# COMPUTER ORGANIZATION

## Basic Processor Structure

James Gil de Lamadrid



CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

# COMPUTER ORGANIZATION

## Basic Processor Structure

# COMPUTER ORGANIZATION

## Basic Processor Structure

James Gil de Lamadrid

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742



Printed on acid-free paper
Version Date: 20180131

International Standard Book Number-13: 978-0-8153-6246-3 (Hardback)
International Standard Book Number-13: 978-1-4987-9951-5 (Paperback)



**Visit the Taylor & Francis Web site at**
**http://www.taylorandfrancis.com**

**and the CRC Press Web site at**
**http://www.crcpress.com**

*To my father and mother, who infused me with the conviction that there are very few human activities more important than pursuing knowledge.*

# Contents

# Introduction and Remarks

## INTRODUCTION

This book tries to answer the question, "How is the processor structured?" This question leads to a second question: "How does the processor function in a general-purpose computer?"

The answers to these questions can be quite complex and quite involved, but the answers to these questions do not need to be all that complex. The complexity of the answers to these questions should be appropriate for the audience to which the responses are addressed. If you are addressing a processor designer, of course the answers must be very detailed. However, if you are addressing a layman, the answers would be fairly simple and abstract.

This book is intended to be used in a computer science curriculum. So, our audience is assumed to be computer science undergraduates, or lower-level graduate students. As such, the answers we supply to our motivating questions do not have to be nearly as detailed as the answers we would give to a potential processor designer, nor should they be as simple as the answers given to a layman.

The pedagogical question that drives the content of this book is, "What is the simplest explanation of a processor you can give to a student of computer science; an explanation that will not overpower the student with information, during the learning process, and yet is sufficiently complete so as to serve the student in their career?" In this book, we believe that we have found the sweet spot between too much, and too little information.

Our choice of topics and depth of coverage in this book are based on a couple of decades of teaching experience. Having taught computer organization, and architecture, over the years we have settled on a set of topics that, we believe, is the essence of the field. The set of topics is small enough so that all of the topics can be taught in a single semester course.

## WHY STUDY COMPUTER ORGANIZATION AND ARCHITECTURE?

There are two topics that are often taught as part of the computer science curriculum: computer organization and computer architecture. Collectively, we will refer to these two topics as computer systems. Computer systems is a little out of line with most other computer science topics, which are mostly concerned with software.

Usually, the job of describing computer hardware is split into two levels. Computer organization concerns itself with low-level circuity, or, as we might say, *how* the computer computes. Computer architecture concerns itself with higher-level devices, and how these are manipulated by software, or, as we might say, *what* the computer is capable of computing.

This book concerns itself with both computer organization and computer architecture, with an emphasis on computer organization. We cover some computer architecture with our material on machine language programming.

Often enough, we are asked by students, and sometimes colleagues, what relevance computer system courses have to computer science. This question usually comes from a view that computer science is the study of software only, and that learning how the computer works on a hardware level is irrelevant. But, there are at least two responses to this question that establish the importance of computer systems in the computer science curriculum.

The first response takes issue with the view of computer science as concerned with software only. There are several fields that are embraced by the computer science field that work on the frontier between hardware and software, and in which the worker must have an understanding of hardware function, as well as software skills: robotics, embedded systems, and even operating systems.

The second response takes issue with the perception that if you work only with software, hardware knowledge is irrelevant. Imagine that you are hiring a computer scientist for software work. If faced with two candidates, the question is which would you view as preferable: a candidate with a solid understanding of how to program a computer, as well as an in-depth understanding of how the computer behaves, or a candidate with only software experience? You would probably view the more in-depth knowledge as an asset. Knowing how a computer functions does help, immensely, in software development. Many of what would seem unexplainable software behaviors often become clear with hardware knowledge, giving the worker an advantage in the testing and ver-

ification of software. Also, notice that knowing how hardware functions can often also lead to smarter design decisions when building software systems.

## USING THE BOOK

This book has been designed as a teaching aid, to be used in a one-semester course on computer systems. It covers most of the essential topics in computer organization, and few topics in computer architecture.

The course with which this book would be used, would be aimed at students at an undergraduate level. The course for which we use this book is taught for graduate students. The course in which we teach this curriculum is used to insure that incoming graduate students all have exposure to a common core, of which this course is one component.

To use the book for a single-semester course, it is possible to cover almost all material in the book. You can start with Chapter 1, which gives the student perspective on the interaction between hardware and software. This chapter takes the reader through the process of getting a program to run. It starts with creating the software, compiling and assembling the software, loading it into memory, and running it. It then briefly explains how executing instructions results in operations in digital circuitry. After this overview, we start detailing the processes described.

