FUNDAMENTALS OF COMPUTER ORGANIZATION AND ARCHITECTURE

Mostafa Abd-El-Barr
King Fahd University of Petroleum & Minerals (KFUPM)

Hesham El-Rewini
Southern Methodist University

CHAPTER ONE

Introduction to Computer Systems

The technological advances witnessed in the computer industry are the result of a long chain of immense and successful efforts made by two major forces. These are the academia, represented by university research centers, and the industry, represented by computer companies. It is; however, fair to say that the current technological advances in the computer industry owe their inception to university research centers. In order to appreciate the current technological advances in the computer industry, one has to trace back through the history of computers and their development. The objective of such historical review is to understand the factors affecting computing as we know it today and hopefully to forecast the future of computation.
A great majority of the computers of our daily use are known as general purpose machines. These are machines that are built with no specific application in mind, but rather are capable of performing computation needed by a diversity of applications. These machines are to be distinguished from those built to serve (tailored to) specific applications. The latter are known as special purpose machines.
Computer systems have conventionally been defined through their interfaces at a number of layered abstraction levels, each providing functional support to its predecessor. Included among the levels are the application programs, the high-level languages, and the set of machine instructions. Based on the interface between different levels of the system, a number of computer architectures can be defined. The interface between the application programs and a high-level language is referred to as language architecture.
The instruction set architecture defines the interface between the basic machine instruction set and the runtime and I/O control. A different definition of computer architecture is built on four basic viewpoints. These are the structure, the organization, the implementation, and the performance. In this definition, the structure defines the interconnection of various hardware components, the organization defines the dynamic interplay and management of the various components, the implementation defines the detailed design

of hardware components, and the performance specifies the behavior of the computer system.

Historical Background

In this section, we would like to provide a historical background on the evolution of cornerstone ideas in the computing industry. We should emphasize at the outset that the effort to build computers has not originated at one single place. There is every reason for us to believe that attempts to build the first computer existed in different geographically distributed places. We also firmly believe that building a computer requires teamwork. Therefore, when some people attribute a machine to the name of a single researcher, what they actually mean is that such researcher may have led the team who introduced the machine. We, therefore, see it more appropriate to mention the machine and the place it was first introduced without linking that to a specific name. We believe that such an approach is fair and should eliminate any controversy about researchers and their names. It is probably fair to say that the first program-controlled (mechanical) computer ever build was the Z1 (1938). This was followed in 1939 by the Z2 as the first operational program-controlled computer with fixed-point arithmetic.

However, the first recorded university-based attempt to build a computer originated on Iowa State University campus in the early 1940s. Researchers on that campus were able to build a small-scale special-purpose electronic computer. However, that computer was never completely operational. Just about the same time a complete design of a fully functional programmable special-purpose machine, the Z3, was reported in Germany in 1941. It appears that the lack of funding prevented such design from being implemented. History recorded that while these two attempts were in progress, researchers from different parts of the world had opportunities to gain first-hand experience through their visits to the laboratories and institutes carrying out the work. It is assumed that such first-hand visits and interchange of ideas enabled the visitors to embark on similar projects in their own laboratories back home.

As far as general-purpose machines are concerned, the University of Pennsylvania is recorded to have hosted the building of the

Electronic Numerical Integrator and Calculator (ENIAC) machine in 1944. It was the first operational general-purpose machine built using vacuum tubes. The machine was primarily built to help compute artillery firing tables during World War II. It was programmable through manual set-ting of switches and plugging of cables. The machine was slow by today's standard, with a limited amount of storage and primitive programmability. An improved version of the ENIAC was proposed on the same campus. The improved version of the ENIAC, called the Electronic Discrete Variable Automatic Computer (EDVAC), was an attempt to improve the way programs are entered and explore the concept of stored programs.

It was not until 1952 that the EDVAC project was completed. Inspired by the ideas implemented in the ENIAC, researchers at the Institute for Advanced Study (IAS) at Princeton built (in 1946) the IAS machine, which was about 10 times faster than the ENIAC.

In 1946 and while the EDVAC project was in progress, a similar project was initiated at Cambridge University. The project was to build a stored-program com-puter, known as the Electronic Delay Storage Automatic Calculator (EDSAC). It was in 1949 that the EDSAC became the world's first full-scale, stored-program, fully operational computer. A spin-off of the EDSAC resulted in a series of machines introduced at Harvard. The series consisted of MARK I, II, III, and IV. The latter two machines introduced the concept of separate memories for instructions and data. The term Harvard Architecture was given to such machines to indicate the use of separate memories. It should be noted that the term Harvard Architecture is used today to describe machines with separate cache for instructions and data.

The first general-purpose commercial computer, the UNIVersal Automatic Computer (UNIVAC I), was on the market by the middle of 1951. It represented an improvement over the BINAC, which was built in 1949. IBM announced its first com-puter, the IBM701, in 1952. The early 1950s witnessed a slowdown in the computer industry. In 1964 IBM announced a line of products under the name IBM 360 series. The series included a number of models that varied in price and performance. This led Digital Equipment Corporation

(DEC) to introduce the first minicomputer, the PDP-8. It was considered a remarkably low-cost machine.

Intel introduced the first microprocessor, the Intel 4004, in 1971. The world witnessed the birth of the first personal computer (PC) in 1977 when Apple computer series were first introduced. In 1977 the world also witnessed the introduction of the VAX-11/780 by DEC. Intel followed suit by introducing the first of the most popular microprocessor, the 80 86 series.

Personal computers, which were introduced in 1977 by Altair, Processor Technology, North Star, Tandy, Commodore, Apple, and many others, enhanced the productivity of end-users in numerous departments. Personal computers from Compaq, Apple, IBM, Dell, and many others, soon became pervasive, and changed the face of computing.

In parallel with small-scale machines, supercomputers were coming into play. The first such supercomputer, the CDC 6600, was introduced in 1961 by Control Data Corporation. Cray Research Corporation introduced the best cost/performance supercomputer, the Cray-1, in 1976.

The 1980s and 1990s witnessed the introduction of many commercial parallel computers with multiple processors. They can generally be classified into two main categories: (1) shared memory and (2) distributed memory systems. The number of processors in a single machine ranged from several in a shared memory computer to hundreds of thousands in a massively parallel system. Examples of parallel computers during this era include Sequent Symmetry, Intel iPSC, nCUBE, Intel Paragon, Thinking Machines (CM-2, CM-5), MsPar (MP), Fujitsu (VPP500), and others.

One of the clear trends in computing is the substitution of centralized servers by networks of computers. These networks connect inexpensive, powerful desktop machines to form unequaled computing power. Local area networks (LAN) of powerful personal computers and workstations began to replace mainframes and minis by 1990. These individual desktop computers were soon to be connected into larger complexes of computing by wide area networks (WAN).

