Chapter 4

Contax T3 / Kodak Tri-X

Big Snow

Computers, Programs And Algorithms

Learning Objectives

- *State the purpose and use of a computer*
- *State the primary characteristic that makes the computer a unique device*
- *List and describe the four stages of the program execution cycle*
- *Explain how a computer stores and retrieves programs for execution*
- *State the difference between a computer and a computer system*
- *Define the concept of a program from both the human and computer perspective*
- *State the purpose and use of main, auxiliary, and cache memory*
- *Describe how programs are loaded into main memory and executed by a computer*
- *State the purpose and use of the Microsoft .NET Common Language Runtime (CLR)*
- *List the similarities between a virtual machine and a real computer*
- *Explain the purpose of Microsoft Intermediate Language (MIL)*
- *Define the concept of an algorithm*

Introduction

Computers, programs, and algorithms are three closely related topics that deserve special attention before you start learning about C# proper. Why? Simply put, computers execute programs, and programs implement algorithms. As a programmer, you will live your life in the world of computers, programs, and algorithms.

As you progress through your studies, you will find it extremely helpful to understand what makes a computer a computer, what particular feature makes a computer a truly remarkable device, and how one functions from a programmer's point of view. You will also find it helpful to know how humans view programs, and how human-readable program instructions are translated into a computer-executable form.

Next, it will be imperative for you to thoroughly understand the concept of an algorithm and to understand how good and bad algorithms ultimately affect program performance.

Finally, I will show you how C# programs are transformed into intermediate language and executed by the .NET Common Language Runtime (CLR). Armed with a fundamental understanding of computers, programs, and algorithms, you will be better prepared to understand the concepts of a *virtual machine*, as well as its execution performance and security ramifications.

What Is A Computer?

A computer is a device whose function, purpose, and behavior is prescribed, controlled, or changed via a set of stored instructions. A computer can also be described as a general-purpose machine. One minute a computer may execute instructions making it function as a word processor or page-layout machine. The next minute it might be functioning as a digital canvas for an artist. Again, this functionality is implemented as a series of instructions. Indeed, in each case the only difference between the computer functioning as a word processor and the same computer functioning as a digital canvas is in the set of instructions the computer is executing. This is what makes a computer a truly amazing device — it is a changeable machine.

Computer vs. Computer System

Due to the ever-shrinking size of the modern computer, it is often difficult for students to separate the concept of the computer from the computer system in which it resides. As a programmer, you will be concerned with both. You will need to understand issues related to the particular processor that powers a computer system in addition to issues related to the computer system as a whole. Luckily though, as a C# programmer, you can be extremely productive armed with only a high-level understanding of each. Ultimately, I highly recommend spending the time required to get intimately familiar with how your computer operates. In this chapter I use the Apple Mac Pro® as an example, but the concepts are the same for any computer or computer system.

Computer System

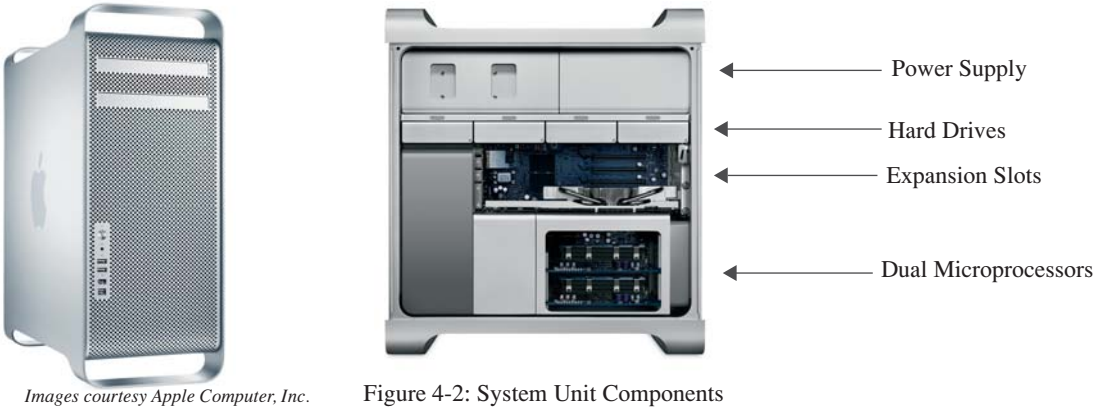
A typical Apple Mac Pro computer system is pictured in Figure 4-1.

Images courtesy Apple Computer, Inc.

Figure 4-1: Typical Apple Mac Pro Computer System

Referring to Figure 4-1 — the *computer system* comprises the system unit, monitor, wireless keyboard, mouse, and any other peripheral devices. The computer system also includes any operating system or utility software required to make all the components work together.

The *system unit* houses dual microprocessors, the power supply, internal hard disk drives, memory, and other system components required to interface the computer to the outside world. These interface components consume the majority of available space within the system unit, as shown in Figure 4-2.



Images courtesy Apple Computer, Inc.

Figure 4-2: System Unit Components

The dual microprocessors, or simply *processors*, are connected to the system unit's main logic board with the help of a set of specialized chips referred to as a *chipset*. Different types of microprocessors require different chipsets to help integrate them into the computer system. Electronic pathways called *buses* connect the processor to the various interface components. Other miscellaneous electronic components located on the main logic board control the flow of communication between the processors and the outside world. Figure 4-3 shows a block diagram of a Mac Pro main logic board.

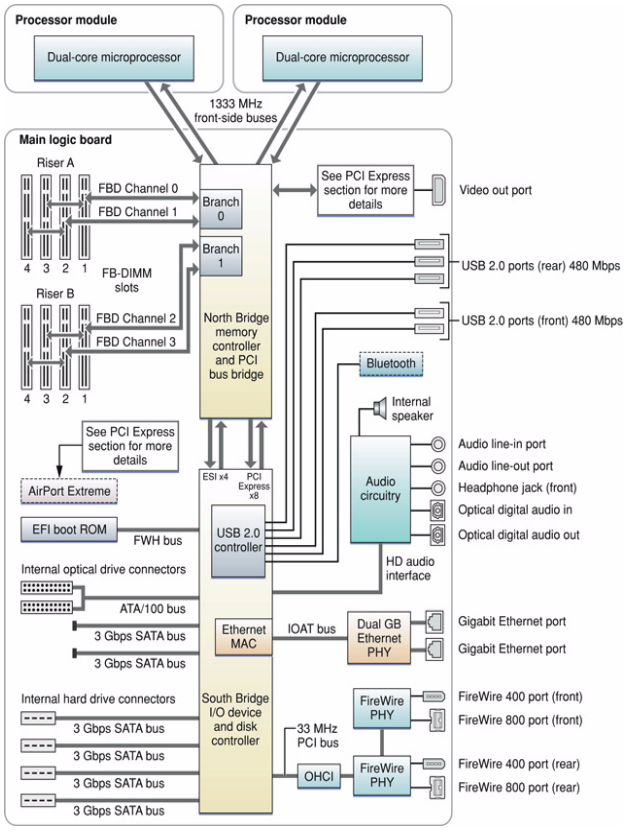


Image courtesy Apple Computer, Inc.