Chapter 2 presents the mathematical basics required in the rest of the book. In particular, we present material on Boolean algebra and the binary number system.

In Chapter 3, the basics of digital circuitry are discussed. We are taken through the basics of combinational circuits. Then, we examine sequential circuits. This is followed by Chapters 4 and 5. In Chapter 4 we talk about the bus communication architecture, used in many computer systems. A brief discussion on interfacing with peripheral devices is included. Chapter 5 talks about the RTL level of circuity. In this material we describe the building blocks of the processor.

Chapter 6 is a discussion of machine language, that finishes off the preparation for processor design. We talk about the different processor architectures, in terms of the number of operands in the machine instructions, from 0-operand stack machines to 3-operand register machines.

It is possible to cover Chapters 1 through 4, and most of Chapter 5, by half-way through the semester. This gives the student a good understanding of the preliminary information required to understand processor design.

In the second half of the semester, you could cover Chapters 7 through 9. In Chapter 7 we design a processor. The processor is designed as an algorithmic circuit, starting with the data path, and finishing with the control unit. A relatively simple register-implicit machine is designed; simple enough so that details do not lead to confusion, yet with enough complexity so that the reader will see it as useful.

In Chapter 8, we talk about ALSU design and computer arithmetic. The usual operations of addition, subtraction, multiplication, and division are covered, for both integer types and floating-point.

Chapter 9 discusses micro-controlled processors. We redesign the control unit for the same processor covered in Chapter 7.

Chapter 9 concludes what we would think of as fundamental computer systems information. In Chapter 10, we briefly consider several more advanced topics. When we teach this material, this information is presented at the very end of the semester, as time allows. Usually we end up giving a short overview of these topics only.

## OUTSIDE RESOURCES

There are a couple of resources available to the student and instructor for enhancing the material presented here.

- The solutions manual. This is available to the instructor, through the publisher. It includes answers to all exercises in the book.
- The BRIM Simulator. The machine language BRIM (Basic Register Implicit Machine) is used throughout the book, as the interface to the main architecture presented. A simulator for the machine level of this machine is available. It includes an assembler and an emulator. An executable build exists for the Linux and Windows platforms. It can be run on the Macintosh, by creating a build from the source Python scripts.

There are other tools that can be used with the book, when exploring some of the topics. In particular, when working with digital circuitry, it is often useful to use a circuit simulator. With the ready availability of circuit simulators,

we invite you to choose your favorite one, and use it in your instruction. Our favorite is Logisim.

Our approach is to do digital circuitry in simulation, but you could opt for something more substantial, like a hardware lab. Or, as we do in our undergraduate course, you could run an FPGA lab. We include a small section on Verilog programming, which can enable the reader to program for a variety of RTL simulators, such as Modelsim, or program for synthesizers.

If instructors wish to introduce a group project into the course, they could ask the students to build the BRIM machine as a class project. With a whole class working on the processor, this project could be completed in Verilog or VHDL, in a semester's time.

CHAPTER 1

# Overview

## CONTENTS



Often students begin their training in the field of computation by learning to program a computer. In this experience the student learns how to construct a program in some *high-level language*. The student learns how the syntax of the language, and the semantics, work together to form a description of a

computation. The student also learns that this description can be run on a computer to perform the computation. Many of us then start wondering what kind of magical machine is capable of performing our computation, given only a simple description.

The truth of the matter is that computers are indeed magical, in the sense that they can do amazing things. However, they are less than magical, although still amazing, when we examine how they work. Two areas that examine the operation of the computer are called *computer organization* and *computer architecture*. The distinction between computer organization and computer architecture is probably best described in terms of two other concepts: *implementation* and *interface*. Computer organization is the study of the implementation of a computer. That is to say, the hardware and circuitry out of which the computing system is built. Computer architecture studies the structure presented to a program, that can be used to perform a computation.

Computers are made up of many separate devices. To understand how a computer performs a computation, probably the most interesting device to examine is the *central processing unit* (CPU), which is often simply called the *processor*. This is the device that actually executes a program, and is the focus of this book. In this section we directly examine the question of how a program performs a computation, by giving an overview of the structure of the computer system. In this discussion the structure is presented as a set of layers, one built on top of another. We begin our discussion at the top level, the high-level language program which has been written, and work our way down to the bottom level, which consists of digital circuitry.

