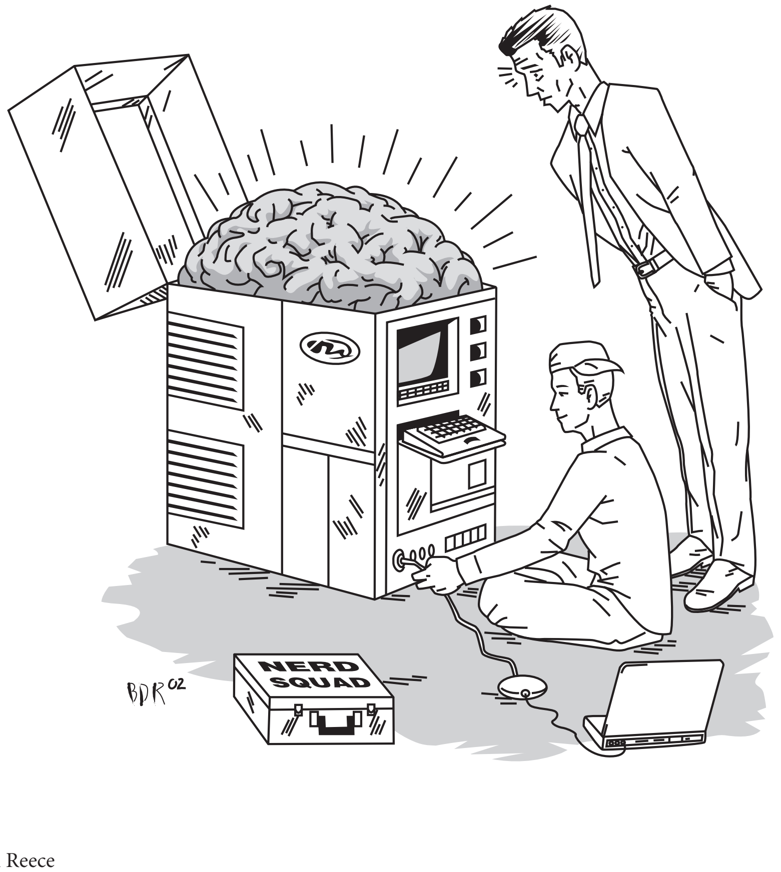
CHAPTER 8

CPU AND MEMORY: DESIGN, ENHANCEMENT, AND IMPLEMENTATION



Adapted by Benjamin Reece

8.0 INTRODUCTION

The Little Man Computer design, implemented in binary form, may be sufficient to implement any program, but it is not necessarily a convenient way to do so. It is like traveling overseas by freight steamer instead of by fast plane: it might be fun, but it sure ain't the easiest way to get the job done! Computers today are more sophisticated and flexible, providing a greater variety of instructions, improved methods of addressing memory and manipulating data, and implementation techniques that allow instructions to be executed quickly and efficiently.

In Chapter 7, we discussed the principal features of a CPU: the basic architecture of the CPU, register concept, instruction set, instruction formats, means of addressing memory, and the fetch-execute cycle. In this chapter we will investigate some of the additional design features and implementation techniques that help to give the modern CPU its power.

It probably won't surprise you to know that there are a large number of different ways of performing these tasks. At the same time, it is important to recognize, right from the outset, that additional features and a particular choice of organization do not change the fundamental operation of the computer as we have already described it. Rather, they represent variations on the ideas and techniques that we have already described. These variations can simplify the programmer's task and possibly speed up program execution by creating shortcuts for common operations. However, nothing introduced in this chapter changes the most important idea: that the computer is nothing more than a machine capable of performing simple operations at very high speeds.

The first section investigates different CPU architectures, with particular focus on the modern manifestation and organization of traditional architectures The section also briefly considers two interesting recent architectures, the Transmeta VLIW and Intel EPIC architectures.

In the second section we consider various CPU features and enhancements, with an emphasis on alternatives to the traditional control unit/ALU CPU organization. We explain how these alternative organizations address major bottlenecks that limit CPU execution speed, with a number of innovative techniques for improving CPU performance.

Section 8.3 looks at memory enhancements. The most significant improvement in memory access speed is cache memory. Cache memory is discussed in considerable depth.

In Section 8.4, we present a general model that includes the features, enhancements, and techniques described in Section 8.2. This model represents the organization of most current CPUs.

Section 8.5 considers the concept of multiprocessing: a computer organization consisting of multiple CPUs directly connected together, sharing memory, major buses, and I/O. This organization adds both performance enhancement and additional design challenges. We also briefly introduce a complementary feature, simultaneous multithreading. Two types of multiprocessors are presented: the symmetrical multiprocessor is more common. It is well-suited for general purpose computing. An alternative, the master-slave multiprocessor is useful for computer applications characterized by computationally intense, repetitive operations, such as graphics processing.

Finally, in Section 8.6, we present a brief commentary on the implementation of the CPU organization that we have discussed in previous sections.

It is not our intention to overwhelm you in this chapter with myriad details to memorize, nor to help you create a new career as an assembly language programmer or computer hardware engineer, but this chapter will at least introduce you to the major concepts, methods, and terminology used in modern computers. When reading this chapter, remember to keep your focus on the larger picture: the details are just variations on a theme.

8.1 CPU ARCHITECTURES

Overview

A CPU architecture is defined by the basic characteristics and major features of the CPU. (CPU architecture is sometimes called **instruction set architecture (ISA)**.) These characteristics include such things as the number and types of registers, methods of addressing memory, and basic design and layout of the instruction set. It does *not* include consideration of the implementation, instruction execution speed, details of the interface between the CPU and associated computer circuitry, and various optional features. These details are usually referred to as the computer's **organization**. The architecture may or may not include the absence or presence of particular instructions, the amount of addressable memory, or the data widths that are routinely processed by the CPU. Some architectures are more tightly defined than others.

These ideas about computer architecture should not surprise you. Consider house architecture. A split-level ranch house, for example, is easily recognized by its general characteristics, even though there may be wide differences in features, internal organization, and design from one split-level ranch to the next. Conversely, an A-frame house or a Georgian house is recognized by specific, well-defined features that must be present in the design to be recognized as A-frame or Georgian.

There have been many CPU architectures over the years, but only a few with longevity. In most cases, that longevity has resulted from evolution and expansion of the architecture to include new features, always with protection of the integrity of the original architecture, as well as with improved design, technology, and implementation of the architecture.

At present, important CPU architectural families include the IBM mainframe series, the Intel x86 family, the IBM POWER/PowerPC architecture, and the Sun SPARC family. Each of these is characterized by a lifetime exceeding twenty years. The original IBM mainframe architecture is more than forty-five years old. Architectural longevity protects the investment of users by allowing continued use of program applications through system upgrades and replacements.

Most CPU architectures today are variations on the traditional design described in Chapter 7.[1] There have also been a few interesting attempts to create other types, including a stack-based CPU with no general-purpose registers, and two recent architectures called **very long instruction word (VLIW)** from Transmeta and **explicitly parallel instruction computers (EPIC)** from Intel. VLIW and EPIC architectures are too new to assess their long-term value.

It should be noted that each of these architectures is consistent with the broad characteristics that define a von Neumann computer.

Traditional Modern Architectures

Early CPU architectures were characterized by comparatively few general-purpose registers, a wide variety of memory addressing techniques, a large number of specialized instructions, and instruction words of varying sizes. Researchers in the late 1970s and early 1980s concluded that these characteristics inhibited the efficient organization of the CPU. In particular, their studies revealed that

- Specialized instructions were used rarely, but added hardware complexity to the instruction decoder that slowed down execution of the other instructions that are used frequently.
- The number of data memory accesses and total MOVE instructions could be reduced by increasing the number of general-purpose registers and using those registers to manipulate data and perform calculations. The time to locate and access data in memory is much longer than that required to process data in a register and requires more steps in the fetch-execute cycle of instructions that access memory than those that don't.
- Permitting the use of general purpose registers to hold memory addresses, also, would allow the addressing of large amounts of memory while reducing instruction word size, addressing complexity, and instruction execution time, as well as simplifying the design of programs that require indexing. Reducing the number of available addressing methods simplifies CPU design significantly.
- The use of fixed-length, fixed-format instruction words with the op code and address fields in the same position for every instruction would allow instructions to be fetched and decoded independently and in parallel. With variable-length instructions it is necessary to wait until the previous instruction is decoded in order to establish its length and instruction format.

The Intel x86 is characteristic of older architectures; it has comparatively few general purpose registers, numerous addressing methods, dozens of specialized instructions, and instruction word formats that vary from 1 to 15 bytes in length. In contrast, every instruction

[1]Historically, traditional architecture was loosely categorized into one of two types, CISC (complex instruction set computers) or RISC (reduced instruction set computers). In modern times, the dividing line between CISC and RISC architectures has become increasingly blurred as many of the features of each have migrated across the dividing line. Because modern architectures contain the major features of both, it is no longer useful to distinguish one from the other.

in the newer Sun SPARC architecture is the same 32-bit length; there are only five primary instruction word formats, shown earlier in Figure 7.21; and only a single, register-based, LOAD/STORE memory addressing mode.

VLIW and EPIC Architectures

VLIW (very long instruction word) and EPIC (explicitly parallel instruction computer) architectures represent recent approaches to architectural design. VLIW architecture is represented by the Transmeta Crusoe and Efficeon families of CPU processors. The Intel Itanium IA-64 series is based on EPIC architecture. The basic goal of each of these architectures is to increase execution speed by processing instruction operations in parallel. The primary difficulty in doing so results from the inherently sequential order of the instructions in a program. In particular, the data used in an instruction may depend on the result from a previous instruction. This situation is known as a **data dependency**. Also, branches and loops may alter the sequence, resulting in **control dependency**. Data and control dependencies are discussed in more depth in Section 8.2.

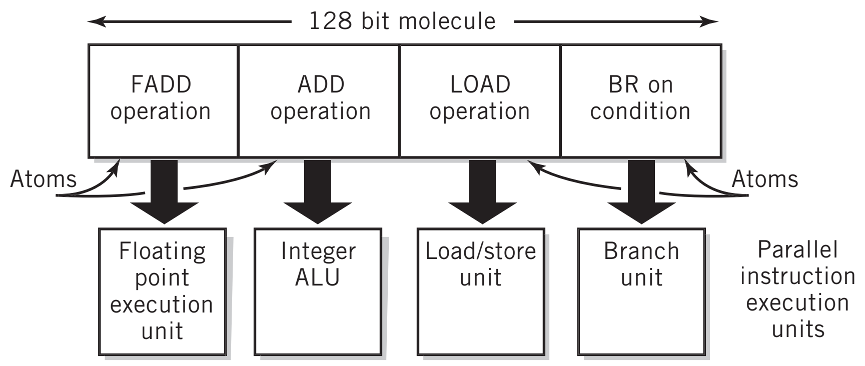
The Transmeta Crusoe architecture is based on a 128-bit instruction word called a molecule. The molecule is divided into four 32-bit *atoms*. Each atom represents an operation similar to those of a normal 32-bit instruction word, however, the atoms are designed in such a way that all four operations may be executed simultaneously in separate execution units. Figure 8.1 shows an example of a typical molecule.

The Crusoe CPU provides 64 general-purpose registers to assure adequate register space for rapid register-to-register processing.

Although a programmer could write programs directly for a Crusoe CPU with the 128-bit word instruction set, that is not the primary goal of the Crusoe architecture. Indeed, the fine details of the instruction set have not been publicly released to date. Instead, the Crusoe CPU is intended for use with a specific software program that translates instruction sets on the fly from other types of CPUs to the Crusoe instruction set for execution on

FIGURE 8.1

VLIW Format



the Crusoe CPU. This translator is called a **code-morphing layer**.[2] It is a fundamental component within the Crusoe architecture. It is permanently resident in memory and processes every instruction prior to execution. In addition to instruction translation, the code-morphing layer also reorders the instructions as necessary to eliminate data dependencies and other bottlenecks. Although this sounds like an inefficient way to process instructions, Transmeta has demonstrated that the simplicity of its VLIW design and the sophistication of its code-morphing software allow execution of the Pentium instruction set at speeds comparable to the native execution speeds of a Pentium processor. Transmeta claims, with apparent justification, that this simplicity allows a much simpler CPU design, with fewer transistors and a much lower power consumption, resulting from the elimination of complicated hardware implementation features commonly used to achieve high execution speeds in a conventional CPU design. The Efficeon CPU extends the Crusoe architecture to 256 bits, representing eight 32-bit atoms to be executed simultaneously.