Figure 4-3: Main Logic Board Block Diagram

Figure 4-3 does a good job of highlighting the number of computer system support components required to help the processors do their job. The *main logic board* supports the addition of main memory, auxiliary storage devices, communication devices such as a modem, a wireless local area network card as well as high-speed Ethernet ports, keyboard, mouse, speakers, microphones, FireWire devices, and third-party system expansion cards. The heart of the system, however, consists of two Intel Xeon™ 5100 dual-core microprocessors. Let's take a closer look.

Processor

The Intel Xeon 5100 dual-core microprocessor pictured in Figure 4-4 is a 64-bit computer that contains two execution cores in one physical package. Furthermore, the Xeon 5100's design provides two logical processors for each execution core. The logical processors are utilized by Intel's Hyper-Threading Technology (HTT) to increase overall instruction processing throughput in multithreaded software applications.

Figure 4-5 shows a simplified block diagram of the Xeon 5100 dual-core processor architecture. As you can see in Figure 4-5, a computer system containing two Xeon 5100 processors actually has four execution units and eight logical processors. **Note:** The acronym APIC stands for Advanced Programmable Interrupt Controller.

Image courtesy of Intel Corp.

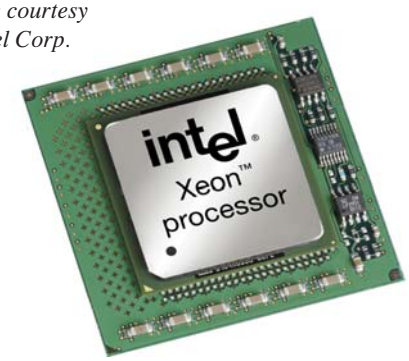


Figure 4-4: Intel Xeon 5100 Dual core Processor

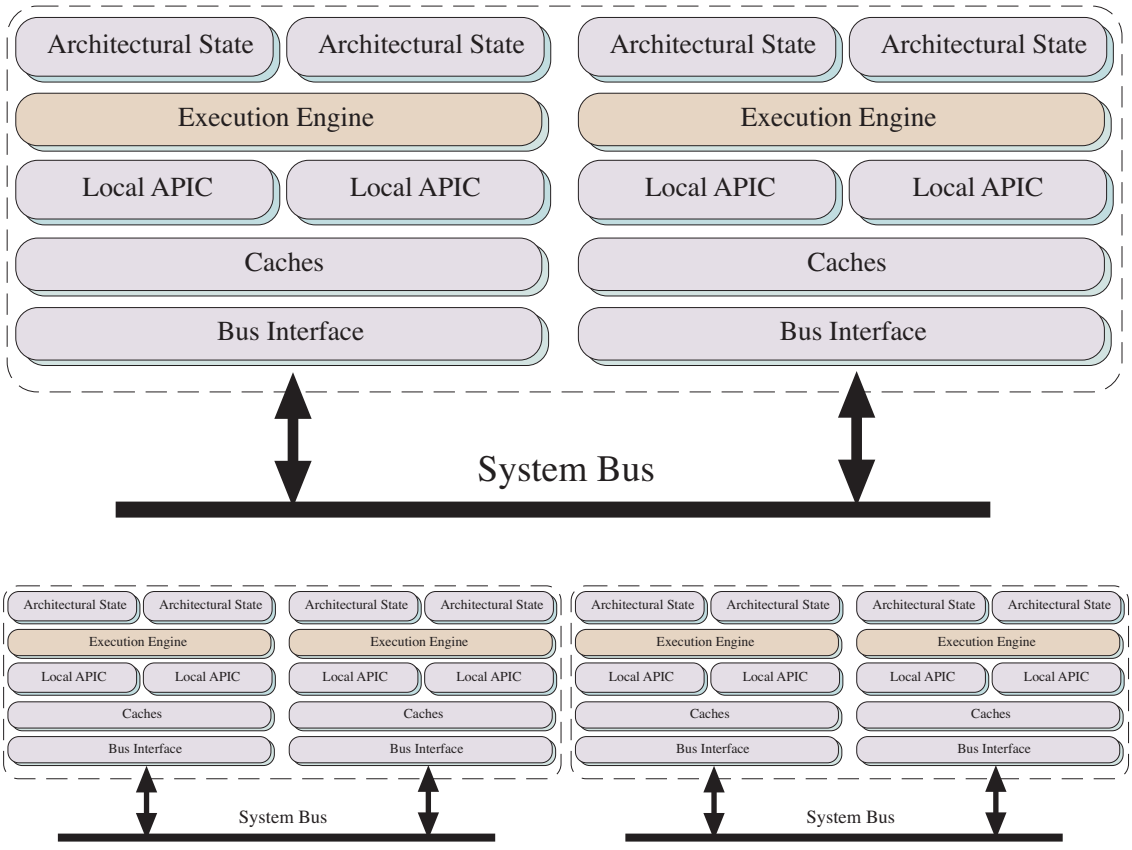


Figure 4-5: Intel Xeon 5100 Dual-Core Microprocessor Block Diagrams

Three Aspects of Processor Architecture

There are three aspects of processor architecture programmers should be aware of: feature set, feature set implementation, and feature set accessibility.

Feature Set

A processor's feature set derives from its design. Can floating point arithmetic be executed in hardware or must it be emulated in software? Must all data pass through the processor, or can input/output be handled off-chip while the processor goes about its business? How much memory can the processor access? How fast can it run? How much data can it process per unit time? A processor's design addresses these and other feature-set issues.

Feature Set Implementation

Feature set implementation primarily determines how a processor's functionality is arranged and executed in hardware. How does the processor implement the feature set? Is it a Reduced Instruction Set Computer (RISC) or a Complex Instruction Set Computer (CISC)? Is it superscalar and pipelined? Does it have a vector execution unit? Is the floating-point unit on the chip with the processor, or does it sit off to the side? Is the super fast cache memory part of the processor, or is it located on another chip? These questions all deal with how processor functionality is achieved or how its design is executed.

Feature Set Accessibility

Feature set accessibility is the aspect of a processor's architecture you are most concerned with as a programmer. Processor designers make a processor's feature set available to programmers via the processor's instruction set. A valid instruction in a processor's raw instruction set is a set of voltage levels that, when decoded by the processor, have special meaning. A high voltage is usually translated as "on" or "1", and a low voltage is usually translated as "off" or "0". A set of on-and-off voltages is conveniently represented to humans as a string of ones and zeros. Instructions in this format are generally referred to as machine instructions or machine code. As processor power increases, the size of machine instructions grows as well, making it extremely difficult for programmers to deal directly with machine code.

From Machine Code To Assembly Language

To make a processor's instruction set easier for humans to understand and work with, each machine instruction is represented symbolically in a set of instructions referred to as assembly language. To the programmer, assembly language represents an abstraction or a layer between programmer and machine intended to make the act of programming more efficient. Programs written in assembly language must be translated into machine instructions before being executed by the processor. A program called an assembler translates assembly language into machine code.

Although assembly language is easier to work with than machine code, it requires a lot of effort to crank out a program in assembly code. Assembly language programmers must busy themselves with issues like register usage and stack conventions.

High-level programming languages like C# add yet another layer of abstraction. C#, with its object-oriented language features, lets programmers think in terms of solving the problem at hand, not in terms of the processor or the machine code that it's ultimately executing.