## 1.1 HIGH-LEVEL, ASSEMBLY, AND MACHINE LANGUAGES

So, a user writes a program in a high-level language, with the intent of running the program on a computer. Let us analyze that statement. We begin by discussing what is meant by a *high-level language*.

### 1.1.1 High-Level Languages

Programming languages are used to describe a computation that is to be executed on a computer. Languages are classified by level. The idea is that low-level languages are closer to the hardware of the computer, and high-level languages are further from the hardware, and more abstract. That is to say, a low-level description of a computation would describe the computation in terms of the various hardware devices that constitute the computer system. A high-level description might describe the computation as the manipulation of some abstract data structure, like an array, which is a data structure that would normally not be implemented directly in hardware.

There are many high-level languages, some at a higher level than others. They support several different computational models. As an example, the language C++ supports a computational model usually called the object-oriented model. *Haskell* supports a paradigm called the functional model of computation. The problem with all high-level languages is that they support computational models which are not implemented directly by the computer hardware. Because of this, any program written in a high-level language, like C++, cannot be executed directly by a computer. In order for a program to be executable it must be written in a language that has a format that can be interpreted by the computer, and it must describe the desired computation in terms of the hardware available on the computer. This is what low-level languages do, and in particular, this is what is done by *machine language*, the lowest-level programming language.

### 1.1.2 Machine Language

Machine language describes a computation in terms of hardware, and, as such, is hardware dependent. That is to say that every processor model, for a general-purpose computer, has a different machine language, and these machine languages are non-portable. Another interesting fact about computers, for those used to interacting with them in a high-level language, is that computers only work with numbers. This is interesting because a program in C++ is written using words, like *if*, *void*, and *cout*, rather than just numbers. In fact, the user of C++ might well wonder what a numeric language would look like.

To illustrate what machine language looks like, let us examine an example. Consider the following C++ code fragment.

```
x = 5 + y * 3;
```
(1.1)

How might this appear in a numeric machine language? Consider the following machine code segment.

```
1, 1, 2, 3
14, 1, 1, 5
```
(1.2)

This segment might do the same computation as the C++ segment. To understand it, we must know something about the format of a machine language program.

The machine language we are using is fictitious, but similar to several actual machine languages. A machine language program is a sequence of instructions, written one per line. Each instruction, in this language, consists of four number, or *fields*. For our fictitious example, the fields have the following names.

*op-code, destination, source, constant*

Each instruction performs an operation on operands, and produces a result. Operations must be specified numerically in machine language, and so each possible operation the processor can perform is assigned a number to represent it, called its *op-code*. In the above example, the op-codes used are 1 for multiply, and 14 for add.

The op-code is followed by three operands. The operands in the C++ code are the variables $x$ and $y$. Variables are a high-level concept. The corresponding concept at the hardware level is the register, which is a device that can store a number. A processor typically has a set of general-purpose registers available to the programmer, and they are often numbered. Each variable in the C++ program might be assigned a register, which is used in the machine language version of the code. In the above example, Register 1 (R1) is used to represent `x`, and R2 is used to represent `y`.

The instructions in Example 1.2 each specify two operands: the source operand, which is always a register, and a constant operand, which is always a constant value. The operation specified by the op-code is performed on these two operands, and the result is left in the destination register. If we were to write the above machine language fragment in a more human-readable form, we might produce something like the following.

```
R1 = R2 * 3
R1 = R1 + 5
```
(1.3)

The first instruction multiplies R2 by 3, and stores the result in R1. The second instruction then adds 5 to R1, storing the result in R1.

We now can see the full scale of the problem of writing a program in a high-level language. In order to execute a C++ program, the C++ code must be rewritten, or translated, into this style of machine code.

### 1.1.3 Assembly Language

When the modern computer was first being developed, in the middle of the twentieth century, the only language available for programming a computer was machine language. As can be imagined, this was a difficult language for human programmers to use. Just reading through a program was difficult, requiring a mental translation between numbers and operations and operands. Mentally, this required the programmer to do what we did in the previous section, where we translated from numeric machine code to a notation that resembles assignment statements, which describes the semantics of the operations in symbolic form. These symbolic machine instructions are, essentially, assembly code. That is to say, *assembly language* is a symbolic form of machine code.

Assembly languages, like the one we used in the previous section, were developed to aid programmers of early computers. They are still useful today for several purposes, including as the target language of a compiler. The assembly example, Example 1.3, is, however, a little unusual in its notation. A more standard notation for the same code fragment might appear more as follows.

```
mult R1, R2, #3
add R1, R1, #5
```

This notation follows the format of the machine language instruction, starting with a symbolic op-code, a symbolic destination register, a symbolic source register, and ending with the constant.

