly, while taking much longer on rare or complex tasks. For this reason, when comparing processor speeds between the CPUs, one can’t just look at number like clock speeds. A single clock cycle on a RISC chip is likely to correspond to a single machine instruction, while a CISC chip is likely to take at least two or three clock cycles to accomplish anything. On the other hand, the larger instruction in the CISC chip may be able to do a more complicated calculation in those few clock cycles, depending upon the application. The performance difference thus depends more on the type of program being run and the exact operations than on clock speed differences. In particular, Apple claims in their sales documentation that the (RISC) 865 MHz Power Mac G4 will perform approximately 60% faster than the (CISC) 1.7 GHz Pentium 4 (and “3 to 4 times faster . . . in graphics and sound manipulation”) simply by getting more work done per clock cycle. Whether you believe Apple’s sales literature or not, their central point — that clock speed is a poor way to compare different CPUs, and that computers can be tuned to different sorts of tasks in the instruction set — remains almost irrefutable, irrespective of which side of the CISC/RISC debate one takes.
2.3 Arithmetic Calculations on the JVM

2.3.1 General comments

The JVM is an example of a stack-based, RISC processor, but with a few twists thrown in reflecting the fact that it doesn’t really exist. Like most computers, the direct arithmetic abilities of the Java Virtual Machine are limited to common and simple operations for efficiency reasons. Few if any machines — and the JVM is not one of them — offer trig functions, but all support elementary arithmetic, including operations such as addition, subtraction, multiplication, and division. The JVM goes beyond most computers, however, by using stack-based computations, like the high-end calculator described earlier. This makes it very easy to emulate on other computers in a way that a fixed set of registers would not be. The JVM itself maintains a stack of binary patterns, holding elements previously pushed and the results of prior computations. The operands of any given operation are taken from the top elements on the computation stack, and the answer is returned to the top of the stack. To calculate $7 \cdot (2 + 3)$, then, would require pushing the seven, two and three (in that order), then executing first an addition and then a multiplication.

This procedure is made slightly more tricky because the JVM (and Java) use typed calculations (in this regard, it’s a little different from most RISC machines, but can offer much greater security). As discussed in the previous chapter, the same bit pattern can represent several different items, and the same number may be represented by several different bit patterns. In order to process these properly, any computer needs to be informed of the the data types represented by the bit patterns. The JVM is unusual only in how strictly this type system is enforced, in an effort to prevent program errors.

As a result, there are, for example, several different ways to express addition, depending upon the types to be added. (This, it could be argued, puts the JVM somewhere in the middle of the CISC/RISC debate.) In general, the first letter of the operation name reflects the type(s) that it expects as arguments and the type of the answer. To add two integers, one uses the mnemonic operation iadd, while to add two floating point numbers one uses fadd. This pattern is followed by other arithmetic operations, so the operation to subtract one integer from another is isub while the operation to divide two double-precision numbers is ddiv.

In detail, the JVM stack is a collection of 32-bit numbers, with no fixed maximum depth. For data types that can fit within a 32-bit register, there
CHAPTER 2. ARITHMETIC EXPRESSIONS

<table>
<thead>
<tr>
<th>Type</th>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>i</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>float</td>
<td>f</td>
<td>32-bit IEEE 754 floating point number</td>
</tr>
<tr>
<td>long</td>
<td>l</td>
<td>64-bit integer, stored in two successive stack locations</td>
</tr>
<tr>
<td>double</td>
<td>d</td>
<td>64-bit IEEE 754 floating point (as above)</td>
</tr>
<tr>
<td>byte</td>
<td>b</td>
<td>8-bit signed integer</td>
</tr>
<tr>
<td>short</td>
<td>s</td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td>char</td>
<td>c</td>
<td>16-bit unsigned integer or UTF-16 character</td>
</tr>
<tr>
<td>address</td>
<td>a</td>
<td>Java objects</td>
</tr>
</tbody>
</table>

Table 2.1: JVM basic types and their representations

is no problem with storing data elements in a single stack location. Longer types are typically stored as pairs of 32-bit numbers, so a 64-bit “double” at the top of the stack actually occupies two positions.

How many positions are there on the JVM stack? In theory, because the JVM is not hardware-limited, there are as many as you need. In practice, every program, method, or function that you write will define a maximum stack size. Similarly, there are no hardware-defined limitations on the amount of memory needed, and instead every method defines a maximum number of local variables, not stored on the stack, that can be used to temporarily store values.

2.3.2 A sample arithmetic instruction set

Data types

The Java virtual machine supports eight basic data types, most of which correspond closely to the basic data types of the Java language itself. These types are listed in Table 2.1. The JVM also provides most of the standard arithmetical operations, including a few that students may not be familiar with. In the interests of reducing the instruction set, though, there are certain design simplifications that have been made.

Notable for its absence from the collection of basic types is the boolean type found in Java, but not in the JVM. Boolean variables, of course, can only hold the values true and false, and as such could be implemented in a single bit. In most computers, however, accessing a single bit is no more efficient — in fact, is much less efficient — than accessing a word. For this reason, in the JVM boolean values are simply represented as the word-sized (32-bit) values 0 or 1, or in other words as integers.
2.3. ARITHMETIC CALCULATIONS ON THE JVM

Similarly, the sub-word storage types of byte, short, and char are also somewhat second-class types. Because in the JVM, doing math on a 32-bit quantity takes no more time than doing math on smaller quantities, variables of this type are automatically promoted to 32-bit integers inside the CPU. On the other hand, there is an obvious difference when variables of this type are stored; for example, an array of one million bytes would take up a quarter of the space of a similar array of integers. For this reason, the JVM supports the operations of loading small types (byte, short, char, and even boolean) from memory and storing into memory, particularly from and into arrays.

Basic arithmetic operations

With this collection of types, each of which requires special processing to support, almost every combination of type and operation needs a special opcode and mnemonic. To simplify the programmer’s task, most mnemonics use a letter code to indicate the type of action. To add two ints, for example, the mnemonic is iadd. Adding two longs would use ladd, while floats or doubles would use fadd and dadd, respectively. For simplicity, this entire family will be abbreviated as ?add, where the ? stands for any of the legal type-letters.

The basic arithmetical operations of addition (?add), subtraction (?sub), multiplication (?mul), and division (?div) are defined for each of the four major types (int, long, float, and double). All of these operations act by popping the first two elements off the stack (n.b. that for a long or a double, the first two “elements” will each involve two stack locations, and hence four stack locations in total), computing the result, and then pushing the result back on the top of the stack. In addition, JVM provides the ?neg operation, which reverses the sign of the item at the top of stack. This could also of course, by pushing the value -1 and then executing a multiply instruction, but a special-purpose operation can do this commonly-executed action faster.

One aspect of the ?div operation requires special attention. Both idiv and ldiv operate upon integers and produce integers (not fractions or decimals) as a result. The result of dividing the number 8 by 5, for instance, yields the answer 1, and not the floating point number 1.6. To perform a floating point division, it is first necessary to convert both arguments to float or double types, as will be discussed in a few subsections. Similarly, there is a special operation for int and long types ?rem which takes the remainder or modulus. This operation does not exist for float/double types,
as the division operation is defined to perform exact division — or, as exact
as the machine representation will allow.

### 2.3.3 Logical operations

The JVM also provides the basic logical operations of AND (\texttt{\&\&}), OR
(\texttt{|\|}), and XOR (\texttt{^}), for int and long types only. These operate in what
is called \textbf{bitwise} fashion, meaning that every bit of the first operand is individ-
ually compared with the corresponding bit of the second operand, and
the result is the collection of the individual bit operations. When applied
to boolean values, 0 and/or 1, the results are as one expects. The repre-
sentation of 0 is 0x0000; the representation of 1 is 0x0001. For all locations
except the last, the corresponding bits are 0 and 0; in the last location, the
bits are of course 0 and 1. The value of 0x0000 OR 0x0001 would thus be,
as expected, 0x0001, or in other words, \texttt{false OR true} is \texttt{true} as desired.

\section*{Shift operations}

In addition to these familiar operations, the JVM also provides some stan-
dard \textbf{shift} operations for changing bit patterns and specifically for moving
bits around in a number. In Java/C/C++, these are represented as the \texttt{<<}
and \texttt{>>} operators. The basic operation, takes every bit in the number and
moves it exactly one place to the right (or left). Using the binary pattern
0xBEEF as an example

<table>
<thead>
<tr>
<th>B</th>
<th>E</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>1110</td>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>0111</td>
<td>1101</td>
<td>1101</td>
<td>1110</td>
</tr>
<tr>
<td>0101</td>
<td>1111</td>
<td>0111</td>
<td>0111</td>
</tr>
</tbody>
</table>

becomes

<table>
<thead>
<tr>
<th>B</th>
<th>E</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>1110</td>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>0101</td>
<td>1111</td>
<td>0111</td>
<td>0111</td>
</tr>
<tr>
<td>1101</td>
<td>1111</td>
<td>0111</td>
<td>0111</td>
</tr>
</tbody>
</table>

when shifted one bit left, and

when shifted one bit to the right.

Thus, a left shift of 0xBEEF yields 0x7AAE when shifted left (by one)
and 0x5F77 when shifted right by one. In both cases, the empty space at
the right (left) side of the pattern is filled with a zero bit. This is some-
times called a \textbf{logical} shift, as opposed to an \textbf{arithmetic} shift, where the the
rightmost (leftmost) bit is duplicated, and a copy stays in the empty spaces.

<table>
<thead>
<tr>
<th>B</th>
<th>E</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>1110</td>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>0101</td>
<td>1111</td>
<td>0111</td>
<td>0111</td>
</tr>
<tr>
<td>1101</td>
<td>1111</td>
<td>0111</td>
<td>0111</td>
</tr>
</tbody>
</table>

becomes

when logically shifted one bit to the right, or

when arithmetically shifted one bit to the right.

Specifically in the case of signed quantities, a logical right shift will al-
ways give a positive result, because a zero is inserted into the leftmost (sign)
2.3. ARITHMETIC CALCULATIONS ON THE JVM

bit. By contrast, an arithmetic right shift will always give a negative result if and only if the initial value were negative, as the sign bit is duplicated in the operation. For unsigned values, a left shift is equivalent to multiplying the value by some power of two, while the logical right shift is equivalent to dividing by some power of two. The usual use of these operations, however, is to put a known set of bits at a particular place in the patterns, for example to use as a operand for later bitwise AND, OR, or XOR operations. The JVM provides three operations to perform such shifts: \texttt{shl} (shift left), \texttt{shr} (arithmetic shift right) and \texttt{ushr} (logical shift right — the mnemonic really stands for “unsigned shift right”), applicable both to ints (32-bit patterns) and longs (64-bit patterns).

**Conversion operations**

In addition to these basic arithmetic and logical operations, there are also a number of unary conversion operations of the form \texttt{2?} — for example, \texttt{i2f}, which converts an int (\texttt{i}) to a float (\texttt{f}). Each of these, in general, will pop the top element off the stack, convert it to the appropriate (new) type, and push the result. This will usually leave the overall size of the stack unchanged, except when the conversion is from a long to a short type (or vice versa). For example, the operation \texttt{i2l} will pop one word (32 bits) from the stack, convert it to 64 bits (two words), and then push the two-word quantity onto the stack, taking two elements. This will have the effect of increasing the depth of the stack by one; similarly, the \texttt{d2i} operation will actually decrease the size of the stack by one.

As before, not all of the possible combinations of types are supported by the JVM, for efficiency reasons. In general, it is always possible to convert to or from an integer. It is also always possible to convert between the basic four types of int, long, float, and double. It’s not directly possible, however, to convert from a char to a float in a single operation, nor from a float to a char. There are two main reasons for this. First, as sub-word types are automatically converted to word-sized quantities, the operation that would be defined as \texttt{b2i} is, in a sense, automatic and cannot even be prevented. Second, conversion of integers to floating point numbers (for example, 2 \leftrightarrow 2.0) will, as discussed earlier, involve not just selecting particular bits from a larger whole, but an entirely different system of representation and changing the fundamental pattern of representation. If for some reason a person needs to convert between a floating point number and a character, it can be done in two steps (\texttt{f2i,i2c}). For analogical reasons, the output of the three operations \texttt{i2s}, \texttt{i2c}, and \texttt{i2b} are a little unusual. Instead of
producing (and pushing) the second type named in the mnemonic, they produce an integer. However, the integer produced will have been truncated to the appropriate size and range. Thus, performing the \texttt{i2b} operation on \texttt{0x24581357} will yield the pattern \texttt{0x00000057}, equivalent to the single byte \texttt{0x57}.

### 2.3.4 Stack manipulation operations

**Typeless stack operations**

In addition to these type-specific operations, the JVM provides some general-purpose and untyped operations for routine stack manipulations. A simple and obvious example is the \texttt{pop} example, which pops a single word from the stack. As the value is only to be thrown away, it doesn’t matter if it’s a int, a float, a byte, or whatever. Similarly, the \texttt{pop2} operation removes two words or a single two-word entry, a long or a double, from the stack. Similar operations of this type include \texttt{dup}, which duplicates and pushes another copy of a single word entry at the top of the stack, \texttt{dup2}, which duplicates and pushes a doubleword entry, \texttt{swap}, which swaps the top two stack words, and \texttt{nop}, short for “no operation,” which does nothing (but takes time, and thus can be useful for causing the machine to wait for a microsecond or so).

In addition, there are a few unusual operations that perform rather specialized stack operations, such as \texttt{dup\_x1}, which duplicates the top word of the stack, and then inserts it beneath the second word — if the stack held, from the top down, (5 3 4 6), then executing \texttt{dup\_x1} would produce (5 3 5 3 4 6). These special operations are listed in Appendix B and will not be discussed further in this text.

**Constants and the stack**

Of course, in order to perform stack-based calculations, some method must be available to put (push) data onto the stack in the first place. The JVM has several such methods, depending upon the type of datum to be pushed, and the location from which the datum comes.

The simplest case is where a single constant is to be pushed into the stack. Depending upon the size of the constant, you can use the \texttt{bipush} instruction (which pushes a one-byte signed integer), the \texttt{sipush} instruction (two-byte signed integer), the \texttt{ldc} instruction (a one-word constant, like an integer, a float, or an address), or the \texttt{ldc2\_w} instruction (for a two-word constant, like a long or a double). So to put the numbers (integers) 3 and 5 onto the stack and then multiply them would require the following code.
The variations

\[
\begin{align*}
\text{bipush} &\quad 5 \\
\text{bipush} &\quad 3 \\
\text{imul} &
\end{align*}
\]

would also accomplish the same thing, less efficiently. Because 5 (and 3) are such small numbers, they will both fit into a single byte, and can be pushed using \textit{bipush}. Also please notice that because multiplication is commutative, it doesn’t matter whether you push the five or the three first. This is not the case for subtraction and division. In these cases, the computer will subtract the second number pushed from the first (divide the first number pushed by the second). So replacing \textit{imul} in the above example by \textit{idiv} would result in dividing 5 by 3. This will leave the value 1 on the stack (not 1.66667, because \textit{idiv} specifies integer division, rounding down).

For efficiency’s sake, there are several special purpose operations that will more quickly push commonly used constants onto the stack. For example, \textit{iconst}_N, where N is one of 0, 1, 2, 3, 4, or 5, will push the appropriate one-word integer onto the stack. Since it’s very common to have to initialize a variable to 1, or to add one to a variable, using \textit{iconst}_1 can be faster than the equivalent \textit{bipush} 1. Thus, we can rewrite the example above to be slightly faster using

\[
\begin{align*}
\text{iconst}_5 \\
\text{iconst}_3 \\
\text{imul}
\end{align*}
\]

Similarly, \textit{iconst}_m1 pushes the integer value -1. There are equivalent shortcuts for floats (\textit{fconst}_N for 0,1, and 2), longs (\textit{lconst}_N for 0 and 1), and doubles (\textit{dconst}_N for 0 and 1).

**Local variables**

In addition to loading constants, one can also load values from memory. Every JVM method has a set of memory locations associated with it that can be accessed freely, randomly, and in any order. As with the stack, the number of memory locations available is not limited by the hardware and can be set by the programmer. Also, as before, the type of pattern loaded
determines the operation and mnemonic; to load an integer, use \texttt{iload}, but to load a float, use \texttt{fload}. Either of these will retrieve the appropriate variable and push its value at the top of the stack.

Variables are referred to by sequential numbers, starting at zero, so if a given method has 25 variables, they will have numbers from 0 to 24. Each variable stores a standard word-sized pattern, so there is no typing of variables. Storage of doubleword patterns (longs and doubles) is a little bit more tricky, as they each need occupy two adjacent variables. Loading a double from variable 4, for instance, will actually read the values from both variables 4 and 5. In addition, the JVM allows several shortcuts of the form \texttt{?load\_N}, so one can load an integer from local variable zero (#0) either by \texttt{iload 0} or the shortcut \texttt{iload\_0}. This shortcut exists for all four basic types, and all variable numbers from zero to three.

Similarly, data can be popped from the stack to be stored in local variables for later use. The command in this case would be \texttt{?store} where the first character, as usual, is the type to be stored. As before, storing a long or a double will actually perform two pop operations and store in two adjacent variables, so the instruction \texttt{dstore 3} would remove two, not one, elements from the stack, and cause changes to both local variables #3 and #4. Also as before, shortcuts exist of the form \texttt{?store\_N} for all types, with \texttt{N} varying between 0 and 3.

\section*{2.3.5 Assembly language and machine code}

Let’s look, then, at a simple code fragment and see how the various conversions would take place. We’ll start with a single, simple, high-level language statement (if the notion of “simple high-level language statement” isn’t a contradiction in terms):

\[
x = 1 + 2 + 3 + 4 + 5;
\]

This statement (obviously?) calculates the sum of the constant numbers 1 through 5, and stores them in a local variable named \texttt{x}. The first problem: the JVM doesn’t understand the idea of named local variables, only numbered ones. The compiler needs to recognize that \texttt{x} is a variable and allocate a corresponding number (we’ll assume we’re using #1, and that it’s an int). One way of writing code to perform this task (there are lots of others) is as follows:

\begin{verbatim}
; x = 1 + 2 + 3 + 4 + 5;
; translate to : 1 2 + 3 + 4 + 5 +, then load into #1
\end{verbatim}
2.3. ARITHMETIC CALCULATIONS ON THE JVM

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Machine code byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>iconst_1</td>
<td>0x04</td>
</tr>
<tr>
<td>iconst_2</td>
<td>0x05</td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
</tr>
<tr>
<td>iconst_3</td>
<td>0x06</td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
</tr>
<tr>
<td>iconst_4</td>
<td>0x07</td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
</tr>
<tr>
<td>iconst_5</td>
<td>0x08</td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
</tr>
<tr>
<td>istore_1</td>
<td>0x3c</td>
</tr>
</tbody>
</table>

Table 2.2: Translation of program fragment #1 into bytecode

iconst_1 ; load constant value 1
iconst_2 ; load constant value 2
iadd ; add
iconst_3 ; load constant value 3
iadd ; add
iconst_4 ; load constant value 4
iadd ; add
iconst_5 ; load constant value 5
iadd ; add
istore_1 ; store in x

This is the main task of the compiler, to convert a high-level statement into several basic operations. At this point, it is the task of the assembler (or of a different part of the compiler) to convert each operation mnemonic into the corresponding machine code byte. The correspondences are given in appendices B and C, but can are summarized in table 2.2.

The corresponding machine code would thus be the byte sequence 0x04, 0x05,0x60, 0x06,0x60, 0x07,0x60, 0x08,0x60, 0x3c, which would be stored on disk as part of the executable.

Translation is not always this simple, because some operations may take more than one byte. For example, the bipush instruction pushes a byte onto the stack (just as iconst_0 pushes the value 0). But which byte? The bipush instruction itself has the value 0x10, but it is always followed by a single byte, telling what needs to be pushed. To push the value 0, the compiler could also use bipush 0, but this would assemble to two successive bytes: 0x10, 0x00. We thus have an alternate version of the program as
CHAPTER 2. ARITHMETIC EXPRESSIONS

<table>
<thead>
<tr>
<th>Revised mnemonic</th>
<th>Machine code byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>bipush 1</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td>0x01</td>
</tr>
<tr>
<td>bipush 2</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td>0x02</td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
</tr>
<tr>
<td>bipush 3</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
</tr>
<tr>
<td>bipush 4</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td>0x04</td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
</tr>
<tr>
<td>bipush 5</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td>0x05</td>
</tr>
<tr>
<td>iadd</td>
<td>0x60</td>
</tr>
<tr>
<td>istore_1</td>
<td>0x3c</td>
</tr>
</tbody>
</table>

Table 2.3: Translation of program fragment #2 into bytecode

given in table 2.3. Notice, however, that this second version is about 50% longer and therefore less efficient.

2.3.6 Illegal operations

Because both the stack and the local variables only store bit patterns, there is a very great danger for confusion on the part of the programmer. This can get especially tricky with pushing constants, because types of constants are not explicitly marked in the mnemonic. A command to put the (integer) value 10 into the stack would be `ldc 10`. The command to put the floating point value 10.0 into the stack would be `ldc 10.0`. Because most JVM assemblers are smart enough to figure out that 10 is an integer and 10.0 is a floating point number, the correct bit pattern will be pushed in either case. However, these bit patterns are not the same. Attempting to push 10.0 twice and then execute an `imul` instruction will almost certainly not give you the value 100.0 (or even 100). Trying to perform arithmetic on a float as though it were an integer, or an integer as though it were a float, is an error. In the best case, the machine will complain. In the worst case, the machine won’t even notice, and give answers that are completely, mysteriously, and untraceably wrong.

It’s equally an error to attempt to access the top (or bottom) half of a
doubleword variable, a long or a double, as though it were a single word. If you’ve stored a double into local variables #4 (and #5), you can’t then load an integer from local variable #5 (or #4). Again, at best the machine will complain, and at worst will silently give meaningless and horribly wrong results. It’s also an error to attempt to pop from an empty stack, to load from (or store into) a variable that doesn’t exist, and so forth.

One of the chief advantages of the JVM is that it has been designed (as will be discussed in a later chapter) to catch these sorts of errors as they occur in a running program, or even as the program is written, so that the programmer cannot get away with these sorts of errors. This has the effect of increasing the security and reliability of JVM programs tremendously. However, a good programmer should not rely on the ability of the computer to catch her errors. Careful planning and careful writing of code is a much better way to make sure that the computer gives you correct answers.

2.4 An Example Program

2.4.1 A annotated example

Returning to the story problem that opened the chapter, the question becomes not only what the answer is, but how it can be implemented on the machine under discussion (the JVM). To briefly recap, the problem was

What is the volume of a circular mountain 450m in diameter at the base and 150m high?

and the formula as it would be written on a chalkboard looks like

$$\frac{1}{3} \cdot \left[ \pi \cdot \left( \frac{450}{2} \right)^2 \right] \cdot 150$$

What sequence of JVM instructions would solve this problem?

First, notice that we need to calculate the floating point value 1/3. Integer division will not work, since 1/3 is 0, while 1.0/3.0 is 0.333333. Thus, we push the two elements 1.0 and 3.0, and execute a division.

```
ldc 1.0
ldc 3.0
fdiv
```
We then push the known value of pi.

\[ \text{ldc 3.141593} \]

To calculate the radius, we simply push 450, push 2, and then divide. Note that 450 is too big to store in a single byte (which only goes up to the integer value 127, so we have to use \text{sipush}.

\[ \begin{align*}
\text{sipush 450} \\
\text{bipush 2} \\
\text{idiv}
\end{align*} \]

To square the radius, we can either recalculate \( \frac{450}{2} \), or, more efficiently, use the \text{dup} instruction to copy the top of the stack and multiply.

\[ \begin{align*}
\text{dup} \\
\text{imul}
\end{align*} \]
2.4. AN EXAMPLE PROGRAM

Next, push the height of 150 and multiply.

```
sipush 150
imul
```

The integer value at the top of the stack must be converted to a floating point number.

```
i2f
```

and then two (floating point) multiplications will calculate the final answer and leave it on the top of the stack.

```
fmul
fmul
```

This entire process will be stored as a sequence of machine code instructions. Each one will individually be fetched (from memory) and executed. As the statements are in sequence, the instruction which is next fetched will be the sequentially next instruction — and thus this entire complex series of statements will do as expected and desired. A similar process can be used to perform any calculation within the basic operations available in the instruction set.

2.4.2 The final JVM code

```
; calculate 1/3
ldc 1.0
ldc 3.0
fdiv

; push pi
ldc 3.141593

; calculate radius
```
CHAPTER 2. ARITHMETIC EXPRESSIONS

```
sipush 450
bipush 2
idiv

; and square it
dup
imul

; push height
sipush 150

; multiply height times radius-squared
imul

; convert to floating point
i2f

; and multiply times pi and 1/3 previously calculated
fmul
fmul
```

2.5 JVM Calculation Instructions Summarized

MARK ME NOTE TO READER ANYTHING ELSE NEEDED?

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>short</th>
<th>char</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>iadd</td>
<td>ladd</td>
<td>fadd</td>
<td>dadd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subtraction</td>
<td>isub</td>
<td>lsub</td>
<td>fsub</td>
<td>dsub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multiplication</td>
<td>imul</td>
<td>lmul</td>
<td>fmul</td>
<td>dmul</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>division</td>
<td>idiv</td>
<td>ldiv</td>
<td>fdiv</td>
<td>ddiv</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remainder</td>
<td>irem</td>
<td>lrem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>negation</td>
<td>ineg</td>
<td>lneg</td>
<td>fneg</td>
<td>dneg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logical AND</td>
<td>iand</td>
<td>land</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logical OR</td>
<td>ior</td>
<td>lor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logical XOR</td>
<td>ixor</td>
<td>lxor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l. shift</td>
<td>ishl</td>
<td>lshl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r. shift (a)</td>
<td>ishr</td>
<td>lshr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r. shift (l)</td>
<td>iushr</td>
<td>lushr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.6 Chapter Review

- Computers are like calculators, in that they can only do the few actions allowed by their hardware. Complex calculations that cannot be done in a single operation or button press must be done as a sequence of several elementary steps within the allowed actions.

- Conventional math, as written on the blackboard, uses infix notation, where operations like division are written between their arguments. Some calculators and computers, the JVM among them, use postfix notation instead, where the operation comes after the arguments.

- Postfix notations can be easily described and simulated using a standard data structure called a stack.

- The basic operation of any computer is the fetch-execute cycle, during which instructions are fetched from main memory, interpreted, and executed. This cycle repeats more or less without end.

- The CPU holds two important pieces of information for the fetch-execute cycle: the Instruction Register (IR), which holds the currently executing instruction, and the Program Counter (PC), which holds the location of the next instruction to be fetched.

- There are two major philosophies of computer design: Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC). These are typified by the Intel Pentium and the Apple/IBM/Motorola PowerPC, respectively.

- The JVM uses typed stack-based computations to perform most of its arithmetic. Mnemonics describe both the operation to be performed and the type of data used. It is an error to perform an operation on a piece of data of the wrong type.

- The JVM also provides some shortcut operations for commonly used operations such as loading a value of zero into the stack.
Simple sequencing of elementary mathematical operations can nonetheless perform very complex computations.

2.7 Exercises

1. What are the advantages to having a $\sqrt{x}$ button on a calculator? What are the disadvantages?

2. What would be the sequence of operations to calculate $(7 + 1) \cdot (8 - 3)$ on a normal (infix notation) calculator? How about on a RPN (postfix) calculator?

3. Is there a corresponding sequence of operations using prefix notation to perform the calculation above. If yes, what is it? If no, why not?

4. Is the fetch-execute cycle more complex on a CISC or a RISC computer? Why?

5. What is the difference between typed and untyped calculations? Can you give two examples of each?

6. What are the advantages and disadvantages of typed arithmetic calculations?

7. Why doesn’t the instruction cadd exist?

8. Which of the following are illegal and why?
   - bipush 7
   - bipush -7
   - sipush 7
   - ldc -7
   - ldc2 -7
   - bipush 200
   - ldc 3.5
   - sipush -300
   - sipush -300.0
   - ldc 42.15

9. How can a shift operation be used to multiply an integer by 8?
10. Describe two ways to get (only) the smallest eight bits of a 64-bit long integer?

11. Is there any basic operation that will work on both an int and a float? How about on both an int and a long?

12. In the ?div instruction, is the number divided by stored at the top of the stack, or the second element?

13. The surface area of a sphere is four times the area of a circle of equivalent radius. Write a postfix expression to calculate the surface area of a hemispherical dome of radius $R$.

14. Write a postfix expression for the arithmetic mean of five numbers, $a$, $b$, $c$, $d$, and $e$.

15. Prove that for every infix expression there is an equivalent postfix expression and vice versa.

### 2.8 Programming Exercises

1. Write a program to interpret a postfix expression and to print the resulting value.

2. Write a program to read in an infix expression and to write out the corresponding equivalent postfix expression.

3. Write a program to read a sequence of JVM instructions and to determine the maximum stack height that would result if they were executed, starting from an empty stack.

4. Write a program to read a sequence of JVM instructions and to determine if any of the instruction will ever attempt to pop from an empty stack. (Note: this is actually one of the tasks performed by the verifier in a real system.)
Chapter 3

Assembly Language Programming in jasmin

3.1 Java, the programming system

As discussed in the first chapter, there is no reason in theory why a program written in Java would need to be run on the JVM, or why a JVM program could not have been written in any high-level language such as C++. In practical terms, though, there is a strong connection between the two, and the design of Java, the language, has strongly influenced the designs both of the Java Virtual Machine and the assemblers for that machine.

In particular, Java strongly supports and encourages a particular style of programming and design called object-oriented programming (abbreviated OOP). As one might suspect from the name, OOP is a programming technique that focuses on objects — individual, active elements of the world (or a model of the world), each with their own set of actions that they can perform or that can be performed upon them. Objects in turn can be grouped into classes of similarly-typed objects that share certain properties by virtue of their types. For example, in the real world, “cars” might be a natural class, and with very few exception, all cars share certain properties : they have steering wheels, gas pedals, brakes, headlights. They also share certain actions : I can turn a car left or right (by turning the wheel), make them slow down (by pressing the brake), or make them stop altogether (by running out of gas). More strongly, if someone said they had just bought a new car, you would assume that this car came with a steering wheel, brake, gas tank, and so forth.

Java supports this style of programming by forcing all functions to be
attached to classes, and forcing all executable program code to be stored as separate (and separable) “class files.” These class files correspond fairly closely to Linux executable files or Windows .EXE files, except that they are not necessarily guaranteed to be complete and functional programs in their own rights. Instead, a particular class file contains only those functions necessary for the operation of that particular class; if it, in turn, relies on the properties and functions of another kind of object, then those would be stored in a class corresponding to the second kind of object.

The other main difference between a Java class files and a typical executable is that the Java class file is supposed to be portable between different machine types. As such, it’s written not in the machine language of the host machine, but in the machine language of the Java Virtual Machine. This is sometimes called bytecode to distinguish it as, specifically, not being tied to a particular machine. As such, any class file, compiled on any machine, can be freely copied to any other machine and will still run. Technically, the JVM bytecode will only run on a copy of the JVM (just like a Windows executable will usually only run on a Windows computer), but the JVM is enabled, via software, on almost every hardware platform.

When a bytecode file is to be executed, the computer is required to run a special program to load the class from disk into the computer’s memory. In addition, the computer will usually perform other actions at this time, for example: loading other classes that are needed to support the class of interest, verifying the structural soundness and safety of the class and its methods, and initializing static and class-level variables to the appropriate values. Fortunately from the point of view of a user or programmer, these steps are all a built-in part of the JVM implementation, and the user doesn’t need to do anything.

Running Java programs is thus a three step process. After the program source code is written, it must be compiled or converted into a class file. The user must then create a (software) instance of the Java Virtual Machine in order to execute the bytecode. The exact commands to do this will vary from system to system. On a typical Linux system, for instance, the command to execute the JVM would look like this

```
java TheClassOfInterest
```

This will look for a file named `TheClassOfInterest.class`, and then proceed to load, verify, and initialize it. It will then look for a specific method in that class called `main()`, and attempt to invoke that method. For this reason, any standalone Java application class must contain a “main”
3.2. USING THE ASSEMBLER

method. On many other systems (Windows and MacOS, for example), just clicking on the icon corresponding to the `.class` file will launch a JVM and run the class file. In addition, certain kinds of class files can be run directly from a Web browser such as Microsoft’s Internet Explorer or Netscape Communications’ Navigator.

However, none of these programs will actually run Java programs, only JVM bytecode. There are a number of compilers available that will convert high-level Java code into JVM bytecode, and, perhaps unsurprisingly, there are also programs available that will convert other languages (than Java) into JVM bytecode. Here, we will focus specifically on one particular kind of language, where each bytecode statement is uniquely associated with a single statement in the program source code. As discussed in the first chapter, this kind of language (where source code statements and machine instructions are in a 1:1 relationship) is usually called assembly language and the program to convert from one to the other is called an assembler. (The corresponding conversion program for a high-level language is usually called a compiler.)

3.2 Using the Assembler

3.2.1 The assembler

As might be expected, the conversion process between assembly language and machine code is fairly straightforward. As a result, the program itself is also fairly easy to write. The task of an assembler is simple enough that there are usually several different ones available to choose from, often with very slight differences between them. Sun has not established an official, standard assembler for the JVM, so in this book, the example programs have been written specifically for the `jasmin` assembler. This program was written in 1996 by Jon Meyer and Troy Downing, of the NYU Media Research Laboratory. This program is available for free download from http://mrl.nyu.edu/~meyer/jasmin and has become a de-facto standard format for assembly language for the JVM.¹ The `jasmin` program is also available at the companion website to this book: http://prenhall.com/juola. The first step, then, is of course to get and install `jasmin` on the machine you are working on. For simplicity, since “Java Virtual Machine assembly language” is such a mouthful, we’ll call this language `jasmin` as well.

3.2.2 Running a program

In order to execute the program in figure 1.11, duplicated here as figure 3.1, it must first be typed into a machine-readable form (like a text file). You can use any editing program you like for this, from simple editors like Notepad up to complicated and feature-ridden publishing packages. Bear in mind, though, that assemblers can almost never handle fancy formatting, or font changes, so save it as plain text. By authorial tradition, programs written in jasmin are usually saved with a .j extension, so the program above would have been written and saved as jasminExample.j to the disk somewhere.

In order to run the program, the same steps must be followed as in executing a Java program. After the program has been written in text format, it must be converted from (human-readable) jasmin syntax into JVM machine code. Second, the JVM (Java run-time engine) must be run to allow the JVM code to be executed. To do the first, (for an appropriately configured Linux machine), simply type

    jasmin jasminExample.j

at the appropriate command prompt. This will execute the assembler (jasmin) on the file and produce a new file, named jasminExample.class, which contains JVM executable code.

This .class file is a standard part of the Java run-time system, and can be run by any of the usual means; the simplest is to type

    java jasminExample

which will spawn an instance of a JVM process and execute the jasminExample.class file (specifically, the main method defined in that file) on this virtual machine.

This or a similar process will work on most machines and on all examples in this book (except for the ones with deliberate errors in them, of course.)

3.2.3 Display to the console vs. a window

Running the program as described in the prior section uses a rather old-fashioned and unpopular style of interfacing. Most modern programmers prefer to use windows interfaces and interact with their computer via mouse presses than to use a command-line and text-based interface. A big reason for the popularity of Java is the extensive support it provides for windowed and networked applications. However, there are slight differences in how these kind of applications interact with the outside world.
3.2. USING THE ASSEMBLER

`; defines the class file associated with this as jasminExample.class`
.class public jasminExample
`; defines jasminExample as a subclass of Object
.super java/lang/Object

`; boilerplate needed for object creation
.method public <init>()V
   aload_0

   invokespecial java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
 ; we need two stack elements, for System.out and the string
 .limit stack 2

 ; find System.out (an object of type PrintStream)
 ; and put it on the stack
 getstatic java/lang/System/out Ljava/io/PrintStream;

 ; find the string (characters) to be printed
 ; and put it on the stack
 ldc "This is a sample program."

 ; invoke the PrintStream/println method
 invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

 ; ... and that’s it!
 return
.end method

Figure 3.1: Sample program in JVM assembly language
The most popular kind of Web application for Java is called an applet. As the name suggests, this is a small, rather lightweight application, specifically designed for interoperation with web browsers, and as such, all major browsers support Web pages incorporating JVM applets. Figure 3.2 shows a very simple example of a Web page, the only contents of which are an \textit{APPLET} tag. The effect of this page is that, when the page is displayed on a browser, the browser will also download and run the applet. The exact method of running the applet are different. Instead of invoking the “main” method, the browser will invoke a “paint” method as the starting point for the code. In addition, the output instructions (such as \textit{println}) are replaced by graphics-specific instructions such as \textit{drawstring} that take not only a particular string to be drawn, but also parameters such as the location (in the window) at which to display the string.

The details of applet programming are a fine art and require some knowledge of the applet-specific functions. Using these functions, however, a skilled-programmer can created detailed pictures, or text in any font, size, shape, and orientation she desires. As is shown in figure 3.3, the overall structure of a jasmin program does not change much, regardless of whether the program was written as an applet, to output to a window, or written as a standalone application instead, to output to the console.

As before, the program (\textit{jasminAppletExample.j}) would be assembled using the jasmin program to produce a class file.

\texttt{jasmin jasminAppletExample.j}

Once the class file \textit{jasminAppletExample.class} has been created, the applet can be run by simply opening the example Web page shown in figure 3.2. This can be opened in any Web browser, for example in Internet Explorer or Netscape, or using a special program such as \textit{appletviewer} supplied with the Java system by Sun. Using technology such as this, Java
3.2. USING THE ASSEMBLER

; defines jasminAppletExample as a subclass of Applet
.class public jasminAppletExample
.super java/applet/Applet

; boilerplate needed for Applet creation
; note similarity to Object creation in previous example
.method public <init>()V
    aload_0
    ; this isn’t an Object, so we have to invoke the Applet <init>
    invokespecial java/applet/Applet/<init>()V
    return
.end method

; Note that Applets start at paint() method instead of main()
; also notice the subtly different definition
.method public paint(Ljava/awt/Graphics;)V
    ; we need 4 stack elements:
    ; the Graphics object
    ; the string to be printed
    ; the x-coordinate of print location
    ; the y-coordinate of print location
    .limit stack 4
    ; Graphics object stored in local #1
    .limit locals 2

    ; This boilerplate is a little unusual as it’s harder to draw
    ; text in an Applet than on System.out
    ; load the four parameters
    aload_1 ; this is the Graphics object passed as a parameter
    ldc "This is a sample applet" ; a string to be printed
    bipush 30 ; the coordinates at which this string prints
    bipush 50

    ; and invoke the drawString method
    invokevirtual java/awt/Graphics/drawString(Ljava/lang/String;II)V

    ; ... th’th’th’that’s all, folks!
    return
.end method

Figure 3.3: A sample applet in jasmin
CHAPTER 3. ASSEMBLY LANGUAGE PROGRAMMING IN JASMIN

(and jasmin) programs can be supplied as executable code to be used by any JVM on a machine anywhere in the world.

3.2.4 Using System.out and System.in

When programming in any assembly language, getting the computer to read and write data can be among the most challenging tasks. This is related to the more general problem of simply interacting with I/O peripherals; because of the wide variety of peripheral types (reading from the network is very different from reading from the keyboard), and even more annoyingly, the wide variety among a given peripheral type (does your keyboard have a numeric keypad or not?), every device may have to be approached on its own individual basis.

The Java class system provides some mitigation for this problem. Just as one can steer an unfamiliar car, because all cars have steering wheels, and they all work about the the same way, so Java defines a `PrintStream` class that includes methods named `print` and `println`. The JVM always defines a particular PrintStream named `System.out`, which is attached to a default print-capable device.

This provides a relatively simple method of generating output, using either the `print` or `println` methods. This has already been demonstrated in the sample program (figure 3.1) for printing of String types, but can be extended to print any type supported by the method (with a few changes). The necessary steps are presented here largely without explanation — you are not necessarily expected to understand them right now. To understand them fully will require a deeper investigation of both the JVM type and class system and how they are represented, and we’ll return to this simple example, at length, in chapter 10.

First, the `System.out` object must be pushed onto the stack from its static and unchanging location in the system.

```
getstatic java/lang/System/out Ljava/io/PrintStream;
```

Second, the data to be printed must be loaded onto the stack via any of the usual ways presented in the previous chapter.

```
iload_2 ; this loads local variable #2 AS AN INTEGER!
```

Third, the `println` method must be invoked, including a representation of the type pushed onto the stack in the second step. Since the statement `iload_2` pushes an integer, the command would be
invokevirtual java/io/PrintStream/println(I)V;

If, instead, we had pushed a float (perhaps with fload.2), the command would be modified to include an F instead of an I as follows

invokevirtual java/io/PrintStream/println(F)V;

For printing out a character string, the complex line given in the sample program is needed.

invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V;

If you find this confusing, don't worry about it for now. Classes and class invocation will be discussed at length in chapter 10. For now, these statements can be treated as a kind of legal boilerplate or black magic. Any time you want to generate output, just use the appropriate version. A similar, but more complicated set of boilerplate can be used to get input from a similarly constructed System.in object, but this will also be deferred until we have a better understanding of classes in general. At this point, understanding the basic statements is more important to getting working jasmin programs.

With the newest version of Java, (Java 1.5), an entirely new method of doing console input and output has become available, using newly defined classes such as Scanner and Formatter, and a newly-structured printf method for formatted output. From the perspective of the underlying assembly language, these are treated simply as new functions/methods to be invoked, and do not involve any radical changes in technology.

### 3.3 Assembly Language Statement Types

#### 3.3.1 Instructions and comments

Assembly language statements can be divided roughly into three types. (In particular, this is also true of jasmin statements.) The first, instructions, correspond directly to instructions in the computer’s machine language or bytecode. In many cases, these are produced by looking them up in a table stored in the assembler; the bytecode corresponding to the mnemonic iconst.0 is the bit pattern 0x03. In addition, most assemblers allow the use of comments to allow the programmer to insert reminders and design notes to help herself understand, later, what she is doing today. In jasmin, any part of the line that follows a semicolon (;) is a comment, so the first
two lines in figure 3.1 are, in their entirety, comments. The assembler is programmed to ignore comments as though they weren’t there, so their content is skipped in the assembly process, but they are visible in the source code.

The jasmin program is actually a little unusual among assemblers in terms of the freedom of formatting it permits its programmers. Assembly language program statements usually have a very stereotyped and inflexible format that looks something like this:

Label:
\[ \text{mnemonic \ argument(s)} \quad ; \text{Comment} \]

The mnemonic/argument combination has already been seen, for example, in statements like \texttt{i1oad 2}. Depending upon the type of mnemonic, there may be any number of arguments from zero on up, although zero, one, and two are the most common. The label will be discussed in detail in a later chapter; for now, it simply marks a part of the program so that you can go back and repeat a section of code. Finally, this stereotyped statement contains a comment. Technically, the computer will never require you to comment programs. On the other hand, your teacher almost certainly will – and good programming practice demands comments. Good assembly language programming, in particular, usually demands at least one comment per line.

This book, and jasmin, take a slightly non-standard view on comments. Because many of the arguments used in jasmin programs can be long, especially string arguments and the locations of standard objects such as the system output, there may not be room on a line to put comments related to that line. A more serious problem with the one comment per line standard is that it can encourage poor and uninformative commenting.

As an example, consider the following line:

\texttt{bipush 5 \quad ; load the int value 5 onto the stack}

The comment, in this case, tells the programmer little or nothing. The statement \texttt{bipush 5}, after all, means “load the int value 5 onto the stack.” Any programmer reading the statement, even in isolation, would know this. In order to understand the program, what she probably needs to know are larger-scale issues. Why does the particular value 5 need to be pushed onto the stack at this particular step (and why should it be loaded as an int)? By focusing on the large-scale roles of statements and the meanings of
blocks and groups of statements, comments are rendered (more) useful and informative.

```
bipush 5 ; load number of pentagon sides to measure
```

### 3.3.2 Assembler directives

The third kind of statement, called a **directive**, is an instruction to the assembler itself, telling it how to perform its task. In jasmin, most directives begin with a period (.), as does the third line (or first non-comment line) of the sample program. This directive (**.class**), for example, informs the assembler that this file defines a particular class named **jasminExample**, and therefore (among other things) the name of the class file to be created is **jasminExample.class**. This doesn’t directly affect the process of converting program instructions to bytecode (and doesn’t correspond to any bytecode instructions), but does directly inform jasmin how to interact with the rest of the computer, the disk, and the operating system.

Many of the directives may not at this point have a particularly clear meaning. This is because the JVM and the class files themselves tie directly into the object-oriented structure and class hierarchy. For example, all classes must be tied into the class hierarchy, and must, in particular, be subtypes of another class. (A Mercedes, for example, is a subtype of car, while a car is a subtype of vehicle, and so one). The Java language enforces this by making any class that does not explicitly mention its relationship in the hierarchy, by default, to be a subtype of Object (**java/lang/Object**). The jasmin assembler enforces a similar requirement, in that any jasmin-defined class must contain a **.super** directive defining the superclass of which the new class is a subtype. The programmer can often, without harm, simply copy this same line

```
.super java/lang/Object
```

from jasmin program to jasmin program.

Other directives (**.method,.end method**) are used to define the interactions between this class and the rest of the universe. In particular, the OOP model enforced by the JVM demands that calls to functions from outside classes be explicitly defined as “public methods” and strongly encourages that all functions be so defined. The details of this sort of definition will be explored in greater detail in a following chapter.
3.3.3 Resource directives

The most important directive, from the point of view of the jasmin programmer, is the \texttt{.limit} directive. This is used to define the limits, and by extension the availability, of resources for computation within a single method. It is one of the unique and very powerful aspects of the JVM’s status as a virtual machine, in that methods can use as much, or as little, resources as they need.

In particular, a typical stack-based microprocessor or controller (the old Intel 8087 math coprocessor chip, later incorporated into the 80486 and beyond as part of the main CPU, is a archetypical example) would hold only a few elements (8, in this case). A complex computation that needed more than 8 numbers would need to store some in the CPU stack and some elsewhere, such as in main memory. The programmer would then be tasked with making sure that data was moved in and out of memory as necessary, with the price of failure usually being a buggy or entirely dysfunctional program. Increasing the amount of stack space available inside the CPU could solve this problem, but only by making each individual CPU chip larger, hotter, more power-hungry, and more expensive. Furthermore, changing fundamental chip parameters between versions will introduce incompatibilities, where newer programs cannot run on legacy hardware.

The corresponding solution on the JVM is characteristically clean. The directive

\begin{verbatim}
.limit stack 14
\end{verbatim}

as a statement immediately inside a defined method (using the \texttt{.method} directive) will set the maximum size of the stack to 14 int- or float-sized elements. (Of course, this will also store 7 long- or double-sized double elements, or 5 long/double and 4 int/float elements, or any appropriate combination.) Similarly, the maximum number of (int-sized) local variables in a method can be set to 12 by use of the related directive

\begin{verbatim}
.limit locals 12
\end{verbatim}

If either of these directives is omitted, then a default limit of one item, enough for a single int or float, and not enough for a larger type, will be applied.
3.4 Example: Random Number Generation

3.4.1 Generating pseudorandom numbers

A common, and yet mathematically sophisticated task that computers are often called upon to perform is the generation of apparently “random” numbers. For example, in computer games, it may be necessary to shuffle a deck of cards into apparently random order. Because of several fundamental limitations of present-day computer hardware, computers are not actually capable of generating “random” data (in the strict sense that a statistician would insist upon). Instead, computers generate deterministic pseudorandom data that, although technically speaking can be predicted, appears to be naively unpredictable.

Specifically, we focus here on the task of generating integers uniformly over the range of 0 to $n$. If, for some reasons, the user wishes to generate random floating point numbers, this can be done by simply dividing the random integer by $n + 1$. If $n$ is large enough, this will give a good approximation of a uniform distribution of reals over the interval $[0,1)$. (For example, if $n$ is 999, the floating point number will be one of the set \{0.000, 0.001, 0.002, \ldots, 0.999\}. If $n$ is a billion, the final number will look pretty random.)

Mathematically, the computer takes a given number (the seed) and returns a related but apparently unpredictable number. The usual method of doing this, which is followed here, is to use a so-called linear congruential generator. With this method, the number to be returned is generated by an equation of the form

$$newvalue = (a \cdot oldvalue + c) \mod m$$

for particular values of $a$, $c$, and $m$. The parameter $m$, for example, determines the maximum size of the returned random number, as computations mod $m$ will give a highest answer of $m - 1$, hence $n = m - 1$. There is a lot of theoretical research behind the section of the “best” values for $a$ and $c$, and to investigate this fully would take us too far afield. The value of $oldvalue$ must be selected anew every time the generator is run, as the value of $newvalue$ strictly depends upon it. However, this generator can be used repeatedly to generate a sequence of (pseudo)random numbers, and so needs to be seeded only once per program. Typical sources for the initial seed would include truly random (to the program) values such as the current time of day, the process ID number, the most recent movements of the mouse, and so forth.
3.4.2 Implementation on the JVM

In order to implement this algorithm on the JVM, a few design decisions need to be taken. For practical reasons, the value of oldvalue will probably be stored in a local variable of some sort as it is likely to change from call to call, but the values of a, c, and m can be stored and manipulated as constants. For simplicity, the values of both oldvalue and newvalue will be stored as ints, in a single stack element, but the intermediate values, especially if a and c are large, may overflow a single stack element and will have to be stored as the long type. Without explanation, we use the largest prime value that can be stored as a (signed) int (2147483647) as our value for m, and select the prime values \(2^{16} + 1 (= 65537)\) for a and 5 for c.

The calculations themselves are straightforward. The expression above

\[(a \cdot oldvalue + c) \mod m\]

can be expressed in JVM (reverse Polish) notation as

\[a \cdot oldvalue \cdot c + m \mod\]

Therefore, appropriate JVM instructions would be as

```java
; compute a * oldvalue
ldc2_w 65537 ; a is \(2^{16} + 1\), stored as a long
iload_1 ; oldvalue is (assumed) stored as local variable #1
i2l ; convert oldvalue to a long
lmul ; and multiply

; add c
ldc2_w 5 ; c is 5, stored as a long
ladd ; and add to a*oldvalue

; and take the remainder \(\mod m\)
ldc2_w 2147483647 ; load value for m
lrem ; calculate modulus
l2i ; convert back to integer

; newvalue is now left at top of stack
```

By inspection, maximum depth of the stack needed in these calculations is two long-sized (double) stack elements, so this could be executed in any
method with a stack limit of 4 or greater. Similarly, the code as presented assumes that oldvalue is stored in local variable #1. Because variables are numbered starting at zero, this means that the program will require two local variables. (The reason for assuming that oldvalue is stored in variable #1 is because in some cases, local variable #0 is reserved by the Java class environment.)

In order for this program to run properly, the method containing this code would need the two directives

```
.limit stack 4
.limit locals 2
```

There are several variations on this code that would also work; like most programming problems, there are several correct solutions. Most obviously, the two directives above could be reversed, defining the local variables first and the stack size second. A more sophisticated change would have the calculation be performed using a different order of operations, perhaps pushing

---

**SIDEBAR : PARAMETER PASSING, LOCAL VARIABLES, AND THE STACK.**
Most programs need input to be useful. In fact, most functions and methods need input to be useful. The normal method of getting information into a method is by passing parameters: for example, the usual definition of sin takes a single formal parameter. When the sin function is used, the corresponding actual parameter is passed to the function and used for computation.

The Java Virtual Machine has a rather odd way of handling this process. In traditional (chip-based) architectures, the computer uses a single shared "stack" in memory to separate memory areas used by different programs or functions. The JVM, in contrast, provides a unique and private stack to every method. This keeps one method from running amok and destroying data sacred to other parts of the program (which enhances security tremendously), but does make it difficult for one function to pass data to another function. Instead, when a function/method is invoked, the parameters are placed (by the JVM) in local variables available to the method. In general, the first parameter will be placed in local variable #1, the second in #2, and so forth.

There are three exceptions to this general rule. First, if a parameter is too big to fit in a single stack element (a long or a double), then it will be placed in two successive elements (and all the later elements shifted down an additional element). Second, this rule leaves local variable #0 free. Normally (with instance methods), a reference to the current object will be passed in #0. Methods defined as static have no current object, and thus start pass the first parameter in local #1, and so forth.

Finally, Java 1.5 defines a new parameter passing method to use when you have a variable number of arguments. In this case, the variable arguments will be converted to an array and passed as a single array argument (probably in #1); the called method will be responsible for determining how many arguments were actually passed and operating upon them properly.

The use of the stack for parameter passing on machines other than the Java Virtual Machine will be described in detail in the later, machine-specific, chapters.
c first, doing the multiplication, and then adding. This would technically be
an implementation of the equivalent but different RPN expression

\[ c \cdot a \cdot \text{oldvalue} \cdot +m \mod \]

If this implementation is chosen, though, the maximum depth of the
stack will be three (long) elements, requiring a `.limit stack 6` directive.

Similarly, there are equivalent ways to perform many of the smaller steps.
Instead of pushing the value 5 as a long directly (using `ldc2_w 5`), the pro-
grammer could have pushed the int value five (`iconst 5`) and then converted
it to a long (`i2l`). This would exchange one instruction for two, but the two
operations replaced may be shorter and quicker of execution. Rarely, how-
ever, do minor changes like this have substantial effects on the size and
running speed of a program; more often they are simply different ways to
perform the same task, at a risk of confusing a novice programmer who
expects there to be a single solution to a given problem.

3.4.3 Another implementation

Not only are there multiple solutions to the random number generator pre-
sented above, but there are also many different algorithms and parameters
that can solve the problem. A detailed examination of the representation
scheme used by the JVM can allow a streamlined and more sophisticated
random number generator. In particular, because mathematics involving
ints is always performed using 32-bit quantities, taking numbers mod 2^{32} is
automatic. By setting \( m \) to be (implicitly) \( 2^{32} \), the programmer can avoid
the part of the computation involving taking the modulus. Furthermore,
there is no need to use long-sized storage or stack elements if all computa-
tions are going to be implicitly done in this modulus.

One proposed set of numbers that may produce a good random number
generator using this modulus is setting \( a \) to 69069 and \( c \) to 0. (These num-
bers are actually part of the “Super-Duper” generator as proposed by the
researcher George Marsaglia.) By setting \( c \) to 0 in particular, this will also
simplify the code because no addition needs to be done.

The resulting code

```java
; compute a * oldvalue
ldc2_w 69069 ; proposed for Super-Duper as a’s value
iload_1 ; oldvalue is (assumed) stored as local variable 1
imul ; and multiply (implicitly taking mod 2^{32})
```
3.4. EXAMPLE: RANDOM NUMBER GENERATION

; newvalue is now left at top of stack

is short, simple, and elegant.

So which random number generator is “better”? Comparing generators can be very difficult and involve fairly high-powered statistics, and depending upon your application, linear congruential generators, in general, may have some very bad properties. Also, depending upon the use to which one puts the generator, some bits of the answer may be more random than others. The second generator, for instance, will always generate an odd number if oldvalue is odd, and an even number if not. Using only the high-order bits will give much better results than using the low-order ones. The easiest way to compare the quality of numbers made by these generators would be to program both into a computer, run them for several thousand, million, or billion numbers, and subject them to statistical tests based on the desired use.

From a speed perspective (and, more importantly, from the perspective of a course on computer organization and assembly language), it should be apparent that the second will run faster. Not only does it involve fewer operations but the operations themselves will run using integer mathematics and therefore might be faster than the long operations of the first generator.

3.4.4 Interfacing with Java classes

(This section may be skipped without loss of continuity, and assumes some Java programming knowledge.)

So why assume that the seed (oldvalue) is stored in local variable number 1? This relates directly to how methods are implemented in Java and how the JVM handles object and method interactions between classes. Specifically, whenever an object’s method is invoked, the JVM passes the object itself (as an a reference type variable) in local variable 0, and the various method parameters are passed in local variables 1, 2, 3, and so forth (for as many variables/parameters as are needed).

In order to run properly, the second generator described above would need to be placed in a method with at least two stack elements, and at least two local variables (one for the object, one for the seed value). A sample complete jasmin program that defines a particular class (jrandGenerator.class and defines two methods, one for object creation and one for generating random numbers using the second method above, is attached as figure 3.4.
; defines jrandGenerator as a subclass of Object
; also defines the class file associated with this as jrandGenerator.class
.class public jrandGenerator
.super java/lang/Object

; boilerplate, same as before
.method public <init>()V
    aload_0

    invokespecial java/lang/Object/<init>()V
    return
.end method

; define a Generate() method that takes an int and returns an int
.method public Generate(I)I
    ; we need two stack elements for calculations
    .limit stack 2

    ; we also need two local variables, of which #1 holds the argument
    ; since this is a normal method, #0 will be set by Java itself
    .limit locals 2

    ; compute a * old_value (and store at top of stack)
    ldc 69069 ; proposed for Super-Duper as a's value
    iload_1 ; old_value is (assumed) stored as local variable 1
    imul ; and multiply (implicitly taking mod 2^32)

    ; new_value is stored at top of stack, return as int
    ireturn
.end method

Figure 3.4: Complete RNG in jasmin
3.5. CHAPTER REVIEW

```java
public class jrandExample {
    public static void main(String args[]) {
        int i;
        int old_value = 1;
        jrandGenerator g = new jrandGenerator();

        for (i=0;i<10;i++) {
            old_value = g.Generate(old_value);
            System.out.println("Generated: "+old_value);
        }
        return;
    }
}
```

Figure 3.5: Java program to call `jrandGenerator`

The structure of this program closely mirrors the previously seen program for printing a string. Unlike the previous program, however, there is no “main” method defined (the `jrandGenerate` class is not expected to be a standalone program), it requires multiple local variables (as defined in the `.limit` directive), and the argument and return types of the Generate method have been changed to reflect their use as a random number generator.

When assembled using the `jasmin` program, the result will be a Java class file named `jrandGenerator.class`. Objects of this class can be created and used as any other within a Java programming environment, as seen in figure 3.5. This simple program merely creates a `jrandGenerator` and invokes the Generate method on it ten times in rapid succession, thus generating ten random numbers.

A similar program could be written to generate ten million random numbers, or to call a different generator written to implement the first RNG described above.

3.5 Chapter Review

- The JVM was designed hand-in-hand with the (high-level) programming language Java, and thus supports a similar style of object-oriented programming. Although it’s not necessary to use OOP in `jasmin` programming, it’s often a good idea.
Java programs and jasmin programs must both be converted to .class files before they can be executed by the JVM.

The command to convert jasmin programs to class files (that is used in this book) is usually named jasmin. At the time of writing, it is available without charge from the Web at Jon Meyer’s web site or the companion website http://prenhall.com/juola.

Input and output is typically a hard problem because of the number of different devices out there. Java and the JVM simplify this problem through the use of the class system. By memorizing the right three jasmin statements, a programmer can output data (of any type she likes) to the standard output whenever she desires.

Assembly language instructions can be divided into three major types: instructions (which are converted to bytecode machine instructions), comments (which are ignored by the computer), and directives (which affect the conversion/assembly process itself).

Directives are used to define how class files fit into the standard Java class hierarchy.

Directives, and specifically the .limit directive, are also used to control the amount of resources available to a given method or function. In particular, .limit stack X will set the maximum stack size, and .limit locals Y will set the maximum number of local variables.

3.6 Exercises

1. What is the difference between a compiler and an assembler?

2. List at least five instructions (mnemonics) that take exactly one argument?

3. How can the computer tell if a given line contains a directive, a comment, or an instruction?

4. Will your jasmin program still work if you forget the .limit directive?
3.7 Programming Exercises

1. Write a jasmin program that will display the following poem in a Web page:

   There once was a lady named Nan
   Whose limericks never would scan
   When she was asked why
   She replied, with a sigh,
   “It’s because I always try to put as many syllables into the
   last line as I possibly can

2. Write a jasmin program to display a triangular pattern of capital O’s as follows:

   O
   0
   0 0
   0 0
   0 0
   0 0

3. Write a jasmin program to display today’s date in the following format:
   Today is Monday, 9/19/2005.

4. Write a jasmin program to calculate and print the wages due me in the following circumstances: This month, I worked 80 hours at a job that pays $25.00/hour, 40 hours at a job paying $15.50/hour, and 45 hours at a job paying $35.00/hour.

5. The Boeing 777-300 is an airplane with a maximum passenger capacity of 386 people. Write a program to determine how many planes I would need to charter in order to take N people on a round-the-world trip. You may assemble a specific value of N into your program. (Note: planes are charted one at a time; chartering three-fifths of a plane doesn’t make sense.)

6. Many operating systems keep track of time not in terms of months and days, but in terms of elapsed time since some major event. Write a program to calculate and print the number of days that have elapsed since December 31, 2000. (For example, 1/1/2001 would be 1 day. 1/1/2002 would be 366 days.) Don’t forget leap years!
7. Unlike Christmas (which is always 25 December), the date of Easter varies from year to year. An anonymous correspondent to Nature\textsuperscript{3} published this algorithm to determine the date upon which Easter falls. (It was later proven correct by Samuel Butcher, Bishop of Meath, and is thus called \textit{Butcher’s Algorithm}. All values are integers, and all division is integer division, and mod means the integer modulus (the remainder after division):

- Let $y$ be the relevant year
- Let $a$ be $y \mod 19$
- Let $b$ be $y/100$
- Let $c$ be $y \mod 100$
- Let $d$ be $b/4$
- Let $e$ be $b \mod 4$
- Let $f$ be $(b + 8)/25$
- Let $g$ be $(b - f + 1)/3$
- Let $h$ be $(19 \cdot a + b - d - g + 15) \mod 30$
- Let $i$ be $c/4$
- Let $k$ be $c \mod 4$
- Let $l$ be $(32 + 2 \cdot e + 2 \cdot i - h - k) \mod 7$
- Let $m$ be $(a + 11 \cdot h + 22 \cdot l)/451$
- Let $p$ be $(h + l - 7 \cdot m + 114) \mod 31$

- The Easter month is $(h + l - 7 \cdot m + 114)/31$. ($3 = \text{March}, 4 = \text{April}$).
- The Easter day is $p + 1$.

Implement this algorithm and determine the date of the next ten Easters.

\textsuperscript{3}Nature, 1876 April 20, vol. 13, p. 487.
Chapter 4

Control Structures

4.1 “Everything They’ve Taught You is Wrong”

4.1.1 Fetch-Execute revisited

In the immediately previous chapters, we’ve explored how to write and evaluate fairly complex mathematical expressions using the JVM’s stack-based computations. By pushing arguments onto the calculation stack and performing an appropriate sequence of elementary operations, one can more-or-less get the computer to do one’s bidding. Once. From a practical standpoint, the real advantage of a computer is its ability to perform tasks over and over again without boredom or error.

Reviewing the representation structure of the JVM, it’s not that difficult to see how the computer could be made to execute the same block of code repeatedly. Program code, remember, is stored as a sequence of successive machine instructions, stored as sequential elements of a computer’s memory. In order to execute a particular statement, the computer first “fetches” the current instruction from memory, interprets and executes that instruction, and then updates its notion of the “current instruction.” The idea of the “current instruction” is, formally, a number stored in the Program Counter (PC) referring to a location in the bytecode of the current method. Every time that the fetch-execute cycle occurs, the value stored in the PC goes up by one or more bytes so that it points to the next instruction to be executed.