TABLE 1.1 Four Decades of Computing

Feature	Batch	Time-sharing	Desktop	Network
Decade	1960s	1970s	1980s	1990s
Location	Computer room	Terminal room	Desktop	Mobile
Users	Experts	Specialists	Individuals	Groups
Data	Alphanumeric	Text, numbers	Fonts, graphs	Multimedia
Objective	Calculate	Access	Present	Communicate
Interface	Punched card	Keyboard & CRT	See & point	Ask & tell
Operation	Process	Edit	Layout	Orchestrate
Connectivity	None	Peripheral cable	LAN	Internet
Owners	Corporate computer centers	Divisional IS shops	Departmental end-users	Everyone

CRT, cathode ray tube; LAN, local area network.

The pervasiveness of the Internet created interest in network computing and more recently in grid computing. Grids are geographically distributed platforms of com-putation. They should provide dependable, consistent, pervasive, and inexpensive access to high-end computational facilities. Table 1.1 is modified from a table proposed by Lawrence Tesler (1995). In this table, major characteristics of the different computing paradigms are associated with each decade of computing, starting from 1960.

Architectural Development And Styles

Computer architects have always been striving to increase the performance of their architectures. This has taken a number of forms. Among these is the philosophy that by doing more in a single instruction, one can use a smaller number of instructions to perform the same job. The immediate consequence of this is the need for fewer memory read/write operations and an eventual speedup of operations. It was also argued that increasing the complexity of instructions and the number of addressing modes has the theoretical advantage of reducing the “semantic gap” between the instructions in a high-level language and those in the low-level (machine) language. A single (machine) instruction to convert several binary coded decimal (BCD) numbers to binary is an example for how complex

some instructions were intended to be. The huge number of addressing modes considered (more than 20 in the VAX machine) further adds to the complexity of instructions. Machines following this philosophy have been referred to as complex instructions set computers (CISCs). Examples of CISC machines include the Intel Pentium™, the Motorola MC68000™, and the IBM & Macintosh PowerPC™.

It should be noted that as more capabilities were added to their processors, manufacturers realized that it was increasingly difficult to support higher clock rcomplexity of computations within a single clock period. A number of studies from the mid-1970s and early-1980s also identified that in typical programs more than 80% of the instructions executed are those using assignment statements, conditional branching and procedure calls. It was also surprising to find out that simple assign-ment statements constitute almost 50% of those operations. These findings caused a different philosophy to emerge. This philosophy promotes the optimization of architectures by speeding up those operations that are most frequently used while reducing the instruction complexities and the number of addressing modes. Machines following this philosophy have been referred to as reduced instructions set computers (RISCs). Examples of RISCs include the Sun SPARC™ and MIPS™ machines.

The above two philosophies in architecture design have led to the unresolved controversy as to which architecture style is "best." It should, however, be mentioned that studies have indicated that RISC architectures would indeed lead to faster execution of programs. The majority of contemporary microprocessor chips seems to follow the RISC paradigm. In this book we will present the salient features and examples for both CISC and RISC machines.

Technological Development

Computer technology has shown an unprecedented rate of improvement. This includes the development of processors and memories. Indeed, it is the advances in technology that have fueled the computer industry. The integration of numbers of transistors (a transistor is a controlled on/off switch) into a single chip has increased from a few hundred to millions. This impressive increase

has been made possible by the advances in the fabrication technology of transistors.

The scale of integration has grown from small-scale (SSI) to medium-scale (MSI) to large-scale (LSI) to very large-scale integration (VLSI), and currently to wafer-scale integration (WSI). Table 1.2 shows the typical numbers of devices per chip in each of these technologies. It should be mentioned that the continuous decrease in the minimum devices feature size has led to a continuous increase in the number of devices per chip,

TABLE 1.2 Numbers of Devices per Chip

Integration	Technology	Typical number of devices	Typical functions
SSI	Bipolar	10– 20	Gates and flip-flops
MSI	Bipolar & MOS	50– 100	Adders & counters
LSI	Bipolar & MOS	100– 10,000	ROM & RAM
VLSI	CMOS (mostly)	10,000– 5,000,000	Processors
WSI	CMOS	.5,000,000	DSP & special purposes

which in turn has led to a number of developments. Among these is the increase in the number of devices in RAM memories, which in turn helps designers to trade off memory size for speed. The improvement in the feature size provides golden opportunities for introducing improved design styles.

CHAPTER 2

Instruction Set Architecture and Design

In this chapter, we consider the basic principles involved in instruction set architecture and design. Our discussion starts with a consideration of memory locations and addresses. We present an abstract model of the main memory in which it is considered as a sequence of cells each capable of storing n bits. We then address the issue of storing and retrieving information into and from the memory. The information stored and/or retrieved from the memory needs to be addressed.

A discussion on a number of different ways to address memory locations (addressing modes) is the next topic to be discussed in the chapter. A program consists of a number of instructions that have to be accessed in a certain order. That motivates us to explain the issue of instruction execution and sequencing in some detail. We then show the application of the presented addressing modes and instruction characteristics in writing sample segment codes for performing a number of simple programming tasks.

A unique characteristic of computer memory is that it should be organized in a hierarchy. In such hierarchy, larger and slower memories are used to supplement smaller and faster ones. A typical memory hierarchy starts with a small, expensive, and relatively fast module, called the cache. The cache is followed in the hierarchy by a larger, less expensive, and relatively slow main memory part. Cache and main memory are built using semiconductor material. They are followed in the hierarchy by larger, less expensive, and far slower magnetic memories that consist of the (hard) disk and the tape. Our concentration in this chapter is on the (main) memory from the programmer's point of view. In particular, we focus on the way information is stored in and retrieved out of the memory.

Memory Locations and Operations

The (main) memory can be modeled as an array of millions of adjacent cells, each capable of storing a binary digit (bit), having value of 1 or 0. These cells are organized in the form of groups of fixed number, say n, of cells that can be dealt with as an atomic entity. An entity consisting of 8 bits is called a byte. In many systems, the entity consisting of n bits that can be stored and retrieved in and out of the memory using one basic memory operation is called a word (the smallest addressable entity). Typical size of a word ranges from 16 to 64 bits. It is, however, customary to express the size of the memory in terms of bytes. For example, the size of a typical memory of a personal computer is 256 Mbytes, that is, 256 2^{20} ¼ 2^{28} bytes.

In order to be able to move a word in and out of the memory, a distinct address has to be assigned to each word.

This address will be used to determine the location in the memory in which a given word is to be stored. This is called a memory write operation. Similarly, the address will be used to determine the memory location from which a word is to be retrieved from the memory. This is called a memory read operation. The number of bits, l, needed to distinctly address M words in a memory is given by l ¼ $\log_2$ M.