## 1.2 COMPILERS AND ASSEMBLY LANGUAGE

The problem of translating from a high-level source language to a low-level machine language is a complex one. This is due to the fact that high-level languages typically have non-trivial syntax, and their semantics are much more involved than the semantics of machine language. As a consequence translation from a high-level language to machine language is typically done in stages, rather than all at once. First, the source code is translated to a middle-level language, which is often assembly language. In further stages the assembly language is translated into lower-level languages, eventually producing machine language as the result of the process. The idea is that splitting the translation process into stages spreads the complexity of the translation process over the stages, making each translation stage simpler.

### 1.2.1 Assembly Language Translation

In the first translation stage, a program called a *compiler* takes as input a source program in a high-level language like C++, and translates it into a program in a target language, which is at a lower level. The target language is typically assembly language.

We now have a reasonable idea of the form of both the source and target languages of a compiler. Although it is beyond the scope of this book to examine the translation done by the compiler in detail, it is worthwhile saying a little on the subject. A compiler inputs a source program and writes a new program in target code. The target program is the equivalent of the source program, meaning that the two programs do exactly the same thing. Given the same input, the target program will produce the same output as the source program.

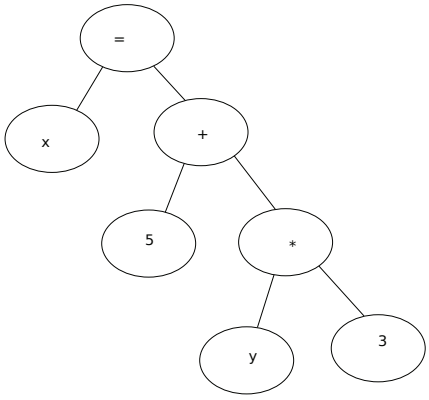


FIGURE 1.1 AST for Example 1.1.

### 1.2.2 The Translation Process

The translation of the source program is the result of a sequence of passes made through the source code. Two of the more important passes are *parsing* and *code generation*. Parsing is the task of deciphering the meaning of the statements of a program. This is done by building a structure out of the statements of the program, that is more machine readable than just simple text. As an example, a compiler might build what is called an *abstract syntax tree* (AST). For the example C++ code given in the previous section, Example 1.1, the AST might appear as in Figure 1.1. In this tree, intermediate nodes are operators, and leaf nodes are operands. The top intermediate node uses the assignment operator to assign a value computed by the right child to the variable $x$. The computation is calculated using an addition operator, and a multiplication operator.

Once the AST has been constructed, target code can be generated from the tree. This is done by traversing the AST, and at each intermediate node, outputting instructions that perform the specified operation. This code generation process, although complex, in its basics, is just printing out the contents of a tree, much as we might learn how to do in a course on data structures. The output of the code generator, however, follows a particular format; namely that the content is printed in the form of assembly instructions. Also, variable names must be replaced by their assigned registers, during the traversal.

## 1.3 THE ASSEMBLER AND OBJECT CODE

The compiler translates a high-level program into an assembly language program. The program must still be translated from assembly language to machine code. This is the job of another translation program called the assembler. However, technically, the assembler does not translate all of the way down to machine language, but rather into a format called *object code*. Object code can be thought of as incomplete machine code. A good analogy is a form, or boilerplate. A boilerplate is a complete document, except that it contains blanks that must be filled in with specific information. In the same way, object code can be thought of as machine code with blanks. The reason why such code would be produced, as opposed to complete machine code, is what we explain now.

### 1.3.1 External References

Larger programs in high level languages are typically split into modules. These modules are submitted to the compiler as separate files. So, for instance, you could have a C++ program composed of two modules: the module *Q*, and the module *Driver*. Splitting the program into modules, in this fashion, is an organizational aid for the programmer. Splitting the code also allows the programmer to work on one part of the code without touching other parts, and running the risk of inadvertently damaging them.

The compiler compiles only one module at a time. It produces assembly language versions of each module. These files are then submitted to the assembler, one at a time. The important point to remember in the following discussion is that the assembler sees only one module at a time, and while analyzing and translating, the module has no information about the other modules.