At present, Transmeta has provided a code-morphing layer only for the Pentium CPU family. However, if there were reason to do so, Transmeta could easily create code-morphing layers for other CPUs.

The EPIC architecture, designed by Intel for its IA-64 processor family, attempts to achieve similar goals by slightly different means. The basic instruction set architecture is new, although Intel has built x86 capability into the CPU to support compatibility with its earlier architecture. The IA-64 offers 128 64-bit general-purpose registers and 128 82-bit floating point registers. All instructions are 41 bits wide.

Like the VLIW architecture, the EPIC architecture also organizes instructions into bundles prior to CPU execution, however the methodology and goal are somewhat different. In this case, the instructions *do* represent the native instruction set of the architecture. Instructions are presented to the CPU for execution in 128-bit bundles that include a group of three instructions plus 5 bits that identify the type of each instruction in the bundle.

An assembly language programmer is expected to follow a set of published guidelines that identify dependencies and allow the parallel execution of each bundle. Additionally, bits within each instruction word provide information to the execution unit that identify potential dependencies and other bottlenecks and help the programmer to optimize the code for fast execution. High-level language EPIC compilers must also create code that satisfies the guidelines.

A fundamental difference between the Transmeta VLIW and the Intel EPIC architectures is the placement of responsibility for correct instruction sequencing. The VLIW architecture allows any sequence of instructions to enter the CPU for processing. The code-morphing software, integral to the architecture, handles proper sequencing. The EPIC architecture places the burden on the assembly language programmer or on the program compiler software.

This does not suggest that one architecture is superior to the other. It simply indicates a different approach to the solution of dependencies. Note that the Transmeta VLIW does

[2]On a lesser scale, code morphing can also be used to translate complex variable-width instruction words to simpler fixed-width equivalents for faster execution. This technique allows the retention of legacy architectures while permitting the use of modern processing methods. Modern x86 implementations use this approach.

not allow direct assembly language access to the CPU. *All* program code must be processed through code-morphing software. Each architecture offers an interesting new approach to program execution with potential benefits.

8.2 CPU FEATURES AND ENHANCEMENTS

Introduction

We have already introduced you to the fundamental model of a traditional CPU, represented by an instruction set, registers, and a fetch-execute instruction cycle. Additionally, we have presented some of the bells and whistles that have enhanced CPU capability and performance. Some of the enhancements that were introduced in Chapter 7 include direct support for floating point arithmetic, BCD arithmetic, and multimedia processing, as well as the inclusion of additional addressing modes, which simplify data access, increase potential memory size capability while maintaining reasonable instruction word sizes, and improve list and array processing. In this chapter we have already presented a number of architectural enhancements that can improve performance including features that allow parallel execution of instructions to improve processing speed, register-oriented instructions, the use of fixed-width instructions, and integral code-morphing software.

Since the purpose of a computer is to execute programs, the ability of the CPU to execute instructions quickly is an important contributor to performance. Once a particular architecture is established, there remain a number of different ways to increase the instruction execution performance of a computer. One method is to provide a number of CPUs in the computer rather than just one. Since a single CPU can process only one instruction at a time, each additional CPU would, in theory, multiply the performance of the computer by the number of CPUs included. We will return to a discussion of this technique later, in Section 8.5.

Of more interest at the moment are approaches that can be used to improve the performance of an individual CPU. In our introduction to CPU architectures, we suggested a number of possibilities. Some of these require new design, such as the large number of registers and register-to-register instructions that are characteristic of newer architectures. As we already noted, even with older instruction sets, it is often possible to use code morphing to create an intermediate instruction set that is used within the CPU as a substitute for the more complex, original instruction set.

Another difficulty to be overcome when attempting system optimization is that some computer instructions inherently require a large number of fetch-execute steps. Integer division and floating point arithmetic instructions are in this category. Obviously, CPU architects cannot create modern instruction sets that omit these instructions.

In this section, we consider a number of different, but interrelated, approaches to CPU optimization that are applicable to nearly any CPU design. Interestingly enough, you will see that similar approaches can be found in such diverse operations as automobile assembly plants and restaurants.

In Chapter 7, you learned that the fetch-execute cycle is the basic operation by which instructions get executed. You also observed that the steps in a fetch-execute cycle generally must be performed in a particular sequence: an instruction must be fetched and identified before it can be executed, for example. Otherwise the machine would have no way of knowing what to execute. And so on, step by step, through the entire instruction cycle. *(The first step in cooking spaghetti is to add water to the pot.)* CPU performance can be improved by any method that can perform the fetch-execute cycle steps more quickly or more efficiently.

Then, a program is executed by performing the fetch-execute cycle in a specified sequence, where the sequence is sometimes determined by the program itself during execution. To be provably correct during program execution, the sequence must be maintained and data dependencies resolved in proper order. *(The "cook spaghetti," "drain spaghetti," and "prepare sauce" instructions must be completed before the sauce is mixed into the spaghetti.)*

Observe that the limitation to performance results from the serial nature of CPU processing: each instruction requires a sequence of fetch-execute cycle steps, and the program requires the execution of a sequence of these instructions. Thus, the keys to increased performance must rely on methods that either reduce the number of steps in the fetch-execute cycle or reduce the time required for each step in the cycle and, ultimately, reduce the time for each instruction in the program.

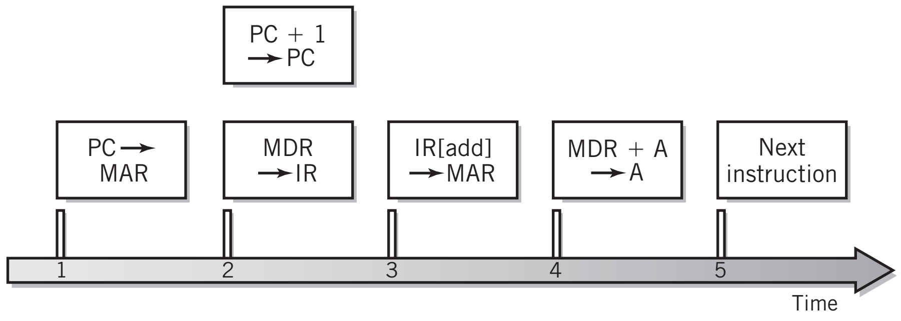
Fetch-Execute Cycle Timing Issues

As a first step, consider the problem of controlling the timing of each step in the fetch-execute cycle to guarantee perfect CPU operation, to assure that each step follows the previous step, in perfect order, as quickly as possible. There must be enough time between steps to assure that each operation is complete and that data is where it is supposed to be before the next step takes place. As you saw in Chapter 7, most steps in the fetch-execute cycle work by copying, combining, or moving data between various registers. When data is copied, combined, or moved between registers, it takes a short, but finite, amount of time for the data to "settle down" in the new register, that is, for the results of the operation to be correct. This occurs in part because the electronic switches that connect the registers operate at slightly different speeds. (We're actually talking billionths of a second here!) Also, design allowances must be made for the fact that some operations take longer than others; for example, addition takes more time than a simple data movement. Even more significant is the amount of time that it takes for the address stored in the MAR to activate the correct address in memory. The latter time factor is due to the complex electronic circuitry that is required to identify one group of memory cells out of several million or billion possibilities. This means that reducing the number of memory access steps by using registers for most data operations will inherently improve performance. (We discuss methods to reduce the memory access time, itself, in Section 8.3.) To assure adequate time for each step, the times at which different events take place are synchronized to the pulses of an electronic clock. The **clock** provides a master control as to when each step in the instruction cycle takes place. The pulses of the clock are separated sufficiently to assure that each step has time to complete, with the data settled down, before the results of that step are required by the next step. Thus, use of a faster clock alone does not work if the circuitry cannot keep up.

A timing cycle for a Little Man ADD instruction is shown in Figure 8.2. Each block in the diagram represents one step of the fetch-execute cycle. Certain steps that do not have to access memory and which are not dependent on previous steps can actually be performed at the same time. This can reduce the overall number of cycles required for the

FIGURE 8.2

Fetch-Execute Timing Diagram



instruction, which speeds up the computer. In this diagram, the data from the program counter has been copied into the memory address register in the first step and is no longer needed. Therefore, the program counter can be incremented at any time after the first step. In Figure 8.2 the PC is incremented in parallel with the MDR → IR step. As shown in the figure, the ADD instruction is completed in four clock cycles.

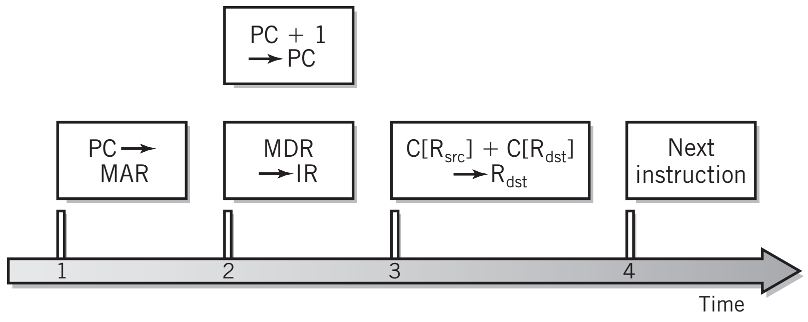
Figure 8.3 shows the improvement possible by using multiple data registers to implement an ADD instruction. Since the register-to-register add can be done directly, the number of steps in the cycle is reduced from four to three, with only a single execute step, and the extra time required for the memory access is eliminated.

The built-in clock runs continuously whenever the power to the computer is on. The frequency of its pulses is controlled by a quartz crystal, similar to that which might control your wristwatch. The frequency of the clock and the number of steps required by each instruction determine the speed with which the computer performs useful work.

The pulses of the clock are combined with the data in the instruction register to control electronic switches that open and close in the right sequence to move data from

FIGURE 8.3

Fetch-Execute Cycle for Register-to-Register ADD Instruction



Note: $C[R_{dst}]$ = contents of destination register

register to register in accordance with the instruction cycle for the particular instruction. The memory activation line described in Section 7.3 is an example of a timing line. The activation line is set up so that it will not turn on until the correct address decode line in the MAR has had time to settle down. If this were not the case, several address lines might be partially turned on, and the data transferred between the memory and MDR might be incorrect. Such errors can obviously not be tolerated, so it is important to control timing accurately.

Conceptually, each pulse of the clock is used to control one step in the sequence, although it is sometimes possible to perform multiple operations within a single step. The clock in the original IBM PC, for example, ran at 4.77 MHz (MHz is pronounced megahertz), which meant that the machine could perform 4.77 million steps every second. If a typical instruction in the IBM PC requires about ten steps, then the original IBM PC could execute about (4.77/10) or about 0.5 million PC instructions per second. A PC running at 8 MHz, with everything else equal, would perform approximately twice as fast.

There are several factors that determine the number of instructions that a computer can perform in a second. Obviously the clock speed is one major factor. Some current PC computers run their clocks at 3 GHz (pronounced gigahertz) or even more to achieve higher instruction cycle rates.

A Model for Improved CPU Performance

The current organizational model of a CPU uses three primary, interrelated techniques to address the limitations of the conventional CU/ALU model and to improve performance.