For example, if the size of the memory is 64 M (read as 64 mega-words), then the number of bits in the address is $\log_2$ (64 2^{20}) ¼ $\log_2$ (2^{26}) ¼ 26 bits. Alternatively, if the number of bits in the address is l, then the maximum memory size (in terms of the number of words that can be addressed using these l bits) is M ¼ 2^l. Figure 2.1 illustrates the concept of memory words and word address as explained above.

As mentioned above, there are two basic memory operations. These are the memory write and memory read operations. During a memory write operation a word is stored into a memory location whose address is specified. During a memory read operation a word is read from a memory location whose address is specified. Typically, memory read and memory write operations are performed by the central processing unit (CPU).

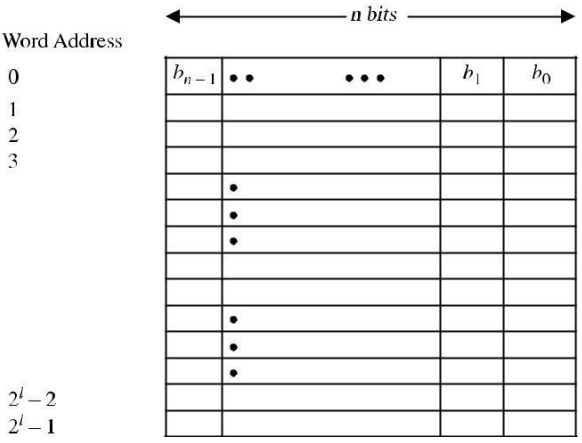

Figure 2.6 Illustration of the direct addressing mode

Three basic steps are needed in order for the CPU to perform a write operation into a specified memory location:

- The word to be stored into the memory location is first loaded by the CPU into a specified register, called the memory data register (MDR).
- The address of the location into which the word is to be stored is loaded by the CPU into a specified register, called the memory address register (MAR).
- A signal, called write, is issued by the CPU indicating that the word stored in the MDR is to be stored in the memory location whose address in loaded in the MAR.

Figure 2.2 illustrates the operation of writing the word given by 7E (in hex) into the memory location whose address is 2005. Part a of the figure shows the status of the registers and memory locations involved in the write operation before the execution of the operation. Part b of the figure shows the status after the execution of the operation. It is worth mentioning that the MDR and the MAR are registers used exclusively by the CPU and are not accessible to the programmer.

Similar to the write operation, three basic steps are needed in order to perform a memory read operation:

- The address of the location from which the word is to be read is loaded into the MAR.
- A signal, called read, is issued by the CPU indicating that the word whose address is in the MAR is to be read into the MDR
- After some time, corresponding to the memory delay in reading the specified word, the required word will be loaded by the memory into the MDR ready for use by the CPU.

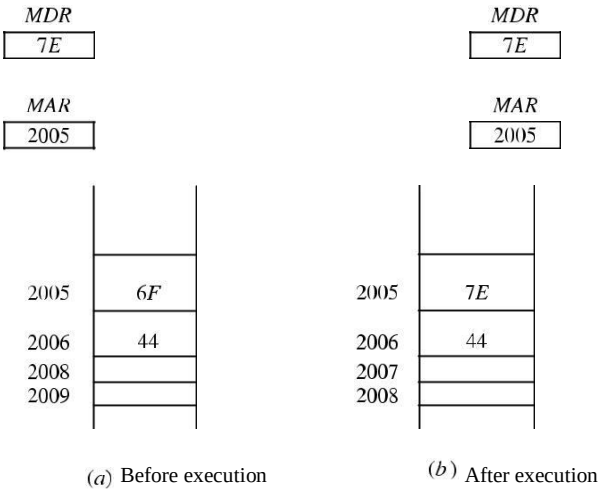

Figure 2.2 Illustration of the memory write operation

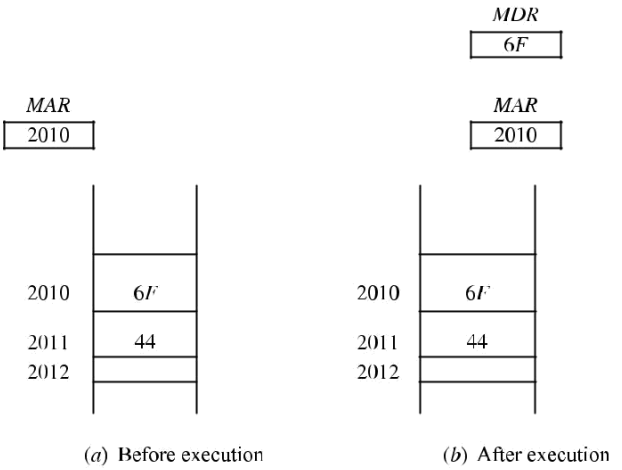

Figure 2.3 Illustration of the memory read operation

Figure 2.3 illustrates the operation of reading the word stored in the memory location whose address is 2010. Part a of the figure shows the status of the registers and memory locations involved in the read operation before the execution of the operation. Part b of the figure shows the status after the read operation.

Addressing Modes

Information involved in any operation performed by the CPU needs to be addressed. In computer terminology, such information is called the operand. Therefore, any instruction issued by the processor must carry at least two types of information. These are the operation to be performed, encoded in what is called the op-code field, and the address information of the operand on which the operation is to be performed, encoded in what is called the address field.

Instructions can be classified based on the number of operands as: three-address, two-address, one-and-half-address, one-address, and zero-address. We explain these classes together with simple examples in the following paragraphs. It should be noted that in presenting these examples, we would use the convention operation, source, destination to express any instruction. In that convention, operation rep-resents the operation to be performed, for example, add, subtract, write, or read. The source field represents the source operand(s). The source operand can be a constant, a value stored in a register, or a value stored in the memory. The destination field represents the place where the result of the operation is to be stored, for example, a register or a memory location.

three-address instruction takes the form operation add-1, add-2, add-3.

In this form, each of add-1, add-2, and add-3 refers to a register or to a memory location. Consider, for example, the instruction ADD R_1,R_2,R_3. This instruction indicates that the operation to be performed is addition. It also indicates that the values to be added are those stored in registers R_1 and R_2 that the results should be stored in register R_3.

An example of a three-address instruction that refers to memory locations may take the form ADD A,B,C. The instruction adds the contents of memory location A to the contents of memory location B and stores the result in memory location C.

A two-address instruction takes the form operation add-1, add-2. In this form, each of add-1 and add-2 refers to a register or to a memory location. Consider, for example, the instruction ADD R_1,R_2. This instruction adds the contents of register R_1 to the contents of register R_2 and stores the results in register R_2.

The original contents of register R_2 are lost due to this operation while the original contents of register R_1 remain intact. This instruction is equivalent to a three-address instruction of the form ADD R_1,R_2,R_2. A similar instruction that uses memory locations instead of registers can take the form ADD A,B. In this case, the contents of memory location A are added to the contents of memory location B and the result is used to override the original contents of memory location B.