Modules may contain what are called *external references*. Consider the following lines that might be contained in the module *Q*.

```
extern int x;
x = 5;
```

The module *Q* is changing the value of a variable *x*. But the keyword *extern* declares that the variable is not in the module *Q*, but rather in another module. Continuing with the example, the module *Driver* might contain the following code.

```
int x ;
```

The assembly language versions of the two modules would contain instruc-

tions to mimic the C++ code. That is to say, the module $Q$ would contain the following code.

```
store x, #5
```
(1.4)

This instruction writes the constant 5 to a memory location, designated as $x$.

The module *Driver* would contain a definition of the variable $x$. In assembly language, variables are typically stored in memory. A memory unit is an array of storage devices, called *words*. When referring to a particular word in the memory unit, in assembly language you would give that word a name, much as in the following definition that you might find in the module *Driver*.

```
x:   .word
```

This declaration allocates a word in memory for the variable, and associates the symbol $x$ with it. This is equivalent to a variable declaration in C++.

In machine code, variables do not exist. When a program is executing, it is located in the computer's memory. As mentioned, memory consists of an array of *words*. In machine language, words are identified by their index. The indexes of the memory locations are called *addresses*. So, we might refer to the word at Address, or Location, 50, often written as M[50].

The instructions, as well as the variables of the program are stored in different words of the memory unit. In machine language, the variable $x$ of our example, is simply a word in memory that has been allocated for the storage of data. That word is known only by its address. As a consequence, the *store* assembly instruction, in Example 1.4, might be translated into the following machine code.

```
19, 50, 5
```
(1.5)

The assumption is that the *store* instruction has op-code 19, and that the variable $x$ has been allocated the location M[50].

Here now is the problem with external references. In this example, the assembler attempts to write a machine instruction corresponding to the *store* assembly instruction. It knows what op-code to write, and it sees the constant value, 5, both from the assembly instruction. However it does not know the address of the variable $x$. The address of $x$ is calculated when the module *Driver* is assembled, and since the assembler has no information about the module *Driver* when assembling the module $Q$, the assembler does not know the address of the variable $x$. Because of this, the assembler ends up writing out an instruction that looks something like the following.

```
19, x?, 5
```

This is an incomplete instruction. It contains a blank, indicated by the question mark, to be filled in later by some program that knows the location of the variable $x$. The blank contains a note indicating that the value to be filled in is the location of $x$.

These types of instructions, being generated by the assembler, are in fact the object code. As can be seen, these instructions can be thought of as machine instructions, but with blanks left in them. The reason that these blanks are required is mostly due to the inevitable presence of external references when programs are split into modules.

### 1.3.2 Compiler versus Assembler

As a final topic in this section, it is instructive to compare the assembler with the compiler. Both of these programs are translators. This is their similarity. There is, however, a significant difference between the two programs, in terms of the process they use to do the translation. The compiler's job is complex. The syntactic analysis, which ends up building the AST, is complex, and the code generation is also complex, often requiring that several assembly instructions be written for each node in the AST. The translation process for the assembler, however, is not all that complex.

Syntactic analysis is fairly simple for the assembler. An assembly instruction is, essentially, just a small sequence of symbols, as opposed to the highly structured form of high-level statements that the compiler deals with. And, once an assembly instruction has been decomposed into separate symbols, the code generation process is equally simple.

The process of writing out a machine instruction is referred to as *assembling* the instruction. This term captures the idea that the machine instruction is being built by connecting sub-parts of the instruction. The sub-parts of the instruction are derived from the symbols of the assembly instruction. Each symbol in the assembly instruction corresponds to a numeric value. For the example we give in Examples 1.4, and 1.5, the symbol *store* corresponds to the numeric value 19. The assembler could easily maintain the bindings between symbols and their numeric values in a table. To translate an assembly instruction into a machine instruction, then, simply requires that the assembler looks up the symbols of the assembly code in the table, procures their numeric values, and assembles the numeric values into machine instruction, following the format rules for the machine instruction.

Since the object code is incomplete, we still do not have an executable program. In order to fill in the blanks in the object code, all of the modules

of the program must be examined simultaneously. This, then requires another step in our translation process.

## 1.4 THE LINKER AND EXECUTABLE CODE

The linker is a program that inputs a set of object code files representing a program, and outputs what is called *executable code*. Executable code, usually, is complete machine code. The executable code is written to a mass storage unit such as disk or flash memory, as a file. In order to do this, the linker must resolve all unresolved external references. Including this job, the linker has three primary jobs.

- Resolving external references.
- Library searches.
- Relocation of module code.

We start our discussion of the linker with the resolution of external references.