- Implementation of the fetch-execute cycle is divided into two separate units: a fetch unit to retrieve and decode instructions and an execution unit to perform the actual instruction operation. This allows independent, concurrent operation of the two parts of the fetch-execute cycle.
- The model uses an assembly line technique called *pipelining* to allow overlapping between the fetch-execute cycles of sequences of instructions. This reduces the average time needed to complete an instruction.
- The model provides separate execution units for different types of instructions. This makes it possible to separate instructions with different numbers of execution steps for more efficient processing. It also allows the parallel execution of unrelated instructions by directing each instruction to its own execution unit. You have already seen this method applied to the Transmeta and Itanium architectures in Section 8.1.

We next consider each of these techniques in turn.

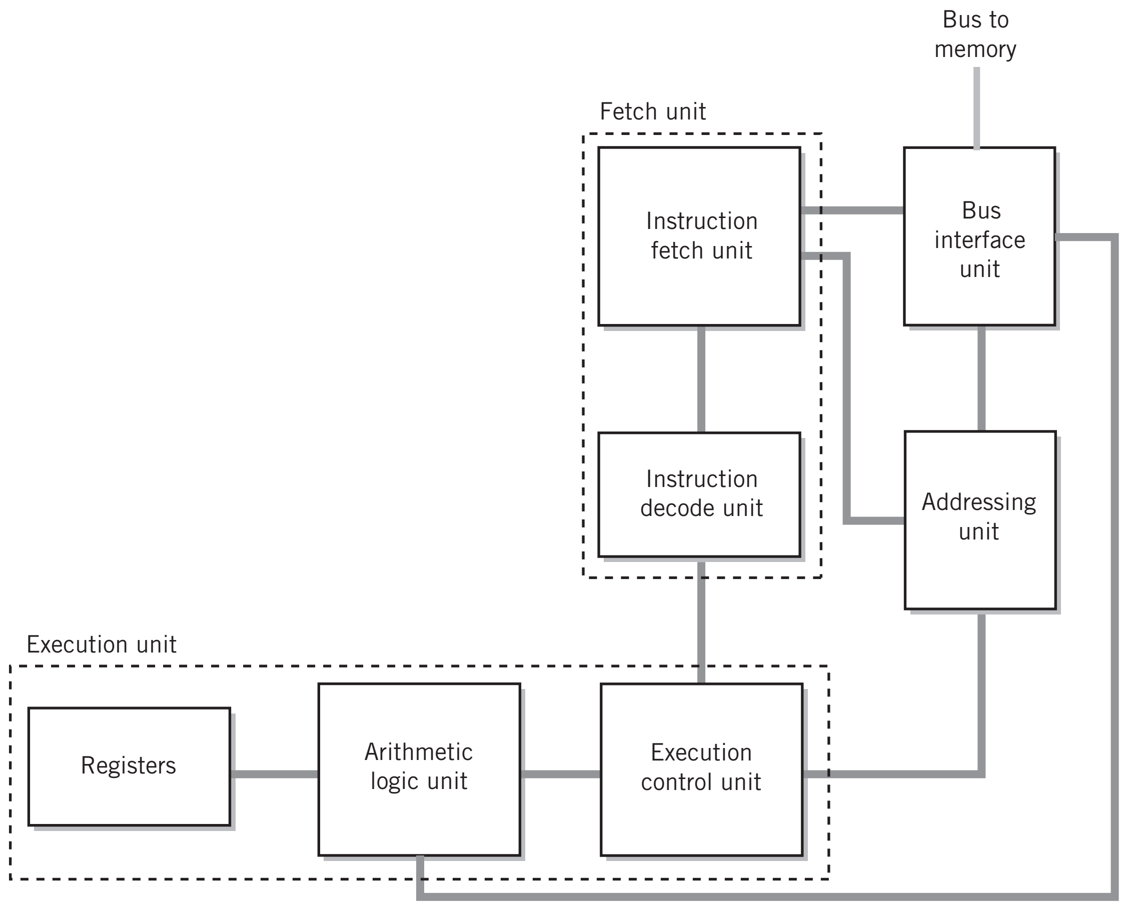
SEPARATE FETCH UNIT/EXECUTE UNIT Picture a modified Little Man Computer in which the Little Man has been given an assistant. The assistant will fetch and decode the instructions from the mailboxes at a pace that allows the Little Man to spend his time executing instructions, one after another. Note that a similar division of labor is used in a restaurant: waiters and waitresses gather the food orders from the customers and pass them to the cooks for processing.

The current preferred CPU implementation model divides the CPU similarly into two units, which correspond roughly to the fetch and execute parts of the instruction cycle. To achieve maximum performance, these two parts operate as independently from each other as possible, recognizing, of course, that an instruction must be fetched before it can be decoded and executed. Figure 8.4 illustrates this alternative CPU organization.

The **fetch unit** portion of the CPU consists of an instruction fetch unit and an instruction decode unit. Instructions are fetched from memory by the fetch unit, based on the current address stored in an instruction pointer (IP) register. The fetch unit is designed to fetch several instructions at a time in parallel. The IP register effectively acts as a program counter, but is given a different name to emphasize that there are a number of instructions in the pipeline simultaneously. There is a bus interface unit that provides the logic and memory registers necessary to address memory over the bus. Once an instruction is fetched, it is held in a buffer until it can be decoded and executed. The number of instructions held will depend upon the size of each instruction, the width of the memory bus and memory

FIGURE 8.4

Alternative CPU Organization



data register[3], and the size of the buffer. As instructions are executed, the fetch unit takes advantage of time when the bus is not otherwise being used and attempts to keep the buffer filled with instructions. In general, modern memory buses are wide enough and fast enough that they do not limit instruction retrieval.

Recall that in Figure 8.3 we showed that register-to-register operations could be implemented with only a single memory access, in the fetch portion of the fetch-execute cycle. Fetching the instructions in advance allows the execution of these instructions to take place quickly, without the delay required to access memory.

Instructions in the fetch unit buffer are sent to the instruction decoder unit. The decoder unit identifies the op code. From the op code it determines the type of the instruction. If the instruction set is made up of variable length instructions, it also determines the length of the particular instruction. The decoder then assembles the complete instruction with its operands, ready for execution.

The **execution unit** contains the arithmetic/logic unit and the portion of the control unit that identifies and controls the steps that comprise the execution part for each different instruction. The remainder of what we previously called the control unit is distributed throughout the model, controlling the fetching and decoding of instructions at the correct times, and in the correct order, address generation for instructions and operands, and so forth. The ALU provides the usual computational abilities for the general registers and condition flags.

When the execution unit is ready for an instruction, the instruction decoder unit passes the next instruction to the control unit for execution. Instruction operands requiring memory references are sent to the addressing unit. The addressing unit determines the memory address required, and the appropriate data read or write request is then processed by the bus interface unit.

The bus interface and addressing units operate independently of the instruction pipeline and provide services to the fetch, decode, and execution units as requested by each unit.

PIPELINING Look at Figure 8.2 again. In the figure, there are two stages to the execution phase of the instruction cycle. If each stage is implemented separately, so that the instruction simply passes from one stage to the next as it is executed, only one stage is in use at any given time. If there are more steps in the cycle, the same is still true. Thus, to speed up processing even more, modern computers overlap instructions, so that more than one instruction is being worked on at a time. This method is known as **pipelining**. The pipelining concept is one of the major advances in modern computing design. It has been responsible for large increases in program execution speed.

In its simplest form, the idea of pipelining is that as each instruction completes a step, the following instruction moves into the stage just vacated. Thus, when the first instruction is completed, the next one is already one stage short of completion. If there are many steps in the fetch-execute cycle, we can have several instructions at various points in the cycle. The method is similar to an automobile assembly line, where several cars are in different degrees of production at the same time. It still takes the same amount of time to complete

[3] Recall that in Chapter 7 we noted that it is common modern practice to retrieve several bytes from memory with each memory access.

one instruction cycle (or one car), but the pipelining technique results in a large overall increase in the average number of instructions performed in a given time.

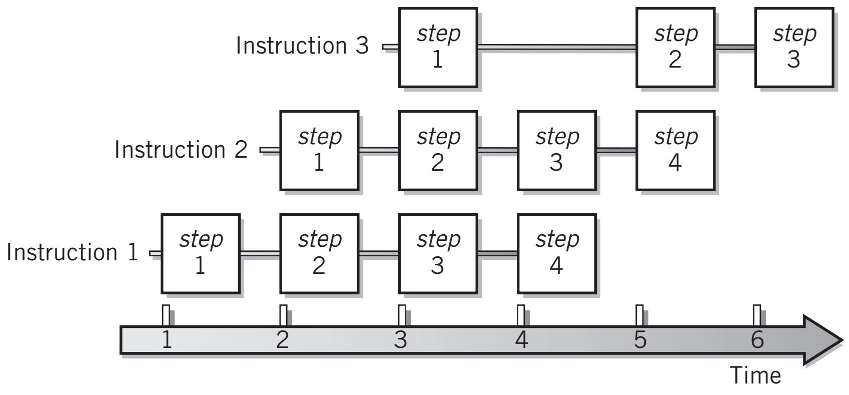
Of course, a branch instruction may invalidate all the instructions in the pipeline at that instant if the branch is taken, and the computer still must have the data from the previous instruction if the next instruction requires it in order to proceed. Modern computers use a variety of techniques to compensate for the branching problem. One common approach is to maintain two or more separate pipelines so that instructions from both possible outcomes can be processed until the direction of the branch is clear. Another approach attempts to predict the probable branch path based on the history of previous execution of the same instruction. The problem of waiting for data results from previous instructions can be alleviated by separating the instructions so that they are not executed one right after the other. Many modern computer designs contain logic that can reorder instructions as they are executed to keep the pipelines full and to minimize situations where a delay is necessary. **Instruction reordering** also makes it possible to provide parallel pipelines, with duplicate CPU logic, so that multiple instructions can actually be executed simultaneously. This technique is equivalent to providing multiple car assembly lines. It is known as superscalar processing. We will look at superscalar processing again in the next section.

Pipelining and instruction reordering complicate the electronic circuitry required for the computer and also require careful design to eliminate the possibility of errors occurring under unusual sequences of instructions. (Remember that the programmer must always be able to assume that instructions are executed in the specified order.) Despite the added complexity, these methods are now generally accepted as a means for meeting the demand for more and more computer power. The additional task of analyzing, managing, and steering instructions to the proper execution unit at the proper time is usually combined with instruction fetching and decoding to form a single **instruction unit** that handles all preparation of instructions for execution.

A diagram illustrating pipelining is shown in Figure 8.5. For simplicity, instruction reordering has not been included. The figure shows three instructions, one for each row in the diagram. The "steps" in the diagram represent the sequence of steps in the fetch-execute

FIGURE 8.5

Pipelining



cycle for each instruction. Timing marks are indicated along the horizontal axis. The F-E cycle for instruction 3 shows a delay between step 1 and step 2; such a delay might result because the second step of the instruction needs a result from step 3 of the previous instruction, for example, the data in a particular register.

MULTIPLE, PARALLEL EXECUTION UNITS It is not useful to pipe different types of instructions through a single pipeline. Different instructions have different numbers of steps in their cycles and, also, there are differences in each step. Instead, the instruction decode unit steers instructions into specific execution units. Each execution unit provides a pipeline that is optimized for one general type of instruction. Typically, a modern CPU will have a LOAD/STORE unit, an integer arithmetic unit, a floating point arithmetic unit, and a branch unit. More powerful CPUs may have multiple execution units for the more commonly used instruction types and, perhaps, may provide other types of execution units as well. Again, an analogy may aid in understanding the concept of multiple, parallel execution units. A simple automobile plant analogy would note that most automobile plants have separate assembly lines for different car models. The most popular models might have multiple assembly lines operating in parallel.

The use of multiple execution units operating in parallel makes it possible to perform the actual execution of several instructions simultaneously.