Memory Organization

Modern computer systems have similar memory organizations. As a programmer, you should be aware of how computer memory is organized and accessed. The best way to get a good feel for how your computer works is to poke around in memory and see what's in there for yourself. This section provides a brief introduction to computer memory concepts to help get you started.

Memory Basics

A computer's memory stores information in the form of electronic voltages. There are two general types of memory: volatile and non-volatile. Volatile memory will lose stored information if power is removed for any length of time. Main memory and cache memory, two forms of random access memory (RAM), are examples of volatile memory. Read-only memory (ROM) and auxiliary storage devices such as CD-ROMs, DVDs, hard disk drives, USB flash drives, floppy disks, and tapes are examples of non-volatile memory.

Memory Hierarchy

Computer systems contain several different types of memory. These memory types range from slow and cheap to fast and expensive. The proportion of slow cheap memory to fast expensive memory can be viewed in the shape of a pyramid commonly referred to as the memory hierarchy, as shown in Figure 4-6.

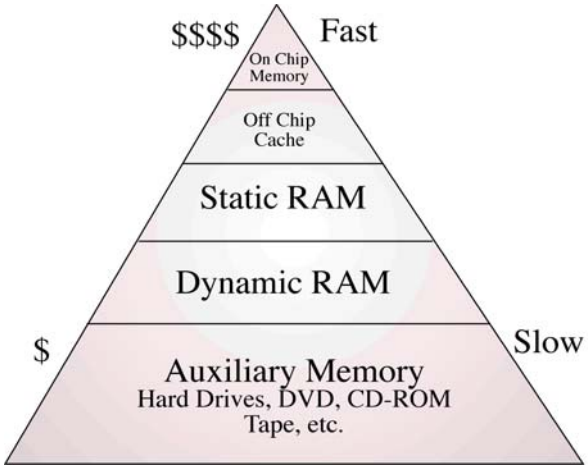


Figure 4-6: Memory Hierarchy

The job of a computer system designer with regards to memory subsystems is to make them perform as if all the memory they contained were fast and expensive. Utilizing cache memory to store frequently used data and instructions and buffering disk reads into memory give the appearance of faster disk access. Figure 4-7 shows a block diagram of the different types of memory used in a typical computer system.

During program execution, the faster cache memory is searched first by the processor for any requested data or instruction. If it's not there, a performance penalty occurs in the form of longer overall access times required to retrieve the information from a slower memory source. As chip densities grow, more cache memory will be located on the processor, thus improving overall processing times.

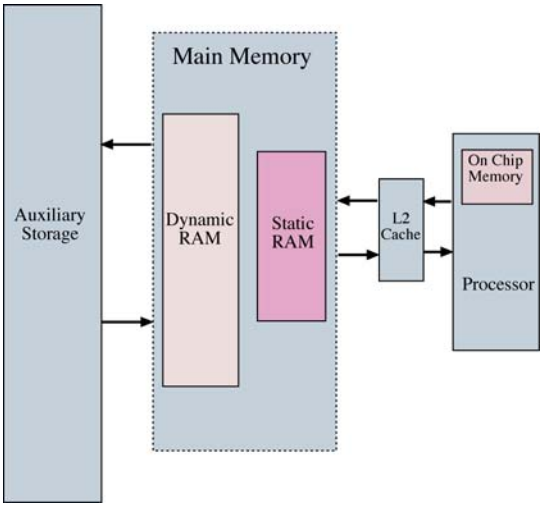


Figure 4-7: Simplified Memory Subsystem Diagram

Bits, Bytes, Words

Program code and data are stored in main memory as electronic voltages. Since I'm talking about digital computers, the voltage levels represent two discrete states depending on the level. Usually, low voltages represent no value, off, or 0, while a high voltage represents on, or 1.

When data is stored on auxiliary memory devices, electronic voltages are translated into either electromagnetic fields (tape drives, floppy and hard disks) or bumps that can be detected by laser beam (CDs, DVDs, etc.)

Bit

The bit represents one discrete piece of information stored in a computer. On most modern computer systems bits cannot be individually accessed from memory. However, after the byte to which a bit belongs is loaded into the processor, the byte can be manipulated to access a particular bit.

Byte

A byte contains 8 bits. Most computer memory is byte addressable, although as processors become increasingly powerful and can manipulate wider memory words, loading bytes by themselves into the processor becomes increasingly inefficient. This is the case with the Xeon processor. For that reason, the fastest memory reads can be done a word at a time.

Word

A word is a collection of bytes. The number of bytes that comprise a word is computer-system dependent. If a computer's data bus is 64 bits wide and its processor's registers are 64-bits wide, then the word size would be 8 bytes long (*64 bits / 8 bits = 8 bytes*). Bigger computers will have larger word sizes. This means they can manipulate more information per unit time than a computer with a smaller word size.

Alignment and Addressability

You can expect to find your computer system's memory to be byte addressable and word aligned. Figure 4-8 shows a simplified diagram of a main memory divided into bytes and the different buses connecting it to the processor. In this diagram, the word size is 64 bits wide.

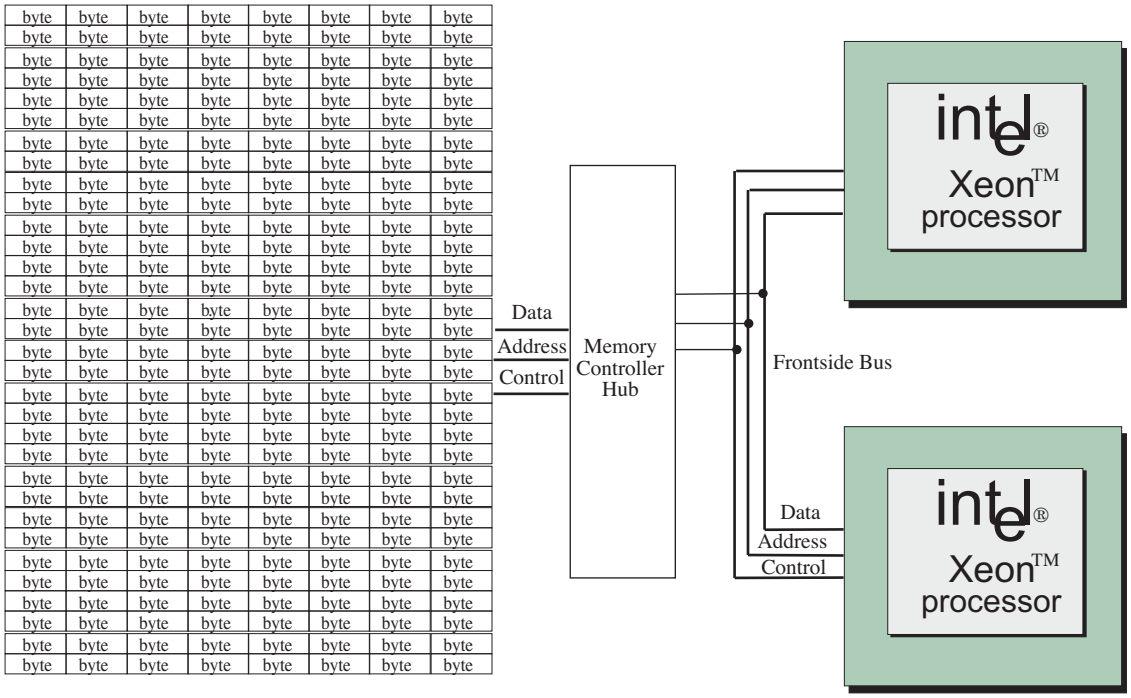


Figure 4-8: Simplified Main Memory Diagram

The memory is byte addressable in that each byte can be individually accessed although the entire word that contains the byte is read into the processor. Data in memory can be aligned for efficient manipulation. Alignment can be to either a natural boundary or other type of boundary. For example, on a Xeon system, the contents of memory assigned to instances of structures are aligned to natural boundaries, meaning a one-byte data element will be aligned to a one-byte boundary. A two-byte element would be aligned to a two-byte boundary, and so on. Individual data elements not belonging to structures are usually aligned to eight-byte boundaries.