### 1.4.1 Resolving External References

The input to the linker consists of all of the object files for the whole program. For our example, the linker would examine both the object file for the module *Q*, as well as the object file for the module *Driver*. The linker would notice that the module *Q* contains a blank to be filled in with the address of the variable *x*. It would then examine the module *Driver*, and determine the address of *x*. The blank in the instruction in Module *Q* would then be replaced with the correct address. This is a basic description of the procedure used by the linker to perform its most important job: resolution of external references.

### 1.4.2 Searching Libraries

The linker's second job is performing library searches. Suppose that the module *Driver* contains the following C++ code fragment.

```
z = sqrt(y);
```
(1.6)

What this code does is to call a function *sqrt*, pass it an argument *y*, and place the return value into the variable *z*.

To implement the function call of Example 1.6 in machine language, a method for passing arguments to a function, and passing a return value back from the function, must be agreed upon. Every processor supports some such method, called the *calling conventions* of the processor. For our simple processor, we have assumed that arguments are passed to the function by loading

them into special-purpose *argument registers*, whose names begin with the letter $A$, as opposed to the letter $R$, to distinguish them from the general-purpose registers of the processor. To access the arguments, the function simply accesses the argument registers. When the function is ready to return a value, it loads its return value into another special-purpose register, called the *return value* (RV) register. This register is then accessed by the calling program, to receive the return value. The registers associated with the calling conventions are called *special-purpose registers*, because the programmer would only use them for function calls, as opposed to general-purpose registers whose use is unrestricted.

When translated into assembly language, the sequence from Example 1.6 might resemble the following.

```
load A0, y
call sqrt                    (1.7)
store z, RV
```

In Example 1.7, the assembly fragment first fetches the operand $y$ from memory, and loads it into the argument register A0. Then a *call* instruction is executed, which causes the processor to jump to the address *sqrt*. When the *sqrt* function executes a *return* instruction, the processor jumps back to the *store* instruction following the call. At this point the return value is transferred from the RV register into the memory location at address $z$.

The call to the function *sqrt* constitutes an external reference. Clearly *sqrt* is not defined in the module *Driver*. But, it is also not defined in the module *Q*. Most C++ programmers know that the function *sqrt* is, in fact, defined in the C++ library, which is a collection of modules, in object code format. *Sqrt* is defined in the module *math*. In order to allow a program to use code contained in library modules, we must alter our procedure for external reference resolution to include library search. That is to say, when the linker encounters an unresolved reference, it first looks for resolution in the actual modules of the program. If the linker does not find a definition for the reference, then it searches all of the modules in all of the libraries it can find. If the linker finds a definition for the reference in a library module, that module is added to the other modules of the program. If the reference is still unresolved after this rather exhaustive search, a *linker error* occurs, and linking is aborted.

### 1.4.3 Relocation

We now turn our attention to the last job of the linker: relocation of modules. In the beginning of our discussion of the linker, we said that when resolving the external reference to the variable $x$, the address of the variable is calculated

by examining the module *Driver*, which contains the variables definition. It was, however, not explain clearly how addresses are calculated.

Addresses inside a module are relatively easy to calculate. The module has a base, or beginning, address. The objects in the module are either instructions or data allocation declarations. The number of words taken by each instruction is known, and the data declaration explicitly provides the number of words being allocated. It is easy to run through a module from top to bottom, and track the number of words written out at any particular point in the module code. In this way the address of any object can be calculated by using the current word count as an offset, which is added to the base address. This job is readily performed by the assembler.

The situation is complicated by the fact that our program is not a single module. However, when the linker finishes its work, it must produce a single monolithic program, occupying a solid, contiguous block of memory. In other words, it must combine the different modules into one block. To do this, the linker must order the modules in the memory block. For our example, the linker might decide to place the module *Driver* first in the machine language program, and then follow it with the module *Q*.

In term of addressing, the module *Driver* would be assigned the base address 0. Let us assume that *Driver* contains 3,000 words of machine code, and that *Q* contains 2,000 words. Then the base address of *Q* would be 3,000, and it would end at address 4,999, which is one subtracted from the sum of 3,000 and 2,000. The upshot of this calculation is that, if each module has been assembled, assuming a base address of 0, all addresses in the module *Q* must be adjusted by adding 3,000 to them. This process of adjusting addresses, after deciding on module order, is the process of module relocation.