Scalar and Superscalar Processor Organization

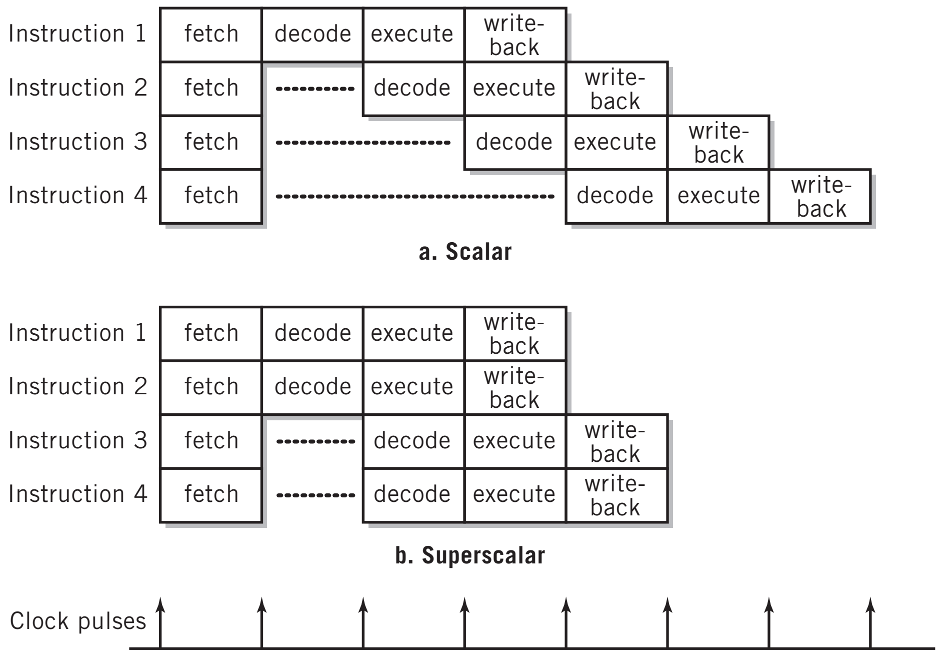
The previous discussion has shown you that modern CPUs achieve high performance by separating the two major phases of the fetch-execute cycle into separate components, then further separating the execution phase into a number of independent execution units, each with pipeline capability. Once a pipeline is filled, an execution unit can complete an instruction with each clock tick. With a single execution unit pipeline, ignoring holes in the pipeline resulting from different instruction types and branch conditions, the CPU can average instruction execution approximately equal to the clock speed of the machine. A processor fulfilling this condition is called a **scalar processor**. With multiple execution units it is possible to process instructions in parallel, with an average rate of more than one instruction per clock cycle. The ability to process more than one instruction per clock cycle is known as **superscalar processing**. Superscalar processing is a standard feature in modern CPUs. Superscalar processing can increase the throughput by double or more. Commonly, current CPU designs produce speed increases of between two and five times.

It is important to remember that pipelining and superscalar processing techniques do not affect the cycle time of any individual instruction. An instruction fetch-execute cycle that requires six clock cycles from start to finish will require six clock cycles whether instructions are performed one at a time or pipelined in parallel with a dozen other instructions. It is the average instruction cycle time that is improved by performing some form of parallel execution. If an individual instruction must be completed for any reason before another can be executed, the CPU must stall for the full cycle time of the first instruction.

Figure 8.6 illustrates the difference between scalar and superscalar processing with pipelining in the execution unit. In the illustration the execution phase of the fetch-execute

FIGURE 8.6

Scalar versus Superscalar Processing



cycle is divided into three parts that can be executed separately. Thus, the diagram is divided into steps that fetch, decode, execute, and write back the results of the execute operation. Presumably, each step is carried out by a separate component within the execution unit. To simplify the illustration, we have also assumed that in each case the pipeline is full. Generally, a single fetch unit pipeline is sufficient to fetch multiple instructions, even when multiple execution units are present.

In the scalar processor, Figure 8.6a, each step is assumed to take one clock cycle. If the instructions are all of the same length, they will finish consecutively, as shown in the diagram. More complexity in the instruction set will create bubbles in the pipeline, but does not alter the basic idea that we are illustrating. Panel b of the figure assumes the presence of two execution units. It also assumes that the instructions executing in parallel are independent of each other; that is, the execution of one does not depend upon results from the other. Therefore, two instructions can be executed at a time in parallel, resulting in a substantial improvement in overall instruction completion performance.

Superscalar processing complicates the design of a CPU considerably. There are a number of difficult technical issues that must be resolved to make it possible to execute multiple instructions simultaneously. The most important of these are

- Problems that arise from instructions completing in the wrong order
- Changes in program flow due to branch instructions
- Conflicts for internal CPU resources, particularly general-purpose registers

OUT-OF-ORDER PROCESSING Out-of-order instruction execution can cause problems because a later instruction may depend on the results from an earlier instruction. This situation is known as a **hazard** or a *dependency*. If the later instruction completes ahead of the earlier one, the effect of the earlier instruction upon the later cannot be satisfied. The most common type of a dependency is a **data dependency**. This is a situation in which the later instruction is supposed to use the results from the earlier instruction in its calculation. There are other types of dependencies also.

With multiple execution units, it is possible for instructions to complete in the wrong order. There are a number of ways in which this can occur. In the simplest case, an instruction with many steps in its cycle may finish after an instruction with just a few steps, even if it started earlier. As a simple example, a MULTIPLY instruction takes longer to execute than a MOVE or ADD instruction. If a MULTIPLY instruction is followed in the program by an ADD instruction that adds a constant to the results of the multiplication, the result will be incorrect if the ADD instruction is allowed to complete ahead of the MULTIPLY instruction. This is an example of data dependency. Data dependency can take several different forms.

Many data dependencies are sufficiently obvious that they can be detected by the CPU. In this case, execution of the dependent instruction is suspended until the results of the earlier instruction are available. This suspension may, itself, cause out-of-order execution, since it may allow another, still later, instruction to complete ahead of the suspended instruction. Some CPUs provide reservation stations within each execution unit or a general instruction pool to hold suspended instructions so that the execution unit may continue processing other instructions.

Finally, some systems intentionally allow out-of-order instruction execution. These CPUs can actually search ahead for instructions without apparent dependencies, to keep the execution units busy. Current Intel x86 CPUs, for example, can search twenty to thirty instructions ahead, if necessary, to find instructions available for execution.

BRANCH INSTRUCTION PROCESSING Branch instructions must always be processed ahead of subsequent instructions, since the addresses of the proper subsequent instructions to fetch are determined from the branch instruction. For unconditional branch instructions, this is simple. Branch instructions are identified immediately as they enter the instruction fetch pipeline. The address in the instruction is decoded and used to fill the instruction fetch pipeline with instructions from the new location. Normally, no delay is incurred.

Unfortunately, conditional branch instructions are more difficult, because the condition decision may depend on the results from instructions that have not yet been executed. These situations are known as *flow* or *branch dependencies*. If the wrong branch is in the pipeline, the pipeline must be flushed and refilled, wasting time. Worse yet, an instruction from the wrong branch, that is, one that should not have been executed, can alter a previous result that is still needed.

The solution to the conditional branching problem may be broken into two parts: methods to optimize correct branch selection and methods to prevent errors as a result of conditional branch instructions. Selection of the wrong branch is time wasting, but not fatal. By contrast, incorrect results *must* be prevented.

Errors are prevented by setting the following guideline: although instructions may be executed out of order, they must be completed in the correct order. Since branches and subtle data dependencies can occur, the execution of an instruction out of order may or may not be valid, so the instruction is executed *speculatively*, that is, on the assumption that its execution will be useful. For this purpose, a separate bank of registers is used to hold results from these instructions until previous instructions are complete. The results are then transferred to their actual register and memory locations, in correct program instruction order. This technique of processing is known as **speculative execution**. On occasion, the results from some speculatively executed instructions must be thrown away, but on the whole, speculative execution results in a performance boost sufficient to justify the extra complexity required.

A few systems place the burden for error prevention on the assembly language programmer or program language compiler by requiring that a certain number of instructions following a conditional branch instruction be independent of the branch. In these systems, one or more instructions sequentially following the branch are *always* executed, regardless of the outcome of the branch.

There are various creative methods that are used in CPUs to optimize conditional branch processing. One possible solution to this problem is to maintain two separate instruction fetch pipelines, one for each possible branch outcome. Instructions may be executed speculatively from *both* branches until the correct pipeline is known. Another solution is to have the CPU attempt to predict the correct path based on program usage or past performance. A loop, for example, may be expected to execute many times before exiting. Therefore, the CPU might assume that a branch to a previous point in the program is usually taken. Some systems provide a **branch history table**, a small amount of dedicated memory built into the CPU that maintains a record of previous choices for each of several branch instructions that have been used in the program being executed to aid in prediction. A few systems even include a ''hint'' bit in the branch instruction word that can be set by the programmer to tell the CPU the more probable outcome of the branch. Of course, when a branch prediction is incorrect, there is a time delay to purge and refill the fetch pipeline and speculative instructions, but, overall, branch prediction is effective.

CONFLICT OF RESOURCES Conflicts between instructions that use the same registers can be prevented by using the same bank of registers that is used to hold the results of speculative instructions until instruction completion. This register bank is given different names by different vendors. They are called variously **rename registers** or **logical registers** or **register alias tables**. The registers in the bank can be renamed to correspond logically to any physical register and assigned to any execution unit. This would allow two instructions using the ''same'' register to execute simultaneously without holding up each other's work At completion of an instruction, the CPU then selects the corresponding physical register and copies the result into it. This must occur in the specified program instruction order.

8.3 MEMORY ENHANCEMENTS

Within the instruction fetch-execute cycle, the slowest steps are those that require memory access. Therefore, any improvement in memory access can have a major impact on program processing speed.

The memory in modern computers is usually made up of dynamic random access memory circuit chips. DRAM is inexpensive. Each DRAM chip is capable of storing millions of bits of data. Dynamic RAM has one major drawback, however. With today's fast CPUs, the access time of DRAM is too slow to keep up with the CPU, and delays must be inserted into the LOAD/STORE execution pipeline to allow memory to keep up. Thus, the use of DRAM is a potential bottleneck in processing. Instructions must be fetched from memory and data must be moved from memory into registers for processing.

The fetch-execute CPU implementation introduced in Section 8.2 reduces instruction fetch delay to a minimum with modern instruction prefetch and branch control technologies, and the increased adoption of register-to-register instructions also reduces delays. Nonetheless, memory accesses are always required ultimately to move the data from memory to register and back, and improvements in memory access still have an impact on processing speed.

As mentioned in Chapter 7, static RAM, or SRAM, is an alternative type of random access memory that is two to three times as fast as DRAM. The inherent memory capacity of SRAM is severely limited, however. SRAM design requires a lot of chip real estate compared to DRAM, due to the fact that SRAM circuitry is more complex and generates a lot of heat that must be dissipated. One or two MB of SRAM requires more space than 64 MB of DRAM, and will cost considerably more.

With today's memory requirements, SRAM is not a practical solution for large amounts of memory except in very expensive computers; therefore, designers have created alternative approaches to fulfill the need for faster memory access. Three different approaches are commonly used to enhance the performance of memory:

- Wide path memory access
- Memory interleaving
- Cache memory

These three methods are complementary. Each has slightly different applicability, and they may be used together in any combination to achieve a particular goal. Of these techniques, the use of cache memory has the most profound effect on system performance.

Wide Path Memory Access

As mentioned in Chapter 7, Section 7.3, the simplest means to increase memory access is to widen the data path so as to read or write several bytes or words between the CPU and memory with each access; this technique is known as **wide path memory access**. Instead of reading 1 byte at a time, for example, the system can retrieve 2, 4, 8, or even 16 bytes, simultaneously. Most instructions are several bytes long, in any case, and most data is at least 2 bytes, and frequently more. This solution can be implemented easily by widening the bus data path and using a larger memory data register. The system bus on most modern CPUs, for example, has a 64-bit data path and is commonly used to read or write 8 bytes of data with a single memory access.