The operation performed by the three-address instruction ADD A,B,C can be per-formed by the two two-address instructions MOVE B,C and ADD A,C. This is because the first instruction moves the contents of location B into location C and the second instruction adds the contents of location A to those of location C (the con-tents of location B) and stores the result in location C.

A one-address instruction takes the form ADD R_1. In this case the instruction implicitly refers to a register, called the Accumulator R_{acc}, such that the contents of the accumulator is added to the contents of the register R_1 and the results are stored back into the accumulator R_{acc}.

If a memory location is used instead of a register then an instruction of the form ADD B is used. In this case, the instruction adds the content of the accumulator R_{acc} to the content of memory location B and stores the result back into the accumulator R_{acc}. The instruction ADD R_1 is equivalent to the three-address instruction ADD R_1,R_{acc},R_{acc} or to the two-address instruction ADD R_1,R_{acc}.

Between the two- and the one-address instruction, there can be a one-and-half address instruction. Consider, for example, the instruction ADD B,R_1. In this case, the instruction adds the contents of register R_1 to the contents of memory location B and stores the result in register R_1. Owing to the fact that the instruction uses two types of addressing, that is, a register and a memory location, it is called a one-and-half-address instruction. This is because register addressing needs a smaller number of bits than those needed by memory addressing.

It is interesting to indicate that there exist zero-address instructions. These are the instructions that use stack operation.

A stack is a data organization mechanism in which the last data item stored is the first data item retrieved. Two specific operations can be performed on a stack. These are the push and the pop operations. Figure 2.4 illustrates these two operations.

As can be seen, a specific register, called the stack pointer (SP), is used to indicate the stack location that can be addressed. In the stack push operation, the SP value is used to indicate the location (called the top of the stack) in which the value (5A) is to be stored (in this case it is location 1023). After storing (pushing) this value the SP is

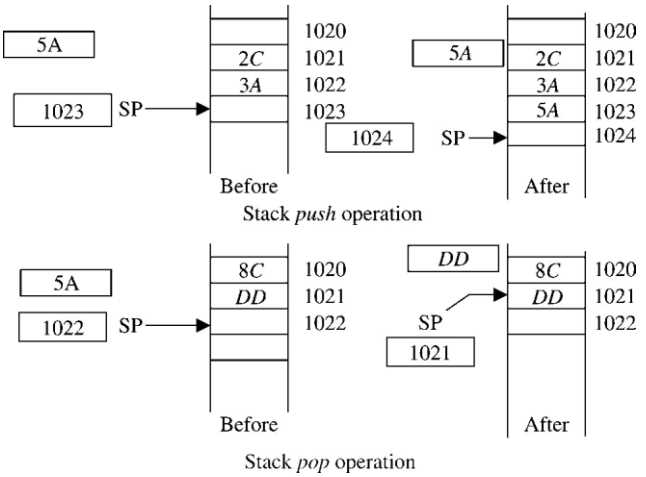

Figure 2.4 The stack push and pop operations

incremented to indicate to location 1024. In the stack pop operation, the SP is first decremented to become 1021. The value stored at this location (DD in this case) is retrieved (popped out) and stored in the shown register. Different operations can be performed using the stack structure. Consider, for example, an instruction such as ADD (SP)þ, (SP). The instruction adds the contents of the stack location pointed to by the SP to those pointed to by the SP þ 1 and stores the result on the stack in the location pointed to by the current value of the SP. Figure 2.5 illustrates such an addition operation. Table 2.1 summarizes the instruction classification discussed above.

The different ways in which operands can be addressed are called the addressing modes. Addressing modes differ in the way the address information of operands is specified. The simplest addressing mode is to include the operand itself in the instruction, that is, no address information is needed. This is called immediate addressing.
A more involved addressing mode is to compute the address of the operand by adding a constant value to the content of a register. This is called indexed addressing. Between these two addressing modes there exist a number of other addressing modes including absolute addressing, direct addressing, and indirect addressing. A number of different addressing modes are explained below.

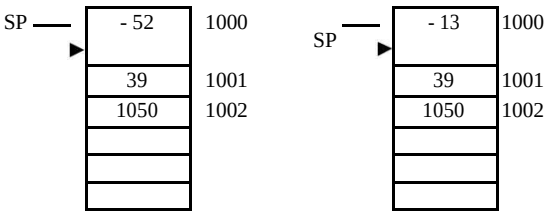

Figure 2.5 Addition using the stack

TABLE 2.1 Instruction Classification

Instruction class	Example
Three-address	ADD R_1,R_2,R_3
	ADD A,B,C
Two-address	ADD R_1,R_2
	ADD A,B
One-and-half-address	ADD B,R_1
One-address	ADD R_1
Zero-address	ADD (SP)þ, (SP)

Immediate Mode

According to this addressing mode, the value of the operand is (immediately) avail-able in the instruction itself. Consider, for example, the case of loading the decimal value 1000 into a register R_i. This operation can be performed using an instruction such as the following: LOAD #1000, R_i. In this instruction, the operation to be per-formed is to load a value into a register. The source operand is (immediately) given as 1000, and the destination is the register R_i. It should be noted that in order to indicate that the value 1000 mentioned in the instruction is the operand itself and not its address (immediate mode), it is customary to prefix the operand by the special character (#). As can be seen the use of the immediate addressing mode is simple. The use of immediate addressing leads to poor programming practice. This is because a change in the value of an operand requires a change in every instruction that uses the immediate value of such an operand. A more flexible addressing mode is explained below.

Direct (Absolute) Mode

According to this addressing mode, the address of the memory location that holds the operand is included in the instruction. Consider, for example, the case of loading the value of the operand stored in memory location 1000 into register R_i. This operation can be performed using an instruction such as LOAD 1000, R_i. In this instruction, the source operand is the value stored in the memory location whose address is 1000, and the destination is the register R_i. Note that the value 1000 is not prefixed with any special characters, indicating that it is the (direct or absolute) address of the source operand. Figure 2.6 shows an illustration of the direct addressing mode.

For

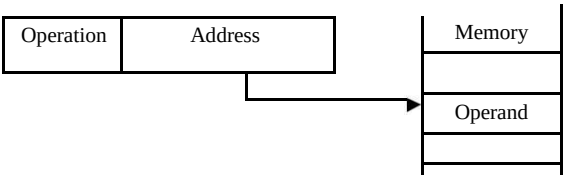

Figure 2.6 Illustration of the direct addressing mode

example, if the content of the memory location whose address is 1000 was (2345) at the time when the instruction LOAD 1000, R_i is executed, then the result of executing such instruction is to load the value (2345) into register R_i. Direct (absolute) addressing mode provides more flexibility compared to the immediate mode. However, it requires the explicit inclusion of the operand address in the instruction. A more flexible addressing mechanism is provided through the use of the indirect addressing mode. This is explained below.