What Is A Program?

Intuitively you already know the answer to this question. A program is something that runs on a computer. This simple definition works well enough for most purposes, but as a programmer you will need to arm yourself with a better understanding of exactly what makes a program a program. In this section I discuss programs from two aspects: the computer and the human. You will find this information extremely helpful, and it will tide you over until you take a formal course on computer architecture.

Two Views of a Program

A program is a set of programming language instructions plus any data the instructions act upon or manipulate. This is a reasonable definition if you are a human, but if you are a processor, it will just not fly. That's because humans are great abstract thinkers and computers are not, so it is helpful to view the definition of a program from two points of view.

The Human Perspective

Humans are the masters of abstract thought; it is the hallmark of our intelligence. High-level, object-oriented languages like C# give us the ability to analyze a problem abstractly and symbolically express its solution in a form that is both understandable by humans and readable by other programs. By other programs, I mean that the C# code a programmer writes must be translated from source code into machine instructions recognizable by a particular processor. This translation is effected by running a compiler that converts the C# code into an intermediate language that is then executed by the C# Common Language Runtime (CLR) Environment.

To a C# programmer, a program is a collection of classes that model the behavior of objects in a particular problem domain. These classes model object behavior by defining object attributes (data) and methods to manipulate these object attributes. On an even higher level, a program can be viewed as an interaction between objects. This view of a program is convenient for humans.

The Computer Perspective

From the computer's perspective, a program is simply machine instructions and data. Usually both the instructions and data reside in the same memory space. This is referred to as a Von Neumann architecture. In order for a program to run, it must first be loaded into main memory. The processor must then fetch the address of its first instruction, at which point execution begins. In the early days of computing, programs were coded into computers by hand and then executed. Nowadays, all of the nasty details of loading programs from auxiliary memory into main memory are handled by an operating system — which, by the way, is a program.

Since both instructions and data reside in main memory, how does a computer know when it is dealing with an instruction or with data? The answer to this question will be discussed in detail shortly, but here's a quick answer: it depends on what the computer is expecting. If a computer reads a memory location expecting to find an instruction and it does, everything runs fine. The instruction is decoded and executed. If it reads a memory location expecting to find an instruction but instead finds garbage, then the decode fails and the computer might lock up!

The Processing Cycle

Computers are powerful because they can do repetitive things really fast. When a computer executes or runs a program, it constantly repeats a series of processing steps commonly referred to as the processing cycle. The processing cycle consists of four primary steps: Instruction *Fetch*, Instruction *Decode*, Instruction *Execution*, and Result *Store*. The step names can be shortened to simply Fetch, Decode, Execute, and Store. Different types of processors implement the processing cycle differently, but generally all processors carry out these four processing steps in some form or another. The processing cycle is depicted in Figure 4-9.

Fetch

In the Fetch step, the processor reads an instruction from memory and presents it to its decode section. If cache memory is present, it is checked first. If the requested memory address contents resides in the cache, the read operation executes quickly. Otherwise, the processor must wait while the data is loaded from the slower main memory.

Decode

In the Decode step, the instruction fetched from memory is translated into voltages. If the computer thinks it is getting an instruction but instead gets garbage, there will be problems. A computer system's ability to recover from such situations is generally the function of a robust operating system.

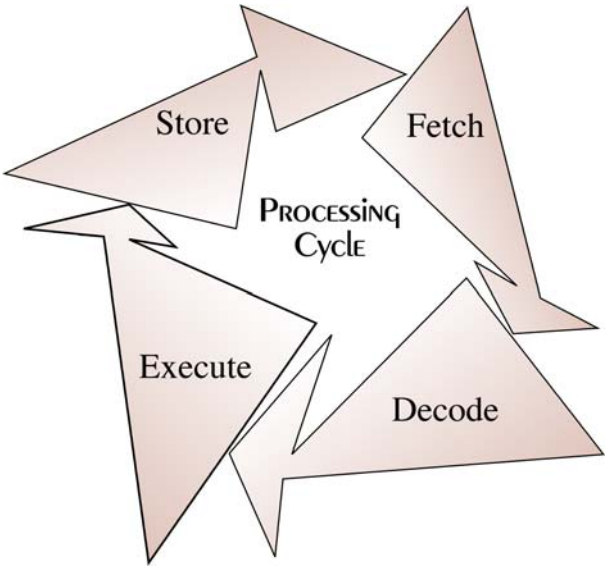


Figure 4-9: Processing Cycle

Execute

If the fetched instruction is successfully decoded as a valid instruction in the processor's instruction set, it is executed. A computer is a bunch of electronic switches. Executing an instruction means the computer's electronic switches are turned either on or off to carry out the actions required by a particular instruction. Different instructions cause different sets of switches to be turned on or off.

Store

When an instruction executes, the results, if any, must be stored somewhere. Most arithmetic instructions leave the result in one of the processor's onboard registers. Memory-write instructions would then be used to transfer these results to main memory. Keep in mind that there is only so much storage space inside a processor. At any given time, almost all data and instructions reside in main memory, and are only loaded into the processor when needed.

Why A Program Crashes

Notwithstanding catastrophic hardware failure, a computer crashes or locks up because what it was told was an instruction was not! The faulty instruction loaded from memory turns out to be an unrecognizable string of ones and zeros. When it fails to decode into a proper instruction, the computer halts.

Algorithms

Computers run programs; programs implement algorithms. A good working definition of an algorithm for the purpose of this book is that an algorithm is a recipe for getting something done on a computer. Pretty much every line of source code you write is considered part of an algorithm. What I'd like to do in this brief section is bring to your attention the concept of good vs. bad algorithms.

Good vs. Bad Algorithms

There are good ways to do something in source code and there are bad ways to do the same exact thing. A good example of this can be found in the act of sorting. Suppose you want to sort in ascending order the following list of integers:

1, 10, 7, 3, 9, 2, 4, 6, 5, 8, 0, 11

One algorithm for doing the sort might go something like this:

Step 1: Select the first integer position in the list
Step 2: Compare the selected integer with its immediate neighbor
Step 2.2: If the selected integer is greater than its neighbor, swap the two integers
Step 2.3: Else, leave it where it is
Step 3: Continue comparing selected integer position with all other integers repeating steps 2.2 - 2.3
Step 4: Select the second integer position on the list and repeat the procedure beginning at step 2
Continue in this fashion until all integers have been compared to all other integers in the list and have been placed in their proper position.