Within the CPU, these bytes can be separated as required and processed in the usual way. With modern CPU implementation, instruction groups can be passed directly to the instruction unit for parallel execution. As the number of bytes simultaneously accessed is increased, there is a diminishing rate of return, since the circuitry required to separate and direct the bytes to their correct locations increases in complexity, fast memory access becomes more difficult, and yet it becomes less likely that the extra bytes will actually be used. Even a 64-bit data path is adequate to assure that a pipeline will remain filled and bursts of consecutive 64-bit reads or writes can handle situations that require high-speed access to large blocks of data. Very few systems read and write more than 8 bytes at a time. Most systems read and write a fixed number of bytes at a time, but there are a few systems that can actually read and write a variable number of bytes.

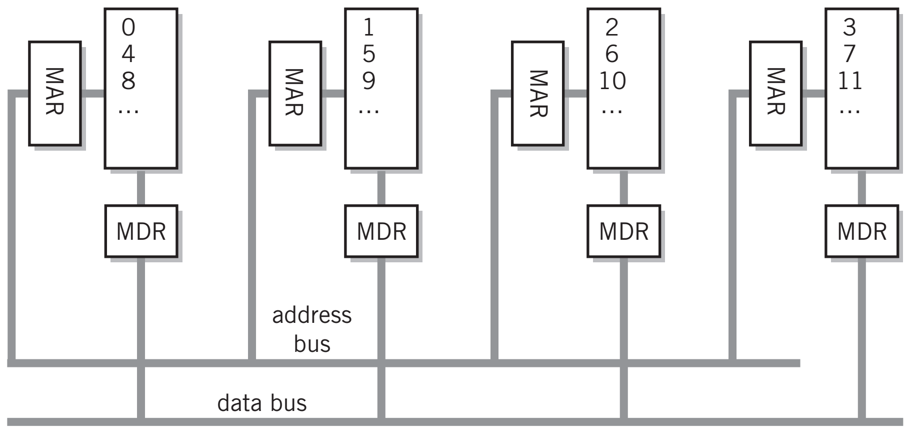
Modern computers are commonly built with standard, off-the-shelf memory circuits and chips that include wide path memory access as a standard feature.

Memory Interleaving

Another method for increasing the effective rate of memory access is to divide memory into parts, called **memory interleaving**, so that it is possible to access more than one location at a time. Then, each part would have its own address register and data register, and each part is independently accessible. Memory can then accept one read/write request from each part simultaneously. Although it might seem to you that the obvious way to divide up memory would be in blocks, for example, by separating the high addresses into one block and the low addresses into the other, it turns out that as a practical matter it is usually more useful to divide the memory so that successive access points, say, groups of 8 bytes (see above), are in different blocks. Breaking memory up this way is known as ***n*-way interleaving**, where a value of 2 or 4 or some other value is substituted for *n*, depending on the number of separate blocks. For example, two-way interleaving would be designed so that it would be possible to access an odd memory address and an even memory address concurrently. If 8-byte wide access is provided, this would allow the concurrent access to 16 successive bytes at a time. A memory with eight-way interleaving would allow access to eight different locations simultaneously, but the system could not access locations 0, 8, 16, or 24 at the same time, for instance, nor 1, 9, 17, or 25. It *could* access locations 16 and 25 or 30 and 31 concurrently, however. Since memory accesses tend to be successive, memory interleaving can be effective. A diagram of four-way interleaving is shown in Figure 8.7.

FIGURE 8.7

Four-Way Memory Interleaving



This method is particularly applicable when multiple devices require access to the same memory. The IBM mainframe architecture, for example, is designed to allow multiple CPUs to access a common memory area; the I/O channel subsystem also has access to the storage area. Thus, several different components may make memory requests at the same time. The IBM S/3033 computer, for example, partitioned memory into eight **logical storage elements**. Each element can independently accept a memory request. Thus, eight memory requests can be processed concurrently.

The personal computer memory that holds images while they are being displayed, known as video RAM, is another example. Changes to part of the video RAM can be made at the same time that another part of the video RAM is being used to produce the actual display on the monitor.

Cache Memory

A different strategy is to position a small amount of high-speed memory, for example, SRAM, between the CPU and main storage. This high-speed memory is invisible to the programmer and cannot be directly addressed in the usual way by the CPU. Because it represents a "secret" storage area, it is called **cache memory**. This concept is illustrated in Figure 8.8.

Cache memory is organized differently than regular memory. Cache memory is organized into blocks. Each block provides a small amount of storage, perhaps between 8 and 64 bytes, also known as a **cache line**. The block will be used to hold an exact reproduction of a corresponding amount of storage from somewhere in main memory. Each block also holds a **tag**. The tag identifies the location in main memory that corresponds to the data being held in that block. In other words, taken together, the tags act as a directory that can be used to determine exactly which storage locations from main memory are also available in the cache memory. A typical 64 KB cache memory might consist of 8000 (actually 8192) 8-byte blocks, each with tag.

A simplified, step-by-step illustration of the use of cache memory is shown in Figure 8.9. Every CPU request to main memory, whether data or instruction, is seen first by cache memory. A hardware **cache controller** checks the tags to determine if the memory location of the request is presently stored within the cache. If it is, the cache memory is used as if it were main memory. If the request is a read, the corresponding word from cache memory is simply passed to the CPU. Similarly, if the request is a write, the data from the CPU is stored in the appropriate cache memory location. Satisfying a request in this way is known as a **hit**.

If the required memory data is not already present in cache memory, an extra step is required. In this case, a cache line that includes the required location is copied from memory to the cache. Once this is done, the transfer is made to or from cache memory, as before. The situation in which the request is not already present in cache memory is known as a **miss**. The ratio of hits to the total number of requests is known as the **hit ratio**.

When cache memory is full, some block in cache memory must be selected for replacement. Various algorithms

FIGURE 8.8

Cache Memory

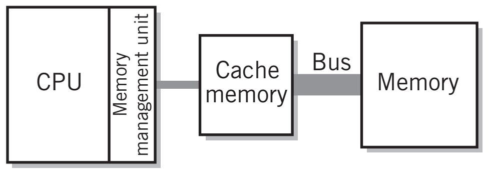
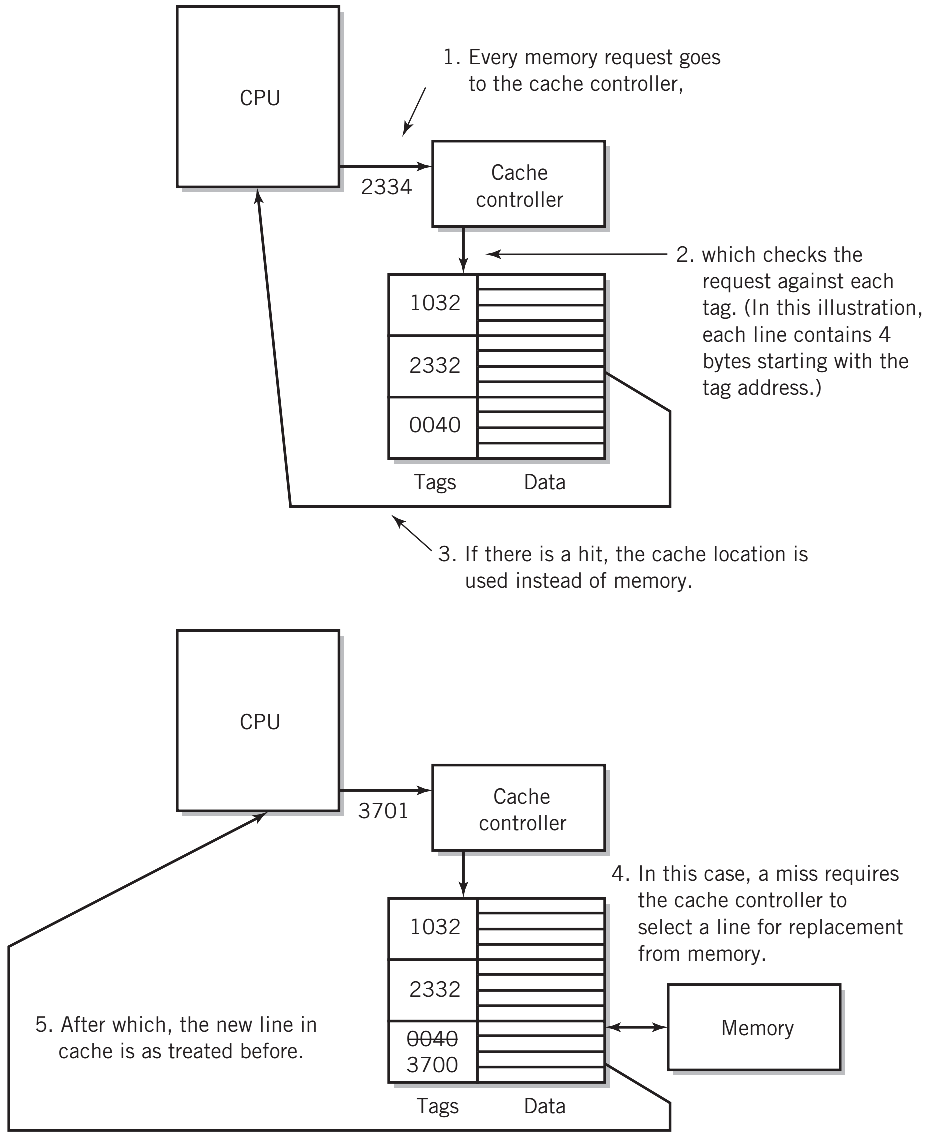


FIGURE 8.9

Step-by-Step Use of Cache


CPU
1. Every memory request goes to the cache controller,
2334
Cache controller
2. which checks the request against each tag. (In this illustration, each line contains 4 bytes starting with the tag address.)
1032
2332
0040
Tags
Data
3. If there is a hit, the cache location is used instead of memory.
CPU
3701
Cache controller
4. In this case, a miss requires the cache controller to select a line for replacement from memory.
1032
2332
~~0040~~
3700
Memory
5. After which, the new line in cache is as treated before.
Tags
Data

have been implemented by different computer designers to make this selection, but most commonly, some variation on a *least recently used*, or *LRU*, algorithm is used. An LRU algorithm, as the name implies, keeps track of the usage of each block and replaces the block that was last used the longest time ago.

Cache blocks that have been read, but not altered, can simply be read over during replacement. Memory write requests impose an additional burden on cache memory operations, since written data must also be written to the main memory to protect the integrity of the program and its data. Two different methods of handling the process of returning changed data from cache to main storage are in common use. The first method, **write through**, writes data back to the main memory immediately upon change in the cache. This method has the advantage that the two copies, cache and main memory, are always kept identical. Some designers use an alternative technique known variously as *store in, write back*, or *copy back*. With this technique, the changed data is simply held in cache until the cache line is to be replaced. The **write back** method is faster, since writes to memory are made only when a cache line is actually replaced, but more care is required in the design to ensure that there are no circumstances under which data loss could occur. If two different programs were using the same data in separate cache blocks, for example, and one program changed the data, the design must assure that the other program has access to the updated data.

The entire cache operation is managed by the cache controller. This includes tag searching and matching, write through or write back, and implementation of the algorithm that is used for cache block replacement. The CPU and software are unaware of the presence of cache memory and the activities of the cache controller. We note in passing that to be effective, these operations must be controlled completely by hardware. It is possible to envision using a program to implement the cache block replacement algorithm, for example, but this is not feasible. Since memory accesses would be required to execute the program, this would defeat the entire purpose of cache memory, which is to provide access quickly to a single memory location.

Cache memory works due to a principle known as **locality of reference**. The locality of reference principle states that at any given time, most memory references will be confined to one or a few small regions of memory. If you consider the way that you were taught to write programs, this principle makes sense. Instructions are normally executed sequentially; therefore, adjoining words are likely to be accessed. In a well-written program, most of the instructions being executed at a particular time are part of a small loop or a small procedure or function. Likewise, the data for the program is likely taken from an array. Variables for the program are all stored together. Studies have verified the validity of the locality principle. Cache memory hit ratios of 90 percent and above are common with just a small amount of cache. Since requests that can be fulfilled by the cache memory are fulfilled much faster, the cache memory technique can have a significant impact on the overall performance of the system. Program execution speed improvements of 50 percent and more are common.