Why one “or more” bytes? Shouldn’t the PC go up by one each time? Not really, since some operations need more than one byte to define. For example, basic arithmetic operations such as irem only require one byte to define. However, operations such as bipush are underspecified. The bipush operation specifies that a byte should be pushed onto the stack (and
CHAPTER 4. CONTROL STRUCTURES

promoted to a 32-bit integer), but does not, by itself, specify which byte. Whenever this instruction is used, the opcode for `bipush` (0x10) is followed by a single byte-to-be-pushed. The `sipush` instruction (0x11), as one might expect, is followed by not one but two bytes (a short) to be pushed. Similarly, the `iload` instruction is followed by one or two bytes describing the local variable to be loaded. By contrast, the `iload_1` shortcut operation (0x1B) automatically loads local variable #1 and can be expressed in a single byte.

Because operations are variably sized, the fetch-execute cycle needs to be smart enough to fetch (possibly) several bytes, either at the same time or in sequence, in order to fetch the entire instruction and all of its arguments; the PC must be updated to reflect the size of the instruction fetched. Once these difficulties are dealt with, setting the PC to the appropriate location will cause the JVM automatically to execute the sequence sequence of instructions. If, therefore, there were some way to force the PC to contain a particular value, one could cause the computer to execute that block of code over and over again. Implicitly, by controlling the PC, one directly controls what (and how many times) the computer does.

To summarize the next few sections, this kind of direct control is equivalent to the often-vilified `goto` statement. Students are taught and indeed practically indoctrinated into an avoidance of such statements as they can introduce more bugs than a shelf full of ant farms. In higher level languages, students are taught programming methods to avoid them. At the level of assembly language, the gloves are off, and the best one can do is to understand them.

4.1.2 Branch instructions and labels

Any statement that might cause the PC to change its value is usually called a “branch” instruction. Unlike the normal fetch/execute cycle, a branch instruction might go anywhere. In order to define the target location, jasmin, like most other assembly languages, allows for individual instructions to receive labels so that they can be referred as individuals. Not all instructions will get such labels, but any statement can. To adjust the PC, one uses an appropriate statement and then gives the label of the target instruction.

So how is a branch statement created and stored? In more detail, what is a “label”, and how can it be stored as a sequence of bits, like a number or a machine instruction? From the programmer’s viewpoint, a label is just a line by itself holding an optional part of any instruction, a word (made up of letters and numbers, beginning with a letter, conventionally a capital letter, and followed by a colon [:)]) that marks a particular instruction. To
4.1. “EVERYTHING THEY’VE TAUGHT YOU IS WRONG”

transfer control to a given location, use that label (without the colon) as the argument in an appropriate branch statement. For example,

\[ \text{goto Somewhere ; set value in PC to location of Somewhere} \]

4.1.3 “Structured Programming” a red herring

From a machine design standpoint, the simplest way to control the PC is to treat it as a register or local variable, and to assign appropriate values to it. When a particular value is placed into the PC, the machine will “go to” that location and commence execution of code. This, of course, is the infinitely abusable “goto” statement.

\[
\begin{align*}
; & \text{do some computation} \\
; & \text{do some more computation} \\
& \text{goto ALabel ; now transfer control directly to ALabel} \\
; & \text{this statement is skipped} \\
; & \text{as is this one} \\
\end{align*}
\]

\[ \text{ALabel:} \]
\[
; & \text{but we pick up with whatever instruction is here} \\
; & \text{and keep going on in normal fashion} \\
\]

According to modern programming practice (since about 1970 and the very influential work of Dijkstra), programming using goto statements is subject to severe disapproval. One reason for this is that a naked goto can be dangerous. Putting a random location into the PC will cause the computer to begin executing whatever is stored at that location. If that location is computer code, fine. If that location happened to be (for instance) in the middle of the storage dedicated to a string variable, the computer would treat each byte of the string as though it were program code and begin executing. (In the JVM, for example, the ASCII character 65 (‘A’) would correspond to \texttt{1store.2} and cause the computer to try to pop a long int off the stack. If there is no such long at the top of the stack, an immediate program crash will ensue.)

A potentially more serious problem is that programs written with goto statements can be confusing and (for that reason) error-prone. From the viewpoint of the statement which is the target of a goto, there is no obvious
relationship between the visual structure of the program, the ordering of the program statements in bytecode, and the ordering of the program statements in execution. An apparently-simple sequence of statements such as

```
  iload_1
  iload_2
Branch:
  iload_3
  iadd
  imul
```

may not mean what the casual reader thinks, because the code might be executed via a goto to the third `iload_N` statement. Thus, the value stored in local variable #3 is added to (and multiplied with) something non-obvious. In particularly bad cases, the control structure may be completely confusing (the usual metaphor is a plate of spaghetti), with the all the usual problems that correspond with programmer confusion.

For this reason, modern “structured programming” recommends the use of high-order control structures such as block-structured loops and decision statements. However, at the level of the machine code, any change reflecting different computations must inherently be expressed through changes to the PC, thus requiring an implicit “goto”!

Why, then, are programmers supposed to program without using goto statements? The idea behind structured programming is not really to avoid using them altogether, but to restrict their use to contexts where they aren’t confusing. In particular, as much of the program as possible should be modular (composed of logically divided code fragments that can be treated conceptually as single operations). These modules, as much as possible, should have a single defined starting point and a single defined exit point, ideally at the top and bottom of the physical code. (This is often formalizes as the “single-entry/single-exit” principle, as part of the definition of structured programming.) High level languages will often preven you from violating the single-entry/exit rule by their design. The same principles can, and should, be applied to assembly language programming in general. Even though the language itself will allow you the flexibility to do very silly and confusing things, a good and disciplined programmer will resist the temptation. A programmer familiar with the higher-level control structures such as if-statements and loops will try to use similar easy-to-understand structures, even in jasmin, in order that she and others who work with her will be able to figure out exactly what was intended.
4.1. “EVERYTHING THEY’VE TAUGHT YOU IS WRONG”

```
for (i=100; i > 0; i--) {
    // do something clever 100 times
}
```

Figure 4.1: Sample loop in Java or C++

```
for i := 100 downto 1 begin
    { do something clever 100 times }
end
```

Figure 4.2: Equivalent sample loop in Pascal

4.1.4 High-level control structures and their equivalents

A simple example may help to illustrate this. Figures 4.1 and 4.1 shows examples of similar loops in Java/C++ (and many other languages) and Pascal, respectively. In both cases, the computer counts from 100 down to 0, no doubt doing something clever each time through the loop. The block of clever stuff is most efficiently written as a continuous group of machine instructions. When the PC is loaded with the address of the first instruction in the group, the entire block will be executed.

In order to execute this block several times, the computer needs to decide at the beginning of each block whether or not the block needs to be executed (at least) one more time. In the case of the loop in the figures, this decision is easy: if the counter is greater than zero, the block should be executed again. In this case, go to the beginning of the block (and decrement the counter). Alternatively, if the counter is less than or equal to zero, do not execute the loop again and go to the remainder of the program.

Informally, this can be expressed as a simple pop-and-if-\(>0\)-goto. Formally, the mnemonic for this particular operation is \textit{ifgt}. A formal translation of the loops into jasmin would be as in figure 4.3.

Actually, this isn’t a perfectly accurate translation. As long as the initial value for the loop index is greater than zero, it will work. If, however, the programmer had specified a loop starting with a negative number (e.g. \texttt{for (i=-1; i>0;i++)}), the loop in Java or C++ would never have been executed. The jasmin version would still have been executed once, because the clever computations would have been performed before the computer had a chance to check whether or not the index were large enough. A more accurate translation would require smarter — or at least, more varied —
CHAPTER 4. CONTROL STRUCTURES

```
  ldc 100     ; load integer 100 (number of times)
  istore_1    ; store the index in #1 as an int

LoopTop:
  ; do something particularly clever using
  ; any needed local variables
  iload_1     ; re-load loop index from #1
  iconst_m1    ; load -1 for subtraction
  iadd         ; decrement loop index
  istore_1     ; store...
  iload_1      ; ... and re-load loop index
  ifgt LoopTop ; if top of stack > 0, put LoopTop into PC
                 ; and repeat
  ; otherwise, fall through
```

Figure 4.3: (mostly) Equivalent sample loop in jasmin
decision and goto statements.

4.2 Types of gotos

4.2.1 Unconditional branches

The simplest form of a goto is an unconditional goto, which transfers control
to the label designated as an argument. By itself, it can produce an infinite
loop (a loop that runs forever), but not a loop that runs for a while and
then terminates when something changes. For that, the programmer needs
conditional branches.

4.2.2 Conditional Branches

The JVM supports six basic conditional branches (sometimes called condi-
tional gotos), plus several shortcut operations and two more branches (to
be described later, in conjunction with the class/object system). A basic
conditional branch operates by popping an integer off the stack and com-
paring whether the popped integer is greater, less than, or equal to zero. If
the desired condition is met, control is transferred to the designated label;
otherwise, the PC is incremented as usual and control passes to the next
statement. With three possible comparison results (greater, less than, or
equal to zero), all seven meaningful combinations can be designated. These
4.2. TYPES OF GOTOS

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Interpretation</th>
<th>top &gt; 0</th>
<th>top = 0</th>
<th>top &lt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifeq</td>
<td>Goto if equal</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ifne</td>
<td>Goto if not equal</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iflt</td>
<td>Goto if less than</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ifge</td>
<td>Goto if greater than or equal</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ifgt</td>
<td>Goto if greater than</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ifle</td>
<td>Goto if less than or equal</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>(goto)</td>
<td>(Goto always)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Conditional and unconditional branch operations in Jasmin

are summarized in Table 4.1. Note that the goto statement does not change the stack, while the if?? operations all pop a single integer.

4.2.3 Comparison operations

So, if the basic conditional branches only operate on integers, how does one compare other types? Comparing other types is performed explicitly by comparison operations that are defined to return integers. The operation lcmp, for example, pops two longs — as always, two longs will be stored as four stack elements — and pushes the integer value 1, 0, or -1, depending upon whether the first element pushed is greater, less, or equal to the second (the one at the top of the stack). For example, the following code fragment

```java
lload_3 ; load local variable #3 (and #4)
lconst_1 ; push 1 (as long) for comparison
lcmp ; compare magnitudes, push integer answer
ifgt Somewhere ; go Somewhere if #3 > 1
; if we got here, #3 <= 1
```

compares the long value stored as local variable #3 against the number 1, and transfers control to Somewhere if, and only if, the stored value is larger.

There is a very important point to pay attention to about stack ordering. The instruction sequence lload 1, lload 3, lcmp will first put local variables #1/#2, then #3/#4, and only then do the comparison. The comparison is order sensitive, and would give a different result if #1/#2 were on top of the stack instead. To remember how order works, think of subtracting the top element of the stack from the second element. The result pushed is the sign (+1, 0, or -1) of the difference. See figure 4.4 for an example.
Comparing floats and doubles is similar, but a bit trickier because of the sophistication of the IEEE representations. Specifically, IEEE 754 allows for certain special bit patterns to represent “not a number,” abbreviated as NaN. This special pattern, in essence, means that a calculation somewhere previously went badly wrong (like trying to take the square root of a negative number, or divide $0/0$). It doesn’t usually make sense to do comparisons against NaN, but the programmer sometimes has some special interpretation in mind.

For example, suppose a college defines the honors list as all students with grade point averages above 3.50. A brief program fragment (in Pascal) to determine whether or not a student is on the honors list would look like the following:

```pascal
if (gpa > 3.50) then
    onhonorslist := true;
```
4.2. TYPES OF GOTOS

A student with no gpa (a first-semester student or a student who has taken an incomplete in every class for medical reasons, for example) would probably not be considered to be on the honors list. In other words, if a student has a gpa of NaN, then it should be treated as being less than 3.50. This same student, however, should probably not be expelled for having too low a gpa; NaN should be higher than the appropriate cutoff. The JVM and jasmin provide two separate comparison instructions to define these cases. The instruction fcmpg compares the two floating point numbers at the top of the stack and pushes 1, 0, -1 as expected, except that if either value is NaN, the result is 1. If either value is NaN, fcmpl returns -1. As expected, comparing doubles uses similarly named dcmpl and dcmpg with similar differences in behavior.

The jasmin equivalent of the Pascal fragment above, then, would look something like this:

```java
fload_1 ; load gpa from #1 as float
ldc 3.50 ; load gpa cutoff (3.50)
fcmpl ; compare gpa to cutoff, NaN being "below"
ifle Skip ; go to Skip gpa >= cutoff
iconst_1 ; push 1 (boolean : true)
istore_2 ; put "true" into #2 (as int/boolean)
Skip: ; do whatever else you needed to do with this student
```

4.2.4 Combination operations

As before, the second-class types such as short and boolean are not directly supported and must be treated as int types in computation. More surprisingly: jasmin does not provide a way to compute the result of comparing two integers! Instead, integer comparison is done via shortcut operations that combine a comparison operation with built-in conditional branches (as in table 4.2)
All of these operations work in mostly the same way. The CPU first pops two (int) elements stack and performs a comparison like the `icmp` instruction would do if it existed. Instead of pushing the result of the comparison, though, the program immediately jumps (or not) to the appropriate location by modifying the program counter.

### 4.3 Building Control Structures

The main advantage of control structures in high-level languages is that they are relatively easy to understand. It’s helpful in assembly language programming (on the JVM or any other computer) to keep this same easy of understanding to the greatest extent possible. One way to do this to maintain a similar structure of logical blocks of code that are structured like high-order control structures.

#### 4.3.1 If statements

Expanding on the example from section 4.2.3 of how high-level control can be expressed in jasmin, the key to error-free and understandable code is to retain a similar block structure. A traditional if/then statement has up to three parts: a boolean expression, a set of statements to be executed if the expression evaluates as “true,” and another set to be executed if the expression evaluates as “false”. The C++ or Java block

```java
if (a > 5) {
    // (if block)
    // do something
    // for several lines
} else {
    // (else block)
    // do something else
    // for several lines
}
// do whatever other operations need doing
```

can be written equivalently in jasmin as (assuming a is a long, in #1)

```jasmin
lload_1 ; if a is in #1
ldc_2 5 ; load 5
lcmp ; compare a to 5
```
4.3. BUILDING CONTROL STRUCTURES

ifle Else
; if we got here, a > 5, hence perform the if-clause
; at the end of the if clause, skip over the else-clause
; via an unconditional goto
; ---- this corresponds to the if block above
goto Quit
Else:
; if we got here, then a <= 5, hence perform the else-clause
; ---- this corresponds to the else block above
Quit:
; do whatever additional operations need doing
; ---- this corresponds to the statement following the if statement

The block-structured similarity between this code and the high-level code above should be apparent. In particular, there is a set of consecutive instructions in both code samples corresponding to the “if block” and to the “else block” that are always performed in sequence, starting at the top and continuing to the end. The jasmin code weaves in between these blocks to set the program counter accordingly. In detail, note that the test in this example is slightly different; instead of branching directly to the if clause, this example skips over the if clause (to the else clause) if the reverse of the condition is true.

Complex boolean conditionals can be handled through appropriate use of the iand, ior, etc. instructions or through repeated comparisons. For example, we can check that a is in the range between 5 and 10 (a > 5 && a < 10) as follows

lload_1 ; if a is in #1
ldc_2 5 ; load 5
lcmp ; compare a to 5
ifle Else
; if we got here, a > 5
lload_1 ; if a is in #1
ldc_2 10 ; load 10
lcmp ; compare a to 10
ifge Else
; if we got here, a > 5 and a > 10
; and program continues as before
4.3.2 Loops

In many programming languages, there are two basic kinds of loops: those in which the program tests the condition at the beginning, and those in which the program tests at the end. The second kind has been illustrated earlier. We retain the block structure of the interior of the loop, place a label at the top of the block, and then jump back to the top if the conditions are not right for loop exit. In broad terms, this looks something like

```
LoopTop: ; do the stuff associated with the loop
    ; whatever operations are needed
    iload_1 ; load first operand of comparison
    iconst_m1 ; load second operand of comparison
    ifne LoopTop ; (this loops until #1 = -1)
    ; fall through
```

This performs what in Pascal would be a `repeat` loop and what in C, C++, or Java would be a `do/while` loop. For a more traditional `while` loop, the condition is placed before the loop body, the loop body is followed by an unconditional goto, and if the loop exit is met, control is transferred to the first statement outside the loop itself, as in the following

```
Control:
    iload_1 ; load first operation of comparison
    iconst_m1 ; load second operand of comparison
    ifeq LoopOut ; (this loops until #1 = -1)
    ; do the stuff associated with the loop
    ; whatever operations are needed
    goto Control ; after the loop, jump to re-check condition
LoopOut: ; you can ONLY get here via the conditional branch above
```

This code fragment performs the equivalent a loop akin to `while (i != 1)`. Alternatively, one can keep the `do/while` structure but enter the loop at the bottom via an unconditional branch, as in

```
goto LoopEnd ; jump directly to loop test
LoopTop: ; do the stuff associated with the loop
```
4.3. BUILDING CONTROL STRUCTURES

; whatever operations are needed
LoopEnd:
    iload_1 ; load first operand of comparison
    iconst_m1 ; load second operand of comparison
    ifne LoopTop ; (this loops until #1 = -1)
    ; fall through

This saves the execution of a single goto statement.

The equivalence of while and for loops is well-known. A counter-controlled loop such as

```java
for (i=0;i<10;i++)
    // do something
```

is equivalent to

```java
i = 0;
while (i<10) {
    // do something
    i++;
}
```

and can be easily written using the framework above. The most significant change is the need to recruit a local variable as a counter; and otherwise the while structure is repeated almost exactly.

```java
bipush 0 ; using #1 as i, set it to zero initially
istore_1

goto LoopEnd ; jump directly to loop test
LoopTop:
    ; do the stuff associated with the loop
    ; whatever operations are needed

    ; increment #1
    iload_1 ; load i from #1
    iconst_1 ; load 1 for incrementing
    iadd ; we’ve now done i++
    istore_1 ; ... and saved i in #1 (again)

    ; for efficiency demons, the four lines above could
```
CHAPTER 4. CONTROL STRUCTURES

; be replaced by a single operation: iinc 1 1
LoopEnd:
  iload_1 ; load first operand of comparison (i)
bipush 10 ; load second operand of comparison (10)
ifle LoopTop ; (this loops until #1 >= 10)
; fall through only when #1 >= 10

4.3.3 The details of branch instructions

From a programmer's point of view, using goto statements and labels is fairly simple. You make a label where you want control to go, and the program automatically jumps there. Under the hood, the picture is a bit more complex. Instead of storing labels, the computer actually stores offsets. For example, the goto instruction (opcode 0xA7) is followed in bytecode by a signed, two-byte (short) integer. This integer is used as a change to the program counter, or in other words, after the instruction is executed, the value of the PC is changed to PC + offset. If the offset value is negative, this has the effect of stepping control back to a previously executed statement (as in the repeat loop example above), while if the offset value is positive, the program jumps forward (as in the if/then/else example). There’s nothing in theory to prevent one from using a an offset value of zero, but this would mean that the statement didn’t change the PC and would create an infinite loop. An offset of zero would correspond to the jasmin statement

Self:
  goto Self

which is probably not what the programmer wanted.

So how does the programmer calculate these offsets? Fortunately, she doesn’t! That’s the job, and one of the main advantages, of an assembler like jasmin. The programmer can simply use labels and find that it’s the assembler’s problem to determine what the offsets should be. A potentially more serious problem (again from the programmer’s point of view) with this implementation is that, with only two bytes of (signed) offset, no jump can be longer than approximately 32,000 bytes (in either direction). What happens in a really long program?

This is addressed in two ways in jasmin; first, jasmin provides, in addition to the goto instruction, a goto_w instruction (sometimes called a “wide goto”, and implemented as opcode 0xC8). In most respects similar to a
4.3. BUILDING CONTROL STRUCTURES

normal `goto`, it is followed by a full-sized integer, allowing programmers to jump forward or back up to two billion or so bytes. Since individual JVM methods are not allowed to be more than $2^{16}$ (about 64,000) bytes long, and since if you find yourself writing a longer method than that, you probably should divide it into sections anyway, this new opcode solves the problem. Again, it’s really the assembler’s job to decide which opcode is needed — technically, when it sees a statement like `goto Somewhere`, it can use either the 0xA7 or 0xC8 instruction. If the necessary offset would be too large to fit into a short, then it will automatically translate the programmer’s `goto` into a machine-internal `goto_w`.

A more serious problem is that there is no direct equivalent to `ifne_w` or other wide conditional branches. However, the assembler can (again) devise an equivalence without the programmer’s knowledge or cooperation. A conditional (wide) branch to a distant location can be simulated by a branch around a branch, as follows.

```plaintext
; what I really want is "ifne DistantLocation", but
; that’s too far to get in a single step.
; Unfortunately, ifne_w doesn’t exist.
; The computer will realize that and instead use

ifeq Skip ; notice the test is reversed
  goto_w DistantLocation ; because we are branching AROUND
Skip:
  ; the branch we don’t want to take
  ; do something else
```

Like calculating branch sizes in the first place, a good assembler can always do the right thing in this instance.

The most serious problem with using branch statements is that they do not, in any way, provide the programmer with local block structures. Many programmers rely on the convenience of being able to redeclare variables inside blocks, as in the following:

```plaintext
if (x > 0) {
  int x; // this is a new x, unrelated to the previous
  x = -1;
  ...
}
// here the old x reappears and reacquires its old value
```
No such convenience is available in assembly language; all local variables (and the stack) retain their values before, during, and after a jump. If an important variable is stored in location #2 before the jump, and the next statement is \texttt{fstore 2}, then the important variable will be overwritten. Block structure is an important convenience for how to think about assembly language programs, but does not provide any practical defense such as information hiding against mistakes and misuse.

4.4 Example: Syracuse numbers

4.4.1 Problem definition

As an example of how this can be put together, we’ll explore the Syracuse number (or $3N+1$) conjecture. It dates back to classical mathematics, when an early Greek noticed that a few simple rules of arithmetic would produce surprising and unpredictable behavior. The rules are

- If the number $N$ you have is 1, stop.
- If the number $N$ you have is odd, let $N$ be $3N + 1$ and repeat
- If the number $N$ you have is even, let $N$ be $\frac{N}{2}$ and repeat

It was noticed early on that this procedure always seemed to end (to get to 1) when you started with any positive integer, but no one could actually prove that conjecture. Furthermore, no one could find a general rule for predicting how many steps it would take to get to one. Sometimes it’s very quick:

$$16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

but close numbers can take different times:

$$15 \rightarrow 46 \rightarrow 23 \rightarrow 70 \rightarrow 35 \rightarrow 106 \rightarrow 53 \rightarrow 160 \rightarrow 80 \rightarrow$$

$$40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

and sometimes it takes a very large number of steps. Even today, no one has a solution for how many steps it takes or even whether it will always go to one (although mathematicians with computers have tested that that all numbers less than several billion will converge to one and thus end). Try it for yourself: how many steps do you think it will take to get to 1 starting from 81?
4.4. EXAMPLE: SYRACUSE NUMBERS

(1) count_of_steps <- 0
(2) current_value <- 81
(3) while (current_value != 1)
    (4) if (current_value is odd)
        (5) current_value <- (current_value * 3) + 1
    (6) else
        (7) current_value <- (current_value / 2)
    (8) endif
    (9) count_of_steps <- count_of_steps + 1
(10) endwhile
(11) final answer is count_of_steps

Figure 4.5: Test of Syracuse conjecture in pseudocode

4.4.2 Design

Even better, don’t try it yourself. Let the computer do the work. Figure 4.5 gives pseudocode for an algorithm that will start at 81 and count steps until the final value reaches one. Implementing this algorithm in jasmin will get a quick answer to this conjecture.

The code in figure 4.5 calls for two integer variables, and for simplicity in calculation, we’ll keep them as int types (instead of long). In particular, count_of_steps can be stored as local variable #1 and current_value as #2. The arithmetic calculations in steps 1, 2, 5, 7, and 9 can be performed with techniques from chapter 2. Output, in line 11, can be done in any of several ways; again, we’ll pick one of the simpler and simply print the final results.

The if/else construction used in lines 4–8 can be modelled by the code block in section 4.3.1. Specifically, we can determine whether or not the current value is odd by taking the remainder (irem) when divided by two; if the result is equal to zero (if icmpeq) then the number is even. The code in figure 4.6 illustrates this. As is typical of structured programming, this block as a whole has a single entry at the initial statement of the block, and a single exit at the bottom (at the point labeled Exit:).

This entire block of code will in turn be used inside a while-loop structure as in figure 4.7
; entry point for if/else
iload_2   ; load current_value
iconst_2   ; and the number 2, to determine if odd or even
irem       ; calculate remainder
iconst_0   ; compare remainder against zero
if_icmpgt CaseOdd ; goto CaseOdd if it’s odd

CaseEven:   ; technically speaking, we don’t need this label.  
; as we will never branch to this location

; divide #2 by 2 and re-save
iload_2   ; load current value
iconst_2   ; push 2 for division
idiv        ; do the division
istore_2   ; and store the new value

goto Exit       ; and skip the Caseodd block

CaseOdd:      ; multiply #2 by 3 and add one
iload_2     ; load current value
iconst_3     ; push 3 for multiplication
imul         ; multiply (value stored is now 3*N)
iconst_1     ; push 1 for addition
iadd         ; add (value stored is 3*N+1)
istore_2     ; and store the new value

Exit:         ; exit point for both branches of if/else

Figure 4.6: If/else structure for internal block of Syracuse number code
4.4. EXAMPLE: SYRACUSE NUMBERS

; entry point for while
LoopEntry:
  iload_2   ; load current_value from #2
  iconst_1  ; compare current_value against 1
  if_icmpeq LoopExit  ; branch to LoopExit if equal

; do necessary statements for odd/even calculations

; do necessary statements for incrementing count_of_steps

goto LoopEntry  ; branch unconditionally to top of loop
                 ; and re-check

LoopExit:
  ; exit point for while loop

Figure 4.7: While structure for main loop of Syracuse number code

4.4.3 Solution and Implementation

The complete solution is presented here. (MARK ME NOTE TO EDITOR: WE NEED to do something to make the following code obviously structured differently and as a unit.)

; program to solve Syracuse number conjecture (for fixed $N = 81$)
.class public syracuse
 .super java/lang/Object

; boilerplate
 .method public <init>()V
   aload_0
   invokespecial java/lang/Object/<init>()V
   return
 .end method

.method public static main([Ljava/lang/String;)V
   .limit stack 2   ; no complex calculations here
   .limit locals 3   ; #0 is reserved (as usual)
                   ; #1 is a loop counter
                   ; #2 is the value of $N$
i Constantin 0 ; #1 <- 0
istore_1

bipush 81 ; #2 <- 81
istore_2

LoopEntry:
  iload_2 ; load current_value from #2
  iconst_1 ; compare current_value against 1
  if _ icmpeq LoopExit
    ; branch to LoopExit if equal
    ; do necessary statements for odd/even calculations
  
  ; entry point for if/else
  iload_2 ; load current_value
  iconst_2 ; and the number 2, to determine if odd or even
  irem ; calculate remainder
  iconst_0 ; compare remainder against zero
  if _ icmpgt CaseOdd ; goto CaseOdd if it’s odd
    ; we only get here if #2 is/was even
    ; divide #2 by 2 and re-save
    iload_2 ; load current_value
    iconst_2 ; push 2 for division
    idiv ; do the division
    istore_2 ; store the new value
    goto Exit ; and skip the CaseOdd block

CaseOdd:
  ; multiply #2 by 3 and add one
  iload_2 ; load current_value
  iconst_3 ; push 3 for multiplication
  imul ; multiply (value stored is now 3*N)
  iconst_1 ; push 1 for addition
  iadd ; add (value stored is 3*N+1)
  istore_2 ; and store the new value

Exit: ; do necessary statements for incrementing count_of_steps
  iinc 1 1 ; increment loop index
  goto LoopEntry
    ; branch unconditionally to top & re-check

LoopExit:
4.5. **TABLE JUMPS**

```java
switch (monthno) {
    // 30 days hath September, April, June, and November
    case 9 :
    case 4 :
    case 6 :
    case 11 :
        days = 30; break;
    // ... all the rest have 31
    case 1 : case 3 : case 5 : case 7 : case 8 : case 10 : case 12:
        days = 31; break;
    // except February, alone (ignoring leap years)
    case 2 :
        days = 28; break;
    default : System.out.println("Error : Bad month!");
} // end switch

Figure 4.8: Multiway branch statement in Java (and C++)

; print result to System.out (as usual)
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 1 ; load loop counter for printing
invokevirtual java/io/PrintStream/println(I)V
return ; and we’re done
.end method

4.5 Table jumps

Most high-level languages also support the concept of multiway decisions, such as
would be expressed in a Java `switch` statement. As an example of these in use,
consider trying to figure out how many days there are in a month, as in figure 4.8.

Perhaps obviously, any multiway branch can be treated as equivalent to a set of
two-way branches (such as `if/else` statements) and written accordingly. The JVM
also provides a shortcut — in fact, two shortcuts — that can make the code simpler
and faster to execute under certain conditions. The main condition is simply that
the case labels (e.g., the numbers 1–12 in the example fragment) must be integers.

As before, there is no direct notion of block structure. What the machine
offers instead is a multiway branch, where the computer will go to any of several
destinations, depending upon the value at the top of the stack. The general format
of a `lookupswitch` instruction consists of a set of value:Label pairs. If the value
matches the top of the stack, then control is transferred to Label.

```
iload_1 ; Assuming that "monthno" is stored in #1```
CHAPTER 4. CONTROL STRUCTURES

lookupswitch ; begin multiway branch
  1 : Days31 ;
  2 : Days28 ;
  3 : Days31 ;
  4 : Days30 ;
  5 : Days31 ;
  6 : Days30 ;
  7 : Days31 ;
  8 : Days31 ;
  9 : Days30 ;
 10 : Days31 ;
 11 : Days30 ;
 12 : Days31 ;
default : ERROR ;

Days28:
  bipush 28 ; Load 28 days
  istore_2 ; and assign to days (#2)
goto ExampleEnd

Days30:
  bipush 30 ; Load 30 days
  istore_2 ; and assign to days (#2)
goto ExampleEnd

Days31:
  bipush 31 ; load 31 days
  istore_2 ; and assign to days (#2)
goto ExampleEnd

Error:
  ; take appropriate error actions like getstatic .../System/out
goto ExampleEnd

ExampleEnd:
  ; do whatever is needful

The dozen or so lines above that begin with lookupswitch and end with default are actually a single very complex machine instruction. Unlike the others that have been discussed, this instruction takes a variable number of arguments, and the task of the jasmin assembler (as well as the JVM bytecode interpreter) is correspondingly tricky. The default branch is mandatory in JVM machine code (unlike in Java). As with other branch statements, the values stored are offsets from the current program counter; unlike most other statements (except goto), the offsets are stored as four-byte quantities, allowing for jumps to “distant” locations within the method.1

1SIDEBAR : MACHINE CODE FOR lookupswitch/tableswitch. Both lookupswitch and tableswitch involve a variable number of arguments, and as such have a complex implementation in bytecode. Essentially, there is a “hidden” implicit ar-
4.5. **TABLE JUMPS**

If, as in the above example, the values are not only integers, but contiguous integers — meaning that they run from a starting value (1, January) to an ending value (12, December) without interruption or skips — then the JVM provides another shortcut operation for multiway decisions. The idea is simply that if the lowest possible value were (for example) 36, the next would have to be 37, then 38, and so forth. By defining the low and high ends of the spectrum, the rest can be filled in as a table. For this reason, the operation is called `tableswitch`, and it’s used as follows:

```
iload_1               ; Assuming that "monthno" is stored in #1
tableswitch 1 12      ; begin multiway branch, from 1 to 12
  Days31 ;
  Days28 ;
  Days31 ;
  Days30 ;
  Days31 ;
  Days30 ;
  Days31 ;
  Days31 ;
  Days30 ;
  Days31 ;
  default : Error ;
Days28:
  bipush 28 ; Load 28 days
  istore_2 ; and assign to days (#2)
  goto ExampleEnd
Days30:
  bipush 30 ; Load 30 days
  istore_2 ; and assign to days (#2)
  goto ExampleEnd
```

In the case of `lookupswitch`, the `jasmin` assembler will count the number of value:label pairs for you. The bytecode created involves not only the `lookupswitch` opcode (0xAB), but a four-byte count of the number of non-default branches. Each branch is stored as a four-byte integer (value) and then a corresponding four-byte offset to be taken if the integer matches the top of the stack.

In the case of `tableswitch`, the values can be computed from the starting and ending values. A `tableswitch` statement is stored internally as the opcode byte (0xAA), the low and high values (stored as four-byte integers), and finally a set of sequential four-byte offsets corresponding to the values low, low + 1, low + 2,... high. See the appendix for more details on both.
CHAPTER 4. CONTROL STRUCTURES

Days31:
    bipush 31 ; load 31 days
    istore_2 ; and assign to days (#2)
    goto ExampleEnd

Error:
    ; take appropriate error actions like getstatic .../System/out
    ; and print an error message
    goto ExampleEnd

ExampleEnd:
    ; do whatever is needful

The only different operation in the tableswitch example is the tableswitch itself; the rest of the code is identical. The important differences are inherently related to the structure of the table. The programmer needs to define the low and high values that the variable of interest can legitimately take, and then to sort the labels into increasing order, but not to pair the labels explicitly with their corresponding values. This will be done automatically by the table structure — in the example above, the fourth label (Days30) is automatically attached to the fourth value. As before, a default case is mandatory.

Of course, whether or not these jump tables will add to program efficiency varies from situation to situation and problem to problem. A switch statement can always be written as an appropriate collection of if statements with appropriately complex conditions; in some cases, it may be easier to calculate a boolean condition than to enumerate the cases.

4.6 Subroutines

4.6.1 Basic instructions

One major limitation of branch-based control is that, after a block of code is executed, it will automatically transfer control back to a single and unchangeable point. Unlike traditional “procedures” in high-level programming languages, it is not possible to set up a block of code that can be run from any point in the program and then return to the place from which it came. In order to do this, more information — and new control structures and operations — are needed.

The main piece of information needed is, of course, the location from which control was passed, so that the program can use the same location as a point of return. This requires two basic modifications — first, that there be a branch instruction that also stores (somewhere) the value of the program counter before the jump, and second, that there be another kind of branch instruction that will return to a variable location. A block of code written using these semantics is usually referred to as a subroutine.

The JVM provides a jsr (Jump to SubRoutine) instruction — all right, technically it provides two instructions, jsr and jsr_w, analogous to goto and goto_w — to fill this need. When the jsr instruction is executed, control is immediately
4.6. SUBROUTINES

<table>
<thead>
<tr>
<th>Location</th>
<th>0x1001</th>
<th>0x1002</th>
<th>0x1003</th>
<th>0x1004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte value</td>
<td>0xA8</td>
<td>0x00</td>
<td>0x10</td>
<td>0x3b</td>
</tr>
<tr>
<td>Interpretation</td>
<td>jsr</td>
<td>Integer : 0x0010 = 16</td>
<td>istore_0</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Bytecode structure of jsr instruction

passed (as with a goto) to the label whose offset is stored in the bytecode. Before this happens, though, the machine calculates the value \((PC + 3)\), the address of the next instruction that immediately follows the jsr instruction itself. This value is pushed onto the stack as a normal 32-bit (4-byte) quantity, and only then is control passed to the label.\(^2\)

All machines that provide calls to subroutines also provide methods for returning from subroutines. It is fairly easy to see how this could be accomplished on a stack-based computer; the return instruction would examine the top of the stack and use the value stored there, which we piously hope was pushed by a jsr instruction, as the location to which to return. This location becomes the target of an implicit branch, control returns to the main program, and computation proceeds merrily on its way.

Things are slightly more complicated on the JVM, primarily for security reasons; the ret instruction does not examine the stack for the location to return, but instead accepts as an argument the number of a local variable. The first task any subroutine must perform, then, is to store the value (implicitly) pushed by the jsr instruction into an appropriate local variable — and once that is done, to leave this location untouched. Trying to perform computations on memory addresses is at best dangerous, usually misleading, and in Java and related languages, outright illegal. Again, this is something that the security model and verifier will usually try to prevent.

\(^2\)SIDEBAR: MACHINE LANGUAGE FOR jsr. To understand exactly how this works, let’s look at the detailed machine code for the jsr instruction. The jsr mnemonic corresponds to a single byte (0xA8). It is then followed by a two-byte offset, stored as a signed short integer. Assume that memory locations 0x1001-0x1004 hold the pattern as shown in table 4.3.

When the jsr instruction is executed, the program counter will have the value 0x1000 (by definition). The next instruction in memory (istore_0) is stored at location 0x1003. When the jsr instruction is executed, the value 0x1003 is pushed (as an address) onto the stack, and the value of the PC is changed to 0x1000 + 0x0010, or 0x1010. This implies that the program will jump sixteen bytes forward and begin executing a new section of code. When the ret instruction is executed, transfer will return to location 0x1003 and the icnst_0 instruction. The jsr_w instruction is similar, except that it involves a four-byte offset (thus possibly a longer jump) and the value PC+5 is pushed.
4.6.2 Examples of subroutines

Why subroutines?

One common use for a subroutine is as something akin to a Java method or C++ procedure: to perform a specific fixed task that may be needed at several points in the program. An obvious example of this would be to print something. As has been seen in earlier examples, something as simple as printing a string can be rather tricky and take several lines. To make the program easier and more efficient, a skilled programmer can make those lines into a single subroutine block, accessed via jsr and ret.

Subroutine max(int A,int B)

Let’s start out with a simple example of a subroutine to do arithmetic calculations. An easy example would calculate (and return) the higher of two integers. Let’s assume that the two numbers are on the stack (as integers), as in the diagram:

```
<table>
<thead>
<tr>
<th>argument 2 (B)</th>
<th>should become</th>
<th>max(A, B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument 1 (A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 4.9: Stack structure for max(A,B)

The if_icmp?? instruction will do the appropriate comparison for us, but pop
(and destroy) the two numbers. So before doing this, we should duplicate the top two words on the stack using \texttt{dup2}. Before doing that, though, any subroutine must handle the return address. For simplicity, we will store that in local variable \#1.

\begin{verbatim}
; assume that this is called via \texttt{jsr Max}, with
; the two integers already on the stack
Max:
    astore_1 ; stores return (a)ddress in \#1
dup2 ; copy two arguments for later use
    if_icmpgt Second ; branch if A < B
First:
    ; A >= B, so need to delete B from stack
    pop
    goto Exit
Second:
    ; A < B, so need to swap B to top and delete it
    swap
    pop
Exit:
    ret 1 ; return to the location stored in \#1
\end{verbatim}

**Subroutine printstring(String s)**

For a second example, we'll write (and use) a subroutine to output a fixed string pushed on the stack. Let's write the subroutine first:

\begin{verbatim}
; assume that this is called via \texttt{jsr Printstring}, with
; the string to be printed already pushed onto the stack
; (below the return address)
PrintString:
    astore_1 ; stores return (a)ddress in \#1

    ; assume that the string to be printed is already on the stack

    ; you should already know the next three lines
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap ; put the arguments in the right order
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

    ret 1 ; return to the location stored in \#1
\end{verbatim}

The new instruction \texttt{astore.1} is just another example of the \texttt{?store_N} instruction, but stores an address (type ‘a’) instead of a float, double, int, or long. The
getstatic and invokevirtual lines should already be familiar. The code assumes that the calling environment pushed the string prior to executing the jump-to-subroutine call — and thus merely pushes the System.out object and invokes the needful method. Once this is done, the ret instruction uses the value newly stored in #1 as the location to return. Notice, however, that there is still not much notion of “information hiding” or block structure, and that, in particular, this subroutine will irretrievably destroy any information that had been stored in local variable #1. If a programmer wants to use this particular code block, she needs to be aware of this behavior. More generally, for any subroutine, it is important to be aware of what local variables are and aren’t used by the subroutine, since they’re in the same set of local variables as used by the main program.

Using subroutines

The main program could be as complex as we like, and involve several calls to this subroutine. How about some poetry?

```java
; push the first string/line to be printed
ldc "'Twas brillig, and the slithy toves"
; call the subroutine beginning at PrintString
jsr PrintString ; return to immediately after this line
; note that no label is needed

; and continue in similar fashion
ldc "Did gyre and gimble in the wabe."
jsr PrintString

ldc "All mimsy were the borogroves"
jsr PrintString

ldc "And the mome raths outgrabe."
jsr PrintString

; never forget to cite your sources
ldc "(from Jabberwocky, by Lewis Carroll)"

return ; quit the method, since we’ve printed the poem
```

A complete version of this program is presented as figure 4.10. (Actually, there is a deliberate error in this figure. One line will not be printed. Can you figure out which line, and why? More importantly, do you know how to correct this error?)
4.6. SUBROUTINES

; program to print first verse of Lewis Carroll’s _Jabberwocky_
.class public jabberwocky
.super java/lang/Object
 ; the usual boilerplate
.method public <init>()V
   aload_0
   invokespecial java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
   .limit stack 2
   .limit locals 2
   ; push the first string/line to be printed
   ldc "'Twas brillig, and the slithy toves"
   ; call the subroutine beginning at PrintString
   jsr PrintString ; return to immedately after this line
   ; note that no label is needed
   ; and continue in similar fashion
   ldc "Did gyre and gimble in the wabe."
   jsr PrintString
   ldc "All mimsy were the borogroves"
   jsr PrintString
   ldc "And the mome raths outgrabe."
   jsr PrintString
   ; never forget to cite your sources
   ldc "(from Jabberwocky, by Lewis Carroll)"
   return ; quit the method, since we’ve printed the poem
   ; assume that this is called via jsr Printstring, with
   ; the string to be printed already pushed onto the stack
   ; (below the return address)

PrintString:
   astore_1 ; stores return (a)ddress in #1
   ; assume that the string to be printed is already on the stack
   ; you should already know the next three lines
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap ; put the arguments in the right order
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
   ret 1 ; return to the location stored in #1
.end method

Figure 4.10: Complete program [with one error] as subroutine example
4.7 Example: Monte Carlo estimation of $\pi$

4.7.1 Problem definition

Today, everyone knows the value of $\pi$ (3.14159 and a bit). How did mathematicians figure out its value? A lot of approaches have been tried over the centuries, including one notable not for its mathematical sophistication, but instead for its simplicity. We present here a modification of this experiment, originally due to Georges de Buffon.

Consider throwing a dart (randomly and uniformly) at the diagram shown in figure 4.11. We assume that the darts can land anywhere within the square, and in particular, some of the darts will land inside the circle and some won’t. Since the square is 2 units on a side, it has total area of 4. The circle, having area $\pi r^2$, has an area of $\pi$. Thus, we expect that the proportion of darts that land inside the circle will be $\frac{\pi}{4}$.

Or, in other words, if we threw 10,000 darts at the diagram, we would expect about 7,854 of them to land inside the circle. We can even restrict our attentions
4.7. EXAMPLE : MONTE CARLO ESTIMATION OF $\pi$

(1) total_hits <- 0
(2) for (total_darts := 1 up to 10000)
(3)   generate (x,y) position for new dart
(4)   if ( (x,y) inside circle )
(5)       total_hits <- total_hits + 1
(6)   endif
(7) endfor
(8) final answer is (totalhits / 10000)
(9) FINAL final answer is (totalhits / 10000) * 4

Figure 4.12: Monte Carlo calculation of $\pi$ in pseudocode

to the upper right hand quadrant, and expect the same result.

This method of exploration is often called Monte Carlo simulation. It can be a
very powerful way of exploring a large probability space when you’re not sure of the
exact parameters of the space. It’s also the sort of task at which computers excel,
since the simple calculations (where did the dart land? Is it inside or outside the
circle?) can be repeated thousands or millions of times until you have an accurate
enough answer.

4.7.2 Design

Section 3.4.1 discussed a bit of the theory and practice of random number genera-
tion. The code developed in that section can provide random integers for us, with
a few changes. The main one will be that every location (in the unit square) is
defined by exactly two numbers, and hence we will need to call the generator from
two separate places. This implies a subroutine.

Other than that, the program will need two counters, one for the number of
darts thrown, and one for the number of darts that land inside the circle. For any
dart, if the position where it lands is ($x$, $y$), then it’s inside the circle if and only if
$x^2 + y^2 \leq 1$.

The pseudocode for solving this problem might look something like figure 4.12.
The structure of the problem itself, repeated generation of random points and then
totalling successes and failures, suggests some sort of loop. The actual decision as
to success or failure (inside or outside the unit circle) will be implemented with an
if/then-equivalent structure. In summary, this program can be approached and de-
signed using the same sort of higher-order constructions that would be appropriate
for other programming languages such as Java or Pascal.

After this block has been executed a large enough number of times, the ratio
of successes to total executions should approximate $\frac{\pi}{4}$.

---

*Use the distance formula if you’re not sure why that works.*
CHAPTER 4. CONTROL STRUCTURES

For variables, we will need at least two integers to serve as counters, another location to hold the current value of the seed of the random number, two more to hold the \((x,y)\) coordinates of the current dart, and a sixth to hold the return location from the random number generation subroutine. The control structures needed have all been developed in earlier sections.

In particular, if \#4 and \#5 hold (as floats) the \(x\) and \(y\) coordinates, respectively, then the following block of code will increment the counter in \#1 only if the dart is inside the circle:

\[
\begin{align*}
&\text{fload}_4; \quad \text{load } x \text{ coordinate from } \#4 \\
&\text{dup}; \quad \text{square it} \\
&\text{fmul} \\
&\text{fload}_5; \quad \text{load } y \text{ coordinate from } \#5 \\
&\text{dup}; \quad \text{square it} \\
&\text{fmul} \\
&\text{fadd}; \quad \text{calculate } x^2 + y^2 \\
&\text{fconst}_1; \quad \text{push } 1 \text{ for comparison} \\
&\text{fcmpg}; \quad \text{compare} \\
&\text{ifgt} \text{ Skip}; \quad \text{go to Skip if outside the circle} \\
&\text{iinc} 1 1; \quad \text{increment the counter in } \#1 \\
\end{align*}
\]

\text{Skip:}; \quad \text{do whatever is needed}

We can modify the first random number generator from section 3.4.2 to generate floating point numbers fairly easily, as in the following code fragment.

\[
\begin{align*}
&\text{; compute } a \times \text{oldvalue} \\
&\text{ldc}_w 65537; \quad \text{a is } 2^{16} + 1, \text{ stored as a long} \\
&\text{i0load}_3; \quad \text{oldvalue is stored as local variable } \#3 \\
&\text{i2l}; \quad \text{convert oldvalue to a long} \\
&\text{lmul}; \quad \text{and multiply} \\
&\text{; add c} \\
&\text{ldc}_w 5; \quad \text{c is 5, stored as a long} \\
&\text{ladd}; \quad \text{and add to } a\times\text{oldvalue} \\
&\text{; and take the remainder } \text{mod m} \\
&\text{ldc}_w 4294967291; \quad \text{load value for } m \\
&\text{lrem}; \quad \text{calculate modulus} \\
&\text{i2i}; \quad \text{convert back to integer} \\
&\text{dup}; \quad \text{duplicate for storage} \\
&\text{istore}_3; \quad \text{store new value for next time} \\
&\text{i2f}; \quad \text{convert to floating point number} \\
&\text{ldc 4294967291.0}; \quad \text{load } m \text{ as floating point}
\end{align*}
\]
4.7. EXAMPLE: MONTE CARLO ESTIMATION OF $\pi$

```java
fdive ; divide to get number in [0,1)

; floating point is left on top of the stack
```

This fragment, in turn, will become the core of a subroutine to generate both $x$ and $y$ coordinates.

### 4.7.3 Solution and Implementation

The complete solution is presented here.

```java
; program to calculate $\pi$ via Monte Carlo simulation

.class public pi
.super java/lang/Object

; boilerplate
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V
  return
.end method

.method public static main([Ljava/lang/String;)V
  .limit stack 4
  .limit locals 7
  iconst_0 ; number of darts thrown so far is 0
  istore_1 ; number of darts is in #1
  iconst_0 ; number of darts inside so far is 0
  istore_2 ; number of darts inside is in #2
  icall 1 ; seed the random number generator with 1
  istore_3 ; RNG seed in #3

Head:
  iload_1 ; load number of darts thrown
  ldc 10000 ; compare to 10,000 darts
  if_icmpgt End ; if more than 10,000, exit the loop
  jsr Random ; get random float for $x$ coordinate
  fstore 4 ; and store in #4
  jsr Random ; get random float for $y$ coordinate
```

CHAPTER 4. CONTROL STRUCTURES

fstore 5 ; and store in #5
fload 4 ; load x coordinate from #4
dup ; square it
fmul
fload 5 ; load y coordinate from #5
dup ; square it
fmul
fadd ; calculate x^2 + y^2
fconst_1 ; push 1 for comparison
fcmpg ; compare
ifgt Skip ; go to Skip if outside the circle
iinc 2 1 ; increment number of darts inside in #2

Skip:
iinc 1 1 ; increment total darts thrown in #1
goto Head ; and go to top of loop

Random:
astore 6 ; store return value

; compute a * oldvalue
ldc2_w 65537 ; a is 2^16 + 1, stored as a long
iload_3 ; oldvalue is stored as local variable #3
i2l ; convert oldvalue to a long
lmul ; and multiply

; add c
ldc2_w 5 ; c is 5, stored as a long
ladd ; and add to a*oldvalue

; and take the remainder mod m
ldc2_w 2147483647 ; load value for m
lrem ; calculate modulus
l2i ; convert back to integer
dup ; duplicate for storage
istore_3 ; store new value for next time

i2f ; convert to floating point number
ldc 2147483647.0 ; load m as floating point
fdiv ; divide to get number in [0,1)
ret 6 ; return to calling environment stored in #6

End:
iload_2 ; load total darts inside
i2f ; calculate ratio in floating point
4.8. CHAPTER REVIEW

```java
iload_1          ; load total darts
i2f              ; calculate ratio in floating point
fdiv             ; divide for ratio (pi/4)
ldc 4.0          ; multiply by 4
fmul             ; to get final answer

; and print it
getstatic java/lang/System/out Ljava/io/PrintStream;
swap             ; arguments in correct order
invokevirtual java/io/PrintStream/println(F)V
return
.end method
```

4.8 Chapter Review

- The location of the instruction currently being executed is stored in the program counter (PC) inside the CPU. Under normal circumstances, every time an instruction is executed, the PC is incremented to point to the immediately following instruction.

- Certain instructions can alter the contents of the PC and thus cause the computer to change its execution path. These statements are often called branch statements or goto statements.

- Any statement that is the target of a branch must have a label; this is just a word, usually beginning with a capital letter, that marks its location in the code.

- The goto statement executes an unconditional branch; control is immediately transferred to the point marked by its argument.

- The if?? family of conditional branches may or may not transfer control to their target. They pop an integer off the stack, and depending upon the size and sign of that integer, they will either branch or else continue with the normal fetch/execute cycle.

- The ?cmp family of statements are used to compare types other than integers; they pop two members of the appropriate type off the stack and push an integer with value 1, 0, or -1, depending on whether the first argument is greater than, equal to, or less than the second. This integer can then be used by a following if?? instruction.

- The if_icmp?? instructions combine the functions of an icmp statement with the conditional branches of the if?? family into a single instruction.

- Higher-order control structures such if/else statements and while loops can — in fact, must — be implemented at the assembly language using conditional and unconditional branches.
• The `lookupswitch` and `tableswitch` statements provide a way to perform multiway branch statements that may be more efficient ways of implementing case or switch statements than a series of if/else statements.

• Subroutines work by pushing the current value on the program counter onto the stack, and then by returning to the previously saved location at the end of the subroutine. In jasmin, these correspond to the `jsr` and `ret` instructions, respectively. Unlike most other machines, the `ret` instruction expects to find its return location in a local variable, for security reasons. Therefore, the first operation in a subroutine is usually the `astore` operation to store the (pushed) return location from the stack to a local variable.

### 4.9 Exercises

1. Why won’t the JVM let you load an integer into the program counter?

2. How can structured programming concepts be implemented in assembly language?

3. Is there a “branch if greater than 0 or less than 0” instruction available in the JVM instruction set? If not, how would you implement it?

4. Is NaN (not a number) greater or less than 0.0?

5. How do you build an if/else-if/else-if/else control structure in jasmin?

6. What’s the difference between `goto` and `goto_w`? Is there a corresponding `ifne_w`?

7. The code in figure 4.6 uses `irem` to figure out if a number is odd or even. Juola’s law of multiple cat skinning states that there’s always at least one more way to do anything. Can you figure out a way to use `iand` to determine if a number is odd or even? How about using `ior`?

8. How about using shift operations?

9. What is the error in figure 4.10? What is the fix?

10. (For advanced programmers) Do the semantics of `jsr` and `ret` as presented in this chapter support recursion? Explain.

### 4.10 Programming Exercises

1. Write a program to determine the largest power of two less than or equal to $N$.

2. There are (at least) two different approaches to writing the previous problem, one using shift instructions, and one using multiplication/division. Which runs faster on your machine?
3. Write a program to determine the largest power of \( N \) that will fit into an integer local variable. Similarly, determine the largest power of \( N \) that will fit into a long local variable. How can you tell when an overflow occurs?

4. Write a program to implement RSA encryption and decryption (you’ll probably have to look this up on the web) using a key of \( N=13*17 \), and \( e=11 \).

5. (a) Write a program to test how good the random number generator is. In particular, it should produce all outputs with about the same frequency. Generate 100,000 numbers (randomly) from 1 to 10. The least frequent number should be at least 90. How good is your generator?

(b) Find values of \( a, c, \) and \( m \) that produce a good generator.

(c) Find values of \( a, c, \) and \( m \) that produce a bad generator.

(d) (For advanced programmers), run a chi-squared test on the output of your generator to determine how good it is.
Chapter 5

General Architecture Issues: Real Computers

5.1 The limitations of a virtual machine

As a virtual machine, the JVM has been designed to be cleanly and simply implementable on most real computers. In at least some ways, the designers have succeeded brilliantly — the JVM is a very simple and easily understandable architecture, and one of the best machines for teaching computer organization and architectures around. However, part of the way this simplicity is obtained is by ignoring some of the real-world limitations of actual computer chips. For example, every method in the JVM is presumed to run in its own self-contained environment; changing a local variable in one method will not affect any other method. By contrast, on a physical computer, there is normally only one CPU and one bank of main memory, which means that two functions running at the same time inside the CPU might compete for registers, memory storage, and so forth. You can imagine the chaos that might result if a check-writing program started picking up the data from, say, a computer game, and started printing out checks payable to Starfleet Academy for the number of photon torpedoes remaining.

Similarly, issues of machine capacity can be aren’t really issues; the JVM machine stack has for all practical purposes an unlimited depth and an unlimited capacity for local variables. By contrast, the PowerPC (the chip inside a modern Mac) has only 32 registers in which to perform calculations, and a Windows PC has even fewer.

Another major issue that the JVM can safely ignore is speed. To run a JVM program faster, you just run a copy of the JVM on a faster physical chip. To build that faster physical chip, though, takes a difficult (and fiercely competitive) job of engineering. Engineers at Intel and Advanced Micro Devices (or any other chip manufacturing company) are always looking for edges that will let their chips run faster. Of course, with some of the most highly trained engineers in the world
working on this problem, the details of how to do this are beyond the scope of this textbook — but the following sections will explore some ways of optimizing the components of a computer to improve performance.

5.2 Optimizing the CPU

5.2.1 Building a better mousetrap

The most obvious way to get more performance out of the computer is simply to increase the overall performance numbers; for example, increasing the word size of the computer from 16 bits to 32 bits. Adding two 32-bit numbers can be done in a single operation on a 32-bit machine, but will take at least two operations (and possibly more) on a 16-bit one. Similarly, increasing the clock speed from 500 MHz to 1 GHz should result in every operation taking half as long, or a 100% increase in machine performance.

In practical terms, this is rarely as effective as one might think. For one thing, almost all machines today are 32 bits, and a 32-bit register is accurate enough for most purposes. Increasing to a 64-bit register would let the programmer do operations involving numbers in the quadrillions more quickly — but how often do you need a quadrillion of anything? Similarly, making a faster CPU chip might not help if the CPU now can process data faster than the memory and bus can deliver it.

More seriously, though, increasing performance this way is expensive and difficult. The arithmetic hardware of a chip, for example, is limited in how fast it can be driven by the physical and electrical response characteristics of the transistors; trying to run them too fast will simply break them. Even if it were physically possible to make faster transistors (which it often isn’t), the cost might end up being prohibitively expensive. This is particularly the case if one is trying to make a 64-bit machine at the same time, which means that one needs not only to make the really expensive transistors, but also to make twice as many of them. So engineers have been forced to look for performance improvements that can be made within the same general technological framework.

5.2.2 Multiprocessing

One fundamental way to make computers more useful is to allow them to run more than one program at a time. (This way, you can be writing a paper for homework and pause to load a web page and check some information, at the same time that the computer is automatically receiving email your roommate sent you and that someone else is downloading your home page to see your latest pictures.) With only one CPU (and therefore only one instruction register), how does the computer juggle the load?

Aside from the possibility of buying another CPU — which is possible, but expensive and technically demanding — a usual choice is time sharing. Like time-sharing a vacation condominium, time is divided into individual slices (weeks for
the condo, perhaps milli- or microseconds for the CPU), and you get to use the equipment for one slice. After that slice is done, someone else comes in and spends their week in the beach cabaña. In order to make this work, the computer must be prepared to stop the program at any point, copy all the program-relevant information (the state of the stack, local variables, the current program counter, etc.) into main memory somewhere, then load another program’s relevant information from a different area. As long as the time slices are kept separate and the memory areas are kept separate (we’ll see how both are done a bit later), the computer appears to be running several different programs at once.

For security reasons, each separate program needs to be able to run independently of each other, and each separate program needs to be prevent from influencing other programs. On the other hand, the computer needs to have a programmatic way to swap user programs in and out of the CPU at appropriate times. Rather than relying on the good citizenship of each individual user program, the solution is to create a special überprogram called the operating system whose primary job is to act as a program control program and enforcer of the security rules. The operating system (abbreviated OS, as in “MacOS,” “OS X,” and even “MS-DOS”) is granted privileges and powers not permitted to normal user-level programs, including the ability to interrupt a running program (to stop it or shut it down), the ability to write to an area of memory irrespective of the program using it, and so forth. These powers are often formalized as programming models and define the difference between supervisor and user level privileges and capacities.

5.2.3 Instruction set optimization

To make a computer run fast, one way to speed it up is simply to make the individual instructions faster. A particular instruction that occurs very frequently, for example, might be “tuned” in hardware to run faster than the rest of the instruction set would lead you to expect. This kind of optimization has already been seen on the JVM, for example, with the special-purpose iload.0 instruction. This instruction is both shorter (one byte vs. two) and faster than the equivalent iload 0 instruction. (Of course, almost every method can be expected to use local variable #0, but relatively few will need, say, local variable #245.) Depending upon the programs that are expected to be run, there may also be kinds of instructions that are expected to be very common, and the designers can optimize for that.

For example, on the multiprogramming system described above, “save all local variables to main memory” might be a commonly-performed action. A more accessible example of a common and demanding application type is a graphics-heavy computer game. Good graphics performance, in turn, demands a fast way of moving data (bitmaps) from main memory to the graphics display peripheral. Loading data one word at a time into the CPU and then storing it (one word at a time) to the graphics card is probably not as fast as a hypothetical instruction to move a large block of data directly from memory to the graphics card. It shouldn’t surprise you to learn that this kind of Direct Memory Access is supported by many modern computers as a primitive instruction type. Similarly, the ability to perform
arithmetic operations on entire blocks of memory (for example, to turn the entire screen orange in a single operation) is part of the basic instruction set of some of the later Intel chips. This kind of “doing the same operation independently to several different pieces of data” is a fundamental step forward in processing power. By permitting parallel operations to proceed at the same time (this kind of parallel operation is called SIMD parallelism, an acronym for “Same Instruction, Multiple Data”), the effective speed of a program can be greatly increased.

5.2.4 Pipelining

Another way to try to make a CPU work faster is somehow to pack more instructions into a given microsecond. One possibility that suggests itself is to try to do more than one different instruction at a time. In order to do this, the CPU has a much more complex, pipelined, fetch-execute cycle that allows it to process several different instructions at once.

Wait a minute! How is this even possible? The trick is that, although the operations themselves have to be processed in sequence, each operation takes several steps and the steps can be processed in a sort of assembly-line fashion. As a physical example, consider the line of people involved in a bucket brigade for carrying water. Rather than carrying water the forty feet from the well to the fire (a task that might take a minute), I instead accept a bucket from my neighbor and hand it off, moving the bucket perhaps four feet. Although the bucket is still thirty-six feet from the fire, my hands are now free to accept another bucket. It still takes each bucket a minute to get from the well to the fire, but ten buckets can be moving at once, so ten times as much water per unit time gets to the fire. A car assembly line is another good example; instead of putting cars together one at a time, everyone has a single well-defined job and thousands of cars are put together via tiny steps. More prosaically, if I have a lot of laundry to do, I can put one load in the washer, then when it’s done, I move that load to the dryer, load another into the washer, and run both machines at once.

This kind of task breakdown occurs within the structure of a modern, high-end CPU. For example, while part of the CPU (the dryer) is actually executing one instruction, a different part (the washer) of the CPU could already be fetching a different instruction. By the time the instruction finishes executing, the next instruction is already here and available to be executed. This trick is sometimes called instruction pre-fetch; an instruction is fetched before the CPU actually needs it, so it’s available at once. Essentially, the CPU is “working” on two instructions at once, and as a result, can get twice as many instructions performed in a given time, as shown in the diagram. This doesn’t improve the latency — each operation still takes the same amount of time from start to finish — but can substantially improve the throughput, the number of instructions that can be handled per second by the CPU as a whole.

The number of stages of a typical pipeline can vary from computer to computer — in general, newer, faster computers will have more stages in their pipeline. As an example, a typical mid-range PowerPC (model 603e, for example) uses a
Figure 5.1: Unpipelined laundry: two loads in six hours
Figure 5.2: Pipelining example in the laundry room: four loads in six hours
Figure 5.3: PowerPC-like pipeline

The first stage is the **fetch** stage, where an instruction is loaded from program main memory, and the next instruction to be performed is determined. Once an instruction has been fetched, the **dispatch** stage analyzes the instruction to determine what kind of instruction it is, gets the source arguments from the appropriate locations, and prepares the instruction for actual execution by the third, **execute** stage of the pipeline. Finally, the **complete/writeback** phase transfers the results of the computation to the appropriate registers and updates the overall machine state as necessary.

In order for this process to work as efficiently as possible, the pipeline must be full at all times, and data must continue to flow smoothly. First, a pipeline can only run as fast as its slowest stage. Simple things, like fetching a particular instruction, can run as fast as the machine can access memory, but the execution of instructions, especially long and involved instructions, can take much more time. When one of these instructions needs to be executed, it can cause a blockage (sometimes called a “bubble”) in the pipeline as other instructions pile up behind it like cars behind a slow-moving commuter. Ideally, each pipeline-stage should consistently take the same amount of time, and designers will do their best to make sure this happens.
The other easy way to break a pipeline is by loading the wrong data, fetching from the wrong location in memory. The worst offenders in this regard are conditional branches, such as “jump if less than.” Once this instruction has been encountered, the next instruction will come either from the next instruction in sequence, or else from the instruction at the target of the jump — and we may not know which. Often, in fact, we have no way of telling which, because the condition depends on the results of a computation somewhere ahead of us in the pipeline and therefore unavailable. Unconditional branches are not that bad if the computer has a way of identifying them quickly enough (which usually means in the first stage of the pipeline). Returns from subroutines create their own problems, because the target of the return is stored in a register somewhere, and again may not be available. In the worst case, the computer may have no choice but to stall the pipeline until it is empty (which can cause a serious performance hit, since branch instructions are very common). For this reason, a lot of research has gone into the idea of being able to predict the target of a branch “well enough” to continue and keep the pipeline full. Branch prediction is the art of guessing whether or not the computer will take a given branch (and to where). The computer will continue to execute instructions based upon this guess, producing results that may or may not be valid. These results are usually stored in special locations within the pipeline, and then later copied to registers if the guess is confirmed correct. If the guess is wrong, these locations (and the pipeline) are flushed and the computer restarts with an empty pipeline. If you think about, even the worst case scenario is no worse than having to stall the pipeline — and if the computer guesses right, then some time has been saved.

Even a stupid algorithm should be able to guess right about 50% of the time, since a branch is either taken or it isn’t. However, it’s often possible to guess much more accurately than that by inspection of the program as a whole. For example, the machine code corresponding to a for loop usually involves a block of code and a (backwards) branch at the end to the start of the loop. Since most such loops are executed more than once (often hundreds or thousands of times), the branch will be taken many, many times and not taken once. A guess of “take the branch” in this case could be accurate 99.9% of the time without much effort. A more sophisticated analysis would look at the individual history of each branch instruction. If this branch instruction has been executed twice and not taken once in either case, then it might be a good bet that it won’t be taken this time, either. By adapting the amount and kind of information available, engineers have gotten very good (well above 90%) at their guessing, enough to make pipelining a crucial aspect of modern design.

5.2.5 Superscalar Architecture

The other multiple different instruction technique involves duplication of pipeline stages or even entire pipelines. One of the inherent difficulties behind pipelining is keeping the stages balanced; if it takes substantially longer to execute a particular instruction than it did to fetch it, the stages behind it may back up. The underlying
5.3. **OPTIMIZING MEMORY**

The idea behind superscalar processing is to perform multiple different instructions at once, in the same clock cycle. To fully understand this, we have to generalize the fetch-execute cycle somewhat, and pretend that instead of just loading one instruction at a time, we instead have an a queue of instructions waiting to be processed. (For obvious reasons, this is sometimes called an *instruction queue* — and there’s no pretense involved.) A typical CPU will have separate modules duplicating possibly time-consuming operations. A good analogy is to think about adding a lane to a busy highway, allowing more traffic to flow. Alternatively, think of the way a typical bank operates, with several tellers each taking the next customer. If one customer presents a real problem, taking up more time than expected, then other tellers can take up the slack. Unlike the SIMD parallelism described earlier, this is an example of **MIMD** (*Multiple Instruction Multiple Data*) parallelism — while one pipeline is performing one instruction (perhaps a floating point multiplication) on a piece of data, another pipeline can be doing an entirely different operation (perhaps loading a register) on entirely different data.

### 5.3 Optimizing memory

To make sure that the computer runs as fast and smoothly as possible requires two things. First, the data the computer needs should be available as quickly as possible, so that the CPU doesn’t need to waste time waiting for it. Second, the memory should be protected from accidental re-writing so that (e.g.) the user mail agent doesn’t mis-read data from a web browser and mail the contents of the page.

---

*SIDEBAR: THE CONNECTION MACHINE. MARK ME STOCK PHOTO OF CM?*

If you want to see a really scary version of parallel operations, check out the architecture of the Connection Machine, built by Thinking Machines Corporation in the late 1980’s. The CM-1 (and the later, faster, CM-2) model incorporates up to 65,536 different “cells”, each an 1-bit individual processor. They are all connected to a central unit called the “microcontroller” which issues the same “nanoinstructions” to each one. The CM-5 model can only handle 16,384 different processors, but they are individually as powerful as a Sun workstation. These processors run individually, but are connected to a very fast and flexible internetwork to allow high-speed parallel computation.

The original CM-1 involved a custom cell architecture, manufactured in groups of 16 cells to a chip. These, in turn, were connected to each other in the form of a 12-way *hypercube*, to create a very dense network, fast enough to keep all the cells informed about each other. Conceptually, the Connection Machines were an attempt to explore the possibilities of massive parallelism as exemplified by the human brain, and to transcend some of the traditional limits of the Von Neumann architecture. A typical neuron isn’t capable of very powerful computations, but the \(10^{12}\) neurons that a normal human has can do amazing things. In practical terms, the CM-1 can be seen as an example of 64K-way SIMD parallelism. Unfortunately, the cost of the special-purpose chips was prohibitive, so the CM-5 switched to a smaller number of commercial SPARC chips, and thereby abandoned SIMD processing (like the human brain) in favor of MIMD. The spiritual descendents of the CM-5 are very much active today, for example, in the kind of parallel processing done by a Beowulf cluster.
you’re looking at to someone else.

5.3.1 Cache memory

On a computer with a 32-bit word size, each register can hold $2^{32}$ (about 4 billion) patterns. In theory, this allows up to about four gigabytes of memory to be used by the processor. In practice, the amount installed on any given machine is usually much less, and the amount actually used by any given program is usually smaller yet. Most importantly, the program generally is only using a small fraction, even of the total program size, at any given instant (for example, the code in a Web browser to download a page only gets used when you actually click a button).

Memory comes in many different speeds, which is to say, the amount of time that it takes to retrieve a bit from memory varies from chip to chip. Because speed is valuable, the fastest memory chips also cost the most. Because most programs use a relatively small amount of memory at a time, most real computers use a multi-level memory structure. Although the CPU chip itself may be running at 2 or 3 gigahertz (executing one instruction every three to five hundred trillionths of a second), most memory chips are substantially slower, sometimes taking fifty or a hundred billionths of a second (a tenth of a microsecond) to respond. This may still seem fast, but is about four hundred times slower than the CPU chip itself. To reduce this memory access bottleneck, the computer will also have a few chips of very high speed memory but with much smaller capacity (usually a few megabytes at most), called cache memory. The word is pronounced “CASH” memory, from the French verb cacher, meaning “to hide.” The basic idea is that frequently and recently used memory locations are copied into cache memory so that they are available more quickly (at CPU speeds!) when the CPU needs them. The proper design and use of cache memory can be a tricky task, but the CPU itself takes care of the details for you, so the programmer doesn’t need to worry about it. Most computers support two different kinds (levels) of cache: level one (L1) cache is built into the CPU chip itself and runs at CPU speed, while level two (L2) cache is a special set of high-speed memory chips placed next to the CPU on the motherboard. As you might expect, L1 cache is faster and more expensive still, which means that it is the smallest but can provide the greatest performance boost.

5.3.2 Memory management

With this same 32-bit word size, A computer can write to a set of $2^{32}$ different memory locations. (On the 64-bit computer, of course, there are $2^{64}$ different addresses/locations.) These define the logical memory available to the program. Of course, the amount of physical memory available on any computer depends on the chips attached, which in turn depends at least partly on how much money the computer’s owner is able and willing to spend. Rather than referring to specific physical locations in memory, the program refers to a particular logical address.
which is reinterpreted by the memory manager to a particular physical location, or possibly even on the hard disk.

Normally memory management is considered to be a function of the operating system, but many computers provide hardware support in the interests of speed, portability, and security. These same security concerns, though, make it almost essential that the user-level programs not have access to this hardware. This means that most of the interesting parts of the memory system are invisible to the user, and available only to programs operating in supervisor mode. As far as the user is concerned, “memory” is simply a flat array the size of logical memory, any element of which can be independently accessed. There is little fuss or complexity involved. This is important enough to be worth repeating — user-level programs can just assume that logical addresses are identical to physical addresses, and that any bit patterns of appropriate length represents a memory location somewhere in physical memory, even if the actual physical memory is considerably larger or smaller than the logical address space.

Under the hood, as it were, is a sophisticated way of converting (and of controlling the conversion of) logical memory addresses into appropriate physical addresses. This process uses a set of address substitutions to convert one address space (logical memory) into a second (physical memory). For simplicity of explanation, we’ll focus on a somewhat abstracted memory manager, taken broadly from a 32-bit PowerPC. There are several different methods of performing this sort of conversion, as detailed here.

5.3.3 Direct address translation

The simplest method of determining the physical address, direct address translation, occurs when hardware address translation has been turned off (only the supervisor can do this, for obvious reasons). In this case, the physical address is bit for bit identical with the logical address, and only 4GB of memory can be accessed, and if two processes for whatever reason try to access the same logical address, there’s no easy way to prevent it. This is usually done only in the interests of speed on a special purpose computer expected to only be running one program at once, and otherwise, most operating systems enforce some kind of

5.3.4 Page Address Translation

To prevent two processes from accessing the same physical address (and if address translation is enabled), the memory management system of the CPU actually expands logical address space into a third space, called virtual address space. For example, we could define a set of 24-bit segment registers to extend the value address value. In our case, the top four bits of the logical address will define and select a particular segment register. The value stored in this register defines a particular virtual segment identifier (VSID) of 24 bits (plus a few extra fields). The virtual address is obtained by concatenating the 24-bit VSID with the lower 28 bits of the logical address, as showing in figure 5.4. The effect of this is to create a new
52-bit address, capable of handling much, much more memory, and thereby prevent collisions.

For example, let’s say that the computer wants to access memory location 0x13572468 (a 32-bit address). The top four bits (0x1), mean that the computer should look at segment register #1. Suppose further that this register contains the (24-bit) value 0xAAAAAA. Concatenating this to the original memory location yields the 52-bit VSID 0xAAAAAA3572468. On the other hand, if another program wanted to access the same logical address, but the value in the segment register were instead 0xBBBBBB, the VSID would for this second program would be 0xBBBBBB3572468. Thus, two different programs, accessing the same logical location would nevertheless get two separate VSID. This explains how local variable #1 could be two different memory locations (VSIDs) for two different programs.

Of course, no machine yet built has had $2^{52}$ bytes of memory (that would be about 4,000,000,000,000,000 bytes, 4 million gigabytes, or 4 petabytes — now there’s a word to drop into dinner conversations). What physical memory is present is actually addressed through yet another table. Physical memory is divided into pages of 4196 ($2^{12}$) bytes each. Each 52-bit virtual address can be thought of as a 40-bit page identifier (this is the 24-bit VSID and a 16-bit page identifier extracted from the original logical address), plus a 12-bit offset within a given page. The computer stores a set of “page tables,” in essence a hash table that stores the physical location of each page as a 20-bit number. The 40-bit page identifier is thus converted, via a table lookup, to a 20-bit physical page address. At the end of this process, the final 32-bit physical address is simply the page address plus the offset.

It’s not quite as confusing as it sounds, especially if you look at the diagram. It is, however, a potentially large amount of work. So why does the computer go through such an involved process? There are several advantages. First, not every page in virtual memory need be stored in physical memory — for pages that are not often used, it may be possible to “swap them out” and store them on long-term storage such as a peripheral. (This is the original reason for talking about “virtual memory,” the idea that the computer can access “memory” that isn’t really there, but instead is on the hard drive. This lets the computer run programs that are much larger than would physically fit into memory, at the expense of speed.) Another advantage is that the same logical address can be made to refer to different physical addresses by changing the value in the segment registers. Finally, this allows a certain degree of security to be put in on a page-by-page basis. A given page (in the page table) can be labeled as “supervisor-only,” meaning that only supervisor-level programs can read or write to locations in that page. A page can similarly be labeled as “read-only,” (programs can load data from locations in that page, but not save to locations in that page); “supervisor write-only” (user-level programs can load, but only supervisor-level programs can save), or the most inclusive “read/write,” (where any program can load or save). This will keep user-level programs from, for instance, scribbling over crucial operating system data.
5.3. **OPTIMIZING MEMORY**

![Diagram of idealized PPC virtual memory structure](image)

Figure 5.4: Diagram of idealized PPC virtual memory structure
5.4 Optimizing peripherals

5.4.1 The problem with busy-waiting

To get the best performance out of peripherals, the key insight is that they not be permitted to prevent the CPU from doing other useful stuff. Computers are so fast that they can usually outrun almost any other physical process. As a simple example, a good human typist can type at about 120 words per minute, which translates to about a character every tenth of a second. A 1GHz computer can add 100,000,000 numbers together between two keystrokes. Thus, a computer should be able to do lots and lots of number crunching while still keeping up with the word processing program. But how does the computer respond to (infrequent, by its standards) keystrokes in a timely fashion while still doing its job?

A dumb method of handling this is via polling, checking to see if anything useful has happened at periodic intervals. In high-level pseudo-code, this would look like this:

```
while (no key is pressed)
    wait a little bit
    figure out what the key was and do something
```

Polling is an inefficient use of the CPU, because the CPU has to spend all this time repeatedly checking whether or not something has happened. For this reason, it’s sometimes called busy-waiting, because the computer is being “busy” waiting for the key to be pressed and can’t do anything else useful.

5.4.2 Interrupt handling

A more intelligent way of dealing with expected future events is to set up a procedure to follow when the event occurs, and then to do whatever else needs doing in the meantime. When the event happens, one will then interrupt the current task to deal with the event using the previously established procedure.

This is more or less how the most computers deal with expected but unpredictable events. The CPU establishes several different kinds of interrupt signals that are generated under preestablished circumstances such as the press of a key. When such an event occurs, the normal fetch-execute cycle is changed slightly. Instead of loading and executing the “next” instruction (defined in the program counter), the CPU will consult a table of interrupt vectors that contains a program location for each of the possible interrupts. Control is then transferred (as though through a call to a subroutine) to that location and the special interrupt handler will be executed to do whatever is needful. (On computers with official programming models, this also usually marks the point at which the computer switches from user to supervisor mode.) At the end of the interrupt handler, the computer will return (normally, as from a subroutine) to the main task at hand.
5.5. CHAPTER REVIEW

On most computers, the possible interrupts for a given chip are numbered from zero to a small value (like ten). These numbers also correspond to locations programmed into the interrupt vector — when interrupt number 0 occurs, the CPU will jump to location 0x00 and execute whatever code is stored there. Interrupt number 1 would jump to location 0x01, and so forth. Usually, all that is stored in the actual interrupt location itself is a single \texttt{JMP} instruction to transfer control (still inside the interrupt handler) to a larger block of code that does the real work.

This interrupt handling mechanism can be generalized to handle system-internal events as well. For example, the time-sharing aspect of the CPU can be controlled by setting an internal \texttt{timer} (details of how such timers might work will be presented in chapter 9). When the timer expires, an interrupt will be generated, causing the machine, first, to switch from user to supervisor mode, and second, to branch to an interrupt handler that swaps the programming context for the current program out, and the context for the next program in. The timer can then be reset and computation resumed for the new program.

5.4.3 Communicating with the peripherals: using the bus

As discussed in the first chapter, data must move between the CPU, memory, and peripherals using one or more \texttt{buses}. This is like getting from your house to the store using one or more roads; depending upon the quality of the roads, the quality of the drivers, and the amount of traffic, it can be a faster or slower trip. Whether you’re a computer or a shopper, you would like the trip to be as fast as possible.

There are two key issues involved in the typical use of a bus. The first is that, electrically, a bus is usually just a set of wires, and so connects all the components together at the same time. This means that a bus acts as a small-scale broadcast medium, where every peripheral gets the same message at the same time. The second is that only one device can be using the bus at once; if the keyboard and hard drive both try to send data, neither will success. To use a bus successfully requires discipline from all parties involved. This discipline usually takes the form of a strict protocol, where communication happens in a very stylized, formal procedure. A typical bus protocol might involve the CPU sending a \texttt{START} message, and then an identifier for a particular device. Every device will receive both of these messages, but only the specific device will respond (typically with some sort of \texttt{ACKNOWLEDGE} message). All other devices have been warned by this \texttt{START} message not to attempt to communicate until the CPU finishes, and sends a similar \texttt{STOP} message. Only the CPU and the specific device are allowed to use the bus during this time, which reduces contention and traffic flow problems.

5.5 Chapter Review

- As a virtual machine, the JVM is freed from some practical limitations that affect the design and performance of real, chip-based architectures.
- With the chip market as competitive as it is, engineers have found many
techniques to squeeze better performance out of their chips. These tend to be improvements in both security and speed.

- One way to get better user-level performance is by improving the basic numbers of the chip, but this is usually a difficult and expensive process.

- Another way to improve the performance of the system (from the user’s perspective) is to allow it to run more than one program at a time. Computers can do this via time-sharing, where the program runs in very short spurts and programs are swapped in and out.

- When engineers know what sort of programs will be run on a to-be-designed computer, they can create special-purpose instructions and hardware specifically to support those programs. An example of such a program would be computer games, which put very specific demands on the graphics processing capability of a computer. The Pentium provides specialist instructions to speed up graphics performance as basic, machine-level instructions.

- Performance can also be increased by parallelism, executing more than one instruction at a time. We can distinguish SIMD parallelism from MIMD parallelism in terms of the flexibility of what kind of instruction can be simultaneously executed.

- One significant type of performance enhancement can be obtained through a form of parallelism called pipelining, where the fetch/execute cycle is broken down into several stages, each of which are independently (and simultaneously) executed. For example, by executing one instruction while fetching the next, the computer can get a hundred times more work done in the same time.

- Superscalar architecture, in which entire pipeline stages are replicated several times, provides another way to speed up processing by doing the same thing several times over.

- Memory access times can be improved by using cache memory to speed up access to frequently used items.

- Memory management techniques such as virtual memory and paging can provide computers with access to greater amounts of memory more quickly and securely.

- By preventing the computer from wasting time looking to see if an expected event has happened yet, the use of interrupts can give substantial performance increases when using peripherals.

- A suitable design of a bus protocol can speed up how fast data moves around the computer by reducing competition for traffic slots.
5.6 Exercises

1. What kind of limitations does the JVM stack ignore as a virtual machine?

2. (a) What kind of advantages would a 128-bit CPU have over a 64-bit CPU?
   (b) How significant are these advantages?

3. Would a special-purpose instruction to store/retrieve the entire contents of the stack to memory be helpful to the JVM?

4. What enhancement(s) would you make to the JVM architecture if you were in charge? Why?

5. Give two real-world examples of pipelining in action beyond those mentioned in the text.

6. How would you apply branch prediction to the `lookuptable` instruction?

7. Give two real-world examples of superscalar processing beyond those given in the text.

8. How should a cache determine what items to store and what items to discard.

9. Explain how memory management can allow two programs to use the same memory location at the same time, without conflict.

10. How would `memory-mapped` I/O interact with a virtual memory system?
Chapter 6

The Intel 8088

MARK ME EDITOR: DO YOU HAVE ANY STOCK PHOTOS OF AN (ORIGINAL) IBM-PC?

6.1 Background

In 1981, IBM released the first generation of its ‘Personal Computer,’ later to be known to all and sundry as the IBM-PC. As a relatively low-cost computer produced by a company whose name was a household word (Apple already existed, but was only known to the hobbyist market, while few of the other computer companies you have heard of even existed at the time), it was a runaway success, dominating business purchases of “microcomputers” almost instantly.

Back then, the computer was sold with only 64K of RAM and no hard drive (people loaded programs onto the computer from 5-1/4 inch floppy disks). The chip inside this machine was manufactured by the Intel corporation and designated model number 8088. Today, the legacy of the 8088 still persists in computers all over the world that are still based on the original 8088 design.

Technically, the 8088 was a second-generation chip, based on the earlier 8086 design. The differences between these two were subtle; both were 16-bit computers with a segmented memory architecture. The big difference between the two was that 8086 had a 16-bit data bus, so that the entire contents of a register could be flushed to memory in a single bus operation. The 8088 had only an 8-bit data bus, so it took a little longer to load or store data from the CPU into memory, but was also a little cheaper to produce, which reduced the overall price of the IBM-PC (and hence improved its marketability).

With the success of the 8088-based IBM-PC, Intel and IBM had a ready market for later and improved chips. The 80286 (the 80186 was designed, but never sold well as the base for a personal computer) incorporated security features into an otherwise laughably insecure 8088, as well as running much more quickly. This chip was the basis for the IBM-PC/Advanced Technology, also known as the PC/AT.
CHAPTER 6. THE INTEL 8088

The 80386 increased the number of registers and made them substantially larger (32 bits each), making the chip ever more powerful. Further improvements followed with the 80486, and the Pentium (a renamed 80586), which will be discussed in detail in a later chapter. These later chips plus the original are sometimes called the 80x86 family.

To a certain extent, the 8088 is of historical interest only; even as a low-cost microcontroller (for example, the kind of chip that figures out how brown you want your toast or which floor to stop the elevator on), there are many other competing architectures based on more modern principles and technology. However, later generations of the Intel 80x86 family have all adhered to the principle of backwards compatibility in the interests of preserving their existing customer base. For example, in 1995, when the Pentium was introduced, there were already millions of people running software for their existing 80486 systems. Rather than force people to buy new software as well as new hardware (which might have caused them to go buy software and hardware from someone else, like Apple), the designers made sure that programs written for the 486 would still run on the Pentium. Since this decision was made at every step, this means that programs written in 1981 for the IBM-PC should still run on a modern P4. Of course, they won’t be able to take advantage of modern improvements such as speed (the original PC ran at 4 Megahertz, while a modern system runs at least 1000 times faster), improved graphics resolution, and even modern devices such as mice, USB keychain drives, and so forth. But because of this compatibility issue, an understanding of the 8088 is important to understand the modern Pentium architecture.

6.2 Organization and Architecture

6.2.1 The Central Processing Unit

At the grossest possible level of abstraction, the CPU of the Intel 8088 looks very much like most other processors, including the Java Virtual Machine. Data is stored in a set of general purpose registers, operated upon by the instructions fetched and decoded inside the control unit, in keeping with the laws of arithmetic as defined by the circuitry of the arithmetic-logical unit. There are a few subtle, but significant, differences.

The first is that, as a physical chip, the capacities (for example, the number of registers) are fixed and unchangeable. The 8088 contains eight so-called “general purpose” registers. Unlike the JVM stack, these are not organized in any particular way, and they are given names instead of numbers. These registers are named

AX  BX  CX  DX
SI  DI  BP  SP

Although these are called “general purpose” registers, most of them are tuned with additional hardware to make specific operations run faster. For example, the CPU has special-purpose instructions tuned to use CX as a loop counter, and the AX/DX pair is the only pair optimized for integer multiplication and division. The SI and DI registers support special high-speed memory transfers (SI and DI stand
for source index and destination index, respectively), and the BP register is usually used for stack instructions for local function variables and function parameters.

In addition to these registers, the 8088 has several special-purpose registers that can’t be used for general computation. The most important of these is probably the IP register, the instruction pointer, which holds the location of the next instruction to be executed. (On other machines, this register might be called the program counter or PC, as they’re synonymous.) Four segment registers (CS, SS, DS, and ES) are used to enable access to more memory and to structure memory accesses, and finally, the FLAGS register holds a set of individual bits that describe the result of the current computation, such as whether or not the last operation resulted in a zero, a positive, or a negative number.

All of these registers except for the segment registers are 16 bits wide. This, in turn, implies that the 8088 has a 16-bit word size and that most operations (can) deal in 16-bit quantities. If for some reason a programmer wants to use smaller representations, she can use fractions of the general-purpose registers. For example, the lower half of the AX register can be subdivided and used as two 8-bit registers, called the AH (high) and AL (low) registers. Really, these are all just (sections of) the same register, so changes to one will effect changes in all. If you somehow loaded the value 0x5678 into the AX register, this would as a side effect set the value in AL to 0x78, and the value in AH to 0x56. Similarly, clearing the AL register at this point would set the value in the AX register to 0x5600. This kind of subdivision is valid for all the general-purpose registers; the the BX register can be divided into the BH and BL registers, and so on.

| AX register (16 bits wide) | AH register (8 bits wide) | AL register (8 bits wide) |

Figure 6.1: Register subdivision in the 8088
Almost all values in the 8088 registers are stored in 16-bit (or smaller) signed two's complement notation. Unlike the JVM, the 8088 also supports unsigned integer notation. Most operations (for example, moving data around, comparing whether two bit patterns are the same, or even adding two patterns) don’t really pay attention to whether or not a given bit pattern is supposed to be a signed or unsigned quantity, but for the few where it is important, there are two different operations to handle the two different cases. As an example, the instruction to multiply two unsigned quantities has the mnemonic MUL; to multiply two different signed quantities, one uses the mnemonic IMUL.

The registers defined above handle most of the work that a 8088 computer might need to use, from memory accesses, to control structures and loops, to high-speed integer processing. For high-speed floating point processing, a separate section of the chip is designed to be a logically separate floating point unit. This FPU has its own set of eight registers, each 80 bits wide for high-precision operations, structured like a stack. The data stored in these registers uses a different, and specialized, kind of floating point representation, similar to standards discussed earlier, but with more bits for both mantissa and exponent. There are also a few additional registers, both as part of the FPU (for example, the FPU has its own instruction register) and for certain special purpose operations.

6.2.2 The Fetch-Execute Cycle

The Fetch-Execute cycle on the 8088 is almost exactly like the cycle with which we are familiar on the JVM; the value stored in the IP register is used as a memory location, and value stored in that location (the one pointed to by the IP register) is copied (“fetched”) into the instruction register. Once the instruction value has been fetched, the value in the IP register is incremented (to point to the next instruction) and the fetched instruction is executed by the CPU.

The only tricky aspect of this is the size of the IP register itself. With only 16 bits, the IP register can only access about 65,000 different locations, and hence only (at most) about 65,000 different instructions. Naively, this would appear to place hard (and very stringent) limits on the size of programs; in particular, no program could be larger than 64K. Fortunately, the memory management techniques described in the next section increased this limit somewhat, essentially by recruiting extra bits from other registers to increase the address space.

6.2.3 Memory

On a computer with a 16-bit word size, each register can hold $2^{16}$ (about 65,000) patterns. This means that any register holding a memory address (for example, the IP) can only point to 65,000 different locations, which by modern standards is hardly enough to be worth the effort. (A quick reality check: the text processing software used to typeset this book takes up 251,864 locations, not counting the editing software or printing software. Eeek.) Even in 1981, 64Kbytes of memory was regarded as a very small amount for a computer to have available.
There are several solutions to this problem. Perhaps the easiest solutions would be to make each pattern/location refer not to a single byte of memory, but to a word (or even larger unit). This causes efficiency problems when storing data items smaller than a word (such as characters in a string), but also makes it possible to address more memory.

The designers of the 8088 took a slightly more complex approach. The memory of the 8088 has been divided into segments of 64 Kbytes each. Each such segment contains exactly as many memory bytes as can be addressed by a normal 16-bit register, but these addresses are interpreted relative to a base address defining a particular segment. As you might expect from the name, the segment definitions are stored in the so-called segment registers.

Unfortunately, the math at this point gets a little tricky. The segment registers themselves are 16 bits wide. The actual (sometimes called the absolute) address used is calculated by multiplying the value in the relevant segment register by 16 (equivalent to shifting its value to the left by four binary places, or one hexadecimal place), then adding to the value in the appropriate general purpose register or IP (called the offset). For example, if the value stored in the segment register were 0xC000, this would define a segment starting value of 0xC0000. An offset of 0x0000 would correspond to the (20-bit) location 0xC0000, while an offset of 0x35A7 would correspond to the absolute location 0xC35A7. Each of these locations corresponds to exactly one byte.

There is no particular reason that a segment must start at an even multiple of 64K. Loading a value of 0x259A would define a segment starting at the value of 0x259A0. In fact, any location ending with a hexadecimal value of 0 is a possible starting location for a segment. In the segment defined above, the segment value of 0x259A plus a hypothetical offset 0x8041 would yield an absolute address of (0x259A0+0x8041) or location/byte 0x2D9E1. For simplicity, such address pairs are often written in the form segment:offset, as in 259A:8041. Under this scheme, legitimate addresses run from 0x00000 (0000:0000) to 0xFFFFF (F000:FFFF). It should also be noted that there are usually many segment:offset pairs that refer to a given location. The location F000:FFFF is also the location FFFF:000F, as well as F888:777F.

In common practice, the four segment registers are used in conjunction with different offset registers to define several different types and uses of memory. They correspond (roughly) to

- **CS (code segment)**: The code segment register is used in conjunction with the IP (instruction pointer) to define the location in memory of the executable machine instructions (the program code).
- **DS (data segment)**: The data segment register is used in conjunction with the general purpose registers AX, BX, CX, and DX to control access to global program data.
- **SS (stack segment)**: The stack segment register is used in conjunction with the stack registers (SP and BP) to define stack frames, function parameters and local variables, as discussed below.
• ES (extra segment) The extra segment register is used to hold additional segments, for example if a program is too large to fit into a single code segment.

Even with this segmented memory model, the computer can still only access \(2^{20}\) different locations, a megabyte. Even this much memory wasn’t really available in practice, since (by design) some of this megabyte was reserved as video memory instead of general-purpose program memory. In practical terms, programmers could really only access the first 512K of memory for their programs. Fortunately, by the time computers with more than 1MB of memory became financially practical, the design of the 8088 had been superceded by later generations in the same family, including the Pentium. Memory access on the Pentium is substantially different, in part to avoid this 1MB restriction.

6.2.4 Devices and Peripherals

The modern computer can be attached to a truly bewildering array of peripherals, ranging from simple keyboards and monitors through business accessories such as scanners and fax machines, through truly unusual specialist devices such as vending machines, toasters, and medical imaging devices. Because 8088-based machines are so common, it’s not unusual to see almost any device being designed and built to interface with a 8088.

From the computer’s (and computer designer’s) point of view, however, the task is simplified somewhat by the use of standardized interface designs and ports. For example, many computers come with an IDE controller chip built-in to the motherboard, and attached directly to the system bus. This controller chip, in turn, is attached to a set of cables that will plug into any IDE-style drive. When the controller gets the appropriate signals (on the system bus), it interprets these signals for the drive. Any manufacturer that building hard drives can simply make sure they follow the IDE specifications, and be sure of being able to work with (almost) any PC. The PCI (Peripheral Component Interconnect) performs a similar job of standardization, connecting main memory directly to a variety of devices. Perhaps the most widespread type of connection out there is the USB (Universal Serial Bus) connection. This provides a standardized high-speed connection to any USB-supported device, ranging from a mouse to a printer. The USB controller itself will query any new gadgets attached to figure out their names, types, and what they do (and how they do it). The USB port can also provide power, as well as communicate with them at almost network speeds. From the programmer’s point of view, however, the advantage is that one need only write a program to communicate with the USB controller, simplifying the task of device communication a lot.
6.3 Programming

6.3.1 Operations and Addressing

The 8088 is considered to be a textbook example of CISC (Complex Instruction Set Computing) at work. The set of possible operations that can be performed is large, and the possible operations are themselves correspondingly powerful. One side effect of this CISC design is that some simple operations can be performed in several different ways using different machine operations; this usually means that Intel’s designers decided to provide a short-cut optimization for some specific use of registers.

Because registers are named, instead of numbered or indexed (as in the JVM), the basic operation format is a little different. On the JVM, it is enough to simply say “add”; the two numbers to be added (addends) are automatically on the top of the stack, and the result is automatically left at the top of the stack when the addition is complete. On the 8088, things are differ — the programmer must specify where the addends are stored and where the sum should be kept.

For this reason, most 8088 instructions use a two argument format, as follows:

\[ \text{OP operand1, operand2 ; possible comment} \]

Usually the first operand is called the destination (sometimes abbreviated dst) and the second operand is called the source (abbreviated src). So to add the value in CX to the value in AX (and leave the resulting sum in AX), the corresponding assembly language instruction would be

\[ \text{ADD AX, CX ; really means AX = AX + CX} \]

Because there are eight general-purpose 16-bit registers, there are 64 (8 \cdot 8) different ADD instructions. But in fact, there are more, since one can also add 16-bit values, or 8-bit values, as in

\[ \text{ADD AH, BL ; AH = AH + BL} \]

The 8088 also supports several different addressing modes, or ways of interpreting a given register pattern to refer to different storage location. The simplest examples, given above, are where the value stored in a register are the values used in the computation. This is sometimes called register mode addressing. By contrast, one can also use immediate data, where the data to be used is written into the program itself, as in

\[ \text{ADD AX, 1 ; AX = AX + 1} \]

Note that this only makes sense if the 1 is the second (src) operand; trying to use immediate mode addressing for the destination operand will result in an error. Also note that this is an example of the power of CISC computing, since
this operations would take two separate instructions on the JVM — one to load
the integer constant 1, another to perform the addition.

Either immediate data or register values can also be used as memory addresses,
to tell the computer not what the value is, but where it can be found. For example,
if location BX held the address of a particular 8-bit value, we could add that value
to the current value held in BL with the following statement. The square brackets
([ ]) show that the register BX holds a value to be used in indirect mode (we’ll get
to direct mode presently), as the address of a spot in memory.

\[
\text{ADD } \text{BL, [BX]} \quad ; \text{BL} = \text{BL + whatever value is stored in the location indexed by BX}
\]

This is rather a tricky concept, so let’s explore it a little bit more. Notice that
these two statements do different things:

\[
\begin{align*}
\text{ADD } & \text{BX, 1} \quad ; \text{increment BX} \\
\text{ADD } & \text{[BX], 1} \quad ; \text{increment the value pointed to by BX}
\end{align*}
\]

How do they differ? Let’s assume that the value stored in BX is 4000. Performing
the first operation would increment the value in BX itself, making it equal
to 4001. By contrast, the second operation would then leave the value in BX itself
alone (it stays 4000), but try to increment the value stored at memory location
4000, whatever that value was. (Actually, there would be an error here, for reasons
we will discuss a little later in section 6.3.7. But basically, although we know there
is a value at location 4000, we don’t know whether it’s a byte, word, doubleword,
or what. So how can the computer increment something it doesn’t know the size
of?) Similarly, executing

\[
\text{ADD } \text{AX, [4000h]} \quad ; \text{add the value stored in 4000h to AX}
\]

looks at memory location 0x4000 (the number in brackets is automatically
interpreted as a hexadecimal quantity, because of the trailing \textit{h}) to figure out what
the 16-bit quantity stored there is, and then adds that quantity — not 0x4000 — to
whatever is stored in AX. In this case, where no register is involved, and a defined
constant memory location is used, we refer to this as direct mode.

These addressing modes can be used with the ADD operation in almost any
combination, with two exceptions. It’s legitimate, for example, to add one register
to another, to add a memory location to a register or a register to a memory
location, or to add an immediate value to either a register or memory. However,
it’s not legal to add one memory location directly to another memory location; one
addend or the other must be placed somehow in a register first. It’s also illegal,
and rather silly, to try to use an immediate value as the destination of an addition.
So in the following list of operations, the first five are legal, but the last two are
not.
6.3. PROGRAMMING

ADD  AX, BX ; register-to-register
ADD  AX, [4000] ; memory-to-register
ADD  AX, 4000 ; immediate-to-register
ADD  [4000], AX ; register-to-memory
ADD  [4000], 4000 ; immediate-to-memory
ADD  [3000],[4000] ; illegal! memory-to-memory
ADD  4000, AX ; illegal! ANYTHING-to-immediate

Each of these operations and addressing modes is expressed using slightly different machine code. In machine code, the complexity is increased the existence of special-purpose (and faster) instructions to be used when the destination is specifically the AX register. Armed with this kind of instruction, an intelligent programmer (or compiler) can make the program faster and shorter by putting addition-heavy code into AX. (We’ve already seen this kind of instruction on the JVM with examples like iload_1 as a shortcut for the more general iload instruction. Confusingly, most assemblers don’t even use different mnemonics for the high-speed AX instructions, and instead the assembler program itself will recognize when a special-purpose instruction can be used and automatically generate it. This means that the assembly language programmer doesn’t need to worry about them, if the assembler is smart enough.) This kind of register designed for high-speed arithmetic is sometimes called an accumulator, and the 8088 technically has two different ones — or at least two parts of the same register used as accumulators. The AX register serves as a 16-bit accumulator, while for 8-bit quantities, the AL register is used.

6.3.2 The Arithmetic Instruction Set

This kind of two-operand format is common for most arithmetic operations; for example, instead of adding two numbers, one can subtract them using the SUB instruction instead

```
SUB  BX, AX ; BX = BX - AX
```

The 8088 also recognizes a MOV instruction for moving to and from memory or registers, using the same two-argument conventions where the first is the destination, the second the source. It also recognizes AND, OR, and XOR instructions of the same format — these all also have the special accumulator shortcuts.

There are also a number of one-argument instructions, such as INC (increment), DEC (decrement), NEG (negate — that is, multiply by \(-1\)), and NOT (toggle/reverse all bits in the argument, equivalent to doing an XOR operation with a second argument of all zeros). The format is fairly simple:

```
DEC  AX ; AX = AX - 1
```

Multiplication and division are a touch more complicated. The reason is simple: when you add (for instance), two 16-bit integers, you get more-or-less a 16-bit result, which will still fit into the 16-bit register. When you multiply these numbers,
CHAPTER 6. THE INTEL 8088

### Table 6.1: Information flow in 8088 multiplication

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Multiplicand</th>
<th>High half of product</th>
<th>Low half of product</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>argument</td>
<td>AH</td>
<td>AL</td>
</tr>
<tr>
<td>AX</td>
<td>argument</td>
<td>DX</td>
<td>AX</td>
</tr>
</tbody>
</table>

The result can be up to 32 bits long (which won’t fit any more). Similarly, integer division actually produces two results, the quotient and the remainder (example: \(22/5 = 4\) remainder 2, just like in elementary school). The 8088 chip thus coopts additional registers for the results of multiplication and division — and because of the complexity of the necessary hardware, can only use a few specific register sets for these operations.

To multiply two numbers, the first number must already be present in the accumulator (of whatever bit size is appropriate). The multiplication instruction itself (MUL) takes one argument, which is the second number to multiply. The product is then placed in a pair of registers, guaranteeing that multiplication will never overflow. Table 6.1 shows how and where information flows in multiplication operations.

(The DX and AX register pair is sometimes abbreviated as DX:AX; The AH:AL pair is usually abbreviated as the AX register.)

To multiply the numbers 59 and 71 as 16-bit values, the code fragment below could be used. First, the (decimal) value 59 is loaded into the AX register via a MOV instruction, while the value 71 is similarly loaded into the BX register. Then the value in the accumulator (AX) is multiplied by the value in BX. The result will be left in DX:AX — specifically, AX will hold the lowest 16 bits of the results (the decimal value 4189, stored as 0x105D), while DX holds the highest 16 bits, which in this specific case would be all zeros).

```assembly
MOV AX, 59 ; AX gets multiplicand
MOV BX, 71 ; BX gets multiplier
MUL BX ; DX:AX = AX * BX
```

There are actually two different multiplication instructions, one for unsigned integer multiplication (MUL), and one for signed integers (IMUL). The register use in both cases is the same, but the only difference is whether the highest bit in the multiplicand and multiplier is treated as a sign bit or a data bit. For example, the value 0xFF (as an 8-bit quantity) is either -1 (as a signed quantity) or 255 (as an unsigned quantity). MULtiplying 0xFF by itself would result in storing 0xFF01 in the AX register, while the IMUL instruction would produce 0x0001 (since -1 times itself is of course 1).

Division uses the same rather complicated register sets, but in reverse. The dividend (number to be divided) is put into a pair of registers, either the AH:AL pair or the DX:AX pair. The argument to the division is used as the dividend. The quotient is stored in the low half of the original register pair, the remainder in the high half. Using a previous example
As with multiplication, there are two instructions, `DIV` and `IDIV`, performing unsigned and signed integer division, respectively.

### 6.3.3 Floating Point Operations

The 8088 FPU is almost a separate, self-contained computer, with its own registers and its own idiosyncratic instruction set, specifically for performing floating point operations. Actually, it is a separate, self-contained chip, sold under the model number 8087, as a math coprocessor. Data is transported as necessary from the main CPU of the 8088 to the coprocessor and back. The unnamed 8087 registers are a stack (yes, just like the JVM) of eight 80-bit wide storage locations, and again, just like the JVM, the instruction set is structured to address the operation type as well as the representation type.

The FPU can store and process three different types of data: standard IEEE floating point numbers, integer values, and a special format called binary-coded-decimal (BCD), where each four-bit group represents the binary encoding of a single base-10 digit. This format was often used specifically by IBM mainframe computers, because it was easy for engineers to re-interpret the binary patterns as (intuitive) decimal numbers and vice versa. Inside the 8087, all these formats are converted to and stored in an 80-bit format that is substantially more accurate than the standard format defined by the IEEE.

Operations for the FPU all begin with the letter `F` (again, this should feel familiar to JVM programmers), and operate as one might expect on the top of the stack like the JVM or a reverse Polish calculator. The `FADD` instruction pops the top two elements of the FPU stack, adds them together, then pushes the result. Other arithmetic operations include `FSUB`, `FMUL`, and `FDIV`, which perform as expected. There are two additional operations: `FSUBR` and `FDIVR` which perform subtraction (division) in “reverse”, subtracting the top of the stack from the second element instead of subtracting the second element from the top.¹

---

¹**SIDE BAR : OTHER FPU ARGUMENT FORMATS.** Although the FPU always manipulates data using an internal 80-bit “extended precision” format, data can be loaded/stored in main memory in a variety of formats. Conversion happens at load/store time, depending upon the operation mnemonic used. There are many different instructions, most of which have several interpretations, as follows:

- **Integers**: Integer data, either 16- or 32- bit, is loaded with the `FILD` instruction. This can only load from memory, not from a register. On later machines, 64-bit quantities can also be loaded with this instruction.
- **BCD integers**: Integer data 80-bit in Binary Coded Decimal format is loaded with
special purpose instructions like FSQRT to handle common math functions like square roots and trigonometric functions.

Data can be pushed (loaded) into the FPU stack via the FLD instruction. This actually comes in three flavors: FLD loads a 32- or 64-bit IEEE floating point number from a given memory location, FILD loads a 16- or 32-bit integer from memory (and converts it to internal floating point), and FBLD loads an 80-bit BCD number from memory (and converts). There are a few special purpose operations for commonly used constants: FLD1 loads the value 1.0, FLDZ loads 0.0, FLDPI loads an 80-bit representation of $\pi$, and a few more instructions specify quantities like commonly used logarithms, like the natural log of 2. To move data from the FPU to storage, use some variation of the FST instruction — again, there are variations for integer (FIST) and BCD (FBST) storage. Some operations have additional variations, marked with a trailing -P, to pop the stack when the operation is complete (for example, FISTP STores the Integer at the top of the stack, and then pops the stack.) One limitation of the FPU is that data can only be loaded/stored from memory locations, not directly from ALU registers. So a statement like

$$\text{FILD AX}$$

is illegal; the value stored in AX must first be MOVED to a memory word and then loaded from that location as follows

$$\text{MOV Location,AX}$$
$$\text{FILD Location}$$

### 6.3.4 Conical mountains revisited

As a worked-out example of how arithmetic on the 8088 looks, let’s re-solve the problem given earlier on the volume of a given mountain. As given in chapter 2, the original problem statement is

What is the volume of a circular mountain 450m in diameter at the base and 150m high?

Because the answer involves using $\pi$, at least part of the computation needs to be in the FPU. We assume (for clarity) that the name/identifier “STORAGE” somehow refers to a 16-bit memory location (we’ll see how to do this later in section 6.3.7), and we can use that location to move data in and out of the FPU.

Floating point numbers: Floating point numbers, in 32-bit, 64-bit, and 80-bit lengths, are loaded with the FLD instruction.

Once these quantities are loaded, they are treated identically by the FPU. To store a value, replace “LD” with “ST” in the mnemonics above.
For simplicity, we’ll use 16-bit integers and registers for our calculations; since none of the numbers involved are very large, this will work without too much problem.

Step one is to calculate the radius

`MOV AX, 450 ; diameter = 450m`
`MOV BX, 2 ; can’t divide by a constant`
`DIV BX ; AX = 450 / 2, DX is still 0`

The area of the base is the radius squared, times $\pi$. In order to use $\pi$, we need to initialize the FPU and move the computation over there.

```
; AX holds radius
MUL AX ; square AX, result in DX:AX
FINIT ; initialize FPU
MOV STORAGE, AX ; move $r^2$ to memory (as integer)
FILD STORAGE ; ... and move to FPU
FLDPI ; load pi = 3.1415...
FMUL ; and calculate base area
```

At this point, we could move the base area from the FPU back into the main ALU, but that would inevitably mean that we lose everything to the right of the decimal point (and thus accuracy). A better solution is to continue our calculations in the FPU, using memory location STORAGE as a temporary holding spot to move integer data. To recap: the volume of a cone is a third the volume of a corresponding cylinder, and the volume of the cylinder is the base area (already calculated) times the height (150m). So the final stage of the calculations looks like

```
MOV STORAGE, 150 ; get height into the FPU
FILD STORAGE ;
FMUL ; cylinder volume = base * height
MOV STORAGE, 3 ; get 3 into the FPU for division
FILD STORAGE ;
FDIV ; ... and divide by three
; for cone volume
; the result is now at the top of the FPU stack
FISTP STORAGE ; ’P’ means pop the stack after
; performing operation (storing)
; STORAGE now holds the final answer, rounded to an integer
FWAIT ; Wait for FPU to finish before
; doing anything in the ALU
```

6.3.5 Decisions and control structures

Like most assembly languages, the 8088 control structures are built on unconditional and conditional jump instructions, where control is transferred to a label declared in the source code. As with the JVM, this is actually handled by computing an
offset and adding/subtracting that offset to the current location in the program counter. The format of the jump instruction (mnemonic: JMP) is also familiar to us.

```
LABEL: JMP LABEL ; silly infinite loop
```

Conditional jumps use a set of binary flags, grouped together in the flags register in the CPU. These flags hold single-bit descriptors of the most recent result of computation: for example, the zero flag ZF is set if-and-only-if the result of the most recent operation (in the ALU) was a zero. The sign flag SF contains a copy of the highest bit (what would be the sign bit, if the result is a signed integer), and is set if-and-only-if the result of the last number is negative. The carry bit CF is set if-and-only-if the most recent computation generated a carry out of the register, which (when unsigned calculations are being performed) signals a result too large for the register. The coverflow bit OF handles similar cases, cases where when signed calculations are being performed, the result would be too large (or too small) for the register. There are several other flags, but these are the main ones that get used.

A conditional jump has a mnemonic of the form “Jcondition,” where condition describes the flag setting that causes the jump to be taken. For example, JZ means jump-if-zero-flag-set, while JNZ means jump-if-zero-flag-not-set. We can use this to test whether two values are equal.

```assembly
SUB AX, BX ; AX = AX - BX
JZ ISEQUAL ; if we got here, AX didn’t equal BX
JMP OUTSIDE ;
ISEQUAL:
; if we got here, AX did equal BX
OUTSIDE:
; rejoin after if/else statement
```

Other conditional jumps include JC/JNC (jump if CF set/clear), JS/JNS (jump if SF set/clear), JO/JNO (jump if OF set/clear), and so forth. Unfortunately, not all of the flags have nice clear arithmetic interpretations, so a second set of conditional jumps are available to handle proper arithmetic comparison such as “greater than,” “less than or equal to,” and so forth. These instructions interpret flags in combination as appropriate to the arithmetic relationship.

In more detail, these additional jumps expect that the flags register contains the result of SUBtracting the first number from the second, as in the example fragment immediately above. (This is a micro-lie; wait a bit for a more detailed explanation of the CMP instruction.) To compare whether or not one signed integer is greater than another, the JG (Jump if Greater) mnemonic can be used. Other mnemonics include JL (Jump if Less), JLE (Jump if Less than or Equal), and JGE (Jump if Greater or Equal). These also exist in negative form — JNGE (Jump if Not Greater...
or Equal) is of course identical to JL, and are in fact implemented as two different mnemonics for the same instructions. Similarly, JE exists, and is equivalent to the previously defined JZ, and JNE is the same as JNZ.

For comparison of unsigned integers, a different set of instructions is needed. To see why, consider the 16-bit quantity 0xFFFF. As a signed integer, this represents -1, which is less than 0. As an unsigned integer, this represents the largest possible 16-bit number, a shade over sixty-five thousand — and this, of course, is greater than 0. So the question “is 0xFFFF > 0x0000” has two different answers, depending upon whether or not the numbers are signed. The 8088 provides, for this purpose, a set of comparison instructions based around Above and Below (i.e. JA, JB, JAE, JBE, JNA, JNB, JNAE, JNBE) for comparing unsigned numbers. So to determine if the value stored in AX would fit into an 8-bit register, the following code fragment suffices:

```
; version one of 8-bit safety test
SUB AX, 100h ; subtract 2^8 from AX
JAE TOOBIG ; unsigned compare
; If it gets here, the number is fine
JMP OUTSIDE

TOOBIG:
; if it gets here, AX is bigger than 8 bits
OUTSIDE:
; continue doing what’s needed
```

One problem with the preceding code fragment is that in order to set the flags properly, the value of AX is modified. One possible solution is to store the value of AX (MOV it to a memory location) and re-load it after setting the flags. This would work...and the MOV instruction has been specifically set up to leave the flags register alone and to preserve the previous settings. So we can rewrite our 8-bit safety test, with slightly less space and time efficiency, as

```
; version two of 8-bit safety test
MOV SOMEWHERE, AX ; store AX SOMEWHERE
SUB AX, 100h ; subtract 2^8 from AX
MOV AX, SOMEWHERE ; restore AX, leave flags
JAE TOOBIG ; unsigned compare
; If it gets here, the number is fine
JMP OUTSIDE

TOOBIG:
; if it gets here, AX is too big
OUTSIDE:
; continue doing what’s needed
```

The Intel instruction set provides a better solution with a special purpose command. Specifically, the CMP mnemonic performs a non-destructive subtraction. This instruction calculates the result of subtracting the second argument from the
first (as has been done in the examples above), and sets the flags register accord-
ingly, but does not save the subtraction result anywhere. So rewriting the first
version as

```
; version three of 8-bit safety test
CMP AX, 100h ; compare 2^8 to AX
JAE TOOBIG ; unsigned compare
; If it gets here, the number is fine
JMP OUTSIDE
TOOBIG:
; if it gets here, AX is bigger than 16 bits
OUTSIDE:
; continue doing what’s needed
```

preserves the efficiency of the first version while not destroying the value stored in
the registers.

In addition to these fairly traditional comparison and branch instructions, Intel
provides a few special-purpose instructions (this is getting repetitive, isn’t it?) to
support efficient loops. The register tuned for this purpose is the CX register.
Specifically, the computer can be told to jump if the value of the CX is zero with
the JCXZ instruction. Using this instruction, one can set up a simple counter-
controlled loop with

```
MOV CX, 100 ; loop 100 times
BEGIN:
; do something interesting
DEC CX ; subtract 1 from counter
JEXZ LOOPEXIT ; quit if done (CX==0)
JMP BEGIN ; return to BEGIN
LOOPEXIT:
; now outside of loop
```

Even more tersely, the LOOP instruction handles both the decrementing and
branch — it will decrement the loop counter and branch to the target label if the
counter has not yet reached zero. Thus, we can simplify the loop above to a single
statement of interest :

```
MOV CX, 100 ; loop 100 times
BEGIN:
; do something interesting, quit when CX == 0
LOOP BEGIN ; return to BEGIN
; now outside of loop
```

Using the results of floating point comparisons can also be tricky. The basic
problem is that the flags register is located in the main CPU (in the control unit),
which also handles branch instructions through the PC in the control unit. At the
same time, all the floating point numbers are stored in the FPU, in a completely
separate section of silicon. The data must be moved from the FPU into the normal flags registers, using a set of special-purpose instructions perhaps beyond the scope of this discussion.\(^2\)

### 6.3.6 Advanced operations

The 8088 provides many more operations and mnemonics than space really permits description of. Many of them, perhaps most, are shortcuts to perform (common) tasks in fewer machine instructions than it would take using the simpler instruction(s). An example is the XCHG (eXChanGe) instruction, which swaps the source and destination arguments around. Another example is the XLAT instruction, which uses the AL register as a table index and adjusts the value in AL by the amount stored in the table. Essentially, this is a one-operation abbreviation for \(AL = [AL + BX]\), which would take several steps to perform using the primitive operations described earlier. On the later computers in the 80X86 family, this tendency to additional operations becomes much more pronounced and the instruction set becomes substantially larger and more complex, to the point where almost no human programmer today knows all of the Pentium 4 instructions in detail. (This is one reason that writing a good compiler is, a) tricky, and b), important. The smarter you make the compiler, the dumber the programmer can be.)

The 8088 also supports operations on string and array types, using (yet another) set of special purpose instructions. We’ll see these in action a little in section 6.3.10, since strings and arrays more or less have to be stored in memory (registers aren’t big enough).

### 6.3.7 Addresses and variables

The segmented organization of the 8088’s memory has already been described. Every possible register pattern represents a possible byte-location in memory. For larger patterns (a 16-bit word or a 32-bit “double”, or even an 80-bit “tbyte”, containing ten bytes and holding an FPU value), one can co-opt two or more adjacent byte-locations. As long as the computer can figure out how many bytes are in use, accessing memory of varying sizes is fairly simple.

\(^2\)SIDEBAR: Oh, all right. If you insist. The FPU provides both an FCOM instruction, which compares the top two elements on the stack, and an FTST instruction, which compares the top element to 0.0. This comparison is stored in a “status word,” the equivalent of the flags register. To actually use the information, though, the data must be moved, first to memory (because the FPU cannot access CPU registers directly), then to a register (because the flags register cannot access memory directly), and finally into its eventual destination. The instruction to do the first is FSTSW (STore Status Word, which takes a memory location as its single argument), for the second an ordinary MOV into the AX register suffices, and for the third, the special purpose SAHF (Save AH in Flags) instruction is used. Ordinary unsigned conditional jumps will then work properly. The complexity of this process explains and illustrates part of why it’s so much faster to use integer variables when writing a computer program.
Unfortunately, this information is not always available to the computer. An earlier micro-lie suggested that

```
ADD [BX], 1 ; increment the value pointed to by BX
```

was legal. Unfortunately, the value stored in BX is a location, and as such, there is no easy way of knowing whether the destination is a 16-bit or 8-bit quantity. (Similarly, we don’t know whether we need to add 0x0001 or 0x01.) Depending upon these sizes, this statement could be interpreted/assembled as any of three different machine instructions. The assembler (and we) need a little hint to know how to interpret [BX]. This would also apply when a specific memory address is used directly, as in

```
ADD [4000h], 1 ; increment the value at 0x4000
```

There are two ways of giving the computer such a hint. The simpler, but less useful, is to explain exactly what is meant in that line, and in particular, that [4000h] should be interpreted as the address of (“pointer to”) a word (16 bits) by re-writing the line.

```
ADD WORD PTR [4000h], 1 ; increment the 16-bit value at 0x4000
```

By contrast, using BYTE PTR would force an 8-bit interpretation, and using DWORD PTR (double word), a 32-bit one.

A more general solution is to simply notify the assembler in advance of one’s intentions to use a particular memory location and of the size one expects to use. The name of this location can then be used as shorthand for the contents of that memory location as a direct mode operation. This has approximately the same effect as declaring a variable in a higher-level language such as Java, C++, or Pascal. Depending upon the version and manufacturer of the 8088 assembly, either of the following will work to define a 16-bit variable (with names selected by the programmer):

```
example1 WORD 1000 ; 1000
example2 DW 2000h ; 2000h == 0x2000
```

The values can now be used in direct mode more or less at will

```
MOV AX, example1 ; AX is now 1000
ADD example2, 16 ; example2 is now 2010h
CMP AX, example2 ; is AX > example2? (no)
```

This statement serves several purposes: first, the computer now knows that two 16-bit chunks of memory have been reserved for program data. Second, the programmer has been relieved of the burden of remembering exactly where these chunks are, since she can refer to them by meaningful names. Third, the assembler already knows their sizes and can take that into account in writing the machine code. That’s not to say that the programmer can’t override the assembler
exple3 WORD 1234h ; define 16-bit space
ADD BYTE PTR exple3, 1 ; legal but silly

but this is very likely to result in a bug in the program. (By contrast, of course, the JVM gets very annoyed if you try to access only part of a stored data element, and generally won’t let you do it.)

Assemblers will accept a wide variety of types for memory reservation, including some types so big they can’t easily be handled in registers. Using the more modern syntax, any of the following are legal definitions.

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny</td>
<td>BYTE 12h</td>
<td>single byte</td>
</tr>
<tr>
<td>Small</td>
<td>WORD 1234h</td>
<td>two bytes</td>
</tr>
<tr>
<td>Medium</td>
<td>DWORD 12345678h</td>
<td>four bytes</td>
</tr>
<tr>
<td>Big</td>
<td>QWORD 1234567812345678h</td>
<td>eight bytes</td>
</tr>
<tr>
<td>Huge</td>
<td>TBYTE 12345678901234567890h</td>
<td>ten bytes,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(used by FPU)</td>
</tr>
</tbody>
</table>

Floating point constants can be defined with either REAL4 (for 32-bit numbers), REAL8 (for 64-bit), or REAL10 (for 80-bit). The number values themselves are usually written either normally or in exponential notation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sqrt2</td>
<td>REAL4</td>
<td>1.4143135</td>
<td>real number</td>
</tr>
<tr>
<td>Avogad</td>
<td>REAL8</td>
<td>6.023E+23</td>
<td>exponential notation</td>
</tr>
</tbody>
</table>

Memory locations can be defined without initializing them by using ‘?’ as the starting value. Of course, in this case the memory will still hold some pattern, but it’s not predictable just what it is, and if you try to use that value, bad things will probably happen.

### 6.3.8 Byte swapping

How does storage in memory compare with storage in registers? Specifically, if we had the 16-bit pattern 0100 1000 1000 0100 stored in the AX register, is that the same as the same 16-bit pattern stored in memory?

The answer, surprisingly, is “no”! (Perhaps it’s not that surprising, since if it really were that simple, this section of the book probably wouldn’t exist.) Again, as a legacy of older machines, the 8088 has a rather odd storage pattern.

When writing down numbers, people usually use so-called big-endian notation. The so-called most significant numbers (the ones corresponding to the highest powers of the base) are written (and stored) first, and the smaller, less significant numbers trail afterwards. Quick check for clarification — numbers are traditionally written on paper in big-endian format; the first digit written involves the largest power of base. The 8088 CPU similarly stores values in registers in big-endian order. The value written above (0x4884) would thus represent the decimal value 18564. Data stored in memory, however, are actually stored in little-endian format, by bytes.
Figure 6.2: Memory storage of 0x12345678, illustrating “byte swapping”

The first byte (0x48, 0100 1000) is the least significant byte, the second one is the most. (Miss Williams, my seventh-grade English teacher, would have insisted that, with only two bytes, one can’t have a “most” significant, only a “more” significant. This is one area where specialized jargon trumps traditional English grammar.) So this pattern in memory would represent 0x8448, almost twice as large. This pattern continues with larger numbers, so the 32-bit quantity 0x12345678 would be stored as four separate memory bytes as in the figure:

Fortunately, the programmer rarely needs to remember this. Any time data is moved to or from memory, the byte swapping happens automatically at the hardware level. Other than when the programmer explicitly overrides the assembler’s knowledge of data sizes (as in the previous section), the only time this might become important is in dealing with large groups of data such as arrays and their extension, strings. (Of course, “rarely” doesn’t mean the same thing as “never,” and when it does become important, this can be a source of the sort of error that can have you pounding your head against a wall for a week.)

### 6.3.9 Arrays and strings

The same notation used to reserve a single memory location can also be used to reserve large, multi-location blocks of memory. Values can either be set using a comma-separated list of values, or by using a shorthand DUP notation for repeated values. For example

```plaintext
Greet BYTE 48h,45h,4Ch,4Ch,4Fh ; ASCII "HELLO"
Thing DWORD 5 DUP 0x12345678 ; 5 identical values
Empty WORD 10 DUP (?) ; 10 empty slots
```

define, respectively, Greet as a five-byte array of the letters (H, E, L, L, and O), Thing as a set of doublewords, and Empty as an array of 10 2-byte values, none of which are initialized.
6.3. PROGRAMMING

To access elements of these arrays, the programmer needs to index or offset from the named array base. If the ‘H’ in Greet were stored at 3000h, then the ‘E’ would be stored at 3001h, the first ‘L’ at 3002h, and the ‘O’ at 3004h. Of course, we don’t know where Greet happens to be stored... but wherever it is, the location of the ‘H’ is one less than the location of the ‘E.’ By convention, the array name itself refers to the location of initial value of the array. So, to load the first three letters into the AH, BH, and CH (byte) registers, we can simply

```assembly
MOV AH, [Greet] ; Load H
MOV BH, [Greet + 1] ; Load E
MOV CH, [Greet + 2] ; Load L
```

This is actually a new (to this book, at least) addressing mode, called index mode. As before, the notation [X] means “the contents of memory stored at location X,” but the X in this case is a rather complex value that the computer calculates on the fly. It should be intuitive that [Greet] is the same as [Greet + 0] and as Greet itself, while [Greet + 1] is the next byte over. And because the computer knows that Greet hold bytes (as defined by the memory reservation statement), it will assume that [Greet + 1] is also a byte.

So how do we access elements of the Empty array? The initial element is simply [Empty], or [Empty+0], or even Empty itself. In this case, though, Empty holds WORD objects, so the next entry is not [Empty+1], but [Empty+2]! Unlike most high-level languages, where arithmetic on indices automatically takes the kind of elements involved into account, index mode addressing on the 8088 requires the programmer to handle size issues.

Index mode addressing has more general uses involving registers. In high level languages, for example, one of the most common array actions to take is to use a variable index; for example, accessing element “a[i]” inside a loop involving an integer variable i. The assembly language equivalent uses a general-purpose register as part of the index expression. The expression [Greet + BX] would refer to the “BX-th” (if that word even makes sense) element of the array Greet. By adjusting the value in BX (say, from 0 to 4), the expression [Greet + BX] will sequentially select each element. Similarly, by adjusting the value in BX by 2, the size of a word, each time, we can initialize the entire Empty array to zero with

```assembly
MOV CX, 10 ; 10 elements in Empty
MOV BX, 0 ; start at [Empty + 0]
BEGIN:
    MOV [Empty+BX], 0 ; zero out this element
    ADD BX, 2 ; move to next WORD
    LOOP BEGIN ; loop until CX == 0
```

Only a few of the 16-bit registers can legally be used as an index in this sort of an expression, and none of the 8-bit ones are legal. Only the BX, BP, SI, and DI registers can be used, and, bluntly, one shouldn’t mess with the BP register.
for this purpose as bad things are likely to occur. The BP register is already used by the operating system itself for its own nefarious purposes, as discussed later in section 6.3.12.

Experienced C and C++ programmers may already be chafing at the bit for a faster and more efficient way. Instead of calculating [Empty+BX] at each pass through the loop, why not set BX itself to the spot where [Empty+0] is stored, and then just use [BX]? This suggests code something like

```assembly
; warning, doesn’t work!
MOV CX, 10 ; 10 elements in Empty
MOV BX, Empty ; start at [Empty+0] (WRONG!)
BEGIN:
   MOV [BX], 0 ; zero out this element
   ADD BX, 2 ; move to next WORD
   LOOP BEGIN ; loop until CX == 0
```

Although the idea is good, the execution falters, mostly because MOV BX, Empty doesn’t actually mean what we hoped it would. The assembler treats Empty as a simple byte variable in direct mode, and will try to move the first byte of Empty (the ‘H’) into BX. This isn’t what we want—in fact, it isn’t even legal, since we’re trying to move a single byte into a four-byte register, which results in a size conflict. To explicitly get a pointer variable, we use the OFFSET keyword, which produces the memory location instead of its contents.