Indirect Mode

In the indirect mode, what is included in the instruction is not the address of the operand, but rather a name of a register or a memory location that holds the (effective) address of the operand. In order to indicate the use of indirection in the instruction, it is customary to include the name of the register or the memory location in parentheses. Consider, for example, the instruction LOAD (1000), R_i. This instruction has the memory location 1000 enclosed in parentheses, thus indicating indirection. The meaning of this instruction is to load register R_i with the contents of the memory location whose address is stored at memory address 1000. Because indirection can be made through either a register or a memory location, therefore, we can identify two types of indirect addressing. These are register indirect addressing, if a register is used to hold the address of the operand, and memory indirect addressing, if a memory location is used to hold the address of the operand. The two types are illustrated in Figure 2.7.

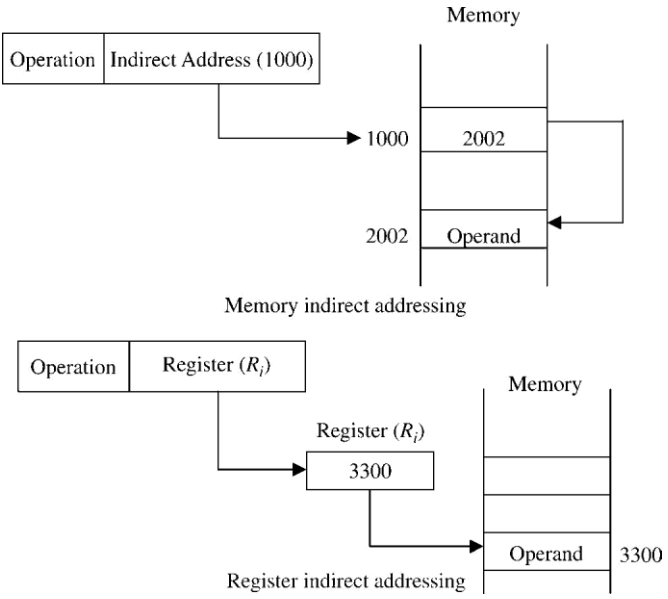

Figure 2.7 Illustration of the indirect addressing mode

Indexed Mode

In this addressing mode, the address of the operand is obtained by adding a constant to the content of a register, called the index register. Consider, for example, the instruction LOAD X(R_{ind}), R_i. This instruction loads register R_i with the contents of the memory location whose address is the sum of the contents of register R_{ind} and the value X. Index addressing is indicated in the instruction by including the name of the index register in parentheses and using the symbol X to indicate the constant to be added. Figure 2.8 illustrates indexed addressing. As can be seen, indexing requires an additional level of complexity over register indirect addressing.

Other Modes

The addressing modes presented above represent the most commonly used modes in most processors. They provide the programmer with sufficient means to handle most general programming tasks. However, a number of other addressing modes have been used in a number of processors to facilitate execution of specific programming tasks. These additional addressing modes are more involved as compared to those presented above.

Among these addressing modes the relative, auto-increment, and the auto-decrement modes represent the most well-known ones. These are explained below.

Relative Mode Recall that in indexed addressing, an index register, R_{ind} , is used. Relative addressing is the same as indexed addressing except that the program counter (PC) replaces the index register. For example, the instruction LOAD X(PC), R_i loads register R_i with the contents of the memory location whose address is the sum of the contents of the program counter (PC) and the value X.

Figure 2.9 illustrates the relative addressing mode.

Auto-increment Mode This addressing mode is similar to the register indirect addressing mode in the sense that the effective address of the operand is the content of a register; call it the auto-increment register, that is included in the instruction.

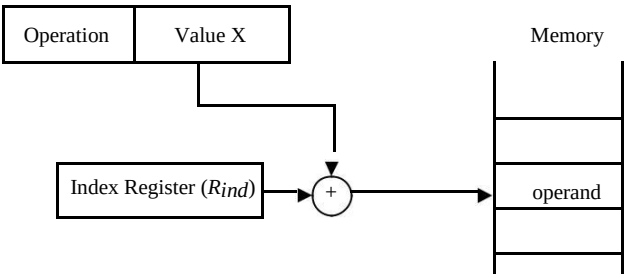

Figure 2.8 Illustration of the indexed addressing mode

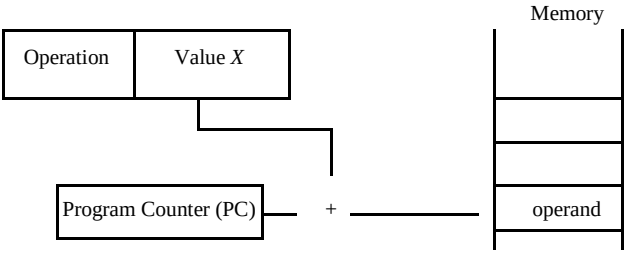

Figure 2.9 Illustration of relative addressing mode

However, with auto-increment, the content of the auto-increment register is automatically incremented after accessing the operand. As before, indirection is indicated by including the auto-increment register in parentheses. The automatic increment of the register's content after accessing the operand is indicated by including a (þ) after the parentheses. Consider, for example, the instruction LOAD (R_{auto})þ, R_i. This instruction loads register R_i with the operand whose address is the content of register R_{auto}. After loading the operand into register R_i, the content of register R_{auto} is incremented, pointing for example to the next item in a list of items. Figure 2.10 illustrates the auto-increment addressing mode.