The hit ratio is an important measure of system performance. Cache hits can access memory data at or near the speed that instructions are executed, even with sophisticated instruction steering and multiple execution units. When a miss occurs, however, there is a time delay while new data is moved to the cache. The time to move data to the cache is called **stall time**. The stall time is typically long compared to instruction execution time. This can result in a condition in which there are no instructions available to feed to the execution units; the pipelines empty and instruction execution is stalled until the needed cache line is available, reducing performance.

Some modern architectures even provide program instructions to request cache preloading for data or instructions that will be needed soon. This improves execution speed even more. Also, some system designers interleave the cache or implement separate caches for instructions and data. This allows even more rapid access, since the instruction and its operands can be accessed simultaneously much of the time. Furthermore, design of a separate instruction cache can be simplified, since there is no need to write the instruction cache back to main memory if the architecture imposes a pure coding requirement on the programmer. The trade-off is that accommodating separate instruction and data caches requires additional circuit complexity, and many system designers opt instead for a combined, or *unified*, cache that holds both data and instructions.

It is also possible to provide more than one level of cache memory. Consider the two level cache memory shown in Figure 8.10. This memory will work as follows. The operation begins when the CPU requests an instruction (or piece of data) be read (or written) from memory. If the cache controller for the level closest to the CPU, which we'll call level 1 (normally abbreviated as L1), determines that the requested memory location is presently in the level 1 cache, the instruction is immediately read into the CPU.

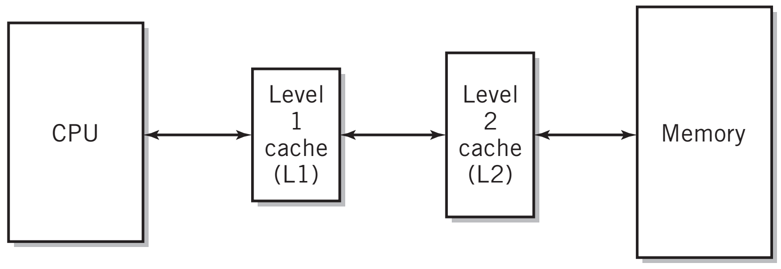
Suppose, however, that the instruction is *not* presently in level 1 cache. In this case, the request is passed on to the controller for level 2 cache. Level 2 cache works in exactly the same way as level 1 cache. If the instruction is presently in the level 2 cache, a cache line containing the instruction is moved to the level 1 cache and then to the CPU. If not, then the level 2 cache controller requests a level 2 cache line from memory, the level 1 cache receives a cache line from the level 2 cache, and the instruction is transferred to the CPU. This technique can be extended to more levels, but there is usually little advantage in expanding beyond level 3.

What does the second level buy us? Most system designers believe that more cache would improve performance enough to be worthwhile. In this case, the system designers provide a second level of cache, external to the chip. A personal computer secondary cache commonly provides an additional 512 KB–2 MB of cache. A typical AMD Athlon 64 processor provides 64 KB of L1 data cache, 64 KB of L1 instruction cache, and 512 KB or 1 MB of level 2 cache within the same package as the CPU. The use of a dedicated on-chip bus between level 1 cache and level 2 cache provides faster response than connecting the level 1 cache to memory or to a level 2 cache on the regular memory bus.

To be useful, the second level of cache must have significantly more memory than the first level; otherwise, the two cache levels would contain the same data, and the secondary cache would serve no purpose. It is also normal to provide a larger cache line in the secondary cache. This increases the likelihood that requests to the secondary cache can be met without going out to main memory every time.

FIGURE 8.10

Two-Level Cache



Before leaving the subject of memory caching, a side note: the concept of caching also shows up in other, unrelated but useful, areas of computer system design. For example, caching is used to reduce the time necessary to access data from a disk. In this case, part of main memory can be allocated for use as a **disk cache**. When a disk read or write request is made, the system checks the disk cache first. If the required data is present, no disk access is necessary; otherwise, a disk cache line made up of several adjoining disk blocks is moved from the disk into the disk cache area of memory. Most disk manufacturers now provide separate buffer memory for this purpose. This feature is implemented within the hardware of a disk controller. Another example is the cache of previous Web pages provided by Web browser application software.

All of these examples of caching share the common attribute that they increase performance by providing faster access to data, anticipating its potential need in advance, then storing that data temporarily where it is rapidly available.

8.4 THE COMPLEAT MODERN SUPERSCALAR CPU

Figure 8.11 is a model of a CPU block diagram that includes all the ideas just discussed. The design shown in this diagram is very similar to the one used in Sun SPARC and IBM Power and PowerPC processors and, with minor variations, to that used in various generations of the Intel Pentium and Itanium families, as well as various IBM mainframe processors. As you would expect, the CPU is organized into modules that reflect the superscalar, pipelined nature of the architecture. Although it is difficult to identify the familiar components that we introduced in Chapter 7, the control unit, arithmetic/logic unit, program counter, and the like, they are indeed embedded into the design, as you saw in Figure 8.4. The control unit operation is distributed through much of the diagram, controlling each step of the usual fetch-execute cycle as instructions flow through different blocks in the CPU. The functions of the arithmetic/logic unit are found within the integer unit. The program counter is part of the instruction unit.

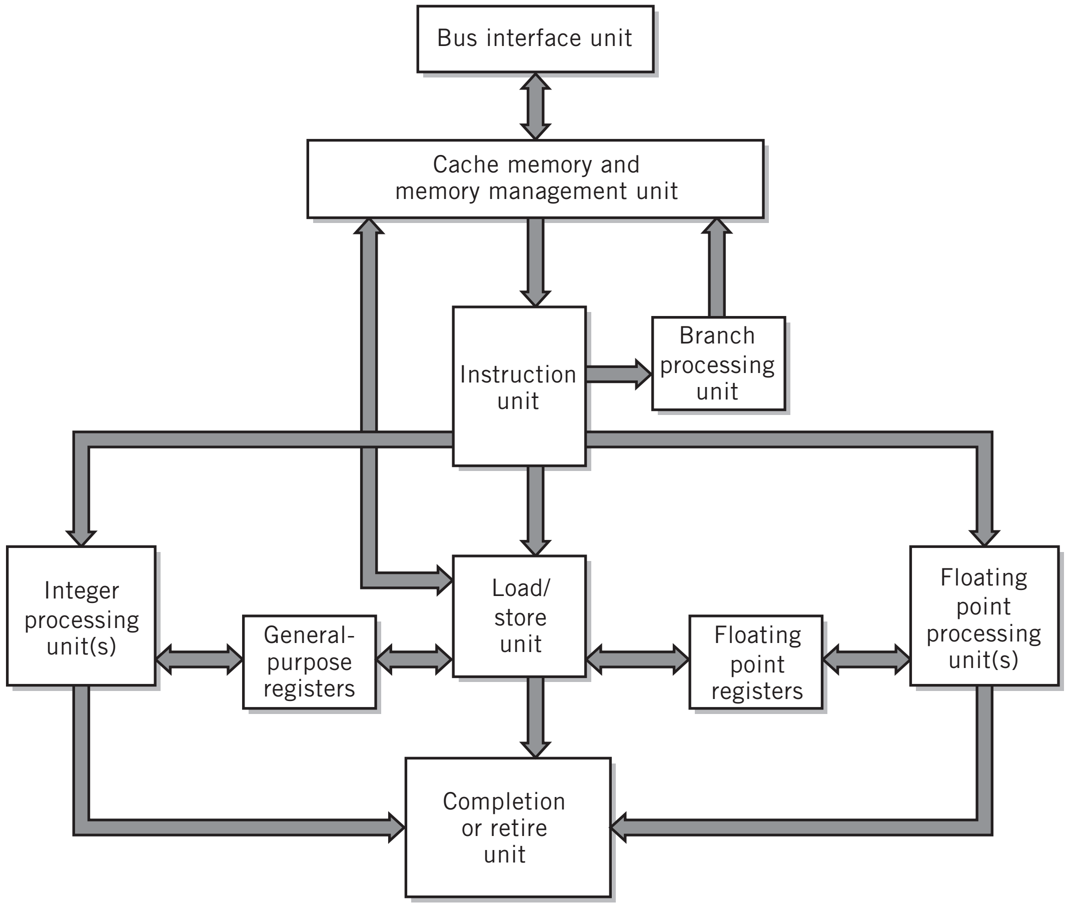
In operation, instructions are fetched from memory by the memory management unit as they are needed for execution, and placed into a pipeline within the instruction unit. The instructions are also partially decoded within the instruction unit, to determine the type of instruction that is being executed. This allows branch instructions to be passed quickly to the branch processing unit for analysis of future instruction flow.

Instructions are actually executed in one of several types of execution units. Each execution unit has a pipeline designed to optimize the steps of the execute cycle for a particular type of instruction.

As you can see from the block diagram, there are separate execution units for branch instructions, for integer instructions, for floating point instructions, and for load and store instructions. Some processors provide multiple integer execution units to increase the processing capacity of the CPU still further. Some models also have a separate system register unit for executing system-level instructions. Some CPUs combine the load/store instructions into the integer unit. The PowerPC provides reservation stations in each execution unit. The Intel Pentium processors provide a general instruction pool where decoded instructions from the instruction unit are held as they await operand data from memory and from unresolved data dependencies. The Pentium instruction pool also holds

FIGURE 8.11

Modern CPU Block Diagram


Bus interface unit
Cache memory and memory management unit
Instruction unit
Branch processing unit
Integer processing unit(s)
General-purpose registers
Load/store unit
Floating point registers
Floating point processing unit(s)
Completion or retire unit

completed instructions after execution until they can be retired in order. The Pentium also separates the LOAD and STORE execution units.

The instruction unit is responsible for maintaining the fetch pipeline and for dispatching instructions. Because branch instructions affect the addresses of the following instructions in the pipeline, they are processed immediately. Other instructions are processed as space becomes available in the appropriate execution unit(s). Branch prediction is usually built into the branch unit. When conditional branches occur, execution of instructions continues speculatively along the predicted branch until the condition is resolved. Also, the use of multiple execution units makes it possible that instructions will execute in the wrong order, since some instructions may have to wait for operands resulting from other instructions and since the pipelines in each execution unit are of different lengths. As we noted earlier, some current superscalar processors can look ahead several instructions to find instructions that can be processed independently of the program order to prevent delays or errors arising from data dependency. The ability to process instructions out of order is an important factor in the effectiveness of these processors. The completion or ‘‘retire’’ unit accepts or rejects speculative instructions, stores results in the appropriate physical registers and cache memory locations, and retires instructions in the correct program order, assuring correct program flow. The Crusoe and Itanium architectures prevent out-of-order retirement by reordering the instructions prior to execution. This simplifies the CPU design.

From this discussion, you can see that the modern CPU includes many sophisticated features designed to streamline the basically simple fetch-execute cycle for high-performance processing. The modern CPU features different types of execution units, tailored to the needs of different types of instructions, and a complex steering system that can steer instructions through the instruction unit to available execution units, manage operands, and retire instructions in correct program order. The goal of each of these techniques is to increase the parallelism of instruction execution, while maintaining the basic sequential characteristic of a von Neumann computer architecture.

As a brief diversion, consider the similarities between the operation of the modern CPU and the operation of a moderately large restaurant. Each of the waiters and waitresses taking orders represent the fetch unit fetching instructions. Customers’ orders are fed to the kitchen, where they are sorted into categories: soup orders to the soup chef, salads to the salad chef, entrées to the entrée chef, and so on. Typically, the entrée chef will have the most complex orders to fill, equivalent to the longest pipeline in the CPU. If the kitchen is large, the entrée area will be further subdivided into multiple execution areas: frying, baking, and so on, and there may be multiple cooks working in the busiest areas. As with the programs being executed in a computer, there are dependencies between the various cooks. For example, green beans must be blanched before they may be placed in the salad. Finally, we observe that, like computer program instructions, the restaurant must provide food from the kitchen to the customers in the proper sequence, and with appropriate timing, to satisfy the customers’ requirements.