## 1.5 THE LOADER

As discussed, the linker produces executable code, that is saved to mass storage. Although the code produced is executable, it is not yet running. The program that starts the executable code running is the loader. What does it take to start a program running? Firstly, the program must be loaded into memory from mass storage. This process is often termed *program relocation.* Notice that the linker and the loader both do relocation. The difference is that the linker is relocating modules, or pieces of a program. The loader, on the other hand, relocates whole programs in memory.

### 1.5.1 Processes and Workspaces

The modern computer is typically a *multi-process* system. This means that the operating system is often running several processes, or programs, simultaneously. Technically, this is impossible, since the processor can only do one thing at a time. Technically, the processor is only making it appear that it is running several programs simultaneously. In reality it is sharing time between the different running processes, working one process for a small amount of time, then switching to another process for a small amount of time, then on to another process, and continuing this procedure, executing small pieces of code from each program. Because the amount of time spent on each program, called the *quantum*, is so small, and the processor returns to the same process in such a short time, it appears to a human observing a particular process that the processor is working on that process without interruption. To the human observer, it appears that the processor is working on several processes, each one uninterrupted, although this is not actually true. Perhaps a more accurate statement would be that, in the modern processor, several processes are active at the same time.

Each process that is active is assigned a part of memory to work with, often referred to as the process's *workspace*. The workspace is allocated to the process by the operating system when the process is started.

Our discussion of the process workspace is leading into an explanation of why loading a program into memory requires relocation. Let us reconsider our continuing example. We wish to run the *Driver* program. It has been compiled, assembled, and linked, and is now in the form of an executable file on disk. To run it, the operating system starts a new process, and assigns a block of memory, its workspace, to the process. Now the loader reads the executable file from disk, and writes the code to the workspace. The problem that the loader has, is that the workspace may be located anywhere in memory. This is a problem because when the linker previously wrote out the executable code, it did not know what the base address of the program would be. What it did is the equivalent of writing code that starts at memory location 0. This is almost certainly not going to be the case.

When the loader loads the program into the workspace, if there is any hope of the program running correctly, all addresses in the program must be altered from using a base address of 0 to using the base address of the workspace. There are several ways of performing the alteration. Often the alteration does not require an explicit change to every address in the program. Often the changes can be made simply by initializing a special-purpose base address register to the base address of the workspace. Whatever the method used, the point is that somehow the executable code must be made to execute at the

location where it is loaded into the workspace. This process, which we have already mentioned is called relocation, is performed by the loader.

### 1.5.2 Initializing Registers

So, now the *Driver* program is loaded into the workspace. The next job of the loader is to initialize the processor in preparation for executing the program. Many of the registers in the processor must be initialized. This is truer of the registers that are used by the processor for specific purposes. In particular, one such register in the processor is often called the *program counter* (PC). The job of the PC is to keep track of where in the program the processor is currently executing. The PC always contains the address in memory of the next instruction to be executed. Every time the processor finishes executing an instruction, it fetches the instruction indicated by the PC from memory, and begins executing that instruction. The PC is also updated to point to a new, next instruction, usually by incrementing it.

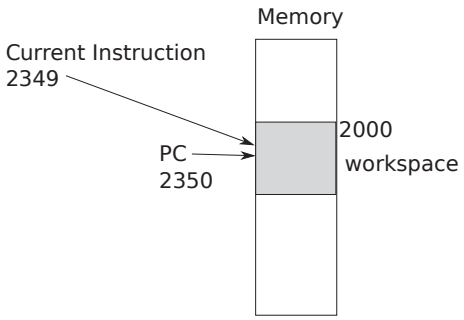


FIGURE 1.2 The PC and memory.

The role of the PC is illustrated in Figure 1.2. The diagram shows the process workspace in memory. The base address of the workspace is 2000. The PC contains the address 2350. The instruction currently being executed would then be at address 2349. When this instruction has been completed, the machine instruction at 2350 would be loaded into the processor to be executed, and the PC would be incremented to 2351.

It should be clear that the PC must be initialized to the base address of the workspace, when the program begins execution. This then is one example of several special-purpose registers which the loader must correctly initialize before the program begins execution. Actually, the initialization of the PC is done as the last register initialization. The reason for this is that if the loader

changes the PC to point to the starting instruction of the *Driver* program, the next instruction executed will be the first instruction in the *Driver* program. In other words, the loader has jumped to the start of our program, and started it up. This is the last action of the loader; after loading the program, and initializing registers, it starts the program running.

## 1.6 SUMMARY OF THE TRANSLATION PROCESS

We now have some sort of understanding of the answer to our original question as to how a C++ program is executed. The process of getting the program to run is actually fairly sophisticated, and involves several steps. These steps are summarized in Figure 1.3.