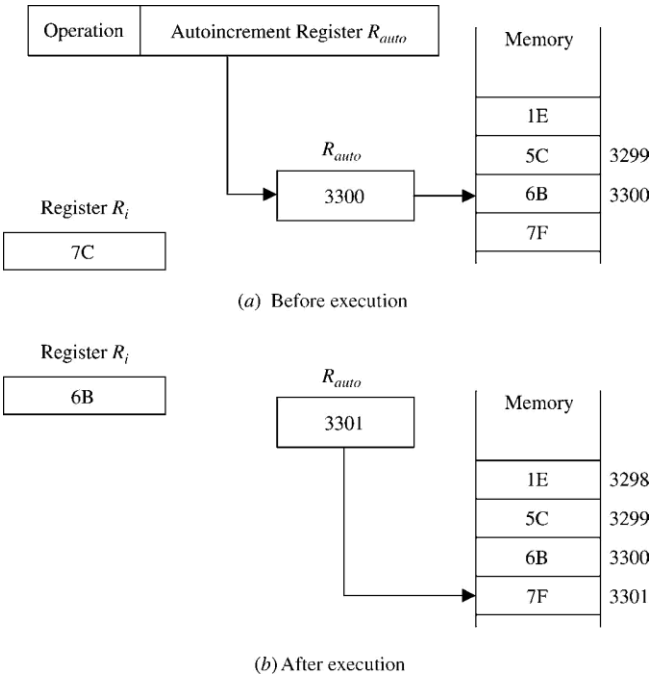

Figure 2.10 Illustration of the auto-increment addressing mode

Auto-decrement Mode Similar to the auto-increment, the auto-decrement mode uses a register to hold the address of the operand. However, in this case the content of the auto-decrement register is first decremented and the new content is used as the effective address of the operand. In order to reflect the fact that the content of the auto-decrement register is decremented before accessing the operand, a (2) is included before the indirection parentheses. Consider, for example, the instruction LOAD (R_{auto}), R_i. This instruction decrements the content of the register R_{auto} and then uses the new content as the effective address of the operand that is to be loaded into register R_i. Figure 2.11 illustrates the auto-decrement addressing mode. The seven addressing modes presented above are summarized in Table 2.2. In each case, the table shows the name of the addressing mode, its definition, and a generic example illustrating the use of such mode.

In presenting the different addressing modes we have used the load instruction for illustration. However, it should be understood that there are other types of instructions in a given machine. In the following section we elaborate on the different types of instructions that typically constitute the instruction set of a given machine.

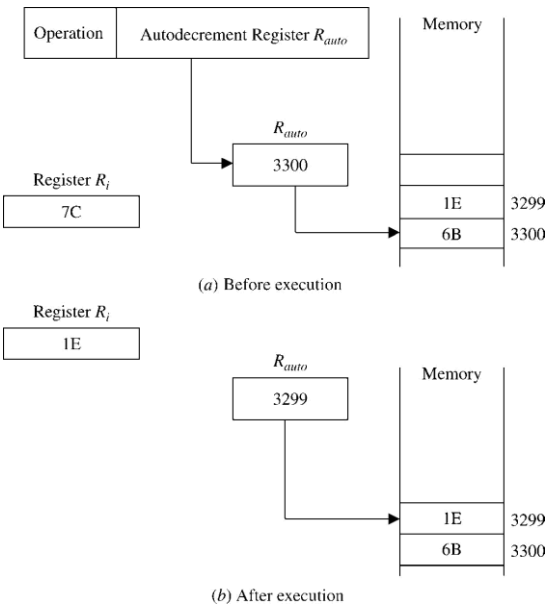

Figure 2.11 Illustration of the auto-decrement addressing mode

TABLE 2.2 Summary of Addressing Modes

Addressing mode	Definition	Example	Operation
Immediate	Value of operand is included in the instruction	load #1000, R_i	R_i 1000
Direct (Absolute)	Address of operand is included in the instruction	load 1000, R_i	R_i M[1000]
Register indirect	Operand is in a memory location whose address is in the register specified in the instruction	load (R_j), R_i	R_i M[R_j]
Memory indirect	Operand is in a memory location whose address is in the memory location specified in the instruction	load (1000), R_i	R_i M[1000]
Indexed	Address of operand is the sum of an index value and the contents of an index register	load X(R_{ind}), R_i	R_i M[R_{ind} þ X]
Relative	Address of operand is the sum of an index value and the contents of the program counter	load X(PC), R_i	R_i M[PC þ X]
Autoincrement	Address of operand is in a register whose value is incremented after fetching the operand	load (R_{auto})þ, R_i	R_i M[R_{auto}] R_{auto} R_{auto} þ 1
Autodecrement	Address of operand is in a register whose value is decremented before fetching the operand	load 2 (R_{auto}), R_i	R_{auto} R_{auto} 2 1 R_i M[R_{auto}]

Instruction Types

The type of instructions forming the instruction set of a machine is an indication of the power of the underlying architecture of the machine. Instructions can in general be classified as in the following Subsections

Data Movement Instructions

Data movement instructions are used to move data among the different units of the machine. Most notably among these are instructions that are used to move data among the different registers in the CPU. A simple register to register movement of data can be made through the instruction

$$\text{MOVE } R_i, R_j$$

TABLE 2.3 Some Common Data Movement Operations

Data movement operation	Meaning
MOVE	Move data (a word or a block) from a given source (a register or a memory) to a given destination
LOAD	Load data from memory to a register
STORE	Store data into memory from a register
PUSH	Store data from a register to stack
POP	Retrieve data from stack into a register

This instruction moves the content of register R_i to register R_j. The effect of the instruc-tion is to override the contents of the (destination) register R_j without changing the con-tents of the (source) register R_i. Data movement instructions include those used to move data to (from) registers from (to) memory. These instructions are usually referred to as the load and store instructions, respectively. Examples of the two instructions are

LOAD 25838, R_j
STORE R_i, 1024

The first instruction loads the content of the memory location whose address is 25838 into the destination register R_j. The content of the memory location is unchanged by executing the LOAD instruction. The STORE instruction stores the content of the source register R_i into the memory location 1024. The content of the source register is unchanged by executing the STORE instruction. Table 2.3 shows some common data transfer operations and their meanings.

Arithmetic and Logical Instructions

Arithmetic and logical instructions are those used to perform arithmetic and logical manipulation of registers and memory contents. Examples of arithmetic instructions include the ADD and SUBTRACT instructions. These are

ADD R_1,R_2,R_0

SUBTRACT R_1,R_2,R_0

The first instruction adds the contents of source registers R_1 and R_2 and stores the result in destination register R_0. The second instruction subtracts the contents of the source registers R_1 and R_2 and stores the result in the destination register R_0. The contents of the source registers are unchanged by the ADD and the SUBTRACT instructions. In addition to the ADD and SUBTRACT instructions, some machines have MULTIPLY and DIVIDE instructions. These two instructions are expensive to implement and could be substituted by the use of repeated addition or repeated subtraction. Therefore, most modern architectures do not have MULTIPLY or

TABLE 2.4 Some Common Arithmetic Operations

Arithmetic operations	Meaning
ADD	Perform the arithmetic sum of two operands
SUBTRACT	Perform the arithmetic difference of two operands
MULTIPLY	Perform the product of two operands
DIVIDE	Perform the division of two operands
INCREMENT	Add one to the contents of a register
DECREMENT	Subtract one from the contents of a register

DIVIDE instructions on their instruction set. Table 2.4 shows some common arithmetic operations and their meanings. Logical instructions are used to perform logical operations such as AND, OR, SHIFT, COMPARE, and ROTATE. As the names indicate, these instructions per-form, respectively, and, or, shift, compare, and rotate operations on register or memory contents. Table 2.5 presents a number of logical operations.

Sequencing Instructions

Control (sequencing) instructions are used to change the sequence in which instructions are executed.