```assembly
; improved, functional version
MOV CX, 10 ; 10 elements in Empty
MOV BX, OFFSET Empty ; start at [Empty + 0]
BEGIN:
   MOV [BX], 0 ; zero out this element
   ADD BX, 2 ; move to next WORD
   LOOP BEGIN ; loop until CX == 0
```

Of course, the actual time/space improvement from this may be rather marginal, since the index addition is still performed within a single machine-operation on the 8088. But every little improvement may help, especially in a tight, small, often-executed loop.

### 6.3.10 String primitives

Strings can be implemented simply as arrays of characters (most often bytes, but sometimes larger), as with the Greet example. The 8088 also provides some so-called string primitive operations for performing common string functions quickly and easily. These basic operations all use SI, DI or both— that’s the special purpose to which SI and DI are optimized— for these operations.

We’ll focus, for the moment, on the simple task of copying or moving a string from one location to the other. Depending upon the size of the string elements,
there are two basic operations: MOVSB (MOVe a String of Bytes) and MOVSW (MOVe a String of Words). This twofold division based on size holds for all string primitives; there are also (for example) two variations for doing comparisons, ending with B and W, respectively. So for simplicity of explanation, we’ll fold these similar-behaving operations together under the name MOVS?.

The MOVS? operation copies data from [SI] to [DI]. By itself, it will copy only a single element, but it’s easy enough to put it in a simple loop structure. The advantage of the string primitives is that the CPU supports automatic looping in the machine instruction, expressed as the assembly language level as a prefix to the mnemonic. The simplest example would be the REP prefix, as in

```assembly
REP MOVSB
```

This acts rather like the LOOP? instruction, in that the CX register is used as a counter. Every time this instruction is performed, the values of SI and DI will be adjusted, the value of CX will be decremented, and the instruction will repeat until CX drops to zero.

There are two main variations in how SI and DI are adjusted. First, the amount of the update automatically corresponds to the size of element specified in the MOVS? command, so SI/DI will be changed by 1 for a Byte instruction or by 2 for a Word instruction. Second, a special flag in the flags register (the Direction flag) controls whether the addresses are adjusted from low to high (by adding to SI/DI) or from high to low (by subtracting). This flag is controlled by two instructions as in table 6.2.

To see how this works, let’s make a couple of arrays and copy them.

```assembly
Arr1 WORD 100 DUP (-1) ; source array : 100 words
Arr2 WORD 100 DUP (?) ; dest array : also 100 words
MOV SI, OFFSET Arr1 ; set source address
MOV DI, OFFSET Arr2 ; set destination address
CLD ; clear direction flag
; so goes from Arr1[0] to Arr1[99]
MOV CX, 100 ; loop over 99 words
REP MOVSW ; ... and do the copy
```

Another common operation is to compare two strings for equality. This can be done with the CMPS? operation. This performs an implicit subtraction of the
destination from the source. Important safety tip: this is the reverse of the CMP instruction, which subtracts the source from the destination! More importantly, this instruction sets the flags such that normal unsigned conditional jumps will do The Right Thing.

Another variant of the REP prefix is particularly useful in this context. REPZ (alternatively, REPE) loops as long as both CX is not zero and the zero flag is set. This really translates to “as long as we haven’t hit the end of the string and the strings so far have been identical,” since the zero flag is set only when the result of the subtraction is zero, meaning the last two characters are the same. Using this, we can perform general string-comparison operations with

\[
\begin{align*}
\text{MOV} & \quad \text{SI, OFFSET Astring} \quad ; \text{set source address} \\
\text{MOV} & \quad \text{DI, OFFSET Bstring} \quad ; \text{set destination address} \\
\text{CLD} & \quad ; \text{clear direction flag} \\
\text{MOV} & \quad \text{CX, 100} \quad ; \text{loop over (up to) 99 words} \\
\text{REPE} & \quad \text{CMPSW} \quad ; \ldots \text{and compare}
\end{align*}
\]

After this code fragment has run, one of two possible things has happened. Possibility one: CX hit zero with the Z flag still set, in which case the two word strings are identical. So a simple statement like

\[
\text{JE STRINGSEQUAL} \quad ; \text{the strings were equal}
\]

can branch to the appropriate section of code.

Alternatively (possibility two), the Z flag was cleared when two elements were compared and found to be different. The detailed results of the difference (was the byte in SI greater or less than the byte in DI?) are stored in the flags register like the result of a CMP operation. By examining the other flags with JB, JA, JAE, etc. we can figure out which whether the source (SI) or destination (DI) register pointed to the smaller string. The main difficulty about this is that the SI and DI register are left pointing to the wrong spot. Specifically, the value in SI (DI) is the location just past where the strings were found to differ . . . or alternatively one slot past the end of the strings.

\[
\begin{align*}
\text{JB} & \quad \text{SOURCESMALLER} \quad ; \text{SI held smaller string} \\
& \quad \text{if we get here, DI < SI}
\end{align*}
\]

A third useful operation is to look for the occurrence (or lack) of a particular value within a string. (For example, a string that holds a floating point number will have a ‘.’ character in it, otherwise it would be an integer.) This is handled by the SCAS? (SCAN String) instruction. Unlike the previous instructions, this only involves one string, and hence one index register (DI). It compares the value in the accumulator (AL or AX, depending upon the size) with each element, setting flags and updating DI appropriately. Again, if the appropriate REP? prefix is used, it will stop either when CX hits zero at the end of the string, or else when the Z flag hits the correct value, in any case leaving DI pointing one location past the spot of interest.
Using the REPZ prefix, we can use this to skip leading or trailing blanks from a string. Assume that Astring contains an array of 100 bytes, some of which (at the beginning or end) are space characters (ASCII 32). To find out where the first non-blank character is, we can use this code fragment.

```
MOV DI, Astring ; load string
MOV AL, 32 ; load byte accumulator with ' '
MOV CX, 100 ; 100 characters in Astring
CLD ; clear direction flag
REPE SCASB ; scan for mismatch
JE ALLBLANKS ; if Z flag set, no mismatch found ; otherwise, DI points one byte past first nonblank character
DEC DI ; back DI up
```

To skip trailing blanks, we simply start at the end (at Astring+99), and set the direction flag so that the operation goes from right to left in the string.

```
MOV DI, Astring ; load string
ADD DI, 99 ; jump to end of string
MOV AL, 32 ; load byte accumulator with ' '
MOV CX, 100 ; 100 characters in Astring
STD ; set direction flag
REPE SCASB ; scan for mismatch
JE ALLBLANKS ; if Z flag set, no mismatch found ; otherwise, DI points one byte past first nonblank character
INC DI ; back DI up
```

A similar prefix, REPNZ (or REPNE) will repeat as long as the Z flag is not set, which is to say, as long as the elements differ. So to find the first '.' (ASCII 44) in a string, we use a slight variation on the first example:

```
MOV DI, Astring ; load string
MOV AL, 44 ; load byte accumulator with '.'
MOV CX, 100 ; 100 characters in Astring
CLD ; clear direction flag
REPNE SCASB ; scan for match
JNE NOPERIOD ; if Z flag clear, no match found ; otherwise, DI points one byte past first '.'
DEC DI ; back DI up
```

Finally, the last commonly useful string primitive will copy a particular value over and over again into a string. This can be useful to quickly zero out an array, for example. The STOS? (STOre String) copies the value in the accumulator to the string. To store all zeros in the Empty array previously defined (an array of ten doublewords), we can simply

```
MOV DI, Astring ; load string
MOV AL, 0 ; load byte accumulator with 0
MOV CX, 20 ; 20 characters in Empty
CLD ; clear direction flag
REPE SCASB ; scan for mismatch
JE ALLBLANKS ; if Z flag set, no mismatch found ; otherwise, DI points one byte past first nonblank character
INC DI ; back DI up
```

```
```
MOV DI, Empty ; store destination string
MOV CX, 10 ; 10 elements to store
MOV AX, 0 ; value to store is 0
REP STOSD

Unfortunately, this is really all the support that it provides for user-defined derived types. If the programmer wants a multidimensional array, for example, she must figure out herself how to tile/parcel the memory locations out. Similarly, a structure or record would be represented just by adjacent memory locations, and no support at all is provided for object-oriented programming. This must be addressed at a higher level through the assembler and/or compiler.

6.3.11 Subroutines, functions, and the stack

One problem with the memory structure defined so far is that every memory location is defined implicitly for the entire computer. In terms common to higher-level language programming, every example of a variable we’ve seen is global — meaning that the Greet array (previously defined) could be accessed or changed from anywhere in the program. It also means that there can only be one variable in the entire program named Greet. Better programming practice calls for the use of local variables, which gives both a certain degree of privacy and security as well as the ability to re-use names.

Similarly, as discussed on the JVM, only having jump instructions available limits the programmer’s ability to reuse code. The solution, for both the JVM and the 8088, is to support subroutines (or subprograms). As with the JVM’s jsr instruction, the 8088 provides a CALL instruction in conjunction with a hardware stack. This instruction pushes the current value of the instruction pointer (IP) and executes a branch to the location given as an argument. The corresponding RET instruction pops the top value from the stack, loads it into the instruction pointer, and continues execution at the saved location.

The 8088 also recognizes standard PUSH and POP instructions for moving data to and from the machine stack. For example, good programming practice suggests that one shouldn’t wantonly destroy the contents of registers inside subroutines, since there’s no way to be sure that the calling environment didn’t need that data. The easiest way to make sure this doesn’t happen is to save (PUSH) the registers that one plans to use at the very beginning of the subroutine, and to restore (POP) them at the end. Both the PUSH and POP statements will accept any register or memory location; both PUSH AX as well as PUSH SomeLocn are legal. To push a constant value, one must first load it into memory or a register — and, of course, POP-ping something into a constant doesn’t make much sense.

Most assemblers discourage the practice of using the same labels for both subroutine calls and jump statements, although the CPU doesn’t care (after all, they’re both “really” just numeric values to be added to the program counter!) However, if not done extremely carefully, the programmer will violate stack discipline, and end up either leaving extra stuff on the stack (resulting in filling it up and getting some
sort of overflow-based error), or else popping and using garbage from an empty stack. In other words, don’t do that. For this reason, setting up a subroutine in 8088 assembly language looks a little bit different than the labels we’ve already seen:

```
MyProc PROC
    PUSH CX ; push CX to save it
    ; do something extremely clever
    MOV CX, 10
    MyLabel:
    ; do something clever inside a loop ten times
    LOOP MyLabel
    POP CX ; restore CX
    RET
MyProc ENDP
```

There are a few points to pay attention to here. First, notice that the declaration of the label MyProc looks different from the label MyLabel (there’s no colon, for instance), to help both you and the assembler keep track of the difference. Second, notice that the procedure begins and ends with PROC/ENDP. These aren’t actually mnemonics, merely directives, as they don’t translate to any machine instructions. They just (again) help you and the assembler structure the program. The last actual machine instruction is RET, which is not only typical, but de facto required for a subroutine. Thirdly, notice that the CX register is used for a loop index inside the routine, but since the value is PUSHed at the top and POP-ped at the bottom of the routine, the calling environment will not see any change in CX. It would be legal (and even typical) to invoke this routine from another routine as follows:

```
Other PROC
    MOV CX, 50 ; call MyProc 50 times
LoopTop:
    CALL MyProc ; execute MyProc subroutine
    LOOP LoopTop ; ... in a loop (50 times)
    RET
Other ENDP
```

The “Other” procedure uses the same loop structure, including CX, to call MyProc fifty times, but since the CX register is protected, no errors will result. Of course, the Other procedure itself clobbers the CX register, so someone calling Other had better be careful. (A better version of Other — the sort expected of a professional programmer — would similarly protect CX before using it, using the same push/pop instructions.)
6.3.12 The stack and stack frames

In addition to providing temporary storage for registers, the stack can also be used to provide temporary storage for local variables in memory. To understand how this works, we'll first look at the details of how the stack itself works:

One of the “general purpose” registers of the 8088, the SP register, is for all practical intents reserved by the CPU and operating system to hold the current location of the top of the “machine stack.” At the beginning of any program, the value of SP is set to a number somewhere near the top of main memory — meaning, the part of main memory that has relatively high addresses, while the program itself is stored in much lower locations. Between the program and the top of the stack is a large no-man’s-land of unused and empty memory.

Whenever data needs to be placed onto the stack (via a PUSH or a CALL, typically), the SP register is decremented by the appropriate amount, usually 2. This pushes the top of the stack two bytes closer to the rest of the program, into no-man’s-land. The value to be pushed is stored in these “new” two bytes. Another PUSH would bring SP down another two bytes, and store another piece of data. Counterintuitively, this means that the “top” of the stack is actually the part of the stack with the lowest (smallest) address. Contrariwise, when data is POP-ped from the stack, the value at [SP] is copied into the argument, then SP is incremented by 2, setting it to the new “top.” (And, of course, this also applies when you execute RET and have to take [pop] the old program counter from the stack.)

However, the stack also provides a perfect location to store local variables, since every time a procedure is called, this results in a new top of the stack. In fact, any information that needs to be local to the procedure, such as function arguments, local variables, saved registers, and stuff, can be put on the stack. Since this is such a common task, especially in high-level languages, there’s a standard way of structuring the stack so that different and complex procedures can “play nice” with each other.

The basic idea is called a stack frame. As usually implemented, it involves two registers, SP and BP. This is why the programmer shouldn’t mess with the BP register for general-purpose indexing, as it’s used already in the stack frames. But because BP works as an index register, expressions like MOV AX, [BP + 4] are legal and can be used to refer to memory locations near BP.

With this in mind, a stack frame looks like this (starting at the top of memory, or the “bottom” of the stack)

- Any arguments to the procedure/subprogram
- Return address
- Old value of BP (pointed to by BP)
- Space for local variables (top pointed to by SP)
- Saved registers
Figure 6.3: The CPU stack (simplified)
Figure 6.4: An 80x86 stack frame
public static int further(int x, int y) {
    int i,j;
    i = x;
    if (i < 0)
        i = -x;
    j = y;
    if (y < 0)
        j = -y;
    if (i < j)
        i = j;
    return i;
}

Figure 6.5: Function further(int x, int y) in Java

How does this work in action? We’ll use a somewhat contrived example: I want to write the assembly language equivalent of a function or method further() that takes two arguments and returns the absolute value of the one more distant from 0. In Java, this method would be written as in figure 6.5; in C/C++, it would look more like figure 6.6.

The code for the comparison itself is simple; assuming that we can get the data into AX and BX, we can simply compare the two, and if BX is the larger, move it into AX. Similarly, by comparing the function parameters (x and y) to zero, we can either use a NEG instruction or not to on their local variables. However, to access these properly, we need to have enough space on the stack to hold these local variables.

The following code will solve that problem nicely

Further PROC
    PUSH BP ; save old BP
    MOV BP, SP ; BP now points to old BP
    SUB SP, 4 ; make two 16-bit local
                ; variables
    PUSH BX ; save BX

    MOV AX,[BP+2] ; get 1st argument
    MOV [SP+2], AX ; and store in 1st local
    CMP [SP+2],0 ; compare to 0
    JGE Skip1 ; if >= don’t negate
    NEG [SP+2] ; otherwise negate
Skip1:
int further(int x, int y)
{
    int i,j;
    i = x;
    if (x < 0)
        i = -x;
    j = y;
    if (y < 0)
        j = -y;
    if (i < j)
        i = j;
    return i;
}

Figure 6.6: Function further(int x, int y) in C/C++

MOV AX,[BP+4] ; get 2nd argument
MOV [SP+4], AX ; and store in 2nd local
CMP [SP+4], 0 ; compare to 0
JGE Skip2 ; if >= don't negate
NEG [SP+4] ; otherwise negate
Skip2:
    MOV AX, [SP+2] ; load 1st local
    MOV BX, [SP+4] ; load 2nd local
    CMP AX, BX ; if 1st local >= 2nd local
    JGE Skip3 ; .. then don't ...
    MOV BX, AX
Skip3: ; answer now in AX
    POP BX ; restore old BX value
    ADD SP, 4 ; destroy local vars
    POP BP ; restore old BP value
    RET ; still need to pop args
Further ENDP

The diagram shows the stack frame as built by this procedure. Note particularly the two arguments passed, and the locations where the old register values are stored. Finally, there are two words of memory, corresponding to the local variables i and j. At the end of the procedure, the stack is essentially un-built in reverse order. Why didn’t we save and restore the value in AX? Because AX is the register that is being used to hold the return value, and as such will have to be clobbered.
6.4. ISSUES OF INTERFACING

How does this work in action? We’ll use a somewhat contrived example: I want to write the assembly language equivalent of further(-100,50) (which should of course return 100). In order to invoke this procedure, the calling environment must first push two integers on the stack, and then must find a way to get rid of those two values after the function returns. An easy way to do this would be the following code

X DW -100 ; first variable
Y DW 50 ; second variable
PUSH X ; er, PUSH X
PUSH Y ; um,...
CALL FURTHER ; check it out
; AX darn well better hold 100 at this point
ADD SP, 4 ; remove X, Y from stack
; AX still better hold 100

The full stack frame as built thus looks like this

<table>
<thead>
<tr>
<th>value of X (-100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of Y (50)</td>
</tr>
<tr>
<td>calling PC value</td>
</tr>
<tr>
<td>old BP register value</td>
</tr>
<tr>
<td>local variable 1</td>
</tr>
<tr>
<td>local variable 2</td>
</tr>
<tr>
<td>saved BX register value</td>
</tr>
</tbody>
</table>

Figure 6.7: Stack frame for further(X,Y)

6.4 Issues of Interfacing

Code like the previous example is actually one way in which assembly language programs can interface with the outside world; these stack frames are also how code
generated from high-level languages (with a few minor variations, so be careful) operates. So if you have a large Pascal or C++ program, you can code a small section (one or a few functions) in assembly language to make sure you have total control over the machine — to get that tiny little extra burst of speed for the animations in your game, for example.

When most people think about interfacing, though, they are usually thinking about interfacing with devices and peripherals. For example, how do I actually get data from the keyboard (where the user typed it) to the CPU, and then to the screen (where it can be seen)? The unfortunate answer is “it depends.”

It depends, in fact, on a lot of things, starting with the type of gadget you want to use, the kind of machine that you have, and the kind of operating system that you’re running. Any operating system (Windows, Linux, MacOS, FreeBSD, etc.) is actually a special kind of computer program, one that’s always running and tries to interpose itself between the other programs on the system and the device hardware. It both provides and controls access to the input and output devices — which means that if you want to do something with a device, you have to call an appropriate function (provided by the operating system) by putting the correct arguments on the stack and then doing a CALL on the right location. The details of these functions are vary from system to system. Linux works one way, using one set of functions. Microsoft Windows does the same thing, only using a different set of functions that need different arguments and different calls. So to interface with most “normal” devices, the secret is to figure out how your operating system does it, and then use the magic handshake to get the OS to do what you want for you.

There are two other major approaches to interfacing with devices. Some devices, such as the video controller, can be attached directly to the computer’s memory, and automatically update themselves whenever the appropriate memory changes. Obviously, this memory is not available for other purposes (like storing program). On the original PC (running MS-DOS), for example, “video memory” (VRAM) started at 0xA0000, which meant that programs couldn’t really use anything beyond 0x9FFFF. However, this also meant that a clever program could cause stuff to appear on the screen by putting exactly the right values in exactly the right locations past 0xA0000. This technique, called memory-mapped I/O, could be easily implemented, for example, by setting the ES segment register to 0xA000, and then using register pairs like ES:AX instead of the more normal DS:AX as the destination argument to a MOV instruction.

The other way devices is through various ports (such as the serial port, a UDP port, and so forth). This is usually called port-mapped I/O. Each of these “ports” (as well as various internal data values, such as the video color palette) can be independently addressed using a 16-bit port identification number. The OUT instruction takes two arguments, a 16-bit port and an 8-bit data value, and simply transmits that data value to that port, and thus to the device attached at that port. What the device does with that value is up to it. A IN instruction will read a byte of data from a specific port. Obviously, programming in this fashion requires very detailed knowledge both of the port numbering system and of the types and meanings of data. But with this kind of control, if you absolutely have
to, you could hook up your fishtank to a 8088’s printer port, and instead of printing, automatically control the temperature and aeration.

6.5 Chapter Review

- The Intel 8088 is the forerunner of a family of several chips that collectively comprise the best-known and best-selling CPU chips the world. Specifically, as the chip inside the original IBM Personal Computer (PC) in 1981, it rapidly became the most common chip on the business desktop and established IBM (and Microsoft) as the dominant industrial players for most of the rest of the 20th century.

- The 8088 is a classic (verily, textbook) example of CISC chip design, a complex chip with a very large and rich instruction set.

- The 8088 has eight named 16-bit “general-purpose” registers (although many of these are optimized for different special purposes), as well as a number of smaller (8-bit) registers that are physical part of the 16-bit registers, and a logically (and often physically) separate floating point unit (FPU).

- Most assembly language operations follow a two-argument format, where the operation mnemonic is followed by a destination and a source argument like this

\[
\text{OP dest,src} \quad ; \quad \text{dest} = \text{dest OP src}
\]

- Available operations include the usual set of arithmetic (although multiplication and division have special formats and use special registers), data transfers, logical operations, and several other special-purpose operational short-cuts.

- The 8088 supports a variety of addressing modes, including immediate mode, direct mode, indirect mode, and index mode.

- Floating point operations are performed in the FPU using stack-based notation and a special set of operations (most of which begin with the letter F).

- The 8088 supports normal branch instructions as well as special loop instructions using the CX register as loop counters.

- As a result of legacy support, the 8088 stores data in memory in a different format than it stores it in registers, which can be confusing to novice programmers.

- Arrays and strings are implemented using adjacent memory locations; there are also special purpose string primitive operations for common string/array operations.
• The SP and BP registers are normally used to support a standardized machine stack with standard stack frames; this helps make it easy to merge assembly language code with code written in a higher-level language.

### 6.6 Exercises

1. What does the idea of “a family of chips” mean?
2. Why does the 8088 have a fixed number of registers?
3. What’s the difference between the BX and BL registers?
4. What is the actual address corresponding to the following segment:offset pairs?
   
   (a) 0000:0000
   (b) ABCD:0000
   (c) A000:BCD0
   (d) ABCD:1234

5. What is an example of a CISC instruction not found on the JVM?
6. How is the MUL instruction different from the ADD instruction.
7. What is the difference between JA and JG?
8. What is the difference between ADD WORD PTR [4000h], 1 and ADD BYTE PTR [4000h], 1?
9. How would the 8088 handle string operations in a language (like Java) where characters are 16-bit UNICODE quantities?
10. How could so-called “global variables” be stored in a computer program for the 8088?
11. Does it matter to the 8088 whether parameters to a function are pushed onto the stack in left-to-right or right-to-left order?
Chapter 7

The PowerPC

MARK ME AGAIN, DO WE HAVE AN APPROPRIATE STOCK PHOTO?

7.1 Background

The single biggest competitor to Intel-designed chips as a CPU for desktop hardware is probably the PowerPC architecture, now used mainly in the Apple Macintosh computers. If the Pentium is the definitive example of Complex Instruction Set Computing (CISC) architecture, the PowerPC is the textbook version of Reduced Instruction Set Computing (RISC) architecture.

Historically, the PowerPC originated as a joint design project in 1991 between Apple, IBM, and Motorola. (Notice that Intel was not a player in this alliance — why should it have been, when it already had a dominant market position with its CISC-based x86 series?) RISC, which IBM had been using for embedded systems (see the later chapter) for almost twenty years, was seen as a way to get substantial performance out of relatively small (and therefore inexpensive) chips.

The key insight into RISC computing is that (as with so much in life) computer programs spend most of their time doing a few relatively common operations. For example, studies have found that around 20% of the instructions in a typical program are just load/store instructions that move data to/from the CPU from main memory. If engineers could double the speed at which only these instructions operated, then they could get about a ten percent improvement in overall system performance! So rather than spend time and effort designing hardware to do complicated tasks, design hardware to do simple tasks well and quickly. At the other extreme, adding a rarely used addressing mode (for example) exacts a performance hit on every instruction carried out, because the computer needs to inspect each instruction to see if it uses that mode, requiring either more time or expensive circuitry.

There are two particular aspects of a typical RISC chip architecture that usually speed up (and simplify) computing. First, the operations themselves are usually the
same size (for the PowerPC, all instructions are 4 bytes long — on the Pentium, instruction length can vary from 1–15). This makes it easier and faster for the CPU to do the fetch part of the fetch-execute cycle, since it doesn’t have to take the time to figure out exactly how many bytes to fetch. Similarly, “decoding” the binary pattern to determine what operation to perform can be done faster and more simply. And, finally, each of the instructions is individually simple enough that it can be done quickly (usually in the time it takes to fetch the next instruction — a single machine cycle).

The second advantage of RISC architecture has to do with groups of instructions. Not only is it usually possible to do each individual instruction more quickly, but the small number of instructions means that there usually aren’t huge numbers of near-synonymous variations in how to carry out a given (high-level) computation. This makes it much easier for code to be analyzed, for example, by an optimizing compiler. Better analysis can usually produce better (read : faster) programs, even without the individual instruction speedup.

Of course, the down side to RISC architectures like the PowerPC is that most of the individually powerful operations (for example, the Pentium’s string processing system as described in the appropriate chapter) don’t exist as individual instructions and must be implemented in software. Even many of the memory access instructions and modes are typically unavailable.

One of the design goals of the PowerPC alliance was to develop not only a useful chip, but a cross-compatible chip as well. Motorola and IBM, of course, already had well-established lines of individually designed chips (for example, IBM’s RS/6000 chip and the Motorola 68000 series). By agreeing in advance on certain principles, the PowerPC alliance could make sure their chips interoperated with each other, as well as design in advance for future technological improvements. Like the 80x86/Pentium family, the PowerPC is actually a family of chips — but unlike them, the family was designed in advance to be extensible.

This focus on flexibility has produced some odd aspects. First, unlike the Intel family, the development focus has not been exclusively on producing newer, larger, machines — from the start, low-end desktops and even embedded system controllers have been part of the target market. (The marketing advantages should be obvious : if your desktop workstation is 100% compatible with the control chips you use in your toaster oven factory, it makes it much easier to write and debug the control software for said toaster ovens. Thus, IBM could sell not only more toaster oven controller chips, but also more workstations.) At the same time, the alliance also planned for eventual expansion to a 64-bit world (now typified by the PowerPC G5), and defined the instruction set to extend to handle 64-bit quantities as part of the original (32-bit design). In addition, the PowerPC is equipped to handle a surprising amount of variation in data storage format, as will be seen below.
7.2 Organization and Architecture

Like most modern computers, there are at least two separate views of the system (formally called programming models on the PowerPC in order to support multi-tasking. The basic idea is that user-level programs get limited-privilege access to part of the computer, while certain parts of the computer are off limits except to programs (typically the operating system) running with special supervisor privilege. (The lack of such a distinct programming model is one reason the 8088 is so insecure.) This keeps user programs from interfering either with each other or with critical system resources. This discussion will focus mostly on the user model, since that’s the most relevant for day-to-day programming tasks.

7.2.1 Central Processing Unit

Most of the features of the PowerPC CPU are familiar by this time. There is a (relatively large) bank of 32 “general purpose” registers, 32 bits wide in early PowerPC models like the 601 (up to the G4), and 64 bits wide in the G5 and the 970. There are also 32 floating point registers, designated at the outset as 64 bits. Both the ALU and the FPU have their designated status/control registers (the CR and the FPSCR), and there are a (relatively few) chip-specific “special purpose registers” that you probably don’t want to use to maintain compatibility among the PowerPC family. Or at least that’s the story told to the users.

The actual underlying physical layout is a little bit more complicated, because there is some special-purpose hardware that is only available to the supervisor (by which read: “the operating system”). For example, there is a machine state register (MSR) that stores critical system-wide supervisor-level information. (Examples of such information would include whether or not the system is currently operating in user or supervisor level, perhaps responding to a supervisor-level event generated by an external peripheral. Another example of this kind of system-wide information would be the memory storage format, whether big-endian or little-endian mode, as discussed later.) Separating the status of the current computation (stored in the CR and FPSCR) from the overall machine state makes it much easier for a multitasking environment to respond to sudden changes without interrupting the current computation. A more subtle advantage is that the chip designer could duplicate the CR/FPSCR, and allow several different user level programs to run at once, each using their own register set. This process could in theory be applied to the entire user-level register space. In particular, the PowerPC G5 duplicates both the integer and floating point processing hardware, allowing fully parallel execution of up to four instructions from up to four separate processes.

Another key distinction between the user’s and supervisor’s view of the chip is in memory access. The operating system must be able to access (and indeed, control) the entire available memory of the computer, while user-level programs are usually confined to their own space for security reasons. The PowerPC has a built-in set of registers to manage memory access at the hardware level, but only the supervisor has access to these. Their function is described in the following section.
7.2.2 Memory

Memory management

At the user level, the memory of the PowerPC is simply organized (much more simply than the 8088). With a 32-bit address space, each user program has access in theory to a flat set of \(2^{32}\) different memory locations. (On the 64-bit versions of the PowerPC, of course, there are \(2^{64}\) different addresses/locations.) These define the logical memory available to the program. These can either be used as is in direct address translation (as previously defined) or as logical addresses to be converted by the memory management hardware. In addition, the PowerPC defines a third access method for speed of access to specific areas of memory.

Block address translation

In instances where a particular block of memory must be frequently (and quickly) accessed, the CPU has another set of special-purpose registers (the block address translation (BAT) registers that define special blocks of physical memory that perform a similar lookup task, but in fewer steps. The BAT registers are most often used for areas in memory representing high-speed data transfers such as graphics devices and other similar I/O devices. If a logical address corresponds to an area of memory labeled by a BAT register, then the whole virtual memory process is skipped and the corresponding physical address is read directly from the BAT register.

Cache access

The final step in memory access, irrespective of how it was arrived at, is to determine whether the desired memory location has been accessed before (recently), or more accurately, whether the data is stored in the CPU’s cache memory or not. Since on-chip memory is so much faster to access than memory stored off-chip in main memory, the PowerPC, like almost all modern chips, keeps copies of recently-used data available in a small bank of very high speed memory.

7.2.3 Devices and Peripherals

Another feature of the memory management system defined earlier is that access to I/O devices can be handled easily within the memory management system. The use of the BAT registers to access high-speed I/O devices has already been mentioned. A similar method (I/O controller interface translation) can perform a similar task for memory addresses within the virtual memory system. As mentioned above, each segment register contains other information besides the VSID. Among that information is a field detailing whether this logical address actually refers to a peripheral of some sort. If this is the case, then page translation is skipped, and instead the logical address is used to generate a sequence of instructions and addresses for dealing with the appropriate device. The overall effect of this is to
make device access on the PowerPC as easy as (and in practical terms identical to) memory access, as long as the operating system has set values in the segment registers properly.

7.3 Assembly Language

7.3.1 Arithmetic

There are two obvious differences between the assembly language of the PowerPC and the systems we’ve already looked at. First, although the PowerPC has a register bank (like the x86 and Pentium), the registers are numbered instead of named. Second, PowerPC instructions have a rather unusual three argument format. The instruction to add two numbers would be written as

```
add r3,r2,r1  # register 3 = register 2 + register 1
```

Notice that the second and third arguments (registers 1 and 2) remain unaffected in this calculation (unlike any other CPU we’ve studied) and thus can be reused later as is. Of course, by repeating the arguments, we can get more traditional style operations; for instance

```
add r2,r2,r1  # is the equivalent of x86 ADD R2,R1
```

In general, any binary operation (arithmetic, logical, or even comparison) will be expressed in this way. This three argument format, combined with the relatively large number of registers available, gives the programmer or compiler a tremendous amount of flexibility in performing computations and in storing intermediate results. Furthermore, this also allows the computer a certain amount of leeway in restructuring computations on the fly as needed. If you think about the following sequence of instructions

```
add r3,r2,r1  # (1) register 3 = register 2 + register 1
add r6,r5,r4  # (2) et cetera
add r9,r8,r7  # (3)
add r12,r11,r10  # (4)
add r15,r14,r13  # (5)
add r18,r17,r16  # (6)
```

there’s no reason that the computer would need to perform them in that particular order — or even that a sufficiently capable computer couldn’t perform them all at once.

A more interesting question is why the PowerPC does it this way. A simple answer is “because it can.” By design, the PowerPC uses a uniform and fairly large size for every instruction — 32 bits. With a bank of thirty-two registers, it takes five bits to specify any given register, so specifying three registers takes
about 15 of the 32 bits available, which still means that there could be $2^{17}$ (about 128,000) different three-argument instructions available. Obviously, this is more than any sane engineer would be expected to design — but rather than making add instructions shorter than others, the engineers found a use for the extra space in providing extra flexibility.

However, one area they did not provide extra flexibility was in memory addressing. In general, PowerPC arithmetic and logical operations cannot access memory. Formally speaking, there are only two address modes, register and immediate, as defined in the previous chapters. These are distinguished by special opcodes and mnemonics; an unmarked mnemonic operates on three registers, while a mnemonic ending with \textit{i} operates on two registers and an immediate 16 bit quantity, as follows.

\begin{verbatim}
add r3,r2,r1    # register 3 = register 2 + register 1
addi r6,r5,4   # register 6 = register5 + 4
\end{verbatim}

(In some assemblers, there is a potential for confusion here, as the programmer is allowed to refer to registers by number, without using the \texttt{r?} notation. The statement \texttt{add 2,2,1} would thus add the contents of register 1 to register 2, not add the number 1. Better yet, don’t get careless with the code you write, even if the assembler lets you.)

Of course, with only a 16-bit immediate quantity available (why?), we can’t do operations on the upper half of a 32-bit register. A few operations (\texttt{add, and, or, and xor}) have another variation with a \texttt{-is} (immediate shifted) suffix that will shift the immediate operand left by 16 bits, allowing the programmer to affect the high bits in the register directly. So the statement

\begin{verbatim}
andis r3,r3,FFFF # r3 = r3 and 0xFFFF0000
\end{verbatim}

would AND the contents of r3 with a 32-bit pattern of OxFFFF0000, thus essentially setting the lower half to zero. Similarly,

\begin{verbatim}
andis r3,r3,0000 # r3 = r3 and 0x00000000
\end{verbatim}

would zero out the entire register. Of course, the same effect could be obtained with register-to-register operations as follows

\begin{verbatim}
xor r3,r3,r3     # XOR r3 with itself (zero it out)
subf r3,r3,r3    # SUBtract r3 From itself (zero it out)
\end{verbatim}

(Of even on a RISC chip, there are often several different ways to accomplish the same thing.)

Most of the arithmetic and logical operations that you would expect are present; there are operations for addition (\texttt{add, addi}), subtraction (\texttt{subf, subfi}), negation (arithmetic inverse, \texttt{neg}), and (\texttt{and, andi}), or (\texttt{or, ori}), xor (et cetera), nand, nor, multiplication, and division. There are also an extensive and powerful set of shift
and rotate instructions. Not all of these have immediate forms, in the interests of simplifying the chip logic a bit (Reduced Instruction Set Computing, after all), but few programmers will miss the convenient of an immediate-mode nand instruction. Some assemblers also provide mnemonic aliases — for example, the not instruction is not provided by the PowerPC CPU. It can be simulated as “nor with a constant zero,” and thus doesn’t need to be provided. Instead, a smart assembler will recognize the not instruction and output something equivalent without fuss. Multiplication/division, as usual, are a little more complicated. The classic problem is that the multiplication of two 32-bit factors generally yields a 64-bit product, which won’t fit back into a 32-bit register. The PowerPC thus supports two separate instructions, mullw, mulhw, which return the low and high words, respectively, from the product. Each of these operations takes the usual three-argument format, so

\[ \text{mullw } r8, r7, r6 \]

calculates the product \( r7 \cdot r6 \), then puts the low word into register 8. Division makes the usual separation between signed and unsigned divides \( \text{divw}, \text{divwu} \). The meaning of the ‘w’ in these mnemonics will hopefully become clear in a few paragraphs.

### 7.3.2 Floating point operations

Arithmetic instructions on floating point numbers are similar, but they use the (64-bit) floating point registers instead of the normal register bank, and the mnemonics begin with an \( f \). The instruction to add two floating point numbers is thus \( \text{fadd} \). By default, all floating point instructions operate on double-precision (64-bit) quantities; but a variation with an internal \( s \) (e.g., \( \text{fadd} s \)) specifies 32-bit values instead. Furthermore, the FPU is capable of storing and loading data in integer format, so conversion from/to integers can be handled within the floating point unit.

One danger of writing code on a PowerPC is that different registers share the same numbers. For example, the instructions

\[ \text{fmul } r7, r8, r9 \quad \text{# multiply something} \]
\[ \text{add } r7, r8, r9 \quad \text{# add something} \]

appear to both operate on the same registers. This is not true. The first statement operates on floating point registers, while the second operates on general-purpose registers. In some assemblers, a raw number will also be interpreted as a register number under the correct circumstances. This means that the statements

\[ \text{add } 7, 8, 9 \quad \text{# add something} \]
\[ \text{addi } 7, 8, 9 \quad \text{# add something} \]

are substantially different. The first adds the values in registers 8 and 9 together, while the second adds the value in register 8 to the immediate integer constant 9. \textit{Caveat lector.}
**7.3.3 Comparisons and condition flags**

Most computers, by default, will update the condition register appropriately after every arithmetic or logical operation. The PowerPC is a little unusual in that regard. First, as already mentioned, there is no single condition register. More importantly, comparisons are only performed when explicitly requested (which helps support the idea of rearranging into a different order for speed). The simplest way to request a comparison is to append a period (.) to the end of most integer operations. This will cause bits in the condition register CR0 to be set according whether the arithmetic result is greater than, equal to, or less than, zero.

More generally, comparisons between two registers (or between a register and an immediate-mode constant) use a variation of the cmp instruction. This is arguably the most bizarre three-argument instruction, because it takes not only the two items to be compared (as the second and third arguments), but also the index of a condition register. For example

```plaintext
cmpw CR1, r4, r5  # compare r4 to r5
```

sets bits in the 4-bit condition register 1 to reflect the relative values of r4 and r5, as in table 7.1.

**7.3.4 Data movement**

To move data in and out of memory, there are special-purpose load and store instructions, as in the JVM. The general load instruction takes two arguments, the first being the destination register, and the second being the logical address to be loaded (perhaps after undergoing memory management as described above). As with the JVM, there are different instructions for different size chunks of data to move; the instructions to load data all begin with the letter `l`, while the next character indicates whether the data to be moved is a byte (b), halfword (h, 2 bytes), word (w, 4 bytes), or doubleword (d, 8 bytes, rather obviously only available on 64-bit version of the PowerPC, since only a 64-bit version could store such a large quantity in a register). The load instruction can also explicitly load both single precision floating point numbers (fs) or double precision (fd). Of course, not all data is loaded from memory. The li instruction will load a constant using immediate mode.
Instruction | 32-bit register result | 64-bit register result
--- | --- | ---
lbz r1, (EA) | 0x000000FF | 0x00000000000000FF
lhz r1, (EA) | 0x0000FFFF | 0x000000000000FFFF
lha r1, (EA) | 0xFFFFFFFF | 0xFFFFFFFFFFFFFFFF
lwz r1, (EA) | 0xFFFFFFFF | 0x00000000FFFFFFFF
ld r1, (EA) | not allowed | 0xFFFFFFFFFFFFFFF

Table 7.2: Some sample PowerPC load instructions

When loading quantities smaller than a word, there are two different ways of filling the rest of the register. If the instruction specifies “zero-loading” (using the letter z), the top part of the register will be set to zeros, while if the instruction specifies “algebraic” (a), the top part will be set by sign-extension. Finally, there is an update mode available, specified with a u, that will be explained in the next section.

To understand the following examples, assume that (EA), an abbreviation of “effective address,” is a memory location that holds a appropriately sized block of memory, currently storing a bit pattern of all 1s. The instruction lwz r1,(EA) would Load the Word at (EA) into register 1, and [on a 64-bit machine] zero out the rest of the register. On a 32-bit PowerPC, register 1 would hold the value 0xFFFFFFFF, while on a 64-bit machine, register 1 would hold the value 0x00000000FFFFFFFF. Table 7.2 gives some other examples of how load instructions work.

There are a few caveats. Perhaps most obviously, there is no way to “extend” a 64-bit doubleword quantity, even on a 64-bit machine. There is also no way to operate directly on doubleword quantities on a 32-bit PowerPC, or for a 32-bit CPU to algebraically extend a word quantity. The instruction lwa is therefore not a part of the 32-bit PowerPC instruction set, and the lwz simply loads a 32-bit quantity but doesn’t actually zero out anything on such a machine. Most annoyingly, the PowerPC architecture doesn’t permit algebraic extension on byte quantities, so the instruction lba doesn’t exist either. With these exceptions, the load instructions are fairly complete and reasonably understandable. For example the instruction lfdx r3, (EA) would load a single precision floating point number from memory, using the as-yet undefined “update” and “index” modes.

The operations to store data from a register into memory are similar but begin with “st,” and don’t need to deal with issues of extension. The sth r6, (EA) instruction would store the lowest 16 bits of register 6 into memory, at location EA, while the sthu, sthx, or sthux instructions would do the same thing, but using update mode, index mode, or both, respectively.

7.3.5 Branches

The final elementary aspect of assembly language programming is the control/branch instructions. As we have come to expect, the PowerPC supports both unconditional
CHAPTER 7. THE POWERPC

branches and conditional branches. The mnemonic for an unconditional branch is simply b, with a single argument for the target address. The mnemonics for conditional branches include an argument for the condition register to look at as well,

\[
\begin{align*}
&\text{bg}t \ CR0, \text{ address} \quad \# \text{ branch if bit 1 set in CR0} \\
&\text{bl}t \ CR1, \text{ address} \quad \# \text{ branch if bit 0 set in CR1} \\
&\text{be}q \ CR2, \text{ address} \quad \# \text{ branch if bit 2 set in CR2} \\
&\text{ble} \ CR3, \text{ address} \quad \# \text{ branch if bit 1 unset in CR3} \\
&\text{bne} \ CR4, \text{ address} \quad \# \text{ branch if bit 2 unset in CR4} \\
&\text{bge} \ CR5, \text{ address} \quad \# \text{ branch if bit 0 unset in CR5}
\end{align*}
\]

The PowerPC does not support separate jump-to-subroutine instructions, but there is a dedicated link register that performs similar tasks. There is also a specialized count register for use in loops. Both of these registers have special-purpose instructions for their use.

Despite the simplicity of RISC design, there are many more instructions available than space reasonably permits us to look at. In particular, the supervisor-level registers such as the BAT registers and the segment registers have their own instructions for access. But if you find yourself needing to write an operating system for a PowerPC, then there are other books you’ll need to read first.

7.4 Conical mountains revisited

Back to our old friend, the conical mountain, as a worked-out example. For simplicity of expression, this example assumes the availability of the 64-bit operations. Also for simplicity, I assume the value of \( \pi \) is available in memory somewhere, expressed as the location (PI). As given in chapter 2, the original problem statement is

What is the volume of a circular mountain 450m in diameter at the base and 150m high?

With the banks of registers available, the problem is fairly simple:

Step one is to calculate the radius, dividing the diameter (450) in half, and then to square it.

\[
\begin{align*}
&\text{li} \ r3, 450 \quad \# \text{ load diameter into r3} \\
&\text{li} \ r4, 2 \quad \# \text{ divide by 2 to get radius} \\
&\text{divw} \ r3, r3, r4 \quad \# \text{ put radius into r3} \\
&\text{mullw} \ r3, r3, r3 \quad \# \text{ square the radius}
\end{align*}
\]

Step two is to move the data (through memory) into the floating point unit.

\[
\begin{align*}
&\text{std} \ (\text{EA}), r3 \quad \# \text{ store r^2 as 64-bit integer} \\
&\text{ldf} \ r10, (\text{EA}) \quad \# \text{ load value of r^2} \\
&\text{fcfid} \ r9, r10 \quad \# \text{ convert to float}
\end{align*}
\]
Step three is to load pi and multiply by it.

```assembly
ldf r8, (PI)  # load value of pi for multiply
fmul r7, r8, r9  # multiply
fcfid r9, r10  # load value of pi for multiply
```

And finally, the height (150) is loaded and multiplied, then the final quantity will be divided by 3. For clarity, we'll first load them as a integers, then pass to the floating point processor as before.

```assembly
li r5, 150  # load height
std (EA),r5  # store as integer
ldf r6, (EA)  # load height into FPU
fcfid r6, r6  # convert to float
fmul r7, r6, r7  # multiply
li r5, 3  # load constant 3
std (EA),r5  # store as integer
ldf r6, (EA)  # load 3 into FPU
fcfid r6, r6  # convert to float
fmul r6, r6, r7  # multiply
```

Note that in this example, registers 3–5 are always used to hold integers, and thus are always general purpose registers, and registers 6–10 are always used to hold floating point numbers. This is for clarity of explanation only.

### 7.5 Memory Organization and Use

Memory organization on the PowerPC is easy once you get past the supervisor-level memory management. As discussed earlier, from the user’s point of view, the PowerPC provides a simple, flat address space. The size of the memory space is a simple function of the word size of the CPU — $2^{32}$ bytes (about 4GB) for a 32-bit CPU, and $2^{64}$ bytes for its big brothers. Of course, no computer that could affordably be built would possess 16 exabytes of physical memory, but this size allows room for future breakthroughs in memory cost. A memory address is thus just a number of appropriate size; halfwords, words, and doublewords are stored at appropriately aligned intervals within memory space. (Actually, this is a lie. Bytes are bytes. But instructions such as `ld r0, (EA)` will only work when (EA) refers to an effective address whose value is a multiple of 8. Otherwise, an intelligent assembler/compiler needs to break down the doubleword load into up to 8 partial load and shift instructions, which can slow your code down to a crawl. So don’t do it. Pretend that objects have to be stored at suitably aligned locations and you’ll feel better for it.)

One key feature of the PowerPC is direct support for both big-endian and little-endian data storage built into the CPU instruction set. This “feature” was
inherited as a legacy from IBM and Motorola, who both had extensive product lines and a large body of code that needed to be supported, but had (historically) made different choices in this regard. Data stored in the CPU is always stored in the same way, but it can be stored in memory as either normal or “byte-reversed” format.

With these complexities out of the way, though, the operation of addressing of memory is fairly simple. The PowerPC supports two basic addressing modes, indirect and indexed; the only difference is in the number of registers involved.

In indirect mode, the computer calculates the effective address as the sum of a 16-bit immediate offset and the contents of the single register specified. For example, if register 3 held the value 0x5678, then the instruction

\[ \text{lwz r4, 0x1000(r3)} \]

loads the lower 32 bits (w) of register 4 with the value stored at location 0x6678 (0x1000 + 0x5678). On a 64-bit machine, the high 32 bits are set to zero, because of the z. For most programs, the value 0x1000 would be compiler-defined and refer to a particular offset or size of a variable (as will be seen presently).

In indexed mode, the effective address is similarly calculated, but using two registers in place of a constant and a register. So the effective address is the same in

\[ \text{lwz r4, r2, r3} \]

only if the value stored in register 2 were already 0x1000. This provides a rather simple but useful two-step way of accessing memory; the second argument can be used to define a particular block of memory, for instance, the area where all the global program variables are stored, while the third argument is used to select a particular offset within that block. (This is similar in spirit, but much more flexible, than the segment registers defined in 8088.) If, for whatever reason, the block is shifted as a unit, only one register needs to be changed and the code will continue to work as designed.

For many applications, particularly applications involving arrays, it is convenient to be able to change the register values as memory access occurs (Java programmers will already be familiar with the ‘++’ and ‘--’ operators). The PowerPC provides some of this functionality through update mode, represented by a u in the mnemonic. In update mode, the calculations of effective address are performed exactly as normal, as is the memory access, but the value stored in the controlling register is then (post)updated to the effective address.

As an example, consider the effect of the following statements, presumably in the middle of a loop of some sort

\[
\begin{align*}
\text{lwzu r4, 4(r3)} & \quad \# \text{access (r3+4) in update mode} \\
\text{add r5, r5, r4} & \quad \# \text{maintain a running sum}
\end{align*}
\]
Assuming that r3 held the value 0x10000 at the start of this block, the first statement would calculate an effective address of 0x10004, and load r4 with the word (four byte) value stored at that address. So far, all is normal. After this load is complete, the value of r3 will now be updated to 0x10004, the effective address. The next time the next four-byte memory location, probably the address of the next element in the word array. This makes array processing, or more generally processing of any collection of items of similar size, very efficient.

Without special-purpose instructions for subroutines, there’s no standardized, hardware-enforced notion of a system stack or system stack pointer. Instead, the programmer (more likely, the operating system) will recruit one or more of the registers (normally r1) to serve as stack pointers and use normal register/memory operations to duplicate the push and pop operations. This is not only in keeping with the RISC philosophy (why create a special-purpose pop instruction when it can already be done?), but also allows different programs and systems to build different stack frames. Again, one can see the hand of the design by committee in this decision, as Apple, IBM, and Motorola all had extensive code bases they wished to support, each with different and incompatible views of the stack.

7.6 Performance Issues

7.6.1 Pipelining

As with the other chips we have examined, performance of the computer is a crucial aspect of success. Each new instantiation of a computer needs to run faster than the previous ones. To accomplish this, the PowerPC provides and extensive pipelined architecture (see section 5.2.4). In order to make the JVM run faster, one easy way is to simply execute it on a faster chip. In order to make a PowerPC chip run faster, well, one has to make the chip itself faster, or somehow to pack more computation into each tick of the system clock. In order to do this, the CPU has a much more complex, pipelined, fetch-execute cycle that allows it to process several different instructions at once.

One feature of the RISC architecture is that it’s designed to work well with pipelining. Remember that that the two key aspects for good pipeline performance are to keep the pipeline moving and to keep the stages approximately uniform. The RISC instruction set is specifically designed so that all instructions take about the same amount of time, and can usually be executed in a single machine cycle. So in the time it takes to perform an instruction fetch, the machine can perform an add instruction and the pipeline remains clear. This also helps explain the limited number of address modes on a PowerPC; an instruction involving adding a memory location to another memory location would require four “load” operations and a store operation besides the simple addition, and so would take four times as long (stalling the pipeline). Instead, the PowerPC forces this operation to be written as four separate instructions. Because of the pipelined operation, these instructions will still take place in the same overall time, but mesh more smoothly with the
overall computation, giving better performance.

Instruction fetch is another area where this kind of optimization can happen. On the Pentium, instructions can vary in length from a single byte up to about fifteen or so bytes. This implies that it can take up to fifteen times as long to load one instruction as another, and while a very long instruction is being loaded, the rest of the pipeline may be standing empty. By contrast, all instructions on the PowerPC are the same length, and so can be loaded in a single operation, keeping the pipeline full.

Of course, some kinds of operations (executing floating point arithmetic, for example) still take substantial time despite our best intentions. To handle this, the execution stage for floating point arithmetic is itself pipelined (for example, handing multiplication, addition, and rounding in separate stages), so that it can still handle arithmetic at a throughput of one instruction per clock cycle. In cases where some sort of delay is inevitable, there is a mechanism to stall the processor as necessary, but a good compiler can write code to minimize this as far as possible.

The other easy way to break a pipeline is by loading the wrong data, fetching from the wrong location in memory. The worst offenders in this regard are conditional branches, such as "jump if less than." Once this instruction has been encountered, the next instruction will come either from the next instruction in sequence, or else from the instruction at the target of the jump — and we may not know which. Often, in fact, we have no way of telling which, because the condition depends on the results of a computation somewhere ahead of us in the pipeline and therefore unavailable. The PowerPC tries to help this by making multiple condition registers available. If you (or the compiler) can perform the comparison early enough that the result is already available, having cleared the pipeline, when the branch instruction starts to execute, then the target of the branch can be determined and the correct instructions loaded.

As we will see with the Pentium, the PowerPC also incorporates elements of superscalar architecture. Actually, superscalar design is more generally associated with RISC architectures than with CISC chips, but a good design idea is a good design idea and likely to be widely adopted. To summarize what will be covered in more detail in section 5.2.5, this design theory incorporates the idea of multiple independent instructions being executed in parallel in independent sections of the CPU. Again, the PowerPC's simplified instruction set aids in this — arithmetic operations are separated, for example, from load/store operations or from comparison operations, and the large number of registers makes it easy for a series of instructions to use non-overlapping (and therefore parallelizable) registers.

A typical PowerPC CPU will have separate modules to handle different kinds of operations (like the distinction drawn earlier between the ALU and the FPU, only more so). A typical PowerPC will have at least one "integer unit," at least one "floating point unit," and at least one "branch unit" (that processes branch instructions), possibly a "load/store unit," and so forth. The PowerPC 603 has five execution modules, separately handling integer arithmetic, floating point arithmetic, branches, loads/stores, and system register operations. In higher-end versions, commonly used units will be physically duplicated on the chip — the PowerPC G5, as
an example, has ten separate modules on the chip, as follows

- one permute unit (doing special-purpose “permute” operations)
- one “logical” arithmetic unit
- two floating point arithmetic units
- two fixed-point (register-to-register) arithmetic units
- two load/store units
- one condition/system register unit
- one branch unit

With this set of hardware, the CPU can execute up to ten different instructions at the same time (within the same fetch-execute cycle). Within the limits of the length of the instruction queue, the first load/store instruction would be sent to the load-store unit, while the first floating point instruction would be sent to one of floating point units, and so forth. You can see that there’s no reason, in theory, that all the following instructions couldn’t be done at the same time