This algorithm is simple and straightforward. It also runs pretty fast for small lists of integers, but it is really slow given large lists of integers to sort. Another sorting algorithm to sort the same list of integers goes as follows:

Step 1: Split the list into two equal sublists
Step 2: Repeat step 1 if any sublist contains more than two integers
Step 3: Sort each sublist of two integers
Step 4: Combine sorted sublists until all sorted sublists have been combined

This algorithm runs a little slow on small lists because of all the list splitting going on, but sorts large lists of integers way faster than the first algorithm. The first algorithm lists the steps for a routine I call “dumb sort”. Example 4.1 gives the source code for a short program that implements the dumb sort algorithm.

4.1 DumbSort.cs

```
1    using System;
2
3    public class DumbSort{
4      public static void Main(String[] args){
5        int[] a = {1,10,7,3,9,2,4,6,5,8,0,11};
6
7        int innerloop = 0;
8        int outerloop = 0;
9        int swaps = 0;
10
11       for(int i=0; i<12; i++){
12         outerloop++;
13         for(int j=1; j<12; j++){
14           innerloop++;
15           if(a[j-1] > a[j]){
16             int temp = a[j-1];
17             a[j-1] = a[j];
18             a[j] = temp;
19             swaps++;
20           }
21         }
22       }
23
24       for(int i=0; i<12; i++){
25         Console.Write(a[i] + " ");
26       }
27
28       Console.WriteLine();
29       Console.WriteLine("Outer loop executed " + outerloop + " times.");
30       Console.WriteLine("Inner loop executed " + innerloop + " times.");
31       Console.WriteLine(swaps + " swaps completed.");
32
33     }
34   }
```

Included in the dumb sort test source code are a few variables intended to help collect statistics during execution. These are `innerloop`, `outerloop`, and `swaps`, declared on lines 7, 8, and 9, respectively. Figure 4-10 gives the results from running the dumb sort test program.

Notice that the inner loop executed 132 times and that 30 swaps were conducted. Can the algorithm run any better? One way to check is to rearrange the order of the integers in the array. What if the list of integers is already sorted? Figure 4-11 gives the results of running dumb sort on an ordered list of integers:

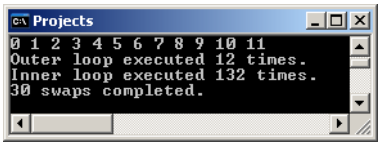


Figure 4-10: Dumb Sort Results 1

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
```

It appears that both the outer loop and inner loop execute the same number of times in each case, which is of course the way the source code is written. But it did run a little faster this time, because fewer swaps were necessary.

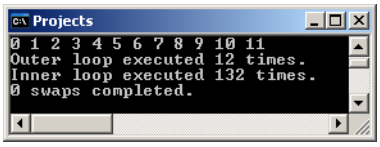


Figure 4-11: Dumb Sort Results 2

Can the algorithm run any worse? What if the list of integers is completely unsorted? Figure 4-12 gives the results of running dumb sort on a completely unsorted list:

```
11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
```

The outer loop and inner loop executed the same number of times, but 66 swaps were necessary to put everything in ascending order. So it did run a little slower this time.

In dumb sort, because we're sorting a list of 12 integers, the inner loop executes 12 times for every time the outer loop executes. If dumb sort needed to sort 10,000 integers, then the inner loop would need to execute 10,000 times for every time the outer loop executed. To generalize the performance of dumb sort, you could say that for some number n integers to sort, dumb sort executes the inner loop roughly n x n times. There is some other stuff going on besides loop iterations, but when n gets really large, the loop iteration becomes the overwhelming measure of dumb sort's performance as a sorting algorithm. Computer scientists would say that dumb sort has order n^2 performance. That is, for a really large list of integers to sort, the time it takes dumb sort to do its job is approximately the square of the number n of integers that need to be sorted.

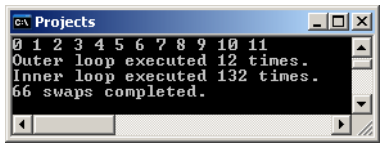


Figure 4-12: Dumb Sort Results 3

When an algorithm's running time is a function of the size of its input, the term used to describe the growth in time to perform its job vs. the size of the input is called the *growth rate*. Figure 4-13 shows a plot of algorithms with the following growth rates: *log n, n, n log n,* n^2, n^3, n^n.

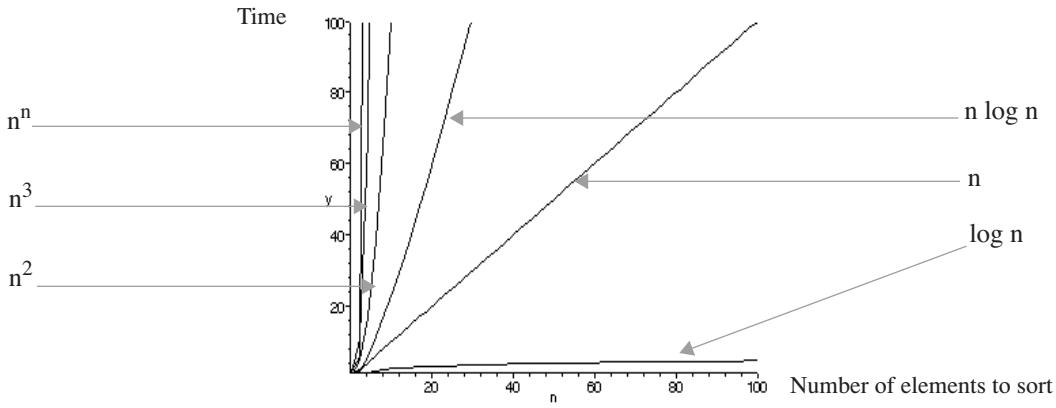


Figure 4-13: Algorithmic Growth Rates

As you can see from the graph, dumb sort, with a growth rate of n^2, is a bad algorithm, but not as bad as some other algorithms. The good thing about dumb sort is that no matter how big its input grows, it will eventually sort all the integers. Sorting problems are easily solved. There are some problems, however, that defy straightforward algorithmic solutions.

Don't Reinvent The Wheel!

If you are new to programming, the best advice I can offer you is to seek the knowledge of those who have come before you. There are many good books on algorithms, some of which are listed in the reference section. Studying good algorithms helps you write better code.

Virtual Machines And The Common Language Infrastructure

Figure 4-14 offers an overview of the C# compile and execute process.

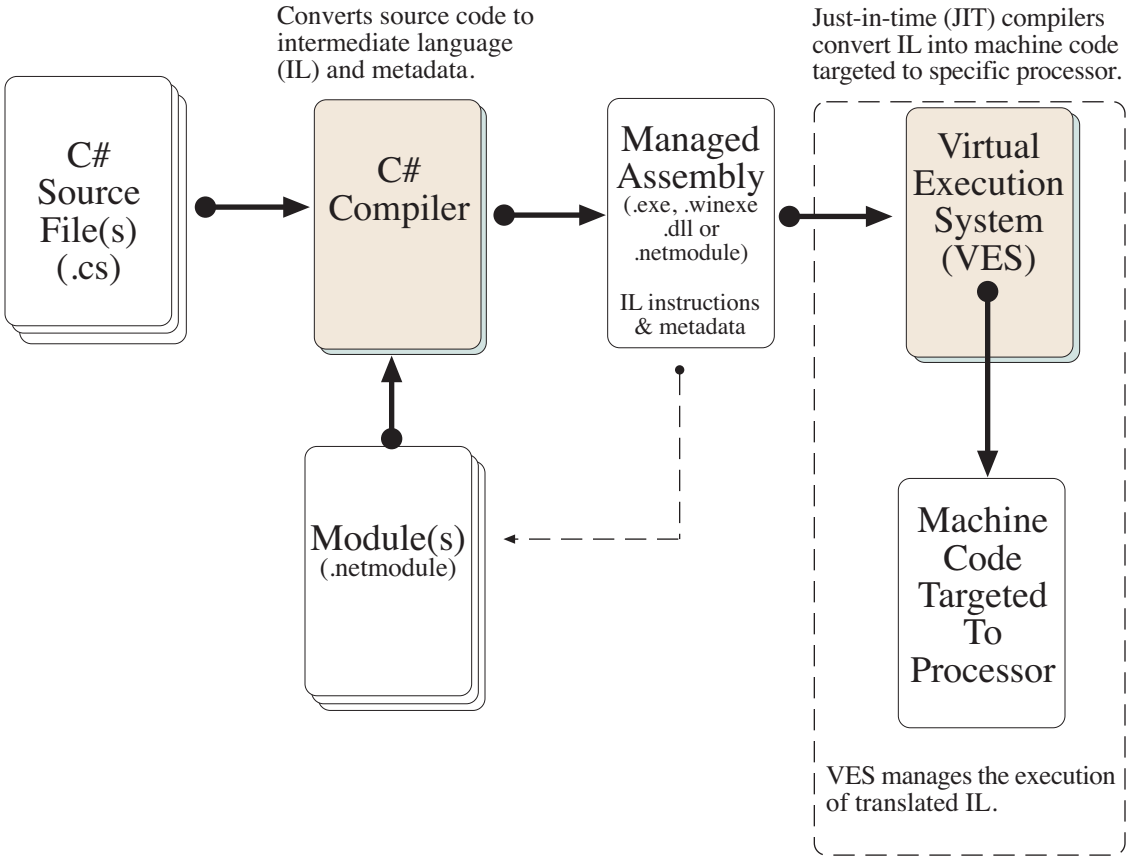


Figure 4-14: The C# Compile and Execution Process Overview

Referring to Figure 4-14 — the csc compiler compiles one or more C# source files into either a code module (.netmodule), a library (.dll), or one of two types of executable files: a console application (.exe) or a windows application (.winexe). Code modules are static IL code libraries whose code is referenced in your source file(s) and linked (added) to your project at compile time. A library, or dynamic link library (.dll), is a code module whose code is referenced in your source file(s) and loaded into the VES at application runtime. Hence the term "dynamic".

Executable files produced by the compiler can be loaded and executed by the VES. The VES executes and translates the IL instructions contained in the executable managed assembly. With the help of a JIT compiler, it produces machine code that is ultimately executed by the target processor. The JIT compiler is so named because it translates

IL into machine code as the IL instructions are executed by the VES. Blocks of compiled machine code are cached and tagged within the VES to prevent recompilation and to speed execution. Figure 4-15 shows what IL instructions look like.

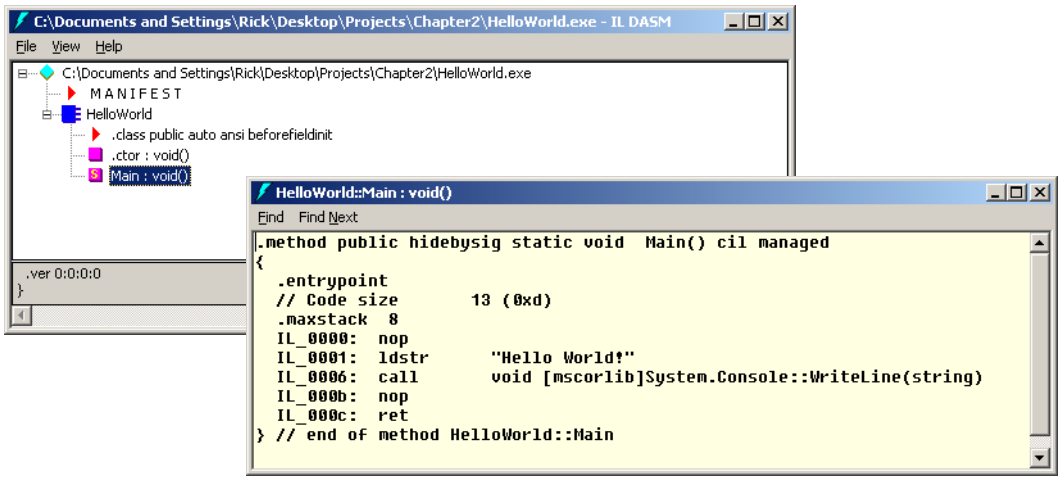


Figure 4-15: MSIL Disassembler Session Showing Main() Method IL Instructions

Referring to Figure 4-15 — the Microsoft Intermediate Language (MSIL) Disassembler tool provided by the Microsoft Windows Software Development Kit (SDK) is used to disassemble the HelloWorld.exe program used in Chapter 2. The IL instructions shown in the foreground window are those of the Main() method. These instructions are executed by the VES and translated into machine code when the HelloWorld.exe program executes.

Virtual Machines

The VES described in the previous section is a program (one or more software components acting in concert together) that executes IL instructions. In reality, the VES does more that simply execute IL instructions. I present a more detailed description of its responsibilities in the next section. Programs like the VES are referred to as *virtual machines*. The benefit of having C# target a virtual machine instead of a specific processor and operating system is the increased flexibility in the range of hardware and operating system environments on which C# programs can run.