They take the form of CONDITIONAL BRANCHING (CONDITIONAL JUMP), UNCONDITIONAL BRANCHING (JUMP), or CALL instructions. A common characteristic among these instructions is that their execution changes the program counter (PC) value. The change made in the PC value can be unconditional, for example, in the unconditional branching or the jump instructions. In this case, the earlier value of the PC is lost and execution of the program starts at a new value specified by the instruction. Consider, for example, the instruction JUMP NEW-ADDRESS. Execution of this instruction will cause the PC to be loaded with the memory location represented by NEW-ADDRESS whereby the instruction stored at this new address is executed. On the other hand,

TABLE 2.5 Some Common Logical Operations

Logical operation	Meaning
AND	Perform the logical ANDing of two operands
OR	Perform the logical ORing of two operands
EXOR	Perform the XORing of two operands
NOT	Perform the complement of an operand
COMPARE	Perform logical comparison of two operands and set flag accordingly
SHIFT	Perform logical shift (right or left) of the content of a register
ROTATE	Perform logical shift (right or left) with wraparound of the content of a register

TABLE 2.6 Examples of Condition Flags

Flag name	Meaning
Negative (N)	Set to 1 if the result of the most recent operation is negative, it is 0 otherwise
Zero (Z)	Set to 1 if the result of the most recent operation is 0, it is 0 otherwise
Overflow (V)	Set to 1 if the result of the most recent operation causes an overflow, it is 0 otherwise
Carry (C)	Set to 1 if the most recent operation results in a carry, it is 0 otherwise

the change made in the PC by the branching instruction can be conditional based on the value of a specific flag.

Examples of these flags include the Negative (N), Zero (Z), Overflow (V), and Carry (C). These flags represent the individual bits of a specific register, called the CONDITION CODE (CC) REGISTER. The values of flags are set based on the results of executing different instructions. The meaning of each of these flags is shown in Table 2.6.
Consider, for example, the following group of instructions.

```
        LOAD                #100, R1
Loop:   ADD                 (R2) þ , R0
        DECREMENT           R1
        BRANCH-IF-GREATER-THAN Loop
```

The fourth instruction is a conditional branch instruction, which indicates that if the result of decrementing the contents of register R_1 is greater than zero, that is, if the Z flag is not set, and then the next instruction to be executed is that labeled by Loop. It should be noted that conditional branch instructions could be used to exe-cute program loops (as shown above).

The CALL instructions are used to cause execution of the program to transfer to a subroutine. A CALL instruction has the same effect as that of the JUMP in terms of loading the PC with a new value from which the next instruction is to be executed. However, with the CALL instruction the incremented value of the PC (to point to the next instruction in sequence) is pushed onto the stack. Execution of a RETURN instruction in the subroutine will load the PC with the popped value from the stack. This has the effect of resuming program execution from the point where branching to the subroutine has occurred. Figure 2.12 shows a program segment that uses the CALL instruction. This pro-gram segment sums up a number of values, N, and stores the result into memory location SUM. The values to be added are stored in N consecutive memory locations starting at NUM. The subroutine, called ADDITION, is used to perform the actual addition of values while the main program stores the results in SUM.

Figure 2.12 A program segment using a subroutine

Input / Output Instructions

Input and output instructions (I/O instructions) are used to transfer data between the computer and peripheral devices. The two basic I/O instructions used are the INPUT and OUTPUT instructions. The INPUT instruction is used to transfer data from an input device to the processor. Examples of input devices include a keyboard or a mouse. Input devices are interfaced with a computer through dedicated input ports. Computers can use dedicated addresses to address these ports. Suppose that the input port through which a keyboard is connected to a computer carries the unique address 1000. Therefore, execution of the instruction INPUT 1000 will cause the data stored in a specific register in the interface between the keyboard and the computer, call it the input data register, to be moved into a specific register (called the accumulator) in the computer. Similarly, the execution of the instruction OUTPUT 2000 causes the data stored in the accumulator to be moved to the data output register in the output device whose address is 2000. Alternatively, the computer can address these ports in the usual way of addressing memory locations.

In this case, the computer can input data from an input device by executing an instruction such as MOVE R_{in}, R_0. This instruction moves the content of the register R_{in} into the register R_0. Similarly, the instruction MOVE R_0, R_{in} moves the contents of register R_0 into the register R_{in}, that is, performs an output operation. This

TABLE 2.7 Some Transfer of Control Operations

Transfer of control operation	Meaning
BRANCH-IF-CONDITION	Transfer of control to a new address if condition is true
JUMP	Unconditional transfer of control
CALL	Transfer of control to a subroutine
RETURN	Transfer of control to the caller routine

latter scheme is called memory-mapped Input/Output. Among the advantages of memory-mapped I/O is the ability to execute a number of memory-dedicated instructions on the registers in the I/O devices in addition to the elimination of the need for dedicated I/O instructions. Its main disadvantage is the need to dedicate part of the memory address space for I/O devices.

CHAPTER 3
Processing Unit Design

In this chapter, we focus our attention on the main component of any computer system, the central processing unit (CPU). The primary function of the CPU is to execute a set of instructions stored in the computer's memory. A simple CPU consists of a set of registers, an arithmetic logic unit (ALU), and a control unit (CU). In what follows, the reader will be introduced to the organization and main operations of the CPU.

CPU BASICS

A typical CPU has three major components: (1) register set, (2) arithmetic logic unit (ALU), and (3) control unit (CU). The register set differs from one computer architecture to another. It is usually a combination of general-purpose and special-purpose registers. General-purpose registers are used for any purpose, hence the name general purpose.

Special-purpose registers have specific functions within the CPU. For example, the program counter (PC) is a special-purpose register that is used to hold the address of the instruction to be executed next. Another example of special-purpose registers is the instruction register (IR), which is used to hold the instruction that is currently executed. The ALU provides the circuitry needed to perform the arithmetic, logical and shift operations demanded of the instruction set. In Chapter 4, we have covered a number of arithmetic operations and circuits used to support computation in an ALU. The control unit is the entity responsible for fetching the instruction to be executed from the main memory and decoding and then executing it. Figure 5.1 shows the main components of the CPU and its interactions with the memory system and the input/ output devices.

The CPU fetches instructions from memory, reads and writes data from and to memory, and transfers data from and to input/output devices.

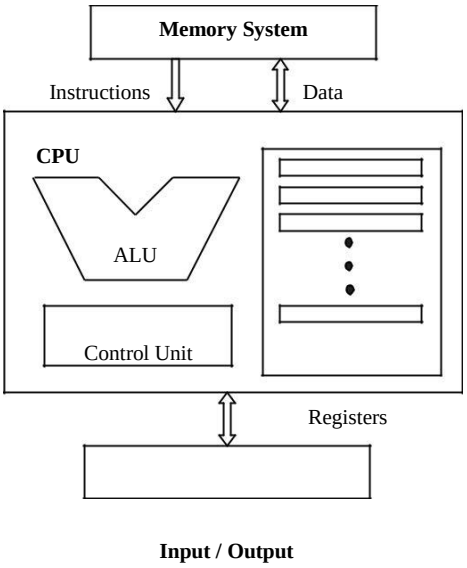

Figure 5.1 Central processing unit main components and interactions with the memory and I/O

A typical and simple execution cycle can be summarized as follows:

- The next instruction to be executed, whose address is obtained from the PC, is fetched from the memory and stored in the IR.
- The instruction is decoded.
- Operands are fetched from the memory and stored in CPU registers, if needed.
- The instruction is executed.