```
add r3, r2, r1  # just some stuff
sub r4, r2, r1  # another fixed-point operation
xor r5, r5, r5  # zero out r5 using logical unit
faddx r7, r6, r6  # add two floating point numbers
fsubx f8, r6, r6  # zero out r8 using floating point unit
b somewhere    # and branch to somewhere
```

Similarly, half of these instructions could be performed during this cycle and the other half next, or these instructions could even be done one at a time, but in any order convenient to the computer, and not necessarily in the order written.

There are a few special properties that need to hold in order to allow this sort of blatant and high-handed treatment of the program code. First, none of the instructions can depend on each other; if the second instruction changed the value in register 4 (it does) and the fifth instruction needed to use the new value, then the fifth instruction can’t happen until at least the clock cycle after the second. However, with 32 general-purpose registers available, an intelligent compiler can usually find a way to distribute the calculations among the registers to minimize this sort of dependency. Similarly, the instructions have to be of different types — with only one logical unit, logical instructions have to be done one at a time. If a part of the program consisted of thirty logical operations in a row, then that would fill the instruction queue, and it would only be able to dispatch one instruction at a time, basically slowing the computer down by a factor of up to ten. Again, an intelligent compiler can try to mix instructions to make sure a good variety of instructions are in the queue.
7.7 Chapter Review

- The PowerPC, designed by a coalition including Apple, IBM, and Motorola, is the chip inside most Apple desktop computers. It is an example of the RISC (Reduced Instruction Set Computing) approach to CPU design, with relatively few instructions that can therefore be executed very quickly.

- The PowerPC design itself is a flexible compromise among existing designs (mainly) from Motorola and IBM. As a result, the chip is actually a family of cross-compatible chips that differ substantially in architectural details. For example, the PowerPC exists in both 32- and 64-bit word size variants, and individual chips have tremendous flexibility in the way they interact with software (for example, any PowerPC can store data in both bigendian and littleendian format).

- The PowerPC CPU has a bank of 32 general purpose registers and 32 floating point registers, plus chip-specific “special purpose” registers, much more than a Pentium/x86. The chip also provides hardware support for memory management in several different modes, including direct (memory-mapped) access to the I/O peripherals.

- All PowerPC instructions are the same size (32 bits) for speed and ease of handling.

- PowerPC assembly language is written in a three-argument format. Like most RISC chips, there are relatively few addressing modes, and data movement in and out of the registers is handled by specific load/store instructions separate from the arithmetic calculations.

- The PowerPC has several different condition registers (CR) that can be independently accessed.

- In order to speed up the chip, the PowerPC executes instructions using a pipelined superscalar architecture.

7.8 Exercises

1. What is an advantage of RISC chips over CISC chips? What is a disadvantage?

2. What are some examples of the flexibility of the PowerPC design?

3. Why does the CPU have so many registers, compared to the 8088 family?

4. What is an advantage of the three-argument format used by the PowerPC arithmetic instructions?

5. What’s the difference between the and and the andi instructions?

6. What’s the difference between the andi and andi. operations?
7. Why isn’t there a standardized, hardware-supported stack frame format for the PowerPC?

8. Why is it important for pipelining that all instructions be the same size?

9. What is an instruction provided by the JVM that is not directly provided by the PowerPC? How could the PowerPC implement such an instruction?

10. How can rearranging the order of computations speed up a PowerPC program?
Chapter 8

The Intel Pentium

When's the last time you priced a computer? And what kind of computer did you price? For most people, through most of the world, the answer to the second question is probably “a Pentium.” The Pentium computer chip, manufactured by Intel, is the best-selling hardware architecture in the world. Even the competing chips, such as those built by AMD, are usually very careful to make sure that their chips operate exactly, at a bug-for-bug level, as the Pentium does. Even most computers that don’t run Windows (for example, most Linux machines) use some sort of a Pentium chip.

This means, among other things, that for the foreseeable future, if you have to write an assembly language program for a real (silicon-based) computer, it will probably be being written on and for a Pentium. In order to take full advantage of the speed of assembly language, you have to understand the chip, its instruction set, and how it’s used.

Unfortunately, this is a complicated task, both because the “Pentium” itself is a complicated chip, and because the term “Pentium” actually refers to a family of slightly different chips. Starting with the original Pentium, manufactured in the early 1990s, the Pentium has undergone continuous development, including the Pentium Pro, Pentium II, Pentium III, and leading up to the (current) Pentium 4 [P4]. Development is expected to continue, and no doubt Pentiums 5, 6, 7 will follow unless Intel decides to change the name while keeping the systems compatible (as happened between the 80486 and the Pentium.)

This successful development has made learning the architecture of the Pentium both easier and harder. Because of the tremendous success of the original Pentium (as well as the earlier x86 family that produced the Pentium), there are millions of programs out there, written for earlier chip versions, that people still want to run on
their new computers. This produces tremendous pressure for backwards compatibility, the ability of a modern computer to run programs written for older computers without a problem. So if you understand the Pentium architecture, you implicitly understand most of the P4 architecture (and much of the x86 architecture). Conversely, every new step forward adds new features, but can’t get away from the old features. This makes the Pentium almost a poster child for CISC (Complex Instruction Set Computing) architecture, since every feature ever desired is still around — and every design decision made, good and bad, is still reflected in the current design. Unlike the designers of the JVM, who were able to start with a clean slate, the Pentium designers at each stage had to start with a working system and improve incrementally.

This makes the fundamental organization of the CPU chip, for example, rather complex, like a house that has undergone addition after addition after addition. Let’s check it out.

8.2 Organization and Architecture

8.2.1 The Central Processing Unit

The logical abstract structure of the Pentium CPU is much like the previously described architecture of the 8088, only more so. In particular, there are more registers, more bus lines, more options, more instructions, and in general, more ways to do everything. As with the 8088, there is a still a set of eight general purpose named registers, but they have expanded to hold 32-bit quantities and received new names. These registers are now

| EAX  | EBX  | ECX  | EDX  |
| ESI  | EDI  | EBP  | ESP  |

In fact, the EAX register (and in similar fashion the others) is simply an extension of the previous (16-bit) AX register. Just as the AX register is divided into the AH/AL pair, so the lower 16 bits of the EAX register (called the extended AX register) is the AX register from the 8088. For this reason, old 8088 programs that use the 16-bit AX register will continue to run on a Pentium.

Similarly, the 16-bit IP register has grown into the EIP register, the extended instruction pointer, which holds the location of the next instruction to be executed. Instead of four, we now have six segment registers (CS, SS, DS, ES, FS, and GS) are used to optimize memory access to often-used areas, and finally, the EFLAGS

<table>
<thead>
<tr>
<th>EAX (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>--- not used ---</td>
</tr>
<tr>
<td>AH (8 bits)</td>
</tr>
</tbody>
</table>
8.2. ORGANIZATION AND ARCHITECTURE

register holds 32 instead of 16 flags. All of these registers except for the segment registers are 32 bits wide. This, in turn, implies that the Pentium has a 32-bit word size and that most operations deal in 32-bit quantities.

Beyond this, a major change between the 8088 and the Pentium is the creation of several different modes of operation to support multitasking. In the original 8088, any program had unfettered access to the entire register set, and by extension to the entire memory bank or to any peripherals attached — essentially, total control over the system and all its contents. This can be useful. It can also be dangerous; at the risk of indulging in old “war stories,” one of the author’s earliest professional experiences involved writing high-speed graphics programs for an original IBM-PC, and by mistake, putting graphics data into the part of system memory used by the hard drive controller. The system didn’t work quite right when it tried to boot using the “revised” parameters.

To prevent this, the Pentium can run in several different modes that control both the sort of instructions that can be executed as well as how memory addresses are interpreted. Two modes of particular interest are real mode essentially a detailed recreation of the 8088 operating environment (one program running at a time, only 1MB of memory available, and no memory protection), and protected mode, which incorporates the memory management system described later (section 8.4.1) with support for multitasking. MS-DOS, the original IBM-PC operating system, runs in real mode, while MS-Windows and Linux both run in protected mode.

8.2.2 Memory

The Pentium actually supports several different structures and ways of accessing memory, but we’ll ignore most of them here. First, they’re complicated — and secondly, the complicated ones (from the programmer’s point of view) are, for the most part, holdouts from the old 8088 architecture. If you have to write programs for a Pentium pretending to be an 8088, in real mode, they become relevant. When one writes modern programs for a modern computer, the task is much simpler. Treating memory as a flat 32-bit structure will handle almost everything necessary at the user-level.

8.2.3 Devices and Peripherals

There are very few major differences between device interfacing on the Pentium and on the earlier 8088, again due to the design for compatibility. In practical terms, of course, this has been a tremendous advantage for the computer manufacturers, since consumers can buy a new computer and continue to use their old peripherals, rather than having to buy a new printer and hard drive every time the upgrade a board.

However, as computers (and peripherals) have become more powerful, new methods of interfacing have come into common use that require the device and the I/O controller to do more of the work. For example, the direct BIOS control
typical of MS-DOS programming requires a level of access to the hardware incompatible with protected mode programming. Instead, a user level program will request access to I/O hardware through a set of operating-system defined device drivers that control (and limit) access.

8.3 Programming

8.3.1 Operations and Addressing

Much of the Pentium instruction set is inherited directly from the 8088 in name of compatibility, enough that (in theory) any program written for the 8088 will still run on a Pentium. The Pentium uses the same two argument format, and even in most cases the same mnemonics. The only major change (to the existing mnemonics) is that they have been updated to reflect the possibility of using the extended (32-bit) registers; the instruction

\[
\text{ADD EAX, EBX ; ADD 32-bit register to 32-bit register}
\]

is legal and does the obvious thing.

There are a number of new instructions created by obvious analogy to handle 32 bit quantities. For example, to the string primitives MOVSB and MOVSW (copy a string of bytes/words, defined in the previous chapter) has been added a new MOVSD that copies a string of doublewords (32-bit quantities) from one memory location to another.

8.3.2 Advanced operations

The Pentium also provides many new-from-the-ground-up operations and mnemonics, more than space really permits description of. Many of them, perhaps most, are shortcuts to perform (common) tasks in fewer machine instructions than it would take using the simpler instruction. For example, one instruction (BSWAP) swaps the end bytes in a 32-bit register (specified as an argument), a task that could be performed using basic arithmetic in a dozen or so steps. Another example is the XCHG (eXCHanGe) instruction, which swaps the source and destination arguments around. The idea behind these particular operations is that a sufficiently powerful compiler can produce highly optimized machine code to squeeze the most out of system performance.

Another example of this sort of new instruction is the new set of control instructions ENTER and LEAVE, which were included to support the semantics of functions and procedures in high-level languages such as C, C++, FORTRAN, Ada, Pascal, and so forth. As seen in section 6.3.12, local variables in such languages are normally made by creating temporary space on the system stack in a procedure-local frame. The new ENTER instructions largely duplicate the CALL semantics in transferring control to a new location while saving the old program counter on the stack, but at the same time also builds the BP/SP stack frame and reserves space.
for a set of local variables, thus replacing a half-dozen simple instructions with a single, rather complex one. It also provides some support for a nested declaration feature found in languages like Pascal and Ada (but not C, C++, or FORTRAN). The **LEAVE** instruction then undoes the stack frame creation as a single instruction (although, again rather oddly, it doesn’t actually perform the return from subroutine, so a **RET** instruction is still needed). The overall effect of these instructions saves a few bytes of program code, but (contrary to the wishes of the designers) these instructions actually take longer to execute as a single slow instruction than the group of instructions they replace.

Another example of an instruction added specifically to support high-level languages is the **BOUND** instructions, which checks that a value is between two specified upper and lower limits. In use, the value to be checked is given as the first operand, and the second operand points to two (adjacent) memory locations giving the upper and lower limits. In practical use, this allows a high-level language compiler to check that an array access can be made safely. Again, this is something that could be done using more traditional comparison/branch instructions — but it would take a half-dozen instructions and mostly likely clobber several registers. Instead, the CISC instruction set lets the system do the operation cleanly, quickly, and with minimal fuss.

The various modes of operation and memory management need their own set of instructions. Examples of such instructions include **VERR** and **VERW**, which “verify” that a particular segment can be read from or written to, respectively. Another example is the **INVD** instruction, which flushes the internal cache memory to make sure that the cache state is consistent with the state of the system’s main memory.

Finally, the continuing development of the Pentium, starting with the Pentium Pro and continuing through the PIII, P4, and P5, has continued to add new capacities — but also new instructions and features — to the basic Pentium architecture. For example, the Pentium III (1999) added a set of special 128-bit registers, each of which can hold up to four (32-bit) numbers. Among the new instructions added were (special purpose, of course) instructions to deal with these registers, including a set of SIMD (Single Instruction Multiple Data) instructions that will apply the same operation in parallel to four separate floating point numbers at once. Using these new instructions and registers, floating point performance can be increased substantially (by about four times), which means that math-heavy programs such as computer games will run substantially faster. . . or, alternatively, that a good programmer can pack four times better graphics into a game without slowing it down. Nice, huh?

At this point, you are probably wondering how you are possibly expected to remember all these instructions. The answer, thank goodness, is that for the most part, you are not expected to. The Pentium instruction set is huge, beyond the capacity of most humans who don’t work with it on a daily basis to remember. In practical terms, only the compiler needs to know about the instructions, so that it can select the appropriate specialized operation as needed.
8.3.3 Instruction formats

Unlike other computers, most notably the PowerPC, and to a lesser extent the JVM, the Pentium does not require that all its instructions take the same number of binary digits. Instead, simple operations, for example, a register-to-register ADD or a RETurn from subroutine, are stored and interpreted in only one or two bytes, making them quicker to fetch and take up less space in memory or on the hard disk. More complicated instruction may require up to fifteen bytes each.

So what kind of information would go into a “complicated” instruction? The most obvious type is immediate-mode data, as (rather obviously), 32 bits of such data will add four bytes to the length of any such instruction. Similarly, an explicit (32-bit) named address for indirect addressing adds four bytes, so the instruction

\[\text{ADD} [\text{EBX}], 13572468\text{H} \quad ; \text{add 32-bit quantity to memory}\]

takes at least four bytes beyond the simple instruction. (In the next chapter, we’ll see how this issue is handled in a typical RISC chip, basically by breaking the instruction above into a half-dozen substeps.)

Beyond this, the complexity of the possible instruction requires more data. With as many addressing modes as the Pentium has, it takes more bits (a full byte, typically, sometimes two) to define which bit patterns are to interpreted as registers, which as memory addresses, and so forth. If an instruction is to use a non-standard segment (one other than the default generated for that particular instruction type), another optional byte encodes that information including the segment register to use. The various \text{REP?} prefixes used in string operations are encoded in yet another byte. These are a few examples of the sort of complexities introduce CISC architecture. Fortunately, these complexities are relatively rare and most instructions don’t exercise them.

8.4 Memory Organization and Use

8.4.1 Memory management

The simplest organization of the Pentium’s memory is as a flat 32-bit address space. Every possible register pattern represents a possible byte-location in memory. For larger patterns (for legacy reasons, a 16-byte pattern is usually called a “word,” while an actual 32-bit word is called a “double”), one can coopt two or more adjacent byte-locations. As long as the computer can figure out how many bytes are in use, accessing memory of varying sizes is fairly simple. This simple approach is also very fast, but has a substantial security weakness in that any memory address, including memory in use by the operating system or by other programs running on the machine, is available to be messed with. (Remember my hard drive controller?) So this simple organization (sometimes called \textit{unsegmented unpaged memory}) is rarely used except for controller chips or other applications where the computer is doing one task and needs to do it really really fast.
8.5. PERFORMANCE ISSUES

Under protected mode, additional hardware is available to prevent this. The segment registers (CS, DS, et cetera) received from the 8088 provide facility for a solution of sorts. As with the 8088, each memory address in a general-purpose register will be interpreted relative to a given segment. The interpretation of the segment registers in the Pentium is a little different, though. Two of the 16 bits are interpreted as a protection level, while the other 14 are define an extension to the 32-bit address, creating an effective 46-bit (14+32) virtual or logical address. This allows the computer to address much more than the 32-bit (4 Gbyte) address space and also to mark large areas of memory as inaccessible to a particular program, thus protecting private data.

However, these 46-bit addresses may still need to be converted into physical addresses to be accessed over the memory bus (which has only 32 lines, and hence takes 32-bit addresses). The task of converting from these virtual addresses to 32-bit physical addresses is handled by the paging hardware. The Pentium contains a directory of page table entries that act as a translation system for this conversion task. Use of the segmentation and/or paging hardware can be enabled or disabled independently, allowing the user of the system (or, more likely, the writer of the operating system) to tune performance to a particular application. From the user’s point of view, much of the complexity is handled by the hardware, so all you have to do is write your program without worrying about it.

8.5 Performance Issues

8.5.1 Pipelining

A standard modern technique for getting more performance out of a chip is pipelining. The Pentium, although less well-suited in instruction set to take advantage of pipelining than some other chips, nevertheless uses this technique extensively.

Even before the Pentium was developed, the Intel 80486 had already incorporated a five-stage pipeline for instruction execution. The five stages are

- Fetch — instructions are fetched to fill one of two prefetch buffers. These buffers store up to 16 bytes (128 bits) each, and operate independently of the rest of the pipeline.
- Decode stage 1 — Instruction decode is performed in two separate stages, with the first stage acting as a type of preanalysis to determine what sort of information must be extracted (and from where) in the full instruction. Specifically, the D1 stage extracts preliminary information about the opcode and the address mode, but not necessarily the full details of the address(es) involved.
- Decode stage 2 — completes the task of decoding the instruction, including identification of displacement or immediate data, and generates control signals for the ALU.
- Execute — executes the instruction
CHAPTER 8. THE INTEL PENTIUM

• Write back — updates registers, status flags, and cache memory as appropriate given the results of the immediately previous stage.

This pipeline is more complicated than others we have seen (such as the PowerPC), in part because of the complexity of the instruction set it needs to process. Remember that Pentium (and even 486) instructions can vary in length from 1 to 15 bytes. This is part of the reason that the instruction prefetch buffers need to be so large, so they can hold the next instruction completely. For similar reasons, it can take a long time to decode complicated instructions — sometimes as long or even longer than it takes to execute simple ones. The number and complexity of the addressing modes is a major factor in this, since they can add several additional layers of interpretation to an otherwise simple logical or arithmetic operations. For this reason, there are two separate decode stages to keep the pipeline flowing smoothly and quickly. The 80486 can perform most operations that do not involve memory loads at a rate of close to one operation per machine cycle. Even so, indirect addresses (where the value in a register is used as a memory address) and branches can still slow the pipeline down.

This pipelining is considerably more extensive on the Pentium and later versions. Not only are there several pipelines (reflecting the superscalar architecture), but each pipeline may have more stages. The original Pentium had two pipelines, each of the five stages listed above. The Pentium II increased the number of pipelines again, and increased the number of stages to 12, including (for example) a special stage just to determine the length of each instruction. The PIII uses 14-stage pipelines and the P4 uses 24-stage. So despite not having an instruction set designed for efficient pipelining, the Pentium has taken to them with a vengeance.

8.5.2 Parallel operations

The Pentium incorporates two other substantial architectural features to allow true parallel operations — performing two instructions at the same time within the CPU. Starting with the Pentium II, the instruction set features a collection of MMX instructions, specifically for multimedia applications such as high-speed graphics (think: games) or sound (again think: games). These implement a sort of SIMD (single instruction, multiple data) parallelism, where the same operation is performed on several independent pieces of data.

A typical multimedia application involves processing large arrays of relatively small numeric data types (for example, pixels in a screen image) and performing the same calculation on every data type fast enough to present the image without noticeable flicker. To support this, the Pentium II defines a set of MMX registers, each 64 bits long, holding either eight bytes, four words, two doublewords, or less frequently an eight-byte quadword. A sample MMX instruction defines a single operation to be performed on all elements of the register simultaneously. For example, the PADD (Parallel ADD Bytes) instruction performs eight additions on the eight separate bytes; the variant PADDW would perform four simultaneous additions on four words. For a simple operations like displaying a simple byte-oriented image,
this allows data to be processed up to eight times as fast as it would using ordinary 8-bit operations.

8.5.3 Superscalar architecture

As discussed before (section 5.2.5), the other major parallel instruction technique involves duplication of pipeline stages or even entire pipelines. One of the inherent difficulties behind pipelining is keeping the stages balanced; if it takes substantially longer to execute a particular instruction than it did to fetch it, the stages behind it may back up. The original Pentium used the 80486 five-stage pipeline, but duplicated key stages to create two pipelines, called the U and V pipelines. Instructions can alternate between the two pipelines, or even execute two at a time if both pipelines are clear. More generally, a superscalar architecture will have multiple pipelines or multiple execution units within specific stages of the pipelines to handle different kinds of operations (like the distinction drawn earlier between the ALU and the FPU, only more so).

Later models of the Pentium, starting with the Pentium II, have extended this further. In particular, the Pentium II’s decode 1 (ID1) stage is more-or-less replicated three times. In theory, this implies that the ID1 stage could take up to three times as long as other stages without slowing the overall pipeline up significantly. In practice, the situation is a little more complex. The first instruction decoder can handle instructions of moderate complexity, while the second and third decoders can only handle very simple instructions. The Pentium II hardware therefore includes a special instruction-fetch stage that will reorder instructions to align them with the decoder, so if any one of the next three instructions in the instruction queue is complicated, it will be automatically placed into decoder #1. Since these instructions are rather rare, it wasn’t considered necessary to add this (expensive) capacity to the second/third decoder. Finally, for the really complex instructions, there’s a fourth decoder, the microcode instruction sequencer (MIS).

There’s also a duplication of hardware units at the execution stage. A special stage of the pipeline, the reorder buffer (ROB) takes instructions and dispatches them in groups of up to five among various execution units. These units handle, for instance, load instructions, store instructions, integer operations (divided into “simple” and “complex”), floating point instructions (similarly divided), and several different kinds of MMX instructions.

8.6 RISC vs. CISC revisited

Having spent several chapters dwelling on the differences between RISC and CISC architecture, you’ve probably notice that the practical differences are few — both the (RISC) PowerPC and the (CISC) Pentium use many of the same techniques to get the most possible performance out of the machine. Part of the reason for this is that the stakes of competition are high enough that both camps are willing to figure out how to use good ideas from their competition. A more significant part of
the reason is the lines themselves are blurring as technology improves. Moore’s law has dictated that transistor density, and hence the amount of circuitry that can be put on a reasonable-sized micro-chip, has been getting larger and larger at a very fast rate. This, in turn, has meant that even “reduced” instruction set chips can have enough circuitry to include useful instructions, even if complicated.

At the same time, the hardware has been getting fast enough to allow extremely small-scale software emulation of traditional hardware functions. This approach, called microprogramming, involves the creation of a CPU within the CPU with its own tiny microinstruction set. A complicated machine instruction — for example, the 8088/Pentium MOVSB instruction that moves a string of bytes from one location to another — could be implemented at the microinstruction level by a sequence of individual microinstructions to move one byte each. The macroinstruction would be translated into a possibly large set of microinstructions which are executed one at a time from a microinstruction buffer invisible to the original programmer.

This kind of translation is the job of the various ID1 decoders in the Pentium superscalar architecture. Specifically, the second and third decoders are only capable of translating instructions that translate to a single microinstruction; the first can handle more complicated instructions that produce up to four microinstructions. For even more complicated instructions, the MIS acts as a lookup table, storing up to several hundred microinstructions for the really complicated parts of the Pentium instruction set.

At a more philosophical level, the true irony is that the Pentium, as currently implemented, is a RISC chip. The individual microoperations at the core of the various execution units are exactly the kind of small-scale fast operations at the heart of a strict RISC design. The major weakness of RISC, that it requires a sophisticated compiler to produce the right set of instructions, is handled instead by a sophisticated instruction compiler/interpreter, using the CISC instruction set as almost an intermediate expression stage. Programs written in a high-level language are compiled to the CISC instructions of the executable file, and then each instruction, when executed, is re-converted into the RISC-like microinstructions.

8.7 Chapter Review

- The Intel Pentium is actually a family of several chips that collectively comprise the best-known and best-selling CPU chips the world. Partly as a result of effective marketing, partly as a result of the success of previous similar chips such as the x86 family, the Pentium (and third-party Pentium clones, made by companies such as Advanced Micro Devices) have been established as the CPU chip in of choice for Windows and Linux-based computers.
- Partly as a result of the CISC design, and partly as a result of legacy improvements and backwards compatibility, the Pentium is a very complex chip with a very large instruction set.
- Available operations include the usual set of arithmetic (although multiplication and division have special formats and use special registers), data
transfers, logical operations, and several other special-purpose operational short-cuts. The entire 8088 instruction set is available in the interests of backwards compatibility.

- The Pentium includes a huge number of special-purpose instructions designed specifically to support specific types of operations. For example, the `ENTER/LEAVE` instructions support the sort of programs typically resulting from compiling high-level languages such as Pascal and Ada.

- The Pentium II adds a set of multimedia (MMX) instructions that provide instruction-level parallelism for simple arithmetic operations. The MMX instruction set allows SIMD (Single Instruction Multiple Data) operations, for instance, doing eight separate and independent additions or logical operations at once instead of one at a time.

- The Pentium also incorporates extensive pipelining and superscalar architecture to allow genuine MIMD (Multiple Instruction Multiple Data) parallelism.

- The actual implementation of the Pentium involves a RISC core where individual CISC instructions are implemented in a RISC-like microinstruction set.

### 8.8 Exercises

1. What are four major changes between the 8088 and the Pentium?
2. What are four examples of new instructions that the Pentium has that the 8088 does not?
3. Why are there two decode stages in the Pentium pipeline, but only one execute stage?
4. How could SIMD parallelism smooth cursor/pointer movement on a computer screen?
5. What purpose is served by reordering the instruction fetches in the Pentium II?
Chapter 9

Microcontrollers : The Atmel AVR

9.1 Background

A microcontroller is the kind of computer used for small-scale control operations inside devices that one doesn’t usually think of as being computers. The classic examples of such devices include traffic lights, toasters, thermostats, and elevators, but a better, more detailed, type would be the microcontrollers that are now installed in modern automobiles. Anti-lock braking, for instance, is only possible because of a microcontroller that observes the braking systems and cuts in when it sees the the wheels lock (and the car thus about to skid). Other microcontrollers will look for opportunities to fire the airbags, adjust fuel mixtures to reduce emissions, and so forth. According to the Motorola company, a microcontroller manufacturer, even a low-end 2002 model passenger vehicle has fifteen or so microcontrollers in it — a luxury car, with much better entertainment and safety features, usually had over a hundred.
There is no formal accepted definition of a microcontroller, but they usually share three main characteristics. First, they are usually found in so-called embedded systems, running specialized single purpose code as part of a larger system, instead of being general-purpose user-programmable computers. Second, they tend to be smaller, less-capable computers (the Zilog Z8 Encore! microcontroller uses 8-bit words, runs at 20MHz, can address only 64K of memory, and retails at about $4. Compare this to a Pentium 4 processor which can easily cost $250 or more for the bare processor — without memory and therefore independently useless). Third, as hinted at, microcontrollers are usually single-chip gadgets, in that they have their memory and most of their peripheral interfaces located on the same physical chip. This isn’t as unusual (now) as it seems, since almost all modern computer architectures have cache memory located on the CPU chip. The real implication is that the memory available to a microcontroller is by definition all cache memory, and therefore small but fast.

In this chapter, we’ll look in detail at the AVR microcontroller, manufactured by the Atmel Corporation. Of course, the AVR isn’t by any stretch of the imagination the only such computer out there; microcontrollers are commodity items, sold by the billions. The field is fiercely competitive, and other companies that make and sell microcontrollers include Microchip, Intel, AMD, Motorola, Zilog, Toshiba, Hitachi, and General Instrumentation. However, the AVR (or more accurately the AVR family, since there are several variant models of the basic AVR design) is fairly typical in its capacities, but differs in interesting ways from more mainstream chips such as the Pentium or PowerPC (made by Intel and Apple/IBM/Motorola, respectively).

9.2 Organization and Architecture

9.2.1 Central Processing Unit

The Atmel AVR uses RISC design principles, in the interests of both speed and simplicity. There are relatively few instructions, making those that do exist both short (two bytes, which can be compared to the PowerPC’s 4 bytes, or the Pentium’s “up to 15”) and fast to execute. Each instruction is constrained to be a standardized length of 16 bits, including the necessary arguments. The instruction set is tuned specifically for the usual needs of a microcontroller, including a (relatively) large number of bit instructions for the manipulation of individual electrical signals. Despite this, there are still only about 130 different instructions (fewer than there are on the JVM). This isn’t even the smallest instruction set out there, by a long chalk. Microchip makes a relatively common, tiny chip — used in toasters, as it happens — with fewer than 35 instructions.

The Atmel AVR contains 32 general purpose registers (numbered from R0 to R31), as well as 64 so-called I/O registers. Each of these registers is 8 bits wide, enough for a single byte or a number from 0..255 (or -128..127). As with the JVM, (some) registers can be used in pairs to permit larger numbers to be accessed.
Unusually (at least as compared with the computers we’ve already studied), these registers are physically part of memory instead of being separate from the memory chip.

For all practical purposes, the AVR provides no support for floating point numbers; the ALU will only do operations on integer types, and on very small integers at that.

Operationally, the AVR is very similar to the computers we already know, having a special-purpose instruction register, program counter, stack pointer, and so forth.

9.2.2 Memory

As a microcontroller, the amount of memory on an AVR is quite limited. Unusually, the memory itself is divided into three separate memory banks that differ not only physically, but also in their sizes and capacities. The exact memory capacity differs from model to model, but the capacities of the AT90S2313 are a good representation. This machine, like most microcontrollers, is an example of the so-called Harvard architecture design, where different physical storage banks (and different data buses) are available for transferring machine instructions and data. By using two separate paths, they can each be independently tuned to maximize performance. Furthermore, the computer can load instructions and data at the same time (over the two buses), an effective doubling of speed. But for general purpose computers, this would also ordinarily require two separate cache memories (one for instructions, one for data), which in turn would reduce the amount of cache available to each, and cut seriously into cache performance. On a microcontroller, where the memory is all cache anyway, it doesn’t have as much of an impact.

On the AVR in particular, there are three separate banks of memory: a read-only bank used for program code (since program code shouldn’t change during the execution of a program), a read/write bank of high speed memory for program variables, and a third bank used for long-term storage of program data that must survive a power outage (for example, logging information or configuration information). Unlike conventional architectures, where all memory is more or less created equal and a single virtual address space suffices to access anything, each of these three banks is independently structured and accessed with its own address space and instructions.

As discussed earlier, there is a fundamental distinction drawn between ROM (read-only memory), which can be read from (but not written to) and RAM (random access memory) which can be both read from and written to. This distinction is more fundamental in theory than it has become in practice, with the development of various types of memory that require extensive equipment to scribble upon, but are still writeable (in an abstract, Platonist, and expensive sense). In practical terms, the modern definition of the ROM/RAM distinction is a difference in use: whether or not the CPU is intended to be able to write to the memory (bank). On the AVR, the first bank of memory is made of FLASH ROM. Although FLASH memory is in theory read/write (it’s the usual type of memory used in keychain
drives), there are no circuits or instructions on the AVR to write to it. From the
AVR’s perspective, the FLASH memory provides (on the AT90S2313, 2048 bytes
of) non-volatile memory, memory that does not lose information when the power is
removed. This memory, for all practical purposes read-only, is organized into 16-
bit words and provides storage for the executable machine code. In particular, the
value stored in the program counter is used as a word address into this particular
block of memory. The AVR chip itself cannot effect any changes to the ROM, and
so it can only be programmed (or reprogrammed) by someone with appropriate
external equipment. (And, in all honesty, the equipment isn’t that expensive.)1

1SIDEBAR : KINDS OF MEMORY. A rose may be a rose may be a rose, but memory
isn’t just memory. We’ve already discussed several different kinds of memory, for example,
the difference between RAM and ROM. In very broad terms, engineers will talk about
many different kinds of memory, each of which has its appropriate use.

• RAM : Random Access Memory. This is the most common thing that people think
of when they hear about memory; binary values are stored as electrical signals.
Each set of signals (usually sets are word or byte sized, but they can be individual
bits) can be addressed independently, in “random” order, hence the name. RAM
is a kind of volatile memory, in that the chips require electrical power in order to
hold the signal; if, for some reason, the power goes out in your building, all the
information you have in RAM will disappear.

There are, broadly speaking, two sub-types of RAM: Dynamic RAM (DRAM) and
Static RAM (SRAM). Most of the memory you buy or see is DRAM, as it’s cheaper
and more compact. Each “bit” is simply stored as a charge on a single electrical
 capacitor (plus an associated transistor), a device that will hold energy for a short
period of time. The problem with DRAM is that the memory trace actually decays
inside the circuit, even if the chip itself has power. SRAM will remember values
as long as the computer has power, without needing the refreshes. In practical
terms, this means that the computer must periodically generate “refresh” signals
to recreate the DRAM memory patterns. (Periodically, in this term, means a few
thousand times a second. This is still a relatively long period of time for a 1 GHz
processor.)

By contrast, SRAM is built to be self-reinforcing, like the flip-flop circuits in
appendix A. Each memory bit will typically require on the order of six to ten tran-
sistors to create, which means that a byte of SRAM memory takes up to ten times
as much space on the chip and costs up to ten times as much money. On the
other hand, because SRAM requires no refresh cycles, it can be accessed faster.
Where DRAM is usually used for main system memory (where the added cost of
several hundred megabytes adds up), SRAM is usually used in small, speed-critical
memory applications such as the computer’s cache memory. (For this reason, the
Atmel microcontroller uses exclusively SRAM for its writable memory; with less
than a thousand bytes, speed dominates over price.) Despite the fact that SRAM
is self-refreshing, the transistors still need power in order to work, and SRAM will
lose its signal if power is cut off for too long (milliseconds). Thus, SRAM is still
considered to be volatile memory.

• ROM : Read Only Memory. Where RAM can be both read from and written to,
ROM chips cannot be written to. A better description is that writing to ROM
chips, when possible at all, requires special operations and sometimes even special equipment. However, the advantage of ROM is that it's a form of non-volatile memory, meaning that if power is removed from the chip, the data still remains. This makes it ideal for storing, for example, photographs taken by a digital camera.

The simplest and oldest version of ROM is structured similarly to DRAM, except instead of using a capacitor, each memory cell contains a diode, which is programmed by the chip manufacturer to pass current if and only if that memory cell is supposed to hold a 1. Since diodes do not need power and don’t usually degrade, this means that the pattern built into the chip will remain unchanged forever, unless you jump on it or something. It also means that the manufacturer needed to know exactly what memory values to use, and Heaven help them if they change their minds, or make a mistake. (The infamous 1993 Pentium fdiv bug is an example of what can go wrong with a ROM.)

ROMs can be built extremely cheaply in quantity — pennies per chip. But what happens if you only need 25 chips? It’s probably not worth buying an entire chip factory. Instead, use PROM (Programmable ROM) chips. Like ROM chips, these are manufactured with a static electrical connection at every memory cell (instead of an active element like a capacitor or transistor). In a PROM, these connections are actually fuses, that can be broken by the application of a high enough voltage. This process, for obvious reasons, is called “burning” a PROM. Once the PROM is burned, the electrical connections that remain (or are now broken) are still fixed, unchanging, and eternal. But because it’s impossible to un-break a fuse, a PROM can only be burned once.

A final type of ROM avoids this issue. EPROM (Erasable Programmable ROM) chips use advanced quantum physics to create a semiconductor-based reusable fuse at each memory element. To create an electrical connection, the memory location is subjected to the same sort of overvoltage we’ve seen with the PROM. However, applying several minutes of ultraviolet light will cause the “fuse” to reset. This will erase the entire chip, allow it to be reprogrammed and reused.

**Hybrid memory.** In an effort to get the best of both words, manufacturers have started making so-called hybrid memory. Hybrid memory is supposed to be field-reprogrammable, while still having the storage advantages of ROM. A variation on EPROM technology, for instance, EEPROM (Electronically Erasable Programmable ROM) chips use a localized electrical field to “erase” each memory cell, instead of erasing the entire chip as a whole. Because this operation is performed electronically, it doesn’t require the cabinet-sized UV chamber and can be part of the distributed system.

On the other hand, writing to EEPROMs takes a long time, because the cell must be exposed to the magnetic field to erase it, which takes several milliseconds. In theory, EEPROMs provide the same functionality as RAM, since each individual cell can be written, read, and re-written, but the timing issues (and cost issues) make it impractical. Another major issue with EEPROMs is that they are typically only capable of a limited number of read/write cycles.

**Flash memory** is one approach to speeding up the process of writing to EEPROMs. The basic idea is simple; instead of erasing each bit as an individual, the electrical field (for erasing) will be applied to large “blocks” on the chip. It takes more or less the same time to erase a single block as it does a single bit, but when data must be
By contrast, the second bank of memory is specifically both readable and writable. This data memory is composed of SRAM (static random access memory). As discussed in the sidebar, the primary difference between SRAM and DRAM (dynamic random access memory) is that dynamic RAM requires periodic “refresh” signals from the computer circuitry to retain its value. The AT90S2313 has fewer than 256 bytes of SRAM.

Since the memory is composed of more or less the fastest data storage circuitry available, there is no need for a separate high-speed register bank as in more typical computers. The AVR data memory is organized as a sequence of 8-bit bytes, and divided into three sub-banks. The first 32 bytes/words (0x00..0x1F, or $00..$1F, to use Atmel’s notation) are used as the general purpose registers R0..R31. The next 64 words (0x20..0x5F) are used as the 64 I/O registers, while the rest of the SRAM provides a bank of general-purpose memory storage for program variables and/or stack frames as necessary. These memory storage locations are addressed using addresses from 0x60 on up to the amount of memory built into the chip (to 0xDF on our “standard” AT90S2313).

Finally, the third bank uses yet another kind of memory, EEPROM (electronically erasable programmable read-only memory). Like the FLASH ROM, the EEPROM bank is non-volatile, so the data persists after the power goes off. Like the SRAM, the EEPROM is electronically programmable, so the AVR CPU can write data to the EEPROM for persistent storage (although it takes a long time, on the order of 4ms). Unlike the SRAM, though, there is a limited number of times that data can be re-written (about 100,000 times, although technology is always improving). This is a physical problem related to the construction of the memory, but should be borne in mind when writing programs for the AVR. Writing a piece of data to the EEPROM bank just once per second will still hit the 100,000 write stored in mass quantities (say, on a pen drive like my beloved Jump Drive 2.0 Pro, manufactured by Lexar Media and the fourth thing I would save from a fire, right after my cats), the time to transfer a block of data to memory dominates over the time to erase the sector where it will be placed. Flash memory is widely used, not only in pen drives, but also in digital camera memory, smart cards, memory cards for game consoles, and solid-state disks in PCIMCIA cards.

Another variation on hybrid memory is NVRAM (Non-Volatile RAM), which is really just SRAM with an attached battery. In the event of power loss, the battery is capable of providing the trickle of power necessary to keep memory alive in an SRAM chip. Of course, the cost of the battery makes this kind of memory substantially more expensive than simple SRAM.

- SAM (Sequential Access Memory): No discussion of memory types would really be complete without mentioning SAM. SAM is intuitively familiar to anyone who has tried to find a particular scene in a videotape. Unlike RAM, SAM can only be accessed in a particular (sequential) order, which can make it slow and awkward to use in small pieces of data. Most kinds of secondary storage — CD-ROMs, DVDs, hard drives, even magnetic tapes — are “really” SAM devices, although they usually try to provide block-level random access.
limit in a little over a day. However, there’s no limit to the number of times that
the computer can safely read from the EEPROM. This makes the EEPROM ideal
for storing field-modifiable data that needs to be kept, but that doesn’t change
very often, such as setup/configuration information or infrequent data logging (say,
once/hour, which would give you about ten years lifetime). On our standard ex-
ample, the EEPROM bank is organized as a set of 128 bytes.

Each of these memory banks has its own individual address space, so the number
zero (0x00) could refer not only to the actual number zero, but to the lowest two
bytes of the FLASH memory, the lowest (single) byte of the EEPROM, or the lowest
byte of the SRAM (also known as R0). As with all assembly language programs, the
key to resolving this ambiguity is context — values stored in the program counter
refer to the FLASH memory, while normal register values (when read as addresses)
refer to locations in the SRAM. The EEPROM is accessed through special purpose
hardware that in practical terms can be treated as a peripheral.

9.2.3 Devices and Peripherals

The AVR implements a simple kind of memory-mapped I/O. It is not designed to
be used in graphics-heavy environments, where one might be expected to display
pictures consisting of millions of bytes twenty to thirty times per second. Instead,
it is expected to drive a chip where the output goes through a few pins that are
physically (and electrically) attached to the CPU circuitry. Specifically, these
pins are addressable through specific, defined, locations in the I/O memory bank.
Writing to I/O memory location 0x18 (SRAM memory location 0x38), for example,
is defined as writing to the “Port B Data Register” (PORTB), which in turn is
equivalent to setting an electrical signal at the eight output pins corresponding to
Port B. The chip can generate enough power to turn on a simple LED, or else to
throw an electrical switch to connect a stronger power source to a more power-
hungry device. Similarly, reading the various bits from the register will cause the
CPU to detect the voltage level currently present at the pins, perhaps to detect
if a photocell (a so-called “electric eye”) is reacting to light, or to determine the
current temperature reading from an external sensor.

The AVR usually provides several bi-directional data ports that can be indi-
vidually defined (on a per-pin basis) to be input or output devices. It also provides
on-chip timer circuits that can be used to measure the passage of time and/or let
the chip take action on a regular, recurring basis (such as changing a stop light
every thirty seconds or taking an engine temperature reading every millisecond).
Depending on the exact model, there may be other built-in peripherals such as
a UART (Universal Asynchronous Receiver and Transmitter) for large scale data
transmission/reception, an analog comparator to compare two analog sensor read-
ings, and so forth. Unlike larger computers, many of the actual output devices (the
pins) are shared between several different output devices; the pins that are used for
the UART are the same physical connections used for the B data port. Without
this sort of overlap, the chip would be physically much more difficult to use (having
hundreds of pins needing individual connections), but the overlap itself means that
there are several device sets that simply cannot be used together. If you are using
the B port, you can’t use the UART at the same time.

The I/O memory is also where information regarding the current state of the
CPU itself is stored. For example, the AVR status register (SREG) is located at
I/O location 0x3F (SRAM location 0x5F), and contains bits describing the current
CPU status (such as whether or not the most recent computation resulted in a zero
and/or a negative number). The stack pointer is stored at location 0x3D (0x5D)
and defines the location (in SRAM) of the active stack location. Because these
registers are treated programmatically as memory locations, interacting with I/O
peripherals is as simple as storing and reading memory locations.

9.3 Assembly Language

Like most chips, the registers on the AVR are not structured in any particular
fashion. Assembly language instructions are thus written in a two-argument format,
where the destination operand comes before the source operand. Thus

```
ADD R0, R1 ; R0 = R0 + R1
```

will cause the value of R1 to be added to the value of R0, storing the result in
R0, and setting the value of various SREG bits to reflect the outcome of the result.
Although superficially this looks very much like an assembly language instruction
for a Pentium or PowerPC, it (of course) corresponds to a different machine lan-
guage value specific to the Atmel AVR.

The AVR provides most of the normal set of arithmetic and logical operations
that we have come to expect: ADD, SUB, MUL (unsigned multiply), MULS (signed
multiply), INC, DEC, AND, OR, COM (bit complement, i.e. NOT), NEG (two’s comple-
ment, i.e. negate), EOR (exclusive or), and TST (which tests a register value and
sets the flags appropriately if the value is zero or negative). Perhaps oddly to our
view, the slow and expensive division operation is not available. Also not available
is the modulus operator, or any kind of floating point support.

In the interests of speeding up the sort of computations typically done by a
microcontroller, there are a number of I variants (SUBI, ORI, ANDI, etc.) that
will take an immediate mode constant and perform that operation to the register.
There are also several instructions to perform operations on individual bits: for
example, SBR (set bit(s) in register) or CBI (clear bit in I/O register) will set/clear
individual bits in a general purpose or I/O register.

The various control operations are also extensive. In addition to a wide range
of branch/jump instructions (unconditionally: JMP, conditionally: BR??, where
?? refers to different flags and flags combinations in the SREG register, and jump
to subroutine: CALL), there are a few new operations. The SB?? operation – the
first ? is an R (general-purpose register) or an I (I/O register), the second is a C
(clear) or S (set), hence SBRIC = Skip if Bit in I/O register is Clear — performs a very
limited branch that just skips the single next instruction if the bit in the appropriate
register is set/clear. The AVR also supports indirect jumps (unconditional: IJMP,
9.4 Memory Organization and Use

Normal data transfer instructions manipulate the SRAM memory bank by default. Since registers and I/O registers are for all practical purposes part of this same bank, there's no different between writing to a register and writing to a simple SRAM byte. However, the arithmetic operations defined in the previous section only work with the general purpose registers (R0..R31), so one must still be prepared to move data around within this bank. The normal instructions for this purpose are the LDS (Load Direct from SRAM) instruction, which takes a register as its first argument and a memory location as its second, or the corresponding SDS (Store Direct to SRAM) instruction, which reverses the arguments and process. The AVR also provides three specific indirect address registers X, Y, and Z, that can be used for indirect addressing. These are the last six general-purpose registers, taken in pairs (so the X register is really the R26:R27 pair) and can be used to hold (variable) addresses in memory. Using these registers and the LD (Load inDirect) instruction, the following code

```
CLR R26 ; Clear R26 (set it to 0)
LDI R27, 0x5F ; Load Immediate (constant value) 5F into R27
LD R0, X ; Move (X) [= value in location 5F] into R0
```

will have the effect of copying memory location 0x005F into register 0. It first sets the two halves of the X register individually to 0x00 and 0x5F, then uses this as an index register. (We have previously seen that 0x005F is actually the SREG register). An easier way of doing this would be to use the IN instruction, which reads the value from the specified I/O register (as follows)

```
IN R0, 0x3F ; Copy SREG into R0
```

Note that although SREG is in memory location 0x5F, it is only in I/O port number 3F.

Access to the FLASH memory (which can of course only be read from, and not written to), is performed indirectly using the LPM (Load from Program Memory) instruction. The value stored in the Z register is used as a memory address inside the program (FLASH) memory area, and the appropriate value is copied to the R0 register.

Accessing EEPROM is more difficult, largely for physics and electronics reasons. Although an EEPROM bank is in theory read/write, writing effects actual physical change to the memory bank. As such, it takes a long time to complete, but also can require substantial preparations (such as powering up “charge pumps” to provide the necessary energy for the changes) that need to take place before the write can be taken. On the AVR, the designers opted for a memory access scheme that looks almost like accessing a device.
The AVR defines three I/O registers (as part of the I/O register bank, in the SRAM), the EEAR (EEPROM Address Register), EEDR (EEPROM Data Register), and the EECR (EEPROM Control Register). The EEAR contains a bit pattern corresponding to the address of interest (a value between 0..127 on our standard example, so the high bit should always be a zero). The EEDR contains either the data to be written, or the data that has just been read, in either case using the data in the EEAR as the destination.

The EECR contains three control bits that individually enable read or write access to the EEPROM memory bank. In particular, bit 0 (the least significant bit) is of the EECR is defined to be the EERE (EEPROM Read Enable) bit. To read from a given location in the EEPROM, the programmer should take these steps:

- load the byte address of interest into the EEAR
- set the EERE set to 1, allowing the read to proceed
- after the read operation completes, the relevant data can be found in the EEDR.

The steps to write are a little (not much!) more complex, because there are actually two enabling bits that need to be set. Bit 2 is defined to be the EEMWE (EEPROM Master Write Enable) bit; when it it set to 1, the CPU is “enabled” to write to the EEPROM. However, this doesn’t actually do any writing — this simply performs the preparations for writing. The actual writing is performed by setting bit 1 (the EEWE/EEPROM Write Enable) bit to 1 after the EEMWE has also been set to 1. The EEMWE will automatically be returned to 0 after a short period of time (about four instructions). This two-phase commit process helps keep the computer from accidentally scribbling onto the EEPROM (and damaging important data) in the event of an unexpected program bug.

In order to write to the EEPROM, the programmer should:

- load the byte address of interest into the EEAR
- load the new data into the EEDR
- set the EEMWE to 1, enabling writing to the EEPROM bank
- (within four clock cycles) set the EEWE to 1, allowing the write actually to happen

The actual writing, however, can be extremely slow, taking as much as 4ms. On a chip running at 10MHz, this is enough time to perform 40,000(!) other operations. For this reason, it’s a good idea to make sure that the EEPROM isn’t in the middle of writing (i.e. waiting for the EEWE bit to go to 0 after a potential current write completes) before trying any other EEPROM operations.
9.5 Issues of Interfacing

9.5.1 Interfacing with external devices

The EEPROM interface described in the previous section is very similar to other peripheral interfaces. Each device is controlled (and interacted with) through a small set of defined registers in the I/O register bank. A few quick examples should suffice to give the flavor of interaction.

The AT90S2313 provides as one of its built-in devices a UART attached to a set of pins configured to drive a standard serial port. We discount here the physical details of the electrical connections, which are interesting but would bring us more into the realm of electrical engineering than of computer architecture. The CPU interacts with the UART hardware through a set of four registers, the UART I/O Data Register (which stores the physical data to be transmitted or received), the UART Control Register (which controls the actual operation of UART, for example by enabling transmission or by setting operational parameters), the UART Baud Rate Register (which controls how fast/slow data is transferred), and the UART Status Register (a read-only register that shows the current status of the UART).

To send data across the UART, and thus across a serial line, the UART I/O Data Register must first be loaded with the data to be transmitted, and the Baud Rate Register must be loaded with a pattern representing the desired speed.\(^2\) To perform

\(^2\)SIDEBAR : CLOCKS, CLOCK SPEED, AND TIMING. So how do computers know what time it is? More importantly, how do they make sure that things that need to happen at the same time happen at the same time (like all bits in a register getting loaded at once)? The usual answer involves a controlling clock or timing circuit. This is just a simple circuit, usually hooked up to a crystal oscillator like the one in a digital watch. This oscillator will vibrate zillions of times per second, and each vibration is captured as an electrical signal and sent to all the electronic components. This, technically, is where the “1.5 GHz” in a computer description comes from — the master clock circuit in such a computer is a signal with a 1.5 GHz frequency, or in other words vibrates 1,500,000,000 times/second. As will be seen in appendix A, this clock signal both allows needed computations to proceed and prevents spurious noise from introducing errors.

For actions that need to be taken repeatedly, such as refreshing the screen every thirtieth of a second, a slave circuit will simply count (in this case) 50,000,000 master clock cycles, and then refresh the screen. Obviously, things that need to happen fast, such as the fetch/execute cycle, will be timed to be as short as possible, ideally at a rate of one per clock circuit. The baud rate on the UART controller is controlled by a similar slave circuit. The “speed pattern” tells this circuit how master clock ticks should occur before the UART should change signals.

This is also the reason that the overclocking trick works. If you have a processor designed to run at 1.5GHz, you can adjust the master clock circuit (perhaps even changing crystals) to run at 2.0Hz. The CPU doesn’t know that its signal is coming in too fast and will try to respond at the higher rate. If it really is running at the rate of one fetch/execute per clock tick, it will try to fetch and execute faster. In a sense, it’s like trying to play a record at a higher speed than normal (45rpm instead of 33rpm). (Ask your parents.) Sometimes this works, and you just got a cheap 30% speed boost. On the other hand, the CPU may not physically be able to respond to the faster speed, and it might die horribly (for
the data transfer, the control register must be set to “Transmitter Enable” (formally speaking, bit 3 of the UARD Control Register must be set to 1). If an error occurs in transmission, appropriate bits will be set in the Status register where the computer can observe them and take appropriate corrective action.

Interacting with the dataport(s) is much the same. Unlike the UART, the dataports are configured to allow up to eight independent electrical signals to be transferred simultaneously. Using this, a single data port could simultaneously monitor three push buttons (as input devices) and a switch (as an input device), while controlling four output LEDs. Each data port is controlled by two registers, one (the Data Direction Register) defining for each individual bit whether it controls an input or output device, and the other (the Data Register) holding the the appropriate value. To turn on an LED connected (say) to pin 6, the programmer would first make sure that the the 6th bit of the DDR was set to 1 (configuring the pin as an output device), then set the value in the 6th bit of the data register to ‘1’, bringing the pin voltage high (about 3–5 volts) and turning on the LED. Setting this bit to ‘0’ would correspondingly turn off the LED by setting the pin voltage to near zero volts.

9.5.2 Interfacing with timers

The AVR also includes several built-in timers to handle normal tasks such as measuring time or performing an operation at regular intervals. (Think about an elevator: the door opens, the elevator waits a fixed number of seconds, and then the door closes again. A door that stayed open only a microsecond would be unhelpful.) Conceptually, these timers are very simple: an internal register is set to an initial value. The timer then counts clock pulses (adding one to the internal register each time), either from the internal system clock or an external source of timing pulses, until the internal register “rolls over” by counting from a set of all ones to a set of all zeros. At this point, the timer goes off and the appropriate amount of time has passed.

Perhaps an example is appropriate here. I will assume that we have a source of clock pulses that comes once every 2 $\mu$s. Loading an eight-bit timing counter with the initial value of 6 will cause it to increment to 7, 8, 9, … every 2 $\mu$s. After 250 such increments (500 $\mu$s, or about 1/2000 of a second), the timer will “roll over” to 256, which will overflow the register to 0. At this point, the timer can somehow signal the CPU that an appropriate amount of time has passed so the CPU can do whatever it was waiting for.

One common and important kind of timer is the so-called watchdog timer, whose purpose is to prevent the system from locking up. For example, a badly written toaster program could have a bug that went into an infinite loop right after the heating element turned on. Since infinity is a very long time, the effect of such example, if the cooling is inadequate). Sometimes the CPU will be overrunning the rest of the components (the CPU is asking for data faster than the memory can provide it or the bus can move it).
9.6. DESIGNING AN AVR PROGRAM

an infinite loop would be (at least) to burn your toast, and very likely your table, your kitchen, and possibly your apartment building. The watchdog works as a normal timer, except that the “signals” it sends the CPU are equivalent to pressing the reset button and thus restarting the system in a known (sane) state. It is the responsibility of the program to periodically reset the watchdog (sometimes called “kicking” it) to keep it from triggering.

Although the timer itself is simple, the CPU’s actions can be less so. There are two ways that the CPU can interact with the timer. The first, dumb, way, is for the CPU to put itself into a loop, polling the appropriate I/O register to see whether or not the timer has completed. If the timer has not completed, the CPU returns to the top of the loop. Unfortunately, this method of continuous polling (sometimes called a busy-wait) prevents the CPU from getting any other, more useful, processing accomplished. You can, if you like, think of busy-waiting as the process of sitting by a telephone waiting for an important call, instead of getting on with your life.

A more intelligent way of dealing with expected future events (this also applies to waiting-by-the-phone, by the way) is to set up an interrupt handler. This is more or less how the AVR deals with expected but unpredictable events. The AVR knows a few very general kinds of interrupts that are generated under hardware-defined circumstances, such as the timer overflowing, an electrical signal on an established pin, or even on power-up. On the AVR in particular, the possible interrupts for a given chip are numbered from zero to a small value (like ten). These numbers also correspond to locations in FLASH ROM (program code) in the interrupt vector — when interrupt number 0 occurs, the CPU will jump to location 0x00 and execute whatever code is stored there. Interrupt number 1 would jump to location 0x01, and so forth. Usually, all that is stored in the actual interrupt location itself is a single JMP instruction to transfer control (still inside the interrupt handler) to a larger block of code that does the real work. (In particular, the watchdog timer is defined to generate the same interrupt that would be created by the reset button or a power-on event, thus providing a degree of protection against infinite loops and other program bugs.)

9.6 Designing an AVR Program

As a final example, here is a design (but not actual completed code) for how a microcontroller program might work in real life. The first observation is that microcontrollers are rather specialized computers, and that there are a lot of kinds of programs that it would be silly to write for a microcontroller. The physical structure of the AVR yields some obvious examples — it would be silly to try to write a program that involves lots of floating point calculations, for example, on a computer that doesn’t have an FPU or floating point instructions. However, the AVR is a very good chip for programs within its capacities.

For a semi-realistic example, we’ll look at a type of software that could be run practically on a microcontroller. Specifically, the design of a traffic light, of
the sort you can see at any typical busy intersection. I assume there’s a street running north/south that crosses another street running east/west, and the city traffic planners want to make sure that only one street can go at a single time. (I also assume the usual pattern of red/yellow[amber]/green lights, meaning stop, caution, and go.)

In order for this to work, there is a set of four different patterns that need to be presented.

<table>
<thead>
<tr>
<th>Pattern number</th>
<th>N/S light</th>
<th>E/W Light</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Green</td>
<td>Red</td>
<td>Traffic flows N/S</td>
</tr>
<tr>
<td>1</td>
<td>Yellow</td>
<td>Red</td>
<td>Traffic slowing N/S</td>
</tr>
<tr>
<td>2</td>
<td>Red</td>
<td>Green</td>
<td>Traffic flows E/W</td>
</tr>
<tr>
<td>3</td>
<td>Red</td>
<td>Yellow</td>
<td>Traffic slowing E/W</td>
</tr>
</tbody>
</table>

Actually, this might not work. For safety’s sake, we might want to set all the lights to red in between traffic going from N/S to E/W and vice versa, to allow the intersection to clear. It would also be nice to have an emergency setting of all reds “just in case.” We can add these as three additional patterns:

<table>
<thead>
<tr>
<th>Pattern number</th>
<th>N/S light</th>
<th>E/W Light</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Red</td>
<td>Red</td>
<td>Traffic about to flow N/S</td>
</tr>
<tr>
<td>5</td>
<td>Red</td>
<td>Red</td>
<td>Traffic about to flow E/W</td>
</tr>
<tr>
<td>6</td>
<td>Red</td>
<td>Red</td>
<td>Emergency</td>
</tr>
</tbody>
</table>

This tabulation leads to two other observations. First, all the program really needs to do is to transition (with appropriate timing) between the patterns in the following order: 0, 1, 5, 2, 3, 4, 0, . . . . Second, like so many other microcontroller programs, there’s no real stopping point for the program. For once, it’s not only useful, but probably essential that the program run in an infinite loop.

The easiest way to make such a program is to implement what’s called a state machine. The “state” of such a program is simply a number representing what pattern the lights are currently displaying (e.g., if the state is 4, all lights should be red). Each state can be held for a certain length of time (as measured by the timer). When the timer interrupt occurs, the computer will change the “state” (and the lights) and reset the timer to measure the next amount of time.

We can also make use of other interrupts in this state table. For example, we attach a special police-only switch to a pin corresponding to an external interrupt. The interrupt handler for this interrupt will be written such that the computer goes into a specific all-red emergency state. When that switch is closed (by the police pressing a button), the controller will immediately execute the interrupt. Similar use of external interrupts could cause the computer to detect when/if a passing pedestrian presses the “walk” button, causing the computer to transition to yet another state, where the appropriate walk light is showing for the right amount of time. And, of course, we can use the watchdog timer to look for possible program bugs, kicking it as necessary (say, every time the lights change); in the unlikely event that the watchdog timer triggers, we could either have the program go to a specific pre-programmed normal state, or else to go to the “emergency” state on the grounds that something had to have gone wrong and the system needs to be looked at.
9.7 Chapter Review

- A microcontroller is a small, single-chip, limited-capacity computer used for small scale operations such as device control or monitoring.

- These microcontrollers are found in many kinds of gadgets and devices, most of which wouldn’t seem (offhand) to be computers at all, such as the braking system of a car.

- The Atmel AVR is a family of related microcontroller with a specialized instruction set for this sort of task. The architecture of the AVR is substantially different from the architecture of a more typical full-service computer — for example, the AVR doesn’t have support for floating point operations, contains fewer than 10,000 bytes of memory, but has extensive on-board peripheral device support.

- Like many microcontrollers, the Atmel AVR is an example of RISC processing. There are a relatively few number of machine instructions tuned for specific purposes.

- The AVR is an example of Harvard architecture, where memory is divided into several (in this case, three) different banks, each with different functions and access methods. The registers (and I/O registers) of the AVR are located in one of three memory banks, along with general-purpose RAM for variable storage. The AVR also has a bank of FLASH ROM for program storage and a bank of EEPROM for storage of static variables whose value must survive a power outage.

- Input and output in the Atmel AVR is accomplished through I/O registers in the memory bank. A typical device will have a control register and a data register. Data to be read or written is placed in the data register, and then bits in the control register will be manipulated appropriately to cause the operation to happen. Different devices will have different registers and potentially different appropriate manipulations.

- Infrequent but expected events are handled efficiently via interrupts and their corresponding interrupt handlers. When an interrupt occurs, the normal fetch-execute cycle is modified to branch to a predefined location where appropriate (interrupt-specific) actions can be taken. Such interrupts are not restricted to the Atmel AVR but happen on most computers including the Pentium and the PowerPC.

- The AVR is only a good chip for certain kinds of programs due to the limitations of its hardware and capacities. A typical microcontroller program is a state machine that simply runs forever (in an infinite loop) performing a well-defined set of actions (like changing traffic lights) in a fixed, predefined sequence.
9.8 Exercises

1. What are three typical characteristics of a microcontroller?

2. Why does the Atmel AVR use RISC principles in its design?

3. What components of a typical computer are not found on the Atmel AVR?

4. Is the Atmel an example of von Neumann architecture? Why or why not?

5. What’s the difference between RAM and ROM?

6. Why does SRAM cost more per byte than DRAM?

7. What are the memory banks of the Atmel AVR, and what are their uses?

8. What is the function of a watchdog timer?

9. What is meant by “Memory-mapped I/O”?

10. Describe the procedure for the Atmel to make an LED flash off and on repeatedly.

11. How would the traffic light example be modified if we wanted the lights in emergency mode to flash RED–OFF–RED–OFF…?
Chapter 10

Advanced Programming
Topics on the JVM

10.1 Complex and derived types

10.1.1 The need for derived types

To this point, the discussion of computing has focused on operations on basic, elementary types such as integers and floating point numbers. Most problems, especially problems large or complex enough to need computers, are focused instead on less basic types — for example, to answer the questions “what’s the cheapest way to fly from Baltimore, MD to San Francisco, CA?” you need to understand planes, routes, and money. The notion of money is intimately connected with floating point numbers, while the notion of routes is more closely connected with the idea of sequences of starting and stopping points.

From a software designer’s point of view, it’s much easier to understand a solution if the solution is presented in terms of these high-level types, while the computer still can only operate on the basic types within its instruction set. The notion of “derived types” bridges this gap nicely. A derived type is a complex type built from (ultimately) basic types and upon which high-order computations can be performed. The derived type “money” can be built in a straightforward fashion from a floating point number (or more accurately, if less straightforwardly, from a combination of integers for the various units, like dollars and cents, or perhaps pounds, shillings, pence and farthings). The derived type “geographical location” can be built in a straightforward fashion from two numbers representing latitude and longitude.

A very abstract concept such as “route” could be built from a “list” of “flights” between “geographic locations,” each “flight” being associated with a cost, expressed in “money.” In this example, “route” would be a derived type. The types “list,” “flight,” “money,” and “geographic location” would also be derived types,
ultimately stored and manipulated as an appropriate collection of primitive types. From the software designer’s point of view, this is a powerful advantage — if such types can be implemented in the computer system itself and if the programmer can use computer implementations of the high level operations. Using these derived types to describe abstract, derived types can allow programmers and system designers to build complex systems more easily than they could build a single, monolithic, program.

Let’s start by looking in detail at some examples of derived types:

10.1.2 An example derived type: arrays

The theory

One of the simplest and most common kinds of derived types is the array. From a theoretical and platform independent perspective, an array is a collection of elements of identical type, indexed by an integer. You can use this definition to impress someone in a data structures class, if you have to. Meanwhile, let’s unpack it a bit: An array is an example of what’s sometimes called a “container type,” meaning that its only purpose is to store other pieces of information for later use. In an array, all the pieces have to be of the same type, such as all integers or all characters — but, of course, they can have different values. Finally, the individual locations for data are addressed using a number and specifically an integer that references that particular element. See, not so bad!

So, if you have an array of a thousand integers, how much space does that take up? Assuming a four byte integer, no bonus points are awarded for answering “at least 4,000 bytes.” This is true no matter what machine one is using. However, a block of memory this big simply won’t fit into a register. Fortunately, this isn’t a problem, since computations have to be performed on individual data elements and not on the array “as a whole.” The programmer can thus use a little trick to get at the data. She stores a number corresponding to the base of the array — a block of memory at least 4,000 bytes long, and usually the address of the initial element of the array. She also stores an offset, an integer index of the element she wants to use. On the 8088 or Pentium, these numbers would be stored in two different registers and a specific addressing mode would tell the computer to combine them appropriately to make an array access.

On the JVM, things are a little different, because there is no real notion of “address modes.” Instead, the two numbers are pushed on the stack and a special-purpose operation is performed to access the array appropriately. Actually, there are five special-purpose operations, depending upon what actually needs to be done. In approximately the order in which they are needed, these operations are:

- Creating a new array
- Loading a data item into an element of an array
- Retrieving a data item from an array element
- Determining the length of an array
• Destroying an array when no longer needed

More importantly, note that from the point of view of a high-level language programmer (or a computer science theorist), there’s no really difference between these two approaches. That’s because an array is fundamentally a derived type defined by its use. As long as there is some way of performing these actions — for example, loading a value into an array element at a particular index — the exact implementation doesn’t really matter, from a theoretical point of view. This is a point that will come up again in the discussion of classes proper.

Creation

Perhaps obviously, any array must be created before it can be used; the computer needs to reserve the appropriate (possibly very large) block of memory. In high-level languages such as C++ or Pascal, this is often performed automatically when the array variable is declared: the statement

```c
int sample[1000];
```

declares sample to be an array variable (holding elements of type ‘int’) and at the same time reserves enough space for a thousand integers, numbered from [0] to [999]. In Java, space for an array is reserved via explicit array creation, such as with

```java
int[] sample = new int[1000];
```

There is a subtle difference here. In the first example, the creation of the array is implicit in the declaration, while in the second, the creation of the array (the allocation of the block of memory) is done via an explicit command (“new”). The JVM, of course, doesn’t support variable declarations in the normal sense, but it does support array creation. This is accomplished through the use of the machine instruction `newarray`. In order to create an array, the programmer (and computer) needs to know both the size of array to be created and the type of element for it to contain.

The `newarray` instruction takes a single argument, the basic type of element for the array (for example, to create the sample array defined above, the type would be integer, abbreviated as I. The length of the array to be created must be available on the stack as an integer. This length will be popped, space reserved for a new array of the appropriate length and type, and then an address corresponding to the new array is pushed at the top of the stack. This address can be loaded and manipulated appropriately.

The JVM equivalent for defining the sample array above would look something like

```java
ldc 1000 ; push 1000 as the new array length
newarray int ; create an array of integers
astore 1 ; store the new array in #1
```
Arrays are one spot where the basic types of byte and char are actually used. In calculations on the stack, byte and char values are automatically promoted to integers, and in local variables, they are still sized as 32-bit quantities. This is a little bit wasteful of space, but it wastes less space (at most 3 bytes per local variable, a tiny number) than it would to make the JVM capable of storing small quantities. With arrays, the wasted space could be much more significant as the arrays grow. In fact, the JVM even provides a basic type for boolean arrays, since a machine could choose to store up to eight boolean values in a single byte (32 in a single word), for a better than 95% improvement in space efficiency.

Technically speaking, the newarray instruction (opcode 0xBC) will only create one-dimensional arrays of primitive, basic types such as ints and floats. An array of derived types is a bit more tricky to create, simply because the type needs to be specified. Because derived objects can be defined more or less at the programmer’s whim, there is not and cannot be a standardized list of all the possible derived types. The JVM provides a anewarray instruction (opcode 0xBD) for one-dimensional arrays of a derived type. As before, the size must be popped off the stack, while the type is given as an argument. However, the argument is itself rather complex (as will be discussed below), and actually refers to a String constant in the constant pool that describes the element type. From the JVM’s perspective, executing opcode 0xBD is much more complicated than executing opcode 0xBC, since it requires that the JVM look up the string in the constant pool, interpret it, check that it makes sense, and possibly load an entirely new class. From the programmer’s perspective, there is very little difference between creating an array of a basic and a derived type, and one possible enhancement to an assembler such as jasmin would be to allow the computer to determine (by examining the type of argument) whether opcode 0xBC or 0xBD is needed.

The most difficult part of creating an array of derived objects is specifying the type. For example, to create an array of String types (see below), the fully qualified type name “java/lang/String” must be used. The example below shows how to create an array of 1000 strings.

```
ldc 1000 ; 1000 elements in the array
anewarray java/lang/String ; create an array of Strings
```

This particular instruction is very unusual and very specific to the JVM; most computers do not provide this sort of support at the level of the instruction set for allocating blocks of memory, and even fewer support the idea of allocating typed blocks. However, the ability to support typed computations even when the types involved are user-defined is critical to the sort of cross-platform security envisioned by the designers of the JVM.

In addition to being able to create simple, one-dimensional arrays, the JVM also provides a shortcut instruction for the creation of multidimensional arrays. The multianewarray instruction (note the somewhat tricky spelling) actually pops a variable number of dimensions off the stack and creates an array of appropriate type. The first argument to multianewarray defines the type of the array (not, it
10.1. COMPLEX AND DERIVED TYPES

<table>
<thead>
<tr>
<th>Type</th>
<th>JVM/jasmin expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>I</td>
</tr>
<tr>
<td>long</td>
<td>J</td>
</tr>
<tr>
<td>float</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>D</td>
</tr>
<tr>
<td>byte</td>
<td>B</td>
</tr>
<tr>
<td>short</td>
<td>S</td>
</tr>
<tr>
<td>char</td>
<td>C</td>
</tr>
<tr>
<td>boolean</td>
<td>Z</td>
</tr>
<tr>
<td>void (return type)</td>
<td>V</td>
</tr>
<tr>
<td>array of X</td>
<td>[X</td>
</tr>
<tr>
<td>class Y</td>
<td>LY;</td>
</tr>
<tr>
<td>function taking X and returning Y</td>
<td>(X)Y</td>
</tr>
</tbody>
</table>

Table 10.1: Type Description Expressions

should be noted, the type of the array element) in a shorthand notation for types, while the second defines of the number of dimensions and thus the number of stack elements that need to be popped. For example, the code below specifies that three numbers are to be popped off the stack, creating a three-dimensional array:

\[
\begin{align*}
\text{bipush 6} & \quad ; \text{size of array in third dimension} = 6 \\
\text{bipush 5} & \quad ; \text{size of array in second dimension} = 5 \\
\text{bipush 3} & \quad ; \text{size of array in first dimension} = 3 \\
\text{multianewarray} & \quad \text{[[[F 3 ; create a 3x5x6 array of floats}
\end{align*}
\]

Note that the final array created has three dimensions, and is of overall size 3x5x6. The first argument, “[[[F” defines the type of the final array as a three-dimensional array (three ‘s) of floating point numbers.

The type system expanded

From previous work with System.out and printing various things, you should already be somewhat familiar with the system for expressing types to the JVM. The top half of table 10.1 lists familiar basic types (which are used in the newarray instruction) and their JVM expressions, as might be used in a call to invokevirtual or multianewarray. In addition to the basic computational types with which we are already familiar, the JVM also recognizes as basic types the byte (B), the short (S), the char (C), and the boolean (Z), also listed in table 10.1.

Derived types are, as might be expected, are derived from the expressions of underlying types. The type description of an array, for example, is an open square bracket ( [ ] ) followed by the type of every element in the array. Please note carefully that no closing bracket is needed or in fact, allowed. This has confused more than
one programmer. The expression for an array of integers would thus be \([I\), while the expression for a two-dimensional array (a matrix) of floats would be \([IF\) — this literally expresses an array (I) each of whose elements is an array of floats (IF).

Classes and class types are expressed using the fully qualified class name, bracketed in front by a capital L and in back by a semicolon (;). The system output System.out, for instance, is an object of class PrintStream, stored in the directory and package of java.io (or java/io). Thus, the proper class expression for System.out would be Ljava/io/PrintStream; as has been used before in many examples.

These type constructors can be combined as needed to express complex types. For example, the standard Java definition of the “main” routine takes as an argument an array of Strings. In Java, this is written with a line like

\[
\text{public static void main(String [] args)}
\]

String is typically a system-defined class in the java.lang package, so it would be expressed as Ljava/lang/String; while the argument is an array of such Strings. The main method does not return anything to the calling environment, and hence is declared with return type void. Thus, our by now familiar statement at the beginning of many methods:

\[
.method public static main([Ljava/lang/String;)V
\]

This declares that the method “main” accepts an array of Strings as its only argument, and returns nothing (void). It is evident from this example that this type system is used not only in the description of array types, but also in the definitions of methods as will be described later.

Storing

To store an item in an array, the JVM provides several similar instructions, depending upon the type of element. For simplicity, assume for the moment that it’s an array of integers (I). In order to store a value at a location in an array, the JVM needs to know three things: which array, which location, and which value. The programmer must push these three things (in that order) onto the stack, and then execute the instruction iastore. This operation is a little bit unusual in that it doesn’t push anything onto the stack afterwards, so it has the net effect of decreasing the stack size by three.

Figure 10.1 shows a simple example of how data can be stored into an array. In particular, this code fragment first creates an array of ten integers, then stores the numbers 0–9 into the corresponding array elements.

For other basic types, including the non-computational types of chars and shorts, the JVM provides appropriate types using the standard method of initial letters to define variants. For example, to store an element into an array of longs, use the lastore instruction. To store an element into an array of non-basic types (address types, such as an array of arrays or an array of objects), the elements are
10.1. COMPLEX AND DERIVED TYPES

bipush 10 ; need 10 elements
newarray I ; create array of 10 integers
astore_1 ; store array in location #1

iconst_0 ; load a zero for looping
istore_2 ; and store in #2

Loop:
aload_1 ; load the array
iload_2 ; load the location
iload_2 ; load the value (which is the same as the location)
iastore ; set array[location] = value
iinc 2 1 ; add 1 to #2 (the location and value)
iload_2 ; are we done yet (is location >= 10)?
bipush 10 ; load 10 for comparison
if_icmplt Loop ; if not at 10 yet, jump to Loop and repeat
; and we’re done!

Figure 10.1: Example of storing in an array

stored as addresses (a), so the instruction is aastore. The only tricky aspect is in distinguishing between an array of booleans and an array of bytes, both of which begin with the letter b. Fortunately, the JVM itself can handle this ambiguity, as it uses the instruction bastore for both byte and boolean arrays, and is capable of distinguishing between them on its own. (All right, this is a micro-lie. On most implementations of the JVM, especially the one coming from Sun Microsystems, the machine simply doesn’t bother to distinguish and just uses bytes to store boolean array elements. This wastes about 7 bits per element, which is still acceptably efficient.)

Storing into a multidimensional array must be performed as a sequence of stores. Because a multidimensional array is really stored (and treated) as an array of arrays, it is first necessary to load the relevant sub-array and then to load or store into it. Figure 10.2 shows a code fragment for placing the number 100 into one slot (specifically, location [1][2]) in a matrix of integers.

Loading

As with storing, so with loading. The JVM provides a set of instructions in the ?aload family that will extract an element from an array. To use these instructions, push the array and the desired location — the instruction will pop these arguments, then extract and push the value stored at that location, as in the example.

aload_1 ; load the 1000 int array stored at #1
bipush 56 ; push the value (desired location) 56
iaload ; extract and push the integer value array[56]
### Table 10.2: Array operations for loading and storing values

<table>
<thead>
<tr>
<th>Array element type</th>
<th>Store operation</th>
<th>Load operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>iastore</td>
<td>iaload</td>
</tr>
<tr>
<td>long</td>
<td>lastore</td>
<td>laaload</td>
</tr>
<tr>
<td>double</td>
<td>dastore</td>
<td>daload</td>
</tr>
<tr>
<td>float</td>
<td>fastore</td>
<td>faload</td>
</tr>
<tr>
<td>char</td>
<td>castore</td>
<td>caload</td>
</tr>
<tr>
<td>short</td>
<td>sastore</td>
<td>saaload</td>
</tr>
<tr>
<td>byte</td>
<td>bastore</td>
<td>baload</td>
</tr>
<tr>
<td>boolean</td>
<td>bastore</td>
<td>baload</td>
</tr>
<tr>
<td>array (address)</td>
<td>aastore</td>
<td>aaload</td>
</tr>
<tr>
<td>object (address)</td>
<td>aastore</td>
<td>aaload</td>
</tr>
</tbody>
</table>

Figure 10.2: Example of creating and storing into a multidimensional array

```java
bipush 3         ; second dimension is 3
bipush 4         ; first dimension is 4
multianewarray [[I 2] ; create 3x3 array of integers
astore_1         ; store array in #1
aload_1          ; load the array
iconst_1         ; we’re interested in a[1][?]
aaload           ; get a[1] (a row of 3 integers)
iconst_2         ; get location 2
bipush 100       ; load 100 to be placed in array a[1]
iastore          ; store 100 in a[1][2]
```
10.1. COMPLEX AND DERIVED TYPES

Getting the length

Getting the length of an array is easy. The `arraylength` instruction pops the first entry off the stack (which must, of course, be an array) and pushes the length of the array. For example, the code below loads the previously created sample array, takes its length, and would leave the (int) value 1000 on the stack.