A program written in any language whose compiler targets a specific processor must be recompiled for each different target processor on which the program must run. Not so with C#. Because the C# compiler targets a virtual machine, it can run on any computer platform that has on it an implementation of the VES. This cross-platform capability is made possible by an international standard known as ECMA - 335 Common Language Infrastructure (CLI) Partitions I to VI. So just what is this CLI and why should you care?

The Common Language Infrastructure (CLI)

ECMA - 335 specifies a CLI. As its name implies, the CLI specifies or lays down a set of rules that language makers, compiler makers, and virtual machine makers must follow if they want their languages and tools to run on different implementations of the CLI.

Four Parts Of The Common Language Infrastructure

The CLI provides architectural specifications for four areas: the Common Type System (CTS), metadata, the Common Language Specification (CLS), and the VES). Figure 14-16 graphically illustrates the relationship between these pieces of the CLI.

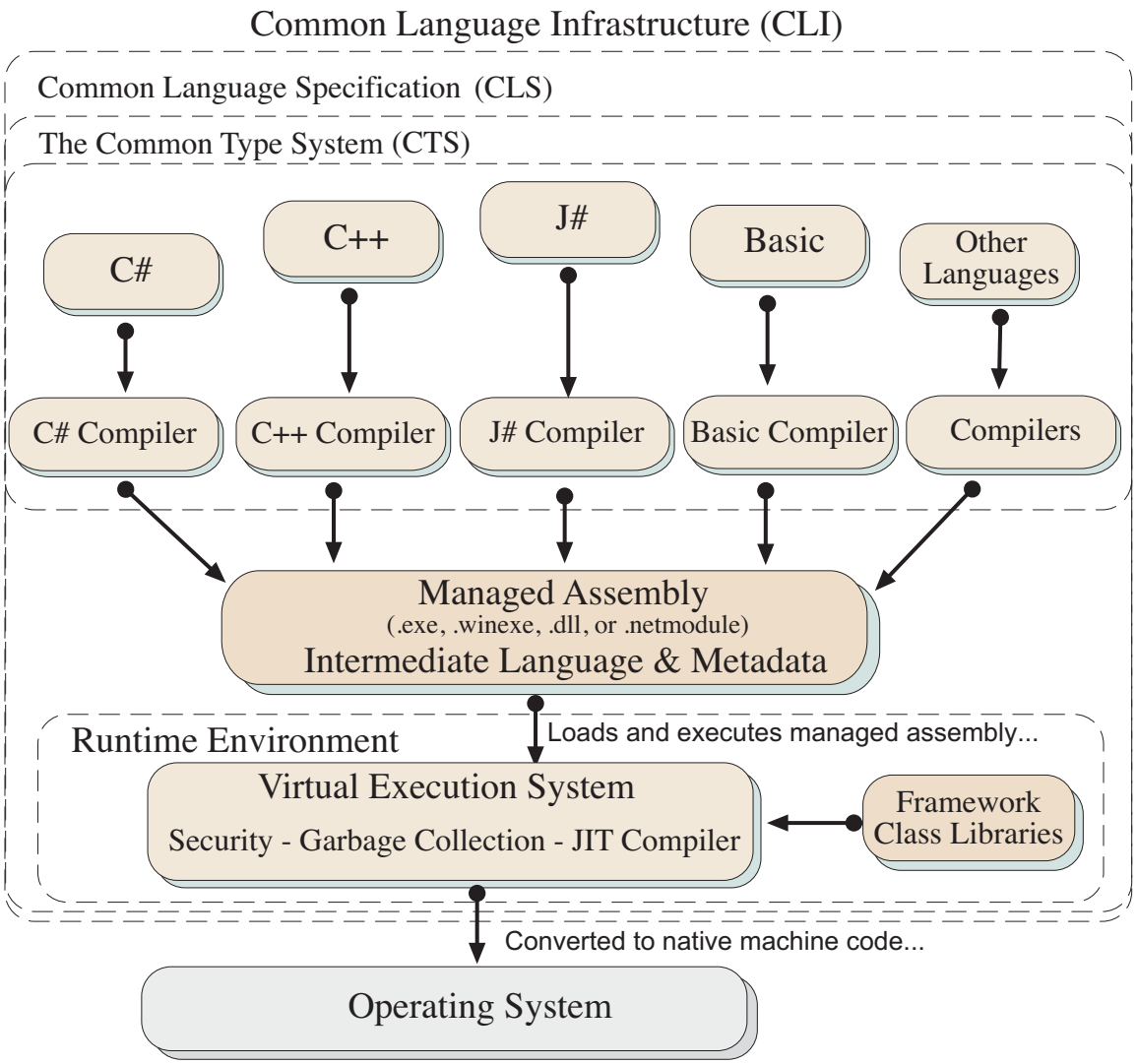


Figure 4-16: The Common Language Infrastructure Architecture

The Common Type System (CTS)

The CTS is the heart of the CLI. The CTS specifies a large set of types and operations common to many programming languages.

Metadata

The CLI uses metadata to describe and reference types defined by the CTS. Metadata can be thought of as data about data. Metadata is used by CLI tools and the Virtual Execution System (VES) to manipulate and manage IL code modules. Metadata is added to managed assemblies during the compilation process.

The Common Language Specification (CLS)

The CLS provides a set of rules that language and compiler implementors must follow to make their language interoperable with other CLI languages. Since languages share a CTS, modules generated by one language can be

used or referenced by programs written in another language. For example, Visual Basic.NET modules can be linked into and used by a program written in C#. This language interoperability is made possible by the CLS.

The Virtual Execution System (VES)

The VES executes *managed code* modules with the help of embedded metadata. **Note:** You can also write programs in C# that include what are referred to as *unmanaged code* segments. Unmanaged code segments allow direct access to the underlying operating system and hardware and thus tie a program to a specific platform.

The Cross Platform Promise

As long as you avoid unmanaged code, you can achieve some degree of cross-platform independence as Figure 4-17 illustrates, although in reality, Microsoft's implementation of the CLI (Microsoft .NET and the growing family of .NET compatible languages) will always, in my opinion, be well ahead of the competition.

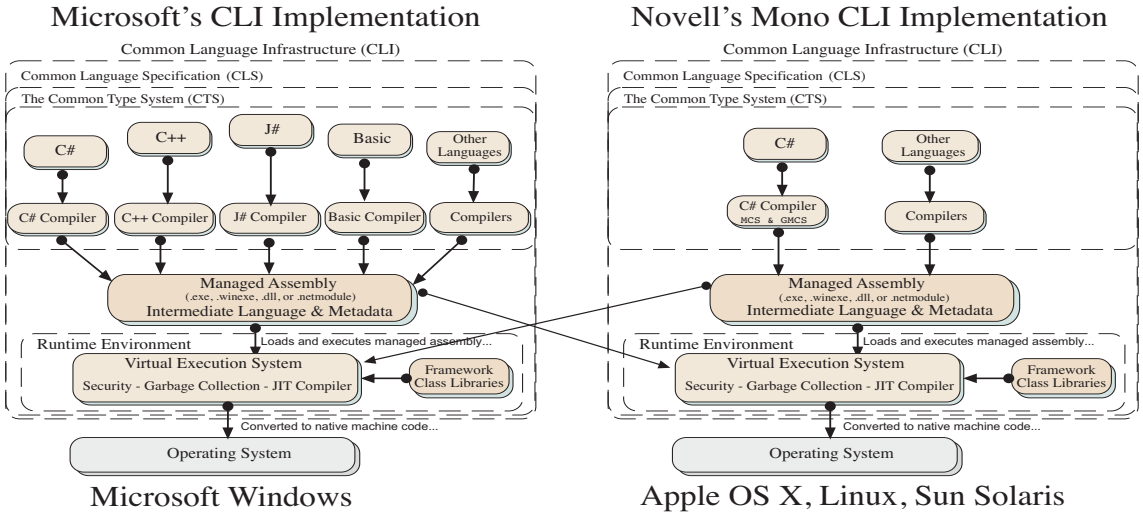


Figure 4-17: Managed Assemblies can be Executed on any System that Implements the Common Language Infrastructure

Referring to Figure 4-17 — as I write these words, the best attempt at a third party implementation of the CLI is the Novell's open-source Mono Project [http://mono-project.org]. As is shown in the diagram, the Mono project provides CLI implementations for Apple's OS X, various Linux platforms, Sun Solaris, and a few others not shown, including Microsoft Windows. Figure 4-18 shows the Robot Rat project of Chapter 3 executing in the Mono environment on a Macintosh G4 running Apple's OS X.

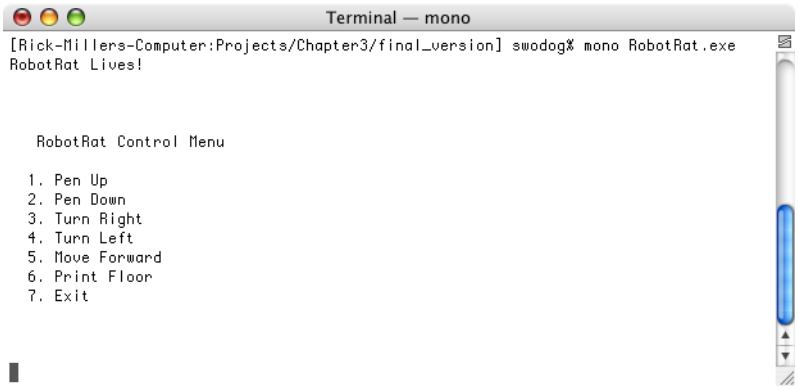


Figure 4-18: Chapter 3's Robot Rat Program Running in the Mono Environment on Apple OS X

Figure 4-19 gives a simple diagram of Microsoft's .NET architecture. Compare this diagram with that of Figure 4-16. Applications created with .NET languages consisting entirely of managed code segments are referred to as managed applications. Applications that combine managed and unmanaged code segments are referred to as hybrid applications. Microsoft's implementation of the VES is called the Common Language Runtime (CLR). The CLR and the .NET class libraries are included with the .NET Framework.

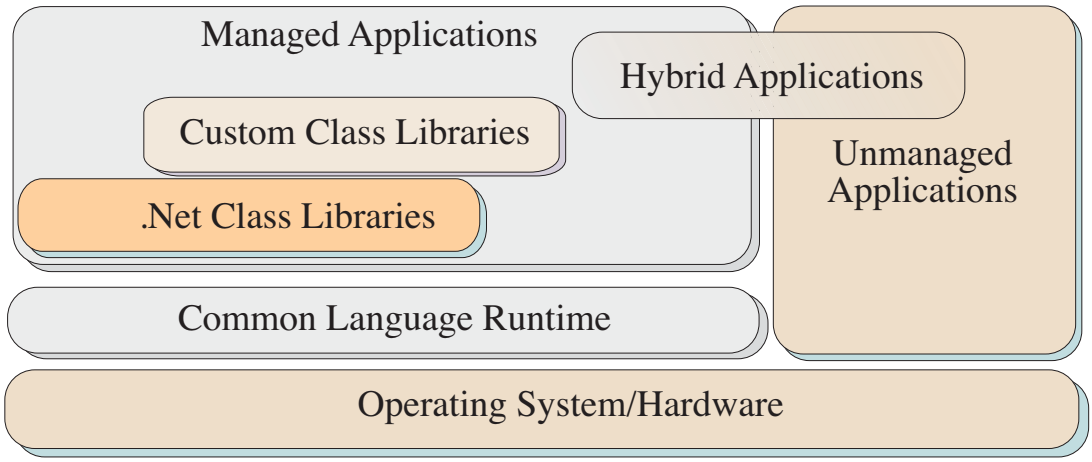


Figure 4-19: Microsoft .NET Architecture

Summary

Computers run programs; programs implement algorithms. As a programmer you need to be aware of development issues regarding your computer system and the processor it is based on.