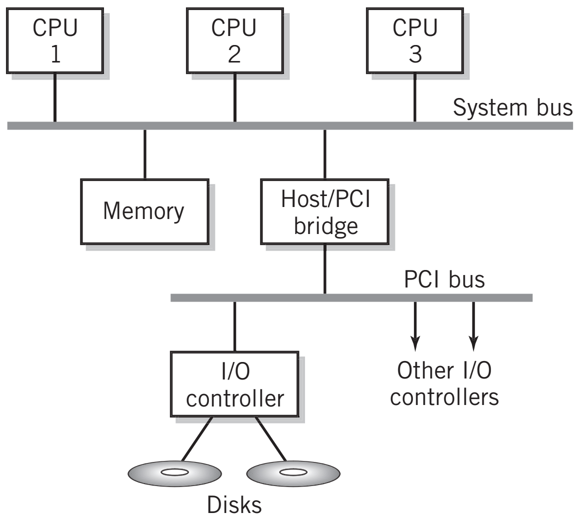
In this section we have introduced the basic ideas of superscalar processing, briefly indicated the difficulties, and explained the reasoning for its use. There are many excellent references listed in the For Further Reading if you are interested in more of the details of superscalar processing and modern CPU design.

8.5 MULTIPROCESSING

One obvious way to increase performance in a computer system is to increase the number of CPUs. Computers that have multiple CPUs within a single computer, sharing some or all of the system’s memory and I/O facilities, are called **multiprocessor systems**, or sometimes **tightly coupled systems**. When multiple CPU processors are supplied within a single integrated circuit, they are more commonly called **multicore processors**. Figure 8.12 shows a typical multiprocessor configuration. All the processors in a multiprocessor configuration have access to the same programs and data in shared memory and to the same I/O devices, so it is possible to divide program execution between different CPUs. Furthermore, programs or pieces of programs may be run in any CPU that is available, so that each additional processor extends the power available for multitasking in a multiprocessing system, at least within the capability of the shared components, the memory, buses, and I/O controllers.

FIGURE 8.12

Typical Multiprocessing System Configuration



Under ideal conditions, each CPU processes its own assigned sequence of program instructions independently. Thus, a dual-core processor effectively doubles the number of instructions executed in a given time, a quad-core processor would quadruple the rate, and so forth. Of course this assumes that there are multiple independent tasks available to be executed simultaneously. Since modern computer systems are normally executing many programs and segments of programs concurrently this is nearly always the case.

In practice, increasing the number of CPUs is, in fact, usually effective, although, as the number of CPUs increases, the value of the additional CPUs diminishes because of the overhead required to distribute the instructions in a useful way among the different CPUs and the conflicts among the CPUs for shared resources, such as memory, I/O, and access to the shared buses. With the exception of certain, specialized systems, there are rarely more than sixteen CPUs sharing the workload in a multiprocessing computer; more commonly today, a multiprocessor might consist of two, four, or eight core CPUs within a single chip. Still, each core in the chip is a full-blown superscalar CPU, of the type discussed in the previous sections of this chapter.

Although increased computing power is a significant motivation for multiprocessing, there are other considerations that make multiprocessing attractive:

- Since the execution speed of a CPU is directly related to the clock speed of the CPU, equivalent processing power can be achieved at much lower clock speeds, reducing power consumption, heat, and stress within the various computer components.
- Programs can be divided into independent pieces, and the different parts executed simultaneously on multiple CPUs.
- With multiprocessing, increasing computational power may be achieved by adding more CPUs, which is relatively inexpensive.
- Data dependencies and cache memory misses can stall the pipelines in a single CPU. Multiprocessing allows the computer to continue instruction execution in the other CPUs, increasing overall throughput.

Assignment of work to the various processors is the responsibility of the operating system. Work is assigned from among the programs available to be executed, or, more commonly, from independent segments of those programs called **threads**. Since each of the CPUs has access to the same memory and I/O, any CPU can theoretically execute any thread or program currently in memory, including the operating system. This raises the question of control of the system. There are two basic ways of configuring a multiprocessing system:

- **Master-slave multiprocessing**, in which one CPU, the *master*, manages the system, and controls all resources and scheduling. Only the master may execute the operating system. Other CPUs are *slaves*, performing work assigned to them by the master.
- **Symmetrical multiprocessing (SMP)**, in which each CPU has identical access to the operating system, and to all system resources, including memory. Each CPU schedules it own work, within parameters, constraints, and priorities set by the operating system. In a normal SMP configuration, each of the CPUs is identical.

A number of CPUs also implement a simplified, limited form of multiprocessing using parallel execution units within a single CPU to process two or more threads simultaneously. This technique is called **simultaneous thread multiprocessing (STM)**. STM is also known as *hyperthreading*. STM is particularly useful in dealing with cache stalls, because the CPU can be kept busy working on the alternative thread or threads. The operating system manages STM in a manner similar to SMP. Since STM operates within a single CPU and SMP operates between CPUs, STM and SMP can be used together.

For general purpose computing, the symmetrical configuration has many advantages. Because every CPU is equal, every CPU has equal access to the operating system. Any CPU can execute any task and can process any interrupt.[4] Processors are all kept equally busy, since each processor can dispatch its own work as it needs it. Thus, the workload is well balanced. It is easy to implement fault-tolerant computing with a symmetrical configuration—critical operations are simply dispatched to all CPUs simultaneously. Furthermore, a failure in a single CPU may reduce overall system performance, but it will not cause system failure. As an interesting aside, note that a program may execute on a different CPU each time it is dispatched, although most SMP systems provide a means to *lock* a program onto a particular CPU, if desired. Thus, the symmetrical configuration offers important capabilities for multiprocessing: maximum utilization of each CPU, flexibility, high reliability, and optional support for fault-tolerant computing. Most modern general purpose multiprocessing systems are SMP systems.

Because of the somewhat limited flexibility in distributing the workload, the master-slave configuration is usually considered less suitable for general purpose computing. In a master-slave configuration the master is likely to be the busiest CPU in the system. If a slave requires a work assignment while the master is busy, the slave will have to wait until the master is free. Furthermore, since the master handles all I/O requests and interrupts, a heavily loaded system will cause a backload in the master. If slaves are dependent on the results of these requests, the system is effectively stalled.

Conversely, there are a number of specialized computing applications for which the master-slave configuration is particularly well suited. These applications are characterized by a need for a master control program, supported by repetitive or continuous, computation- and data-intensive, time-critical tasks. For example, the processor in a game controller

[4] Interrupts are a special feature of the CPU in which outside events such as mouse movements and power interruptions can affect the sequence of instructions processed by the CPU. Interrupts are discussed in detail in Chapter 9.

must execute the code that plays the game. At the same time, it requires support that can rapidly calculate and display new images based on movements of the objects in the image, compute shadings and light reflections resulting from the movements; often, the processor must create new pixel values for every pixel in the image. It must also be able to create the appropriate reactions and display in response to events that occur, such as explosions or fires or objects bouncing off a wall, and more.

Many important applications in economics, biology, physics, and finance, particularly those based on simulation and modeling, have similar requirements.

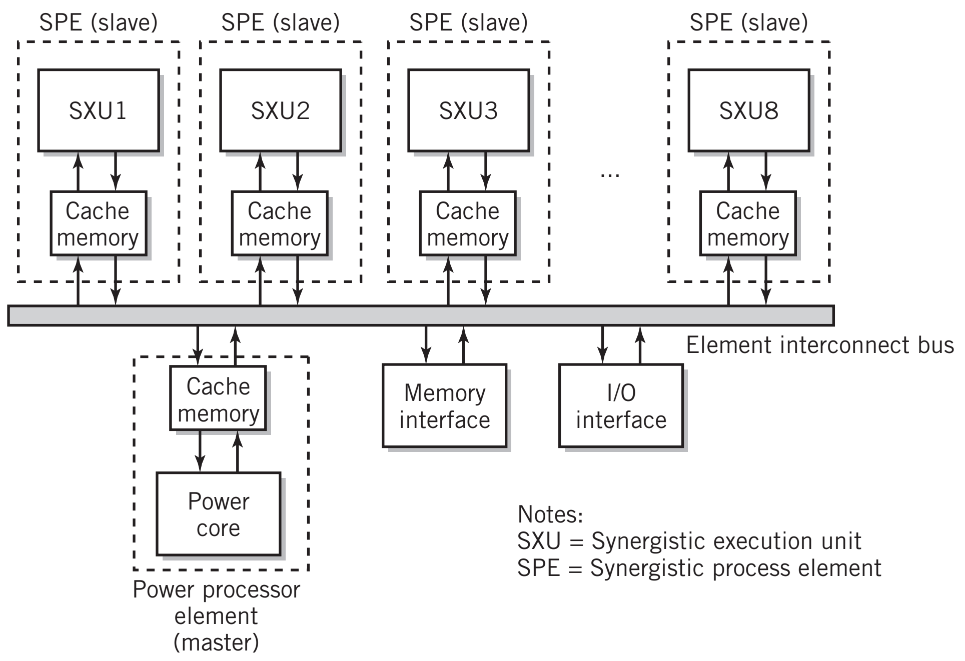
EXAMPLE

The recently developed Cell Broadband Engine processor is organized in a master-slave configuration. It was developed jointly by IBM, Sony, and Toshiba as the first of a new generation of processors intended for use in high-performance intensive computing applications. It is the main processor used in the Sony PlayStation 3.

A block diagram of the Cell processor is shown in Figure 8.13. The master processor is similar to a 64-bit PowerPC CPU. There are eight slave processors. A high-speed bus interconnects the master processor and each of the slave processors. For those interested, a more detailed description of the PowerPC Cell processor can be found in Gschwind, et. al. [GSCH06].

FIGURE 8.13

Cell Processor Block Diagram



8.6 A FEW COMMENTS ON IMPLEMENTATION

It is not the intention of this book to discuss the electronic implementation of the computer in any detail. A brief introduction is provided in Supplementary Chapter 1, but the details of such a discussion are better left to an engineering textbook. There are several good computer engineering textbooks listed in the *For Further Reading* for the supplementary chapter if you are interested in learning more about how the computer works.

Although the increased capacity of current integrated circuit technology has allowed computer designers the option of creating very complex circuits, much of that capacity is currently used to provide increased capability in the form of multiple execution units, increased amounts of cache memory, and multicore processing; the basic design and implementation of a processor is simpler than you might imagine.

If you look again at the instruction classes that constitute the operations of a CPU together with the fetch-execute cycles that make up each of the instructions, you can see that the great majority of operations within the CPU consist of moving data from one register to another. The steps

```
PC → MAR and
MDR → IR
```

are examples of this type of operation.

In addition, we must include the capability to add data to a register with data from another register or from a constant (usually the constant 1 or −1), the capability to perform simple Boolean functions (AND, OR, NOT) on data in registers, and the capability to shift the data in a register to the left or right. Finally, the CPU must include the capability to make simple decisions based on the values stored in flags and registers (conditional branches). All of these operations are under the timed control of a clock. Control unit logic opens and closes switches at the right times to control the individual operations and the movement of data from one component within the CPU to another.

And for all practical purposes, that's about it. The small number of different operations used in a CPU suggest that the CPU can be directly implemented in electronic hardware, and indeed that is the case. In Supplementary Chapter 1, we demonstrate for the curious reader, in somewhat simplified fashion, that all of the preceding functions can be implemented using logic gates that perform Boolean algebra. The registers, flags, and counters that control timing are made up of electronic devices called flip-flops, which are, themselves, made up of logic gates.