```
aload_1 ; load previously defined sample (1000 int array)
arraylength ; pop sample, push sample’s length
```

Destroying

In contrast to the previous operations, destroying an array when its contents are no longer needed is very simple. In fact, it’s something that you, as the programmer, need not even think about. The JVM standard defines that the machine itself should periodically perform garbage collection, finding memory, class instances, and variables that are no longer used by the program. Once such things are found, they are collected and made available for re-use.

The exact definition and operation of the garbage collection routine will vary from JVM implementation to implementation. The general definition of “garbage” is that the memory location is no longer reachable from the program. For example, if local variable #1 holds (the only copy of the address of) an array, the array and every element in it are reachable and possibly available for computation. If the programmer were to write over local variable #1, then, although the array itself has not changed, it’s no longer to access the information in it. At this point, the memory taken up by the array (which might be very extensive) doesn’t store anything useful, and might as well be recycled for other, useful, purposes. The only downside is that there’s no way to predict exactly when this recycling might happen, and it’s technically legal for a JVM implementation not to perform this garbage collection at all.

From the programmer’s perspective, there is no need to explicitly destroy an array. Simply by popping or overwriting all the references to the array, something that will usually happen by itself in the normal course of running the program, will cause the array to become unreachable, “garbage,” and therefore recycled.

10.1.3 Records : classes without methods

The theory

The next simplest derived type is called, variously, a structure or a record. Again, like an array, this is a container type. Unlike an array, the data stored in a record is contained in named fields, and may be of different types. The record provides a method of keeping related data together in a single logical location. If you like, you can think of a record like an electronic baseball trading card, which carries all the information relevant to a single player (batting average, home runs hit, times at bat, runs batted in, stolen bases, etc.) in one consistent easy-to-transport format.
; structure definition for 'fraction' types

.class public fraction
.super java/lang/Object

.field public numerator I
.field public denominator I

; boilerplate -- needed because 'structures' are really 'classes'
.method public <init>()V
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

Figure 10.3: A sample record defining “fraction” as a derived type

Each of these pieces of information would be associated with a particular field name
(e.g. “RBI” for runs batted in), and possibly have different types (batting average
is defined as a float, while the number of home runs is an integer, and position —
“shortstop” — might even be a String). A simpler example would be a fraction,
with two integer fields.

Unlike an array, each record type must be defined separately at compile time
by the programmer. The definition will mostly be a list of the necessary field names
and their respective types. These will be stored in a suspiciously familiar-looking
file, as shown in figure 10.3.

This example program shows a simple instance of a record type, specifically a
fraction (or as a mathematician might put it, a rational number). These are formally
defined as the ratio between two integers, named the numerator (the number on
the top) and the denominator (on the bottom) respectively. The two key lines in
this file that define these fields are the lines beginning with the .field directive.
Specifically, the line

.field public numerator I

defines a field named numerator whose value is of type “I” (for integer, as
discussed above. A field with a long value would use a “J”, while a field with a
String (a derived type) would use the expression “Ljava/lang/String;” as we have
seen before. The public keyword means that the numerator value is “public,”
meaning that it can be accessed, read, and modified by other functions and other
objects in the system.
So how about the rest of the file? A close inspection reveals that it’s the same boilerplate we have been using since the first examples for the definitions of classes. The reason for this is very simple: a record, in the JVM, is actually implemented as a class, and so there is some minimal class overhead that also must be present to allow the rest of the system to interact with our newly-defined record. So without further ado, let’s look specifically at classes as a derived type and see what their advantages are.

10.2 Classes and Inheritance

10.2.1 Defining classes

A major advance in programming technology — or more accurately, in program design — was the development of object-oriented programming. Under this framework, large programs are designed using systems of smaller, interactive objects that are individually responsible for their own data processing. A common metaphor is that of a restaurant patron. Diners can order any dish they like without having to worry about preparation details; that’s the kitchen’s responsibility. (And the cook doesn’t worry about who ordered a particular dish; that’s the server’s responsibility.) This division of responsibility really pays off because a code fragment written for one purpose can often be coopted and reused for another purpose, and a large system can be built with relative ease as a collection of interacting objects. To continue a repeated example, the random number generator designed in section 3.4.1 can be used any time a random number is needed, whether for a Monte Carlo simulation, a Vegas-style casino game, or a first-person shoot-em-up.

In order to structure these objects into a coherent framework, they are usually grouped into (and written as) classes of similarly-propertied objects. One random number generator is much the same as any other, even if the detailed parameters or detailed operations may differ. In particular, the things that an outside observer would want to do to or with a random number generator (seed it to a state to start generating, or get a new random number from the generator) are the same. We can then define the idea of a “random number generator” operationally — in terms not of how it works, but of what we can do with it. Anything that calls itself a “random number generator” will have to satisfy those two operations.

This leads us to a familiar formal definition of a class as an abstract description of a derived type, consisting of a set of named (instead of numbered) fields of potentially different types. (Sounds a lot like a record, doesn’t it?) The difference is that a class also contains methods, functions that define legitimate ways to interact with the class. Finally, an object is an example or instantiation of a class, so where a class might be an abstract concept (like a “car”), an object would be a particular car, like the specific old VW Bug that I used to work on in high school.

As a quick review, a class (from the outside) is a way of grouping together similarly-propertied objects that can all be interacted with in the same way. On a computer running Apple’s OS X operating system, all windows have three buttons...
at the upper left corner: red, for deleting the window; yellow, for iconifying it; and green, for expanding it. Once the user understands how to work with any one window, she can work with all of them. This idea generalizes to the notion of classes, objects, and methods. In particular, each object (or instance of a class) shares the same methods, or defined functions for interacting with that object. If you understand how to steer one example of a VW Bug, you know how to steer all of them, because the “method” of steering is identical.

This view continues to hold when programming on the JVM, because two objects are represented as independent instances of class files, where each class is largely independent, and relies on the same methods for communicating at the JVM bytecode level. We have already seen examples of this in previous programs; mostly in interacting with the system output.

In detail:

- System.out (in jasmin, System/out) is a particular object
- System/out instantiates the java/io/PrintStream class
- All PrintStream objects have a println method
- The println method causes a String to appear in “the usual place,” which can vary from object to object; for System/out, it appears at the standard system output such as the screen.
- To make this happen, one uses the invokevirtual instruction, which triggers the appropriate method (println) on the appropriate object (System/out) [of the appropriate type (PrintStream)]

Each system is required (by the Java/JVM standards documents) to provide a PrintStream class and a System.out object as a member of that class. The exact details — for example, whether to print to a file on disk, a window on the screen, or a printer — are left to the individual classes. Furthermore, it’s easy to set up another object (of class PrintStream) that prints its data somewhere different; then, instead of invoking the println method of System/out, one invokes that same method of the new object, and thereby print to a file instead of the screen.

Java does not enforce object-oriented programming, but the designed structure of the language makes it very easy and profitable to use it. Similarly, the structure of the JVM does not enforce object-oriented programming, but does encourage it. In particular, the JVM specifically stores executable programs as class files and, as shown, makes it very easy to build a large-scale system using interoperating classes. We present some examples of such derived classes below.

10.2.2 A sample class: String

Classes, like arrays and fields, are derived types, but the various methods included as class definitions can make them much more difficult to understand. A typical, but still relatively understandable, example of a derived type is the standard Java “String” class, defined as part of the Java.lang (or Java/lang) package. The String
class simply holds an immutable and unchangeable string such as “Hello, world!”, perhaps to be printed; the class defines both the type of data used in a String (usually an array of characters) as well as a set of methods, functions and operations which are defined to be valid ways of interacting with the String class. We’ve been using String objects for some time without a formal understanding of their properties.

Using a String

In addition to storing the actual string value, the String also supports a wide collection of computational methods that will inspect the String to determine properties of the string. For example, the charAt() method takes an integer and returns the character at the location specified.

In Java (or an equivalent object-oriented, high-level language), this function would be defined to fit a calling scheme something like

```java
class java.lang.String {
    char charAt(int);
}
```

This scheme, used both for defining the function and for calling it, states that the method `charAt()` is part of the class `String` (itself part of the `java/lang` package), takes a single integer as a parameter and returns a character value. In jasmin, these same concepts would be expressed in a slightly different syntax, using the same syntax given in table 10.1:

```java
java/lang/String/charAt(I)C
```

(Quick review: this means that the symbol ‘charAt’ is a function taking type I and returning type C.) As we shall see, this kind of syntax is used both for defining the method itself and for invoking the method on any particular string.

The compareTo() method compares the current String with another and determines which one is the alphabetically prior; if this (the current string) would come before the argument String in a dictionary, either by being shorter or by having a letter earlier in the usual sorting sequence, the integer returned will be negative. If this comes after the String argument, the return value will be positive, and if the Strings are exactly equal, a zero is returned. In our jasmin notation, this method looks like

```java
java/lang/String/compareTo(Ljava/lang/String;)I
```

Other methods specified (by the standard) as belonging to the String class include `equals()`, which returns a boolean value to tell you whether one string is identical to another; `equalsIgnoreCase()`, which does the same calculation but ignoring case (so “Fred” and “fred” are not `equal`, but would be `equalIgnoreCase`);
indexOf(), which returns the location of the first occurrence of the character specified as an argument; length(), which returns the length of the String, and toUpperCase(), which returns a new String in which all characters have been converted to CAPITAL LETTERS. All in all, there are more than fifty separate methods, not counting the ones implicitly inherited from the Object class, that are defined by the standard as part of the JVM java.lang.String class.

10.2.3 Implementing a String

Under the hood, and at the bytecode level, how is a String actually implemented? The answer, annoying but brilliant, is that it doesn’t matter! Any valid JVM class that implements the appropriate fifty methods, irrespective of the actual details, is a valid version of the class. As long as other classes only use the well-defined standard methods to interact with the String class will find that a well-behaved String class will serve their needs.

One (stupid) possibility for implementing a String, for example, would be as an ordered collection of individual character variables, with each one representing a separate character in the String. This has some obvious drawbacks from a programmer’s point of view, as he would need to create lots and lots of variables with names like seventyeighthcharacter. A better solution would be to use some sort of simple derived type such as a character array (see above). Even here, there are a few choices the designer could make. For instance, he could try to save space and use a byte array (if most of the Strings he expects to deal with are ASCII strings, then he doesn’t need to deal with UTF-16 on a regular basis). He could also use an array of integer types, to simplify any needed calculations. The String could be stored in this array in normal order, so that the first element of the array corresponds to the initial character of the string, or he could store the array in reverse order (to simplify his implementation of the endsWith() method).

Similarly, he may or may not want to create a special field holding the length of the string as an integer value. If he does so, this will make each individual String object a little larger and a little more complex, but it will also make it faster to use the length() method. Tradeoffs like these can be important to the overall performance of the system, but they have no effect on whether or not his version of String is legitimate — and anyone else who uses his String class will find that, if all methods are there and correct, their program will still work. There doesn’t need to be any specific relationship between the String class file and the class files that use Strings.

Constructing a String

String has a special method (usually called a constructor function) that can be used to make a new String. This method takes no arguments, and most of the time isn’t very useful, because the String created is of zero length and has no characters. With the notation used so far in the text this constructor function would be described as
10.3. CLASS OPERATIONS AND METHODS

To make it easier and more useful, the String class also has many (about eleven) other constructors that allow you to specify the contents of the String to be created. For example, a programmer can duplicate an existing String by creating a new String from it as follows:

```java
java/lang/String/<init>()V
```

He can also create a string from a StringBuffer (which makes an immutable string from a mutable one):

```java
java/lang/String/<init>(Ljava/lang/StringBuffer;)V
```

or directly from an array of characters:

```java
java/lang/String/<init>([C)V
```

any one of which would give him control over both the creation of the String as well as its contents.

10.3 Class Operations and Methods

10.3.1 Introduction to class operations

Classes are, in general, much more complex than arrays, because they are required (or at least, allowed) to provide many more operations and many more kinds of operations than arrays. Because of this complexity, it’s not usually practical to create a new class on the fly (although, of course, one can always create a new object instantiating an existing class). Classes are, instead, defined via .class files as we have been doing; the basic class properties, such as fields and methods, are defined via jasmin directives at compile-time. Once these fields and methods have been created, they can be used by anyone on the system (with appropriate access permissions).

10.3.2 Field operations

Creating fields

Unlike arrays, fields within classes are named, not numbered. However, fields within different classes may be called the same thing, raising the possibility of ambiguity and confusion. For this reason, when fields are used, they should be fully described including the name of the class that introduces the field.

To create a field, jasmin uses the directive .field, as illustrated below (and also in figure 10.3)
.field public AnExampleField I

This example creates a field in the current class named AnExampleField, which holds a variable of type int. Because this field is declared “public,” it can be accessed and manipulated from methods that are not themselves part of the current class — presumably the programmer has some reason for this.

The .field directive allows several other access specifications and arguments. For example, a field can be declared “final,” meaning that its value cannot be changed from the value set in the field definition, or it can be declared “static,” meaning that it is a field associated with a class rather than with individual objects within the class.

.field public static final double PI D = 3.1415926537898

This example defines “PI” as a static (class-oriented), final (unchangeable) double with the value of \( \pi \). If for some reason the programmer wanted to restrict the use of PI to methods within the Example class, it could be declared “private” instead. Valid access specifications include private, public, protected, final, and static, all of which take their usual meaning in Java.

Using fields

Since fields are automatically part of their objects/classes, they are automatically created as part of object creation (or in the case of static fields, as part of class loading), and destroyed when the corresponding object/class is swept in garbage collection. The two operations of separate interest to the programmer are therefore storing data into a field, or loading it from a field (into the stack).

The procedure for storing into a field is similar to the procedure for storing into an array, with the significant difference that fields are named, instead of numbered. As such, the putfield operation takes the name of the relevant field as an argument (because names, as such, cannot be placed on the stack). The programmer needs to push the address of the relevant object (whose field is to be set) and the new value (which must be of the correct type), then executes the appropriate putfield instruction as shown.

    aload_1 ; #1 should hold the address (hence the 'a')
    ; of an instance of the Example class
    ; (an object of class Example)
    bipush 10 ; push 10 to be put into the AnExampleField
    putfield Example/AnExampleField I ; put 10 into the field as an int

Static fields, being attached to a class instead of a particular object, have a similar structure but do not need to have an object on the stack. Instead, they are simply placed into the appropriate class field using the instruction putstatic. If the PI example above had not been defined as “final,” then a crazy programmer could adjust the internal value of PI by
ldc_2w 3.0 ; load 3.0 to become the new value of PI
putstatic Example/PI D ; set PI to be 3.0 for the Example class
; of course, this wouldn't work, since PI
; was defined as 'final' above

The JVM also provides instructions (getfield and getstatic) for retrieving values from fields. The System class (defined in java.lang, and thus formally java/lang/System) contains a statically defined field called out that contains a PrintStream object. To get the value of this object, we use the by now familiar line

getstatic java/lang/System/out Ljava/io/PrintStream;

Because java/lang/System is a class, nothing need be on the stack and nothing will be popped in the process of executing getstatic. When accessing a non-static field (using getfield), since the value needs to be selected from a particular object, the object must first be pushed onto the stack as illustrated

aload_1 ; load Example object from #1
getfield Example/AnExampleField I ; push AnExampleField as int

10.3.3 Methods

Method introduction

In addition to fields, most classes also possess methods, ways of acting upon the data stored in the class or its objects. Methods differ from fields in that they actually compute things, and therefore contain bytecode. Like fields, there are several different kinds of methods, with different properties — the “main” method, for instance, must nearly always be defined as both “public” and “static” because of the way the JVM interpreter works. When the JVM attempts to execute a class file, it looks for a method defined as part of the class (and not as part of any particular object, since there are no objects of that class; thus, “static”) to execute. Since there are no objects of that class, this method must be publicly accessible. For this reason, every program we have yet written includes the line

.method public static main([Ljava/lang/String;)V

as a definition of main as a public, static, method.

Method invocation via invokevirtual

Methods are declared and defined, of course, in their corresponding class file. To actually use a method requires that it be invoked on an appropriate object or class. There are a few different basic operations that correspond to method invocation, used in slightly different ways depending upon the exact circumstances.
The most common and most straightforward way to invoke a method uses the `invokevirtual` operation (opcode 0xB6). Here again, we have been using this for several chapters already, and there is nothing especially new or conceptually difficult about it. The operation pops an object (and the arguments to the method) from the stack, invokes the appropriate method on the object, and pushes the result, as in the following (standard) code:

```java
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Hello, world!"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

This code pushes the object `System.out` (a `PrintStream`) and one argument. By inspection of the `invokevirtual` line, we can see that the stack must contain, at the top, a single argument of type `java/lang/String`, and then below it a `PrintStream` object. A more complicated method might take several arguments, but they will all be specified in the `invokevirtual` line, and so the computer knows exactly how many arguments to pop. Below all arguments is the object whose method is to be invoked.

When such a method is invoked, control will be passed to the new method, in similar fashion to a subroutine call. However, there are a few crucial differences. First, the arguments to the method are placed sequentially in local variables starting at #1. Local variable #0 gets a copy of the object itself whose method is being invoked (in Java terms, #0 gets a copy of `this`). The new method also gets a completely new set of local variables and a completely new (and empty) stack.

When computation is completed, the method must return, and return with the appropriate type expected by the method definition. From the calling environment’s viewpoint, the method should push an element of the appropriate type at the top of the stack. From the method’s viewpoint, it needs to know which element (and what type.) There are several commands to return, starting with `return` (which returns nothing, i.e. a void type), and the `?return` family, which will return the appropriate element of the appropriate type at the top of the stack — members of this family include the usual suspects `i`, `l`, `f`, `d`, as well as `areturn` to return an address or object. This family does not, however, include the `ret` instruction, which returns from a subroutine. It is also not legal to “fall-off” the end of a method; unlike Java and most high-level languages, there is no implicit `return` at the end of a method.

This can be seen in this very simple method, which simply returns 1 if and only if the argument is the integer 3.

```java
.method public isThree(I)I
  .limit locals 2
  .limit stack 2
  iload_1     ; argument, an int, is in #1
  bipush 3    ; load 3 for comparison
  if_icmpeq   Yes       ; if #1 == 3, then return 1
```

```java
return
```
bipush 0 ; not equal, so return 0
ireturn

Yes:
bipush 1 ; equal, so return 1
ireturn
.end method

There is a very important difference between subroutines (accessed via jsr/ret) and methods (accessed via invokevirtual/?return). When a subroutine is called, the calling environment and the called routine share local variables and the current stack state. With methods, each time you start a method, you get a brand new set of local variables (all uninitialized) and a brand new stack (empty). Because each new method invocation gets a new stack and a new set of local variables, methods support recursion (having a method re-invoke itself) in a way that subroutines do not. To invoke a method recursively, one can simply use the value stored in #0 as the target object and write the invokevirtual line normally. When the method executes, it will use its own private version of the stack and local variables, without affecting the main stream of computation — upon return from the method, the calling environment can simply pick up where it left off using the results of the method invocation.

Other invoke? instructions

Since invokevirtual takes an object and its arguments to call a method, it (perhaps obviously) isn’t suitable for use on a static method. Static methods don’t have associated objects. The JVM provides a special invokevirtual operation for invoking static methods upon classes. This operates just as invokevirtual would (and looks very similar), except that it does not attempt to pop an object from the stack, just the arguments. It also does not bother to put the object (this) into #0, and instead fills the local variables up with the arguments starting from #0.

There are also a few special circumstances that call for special handling. Specifically, when one initializes a new object (using the

<init>

method, or has to deal with a few tricky situations involving superclasses and private methods, there is a special operation invokespecial that needs to be used. From the programmer’s viewpoint, there is no difference between invokevirtual and invokespecial, but our standard boilerplate code
illustrates the main use of \texttt{invokespecial}. In order to initialize an object (of any class), it is first necessary to confirm that it has all the properties of the superclasses (for example, if a Dog is a subclass of Animal, to initialize a Dog, one needs to make sure it's a valid Animal). This is accomplished by calling the initialization method on the current object (\texttt{this}, or local variable \#0). But because it's an initialization method, the computer has to use \texttt{invokespecial}.

### Declaring classes

Classes are typically defined and declared in files with the \texttt{.class} extension. These files, though, contain necessary information about the classes themselves and their relationship with other classes that the JVM needs in order to operate properly.

For this reason, every class file created needs to have the two directives

\begin{verbatim}
.class Something
.super ParentOfSomething
\end{verbatim}

\texttt{.class} to define the class itself and its place in the class hierarchy. Like fields and methods, the \texttt{.class} directive can also take various access specifications such as \texttt{public}. The class name (\texttt{Something} in the above example) should be the fully-qualified name, including the name of any packages: the class \texttt{System}, defined as part of the java.lang package, would contain the fully-qualified class name of \texttt{java/lang/System}. Similarly, the superclass should include the fully-qualified name of the immediate superclass, and must be present; although most student-written classes are subclasses of \texttt{java/lang/Object}, this is not a default and must be specified explicitly.

There are a number of other directives that may or may not be present in a class file, mostly directives of use to source debuggers. For example the \texttt{.source} directive tells the JVM program the name of the file that was used to generate the class file; if something goes wrong with the program at run time, the JVM interpreter can use this information to print more useful error messages.

### 10.3.4 A taxonomy of classes

One important subtlety glossed over in the previous section is the existence of several different types of class files and methods. Although they are all very similar (the actual difference in storage is usually just setting or clearing a few bits in the access flags in the class file), they represent profound differences in the class semantics, especially as viewed from outside the class.

The most common relationship between classes, objects, and methods is where every object in a class has its own individual set of data, but is operated on via a unified collection of methods. For example, consider the “class” of any specific model of car (let’s take as a specific, the 2001 Honda Accord). Obviously, these cars all handle (in theory) identically, but they have individual properties such as color, amount of gas in the tank, and Vehicle Identification Number. These
properties would be stored as fields in the individual Honda objects themselves. On the other hand, the headlights are controlled in exactly the same way on every car; the “method” of turning on the headlights is a property of the class, not of any individual car.

There are, however, certain properties of the class as a whole, such as the length of the car, the width of the wheelbase, and the size of the gas tank. These properties can even be read directly from the blueprints and do not need any cars to actually exist! We distinguish between class variables, which are properties of the class itself, and instance variables, that are properties of individual instances. In the JVM fields that are class variables are declared as static and stored in the class itself, instead of in individual objects.

```
.field colorLjava/lang/String;
.field static carLengthD
```

Similarly, fields can be declared as final to represent that the value, whether an instance variable or a class variable, cannot be changed.

```
.field final vehicleIdentificationLjava/lang/String;&quot;whatever&quot;
.field static final numberOfWheelsI = 4
```

In this example, the field vehicleIdentification is an instance variable that doesn’t change (but may vary from car to car), while all cars in the class have the same fixed and immutable number of wheels. The length of a car is a property of the class, while the color of a car is a property of the individual car-object, and can be changed (if one decides to repaint, for example).

A similar distinction between instance methods and class methods holds. Most of the programs written so far have involved a method defined as

```
.method public static main([Ljava/lang/String;)V
```

This “main” method is very important, because, by default, whenever a Java program is run (or a class file is executed using java, more exactly), the java program will load the class file and look for a method named “main.” If it finds one, it will then attempt to invoke that method with a single argument corresponding to the rest of the words typed on the command line. The static keyword indicates that the “main” method is associated with the class, and not with any particular instance of the class. For this reason, it is not necessary to create any instances of the appropriate class in order to invoke the main method.

In addition, we also have the necessary property “public,” which says that this method (or field) can be accessed from outside the class. A public method (or class) is visible and executable to the entire system, including the JVM startup sequence, while a private method is only executable from objects/methods within the defining class. This is, of course, implicit in the structure of “main” itself, since it must be executable as the first method of the overall program. Other variations
on the access flags can define a class, field, or method as “final,” as before; or even “abstract,” which means that the class itself is just an abstraction from which no instances can be made, and that one should use one of the subclasses of this class instead. Again, the details of these are more relevant to an advanced Java programmer than to an understanding of how the JVM itself works.

10.4 Objects

10.4.1 Creating objects as instances of classes

A class definition, by itself, is not usually very useful for performing computations. More useful are the instances of these “classes” as objects, actual examples of classes that can hold data, deal with method invocations, and in general, do useful stuff. For example, the “fraction” class defined above in the record section simply states that a fraction has two fields, a numerator and a denominator. An actual fraction would have a specific set of values in those fields that could be used in computation.

To create an instance of a class, it is first necessary to know the name of the class. The `new ExampleClassType` statement creates (allocates memory inside the computer for) a new instance of the `ExampleClassType` class, and pushes an address onto the method stack pointing to the new object. Merely making space is not enough, as it must also be initialized to a useful/meaningful state using one of the constructor methods defined for that type. To do this, it is necessary to use `invokespecial` with an appropriate method and set of arguments, as in figure 10.4.

Remember that the definition of the fraction class defined (using the standard boilerplate) a single constructor method `<init>()V`, which takes no arguments. Therefore, we construct our new fraction using the two lines

```
new fraction
invokespecial fraction/<init>()V
```

to create and initialize our fraction.

Actually, this is pretty dumb. The reason is that, although we have just created and initialized it, the `invokespecial` instruction pops our only reference to the fraction away when it returns. As a result, our newly allocated block of memory has just been lost, and is probably being garbage collected right now. In order to retain access to our new fraction object, we need to duplicate the address before calling `invokespecial` on it, as in figure 10.5. This same figure also gives an example of how data can be moved to and from the fields of an object.
; program to illustrate use of 'fraction' structure type

.class public fractionfront
.super java/lang/Object

; boilerplate
.method public <init>()V
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    new fraction
    invokespecial fraction/<init>()V
    return
.end method

Figure 10.4: Creating a “fraction” object
; second program to illustrate use of 'fraction' structure type

.class public fractionfront2
.super java/lang/Object

; boilerplate
 méthode public <init>()V
   aload_0
   invokespecial java/lang/Object/<init>()V
   return
.end method

.metodo public static main([Ljava/lang/String;)V
   .limit locals 2
   .limit stack 2

   ; create a new 'fraction' and store in local variable 1
   new fraction ; create a new 'fraction'
   dup ; duplicate to call <init>
   invokespecial fraction/<init>()V ; initialize
   astore_1 ; store new fraction

   ; assign the numerator the value 2
   aload_1 ; load the fraction
   iconst_2 ; push numerator value
   putfield fraction/numerator I ; place 2 in numerator (as int)
   ; n.b. re-storage of the fraction not needed!

   ; assign the denominator the value 2
   aload_1 ; load fraction (again)
   iconst_3 ; push denominator
   putfield fraction/denominator I ; place 3 in denominator (as int)

   ; print the numerator
   getstatic java/lang/System/out Ljava/io/PrintStream;
   aload_1 ; load the fraction
   getfield fraction/numerator I ; get and push the numerator
   invokevirtual java/io/PrintStream/print(I)V ; ... and print

   ; print a slash
   getstatic java/lang/System/out Ljava/io/PrintStream;
   ldc "/",
   invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V

   ; print(ln) the denominator
   getstatic java/lang/System/out Ljava/io/PrintStream;
   aload_1 ; load the fraction
   getfield fraction/denominator I ; get and push the denominator
   invokevirtual java/io/PrintStream/println(I)V ; ... and print(ln)

   return
.end method
10.4.2 Destroying objects

Object destruction, again, is handled by the garbage collection system and happens any time that no pointers to an object remain in accessible locations (such as in a local variable or on the stack).

10.4.3 The type Object

The fundamental type in any JVM system is named “Object”, or more completely `java/lang/Object`. As the root of the entire inheritance hierarchy, everything on the system is an Object in some form or another. As such, the properties of Objects are basic properties shared by everything in the system, and the methods of Objects are methods that can be called on anything. Not that these methods are particularly exciting, because they are so basic; for example, all Objects support an `equals()` method, which returns true if two objects are the same and false if they are different.

As the truly generic type, Objects can be used as general placeholders for data; a programmer could define (or more likely, use) a standardized List type that holds a collection of Objects, and then use this type to store his grocery list, his class schedule, and the win/loss record of a favorite team, without modification. Most of the standard data structures defined as part of the Java language are defined in this way, so that the data they hold, being an Object, imposes no restrictions on how to use the structures.

10.5 Class Files and .class File Structure

10.5.1 Class files

In a typical JVM system, each independent class is stored in a class file that maintains the necessary information for the use and execution of that particular class. Most of this information is fairly obvious — for example, the name of the class, its relationship to other classes in the inheritance hierarchy, and the methods defined by and characteristic of the class. The exact details of storage can be rather technical and can even vary from one JDK version to another, so if you have a specialist’s need for the details of class file format (for example, if you are writing a compiler that outputs JVM machine instructions), you should probably consult a detailed technical reference like the JVM references specifications themselves (Lindholm and Yellin, 1999). For a non-specialist, the following description (and the appropriate appendix, in a little more detail) will give something of the flavor of a class file.

In broad terms, a class file is stored as a set of nested tables. The top level table contains basic information regarding the class, such as the version number of the JVM for which it was compiled, the class name, and fundamental access properties. This table also contains a set of subtables, including a table of defined methods, fields, attributes, and direct interfaces that the class implements. Another
subtable contains the constant pool, which stores the fundamental constant values and strings used by the program. For example, if the value 3.1416 were needed by the class, instead of using the four-byte floating point value itself every place it were needed, this value would be placed in a small table of constants, and the table index would be used.

This has the overall effect of increasing the space efficiency of class files. Although there are obviously over four billion different floating point constants that a program might want to use, in practical terms, few programs will use more than a hundred or so. A constant pool of only two hundred entries can be addressed using a single-byte index, and thus save three bytes per constant access. Even a huge program that uses sixty thousand different constants can address any one of them with a two byte index. In addition to storing floating point constants, the constant pool will also hold integers, longs, doubles, and even objects such as String constants (like the prompt “Please enter your password”, which might be stored to be printed via a call to println). In fact, the ldc operation with which we are already familiar actually stands for load from the constant pool, and can be used (as we have seen) for almost any type.

10.5.2 Starting up classes

Before a class is actually available to be used by a running program, it must first be loaded from the disk, linked into an executable format, and finally initialized to a known state. This process is one of the fundamental services that a JVM program must provide, in the form of the primordial class loader, an instantiation of the standard-defined type java.lang.ClassLoader. The JVM designer has a certain amount of leeway in exactly what services the primordial class loader can provide, above a certain minimum level.

For example, to load a class, the loader usually has to be able to find the class on local storage (usually by adding the suffix .class to the name of the class), read the data stored there, and produce an instance of java.lang.Class to describe that class. In some circumstances (like a Web browser or a sophisticated JVM implementation), one may need to pull individual classes out of an archive, or download appropriate applets across the network. The primordial loader must also understand enough of the class structure to pull out the superclasses as needed; if the class you have written extends Applet, then JVM needs to understand Applets and their properties to run your program correctly. It is the task of the linker (which is also usually grouped into the class loader) to connect (link these different classes into a suitable runtime representation.

Another important task of the JVM class loader is to verify that the bytes in the bytecode are actually safe to execute. Trying to execute an integer multiplication when there are no integers, on the stack, would not be safe. Trying to create a new instance of a class that doesn’t exist would not be safe. The verifier is responsible for enforcing most of the security rules discussed throughout the text. Finally the class is initialized by calling the appropriate routines to set static fields to appropriate values and otherwise make the class into a fit state for execution.
10.6 Class Hierarchy Directives

This section can be skipped without loss of continuity, and pertains mainly to advanced features of the Java class hierarchy.

In the real world, there can be problems with the strict type hierarchy that the JVM supports. Because each subclass can have only one superclass, each object (or class) will inherit at most one set of properties. Real life is rarely that neat or clean. For example, one fairly obvious hierarchy is the standard set of biological taxa. A dog is a mammal, which in turn is a vertebrate, which in turn is an animal, and so forth. This could be modelled easily in the JVM class hierarchy by making Dog a subclass of Mammal, and so forth. However, in practice, Dog also inherits a lot of properties from the Pet category as well, a category that it shares with Cat and Hamster, but also with Guppy, Parakeet, and Iguana (and excludes Bear, Cheetah, and other non-Pet mammals). So we can see that, whatever category Pet is, it crosses the lines of the standard biological taxonomy. And because there’s no way to inherit from more than one class, there’s no easy way within the class structure to create a class (or an object) that is both a Mammal and a Pet.

Java and the JVM provide a process to cover these sorts of regularities via the mechanism of **interfaces**. An interface is a special sort of class-like Object (in fact, it’s stored in a file of identical format, only with a few access flags change) that defines a set of properties (fields) and functions (methods). Individual objects are never instances of an interface, but instead they implement an interface by explicitly encoding these properties and functions within their classes. For example, it may be decided that among the defining properties of a Pet is the function of having a Name and an Owner (presumably these would be access methods returning some sort of string value), and thus appropriate methods to determine what the name and owner actually are. Interfaces never actually define the code for a method, but do define methods that a programmer is required to implement. Similarly, although interfaces can define fields, fields defined as part of an interface must be both **static** and **final**, reflecting the fact that an individual object is never an instance of an interface — and thus has no storage space available by virtue of the fields it implements.

At this point, a skilled programmer can define the Dog class such that it inherits from (extends) the Mammal class, and thus gets all the properties associates with Mammals. She can also define the class as “implementing” the Pet interface, by making sure that, among the methods she writes in his Dog class are the appropriate methods required of every Pet.

She would then define in her class file not only that the class Dog had Mammal as a superclass, but also that it implemented the Pet interface. This would involve a new directive, as follows

```
.class public Dog
.super Mammal
.implements Pet
```

The declaration and use of the Pet interface would be very similar to writing a
normal class file, but with two major differences. First, instead of using the .class directive to define the name of the class, the programmer would use the .interface directive in the same way.

```
.interface public Pet
```

Second, the methods in the Pet.j files would have method declarations, but no actual code associated with them (implying that they are abstract).

```
.method public abstract getName()Ljava/lang/String;
.end method
.method public abstract getOwner()Ljava/lang/String;
.end method
```

There are also a few minor differences in use between interface types and class types. First, interface types cannot be directly created as objects, although one can certainly create objects that implement interface types. In the example above, although creating a Dog is legal, creating a Pet directly is not.

```
new Dog ; this is fine
new Pet ; this would produce an error
```

However, interfaces are acceptable as arguments types and return values of methods. The following code will accept any valid object whose class implements the Pet interface.

```
invokespecial Dog/isFriendlyTo(LPet;)I
```

and will tell you, for example, whether or not any particular Dog is friendly to (e.g.) a particular Iguana. Finally, if you find yourself in a position where you have an Object that implements a particular interface, but you don’t know exactly what class it is (as in the isFriendlyTo example above), the JVM provides a basic instruction `invokeinterface` to allow interface methods to be invoked directly, without regard to the underlying class. The syntax of `invokeinterface` is similar to the other `invoke?` instructions, but slightly more complicated — and usually slower and less efficient than the other method invocation instructions. Unless you have a specific need for interfaces, it’s perhaps best left to specialists.

### 10.7 An Annotated Example: Hello, World revisited

At this point, we are now at a position to give a detailed explanation and understanding of every line in the first jasmin example given, including the so-called “boilerplate”:
.class public jasminExample
A class directive that specifies the name of the current class
.super java/lang/Object
... and where it fits into the standard hierarchy (specifically, as a subclass of java/lang/Object)
.method public <init>()V
This is the method for constructing an element of type jasminExample. It takes no arguments, and returns nothing, with name <init>.
aload_0

 invokevirtual java/lang/Object/<init>()V
To initialize a jasminExample, we load the example itself and make sure it is successfully initialized as an Object, first.
return
No other initialization steps are needed, so quit the method.
.end method
This directive ends the method definition.

.method public static main([Ljava/lang/String;)V
This is the “main” method, called by the java program itself. It takes one argument, an array of Strings, and returns nothing. It is defined both public and static, so it can be called from outside the class without any objects of that class existing.
.limit stack 2
We need two stack elements (one for System.out, one for the String)
  getstatic java/lang/System/out Ljava/io/PrintStream;
Get the (static) field named “out” from the System class; it should be of type PrintStream, defined in the java.io package.
  ldc "This is a sample program."
Push the string to be printed (more accurately, pushes the index of that string in the constant pool).
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
Invoke the “println” method on System.out.
  return
Quit the method.
.end method
This directive ends the method definition.

10.8 Input and Output : An Explanation

10.8.1 Problem statement

The Java programming language defines much more than the syntax of the language itself. Key to the acceptance and widespread use of the language has been the existence of various packages to handle “difficult” programming aspects, such as input and output. The java.io package, for example, is a set of packages specif-
ically designed to handle system input and output at a sufficiently abstract level to be portable across platforms.

Using input and output peripherals is often one of the most difficult parts of any computer program, especially in assembly language programming. The reason, quite simply put, is the bewildering variety of devices and ways in which these devices can work.

There is a big difference, at least in theory, between devices like disk drives where the entire input is available at any one time, and devices like a network card where the data is available only on a time-sensitive and limited basis. Not only are there different kinds of devices available, but even within one broad category, there might be subtle but important differences, such as the placement of keys on a keyboard, the presence or absence of a second mouse button, and so forth. However, from the point of view of the person writing a user program, these differences not only don’t matter, but would be actively confusing. An electronic mail program, for instance, has the primary job of reading input in (from the user), figuring out where and to whom the mail should go, and sending it on its merry way. The details of how the data arrives — does it come in bursts over an Ethernet connection? in single keystrokes from an attached keyboard? as a massive chunk of data arriving through a cut-and-paste operation? as a file on the hard disk? — don’t (or at least, shouldn’t) matter to the mail program.

Unfortunately, at the level of basic machine instructions, these differences can be crucial; reading data from an Ethernet card is not at all the same as reading data from the disk or reading data from the keyboard. The advantage of a class-based system such as the one supported by the JVM is that the classes themselves can be handled, and unified, so that the programmer’s task is made a little less confusing.

10.8.2 Two systems contrasted

General peripheral issues

To illustrate this sort of confusion, consider the task of reading data and printing it somewhere (to the screen or the disk). For simplicity, assume that the input device (the spot where the data is coming) is either an attached keyboard, or an attached disk. Without going into too much detail, a keyboard is just a complicated kind of switch — when someone presses down on a key, there are certain electrical connections made that allow the computer to determine which key was pressed (and hence which character was intended) at that time. A disk is a gadget for information storage — logically, you can think of it as a huge array of “sectors,” where each sector holds some fixed number of bytes (usually 512 bytes, but not always). Actually, it’s a little more complicated, since the information is actually ordered as a set of multiple platters (each of which looks kind of like a CD made out of magnetic tape), each platter having several numbered concentric tracks, and each track being divided into several sectors like slices of pizza. To read or write a sector of data, the computer needs to know the platter number, track number, and sector within the track — but most of the time (whew!) the hard drive controller
10.8. INPUT AND OUTPUT: AN EXPLANATION

will take care of this.

It’s important to notice a few crucial differences between these two gadgets. First, when you read from the keyboard, you will get at most one character of information, since you only learn the key that’s being pressed at that instant. Reading from a disk, by contrast, gives you an entire sector-full of data at once. It’s also possible to read ahead on a disk, and see what the next sector contains, but entirely impossible on a keyboard.

Similar differences hold depending upon the exact details of the screen to which we print. If there is a window manager running on the computer, then (of course) we will actually print to an individual window, and not to “the screen” as we would in text mode. In text mode, for example, we always know that printing starts at the left hand side of the physical screen, while we have no idea where the window is at any given time, and it might even be moving while we’re trying to print. And, of course, printing to the screen is entirely different from writing data to the disk, with all the structure defined above.

As will be seen, the class structure of the JVM allows much of this confusion to be avoided through the use of a proper class structure, unlike another common system, the Intel Pentium, running Windows 98.

The Intel Pentium

Every device you attach to a Pentium comes with a set of device drivers. This is true, in fact, for every computer; the drivers define how to interact with the hardware. They are a little bit like classes and methods in this regard, except that they don’t have a useful and unified interface. One of the simplest set of device drivers on any Windows box is the BIOS (Basic Input-Output System) shipped as part of the operating system. (Actually, the BIOS predates the actual Pentium chip by nearly 20 years, but the functionalism, a holdout from the original IBM-PC, is still around.)

On the Intel Pentium, there is a single machine instruction (INT) used to transfer control to the BIOS; when this is executed, the computer inspects the value stored in a particular location (the AX register, if you must know) to determine what, exactly should be done. If the value stored is 0x7305, the computer will access the attached disk. To do this properly, it inspects a number of other values stored in other registers. Among these values is the number of the disk sector of interest, a location of memory to place the new data, and whether the disk should be read from or written to. In either case, at least one sector of data will be transferred.

The same instruction that transfers control to the BIOS will also cause the computer to read a character — if the value stored in the lower half of the AX register is a 0x01. Actually, it’s a little more complicated than that. If the value stored is 0x01, the computer will wait until a character is pressed, and return that character (and will also print the same character to the screen). If the value stored is 0x06, then the computer will check to see if a key is pressed this instant, and return it (without printing). If no character is being pressed, then nothing is returned (and a special flag is set to indicate that). Both of these functions only read one
character at a time. To read lots of characters at once, if they are available in a
typeahead buffer, use the stored value 0x0A, which does not return the characters,
but instead stores them somewhere in main memory.

Output has similar issues of detail: to write to a disk, one uses the same
BIOS operation as to read, while to write to the screen in text mode or a window
system require two different basic operations (and different from the read opera-
tions/values).

All this is after the device drivers and hardware controllers have “simplified”
the task of accessing the peripherals. The fundamental problem is that the various
sorts of hardware are too different from each other. It would be possible for a
programmer to write a special purpose function whose sole job is, for instance, to
handle input — if input is supposed to be read from the keyboard, use operation
0x01, if from a disk, use operation 0x7305, and in either case, move the value read
into a uniform place and format. Such a program may or may not be difficult to
write, but it requires attention to detail, hardware knowledge, and time that not
every programmer wants to spend.

This mess is part of why Windows, the operating system, exists. Part of the
functionality that Windows assumes is to provide exactly this kind of broad-brush
interface for the programmer. The programmers at Microsoft have taken the time
to write these detailed, case-laden functions.

The Java virtual machine

By contrast, in a properly designed and constructed class-based system, this sort
of unification is delivered by the class structure itself. In Java (and by extension in
the JVM), most input and output is handled, properly, through the class system.
This allows the classes to take care of information hiding, and only present the
important, shared, properties via methods.

To briefly review: the java.io.* package as defined for Java 1.4 provides a
variety of different classes, each of which present different properties for reading
and writing. The most useful class for input, and the one most often used, is
the BufferedReader class. This class is a fairly-powerful, high-level class that
allows reading from many sorts of generalized “stream” objects. Version 1.5 of
Java includes additional classes such as a Scanner that are handled in a similar
manner.

Unfortunately, the keyboard lacks many properties that are typical of these
BufferedReader objects, but the class system provides a way to construct a Buffere-
dReader object from other sorts of objects. The standard Java libraries do provide
an object, a field in the System class, called System.in that usually attaches to the
keyboard. This field is defined, however, to hold one of the lowest, most primitive
types of input objects — a java.io.InputStream. (Some of the key differences be-
tween an InputStream and a BufferedReader include the absence of buffering and
the inability to read any type other than a byte array).

Similarly, the java.io.* package provides a special type for reading from the
disk, either as a FileInputStream or as a FileReader, both of which can be used
10.8. INPUT AND OUTPUT : AN EXPLANATION

import java.io.*;

class jvmReaderExample {
    public static void main(String[] args) throws IOException {
        InputStreamReader i = new InputStreamReader(System.in);
        BufferedReader b = new BufferedReader(i);

        String s = b.readLine();
        System.out.println(s);
    }
}

Figure 10.6: Sample Java program to read and echo a line from the keyboard to construct a BufferedReader. Once this is performed, access to a file is identical to access to the keyboard, since both use the identical methods that comprise the BufferedReader class.

In the following section, we will construct a program (using the more widely available Java 1.4 semantics) to read a line from the keyboard and to copy that line to the standard output. Although still complex, the complexity lies all in construction of the BufferedReader and not in any of actual data reading. For a simple one-time cost of object construction, any data can be read through a BufferedReader, while the corresponding program on the Intel would need special cases and magic numbers at every i/o operation.

10.8.3 Example : Reading from the Keyboard in the JVM

An example of code to perform this task in Java is presented as figure 10.6. Note that two conversions are needed, first to construct an InputStreamReader from the InputStream, and second to construct a BufferedReader from the InputStream-Reader. In fact, there is even a bit of complexity hidden in the Java code, since the actual constructor function for the BufferedReader is defined as

    public BufferedReader(Reader in)

    meaning that one can construct a BufferedReader out of any sort of Reader, of which InputStreamReader is merely one subclass.

Object construction is handled as before in two steps. First, the new object itself must be created (via the new instruction), and then an appropriate initialization method must be invoked (via invokespecial. This program will actually require that two new objects be created, an InputStreamReader and a BufferedReader. Once these have been created, the BufferedReader class defines a standard method (called “readLine”) that will read a line of text from the keyboard and return it as
a String (Ljava/lang/String;). Using this method, we can get a string and then print as usual through System.out’s println method.

10.8.4 Solution

.class public jvmReader
.super java/lang/Object

; boilerplate needed for object creation
.method public <init>()V
   aload_0
   invokespecial java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
 .limit stack 4

; create a new object of type InputStreamReader
new java/io/InputStreamReader

; initialize constructor from System.in (InputStream)
dup
getstatic java/lang/System/in Ljava/io/InputStream;
invokespecial java/io/InputStreamReader/<init>(Ljava/io/InputStream;)V
; equivalent to new InputStreamReader(InputStream)

; now create a new BufferedReader
new java/io.BufferedReader

; duplicate it and put it underneath InputStreamReader
dup_x1

; duplicate again and put it underneath InputStreamReader
dup_x1
; stack now holds BR, BR, ISR, BR

; eliminate unneeded BufferedReader
pop

; call constructor of BufferedReader using InputStreamReader
invokespecial java/io/BufferedReader/<init>(Ljava/io/Reader;)V
; initialized BufferedReader now at top of stack

; invoke readline method to get a string (and leave at top of stack)
10.9. Example: Factorials via Recursion

10.9.1 Problem Statement

As a final example, we present code to do recursion (where one function or method invokes itself) on the JVM using the class system. The factorial function is a widely-used mathematical operation in combinatorics and probability. For example, if (for some bizarre reason) someone wants to know how many different ways there are to shuffle a (52-card) deck of cards, the answer is 52!, or 52 · 51 · . . . · 1. One nice property of factorials is that they have an easy recursive definition in that

\[ N! = N \cdot (N - 1)! \text{ for } N \geq 1, \text{ and } 0! = 1 \]

Using this identity, we can construct a recursive method that accepts an integer (as a value of \( N \)) and returns \( N! \).

10.9.2 Design

The pseudocode for solving such a problem is straightforward, and in fact, presented as an example in most first-year programming texts.

To calculate \( N! \):

\[
\begin{array}{l}
\text{if } (N \leq 0) \text{ then} \\
\quad \text{return } 1 \\
\text{else begin} \\
\quad \text{recursively calculate } (N-1)! \\
\quad \text{multiply by } N \text{ to get } N \cdot (N-1)! \\
\quad \text{return } N \cdot (N-1)! \\
\end{array}
\]

(Strict mathematicians will no doubt note that this pseudocode also defines the factorial of a negative integer as 1, as well.)

In addition, a (public, static) main routine will be needed to set the initial value of \( N \) and to print the final results. Because the main method is static, if the factorial method is not static, we would need to create an object instance of the appropriate class; if we defined the factorial method as static, instead, it can just be invoked directly.
10.9.3 Solution

A worked-out solution to calculate 5! is presented here. Similar code would work to solve almost any recursively-defined problem.

.class public factorialCalculator
.super java/lang/Object

; boilerplate needed for object creation
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V
  return
.end method

.method public static main([Ljava/lang/String;)V
  ; we need two stack elements, one for System.out, one for the string
  .limit stack 2
  bipush 5          ; push 5 to calculate 5!
  invokestatic factorialCalculator/fact(I)I
  ; 5! is now on stack to be calculated

  ; get and push System.out
  getstatic java/lang/System/out Ljava/io/PrintStream;

  ; put System.out and 5! in right order
  swap

  ; invoke the PrintStream.println method
  invokevirtual java/io/PrintStream/println(I)V

  return
.end method

.method static fact(I)I
  .limit stack 2
  iload_0 ; check if argument <= 0
  ifle Exit ; if so, return 1 immediately

  iload_0 ; push N
  iinc 0 -1 ; compute (N-1)
  iload_0 ; load (N-1)
10.10 Chapter Review

- The JVM, because of its strong association with the object-oriented programming language Java, provides direct support for object-oriented programming and the use of user-defined “class” types. Both array and object references are supported through the use of the basic type address.

- An array is a collection of identical type, indexed by a integer(s). There are individual machine-level instructions to create, read from, or write to single and multidimensional arrays, as well as to get the length of an array.

- When arrays (or any data) are no longer useful or accessible, the JVM will automatically reclaim the used memory via garbage collection and make it available for reuse.

- The JVM also supports records as collections of named fields and classes as records with defined access methods. It also supports interfaces as collections of abstract methods. The class file is the basic unit of program storage for the JVM and incorporates all three of these types.

- Fields are accessed via getfield and putfield instructions; static (class) fields use getstatic and putstatic instead.

- Classes are accessed via one of four invoke? instructions, depending the class and method to be involved.

- Objects are created as instances of classes by the new instruction. Every class must include appropriate constructor functions (typically named <init>) to initialize a new instance to a sane value.

- Access to the outside world via I/O primitives is accomplished on the JVM through a standardized set of classes and methods. For example, System.in is a static field of type InputStream whose properties and access methods are defined by the standards documents. Reading from the keyboard can be accomplished by invoking the proper methods and creating the proper classes using System.in as a base.
Recursion can also be supported using the class system by the creation of new sets of local variables at each new method invocation. This differs from the previous `jsr/ret` techniques, as well as from the techniques employed on systems like the Pentium or PowerPC, where new local variables are created on stack frames.

10.11 Exercises

1. How does the implementation of user-defined types on the JVM differ from other machines like the 8088 or PowerPC?

2. How would space for a local array be created on a PowerPC? How would the space be reclaimed? How do these answers differ on a JVM?

3. An array element is typically accessed via index mode on a typical computer like the Pentium. What does the JVM use instead?

4. How are standard methods like String.toUpperCase() incorporated into the JVM?

5. What is the difference between a field and a local variable?

6. What is the difference between `invokevirtual` and `invokespecial`?

7. Why do static methods take one fewer arguments?

8. What is the corresponding type string for the following methods?
   
   (a) float toFloat(int)
   (b) void printString(String)
   (c) float average(int, int, int, int)
   (d) float average(int[])
   (e) double[][] convert(long[][])
   (f) boolean isTrue(boolean)

9. What is special about the `<init>` method?

10. What is the fourth byte in every class file (see appendix D)?

11. Approximately how many different String values could a single method use?

10.12 Programming Exercises

1. Write a `jasmin` program using the Java 1.5 `Scanner` class to read a line from the keyboard and echo it.

2. Write a `jasmin` program using the Java AWT (or similar graphics package) to display a copy of the Jamaican flag on the screen.
3. Write a jasmin program to determine the current date and time.

4. Write a Time class to support arithmetic operations on times. For example, 1:35 + 2:15 is 3:50, but 1:45 + 2:25 is 4:10.

5. Write a Complex class to support addition, subtraction, and multiplication on complex numbers, such as $3 + 4i$.

6. The Fibonacci sequence can be recursively defined as follows: the first and second element of the sequence have the value 1. The third element is the sum of the first and second, or 2. In general, the $N$-th Fibonacci number is the sum of the $N-1$st and the $N-2$nd. Write a program to read a value $N$ from the keyboard and to (recursively) determine the $N$th Fibonacci number. Why is this an inefficient way to solve this problem?

7. Write a program (in any language approved by the instructor) to read a class file from the disk and print the number of elements in the constant pool.

8. Write a program (in any language approved by the instructor) to read a class file from the disk and print the names of the methods defined in it.
Appendix A

Digital Logic

A.1 Gates

Without the transistor, the modern computer would not be possible. As suggested by Moore’s Law (transistor density doubles every eighteen months), the ability to fabricate and arrange transistors is the fundamental way that data is controlled, moved, and processed within the chips at the heart of a computer.

![NPN transistor symbol]

Figure A.1: Sample transistors and symbols

Electronically speaking, a transistor can be regarded as a kind of electronically controlled switch. A typical transistor and its electronic diagram are shown in...
figure A.1. Under normal circumstances, electricity flows from the emitter to the collector, like water through a pipe or cars through a tunnel. However, this is only possible as long as appropriate application of an electrical signal to the base permits the flow of electricity to pass. Without this control signal, it’s as though someone had turned a faucet (or set up a traffic light). When this happens, electricity can’t get through. By combining these switches in combination, engineers can create dependency structures — electricity will flow only if all transistors are energized, for example.

Figure A.2: Transistors in “series” (top) and “parallel” (bottom)

Figure A.2 shows two examples of dependency structure. In the series circuit, both transistors share a common path, and electricity must be able to flow through both circuits at the same time if it is to flow at all from point A to point B. In the parallel circuit, each transistor has its own current path, and any one can independently allow current to flow across the circuit.

The electrical circuit shown in figure A.3 is an example of two transistors connected in series. In order for electricity to flow from the power source (\(V_{cc}\)) to the output, both of the transistors need to be signaled to allow current to pass. If either transistor is an open switch, then electricity can’t flow. This implies that electricity can flow (there is power at the output) only if transistor 1 is closed AND transistor 2 is closed. Similarly, figure A.4, two transistors in parallel, will allow current to flow if either the first transistor OR the second transistor is closed (or both).

The basic building blocks of a computer consist of simple circuits, called gates, that implement this kind of simple logical signals. These gates typically contain between one and six transistors each and provide an implementation of a very basic
A.2. COMBINATIONAL CIRCUITS

Figure A.3: Simplified AND gate and symbol

Logical or arithmetic operations. Remember that the basic values used in logic are just True and False. If we consider “current is flowing” to represent True, then the circuit in figure A.3 would be an implementation of the logical function AND in a simple (and somewhat idealized — don’t try to build this at home out of Radio Shack components!) gate. Other functions available include OR (figure A.4), NOT, NAND, and NOR. The symbols used for drawing various gates are given in figure A.5. Note that each gate (except NOT) has two inputs and a single output. Also noted that the NOT gate symbol has a circle (representing signal inversion) at the output line. The symbol for a NAND (Not-AND) gate is the same as the symbol for an AND gate, but with the little inversion circle at the output. Similarly, a NOR (Not-OR) gate is just an OR gate with the inversion circle.

A.2 Combinational circuits

Complicated networks of gates can implement any desired logical response. For example, the XOR (eXclusive OR) function is True if and only if exactly one, but
not both, of the inputs is True. In logical notation, this can be written as:

\[
A \text{ XOR } B = (A \text{ OR } B) \text{ AND } (\text{ NOT } (A \text{ AND } B))
\]

As a truth table, this can be written as in table A.1. And finally, as a circuit, as figure A.6.

The basic concept of combinatorial circuits is that the output is always a function of the current input signal(s), without regard to signal history. Such circuits can be very useful for implementing simple decisions or arithmetic operations. Although a full description of how to design such circuits is beyond the scope of this appendix, figure A.7 shows how a set of gates could implement simple binary addition (as part of the ALU of a computer.)

Specifically, this circuit will accept two single-bit signals (A and B) and output both their sum, and whether or not there is a carry. (This circuit is sometimes called a “half adder.”) Examination of the binary addition table shows that the sum of two bits is simply their XOR, while a carry is generated if and only if both bits are ones, or in other words, their AND. Similar analysis can yield a more complicated design capable of incorporating a carry bit from a previous addition stage (a “full adder,”), or adding several bits at one time (as in a register). Figure A.9 shows...
A.2. COMBINATIONAL CIRCUITS

Figure A.5: Gate types and symbols

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Table A.1: Truth table defining “exclusive or”
Figure A.6: Implementation of "exclusive or" in gates
Figure A.7: Single-bit half adder
how a four-bit addition can be performed by four replications of the full adder circuit; thirty-two replications, of course, could add together two registers on the Pentium. One can also build a circuit to perform binary multiplication, and so forth. Multiply these simple gates by several billion, and you approach the power and complexity of a Pentium.

### A.3 Sequential circuits

In contrast to combinatorial circuits, a sequential circuit retains some notion of memory and of the history of the circuit. The output of a sequential circuit depends not only on the present input, but also what its inputs have been in the past. Another way of expressing this is that such a circuit has an internal state. By using the internal state, one can store information for later use, essentially creating the sort of memory necessary for a register.

One of the simplest sequential circuits is the S–R flip-flop, the circuit illustrated in figure A.10. To understand this circuit, let’s first pretend that S and R are both...
Figure A.9: Cascaded 4-bit adder, including internal carries
Figure A.10: SR flip-flop, a circuit with memory

0 (False), that Q is 0, and $\overline{Q}$ is 1 (True). Since $\overline{Q}$ is True, R NOR $\overline{Q}$ is 1. Similarly, S NOR Q is 1. We thus see that this configuration is internally consistent and will remain stable as long as the individual components work (which usually means as long as they have power). A similar analysis will show that if S and R are both 0, while Q is 1, will also yield a self-consistent, stable state.

This simple flip-flop can be used as a one-bit memory; the value of Q is the value stored in the memory. The signals S and R can be used to set or reset (set to 1 or 0) the flip-flop, respectively. Observe that if S becomes 1, this will force the output of the upper NOR gate to be zero. With R 0 and $\overline{Q}$ 0, the output of the lower NOR gate is 1, and so the value stored is now 1. A similar process, if you set R to 1, forces $\overline{Q}$ to 1 and Q to 0.

Unfortunately, if S and R are both 1, then bad things happen. According to the notation, Q and $\overline{Q}$ should always be opposites of each other, but if both inputs are 1, both outputs will be zero. (you can confirm this for yourself). In a purely mathematical sense, we can regard this as the logical equivalent of dividing by zero or something — something to be avoided instead of analyzed. With real, physical circuits, there is inevitably a delay in the response of a gate (if R goes high, it
takes a moment for \( Q \) to respond by going high) or even in the propagation of an electrical signal along a wire. Putting both control signals high will thus result in an unpredictable state for both \( Q \) and \( \bar{Q} \). Similarly, even a brief power spike on one input wire can cause the flip-flop to unexpectedly change state.

However, since this is to be avoided, there are other sorts of sequential circuits more commonly used in computers. Most common circuits combine the idea of control signals with timing signals (usually called a \textit{clock signal}) to synchronize the control signals and keep short-term fluctuations from influencing the circuit’s memory. Notice that in the clocked flip-flop diagram, the clock signal acts to enable the flipflop to change state. If the clock signal is low, then changes on \( S \) or \( R \) cannot affect the memory state of the circuit.