A computer system contains a processor, I/O devices, and supporting operating system software. The processor is the heart of the computer system.

Programs can be viewed from two perspectives: human and computer. From the human perspective, programs, are a high-level solution statement to a particular problem. Object-oriented languages like C# help humans model extremely complex problems algorithmically. C# programs can also be viewed as the interaction between objects in a problem domain.

To a computer, programs are a sequence of machine instructions and data located in main memory. Processors run programs by rapidly executing the processing cycle of fetch, decode, execute, and store. If a processor expects an instruction but instead gets garbage, it is likely to lock up. Robust operating systems can mitigate this problem to a certain degree.

There are bad algorithms and good algorithms. Study from the pros to improve your code-writing skills.

Microsoft.NET is Microsoft's implementation of the Common Language Infrastructure (CLI) specification. Managed assemblies produced by the C# compiler contain descriptive metadata and execute within a Virtual Execution System (VES). The benefit of targeting a virtual machine is cross-platform execution.

C# is compiled into intermediate language (IL) instructions. The VES translates IL instructions into target processor machine code with the help of just-in-time (JIT) compilers.

If an assembly contains unmanaged code segments, then its cross-platform capabilities are limited.

Skill-Building Exercises

1. **Research Sorting Algorithms:** The second sorting algorithm listed on page 84 gives the steps for a Merge Sort. Obtain a book on algorithms, look for C# code that implements the Merge Sort algorithm, and compare it to Dumb Sort. What's the growth rate for a Merge Sort algorithm? How does it compare to Dumb Sort's growth rate?

2. **Research Sorting Algorithms:** Look for an example of a Bubble Sort algorithm. How does the Bubble Sort algorithm compare to Dumb Sort? What small changes can be made to Dumb Sort to improve its performance to that of Bubble Sort? What percentage of improvement is obtained by making the code changes? Will it make a difference for large lists of integers?

3. **Research The CLI:** Visit the ECMA website, download a copy of the CLI specification, and study the relationships between the CTS, metadata, the CLS, and the VES. [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

Suggested Projects

1. **Research Computer Systems:** Research your computer system. List all of its components including the type of processor. Go to the processor manufacturer's website and download developer information for your systems processor. Look for a block diagram of the processor and determine how many registers it has and their sizes. How does it get instructions and data from memory? How does it decode the instructions and process data?

2. **Compare Different Processors:** Select two different microprocessors and compare them to each other. List the feature set of each and determine how the architecture of each implements the feature set.

3. **Disassemble a Managed Assembly:** The Microsoft Windows SDK is separate from the .NET Framework Runtime that you may have downloaded in Chapter 2. Download and install the SDK, and use the MSIL Disassembler to disassemble one of your C# project's executable file and inspect its intermediate language instructions.

Self-Test Questions

1. List at least five components of a typical computer system.

2. What device do the peripheral components of a computer system exist to support?

3. From what two perspectives can programs be viewed? How does each perspective differ from the other?

4. What are the four steps of the processing cycle?

5. What, in your own words, does the term *algorithm* mean?

6. How does a processor's architecture serve to implement its feature set?

7. How can programmers access a processor's feature set?

8. What are the advantages of targeting a virtual machine vs. a physical processor? Can you think of any disadvantages?

9. What, if any, are the disadvantages of having unmanaged code segments in a C# program?

10. What is meant by the term just-in-time compiler?

References

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Intel Corporation Design Documentation, *IA-32 Intel Architecture Optimization Reference Manual*, Order Number: 248966-013US, April 2006

Intel Corporation Design Documentation, *Intel Xeon Processor with 512kb L2 Cache at 1.8 GHZ to 3 GHZ Datasheet*

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Apple Computer, Incorporated website [http://www.apple.com]

Notes