So, as you can see, the basic hardware implementation of the CPU is relatively straightforward and simple. Although the addition of pipelining, superscaling, and other features complicates the design, it is possible, with careful design, to implement and produce an extremely fast and efficient CPU at low cost and in large quantities.

SUMMARY AND REVIEW

In this chapter we presented a number of different techniques that are used to enhance the power and flexibility of a CPU. We began with a discussion of three different approaches to CPU architecture, with particular emphasis on traditional computer architecture. We presented the advantages, disadvantages, and trade-offs for each architecture.

Next, we looked at the various aspects of instruction execution in a CPU, with the purpose of improving performance. This discussion culminated in the presentation of a model for an alternative organization that preserves the basic rules of execution, but allows much faster instruction execution. The important features of this model include separating the fetch-execute cycle into two separate fetch and execute units that can operate in parallel, pipelining to allow instructions to execute in an assembly line, and multiple execution units to allow parallel execution of unrelated instructions. A variety of innovative techniques, including rename registers, speculative execution, out-of-order execution, and branch prediction help to reduce bottlenecks and contribute to the performance. We noted that the resulting model is capable of superscalar processing, with instructions processed at average rates far exceeding the cycle rate of the clock.

We then turned our attention to memory enhancements, particularly the techniques and benefits of cache memory, a fast, intermediate memory between then CPU and regular memory. Following this, we put together a model of the compleat susperscalar CPU that contained all of the features that we had presented up to this point.

To increase performance even further, it is possible to combine CPUs into multiple units that share memory, buses, I/O, and other resources, a concept called multiprocessing. We presented two different configurations of multiprocesors.

We concluded the chapter with a brief introduction to the technology used to implement the modern processor.

With all of the advances and variations in technology, architecture, and organization that we have introduced in this chapter, and despite all of the different types of computers, applications, and uses for computers that are available today, it is important to remember that, regardless of the specifics, every current CPU conforms to the basic model created by Von Neumann more than a half century ago. There is little evidence that the basic concepts that govern CPU operation are likely to change in the near future.

FOR FURTHER READING

The many references used to write this chapter are all listed in the bibliography section at the end of the book. The following books and articles are particularly useful for their clear descriptions and explanations of these topics. Stallings [STAL05] and Tanenbaum [TANE05] describe the different types of architectures, focusing on the differences between CISC and RISC architectures in great depth. Information about the VLIW and EPIC architectures may be found at the Transmeta and Intel websites, respectively. The IBM website contains a wealth of information about zSeries, POWER, and Cell architectures, including the *Redbooks*, which are free, downloadable, book-length explanations of various computer topics. These range in difficulty from beginner to highly technical. Intel.com (for the x86 series) and sun.com (for the SPARC architecture) are other useful brand-specific websites.

Instruction sets, instruction formats, and addressing are discussed at length in every computer architecture textbook. The book by Patterson and Hennessy [PATT07] covers the topics of Chapters 7 and 8 thoroughly, and has the additional benefit of being highly readable. A more advanced treatment, by the same authors, is found in [HENN06]. Good discussions of multiprocessing are also found in Patterson and Hennessy and in Tanenbaum. Two readable websites introducing the Cell processor are Gschwind, et. al. [GSCH06] and Moore [MOOR06].

A different approach to this material is to compare the architectures of various machines. The book by Tabak [TABA95] looks at several different CPUs in detail. Most of these CPUs are obsolete, but the comparison between different architectures is useful. There are textbooks and trade books devoted to the architecture of every major CPU. The most thorough discussion of the x86 architecture is found in Mueller [MUEL08]. I also recommend Brey [BREY08], Messmer [MESS01], and Sargent and Shoemaker [SARG95] for the Intel x86 series. The PC System Architecture series is a collection of short books describing the architectures of various parts of computers. Volume 5 [SHAN04] describes the evolution and architecture of the Pentium 4. The case studies provided in Supplementary Chapter 2 are additional information resources.

Stallings also includes an entire chapter on superscalar processors. Clear, detailed discussions of all aspects of CPU and memory design can be found in the two books by Patterson and Hennessy [PATT07, HENN06]. There are many additional references in each of these books. Specific discussions of the superscalar processing techniques for particular CPUs can be found in Liptay [LIPT92] for the IBM ES/9000 mainframe, in Becker and colleagues [BECK93], Thompson and Ryan [THOM94], Burgess and colleagues [BURG94], and Ryan [RYAN93] for the PowerPC, and "Tour of the P6" [THOR95] for the P6.

An alternative approach to the topics in this chapter can be found in any assembly language textbook. There are many good books on these topics, with new ones appearing every day. A website such as Amazon is a good resource for identifying the best currently available.

KEY CONCEPTS AND TERMS

branch history table
cache controller
cache line
clock
code morphing layer
control dependency
data dependency
disk cache
execution unit
explicitly parallel instruction computer (EPIC)
fetch unit
hazard
hit
hit ratio
instruction reordering
instruction set architecture (ISA)
instruction unit
locality of reference
logical register
logical storage elements
master-slave multiprocessing
memory interleaving
miss
multiprocessor systems
multicore processor
n-way interleaving
organisation
pipelining
register alias table
rename register
scalar processing
simultaneous thread multiprocessing (STM)
speculative execution
stall time
superscalar processing
symmetrical multiprocessing (SMP)
tag
threads
tightly coupled system
very long instruction word (VLIW)
wide path memory access
write-back
write through

READING REVIEW QUESTIONS

8.1 The x86 series is an example of a CPU *architecture*. As you are probably aware, there are a number of different chips, including some from different manufacturers even, that qualify as x86 CPUs. What, *exactly*, defines the x86 architecture? What word defines the difference between the various CPUs that share the same architecture? Name at least one different CPU architecture.

8.2 What is the major performance advantage that results from the use of multiple general-purpose data registers?

8.3 Explain the advantage in implementing separate fetch and execute units in a CPU. What additional task is implemented in the fetch unit as a performance enhancement measure?

8.4 Explain how pipelining serves to reduce the average number of steps in the execution part of the fetch-execute cycle.

8.5 Which class of instructions can reduce performance by potentially invalidating the instructions in a pipeline? Identify two methods that can be used to partially overcome this problem.

8.6 Most CPUs today are *superscalar*. What does that mean?

8.7 The use of multiple execution units can improve performance but also cause problems called *hazards* or *dependencies*. Explain how a hazard can occur. How can hazards be managed?

8.8 What is a *rename register*? What is it used for?

8.9 What performance improvement is offered by *memory interleaving*?

8.10 What specific performance improvement is offered by the use of cache memory?

8.11 Describe how cache memory is organized. What is a *cache line*? How is it used?

8.12 Explain the *hit ratio* in cache memory.

8.13 Explain the difference between cache *write-through* and cache *write-back*. Which method is safer? Which method is faster?

8.14 Explain what takes place when cache memory is full.

8.15 Explain the *locality of reference* principle and its relationship to cache memory performance and the hit ratio.

8.16 When a system has multiple levels of cache memory, L2 always has more memory than L1. Why is this necessary?

8.17 Modern computers are usually described as multicore. What does this mean? Under ideal conditions, what performance gain would be achieved using a four-core processor over a single-core processor?

8.18 Identify and briefly explain two different ways of configuring a multiprocessing system. Which configuration is more effective for general purpose computing? Which configuration is more effective for handling specialized processing tasks, such as those used in game applications?

EXERCISES

8.1 Find a good reference that describes the x86 chip. Discuss the features of the architecture that make superscalar processing possible in this chip. What limitations does the Pentium architecture impose on its superscalar processing?

8.2 Consider a CPU that implements a single instruction fetch-decode-execute-write-back pipeline for scalar processing. The execution unit of this pipeline assumes that the execution stage requires one step. Describe, and show in diagram form, what happens when an instruction that requires one execution step follows one that requires four execution steps.

8.3 **a.** Consider a CPU with two parallel integer execution units. An addition instruction requires 2 clock pulses to complete execution, and a multiplication requires 15 clock pulses. Now assume the following situation: the program is to multiply two numbers, located in registers R2 and R4, and store the results in R5. The following instruction adds the number in R5 to the number in R2 and stores the result in R5. The CPU does not stall for data dependencies, and both instructions have access to an execution unit simultaneously. The initial values of R2, R4, and R5 are 3, 8, and 0, respectively. What is the result? Now assume that the CPU does handle data dependencies correctly. What is the result? If we define wasted time as time in which an execution unit is not busy, how much time is wasted in this example?

b. Now assume that a later instruction in the fetch pipeline has no data dependencies. It adds the value in R1, initially 4, to the value in R4 and stores the result in R5. Data dependencies are handled correctly. There are no rename registers, and the CPU retires instructions in order. What happens? If the CPU provides rename registers, what happens? What effect does out-of-order execution have upon the time required to execute this program?

8.4 Suppose that a CPU always executes the two instructions following a branch instruction, regardless of whether the branch is taken or not. Explain how this can eliminate most of the delay resulting from branch dependency in a pipelined CPU. What penalties or restrictions does this impose on the programs that are executed on this machine?

8.5 Some systems use a branch prediction method known as static branch prediction, so called because the prediction is made on the basis of the instruction, without regard to history. One possible scenario would have the system predict that all conditional backward branches are taken and all forward conditional branches are not taken. Recall your experience with programming in the Little Man Computer language. Would this algorithm be effective? Why or why not? What aspects of normal programming, in any programming language, support your conclusion?

8.6 How would you modify the Little Man Computer to implement the pipelined instruction fetch-execution unit model that was described in this chapter? What would it take to supply multiple execution units? Describe your modified LMC in detail and show how an instruction flows through it.

8.7 **a.** Suppose we are trying to determine the speed of a computer that executes the Little Man instruction set. The LOAD and STORE instructions each make up about 25% of the instructions in a typical program; ADD, SUBTRACT, IN, and OUT take 10% each. The various branches each take about 5%. The HALT instruction is almost never used (a maximum of once each program, of course!). Determine the average number of instructions executed each second if the clock ticks at 100 MHz.

b. Now suppose that the CPU is pipelined, so that each instruction is fetched while another instruction is executing. (You may also neglect the time required to refill the pipeline during branches and at the start of program execution.) What is the average number of instructions that can be executed each second with the same clock in this case?

8.8 The goal of scalar processing is to produce, on average, the execution of one instruction per clock tick. If the clock ticks at a rate of 2 GHz, how many instructions per second can this computer execute? How many instructions would a 2 GHz superscalar processor that processes three instructions per clock cycle execute?

8.9 Consider a cache memory that provides three hundred 16-byte blocks. Now consider that you are processing all the data in a two-dimensional array of, say, four hundred rows by four hundred columns, using a pair of nested loops. Assume that the program stores the array column by column. You can write your program to nest the loops in either direction, that is, process row by row or column by column. Explain which way you would choose to process the data. What is the advantage? Conversely, what is the disadvantage of processing the data the other way? What effect does choosing the incorrect way have on system performance?

8.10 Carefully discuss what happens when a cache miss occurs. Does this result in a major slowdown in execution of the instruction? If so, why?

8.11 What is the purpose of the tag in a cache memory system?

8.12 Describe the trade-offs between the memory cache write-through and write-back techniques.

8.13 Carefully describe the advantages and disadvantages of master-slave multiprocessing and symmetrical multiprocessing. Which would you select for fault-tolerant computing? Why?

8.14 Locate information about the Cell Processor. Describe the tasks performed by the various slave processors. What is the primary role of the master processor? Explain the advantages of master-slave multiprocessing over other forms of processing for this application. Can you think of some other types of computer problems that would benefit from this approach?

8.15 As you know, a single CPU processes one instruction at a time. Adding a second CPU (or *core*, in current terminology) allows the system to process two instructions at a time, simultaneously, effectively doubling the processing power of the system. A third core will offer triple the processing power of a single CPU, and so on. However, studies have shown that, in general, the expected increase in computing power starts to decline when the number of cores grows large, beyond eight or so. Why would you expect this to be the case? For what types of computing problems might this *not* be true?