We can extend this clocked flip-flop further to something perhaps more useful as well as safer. The circuit diagram in A.12 illustrates a D flip-flop. As you can see, this circuit has only one input beyond the clock. The D input is tied to the \( S \) input of the flip-flop, while \( \bar{D} \), the complement of D, is tied to the \( R \) input. This keeps both \( S \) and \( R \) from being 1 at the same time. When a clock pulse happens, the value of \( D \) will be stored in the flip-flop (if the value of \( D \) is 1, then \( Q \) becomes
APPENDIX A. DIGITAL LOGIC

Figure A.12: A D flip-flop

1, if the value of D is 0, Q becomes 0). Until a clock pulse occurs, changes in D will have no effect on the value of the flip-flop. This makes a D flip-flop very valuable for copying and storing data, for example, in moving a reading from an I/O device to a register for later use.

Another variation on the SR flip-flop yields the T flip-flop (figure A.13). Like the D flip flop, this is a single-input extension of the clocked SR flip-flop, but the additional feedback wires control how the input/clock pulse is gated through; in this circuit, the flip-flop will change state (“toggle”) each time the input is triggered. For example, if Q is 1 (and $\bar{Q}$ 0, by extension), then the next pulse will trigger the bottom input wire (essentially the R input in the previous circuits), and reset the flip-flop so that Q is 0.

A.4 Computer operations

Using these building blocks, it’s possible to build the higher-level structures associated with computer architectures. In particular, a collection of simple one-bit
Figure A.13: A T flip-flop
flip-flops such as the S-R flip-flop described earlier can implement a simple register and hold data until such time as it is needed. To add, for example, two one-bit registers, the Q output of each register can be electrically connected to one input of a simple addition circuit. The Q output of the other register would be connected to the other adder input. The resulting output bit would be the addition of these two register bits, which could be captured and stored (via a D flip-flop) in yet another register. The T flip-flop could be used in conjunction with adder circuits to build a simple pulse counter. Of course, these descriptions oversimplify dreadfully, but they give something of the feel for the task presented to the computer designer.
Appendix B

JVM Instruction Set

**Sample mnemonic in context** (Numeric opcode value in hex) — description
Summary: This opcode doesn’t exist but illustrates the entry format. To the right are the required initial and final stack states as needed/created by the operation in question. Note that long and double types take two stack slots, as shown.

### aaload (0x32) — Load value from array of addresses
Summary: Pops an integer and an array of addresses (object references) from the stack, then retrieves a value from that location in the 1-dimensional array of addresses. The value retrieved is pushed on the top of the stack

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>float</td>
</tr>
</tbody>
</table>

### aastore (0x53) — Store value in array of addresses
Summary: Stores an address (array reference) in an array of such addresses. The top argument popped is the index defining the array location to be used. The second argument popped is the address value to be stored, and the third and final argument is the array itself.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>—</td>
</tr>
</tbody>
</table>

301
APPENDIX B. JVM INSTRUCTION SET

aconst_null (0x1) — Push array constant null
Summary: Pushes the machine-defined constant value null as an address onto the operand stack
Initial: —
Final: address(null)

aload <varnum> (0x19 [byte/short]) — Load address from local variable
Summary: Loads address (object reference) from local variable #<varnum> and pushes the value. The value of <varnum> is a byte in the range 0..255 unless the wide operand prefix is used, in which case it is a short in the range 0..65536. Note that subroutine return locations cannot be loaded from stored locations, via aload or any other opcode.
Initial: —
Final: address

aload_0 (0x2a) — Load address from local variable #0
Summary: Loads address (object reference) from local variable #0, and pushes the value loaded onto the stack. This is functionally equivalent to aload 0 but takes fewer bytes and is faster.
Initial: —
Final: address

aload_1 (0x2b) — Load address from local variable #1
Summary: Loads address (object reference) from local variable #1, and pushes the value loaded onto the stack. This is functionally equivalent to aload 1 but takes fewer bytes and is faster.
Initial: —
Final: address

aload_2 (0x2c) — Load address from local variable #2
Summary: Loads address (object reference) from local variable #2, and pushes the value loaded onto the stack. This is functionally equivalent to aload 2 but takes fewer bytes and is faster.
Initial: —
Final: address

aload_3 (0x2d) — Load address from local variable #3
Summary: Loads address (object reference) from local variable #3, and pushes the value loaded onto the stack. This is functionally equivalent to aload 3 but takes fewer bytes and is faster.
Initial: —
Final: address
anewarray `<type>` (0xbd [int]) — Create unidimensional array of objects
Summary: Allocates space for an `<t>`-dimensional array of type `<type>` and pushes a reference to the new array. The type is stored in bytecode as a four-byte index into the constant pool. The size of the new array is popped as an integer from the top of the stack.

Initial : int(size)
Final : address(array)

anewarray_quick (0xde) — Quick version of `anewarray` opcode
Summary: Optimized version of `anewarray` opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a `.class` file that isn’t currently loaded and being executed.

Initial : see original opcode
Final : see original opcode

areturn (0xb0) — Return from method with address result
Summary: Pops an address (object reference) from current method stack. This object is pushed onto the method stack of the calling environment. The current method is terminated and control is transferred to the calling environment.

Initial : address
Final : (n/a)

arraylength (0xbe) — Take length of an array
Summary: Pops an array (address) off the stack and pushes the length associated with that array (as an integer). For multidimensional arrays, the length of the first dimension is returned.

Initial : address(array ref)
Final : int

astore `<varnum>` (0x3a [byte/short]) — Store address in local variable
Summary: Pops address (object reference or subroutine return location) from top of stack and stores that address in local variable `#<varnum>`. The value of `<varnum>` is a byte in the range 0..255 unless the wide operand prefix is used, in which case it is a short in the range 0..65536.

Initial : address
Final : —

astore_0 (0x4b) — Store address in local variable #0
Summary: Pops address (object reference) from top of stack and stores that address value in local variable #0. This is functionally equivalent to `astore 0`, but takes fewer bytes and is faster.

Initial : address
Final : —
astore_1 (0x4c) — Store address in local variable #1
Summary: Pops address (object reference) from top of stack and stores that address value in local variable #1. This is functionally equivalent to a store 1, but takes fewer bytes and is faster.

astore_2 (0x4d) — Store address in local variable #2
Summary: Pops address (object reference) from top of stack and stores that address value in local variable #2. This is functionally equivalent to a store 2, but takes fewer bytes and is faster.

astore_3 (0x4e) — Store address in local variable #3
Summary: Pops address (object reference) from top of stack and stores that address value in local variable #3. This is functionally equivalent to a store 3, but takes fewer bytes and is faster.

athrow (0xbf) — Throw an exception/error
Summary: Pops an address (object reference) and “throws” that object as an exception to a pre-defined handler. The object must be of type Throwable. If no handler is defined, the current method is terminated and the exception is rethrown in the calling environment. This process is repeated until either a handler is found or there are no more calling environments, at which point the process/thread terminates. If a handler is found, the object is pushed onto the hand handler’s stack and control is transferred to the handler.

baload (0x33) — Load value from array of bytes
Summary: Pops an integer and an array from the stack, then retrieves a value from that location in the 1-dimensional array of bytes. The value retrieved is converted to an integer and pushed on the top of the stack. The baload operand is also used to load values from a boolean array, using similar semantics.
**bastore** (0x54) — Store value in array of bytes  
Summary: Stores an 8-bit byte in a byte array. The top argument popped is the index defining the array location to be used. The second argument popped is the byte value to be stored, and the third and final argument is the array itself. The second argument is truncated from an int to a byte and stored in the array.  
The *bastore* operand is also used to store values in a boolean array, using similar semantics.

```
Initial : int(index)  
int(byte or boolean)  
address(array ref)  
Final : —
```

**bipush <constant>** (0x10 [byte]) — Push [integer] byte  
Summary: The byte value given as an argument (-128..127) is sign-extended to an integer and pushed on the stack

```
Initial : —  
Final : int
```

**breakpoint** (0xca) — Breakpoint (reserved opcode)  
Summary: This opcode is reserved for internal use by a JVM implementation, typically for debugging support. It is illegal for such an opcode to appear in a .class file, and such a .class file will fail verification.

```
Initial : (n/a)  
Final : (n/a)
```

**caload** (0x34) — Load value from array of bytes  
Summary: Pops an integer and an array from the stack, then retrieves a value from that location in the 1-dimensional array of characters. The value retrieved is converted to an integer and pushed on the top of the stack

```
Initial : int(index)  
address(array ref)  
Final : int
```

**castore** (0x55) — Store value in array of characters  
Summary: Stores a 16-bit UTF-16 character in an array of characters. The top argument popped is the index defining the array location to be used. The second argument popped is the character value to be stored, and the third and final argument is the array itself. The second argument is truncated from an int to a character and stored in the array.

```
Initial : int(index)  
int(character)  
address(array ref)  
Final : —
```
checkcast <type> (0xc0 [constant pool index]) — Confirm type compatibility

Summary: Examines (but does not pop) the top element of the stack to confirm that it is an address (object or array reference) that can be cast to the type given as an argument — in other words, that the object is either null, an instance of <type> (see instanceof), or of a superclass of <type>. In bytecode, the type is represented as a two-byte index into the constant pool (q.v.). If the types are not compatible, a ClassCastException will be thrown (see athrow).

checkcast_quick (0xe0) — Quick version of checkcast opcode

Summary: Optimized version of checkcast opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

d2f (0x90) — Convert double to float

Summary: Pops a two-word double off the stack, converts it to a single-word floating point number, and pushes the result.

d2i (0x8e) — Convert double to integer

Summary: Pops a two-word double off the stack, converts it to a single-word integer, and pushes the result.

d2l (0x8f) — Convert double to long

Summary: Pops a two-word double off the stack, converts it to a two-word long, and pushes the result.
**dadd** (0x63) — Double precision addition

Summary: Pops two doubles and pushes their sum

Initial: double-1
double-1
double-2
double-2

Final: double
double

**daload** (0x31) — Load value from array of bytes

Summary: Pops an integer and an array from the stack, then retrieves a value from that location in the 1-dimensional array of doubles. The value retrieved is pushed on the top of the stack

Initial: int(index)
address(array ref)

Final: double
double

**dastore** (0x52) — Store value in array of doubles

Summary: Stores a two-word double in an array of such double. The top argument popped is the index defining the array location to be used. The second/third arguments popped are the double value to be stored, and the final argument is the array itself.

Initial: int(index)
double
double
address(array ref)

Final: —

**dcmpg** (0x98) — compare doubles, returning 1 on NaN

Summary: Pops two two-word doubles off the operand stack and pushes a -1, 0, or +1 (as an integer) as a result. If the next-to-top number is greater than the top number, the value pushed is +1. If the two numbers are equal, the value pushed is 0, otherwise the value is -1. If either or both words popped equal IEEE NaN (Not a Number), when interpreted as a double, the result pushed is 1.

Initial: double-1
double-1
double-2
double-2

Final: int

**dcmpl** (0x97) — compare doubles, returning -11 on NaN

Summary: Pops two two-word doubles off the operand stack and pushes a -1, 0, or +1 (as an integer) as a result. If the next-to-top number is greater than the top number, the value pushed is +1. If the two numbers are equal, the value pushed is 0, otherwise the value is -1. If either or both words popped equal IEEE NaN (Not a Number), when interpreted as a double, the result pushed is -1.

Initial: double-1
double-1
double-2
double-2

Final: int
dconst_0 (0xe) — Push double constant 0.0

Summary: Pushes the constant value 0.0 as a 64-bit IEEE double-precision floating point value onto the operand stack

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>double(0.0) double(0.0)</td>
</tr>
</tbody>
</table>


dconst_1 (0xf) — Push double constant 1.0

Summary: Pushes the constant value 1.0 as a 64-bit IEEE double-precision floating point value onto the operand stack

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>double(1.0) double(1.0)</td>
</tr>
</tbody>
</table>


ddiv (0x6f) — Double precision division

Summary: Pops two two-word double precision floating point numbers, then pushes the result of the next-to-top number divided by the top number.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>double-1 double-1 double-2 double-2</td>
<td>double double</td>
</tr>
</tbody>
</table>

dload <varnum> (0x18 [byte/short]) — Load double from local variable

Summary: Loads two-word double precision floating point number from local variables #<varnum> and #<varnum>+1 and pushes the value. The value of <varnum> is a byte in the range 0..255 unless the wide operand prefix is used, in which case it is a short in the range 0..65536.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>double double</td>
</tr>
</tbody>
</table>

dmul (0x6b) — Double precision multiplication

Summary: Pops two doubles and pushes their product

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>double-1 double-1 double-2 double-2</td>
<td>double double</td>
</tr>
</tbody>
</table>
**dload_0** (0x26) — Load double from local variable #0/#1
Summary: Loads double precision floating point number from local variables #0 and #1, and pushes the value loaded onto the stack. This is functionally equivalent to **dload 0** but takes fewer bytes and is faster.

Initial: —
Final: double
double

**dload_1** (0x27) — Load double from local variable #0/#1
Summary: Loads double precision floating point number from local variables #1 and #2, and pushes the value loaded onto the stack. This is functionally equivalent to **dload 1** but takes fewer bytes and is faster.

Initial: —
Final: double
double

**dload_2** (0x28) — Load double from local variable #0/#1
Summary: Loads double precision floating point number from local variables #2 and #3, and pushes the value loaded onto the stack. This is functionally equivalent to **dload 2** but takes fewer bytes and is faster.

Initial: —
Final: double
double

**dload_3** (0x29) — Load double from local variable #0/#1
Summary: Loads double precision floating point number from local variables #3 and #4, and pushes the value loaded onto the stack. This is functionally equivalent to **dload 3** but takes fewer bytes and is faster.

Initial: —
Final: double
double

**dneg** (0x77) — Double precision negation
Summary: Pops a two-word double precision floating point number from the stack, reverses its sign (multiplies by -1), then pushes the result.

Initial: double
double
Final: double
double
**APPENDIX B. JVM INSTRUCTION SET**

**drem** (0x73) — Double precision remainder

Summary: Pops two two-word double precision floating point numbers, then pushes the remainder resulting when the next-to-top number is divided by the top number.

Initial: double-1
double-1
double-2
double-2

Final: double
double

**dreturn** (0xaf) — Return from method with double result

Summary: Pops a two-byte double precision floating point number from current method stack. This number is pushed onto the method stack of the calling environment. The current method is terminated and control is transferred to the calling environment.

Initial: double
double

Final: (n/a)

**dstore <varnum>** (0x39 [byte/short]) — Store double in local variable

Summary: Pops double from top of stack and stores that double value in local variables #<varnum> and #<varnum> +1. The value of <varnum> is a byte in the range 0..255 unless the wide operand prefix is used, in which case it is a short in the range 0..65536.

Initial: double
double

Final: —

**dstore_0** (0x47) — Store double in local variable #0/#1

Summary: Pops integer from top of stack and stores that integer value in local variables #0 and #1. This is functionally equivalent to dstore 0, but takes fewer bytes and is faster.

Initial: double
double

Final: —

**dstore_1** (0x48) — Store double in local variable #1/#2

Summary: Pops double from top of stack and stores that double value in local variables #1 and #2. This is functionally equivalent to dstore 1, but takes fewer bytes and is faster.

Initial: double
double

Final: —
**dstore_2** (0x49) — Store double in local variable #2/#3
Summary: Pops double from top of stack and stores that double value in local variables #2 and #3. This is functionally equivalent to *dstore 2*, but takes fewer bytes and is faster.

**dstore_3** (0x4a) — Store double in local variable #3/#4
Summary: Pops double from top of stack and stores that double value in local variables #3 and #4. This is functionally equivalent to *dstore 3*, but takes fewer bytes and is faster.

**dsub** (0x67) — Double precision subtraction
Summary: Pops two two-word double precision floating point numbers, then pushes the result of the next-to-top number minus the top number.

**dup** (0x59) — Duplicate top stack word
Summary: Duplicates top word of stack and pushes a copy.

**dup2** (0x5c) — Duplicate top two stack words
Summary: Duplicates top two words of stack and pushes copies at top of stack. Items duplicated can be two separate one-word entries (such as ints, floats, or addresses) or a single two-word entry (such as a long or double).
**dup2_x1** (0x5d) — Duplicate top two words of stack and insert under third word

Summary: Duplicates top two words of stack and inserts the duplicate below the third word as new fourth and fifth words. Items duplicated can be two separate one-word entries (such as ints, floats, or addresses) or a single two-word entry (such as a long or double).

Initial: word-1
word-2
word-3
Final: word-1
word-2
word-3
word-1
word-2

**dup2_x2** (0x5e) — Duplicate top two words of stack and insert under fourth word

Summary: Duplicates top two words of stack and inserts the duplicate below the fourth word as new fifth and sixth words. Items duplicated can be two separate one-word entries (such as ints, floats, or addresses) or a single two-word entry (such as a long or double).

Initial: word-1
word-2
word-3
word-4
Final: word-1
word-2
word-3
word-4
word-1
word-2

**dup_x1** (0x5a) — Duplicate top word of stack and insert under second word

Summary: Duplicates top word of stack and inserts the duplicate below the second word as a new third word.

Initial: word-1
word-2
Final: word-1
word-2
word-1

**dup_x2** (0x5b) — Duplicate top word of stack and insert under third word

Summary: Duplicates top word of stack and inserts the duplicate below the third word as a new fourth word.

Initial: word-1
word-2
word-3
Final: word-1
word-2
word-3
word-1
f2d (0x8d) — Convert float to double

Summary: Pops a single-word floating point number off the stack, converts it to a two-word double, and pushes the result.

Initial: float
Final: double

double

f2i (0x8b) — Convert float to int

Summary: Pops a single-word floating point number off the stack, converts it to a single-word integer, and pushes the result.

Initial: float
Final: int

f2l (0x8c) — Convert float to long

Summary: Pops a single-word floating point number off the stack, converts it to a two-word long integer, and pushes the result.

Initial: float
Final: long

long

fadd (0x62) — Floating point addition

Summary: Pops two floats and pushes their sum

Initial: float
Final: float

faload (0x30) — Load value from array of bytes

Summary: Pops an integer and an array from the stack, then retrieves a value from that location in the 1-dimensional array of floats. The value retrieved is pushed on the top of the stack

Initial: int(index)
address(array ref)
Final: float

fastore (0x51) — Store value in array of addresses

Summary: Stores a single-word floating point number in an array of such floats. The top argument popped is the index defining the array location to be used. The second argument popped is the float value to be stored, and the third and final argument is the array itself.

Initial: int(index)
float
address(array ref)
Final: —
APPENDIX B. JVM INSTRUCTION SET

{}314

**fcmpg** (0x96) — compare floats, returning 1 on NaN
Summary: Pops two single-word floating point numbers off the operand stack and pushes a -1, 0, or +1 (as an integer) as a result. If the next-to-top number is greater than the top number, the value pushed is +1. If the two numbers are equal, the value pushed is 0, otherwise the value is -1. If either or both words popped equal IEEE NaN (Not a Number), when interpreted as a floating point number, the result pushed is 1.

Initial: float
Final: int

**fcmpl** (0x95) — compare floats, returning -1 on NaN
Summary: Pops two single-word floating point numbers off the operand stack and pushes a -1, 0, or +1 (as an integer) as a result. If the next-to-top number is greater than the top number, the value pushed is +1. If the two numbers are equal, the value pushed is 0, otherwise the value is -1. If either or both words popped equal IEEE NaN (Not a Number), when interpreted as a floating point number, the result pushed is -1.

Initial: float
Final: int

**fconst_0** (0xb) — Push floating point constant 0.0
Summary: Pushes the constant value 0.0 as an IEEE 32-bit floating point value onto the operand stack

Initial: —
Final: float(0.0)

**fconst_1** (0xc) — Push floating point constant 1.0
Summary: Pushes the constant value 1.0 as an IEEE 32-bit floating point value onto the operand stack

Initial: —
Final: float(1.0)

**fconst_2** (0xdd) — Push floating point constant 2.0
Summary: Pushes the constant value 1.0 as an IEEE 32-bit floating point value onto the operand stack

Initial: —
Final: float(1.0)
**fdiv** (0x6e) — Floating point division

Summary: Pops two single-word floating point numbers, then pushes the result of the next-to-top number divided by the top number.

Initial: float float
Final: float

**fload** <varnum> (0x17 [byte/short]) — Load int from local variable

Summary: Loads single-word floating point number from local variable <varnum> and pushes the value. The value of <varnum> is a byte in the range 0..255 unless the wide operand prefix is used, in which case it is a short in the range 0..65536.

Initial: —
Final: float

**fmul** (0x6a) — Floating point multiplication

Summary: Pops two floats and pushes their product

Initial: float float
Final: float

**fload_0** (0x22) — Load float from local variable #0

Summary: Loads single-word floating point number from local variable #0 and pushes the value loaded onto the stack. This is functionally equivalent to **fload 0** but takes fewer bytes and is faster.

Initial: —
Final: float

**fload_1** (0x23) — Load float from local variable #1

Summary: Loads single-word floating point number from local variable #1 and pushes the value loaded onto the stack. This is functionally equivalent to **fload 1** but takes fewer bytes and is faster.

Initial: —
Final: float

**fload_2** (0x24) — Load float from local variable #2

Summary: Loads single-word floating point number from local variable #2 and pushes the value loaded onto the stack. This is functionally equivalent to **fload 2** but takes fewer bytes and is faster.

Initial: —
Final: float
**APPENDIX B. JVM INSTRUCTION SET**

**fload_3** (0x25) — Load float from local variable #3
Summary: Loads single-word floating point number from local variable #3 and pushes the value loaded onto the stack. This is functionally equivalent to *fload 3* but takes fewer bytes and is faster.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>float</td>
</tr>
</tbody>
</table>

**fneg** (0x76) — Floating point negation
Summary: Pops a floating point number from the stack, reverses its sign (multiplies by -1), then pushes the result.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>

**frem** (0x72) — Floating point remainder
Summary: Pops two single-word floating point numbers, then pushes the remainder resulting when the next-to-top number is divided by the top number.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>

**freturn** (0xae) — Return from method with float result
Summary: Pops a single-word floating point number from current method stack. This number is pushed onto the method stack of the calling environment. The current method is terminated and control is transferred to the calling environment.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>(n/a)</td>
</tr>
</tbody>
</table>

**fstore <varnum>** (0x38 [byte/short]) — Store float in local variable
Summary: Pops float from top of stack and stores that float value in local variable #<varnum>. The value of <varnum> is a byte in the range 0..255 unless the *wide* operand prefix is used, in which case it is a short in the range 0..65536.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>—</td>
</tr>
</tbody>
</table>

**fstore_0** (0x43) — Store float in local variable #0
Summary: Pops float from top of stack and stores that float value in local variable #0. This is functionally equivalent to *fstore 0*, but takes fewer bytes and is faster.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>—</td>
</tr>
</tbody>
</table>
**fstore.1** (0x44) — Store float in local variable #1  
Summary: Pops float from top of stack and stores that float value in local variable #0. This is functionally equivalent to `fstore 1`, but takes fewer bytes and is faster.

Initial: float  
Final: —

**fstore.2** (0x45) — Store float in local variable #2  
Summary: Pops float from top of stack and stores that float value in local variable #0. This is functionally equivalent to `fstore 2`, but takes fewer bytes and is faster.

Initial: float  
Final: —

**fstore.3** (0x46) — Store float in local variable #3  
Summary: Pops float from top of stack and stores that float value in local variable #0. This is functionally equivalent to `fstore 3`, but takes fewer bytes and is faster.

Initial: float  
Final: —

**fsub** (0x66) — Floating point subtraction  
Summary: Pops two single-word floating point numbers, then pushes the result of the next-to-top number minus the top number.

Initial: float  
Final: float

**getfield <typename> <type>** (0xb4 [short][short]) — Get object field  
Summary: Pops an address (object reference) from the stack and retrieves and pushes the value of the identified field. The `getfield` opcode takes two parameters, the field identifier and the field type, respectively. These are stored in the bytecode as two-byte indices into the constant pool (q.v.). Unlike in Java, the field name must always be a fully qualified name, including the name of the relevant class and any relevant packages.

Initial: address(object)  
Final: value

**getfield2 quick** (0xd0) — Quick version of `getfield` for two-word fields  
Summary: Optimized version of `getfield` opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a `.class` file that isn’t currently loaded and being executed.

Initial: see original opcode  
Final: see original opcode
APPENDIX B. JVM INSTRUCTION SET

getfield_quick (0xce) — Quick version of getfield opcode

Summary: Optimized version of getfield opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

getfield_quick_w (0xe3) — Quick, wide version of getfield opcode

Summary: Optimized version of getfield opcodes used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

getstatic <fieldname> <type> (0xb2 [short][short]) — Get class field

Summary: Retrieves and pushes the value of the identified class field. The getfield opcode takes two parameters, the field identifier and the field type, respectively. These are stored in the bytecode as two-byte indices into the constant pool(q.v.). Unlike in Java, the field name must always be a fully qualified name, including the name of the relevant class and any relevant packages.

Initial: —
Final: value

gostatic2.quick (0xd4) — Quick version of getstatic opcode for two-byte fields

Summary: Optimized version of getstatic opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

gostatic_quick (0xd2) — Quick version of getstatic opcode

Summary: Optimized version of getstatic opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: initstack
Final: finalstack
**goto <label>** (0xa7 [short]) — Go to label unconditionally
Summary: Transfers control unconditionally to the location marked by <label>. In bytecode, this opcode is followed by a two-byte offset to be added to the current value in the program counter. If the label is further away than can be represented in a two-byte offset, use goto_w instead; the jasmin assembler is capable of determining which opcode to use based on its analysis of the distances.

*Initial*: —  
*Final*: —

**goto_w <label>** (0xc8 [int]) — Go to label unconditionally using wide offset
Summary: Transfers control unconditionally to the location marked by <label>. In bytecode, this opcode is followed by a four-byte offset to be added to the current value in the program counter. Using this opcode makes it possible to branch to locations more than 32767 bytes away from the current locations. The jasmin assembler will automatically determine whether goto or goto_w should be used based on its analysis of the distances.

*Initial*: —  
*Final*: —

**i2b** (0x91) — Convert integer to byte
Summary: Pops a single-word integer off the stack, converts it by truncation to a single byte (value 0..255), zero extends the result to 32-bits, and pushes the result (as an integer)

*Initial*: int  
*Final*: int

**i2c** (0x92) — Convert integer to character
Summary: Pops a single-word integer off the stack, converts it by truncation to a two byte UTF-16 character, zero extends the result to 32-bits, and pushes the result (as an integer)

*Initial*: int  
*Final*: int

**i2d** (0x87) — Convert integer to double
Summary: Pops a single-word integer off the stack, converts it to a two-word double, and pushes the result

*Initial*: int  
*Final*: double

**i2f** (0x86) — Convert integer to float
Summary: Pops a single-word integer off the stack, converts it to a single-word floating point number, and pushes the result

*Initial*: int  
*Final*: float
i2l (0x85) — Convert integer to long

Summary: Pops a single-word integer off the stack, sign extends it to a two-word long integer, and pushes the result

Initial: int
Final: long

i2s (0x93) — Convert integer to short

Summary: Pops a single-word integer off the stack, converts it by truncation to a signed short integer (value -32768..32767), sign extends the result to 32-bits, and pushes the result (as an integer). Note that the truncation can cause a change in sign as the integer’s original sign bit is lost.

Initial: int
Final: int

iadd (0x60) — Integer addition

Summary: Pops two integers and pushes their sum

Initial: int
Final: int

iaload (0x2e) — Load value from array of bytes

Summary: Pops an integer and an array from the stack, then retrieves a value from that location in the 1-dimensional array of integers. The value retrieved is pushed on the top of the stack

Initial: int(index)
address(array ref)
Final: int

iand (0x7e) — Integer logical AND

Summary: Pops two integers from the stack, calculates their bit-wise AND and pushes the 32-bit result as an integer.

Initial: int
Final: int
**iastore** (0x4f) — Store value in array of integers

Summary: Stores a single-word integer in an array of such ints. The top argument popped is the index defining the array location to be used. The second argument popped is the integer value to be stored, and the third and final argument is the array itself.

**iconst_0** (0x03) — Push integer constant 0

Summary: Pushes the constant value 0 (0x0) as a 32-bit integer onto the operand stack

**iconst_1** (0x4) — Push integer constant 1

Summary: Pushes the constant value 0 (0x1) as a 32-bit integer onto the operand stack

**iconst_2** (0x5) — Push integer constant 2

Summary: Pushes the constant value 0 (0x2) as a 32-bit integer onto the operand stack

**iconst_3** (0x6) — Push integer constant 3

Summary: Pushes the constant value 0 (0x3) as a 32-bit integer onto the operand stack

**iconst_4** (0x7) — Push integer constant 4

Summary: Pushes the constant value 0 (0x4) as a 32-bit integer onto the operand stack

**iconst_5** (0x8) — Push integer constant 5

Summary: Pushes the constant value 0 (0x5) as a 32-bit integer onto the operand stack
iconst_m1 (0x2) — Push integer constant -1
Summary: Pushes the constant value -1 (0xFFFF) as a 32-bit integer onto the operand stack
Initial: —
Final: int(-1)

idiv (0x6c) — Integer division
Summary: Pops two integers, then pushes the integer part of the result of the next-to-top number divided by the top number.
Initial: int
Final: int

if_acmpeq <label> (0xa5 [short]) — Compare addresses and branch if equal
Summary: Pops two addresses (object references) from the stack. If the next-to-top element is equal to the top element, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.
Initial: address
address
Final: —

if_acmpne <label> (0xa6 [short]) — Compare addresses and branch if not equal
Summary: Pops two addresses (object references) from the stack. If the next-to-top element is not equal to the top element, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.
Initial: address
address
Final: —

if_icmpeq <label> (0x9f [short]) — Compare integers and branch if equal
Summary: Pops two integers from the stack. If the next-to-top element is equal to the top element, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.
Initial: int
Final: —
if_icmpge <label> (0xa2 [short]) — Compare integers and branch if greater than or equal
Summary: Pops two integers from the stack. If the next-to-top element is greater than or equal to the top element, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

if_icmpgt <label> (0xa3 [short]) — Compare integers and branch if greater than
Summary: Pops two integers from the stack. If the next-to-top element is greater than the top element, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

if_icmple <label> (0xa4 [short]) — Compare integers and branch if less than or equal
Summary: Pops two integers from the stack. If the next-to-top element is less than or equal to the top element, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

if_icmplt <label> (0xa1 [short]) — Compare integers and branch if less than
Summary: Pops two integers from the stack. If the next-to-top element is less than the top element, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

if_icmpne <label> (0xa0 [short]) — Compare integers and branch if not equal
Summary: Pops two integers from the stack. If the next-to-top element is not equal to the top element, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.
ifeq <label> (0x99 [short]) — Branch if equal
Summary: Pops integer off top of operand stack. If the value of the integer popped is equal to zero, control is transferred to <label>.
Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

Initial: int
Final: —

ifge <label> (0x9c [short]) — Branch if greater than or equal
Summary: Pops integer off top of operand stack. If the value of the integer popped is greater than or equal to zero, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

Initial: int
Final: —

ifgt <label> (0x9d [short]) — Branch if greater than
Summary: Pops integer off top of operand stack. If the value of the integer popped is greater than zero, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

Initial: int
Final: —

ifle <label> (0x9e [short]) — Branch if less than or equal
Summary: Pops integer off top of operand stack. If the value of the integer popped less than or equal to zero, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

Initial: int
Final: —

iflt <label> (0x9b [short]) — Branch if less than
Summary: Pops integer off top of operand stack. If the value of the integer popped is less than zero, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

Initial: int
Final: —
**ifne <label>** (0x9a [short]) — Branch if not equal  
Summary: Pops integer off top of operand stack. If the value of the integer popped is not equal to zero, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

Initial: int  
Final: —

**ifnonnull <label>** (0xc7 [short]) — Branch if not null  
Summary: Pops address (object reference) off top of operand stack. If the value of the address popped is not null, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

Initial: address  
Final: —

**ifnull <label>** (0xc6 [short]) — Branch if null  
Summary: Pops address (object reference) off top of operand stack. If the value of the address popped is null, control is transferred to <label>. Internally, the opcode is followed by a two-byte quantity which is treated as an offset and added to the current value of the program counter if the branch is to be taken.

Initial: address  
Final: —

**iinc <varnum> <increment>** (0x84 [byte/short] [byte/short]) — Increment integer in local variable  
Summary: Increments a local variable containing an increment. The first argument defines the variable number to be adjusted, while the second is a signed constant amount of adjustment. Normally, the variable can be from 0..255, while the “increment” can be any number from -128..127. If the wide prefix is specified, the variable can be from 0..65536 and the increment can range from -32768..32767. The stack is not changed.

Initial: —  
Final: —

**iload <varnum>** (0x15 [byte/short]) — Load int from local variable  
Summary: Loads integer from local variable #<varnum> and pushes the value. The value of <varnum> is a byte in the range 0..255 unless the wide operand prefix is used, in which case it is a short in the range 0..65536.

Initial: —  
Final: int
**APPENDIX B. JVM INSTRUCTION SET**

**iload_0 (0x1a)** — Load int from local variable #0
Summary: Loads single-word integer from local variable #0 and pushes the value loaded onto the stack. This is functionally equivalent to `iload 0` but takes fewer bytes and is faster.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>int</td>
</tr>
</tbody>
</table>

**iload_1 (0x1b)** — Load int from local variable #1
Summary: Loads single-word integer from local variable #1 and pushes the value loaded onto the stack. This is functionally equivalent to `iload 1` but takes fewer bytes and is faster.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>int</td>
</tr>
</tbody>
</table>

**iload_2 (0x1c)** — Load int from local variable #2
Summary: Loads single-word integer from local variable #2 and pushes the value loaded onto the stack. This is functionally equivalent to `iload 2` but takes fewer bytes and is faster.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>int</td>
</tr>
</tbody>
</table>

**iload_3 (0x1d)** — Load int from local variable #3
Summary: Loads single-word integer from local variable #3 and pushes the value loaded onto the stack. This is functionally equivalent to `iload 3` but takes fewer bytes and is faster.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>int</td>
</tr>
</tbody>
</table>

**impdep1 (0xfe)** — Reserved opcode
Summary: This opcode is reserved for internal use by a JVM implementation. It is illegal for such an opcode to appear in a class file, and such a class file will fail verification.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n/a)</td>
<td>(n/a)</td>
</tr>
</tbody>
</table>

**impdep2 (0xff)** — Reserved opcode
Summary: This opcode is reserved for internal use by a JVM implementation. It is illegal for such an opcode to appear in a class file, and such a class file will fail verification.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n/a)</td>
<td>(n/a)</td>
</tr>
</tbody>
</table>
**imul** (0x68) — Integer multiplication

Summary: Pops two integers and pushes their product

Initial: int

Final: int

**ineg** (0x74) — Integer negation

Summary: Pops an integer the stack, reverses its sign (multiplies by -1), then pushes the result.

Initial: int

Final: int

**instanceof <type>** (0xc1 [short]) — Test if object/array is of specified type

Summary: Pops an address (object or array reference) from the stack and determines if that object/array is compatible with that type; either an instance of that type, a implementation of that interface, or an instance of a relevant supertype. If it is compatible, the integer value 1 is pushed, 0 otherwise.

Initial: address

Final: int

**instanceof.quick** (0xe1) — Quick version of **instanceof** opcode

Summary: Optimized version of **instanceof** opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode

Final: see original opcode
invokeinterface <method> <Nargs> (0xb9 [short][byte][byte]) — Invoke interface method

Summary: Invokes a method defined within an interface (as opposed to a class). Arguments to invokeinterface include the fully-qualified name of the method to be invoked (including the interface name, parameter types, and return type) and the number of arguments. These arguments are popped from the stack along with an address (object reference) of an object implementing that interface. A new stack frame is created for the called environment, and the object and arguments are pushed onto this environment’s stack. Control then passes to the new method/environment. Upon return, the return value (given by ?return) is pushed onto the calling environment’s stack.

In bytecode, the method name is stored as a two-byte index into the constant pool (q.v.). The next byte stores the number of argument words, up to 255 passed to the method. The next byte must store the value 0 and can be used internally by the JVM to store hash values to speed up method lookup.

invokeinterface_quick (0xda) — Quick version of invokeinterface opcode

Summary: Optimized version of invokeinterface opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

invokenonvirtual_quick (0xd7) — Quick version of invokespecial opcode

Summary: Optimized version of invokespecial opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.
invokespecial <method> (0xb7 [short]) — Invoke instance method

Summary: Invokes an instance method on an object in certain special cases. Specifically, use invokespecial to invoke:

- the instance initialization method <init>
- a private method of this, or
- a method in a superclass of this.

This opcode is otherwise similar to invokevirtual (q.v.). Arguments to invokespecial include the fully-qualified name of the method to be invoked (including the class name, parameter types, and return type) and the number of arguments. These arguments are popped from the stack along with an address (object reference) of an instance of the relevant class. A new stack frame is created for the called environment, and the object and arguments are pushed onto this environment's stack. Control then passes to the new method/environment. Upon return, the return value (given by ?return) is pushed onto the calling environment's stack. In bytecode, the method name is stored as a two-byte index into the constant pool (q.v.).

invokestatic <method> (0xb8 [short]) — Invoke static method

Summary: Invokes a static method on an class. Arguments to invokestatic include the fully-qualified name of the method to be invoked (including the class name, parameter types, and return type) and the number of arguments. These arguments are popped from the stack. A new stack frame is created for the called environment, and arguments are pushed onto this environment's stack. Control then passes to the new method/environment. Upon return, the return value (given by ?return) is pushed onto the calling environment's stack. In bytecode, the method name is stored as a two-byte index into the constant pool (q.v.).

invokestatic_quick (0xd9) — Quick version of invokestatic opcode

Summary: Optimized version of invokestatic opcode used internally in Sun's Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn't currently loaded and being executed.
invokesuper_quick (0xd8) — Quick version of invokespecial opcode

Summary: Optimized version of invokespecial opcode used internally in Sun's Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

invokevirtual <method> (0xb6 [short]) — Invoke instance method

Summary: Invokes an instance method on an object. Arguments to invokevirtual include the fully-qualified name of the method to be invoked (including the class name, parameter types, and return type) and the number of arguments. These arguments are popped from the stack along with an address (object reference) of an instance of the relevant class. A new stack frame is created for the called environment, and the object and arguments are pushed onto this environment’s stack. Control then passes to the new method/environment. Upon return, the return value (given by ?return) is pushed onto the calling environment’s stack. In bytecode, the method name is stored as a two-byte index into the constant pool (q.v.).

invokevirtual_quick (0xd6) — Quick version of invokevirtual opcode

Summary: Optimized version of invokevirtual opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

invokevirtual_quick_w (0xe2) — Quick version of invokevirtual opcode (wide index)

Summary: Optimized version of invokevirtual opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.
invokevirtualobject_quick (0xdb) — Quick version of invokevirtual for methods on Object

Summary: Optimized version of invokevirtual opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

ior (0x80) — integer logical OR

Summary: Pops two integers from the stack, calculates their bitwise inclusive AOR and pushes the 32-bit result as an integer.

irem (0x70) — Integer remainder

Summary: Pops two single-word integers, then pushes the remainder resulting when the next-to-top number is divided by the top number. This operation is rather like the C or Java % operation.

ireturn (0xac) — Return from method with integer result

Summary: Pops an integer from current method stack. This integer is pushed onto the method stack of the calling environment. The current method is terminated and control is transferred to the calling environment.

ishl (0x78) — Shift integer to the left

Summary: Pops an integer and another integer from the stack. The value of the next-to-top integer is shifted to the left the number of bits indicated by the lower six bits of the top integer, then the resulting long is pushed. Newly emptied places are filled with 0 bits. This is equivalent to multiplying the value by a power of 2, but may be faster.
APPENDIX B. JVM INSTRUCTION SET

ishr (0x7a) — Shift integer to the right
Summary: Pops an integer and another integer from the stack. The value of the next-to-top integer is shifted to the right the number of bits indicated by the lower six bits of the top integer, then the resulting value is pushed. N.b., this is an arithmetic shift, meaning that the sign bit is copied to fill the newly emptied places.

\[
\text{Initial: } \text{int(shift)} \quad \text{int(value)} \\
\text{Final: } \text{int}
\]

istore <varnum> (0x36 [byte/short]) — Store integer in local variable
Summary: Pops integer from top of stack and stores that integer value in local variable #<varnum>. The value of <varnum> is a byte in the range 0..255 unless the wide operand prefix is used, in which case it is a short in the range 0..65536.

\[
\text{Initial: } \text{int} \\
\text{Final: } —
\]

istore_0 (0x3b) — Store integer in local variable #0
Summary: Pops integer from top of stack and stores that integer value in local variable #0. This is functionally equivalent to istore 0, but takes fewer bytes and is faster.

\[
\text{Initial: } \text{int} \\
\text{Final: } —
\]

istore_1 (0x3c) — Store integer in local variable #1
Summary: Pops integer from top of stack and stores that integer value in local variable #1. This is functionally equivalent to istore 1, but takes fewer bytes and is faster.

\[
\text{Initial: } \text{int} \\
\text{Final: } —
\]

istore_2 (0x3d) — Store integer in local variable #2
Summary: Pops integer from top of stack and stores that integer value in local variable #3. This is functionally equivalent to istore 2, but takes fewer bytes and is faster.

\[
\text{Initial: } \text{int} \\
\text{Final: } —
\]

istore_3 (0x3e) — Store integer in local variable #3
Summary: Pops integer from top of stack and stores that integer value in local variable #3. This is functionally equivalent to istore 3, but takes fewer bytes and is faster.

\[
\text{Initial: } \text{int} \\
\text{Final: } —
\]
**isub** (0x64) — Integer subtraction

Summary: Pops two single-word integers, then pushes the result of the next-to-top number minus the top number.

Initial: int
Final: int

**iushr** (0x7c) — Shift unsigned int to the right

Summary: Pops an integer and another integer from the stack. The value of the next-to-top integer is shifted to the right the number of bits indicated by the lower six bits of the top, then the resulting value is pushed. N.b., this is an arithmetic shift, meaning that the sign bit is ignored and the bit-value 0 is used to fill the newly emptied places.

Initial: int(shift) int(value)
Final: int

**ixor** (0x82) — integer logical XOR

Summary: Pops two integers from the stack, calculates their bitwise XOR (exclusive OR) and pushes the 32-bit result as an integer.

Initial: int
Final: int

**jsr_w** <label> (0xc9 [int]) — Jump to subroutine using wide offset

Summary: Pushes the location of next instruction (PC + 5, representing the length of the jsr_w instruction itself), then executes an unconditional branch to <label>.

Initial: —
Final: address(locn)

**jsr** <label> (0xa8 [short]) — Jump to subroutine

Summary: Pushes the location of next instruction (PC + 3, representing the length of the jsr instruction itself), then executes an unconditional branch to <label>.

Initial: —
Final: address(locn)

**l2d** (0x8a) — Convert long to double

Summary: Pops a double-word long integer point number off the stack, converts it to a two-word double, and pushes the result.

Initial: long
Final: double
APPENDIX B. JVM INSTRUCTION SET

l2f (0x89) — Convert long to float
Summary: Pops a double-word long integer point number off the stack, converts it to a single-word floating point number, and pushes the result.

Initial: long
Final: float

l2i (0x88) — Convert long to int
Summary: Pops a double-word long integer point number off the stack, converts it by to a single-word integer, and pushes the result. Note that this may cause a change in sign as the long’s original sign bit is lost.

Initial: long
Final: int

ladd (0x61) — long addition
Summary: Pops two longs and pushes their sum

Initial: long-1
long-2
Final: long
long

laload (0x2f) — Load value from array of bytes
Summary: Pops an integer and an array from the stack, then retrieves a value from that location in the 1-dimensional array of longs. The value retrieved is pushed on the top of the stack

Initial: int(index)
address(array ref)
Final: long
long

land (0x7f) — long logical AND
Summary: Pops two longs from the stack, calculates their bitwise AND and pushes the 64-bit result as a long.

Initial: long-1
long-2
Final: long
long
lastore (0x50) — Store value in array of longs

Summary: Stores a two-word long integer in an array of such longs. The top argument popped is the index defining the array location to be used. The second/third arguments popped are the long value to be stored, and the final argument is the array itself.

Initial: int(index) long long address(array ref)
Final: —

lcmp (0x94) — Compare longs

Summary: Pops two two-word integers off the operand stack and compares them. If the next-to-top value is greater than the top value, the integer 1 is pushed. If the two values are equal, the integer 0 is pushed, and otherwise the integer -1 is pushed.

Initial: long-1 long-1 long-2 long-2
Final: int

lconst_0 (0x9) — Push integer constant zero

Summary: Pushes the constant value 0 (0x0) as a 64-bit long integer onto the operand stack

Initial: —
Final: long(0) long(0)

lconst_1 (0xa) — Push integer constant zero

Summary: Pushes the constant value 0 (0x0) as a 64-bit long integer onto the operand stack

Initial: —
Final: long(1) long(1)

ldc <constant> (0x12 [short]) — Load one-word constant

Summary: Loads and pushes a single-word entry from the constant pool (q.v.). <constant> can be an int, a float, or a literal string, which is stored as an entry numbered from 0..255 in the constant pool.

Initial: —
Final: word
APPENDIX B. JVM INSTRUCTION SET

**ldc2_w** <constant> (0x14 [int]) — Load two-word constant

Summary: Loads and pushes a double-word entry from the constant pool (q.v.). <constant> can be a double or long, which is stored as an entry numbered from 0..65536 in the constant pool.

Initial: —
Final: word

**ldc quick** (0xcb) — Quick version of ldc opcode

Summary: Optimized version of ldc opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

**ldc.w** <constant> (0x13 [int]) — Load one-word constant with wide access

Summary: Loads and pushes a single-word entry from the constant pool (q.v.). <constant> can be an int, a float, or a literal string, which is stored as an entry numbered from 0..65536 in the constant pool.

Initial: —
Final: word

**ldc.w_quick** (0xcd) — Quick, wide version of ldc opcode

Summary: Optimized version of ldc opcodes used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

**ldiv** (0x6d) — Long integer division

Summary: Pops two two-word long integers, then pushes the integer part of the result of the next-to-top number divided by the top number.

Initial: long-1
long-1
long-2
long-2
Final: long
long
**lload <varnum>** (0x16 [byte/short]) — Load long from local variable
Summary: Loads two-word long integer from local variables 
#<varnum> and #<varnum>+1 and pushes the value. The value of <varnum> is a byte in the range 0..255 unless the wide operand prefix is used, in which case it is a short in the range 0..65536.

Initial: —
Final: long

**lload_0** (0x1e) — Load long from local variable #0/#1
Summary: Loads two-byte long integer from local variables #0 and #1, and pushes the value loaded onto the stack. This is functionally equivalent to lload 0 but takes fewer bytes and is faster.

Initial: —
Final: long

**lload_1** (0x1f) — Load long from local variable #1/#2
Summary: Loads two-byte long integer from local variables #1 and #2, and pushes the value loaded onto the stack. This is functionally equivalent to lload 1 but takes fewer bytes and is faster.

Initial: —
Final: long

**lload_2** (0x20) — Load long from local variable #2/#3
Summary: Loads two-byte long integer from local variables #2 and #3, and pushes the value loaded onto the stack. This is functionally equivalent to lload 2 but takes fewer bytes and is faster.

Initial: —
Final: long

**lload_3** (0x21) — Load long from local variable #3/#4
Summary: Loads two-byte long integer from local variables #3 and #4, and pushes the value loaded onto the stack. This is functionally equivalent to lload 3 but takes fewer bytes and is faster.

Initial: —
Final: long

**lmul** (0x69) — Long integer multiplication

Summary: Pops two longs and pushes their product

Initial: long-1
long-1
long-2
long-2
Final: long
long
lookupsword
1 : One
2 : Two
3 : Three
5 : Five
default: Elsewhere

Figure B.1: Example of lookupsword

lneg (0x75) — Long integer negation
Summary: Pops a two-word long integer from the stack, reverses its sign (multiplies by -1), then pushes the result.

lookupsword<args>(0xab [args]) — Multiway branch
Summary: Performs a multiway branch, like the Java/C++ switch statement. The integer at the top of the stack is popped and compared to a set of value:label pairs. If the integer is equal to value, control is passed to the corresponding label. If no value matches the integer, control passes instead to a defined default label. Labels are implemented as relative offsets and added to the current contents of the program counter to get the location of the next instruction to execute.
See the figure for example of this statement in use.
The lookupsword instruction has a variable number of arguments and is thus rather tricky in its bytecode storage. After the opcode (0xab) itself, there follows from 0 to 3 bytes of padding, so that the the four-byte default offset begins at a byte that is a multiple of four. The next four bytes define how many value:label pairs there are. Each pair is stored in succession, in order of increasing value, as a four-byte integer and a corresponding four-byte offset. The table illustrates this.
Table B.1: Bytecode layout for `lookupswitch`

<table>
<thead>
<tr>
<th>opcode (0xab) and padding</th>
<th>default offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of entries</td>
</tr>
<tr>
<td></td>
<td>value 1</td>
</tr>
<tr>
<td></td>
<td>offset 1</td>
</tr>
<tr>
<td></td>
<td>value 2</td>
</tr>
<tr>
<td></td>
<td>offset 2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>value N</td>
</tr>
<tr>
<td></td>
<td>offset N</td>
</tr>
</tbody>
</table>

**lor** (0x81) — long logical OR

Summary: Pops two longs from the stack, calculates their bitwise inclusive OR and pushes the 64-bit result as a long.

Initial: long-1

<table>
<thead>
<tr>
<th>long-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>long-2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>long</td>
</tr>
</tbody>
</table>

Final: long

**lrem** (0x71) — Long integer remainder

Summary: Pops two two-word integers, then pushes the remainder resulting when the next-to-top number is divided by the top number. This operation is rather like the C or Java % operation.

Initial: long-1

<table>
<thead>
<tr>
<th>long-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>long-2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>long-2</td>
</tr>
</tbody>
</table>

Final: long

**lreturn** (0xad) — Return from method with address result

Summary: Pops a two-byte long integer from current method stack. This long is pushed onto the method stack of the calling environment. The current method is terminated and control is transferred to the calling environment.

Initial: long

| long |

Final: (n/a)
APPENDIX B. JVM INSTRUCTION SET

**lshl** (0x79) — Shift long to the left

Summary: Pops an integer and a 64-bit long integer from the stack. The value of the long is shifted to the left the number of bits indicated by the lower six bits of the integer, then the resulting long is pushed. Newly emptied places are filled with 0 bits. This is equivalent to multiplying the value by a power of 2, but may be faster.

Initial: \( \text{int(shift)} \) long

Final: long

**lshr** (0x7b) — Shift long to the right

Summary: Pops an integer and a 64-bit long integer from the stack. The value of the long is shifted to the right the number of bits indicated by the lower six bits of the integer, then the resulting long is pushed. N.b., this is an arithmetic shift, meaning that the sign bit is copied to fill the newly emptied places.

Initial: \( \text{int(shift)} \) long

Final: long

**lstore <varnum>** (0x37 \{byte/short\}) — Store long in local variable

Summary: Pops long from top of stack and stores that long value in local variables \#<varnum> and \#<varnum>+1. The value of <varnum> is a byte in the range 0..255 unless the \text{wide} operand prefix is used, in which case it is a short in the range 0..65536.

Initial: long

Final: —

**lstore 0** (0x3f) — Store long in local variable \#0/#1

Summary: Pops long from top of stack and stores that long value in local variables \#0 and \#1. This is functionally equivalent to lstore 0, but takes fewer bytes and is faster.

Initial: long

Final: —

**lstore 1** (0x40) — Store long in local variable \#1/#2

Summary: Pops long from top of stack and stores that long value in local variables \#1 and \#2. This is functionally equivalent to lstore 1, but takes fewer bytes and is faster.

Initial: long

Final: —
**lstore.2** (0x41) — Store long in local variable #2/#3
Summary: Pops long from top of stack and stores that long value in local variables #2 and #3. This is functionally equivalent to `lstore 2`, but takes fewer bytes and is faster.

Initial: long
long

Final: —

**lstore.3** (0x42) — Store long in local variable #3/#4
Summary: Pops long from top of stack and stores that long value in local variables #3 and #4. This is functionally equivalent to `lstore 3`, but takes fewer bytes and is faster.

Initial: long
long

Final: —

**lsub** (0x65) — Long integer subtraction
Summary: Pops two two-word long integers, then pushes the result of the next-to-top number minus the top number.

Initial: long-1
long-1
long-2
long-2

Final: long
long

**lushr** (0x7d) — Shift unsigned long to the right
Summary: Pops an integer and a 64-bit long integer from the stack. The value of the long is shifted to the right the number of bits indicated by the lower six bits of the integer; then the resulting long is pushed. N.b., this is a logical shift, meaning that the sign bit is ignored and the bit-value 0 is used to fill the newly emptied places.

Initial: int(shift)
long
long

Final: long
long

**lxor** (0x83) — Long logical XOR
Summary: Pops two longs from the stack, calculates their bitwise XOR (exclusive OR) and pushes the 64-bit result as a long.

Initial: long-1
long-1
long-2
long-2

Final: long
long
**APPENDIX B. JVM INSTRUCTION SET**

**monitorenter (0xc2)** — Obtain lock on object

Summary: The JVM monitor system enables synchronization and coordinated access to objects among multiple threads. The `monitorenter` statement pops an address (object reference) and requests an exclusive lock on that object from the JVM. If no other thread has locked that object, the lock is granted and execution continues. If the object is already locked, the thread blocks and ceases to execute until the other thread releases the lock via `monitorexit`.

Initial: address
Final: —

**monitorexit (0xc3)** — Release lock on object

Summary: Pops an address (object reference) and releases a previously obtained (via `monitorenter`) lock on that object, enabling other threads to get locks in their turn.

Initial: address
Final: —

**multianewarray <type> <N> (0xc5 [short] [byte])** — Create multidimensional array

Summary: Allocates space for an `<N>`-dimensional array of type `<type>` and pushes a reference to the new array. The type is stored in bytecode as a two-byte index into the constant pool, while the number of dimensions `<N>` is stored as a byte value from 0..255. Executing this opcode pops `<N>` integer elements off the stack, representing the size of the array in each of the dimensions. The array is actually built as an array of (sub)arrays.

Initial: int(size N)
Initial: int(size 2)
Initial: int(size 1)
Final: address(array)

**multianewarray_quick (0xdf)** — Quick version of `multianewarray` opcode

Summary: Optimized version of `multianewarray` opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a `.class` file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

**new <class> (0xbb [short])** — Create new object

Summary: Creates a new object of the class specified. The type is stored internally as a two-byte index into the constant pool (q.v.)

Initial: —
Final: address(object)
new\_quick (0xdd) — Quick version of new opcode

Summary: Optimized version of new opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

newarray <typename> (0xbc [type-byte]) — Create unidimensional array of objects

Summary: Allocates space for an `<i>`-dimensional array of type `<typename>` and pushes a reference to the new array. The of the new array is popped as an integer from the top of the stack, while the type of the array is determined by examining the byte following this opcode according to the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>5</td>
</tr>
<tr>
<td>float</td>
<td>6</td>
</tr>
<tr>
<td>double</td>
<td>7</td>
</tr>
</tbody>
</table>

This can be used to initialize a new array of the specified type.

Initial: int(size)
Final: address(array)

nop (0x0) — No operation

Summary: Does nothing. An operation that does nothing is sometimes useful for timing or debugging, or as a placeholder for future code.

Initial: —
Final: —

pop (0x57) — Pop single word from stack

Summary: Pops and discards the top word (an integer, float, or address). Note that there is no matching push instruction as pushing is a typed operation; use, for instance, sipush or ldc.

Initial: word
Final: —

pop2 (0x58) — Pop two words from stack

Summary: Pops and discards the top two words (either two single-word quantities like integers, floats, or addresses, or a single two-word quantity such as a long or double). Note that there is no matching push instruction as pushing is a typed operation; use, for instance, ldc2w.

Initial: word
Final: —
**APPENDIX B. JVM INSTRUCTION SET**

`putfield <fieldname> <type> (0xb5 [short][short])` — Put object field

Summary: Pops an address (object reference) and value from the stack and stores that value into the identified field of the object. The `putfield` opcode takes two parameters, the field identifier and the field type, respectively. These are stored in the bytecode as two-byte indices into the *constant pool* (q.v.). Unlike in Java, the field name must always be a fully qualified name, including the name of the relevant class and any relevant packages.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>value address(object)</td>
<td>—</td>
</tr>
</tbody>
</table>

`putfield2_quick (0xd1)` — Quick version of `putfield` opcode for two-byte fields

Summary: Optimized version of `putfield` opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a `.class` file that isn’t currently loaded and being executed.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>see original opcode</td>
<td>see original opcode</td>
</tr>
</tbody>
</table>

`putfield_quick (0xcf)` — Quick version of `putfield` opcode

Summary: Optimized version of `putfield` opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a `.class` file that isn’t currently loaded and being executed.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>see original opcode</td>
<td>see original opcode</td>
</tr>
</tbody>
</table>

`putfield_quick.w (0xe4)` — Quick, wide version of `putfield` opcode

Summary: Optimized version of `putfield` opcodes used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a `.class` file that isn’t currently loaded and being executed.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>see original opcode</td>
<td>see original opcode</td>
</tr>
</tbody>
</table>

`putstatic <fieldname> <type> (0xb3 [short][short])` — Put class field

Summary: Pops a and value from the stack and stores that value in the identified field of the specified class. The `putstatic` opcode takes two parameters, the field identifier and the field type, respectively. These are stored in the bytecode as two-byte indices into the *constant pool* (q.v.). Unlike in Java, the field name must always be a fully qualified name, including the name of the relevant class and any relevant packages.

<table>
<thead>
<tr>
<th>Initial</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>—</td>
</tr>
</tbody>
</table>
**putstatic2_quick** (0xd5) — Alternate quick version of putstatic opcode

Summary: Optimized version of putstatic opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

**putstatic_quick** (0xd3) — Quick version of putstatic opcode

Summary: Optimized version of putstatic opcode used internally in Sun’s Just-in-time (JIT) compiler. This opcode should never appear in a .class file that isn’t currently loaded and being executed.

Initial: see original opcode
Final: see original opcode

**ret <varnum>** (0xa9 [byte/short]) — Return from subroutine

Summary: Returns to the address stored in <varnum> after a jump to a subroutine via jsr or jsr_w. The variable number is stored as a byte in the range 0..255 unless the wide prefix is used, which causes the variable to be stored as a two-byte quantity (range 0..65536) instead.

Initial: —
Final: —

**return** (0xb1) — Return from method without result

Summary: Terminates the current method and transfers control back to the calling environment.

Initial: —
Final: (n/a)

**saload** (0x35) — Load value from array of bytes

Summary: Pops an integer and an array from the stack, then retrieves a value from that location in the 1-dimensional array of 16-bit short. The value retrieved is sign-extended to an integer and pushed on the top of the stack.

Initial: int(index)
address(array ref)
Final: int
**sastore** (0x56) — Store value in array of characters

Summary: *Stores a 16-bit short integer in an array of shorts. The top argument popped is the index defining the array location to be used. The second argument popped is the short value to be stored, and the third and final argument is the array itself. The second argument is truncated from an int to a short and stored in the array.*

Initial: int(index)  
int(short)  
address(array ref)  
Final: —

**sipush** `<constant>` (0x11 [short]) — Push [integer] short

Summary: *The short value given as an argument (-32768..32767) is sign-extended to an integer and pushed on the stack*

Initial: —  
Final: int

**swap** (0x5f) — Swap top two stack elements

Summary: *Swaps the top two single-word elements on the stack. There is unfortunately no swap2 instruction.*

Initial: word-1  
word-2  
Final: word-2  
word-1
tables\texttt{witch} 1 3
  One
  Two
  Three
  default: Elsewhere

Figure B.2: Example of \texttt{tables\texttt{witch}}

\texttt{tables\texttt{witch} \langle args \rangle} (0xaa \langle args \rangle) — Computed branch
Summary: Performs a multiway branch, like the Java/C++ \texttt{switch} statement. The integer at the top of the stack is popped and compared to a set of value:label pairs. If the integer is equal to value, control is passed to the corresponding label. If no value matches the integer, control passes instead to a defined default label. Labels are implemented as relative offsets and added to the current contents of the program counter to get the location of the next instruction to execute. This instruction can be executed more efficiently than the similar \texttt{lookup\texttt{switch}} statement (q.v.), but the values need to be consecutive and sequential.

The arguments to \texttt{tables\texttt{witch}} include the lowest value and highest value represented by values in the table. If the integer popped from the stack is less than the lowest value, or greater than the highest value, control is transferred to the default label. Otherwise, control is transferred directly (without a need for comparisons) to the (integer-low)-th label in the table.

See the figure for an example of this operation in use.

The \texttt{tables\texttt{witch}} instruction has a variable number of arguments and is thus rather tricky in its bytecode storage. After the opcode (0xaa) itself, there follows from 0 to 3 bytes of padding, so that the four-byte default offset begins at a byte that is a multiple of four. The next eight bytes define the lowest table entry and highest table values. Each offset is then stored in numerical order as a four-byte offset. The table illustrates this.