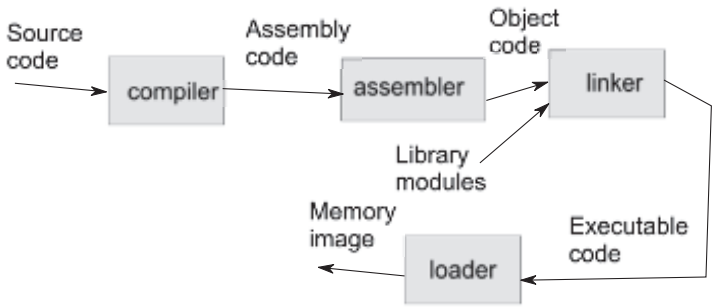


FIGURE 1.3 Workstream for source code translation and execution.

Figure 1.3 shows the compiler first translating the source code program into assembly language. The assembler then translates the program down into object code. The linker links the object modules together, and adds in any required library object modules. The output of the linker is executable code, which is loaded into memory by the loader.

## 1.7 THE PROCESSOR

We have now partially answered the question as to how a C++ program executes on a computer. We have examined the translation process that produces and starts machine code. But if we dig deeper, we discover that this is only part of the answer. We know how to get the program to run, but not how the processor actual executes the instructions. To answer that part of the question, we must examine what sort of machine the processor is, and how it is

designed. This is the point at which we must start looking at the circuitry in the processor. We stop talking about the software used, like the compiler and linker, and start talking about the hardware device from which the processor is built.

When we discussed the software tools used to translate a program, we did so by describing language levels. We use this same level approach when discussing hardware. We have two major levels of abstraction used when specifying processor design.

- The *register transfer level* (RTL), or *behavioral level*.
- The *gate level*, or *structural level*.

The RTL level is the higher level of abstraction, and the gate level is the lower level. As with the software, we will be examining the two levels from the top down, starting with the RTL level.

### 1.7.1 Processor Behavior

To begin our discussion of the RTL level, we examine the functioning of the processor. A processor is a device that fetches instructions stored in memory, and executes them. This is a slight simplification of the functioning of the processor, but not much of one. The processor performs a sequence of steps, over and over, as long as it is supplied with electrical power. This sequence of steps is called the *machine cycle*, or *instruction cycle*. The instruction cycle is a three-step procedure.

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Execute the instruction.

In Step 1 the processor reads the next instruction to be executed, as indicated by the PC register, and brings it into the processor. The processor also increments the PC at this time. In Step 2 the processor splits the instruction into fields. By so doing it discovers what operation is being performed, and on what operands.

In Step 3 the appropriate circuitry in the processor is activated to execute the operation on the operands. The result of the operation is then written back to the destination operand. The cycle is then repeated, to execute the next instruction.

### 1.7.2 Processor Structure

We now have described what the processor does. Let us now look at the structure of the processor. The processor is a machine composed of several devices. For instance, we have already discussed the fact that the processor contains a set of both general-purpose and special-purpose registers. These registers, collectively, form a device called the *register file*. Another device called the *arithmetic-logic unit* (ALU) performers arithmetic and logic machine operations.

All of the devices in the processor must be connected, in order to communicate with each other. How the devices are connected is called the *data-path* of the processor. The different devices must also be told when to perform their function and often which of several functions to perform. The circuit that controls the devices, in this fashion, is called the *control unit*.

#### *1.7.2.1 The Data Path, Registers, and Computational Units*

The best way to explain a data path is with an example. We will be building an example that contains just two registers. The circuit will perform just two operations, as follows.

$$\begin{aligned} R1 &\leftarrow R1 + R2 \\ R2 &\leftarrow 0 \end{aligned} \tag{1.8}$$

In the first operation of Example 1.8, the contents of the registers R1 and R2 are added, and the result is placed into the register R1. In the second operation, the register R0 is set to 0.

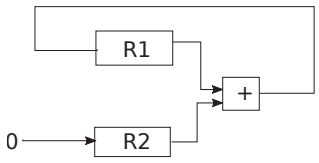


FIGURE 1.4 Example data path for Example 1.8.

The data path of this device is the circuitry that allows these two operations to be performed. This data path would include two major components: registers, and for each register, a *computational unit*. A computational unit computes the new value of a register, resulting from an operation. In our example, we would have one computational unit to compute the new value of R1, as the sum of R1 and R2, and another that computes the new value of R2 when it is cleared. All computational units and registers must be connected