\texttt{wide} (0xc4) — Specify “wide” interpretation of next opcode
Summary: This is not really an opcode, but an opcode prefix. It indicates that the arguments of the next operation are potentially larger than usual; for example, using a local variable greater than 255 in \texttt{iload}. The \texttt{jasmin} assembler will generate this prefix as needed automatically.
APPENDIX B. JVM INSTRUCTION SET

<table>
<thead>
<tr>
<th>opcode (0xa) and padding</th>
<th>default offset</th>
<th>low value</th>
<th>high value</th>
<th>offset 1</th>
<th>offset 2</th>
<th>...</th>
<th>offset N</th>
</tr>
</thead>
</table>

Table B.2: Bytecode layout for `tablesawitch`
Appendix C

Opcode Summary by Number

MARK ME NOTE TO EDITOR WE HAVE LOTS OF LEEWAY ON REFOR-MATTING THESE
### C.1 Standard Opcodes

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Opcode</th>
<th>Value</th>
<th>Opcode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>101</td>
<td>0x65</td>
<td>104</td>
<td>0x68</td>
</tr>
<tr>
<td>1</td>
<td>0x1</td>
<td>102</td>
<td>0x66</td>
<td>105</td>
<td>0x69</td>
</tr>
<tr>
<td>2</td>
<td>0x2</td>
<td>103</td>
<td>0x67</td>
<td>106</td>
<td>0x6a</td>
</tr>
<tr>
<td>3</td>
<td>0x3</td>
<td>107</td>
<td>0x6b</td>
<td>108</td>
<td>0x6c</td>
</tr>
<tr>
<td>4</td>
<td>0x4</td>
<td>109</td>
<td>0x6d</td>
<td>110</td>
<td>0x6e</td>
</tr>
<tr>
<td>5</td>
<td>0x5</td>
<td>111</td>
<td>0x6f</td>
<td>112</td>
<td>0x70</td>
</tr>
<tr>
<td>6</td>
<td>0x6</td>
<td>113</td>
<td>0x71</td>
<td>114</td>
<td>0x72</td>
</tr>
<tr>
<td>7</td>
<td>0x7</td>
<td>115</td>
<td>0x73</td>
<td>116</td>
<td>0x74</td>
</tr>
<tr>
<td>8</td>
<td>0x8</td>
<td>117</td>
<td>0x75</td>
<td>118</td>
<td>0x76</td>
</tr>
<tr>
<td>9</td>
<td>0x9</td>
<td>119</td>
<td>0x77</td>
<td>120</td>
<td>0x78</td>
</tr>
<tr>
<td>10</td>
<td>0xa</td>
<td>121</td>
<td>0x79</td>
<td>122</td>
<td>0x7a</td>
</tr>
<tr>
<td>11</td>
<td>0xb</td>
<td>123</td>
<td>0x7b</td>
<td>124</td>
<td>0x7c</td>
</tr>
<tr>
<td>12</td>
<td>0xc</td>
<td>125</td>
<td>0x7d</td>
<td>126</td>
<td>0x7e</td>
</tr>
<tr>
<td>13</td>
<td>0xd</td>
<td>127</td>
<td>0x7f</td>
<td>128</td>
<td>0x80</td>
</tr>
<tr>
<td>14</td>
<td>0xe</td>
<td>129</td>
<td>0x81</td>
<td>130</td>
<td>0x82</td>
</tr>
<tr>
<td>15</td>
<td>0xf</td>
<td>131</td>
<td>0x83</td>
<td>132</td>
<td>0x84</td>
</tr>
<tr>
<td>16</td>
<td>0x10</td>
<td>133</td>
<td>0x85</td>
<td>134</td>
<td>0x86</td>
</tr>
<tr>
<td>17</td>
<td>0x11</td>
<td>135</td>
<td>0x87</td>
<td>136</td>
<td>0x88</td>
</tr>
<tr>
<td>18</td>
<td>0x12</td>
<td>137</td>
<td>0x89</td>
<td>138</td>
<td>0x8a</td>
</tr>
<tr>
<td>19</td>
<td>0x13</td>
<td>139</td>
<td>0x8b</td>
<td>140</td>
<td>0x8c</td>
</tr>
<tr>
<td>20</td>
<td>0x14</td>
<td>141</td>
<td>0x8d</td>
<td>142</td>
<td>0x8e</td>
</tr>
<tr>
<td>21</td>
<td>0x15</td>
<td>143</td>
<td>0x8f</td>
<td>144</td>
<td>0x90</td>
</tr>
<tr>
<td>22</td>
<td>0x16</td>
<td>145</td>
<td>0x91</td>
<td>146</td>
<td>0x92</td>
</tr>
<tr>
<td>23</td>
<td>0x17</td>
<td>147</td>
<td>0x93</td>
<td>148</td>
<td>0x94</td>
</tr>
<tr>
<td>24</td>
<td>0x18</td>
<td>149</td>
<td>0x95</td>
<td>150</td>
<td>0x96</td>
</tr>
<tr>
<td>25</td>
<td>0x19</td>
<td>151</td>
<td>0x97</td>
<td>152</td>
<td>0x98</td>
</tr>
<tr>
<td>26</td>
<td>0x1a</td>
<td>153</td>
<td>0x99</td>
<td>154</td>
<td>0x9a</td>
</tr>
<tr>
<td>27</td>
<td>0x1b</td>
<td>155</td>
<td>0x9b</td>
<td>156</td>
<td>0x9c</td>
</tr>
<tr>
<td>28</td>
<td>0x1c</td>
<td>157</td>
<td>0x9d</td>
<td>158</td>
<td>0x9e</td>
</tr>
<tr>
<td>29</td>
<td>0x1d</td>
<td>159</td>
<td>0x9f</td>
<td>160</td>
<td>0xa0</td>
</tr>
<tr>
<td>30</td>
<td>0x1e</td>
<td>161</td>
<td>0xa1</td>
<td>162</td>
<td>0xa2</td>
</tr>
<tr>
<td>31</td>
<td>0x1f</td>
<td>163</td>
<td>0xa3</td>
<td>164</td>
<td>0xa4</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td>165</td>
<td>0xa5</td>
<td>166</td>
<td>0xa6</td>
</tr>
</tbody>
</table>

**APPENDIX C. OPCODE SUMMARY BY NUMBER**
### C.1. STANDARD OPCODES

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>FORMAT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 0x21</td>
<td>lload_3</td>
<td>load long value from local variable</td>
</tr>
<tr>
<td>34 0x22</td>
<td>fload_0</td>
<td>load float value from local variable</td>
</tr>
<tr>
<td>35 0x23</td>
<td>fload_1</td>
<td>load float value from local variable</td>
</tr>
<tr>
<td>36 0x24</td>
<td>fload_2</td>
<td>load float value from local variable</td>
</tr>
<tr>
<td>37 0x25</td>
<td>fload_3</td>
<td>load float value from local variable</td>
</tr>
<tr>
<td>38 0x26</td>
<td>dload_0</td>
<td>load double value from local variable</td>
</tr>
<tr>
<td>39 0x27</td>
<td>dload_1</td>
<td>load double value from local variable</td>
</tr>
<tr>
<td>40 0x28</td>
<td>dload_2</td>
<td>load double value from local variable</td>
</tr>
<tr>
<td>41 0x29</td>
<td>dload_3</td>
<td>load double value from local variable</td>
</tr>
<tr>
<td>42 0x2a</td>
<td>aload_0</td>
<td>load array value from local variable</td>
</tr>
<tr>
<td>43 0x2b</td>
<td>aload_1</td>
<td>load array value from local variable</td>
</tr>
<tr>
<td>44 0x2c</td>
<td>aload_2</td>
<td>load array value from local variable</td>
</tr>
<tr>
<td>45 0x2d</td>
<td>aload_3</td>
<td>load array value from local variable</td>
</tr>
<tr>
<td>46 0x2e</td>
<td>i2s</td>
<td>convert short to int</td>
</tr>
<tr>
<td>47 0x2f</td>
<td>l2f</td>
<td>convert long to float</td>
</tr>
<tr>
<td>48 0x30</td>
<td>i2d</td>
<td>convert int to double</td>
</tr>
<tr>
<td>49 0x31</td>
<td>f2i</td>
<td>convert float to int</td>
</tr>
<tr>
<td>50 0x32</td>
<td>f2l</td>
<td>convert float to long</td>
</tr>
<tr>
<td>51 0x33</td>
<td>f2d</td>
<td>convert float to double</td>
</tr>
<tr>
<td>52 0x34</td>
<td>d2i</td>
<td>convert double to int</td>
</tr>
<tr>
<td>53 0x35</td>
<td>d2l</td>
<td>convert double to long</td>
</tr>
<tr>
<td>54 0x36</td>
<td>d2f</td>
<td>convert double to float</td>
</tr>
<tr>
<td>55 0x37</td>
<td>saload</td>
<td>load short from array</td>
</tr>
<tr>
<td>56 0x38</td>
<td>faloa</td>
<td>load float from array</td>
</tr>
<tr>
<td>57 0x39</td>
<td>daloa</td>
<td>load double from array</td>
</tr>
<tr>
<td>58 0x3a</td>
<td>i2i</td>
<td>convert int to int</td>
</tr>
<tr>
<td>59 0x3b</td>
<td>ifix</td>
<td>fixup int value</td>
</tr>
<tr>
<td>60 0x3c</td>
<td>ifixl</td>
<td>fixup long value</td>
</tr>
<tr>
<td>61 0x3d</td>
<td>ifixd</td>
<td>fixup double value</td>
</tr>
<tr>
<td>62 0x3e</td>
<td>ifixf</td>
<td>fixup float value</td>
</tr>
<tr>
<td>63 0x3f</td>
<td>istore_0</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>64 0x40</td>
<td>istore_1</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>65 0x41</td>
<td>istore_2</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>66 0x42</td>
<td>istore_3</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>67 0x43</td>
<td>istore_4</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>68 0x44</td>
<td>istore_5</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>69 0x45</td>
<td>istore_6</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>70 0x46</td>
<td>istore_7</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>71 0x47</td>
<td>istore_8</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>72 0x48</td>
<td>istore_9</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>73 0x49</td>
<td>istore_10</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>74 0x4a</td>
<td>istore_11</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>75 0x4b</td>
<td>istore_12</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>76 0x4c</td>
<td>istore_13</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>77 0x4d</td>
<td>istore_14</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>78 0x4e</td>
<td>istore_15</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>79 0x4f</td>
<td>istore_16</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>80 0x50</td>
<td>istore_17</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>81 0x51</td>
<td>istore_18</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>82 0x52</td>
<td>istore_19</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>83 0x53</td>
<td>istore_20</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>84 0x54</td>
<td>istore_21</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>85 0x55</td>
<td>istore_22</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>86 0x56</td>
<td>istore_23</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>87 0x57</td>
<td>istore_24</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>88 0x58</td>
<td>istore_25</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>89 0x59</td>
<td>istore_26</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>90 0x5a</td>
<td>istore_27</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>91 0x5b</td>
<td>istore_28</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>92 0x5c</td>
<td>istore_29</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>93 0x5d</td>
<td>istore_30</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>94 0x5e</td>
<td>istore_31</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>95 0x5f</td>
<td>istore_32</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>96 0x60</td>
<td>istore_33</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>97 0x61</td>
<td>istore_34</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>98 0x62</td>
<td>istore_35</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>99 0x63</td>
<td>istore_36</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>100 0x64</td>
<td>istore_37</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>101 0x65</td>
<td>istore_38</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>102 0x66</td>
<td>istore_39</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>103 0x67</td>
<td>istore_40</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>104 0x68</td>
<td>istore_41</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>105 0x69</td>
<td>istore_42</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>106 0x6a</td>
<td>istore_43</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>107 0x6b</td>
<td>istore_44</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>108 0x6c</td>
<td>istore_45</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>109 0x6d</td>
<td>istore_46</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>110 0x6e</td>
<td>istore_47</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>111 0x6f</td>
<td>istore_48</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>112 0x70</td>
<td>istore_49</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>113 0x71</td>
<td>istore_50</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>114 0x72</td>
<td>istore_51</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>115 0x73</td>
<td>istore_52</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>116 0x74</td>
<td>istore_53</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>117 0x75</td>
<td>istore_54</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>118 0x76</td>
<td>istore_55</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>119 0x77</td>
<td>istore_56</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>120 0x78</td>
<td>istore_57</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>121 0x79</td>
<td>istore_58</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>122 0x7a</td>
<td>istore_59</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>123 0x7b</td>
<td>istore_60</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>124 0x7c</td>
<td>istore_61</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>125 0x7d</td>
<td>istore_62</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>126 0x7e</td>
<td>istore_63</td>
<td>store int to local variable</td>
</tr>
<tr>
<td>127 0x7f</td>
<td>istore_64</td>
<td>store int to local variable</td>
</tr>
</tbody>
</table>
APPENDIX C. OPCODE SUMMARY BY NUMBER

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 0x41</td>
<td>lstore_2</td>
</tr>
<tr>
<td>66 0x42</td>
<td>lstore_3</td>
</tr>
<tr>
<td>67 0x43</td>
<td>fstore_0</td>
</tr>
<tr>
<td>68 0x44</td>
<td>fstore_1</td>
</tr>
<tr>
<td>69 0x45</td>
<td>fstore_2</td>
</tr>
<tr>
<td>70 0x46</td>
<td>fstore_3</td>
</tr>
<tr>
<td>71 0x47</td>
<td>dstore_0</td>
</tr>
<tr>
<td>72 0x48</td>
<td>dstore_1</td>
</tr>
<tr>
<td>73 0x49</td>
<td>dstore_2</td>
</tr>
<tr>
<td>74 0x4a</td>
<td>dstore_3</td>
</tr>
<tr>
<td>75 0x4b</td>
<td>astore_0</td>
</tr>
<tr>
<td>76 0x4c</td>
<td>astore_1</td>
</tr>
<tr>
<td>77 0x4d</td>
<td>astore_2</td>
</tr>
<tr>
<td>78 0x4e</td>
<td>astore_3</td>
</tr>
<tr>
<td>79 0x4f</td>
<td>iastore</td>
</tr>
<tr>
<td>80 0x50</td>
<td>lastore</td>
</tr>
<tr>
<td>81 0x51</td>
<td>fastore</td>
</tr>
<tr>
<td>82 0x52</td>
<td>dastore</td>
</tr>
<tr>
<td>83 0x53</td>
<td>astore</td>
</tr>
<tr>
<td>84 0x54</td>
<td>bastore</td>
</tr>
<tr>
<td>85 0x55</td>
<td>castore</td>
</tr>
<tr>
<td>86 0x56</td>
<td>sastore</td>
</tr>
<tr>
<td>87 0x57</td>
<td>pop</td>
</tr>
<tr>
<td>88 0x58</td>
<td>pop2</td>
</tr>
<tr>
<td>89 0x59</td>
<td>dup</td>
</tr>
<tr>
<td>90 0x5a</td>
<td>dup_x1</td>
</tr>
<tr>
<td>91 0x5b</td>
<td>dup_x2</td>
</tr>
<tr>
<td>92 0x5c</td>
<td>dup2</td>
</tr>
<tr>
<td>93 0x5d</td>
<td>dup2_x1</td>
</tr>
<tr>
<td>94 0x5e</td>
<td>dup2_x2</td>
</tr>
<tr>
<td>95 0x5f</td>
<td>swap</td>
</tr>
<tr>
<td>96 0x60</td>
<td>iadd</td>
</tr>
<tr>
<td>97 0x61</td>
<td>ladd</td>
</tr>
<tr>
<td>98 0x62</td>
<td>fadd</td>
</tr>
<tr>
<td>99 0x63</td>
<td>dadd</td>
</tr>
<tr>
<td>100 0x64</td>
<td>isub</td>
</tr>
<tr>
<td>101 0x65</td>
<td>lsub</td>
</tr>
<tr>
<td>102 0x66</td>
<td>fsub</td>
</tr>
<tr>
<td>103 0x67</td>
<td>dsub</td>
</tr>
<tr>
<td>104 0x68</td>
<td>iinc</td>
</tr>
<tr>
<td>105 0x69</td>
<td>linc</td>
</tr>
<tr>
<td>106 0x6a</td>
<td>finc</td>
</tr>
<tr>
<td>107 0x6b</td>
<td>dincc</td>
</tr>
<tr>
<td>108 0x6c</td>
<td>i2l</td>
</tr>
<tr>
<td>109 0x6d</td>
<td>i2f</td>
</tr>
<tr>
<td>110 0x6e</td>
<td>i2d</td>
</tr>
<tr>
<td>111 0x6f</td>
<td>l2i</td>
</tr>
<tr>
<td>112 0x70</td>
<td>l2f</td>
</tr>
<tr>
<td>113 0x71</td>
<td>l2d</td>
</tr>
<tr>
<td>114 0x72</td>
<td>f2i</td>
</tr>
<tr>
<td>115 0x73</td>
<td>f2l</td>
</tr>
<tr>
<td>116 0x74</td>
<td>f2d</td>
</tr>
<tr>
<td>117 0x75</td>
<td>d2i</td>
</tr>
<tr>
<td>118 0x76</td>
<td>d2l</td>
</tr>
<tr>
<td>119 0x77</td>
<td>d2f</td>
</tr>
<tr>
<td>120 0x78</td>
<td>i2b</td>
</tr>
<tr>
<td>121 0x79</td>
<td>i2c</td>
</tr>
<tr>
<td>122 0x7a</td>
<td>i2s</td>
</tr>
<tr>
<td>123 0x7b</td>
<td>l2b</td>
</tr>
<tr>
<td>124 0x7c</td>
<td>l2c</td>
</tr>
<tr>
<td>125 0x7d</td>
<td>l2s</td>
</tr>
<tr>
<td>126 0x7e</td>
<td>f2s</td>
</tr>
<tr>
<td>127 0x7f</td>
<td>d2s</td>
</tr>
<tr>
<td>128 0x80</td>
<td>i2byte</td>
</tr>
<tr>
<td>129 0x81</td>
<td>i2char</td>
</tr>
<tr>
<td>130 0x82</td>
<td>i2short</td>
</tr>
<tr>
<td>131 0x83</td>
<td>l2byte</td>
</tr>
<tr>
<td>132 0x84</td>
<td>l2char</td>
</tr>
<tr>
<td>133 0x85</td>
<td>l2short</td>
</tr>
<tr>
<td>134 0x86</td>
<td>f2byte</td>
</tr>
<tr>
<td>135 0x87</td>
<td>f2char</td>
</tr>
<tr>
<td>136 0x88</td>
<td>f2short</td>
</tr>
<tr>
<td>137 0x89</td>
<td>d2byte</td>
</tr>
<tr>
<td>138 0x8a</td>
<td>d2char</td>
</tr>
<tr>
<td>139 0x8b</td>
<td>d2short</td>
</tr>
<tr>
<td>140 0x8c</td>
<td>i2short</td>
</tr>
<tr>
<td>141 0x8d</td>
<td>i2java_long</td>
</tr>
<tr>
<td>142 0x8e</td>
<td>i2java_int</td>
</tr>
<tr>
<td>143 0x8f</td>
<td>i2java_byte</td>
</tr>
<tr>
<td>144 0x90</td>
<td>l2short</td>
</tr>
<tr>
<td>145 0x91</td>
<td>l2java_long</td>
</tr>
<tr>
<td>146 0x92</td>
<td>l2java_int</td>
</tr>
<tr>
<td>147 0x93</td>
<td>l2java_byte</td>
</tr>
<tr>
<td>148 0x94</td>
<td>f2short</td>
</tr>
<tr>
<td>149 0x95</td>
<td>f2java_long</td>
</tr>
<tr>
<td>150 0x96</td>
<td>f2java_int</td>
</tr>
<tr>
<td>151 0x97</td>
<td>f2java_byte</td>
</tr>
<tr>
<td>152 0x98</td>
<td>d2short</td>
</tr>
<tr>
<td>153 0x99</td>
<td>d2java_long</td>
</tr>
<tr>
<td>154 0x9a</td>
<td>d2java_int</td>
</tr>
<tr>
<td>155 0x9b</td>
<td>d2java_byte</td>
</tr>
<tr>
<td>156 0x9c</td>
<td>multianewarray</td>
</tr>
<tr>
<td>157 0x9d</td>
<td>anewarray</td>
</tr>
<tr>
<td>158 0x9e</td>
<td>arraylength</td>
</tr>
<tr>
<td>159 0x9f</td>
<td>tableswitch</td>
</tr>
<tr>
<td>160 0xa0</td>
<td>lookupswitch</td>
</tr>
<tr>
<td>161 0xa1</td>
<td>ireturn</td>
</tr>
<tr>
<td>162 0xa2</td>
<td>dreturn</td>
</tr>
<tr>
<td>163 0xa3</td>
<td>freturn</td>
</tr>
<tr>
<td>164 0xa4</td>
<td>lreturn</td>
</tr>
<tr>
<td>165 0xa5</td>
<td>goto</td>
</tr>
<tr>
<td>166 0xa6</td>
<td>if_acmpeq</td>
</tr>
<tr>
<td>167 0xa7</td>
<td>if_icmpne</td>
</tr>
<tr>
<td>168 0xa8</td>
<td>jsr</td>
</tr>
<tr>
<td>169 0xa9</td>
<td>ret</td>
</tr>
<tr>
<td>170 0xaa</td>
<td>goto_w</td>
</tr>
<tr>
<td>171 0xab</td>
<td>jsr_w</td>
</tr>
<tr>
<td>172 0xac</td>
<td>ireturn</td>
</tr>
<tr>
<td>173 0xad</td>
<td>dreturn</td>
</tr>
<tr>
<td>174 0xae</td>
<td>freturn</td>
</tr>
<tr>
<td>175 0xaf</td>
<td>lreturn</td>
</tr>
</tbody>
</table>

C.2 Reserved opcodes

The JVM standard also reserves the following opcodes.

202 0xca breakpoint
254 0xfe impdep1
255 0xff impdep2

Opcode 202 (breakpoint) is used by debuggers, while opcodes 254 and 255 are reserved for internal use by the JVM itself. They should never appear in a store.
C.3. “QUICK” PSEUDO-OPCODES USED BY SUN IN INTERNAL OPTIMIZATION

In 1995, researchers at Sun proposed the use of internal “quick” opcodes as a method of increasing the speed and efficiency of the Java compiler. Normally, when an entry in the constant pool is referenced, the entry must be resolved to confirm its availability and type compatibility. If a single statement must be executed several times, this re-resolution can slow the computer down. As part of its Just In Time (JIT) compiler, Sun has proposed a set of opcodes that assume the entry has already been resolved. When a normal opcode is executed (successfully), it can then be replaced internally with a “quick” pseudo-opcode to skip the resolution step and speed up subsequent executions.

These pseudo-opcodes should never appear in a class file in long-term storage. Instead, the JVM itself may rewrite the opcodes on an executing class. If done properly, this kind of change is completely invisible to a Java/jasmin programmer, or even a writer of compilers. The set of optimization pseudo-opcodes proposed includes the following:

\begin{verbatim}
203 0xcb ldc_quick
205 0xcd ldc_w_quick
206 0xce getfield_quick
207 0xcf putfield_quick
208 0xd0 getfield2_quick
209 0xd1 putfield2_quick
210 0xd2 getstatic_quick
211 0xd3 putstatic_quick
212 0xd4 getstatic2_quick
213 0xd5 putstatic2_quick
214 0xd6 invokevirtual_quick
215 0xd7 invokespecial_quick
216 0xd8 invokeclass_quick
217 0xda invokeinterface_quick
219 0xdb invokevirtualobject_quick
221 0xdd new_quick
222 0xde anewarray_quick
223 0xdf multianewarray_quick
224 0xe0 checkcast_quick
225 0xe1 instanceof_quick
226 0xe2 invokevirtual_quick_w
227 0xe3 getfield_quick_w
228 0xe4 putfield_quick_w
\end{verbatim}

There is, of course, nothing to prevent a different implementor from using a
different optimization method or set of “quick” opcodes.

C.4 Unused opcodes

Opcode 186 is unused “for historical reasons,” as its previous use is no longer valid in the current version of the JVM. Opcodes 204, 220, and 229–253 are unassigned in the current JVM specifications, but might acquire assignments (and uses) in later versions.
Appendix D

Class file format

D.1 Overview and fundamentals

As alluded to briefly in chapter 10, class files for the JVM are stored as a set of nested tables. This is actually a slight misnomer, as the same format is used whenever classes are stored or transmitted, so a class received over the network comes across in exactly the same format — if not a “file.” Each “file” contains the information needed for exactly one class, including the bytecode for all of the methods, fields and data internal to the class, and properties for interacting with the rest of the class system including name and inheritance details.

All data in class files is stored as 8-bit bytes or as multiple byte groups of 16-bits, 32-bits, or 64-bits. These are referred to in standards documents as u1, u2, u4 and u8, respectively, but it’s probably easier to think of them just as bytes, shorts, ints, and long. To prevent any confusion between machines with different storage conventions (such as the 8088 with its byte-swapping), the bytes are defined as coming in “network order” (also called “big-endian” or “MSB” order), where the “most significant” byte comes first. For example, the first four bytes of any JVM class file must be the so-called magic number 0xCAFEBAEBE (an int, obviously). This would be stored as a sequence of four bytes in the order 0xCA, 0xFE, 0xBA, 0xBE.

The top-level table of a class file contains a little bit of housekeeping information (of fixed size) and five variable-sized lower-level tables. There is no padding or alignment between the various components, which makes it a little bit tricky to pull a specific part (such as the name of a method) out of a class file.

The detailed top-level format of a class file is as follows:
APPENDIX D. CLASS FILE FORMAT

<table>
<thead>
<tr>
<th>Size</th>
<th>Identifier</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>magic number</td>
<td>Defined value of 0xCAFEBABE</td>
</tr>
<tr>
<td>short</td>
<td>minor version</td>
<td>Defines which version (major and minor revision)</td>
</tr>
<tr>
<td>short</td>
<td>major version</td>
<td>of JVM class file is compatible with</td>
</tr>
<tr>
<td>short</td>
<td>constant pool</td>
<td>Maximum entry number in following table</td>
</tr>
<tr>
<td>variable</td>
<td>constant pool</td>
<td>Pool of constants used by class</td>
</tr>
<tr>
<td>short</td>
<td>access flags</td>
<td>Valid access types (public, static, interface, etc.)</td>
</tr>
<tr>
<td>short</td>
<td>this class</td>
<td>Identifier of this class type</td>
</tr>
<tr>
<td>short</td>
<td>super class</td>
<td>Identifier of this class’s superclass type</td>
</tr>
<tr>
<td>short</td>
<td>interfaces count</td>
<td>Number of entries in following table</td>
</tr>
<tr>
<td>variable</td>
<td>interfaces</td>
<td>Interfaces implemented by this class</td>
</tr>
<tr>
<td>short</td>
<td>fields count</td>
<td>Number of entries in following table</td>
</tr>
<tr>
<td>variable</td>
<td>fields</td>
<td>Fields declared as part of this class</td>
</tr>
<tr>
<td>short</td>
<td>methods count</td>
<td>Number of entries in following table</td>
</tr>
<tr>
<td>variable</td>
<td>methods</td>
<td>Methods defined as part of this class</td>
</tr>
<tr>
<td>short</td>
<td>attributes count</td>
<td>Number of entries in following table</td>
</tr>
<tr>
<td>variable</td>
<td>attributes</td>
<td>Other attributes of this class</td>
</tr>
</tbody>
</table>

The “magic number” has already been described: it serves no real purpose except to make it easy to identify class files quickly on a system. The major and minor version numbers help track compatibility. For example, a very old class file (or a class file compiled with a very old version of Java) might use opcodes that no longer exist, or that have changed semantics. The current version of Java (as of summer 2004) uses minor version 3 and major version 45 (0x2d).

The fields for this class and the super class refer to entries in the constant table (see the next section for details), and define the name of the current class, as well as the immediate superclass. Finally, the access flags field defines the access-related properties of the current class; for example, if this class is defined as abstract, which prevents other classes from using it as an argument to the new instruction. These properties are stored as individual flags in a single two-word bit vector as follows:

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Bit value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>0x0001</td>
<td>Class is accessible to others</td>
</tr>
<tr>
<td>final</td>
<td>0x0010</td>
<td>Class cannot be subclassed</td>
</tr>
<tr>
<td>super</td>
<td>0x0020</td>
<td>New invocation semantics</td>
</tr>
<tr>
<td>interface</td>
<td>0x0200</td>
<td>File is actually an interface</td>
</tr>
<tr>
<td>abstract</td>
<td>0x0400</td>
<td>Class can’t be instantiated</td>
</tr>
</tbody>
</table>

So an access flags field of 0x0601 would define a “public” “abstract” “interface.”

D.2 Subtable structures

D.2.1 Constant pool

The constant pool is structured as a sequence of individual entries representing constants used by the program. For example, a constant representing the integer value 1010 would be stored in five successive bytes. The last four bytes would be the binary representation of 1010 (as an integer), while the first byte is a tag value
D.2. SUBTABLE STRUCTURES

defining this entry as an integer (and thus as having five bytes). Depending upon
the tag types, the size and internal format of entries varies as in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF8 String</td>
<td>1</td>
</tr>
<tr>
<td>Integer</td>
<td>3</td>
</tr>
<tr>
<td>Float</td>
<td>4</td>
</tr>
<tr>
<td>Long</td>
<td>5</td>
</tr>
<tr>
<td>Double</td>
<td>6</td>
</tr>
<tr>
<td>Class</td>
<td>7</td>
</tr>
<tr>
<td>String</td>
<td>8</td>
</tr>
<tr>
<td>Field reference</td>
<td>9</td>
</tr>
<tr>
<td>Method reference</td>
<td>10</td>
</tr>
<tr>
<td>Interface method reference</td>
<td>11</td>
</tr>
<tr>
<td>Name and Type</td>
<td>12</td>
</tr>
</tbody>
</table>

The structure of integers, longs, floats, and doubles is self-explanatory. For
example, a constant pool entry for an integer consists of five bytes, an initial byte
with the value 3 (defining the entry as an integer) and four bytes for the integer
value itself. UTF8 strings are stored as an unsigned length value (two bytes in
length, allowing string of up to 65,536 characters) and a variable-length array of
bytes containing the character values in the string. All literal strings in Java class
files — string constants, class names, methods and field names, etc. — are stored
internally as UTF8 string constants.

The internal fields in other types refer to the (two-byte) indices of other entries
in the constant pool. For example, a “field” reference contains five bytes. The
first byte would be the tag value defining a field (value 9). The second and third
bytes would hold the index of another entry in the constant pool defining the class
to which the field belongs. The fourth and fifth hold the index of a “name and
type” entry defining the name and field. This “name and type” entry, would have
an appropriate tag (value 12), then the indices of two UTF8 strings defining the
name/type.

There are two important caveats regarding the constant pool. For historical
reasons, constant pool entry number zero is never used, and the initial element
takes index zero. For this reason, unlike most other array-type elements in Java
(and other class file entries), if there are $k$ constant pool entries, the highest entry
is $k$, not $k-1$. Also, for historical reasons, constant pool entries of type long or
double are treated as two entries; if constant pool entry 6 is a long, then the next
pool entry would have index 8. Index 7, in this case, would be unused and illegal.

D.2.2 Field table

Each field of the class is defined internally as a table entry with the following fields:
### APPENDIX D. CLASS FILE FORMAT

<table>
<thead>
<tr>
<th>Size</th>
<th>Identifier</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>access flags</td>
<td>access properties of this field</td>
</tr>
<tr>
<td>short</td>
<td>name</td>
<td>index of name in constant pool</td>
</tr>
<tr>
<td>short</td>
<td>descriptor</td>
<td>index of type string</td>
</tr>
<tr>
<td>short</td>
<td>attributes count</td>
<td>number of field attributes</td>
</tr>
<tr>
<td>variable</td>
<td>attributes</td>
<td>array of field attributes</td>
</tr>
</tbody>
</table>

The name and type fields are simply indices into the constant pool for the name and type descriptor strings, respectively, of the field. The access flags field is a bit vector of flags as before (interpretation given by the following table) defining valid access attributes for the field. Finally, the attributes table defines field attributes as the class attributes table does for the class as a whole.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Bit value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>0x0001</td>
<td>Field is accessible to others</td>
</tr>
<tr>
<td>private</td>
<td>0x0002</td>
<td>Field is usable only to defining class</td>
</tr>
<tr>
<td>protected</td>
<td>0x0004</td>
<td>Field is accessible to class and subclasses</td>
</tr>
<tr>
<td>static</td>
<td>0x0008</td>
<td>Class, not instance, field</td>
</tr>
<tr>
<td>final</td>
<td>0x0010</td>
<td>Field cannot be changed</td>
</tr>
<tr>
<td>volatile</td>
<td>0x0040</td>
<td>Field cannot be cached</td>
</tr>
<tr>
<td>transient</td>
<td>0x0080</td>
<td>Field cannot be written/read by object mgr.</td>
</tr>
</tbody>
</table>

### D.2.3 Methods table

Each method defined in a class is described internally as a table entry of almost identical format as the field entry described above. The only difference is the specific values used to represent different access flags, as given in the following table.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Bit value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>0x0001</td>
<td>Field is accessible to others</td>
</tr>
<tr>
<td>private</td>
<td>0x0002</td>
<td>Field is usable only to defining class</td>
</tr>
<tr>
<td>protected</td>
<td>0x0004</td>
<td>Field is accessible to class and subclasses</td>
</tr>
<tr>
<td>static</td>
<td>0x0008</td>
<td>Class, not instance, field</td>
</tr>
<tr>
<td>final</td>
<td>0x0010</td>
<td>Field cannot be changed</td>
</tr>
<tr>
<td>synchronized</td>
<td>0x0020</td>
<td>Invocation is locked</td>
</tr>
<tr>
<td>native</td>
<td>0x0040</td>
<td>Implemented in hardware-native language</td>
</tr>
<tr>
<td>abstract</td>
<td>0x0100</td>
<td>No implementation defined</td>
</tr>
<tr>
<td>strict</td>
<td>0x0800</td>
<td>Strict floating point semantics</td>
</tr>
</tbody>
</table>

### D.2.4 Attributes

Almost all parts of the class file, including the top-level table itself, contains a possible attributes subtable. This subtable contains “attributes” created by the compiler to describe or support computations. Each individual compiler is permitted to define specific attributes, and JVM implementations are required to ignore attributes that they don’t recognize, so this chapter cannot provide a definitive and complete list of possible attributes. On the other hand, there are certainly attributes that must be present, and the JVM may not run correctly if it requires a certain attribute that isn’t present.
### D.2. SUBTABLE STRUCTURES

Each attribute is stored as a table of the following format:

<table>
<thead>
<tr>
<th>Size</th>
<th>Identifier</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>attribute name</td>
<td>Name of attribute</td>
</tr>
<tr>
<td>int</td>
<td>attribute length</td>
<td>Length of attribute in bytes</td>
</tr>
<tr>
<td>variable</td>
<td>info</td>
<td>Contents of attribute</td>
</tr>
</tbody>
</table>

Probably the most obvious (and important) attribute is the **Code** attribute, which, as an attribute for a method, contains the bytecode for that particular method. The **Exceptions** attribute defines the type of exceptions that a particular method may throw. In support of debuggers, the **Sourcefile** attribute stores the name of the source file from which this class file was created, and the **LineNumberTable** stores which byte(s) in bytecode correspond to which individual lines in the source code. The **LocalVariableTable** attribute will similarly define which variable (in the source file) corresponds to which local variable in the JVM. Compiling (or assembling) with the `-g` flag (on a *NIX system) will usually cause these attributes to be put into the class file. Without this flag, they are often omitted to save space and time.
Appendix E

The ASCII Table

E.1 The table

<table>
<thead>
<tr>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>nul</td>
<td>01</td>
<td>soh</td>
<td>02</td>
<td>stx</td>
<td>03</td>
<td>etx</td>
<td>04</td>
<td>eot</td>
<td>05</td>
<td>enq</td>
<td>06</td>
<td>ack</td>
</tr>
<tr>
<td>08</td>
<td>bs</td>
<td>09</td>
<td>ht</td>
<td>0a</td>
<td>la</td>
<td>0b</td>
<td>vt</td>
<td>0c</td>
<td>ff</td>
<td>0d</td>
<td>cr</td>
<td>0e</td>
<td>so</td>
</tr>
<tr>
<td>20</td>
<td>sp</td>
<td>21</td>
<td>!</td>
<td>22</td>
<td>&quot;</td>
<td>23</td>
<td>#</td>
<td>24</td>
<td>$</td>
<td>25</td>
<td>%</td>
<td>26</td>
<td>&amp;</td>
</tr>
<tr>
<td>30</td>
<td>(</td>
<td>31</td>
<td>)</td>
<td>32</td>
<td>*</td>
<td>33</td>
<td>+</td>
<td>34</td>
<td>,</td>
<td>35</td>
<td>-</td>
<td>36</td>
<td>.</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>39</td>
<td>1</td>
<td>40</td>
<td>2</td>
<td>41</td>
<td>3</td>
<td>42</td>
<td>4</td>
<td>43</td>
<td>5</td>
<td>44</td>
<td>6</td>
</tr>
<tr>
<td>46</td>
<td>8</td>
<td>47</td>
<td>9</td>
<td>48</td>
<td>0</td>
<td>49</td>
<td>a</td>
<td>4a</td>
<td>b</td>
<td>4b</td>
<td>c</td>
<td>4c</td>
<td>d</td>
</tr>
<tr>
<td>4e</td>
<td>f</td>
<td>4f</td>
<td>g</td>
<td>50</td>
<td>h</td>
<td>51</td>
<td>i</td>
<td>52</td>
<td>j</td>
<td>53</td>
<td>k</td>
<td>54</td>
<td>l</td>
</tr>
<tr>
<td>56</td>
<td>n</td>
<td>57</td>
<td>o</td>
<td>58</td>
<td>p</td>
<td>59</td>
<td>q</td>
<td>5a</td>
<td>r</td>
<td>5b</td>
<td>s</td>
<td>5c</td>
<td>t</td>
</tr>
<tr>
<td>5e</td>
<td>v</td>
<td>5f</td>
<td>w</td>
<td>60</td>
<td>x</td>
<td>61</td>
<td>y</td>
<td>62</td>
<td>z</td>
<td>63</td>
<td>{</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>79</td>
<td>7a</td>
<td>7b</td>
<td>7c</td>
<td>7d</td>
<td>7e</td>
<td>7f</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
</tr>
</tbody>
</table>

Note that ASCII 0x20 (32 decimal) is a space (" ") character.

E.2 History and overview

ASCII, the American Standard Code for Information Interchange, has for decades (it was formalized in 1963, internationalized in 1968) been the most common standard for encoding character data in binary format. Other once-common formats, such as EBCDIC and Baudot are largely of historical interest only. The original ASCII character set defined a 7-bit standard for encoding 128 different “characters,” including a complete set of both upper and lower case letters (for American English), digits, and many symbols and punctuation marks. In addition, the first 32 entries in the ASCII table define mostly unprintable control characters such as a backspace (entry 0x08), horizontal tab (entry 0x09), vertical tab (0x10), or even an audible “bell” (now usually a beep, entry 0x07). Unfortunately, ASCII does not well support non-English languages, or even many commonly useful symbols such as →, ≤, or the British pound sign (£).

Since almost all computers store 8-bit bytes, byte values 0x80...0xFF, which
do not have a “standard” character interpretation, are often used for machine-
dependent proprietary extensions to the ASCII table. For example, the letter ö is
used in German, but not English. Microsoft has defined an extended ASCII table
(among several different sets in use by various Windows-based programs) that uses
entry 0x94 to represent this value as part of a fairly complete set of German-specific
characters; this same table defines entry 0xE2 as an uppercase Γ, but oddly enough
does not define an entry for lowercase γ. I suppose the German-speaking market
was more important to the designers of this character set than the Greek-speaking
one. Apple, by contrast, does not define a meaning for the extended characters (at
least for its ‘terminal’ environment under OS X).

The core of the problem is that a single byte, with only 256 different storage
patterns, can’t store enough different characters. The solution taken by Java is to
use a larger character set (Unicode) and store them as two-byte quantities instead.
Appendix F

Glossary of Terms

**80x86 family**  A family of chips manufactured by Intel, beginning with the Intel 4004 and extending through the 8008, 8088, 80086, 80286, 80386, 80486, and the various Pentiums. These chips form the basis of the IBM-PC and its successors and are the most common chip architecture in use today.

**absolute address**  The 20-bit address obtained by combining the segment:offset memory address in an 8086-base computer.

**abstract**  A class that contains no direct instances, only subclasses. Alternatively, an unimplemented method that must be implemented in subclasses.

**accumulator**  A designated single register for high speed arithmetic, particularly addition and multiplication. On 80x86 computers, the [E]AX register.

**actual parameter**  The actual value used in place of the formal parameter in a call to a function or method.

**address**  A location in memory, alternatively, a number used to refer to a location in memory.

**addressing mode**  The way to interpret a bit pattern to define the actual operand for a statement. For example, the bit pattern 0x0001 could refer to the actual constant 1, the first register, the contents of memory location 1, and so forth. See individual modes: immediate mode, register mode, direct mode, indirect mode, index mode.

**algorithm**  A step-by-step, unambiguous procedure for achieving a desired goal or performing a computation.

**ALU**  A component of a typical machine architecture where the arithmetic and logical operations are performed. Part of the CPU.

**array**  A derived type, a collection of subelements of identical type, indexed by an integer.
American Standard Code for Information Interchange  See ASCII.

AND  A boolean function that returns True if and only if all arguments are True, and otherwise returns false. An AND gate is a hardware circuit that implements an AND function on the input electrical signals.

applet  A small portable program (application), typically delivered as part of a Web page.

arithmetic shift  A shift operation where the leftmost (rightmost) bit is duplicated to fill the emptied bit locations.

Arithmetic and Logical Unit  See ALU.

ALU  {A component of a typical machine architecture where the arithmetic and logical operations are performed. Part of the CPU.

array  A derived type, a collection of subelements of identical type, indexed by an integer.

ASCII  A standard way of representing character data in binary-format. The ASCII set defines a 7-bit pattern for letters, digits, and some commonly used punctuation marks.

assembler  A program to convert a source file written in assembly language to an executable file.

assembly language  A low-level language where each human-readable statement corresponds to exactly one machine instruction. Different computer types will have different assembly languages.

attributes  A data structure in the JVM class file format used to store miscellaneous information.

backwards compatibility  A computer or system is backwards compatible when it is capable of duplicating the operations of previous versions. For example, the Pentium is backwards compatible with the 8088 and so will still run programs written for the original IBM-PC.

base  1. The numeric value represented by a digit-place in numbering system. For example, binary is base-2, while decimal is base-10 and hexadecimal is base-16.

  2. The electrical connection in a transistor that controls the flow of current from the emitter to the collector.

big-endian  A storage format where the most significant bit is stored as the first and highest numbered bit in a word. In big-endian format, the number 32770 would be stored as binary 10000010.

binary  1. A number system where all digits are 1 or 0 and successive digits are larger by a factor of two. The number 31 in binary would be written as 11111.
2. A mathematical operator such as addition that takes exactly two operands.

**Binary Coded Decimal**  See BCD

**BCD**  A method used by the math processor in the 80x86 family where each decimal digit of a number is written as a corresponding four-bit binary pattern. The number 4096, for example, would be written in BCD as 0100 0000 1001 0110.

**bit**  A “binary digit,” the basic unit of information inside a computer. A bit can take two basic values, 0 or 1.

**bitwise**  Operating bit-by-bit, for example, taking the AND of two 32-bit numbers by taking the individual ANDs of their corresponding bits.

**block address translation**  A method of translating logical addresses inside a memory manager to a fixed block of physical memory.

**boolean logic**  A logic system, named after George Boole, where operations are defined and performed as functions on binary quantities.

**branch**  A machine instruction that changes the value of the program counter and thus causes the computer to begin executing at a different memory location. An equivalent term is goto.

**branch prediction**  An optimization technique to speed up a computer by predicting whether or not a conditional branch will be taken.

**bus**  A component of a typical machine architecture that acts as a connection between the CPU, the memory, and/or peripherals.

**busy-wait**  Waiting for an expected event by checking to see if the event has happened inside a loop. Compare interrupt.

**byte**  A collection of 8 bits. Used as a unit of memory, of representation size, or of register capacity.

**bytecode**  The machine language of the JVM.

**cache memory**  A bank of high-speed memory to store frequently accessed items to improve overall memory performance.

**Central Processing Unit**  See CPU.

**CF**  The Carry Flag, set when the most recent operation generated a carry out of the register, as when two numbers are added for which the sum is too large.

**CISC**  Complex Instruction Set Computing. A computer design philosophy using many complicated special-purpose instructions. Compare RISC

**class**  A collection of fields and methods defining a type of object in an object-oriented programming environment.

**class files**  The format used by the JVM to store classes (including records and interfaces) in long-term storage or to deliver them over a network.
class method  A method defined by an object-oriented system as a property of a class, rather than of any particular instance of that class.

class variable  A variable defined by an object-oriented system as a property of a class, rather than of any particular instance of that class.

clock signal  An electrical signal used by the computer to synchronize events and to measure the passage of time.

code  Executable machine instructions, as opposed to data.

Common Language Runtime  The virtual machine underlying Microsoft’s .NET framework.

compiler  A program to convert a source file written in a high-level language to an executable file.

complete/writeback  A typical phase in a pipelined architecture, where the results of an operation are stored in the target locations.

conditional branch  A branch (like if/else) that may or may not be taken, depending upon the current machine state.

constant pool  The set of constants used by a particular JVM class, as stored in the class file.

control characters  The ASCII characters with values below 0x20, which represent non-printing characters such as a carriage return or a ringing bell.

Control Unit  The part of the CPU that moves data in and out of the CPU and determining which instruction to execute.

CPU  The heart of the computer, where calculations take place and the program is actually executed. Usually consists of the Control Unit plus the ALU.

data memory  Memory used to store program data (such as variables) instead of program code.

decimal  Base 10. The usual way of writing numbers that you have been familiar with.

derived type  A representation for data built up by combining basic types. For example, a fraction type could be derived from two integers, the numerator and the denominator.

destination  Where data goes. For example, in the instruction istore_3, the destination is local variable #3.

destination index  A register in the 80x86 family that controls the destination of string primitive operations.

device  Another name for a peripheral.

device driver  A program (or part of the operating system) that controls a device.
diode An electrical component that will only let electricity pass in one direction; part of the makeup of a transistor.

Direct Memory Access The capacity of a computer to let data move between main memory and a peripheral like a graphics card without having to go through the CPU.

direct mode An addressing mode where a bit pattern is interpreted as a location in memory holding the desired operand. In direct mode, the bit pattern 0x0001 would be interpreted as memory location 1.

dispatch A typical state in a pipelined architecture where the computer analyzes the instruction to determine what kind of instruction it is, gets the source arguments from the appropriate locations, and prepares the instruction for actual execution.

DRAM A kind of RAM that requires continuous refreshing to hold data. Slower but cheaper than SRAM. Like all RAM, if the power goes out, the memory loses its data.

dst An abbreviation for destination.

Dynamic Random Access Memory See DRAM.

EEPROM A kind of hybrid memory combining the field programmability of RAM with the data persistence of ROM.

Electronically Erasable Programmable ROM See EEPROM.

embedded system A computer system where the computer is an integral part of a larger environment, and not an independently usable tool. For example, the computer that runs a DVD player.

emitter Part of a standard transistor, through which current flows to the collector unless shut off at the base.

Erasable Programmable ROM A type of PROM that can be erased, typically by several seconds of exposure to high-powered ultraviolet light. These memories are reprogrammable, but typically not in the field.

execute 1. To run a program or machine instruction

2. A typical state in a pipelined architecture where the computer actually runs a previously fetched (and dispatched) instruction.

exponent A field in an IEEE floating point representation controlling the power of two by which the mantissa is multiplied.

extended AX register The 32-bit accumulator on an 80386 or later chip in the 80x86 family, including the Pentium.

fetch 1. To load a machine instruction in prepare for performing it
2. A typical state in a **pipelined** architecture where the computer loads an instruction from main memory.

**fetch**  The process by which a computer loads an instruction to be executed. Also, one of the stages in a typical **pipelined** architecture where such a process is performed.

**fetch-execute cycle**  The process by which a computer **fetches** an instruction to be performed, **executes** that instruction, and then cyclically fetches the next instruction in sequence until the end of the program.

**Fibonacci sequence**  The sequence 1,1,2,3,..., where every term is the sum of its two immediate predecessors.

**fields**  Named data storage locations in a record or class.

**flags**  Binary variables used to store data. See also **flags register**

**flags register**  A special register in the CPU that holds a set of binary **flags** regarding the current state of computation. For example, if machine overflow occurs on an arithmetic calculation, a flag (typically called the “overflow flag” or **OF** will be set to 1). A later **conditional branch** can examine this flag.

**flash**  A kind of **hybrid** memory combining the field programmability of **RAM** with the data persistency of **ROM**. Commonly used in pen-drives and digital cameras.

**floating point**  1. Any non-integer value as stored by a computer.

2. A specific format for storing non-integer values in a binary form of scientific notation. Values are stored as the product of a **mantissa** times two raised to the power of a biased **exponent**.

**formal parameter**  The variables used in the definition of a function or method that serve as place-holders for later **actual parameters**.

**garbage collection**  The process of reclaiming memory locations that are no longer in use. Automatic on the JVM.

**gates**  Electrical circuits that implement **boolean** functions.

**goto**  See **branch**.

**Harvard architecture**  A kind of non-von Neumann architecture where code storage (for programs) is separated from data storage (for variables.)

**hexadecimal**  Base 16. A number system where all digits are 0-9 or the letters A-F and successive digits are larger by a factor of 16. The number 33 in hexadecimal would be written as 0x31.

**high**  1. The upper part of a register or data value, in particular, the most significant byte of a general purpose register on the 8088.

2. See **high-level**.
high-level  A language like Java, C++, or Pascal where a single statement may
correspond to several machine language instructions.

hybrid memory  A kind of memory designed to combine the field rewritability
of RAM with the data persistence of ROM. For examples, see EEPROM or
Flash.

immediate mode  An addressing mode where a bit pattern is interpreted as a
constant operand. In direct mode, the bit pattern 0x0001 would be the
constant 1.

implement  Of an interface, to follow the interface without being a instance of it.

index mode  An addressing mode where a bit pattern is interpreted as an offset
from a memory address stored in a register.

indirect mode  An addressing mode where a bit pattern is interpreted as a a
memory location holding a pointer to the actual operand.. In indirect mode,
the bit pattern 0x0001 would be interpreted as the value stored at the location
referred to by the pattern in memory location 1.

indirect address register  A register, like the Pentium’s BX or the Atmel’s Y
register, tuned to be used in indirect or index mode.

infix  A method of writing expressions where a binary operation comes between
its arguments, as in 3 + 4. Compare postfix or prefix.

initialize  To set to a particular initial value prior to use, or to call a function
that performs this task.

instance method  A method defined by an object-oriented system as a property
of an object, rather than its controlling class.

instance variable  A variable defined by an object-oriented system as a property
of an object, rather than its controlling class.

instruction  A single basic operation that the computer can perform in a single
indivisible step.

instruction queue  An ordered set of instructions either waiting to be loaded or
already loaded and waiting to be executed.

instruction register  The register inside the CPU that holds the current instruc-
tion for dispatch and execution.

instruction set  The set of instructions that a particular computer can do. Each
type of computer (for example, a Pentium II or the JVM) has its own in-
struction set.

integrated circuit  A circuit fabricated as a single silicon chip instead of as many
individual components.

interface  An abstract class that defines common behavior shared by different
objects, but outside of the normal inheritance structure.
interrupt 1. A small piece of code, set up in advance to be executed when a particular event occurs.

2. The notification to the CPU that such an event has happened and that this piece of code should be executed.

interrupt handler A system for dealing with expected events without the overhead of busy waiting.

invoke To execute a method.

I/O controller A component of a typical machine architecture that controls a particular peripheral for input and output. The I/O controller usually accepts and interprets signals from the bus and takes care of the details of operating a particular gadget like a hard drive.

I/O registers Registers used to send signals to an I/O controller, especially on the Atmel AVR.

jasmin An assembler written by Meyer and Downing for the JVM and the primary teaching language of this book.

Java Virtual Machine See JVM.

JIT compilation A technique for speeding up execution of JVM programs by converting each statement to an equivalent native machine code instruction.

Just In Time See JIT compilation

JVM A virtual machine used as the basis for the Java programming language and the primary teaching machine of this book.

label In assembly language, a human-readable marker for a particular line of code, so that that line can be the target of a branch instruction.

latency The amount of time it takes to accomplish something. On a computer with an instruction latency of 1 µsecond, executing an instruction will take at least that much time.

linear congruential generator A common kind of pseudorandom number generator, where successive values of equations of the form newvalue = (a · oldvalue + cl)%m are the values returned from the generator.

link The process of converting a set of bytecode (or machine instructions) stored on disk into an executable format.

little-endian A storage format where the least significant bit is stored as the first and highest numbered bit in a word. In little-endian format, the number 32770 would be stored as binary 01000001.

llasm The assembler used with Microsoft’s .NET Framework.

load The process of getting a set of bytecode (or machine instructions) from disk into memory.
**logical address**  The bit pattern stored in a register and used to access memory, prior to interpretation by any **memory management** routines.

**logical memory**  The address space defined by the set of logical addresses, as distinguished from the physical memory where the **memory manager** stores data.

**logical shift**  A **shift operation** where the newly emptied bit location(s) are filled with the value 0. Compare **arithmetic shift**

**long**  In Java or **jasmin**, a data storage format for 64-bit (two word) integer types.

**low-level**  A language like **jasmin** or other assembly languages where a single statement corresponds to a single **machine language** instruction.

**machine code**  See machine language.

**machine cycle**  A basic time unit of a computer, typically defined as the time to execute a single instruction, or alternatively as one unit of the system clock.

**machine language**  The binary encoding of the basic instructions of a computer program. This is not typically written by humans, but by other programs such as **compilers** or **assemblers**.

**machine state register**  A register describing the overall state of the computer as a set of flags.

**mantissa**  The fractional part of a **floating point** number, to be multiplied by a scale factor consisting of 2 raised to the power of a specific **exponent**.

**math coprocessor**  An auxilliary chip, usually used for **floating-point** calculations, while the **ALU** handles integer calculations.

**memory manager**  A system for controlling a program’s access to physical memory. It typically improves performance, security, and enhances the amount of memory that a program can use.

**memory-mapped I/O**  A method of performing I/O where specific memory locations are automatically read by the I/O controller, instead of using the bus.

**microcontroller**  A small computer, usually part of an **embedded system** instead of a standalone, independently programmable computer.

**microprogramming**  Implementing a computer’s (complex) instruction set as a sequence of smaller, RISC-like instructions.

**Microsoft Intermediate Language**  The language, corresponding to JVM bytecode, underlying Microsoft’s .NET Framework.

**MIMD**  Multiple Instruction Multiple Data parallelism. The ability of a computer to carry out two different instructions on two different pieces of data at the same time.
APPENDIX F. GLOSSARY OF TERMS

MMX instructions A kind of instruction implementing SIMD parallelism on later models of the 80x86 family of chips.

mnemonic A human-readable statement written as part of an assembly language program that corresponds to a specific operation or opcode.

mode See addressing mode.

modulus The formal mathematical definition of “remainder.”

monitor A subsystem of the JVM used to ensure that only one method/thread has access to a piece of data at once.

Monte Carlo simulation A technique for exploring a large space of possible answers by repeated use of random numbers.

most significant The digit or byte corresponding to the highest power of the base. For example in the number 361,402, the number 3 is the most significant digit.

motherboard The board of a computer to which the CPU and the most crucial other components are attached.

non-volatile memory A kind of memory where the stored data is still present even after power is lost to the system.

non-Volatile RAM A kind of hybrid memory combining the field programmability of RAM with the data persistance of ROM.

n-type A type of semiconductor that has been doped with an electron-donating substance, making these electrons available for passing current.

null A designated address referring to nothing in particular.

nybble A collection of 4 bits, referring to a single hexadecimal digit. Used as a unit of memory, of representation size, or of register capacity. Rare.

object-oriented programming A style of programming, popularized by language like Smalltalk, C++, and Java, where programs are composed of interactive collections of classes and objects, and communication is performed by invoking methods on specific objects.

octal Base 8. Rarely used today.

OF The Overflow Flag, set when the most recent operation on signed numbers generated an answer too large for the register.

offset The distance in memory between two locations. Usually used with regard to either a base register (as in the 8088 memory segmentation) or with regard to the amount by which a branch instruction changes the program counter.

opcode The byte(s) corresponding to a particular operation in machine code. Compare mnemonic
operands  The arguments given with a particular operation; for example, the ADD instruction usually takes two operands. On the JVM, however, the iadd instruction takes no operands, because both arguments are already available on the stack.

operating system  A program-control program used to control the availability of the machine to user-level programs, and to launch and recover from them at appropriate time. Common examples of operating systems include Windows, Linux, MacOS, and OS X.

operator  The symbol or code indicating which particular mathematical or logical function should be performed. For example, the operator ‘+’ usually indicates addition.

OR  A boolean function that returns True if and only if all arguments are True, and otherwise returns false. An OR gate is a hardware circuit that implements an OR function on the input electrical signals.

overclocking  Attempting to run a computer chip with a clock running faster than the chip’s rated capacity.

overflow  Broadly, when an arithmetic operation results in a value too large to be stored in the destination. For example, multiplying two 8-bit numbers is likely to produce up to a 16-bit result, which would overflow an 8-bit destination register.

page  A block of memory used in the memory management system.

page table  A table used by the memory manager to determine which logical addresses correspond to which physical addresses.

paging  Dividing memory into “pages.” Alternatively, the ability of a computer to move pages from main memory to long-term storage and back in an effort to expand the memory available to a program and to improve performance.

parallel  Of an electrical circuit, two components are in parallel if there is a separate path for current that passes through each component as an individual.

parallelism  Of a computer, being able to perform multiple operations at the same time. See also MIMD and SIMD.

peripheral  A component of a computer used to read, write, display, or store data, or more generally to interact with the outside world.

pipelining  Breaking a process (typically a machine instruction execution into several phases like an assembly line. For example, a computer might be executing an instruction while already fetching the next one. A typical pipelined architecture can actually be executing several different instructions at once.

platters  An individual storage surface in a hard drive.

polling  Testing to see whether or not an event has happened. See busy waits.
port-mapped I/O  A method of performing I/O where communication with the I/O controller happens through specific ports attached to the main bus.

postfix  A method of writing expressions where a binary operation comes between its arguments, as in 34+. Compare infix or prefix.

prefetch  Fetching an instruction before the previous instruction has been executed; a crude form of pipelining.

prefix  A method of writing expressions where a binary operation comes between its arguments, as in +34. Compare infix or postfix.

primordial class loader  The main class loader responsible for loading and linking all classes in the JVM.

program counter  A register holding the memory location of the instruction currently being executed; changing the value of this register will result in loading the next instruction from a different location. This is how a branch instruction works.

programmable ROM  Field-programmable but not erasable ROM. Unlike conventional ROM chips, they can be programmed in small quantities without needing to set up an entire fabrication line.

programming models  A defined view of the architecture and capacity of a computer that may be limited in some way for security reasons

protected mode  A programming model where the capacity of the programs are typically limited by memory management and security issues; useful for user-level programs on a multiprocessing system.

pseudorandom  An algorithmically generated approximation of randomness.

radix point  A generalization of the idea of “decimal point” to bases other than base 10.

RAM  Memory that can be both read from and written to in arbitrary locations. RAM is typically volatile in that if power is lost, the data stored in memory will also be lost.

Random Access Memory  See RAM.

Read-Only Memory  See ROM.

real mode  A programming model where the program has access to the entire capability of the machine, bypassing security and memory management. Useful primarily for operating systems and other supervisor-level programs.

record  A collection of named fields but without methods.

register  A memory location within the CPU used to store the data or instructions currently being operated upon.
**register mode**  An addressing mode where a bit pattern is interpreted as a specific register. In register mode, the bit pattern 0x0001 would be interpreted as the first register.

**return**  The process (and typically the instruction used) to transfer control back to the calling environment at the termination of a subroutine.

**RISC**  Reduced Instruction Set Computing. A computer design philosophy using a few short general-purpose instructions. Compare CISC.

**ROM**  Memory that can be both read from, but that cannot be written to ROM is typically non-volatile in that if power is lost, the data stored in memory will persist.

**roundoff error**  Error that happens when a floating point representation is unable to represent the exact value, usually because the defined mantissa is too short. The desired value will be “rounded off” to the closest available representable quantity.

**seed**  A value used to begin a sequence of generation of pseudorandom numbers.

**segment:offset**  An alternative way of representing up to 20-bit logical addresses within the 16-bit registers of an Intel 8088 (or later models).

**segment register**  A register in the 80x86 family used to define blocks of memory for purposes such as code, stack, and data storage and to extend the available address space.

**segment**  Broadly speaking, a contiguous section of memory. More specifically, a section of memory referenced by one of the segment registers of the 80x86 family.

**semiconductor**  A kind of electrical material, midway between a conductor and an insulator, that can be used to produce diodes, transistors, and integrated circuits.

**Sequential Access Memory**  Memory that cannot be accessed in arbitrary order, but that must be accessed in a pre-defined sequence, like a tape-recording.

**series**  Of an electrical circuit, two components are in series if there is only a single path for current that passes through both components.

**SF**  The Sign Flag, set when the most recent operation generated a negative result.

**shift**  A kind of operation where bits are moved to adjacent locations (left or right) within a register.

**sign bit**  In a signed numeric representation, a bit used to indicate whether the value is negative or non-negative. Typically, a sign bit value of 1 is used to represent a negative number. This is true for both integer and floating point representations.
signed  A quantity that can be both positive and negative, as opposed to unsigned quantities that can only be positive.

SIMD  Single Instruction Multiple Data parallelism. The ability of a computer to carry out the same instructions on two different pieces of data at the same time, for example, to zero out several elements in memory simultaneously.

source  Where data comes from. For example, in the instruction iload_3, the destination is local variable #3.

source index  A register in the 80x86 family that controls the source of string primitive operations.

crc  An abbreviation for source.

S–R flip-flop  A kind of circuit used to store a single bit value as an voltage across a self-reinforcing set of transistors.

stack  A data structure where elements can only be inserted (pushed) and removed (popped) at one end. Stacks are useful in machine architectures as ways of creating and destroying new locations for short-term storage.

stack frame  A stack-based internal data structure, used to store the local environment of a currently executing program or subroutine.

state machine  A computer program where the computer simply changes from one named “state” to another upon receipt of a specific input or event.

static  See class field, class method.

string primitive  An operation on the 80x86 family that moves, copies, or otherwise manipulates byte arrays such as strings.

structure  See record.

subroutine  An encapsulated block of code, to be called (possibly) from several different locations over the course of a program, after which control is passed back to the point from which it was called.

superscalar  A method of achieving MIMD parallelism by duplicating pipelines or pipeline stages to enhance performance.

supervisor  A privileged programming model where the computers have the capacity to control the system in a way normally prohibited by the security architecture, usually used by the operating systems. See also real mode.

this  In Java, the current object whose instance method is being invoked. In jasmin, an object reference to this is always passed as local variable #0.

throughput  The number of operations that can be accomplished per time unit. This may be different from the latency on a multiprocessor that may (for example) be able to complete several operations with identical latency at the same time, getting effectively several times the expected throughput.
**Throwable**  In the JVM, an Object that can be thrown by the `athrow` instruction; an exception or error.

**timer**  A circuit that counts pulses of a clock in order to determine the amount of lapsed time.

**time sharing**  A form of multiprocessing where the CPU runs one program at a time in small blocks of time, giving the illusion of running multiple programs at once.

**two's-complement notation**  A method of storing `signed` integers such that addition is an identical operation with both positive and negative numbers. In two's complement notation, the representation of -1 is a vector of bits whose values are all 1.

**typed**  Of a computation, representation, or operation, when the type of data processed affects the legitimacy or validity of the results. For example, `istore_0` is a typed operation as it will only store integers. The `dup` operation, by contrast, is untyped as it will duplicate any stack element.

**U**  One of the two five-stage pipelines on the Intel Pentium.

**unary**  A mathematical operator such as negation or cosine that takes exactly one operand.

**unconditional**  Always, as in an an unconditional `branch` that is always taken.

**unsigned**  A quantity that can only be positive and negative, as opposed to `signed` quantities that can be positive or negative, but typically cannot express as large a range of positive values.

**update mode**  An addressing mode where the value of a register is updated after access, for example, by incrementing to the next array location.

**UTF-16**  An alternate character encoding to ASCII where each character is stored as a 16-bit quantity. This allows up to 65,536 different characters in the character set, enough to include a large number of non-English or non-Latin characters.

**V**  One of the two five-stage pipelines on the Intel Pentium.

**verifier**  The phase of loading where the class file is `verified`, or the program that performs such verification.

**verify**  On the JVM, the process of validating that a method or class can be successfully run without security issues. For example, attempting to store a value as an integer that had previously loaded as a floating point number will cause an error, but this error can be caught when the class file is `verified`.

**virtual address**  See logical address.

**virtual address space**  See logical address space.
virtual memory  The capacity of a computer to interpret logical addresses and to convert them to physical addresses. Also, the ability of a computer to access “logical” addresses that are not physically present in main memory, by storing parts of the logical address space on disk and loading into memory as necessary.

virtual segment identifier  An bit pattern used to expand a logical address into a much address space for use by the memory manager.

watchdog timer  A timer that, when triggered, resets the computer or checks to confirm that the machine has not gone into an infinite loop or otherwise unresponsive state.

word  The basic unit of data processing in a machine, formally the size of the general purpose registers. As of this writing (2004), 32-bit words are typical of most commercially available computers.

ZF  The Carry Flag, set when the result of the most recent operation is a zero