Results are transferred from CPU registers to the memory, if needed.

The execution cycle is repeated as long as there are more instructions to execute. A check for pending interrupts is usually included in the cycle. Examples of interrupts include I/O device request, arithmetic overflow, or a page fault .When an interrupt request is encountered, a transfer to an interrupt handling routine takes place. Interrupt handling routines are programs that are invoked to collect the state of the currently executing program, correct the cause of the interrupt, and restore the state of the program.

The actions of the CPU during an execution cycle are defined by micro-orders issued by the control unit. These micro-orders are individual control signals sent over dedicated control lines. For example, let us assume that we want to execute an instruction that moves the contents of register X to register Y. Let us also assume that both registers are connected to the data bus, D. The control unit will issue a control signal to tell register X to place its contents on the data bus D. After some delay, another control signal will be sent to tell register Y to read from data bus D. The activation of the control signals is determined using either hardwired control or microprogramming.

REGISTER SET

Registers are essentially extremely fast memory locations within the CPU that are used to create and store the results of CPU operations and other calculations.

Different computers have different register sets. They differ in the number of registers, register types, and the length of each register. They also differ in the usage of each register. General-purpose registers can be used for multiple purposes and assigned to a variety of functions by the programmer.

Special-purpose registers are restricted to only specific functions. In some cases, some registers are used only to hold data and cannot be used in the calculations of operand addresses. The length of a data register must be long enough to hold values of most data types. Some machines allow two contiguous registers to hold double-length values. Address registers may be dedicated to a particular addressing mode or may be used as address general purpose. Address registers must be long enough to hold the largest address. The number of registers in a particular architecture affects the instruction set design. A very small number of registers may result in an increase in memory references. Another type of registers is used to hold processor status bits, or flags. These bits are set by the CPU as the result of the execution of an operation. The status bits can be tested at a later time as part of another operation.

Memory Access Registers

Two registers are essential in memory write and read operations: the memory data register (MDR) and memory address register (MAR). The MDR and MAR are used exclusively by the CPU and are not directly accessible to programmers.

In order to perform a write operation into a specified memory location, the MDR and MAR are used as follows:

The word to be stored into the memory location is first loaded by the CPU into MDR.

The address of the location into which the word is to be stored is loaded by the CPU into a MAR.

Instruction Fetching Registers

Two main registers are involved in fetching an instruction for execution: the pro-gram counter (PC) and the instruction register (IR). The PC is the register that contains the address of the next instruction to be fetched. The fetched instruction is loaded in the IR for execution. After a successful instruction fetch, the PC is updated to point to the next instruction to be executed. In the case of a branch operation, the PC is updated to point to the branch target instruction after the branch is resolved, that is, the target address is known.

Condition Registers

Condition registers, or flags, are used to maintain status information. Some architectures contain a special program status word (PSW) register. The PSW contains bits that are set by the CPU to indicate the current status of an executing program. These indicators are typically for arithmetic operations, interrupts, memory protection information, or processor status.

Special-Purpose Address Registers

Index Register, in index addressing, the address of the operand is obtained by adding a constant to the content of a register, called the index register. The index register holds an address displacement. Index addressing is indicated in the instruction by including the name of the index register in parentheses and using the symbol X to indicate the constant to be added.

Segment Pointers support segmentation, the address issued by the processor should consist of a segment number (base) and a displacement (or an offset) within the segment. A segment register holds the address of the base of the segment.

Stack Pointer is a data organization mechanism in which the last data item stored is the first data item retrieved. Two specific operations can be performed on a stack. These are the Push and the Pop operations. A specific register, called the stack pointer (SP), is used to indicate the stack location that can be addressed. In the stack push operation, the SP value is used to indicate the location (called the top of the stack). After storing (pushing) this value, the SP is

incremented (in some architectures, e.g. X86, the SP is decremented as the stack grows low in memory).

DATAPATH

The CPU can be divided into a data section and a control section. The data section, which is also called the datapath, contains the registers and the ALU. The datapath is capable of performing certain operations on data items. The control section is basically the control unit, which issues control signals to the datapath. Internal to the CPU, data move from one register to another and between ALU and registers. Internal data movements are performed via local buses, which may carry data, instructions, and addresses. Externally, data move from registers to memory and I/O devices, often by means of a system bus. Internal data movement among registers and between the ALU and registers may be carried out using different organizations including one-bus, two-bus, or three-bus organizations. Dedicated datapaths may also be used between components that transfer data between them-selves more frequently. For example, the contents of the PC are transferred to the MAR to fetch a new instruction at the beginning of each instruction cycle. Hence, a dedicated datapath from the PC to the MAR could be useful in speeding up this part of instruction execution.

One-Bus Organization

Using one bus, the CPU registers and the ALU use a single bus to move outgoing and incoming data. Since a bus can handle only a single data movement within one clock cycle, two-operand operations will need two cycles to fetch the operands for the ALU. Additional registers may also be needed to buffer data for the ALU. This bus organization is the simplest and least expensive, but it limits the amount of data transfer that can be done in the same clock cycle, which will slow down the overall performance. Figure 5.3 shows a one-bus datapath consisting of a set of general-purpose registers, a memory address register (MAR), a memory data register (MDR), an instruction register (IR), a program counter (PC), and an ALU.

Figure 5.3: One-bus datapath

Two-Bus Organization

Using two buses is a faster solution than the one-bus organization. In this case, general-purpose registers are connected to both buses. Data can be transferred from two different registers to the input point of the ALU at the same time. Therefore, a two-operand operation can fetch both operands in the same clock cycle. An additional buffer register may be needed to hold the output of the ALU when the two buses are busy carrying the two operands. Figure 5.4a shows a two-bus organization. In some cases, one of the buses may be dedicated for moving data into registers (in-bus), while the other is dedicated for transferring data out of the registers (out-bus). In this case, the additional buffer register may be used, as one of the ALU inputs, to hold one of the operands. The ALU output can be connected directly to the in-bus, which will transfer the result into one of the registers. Figure 5.4b shows a two-bus organization with in-bus and out-bus.

Three-Bus Organization

In a three-bus organization, two buses may be used as source buses while the third is used as destination. The source buses move data out of registers (out-bus), and

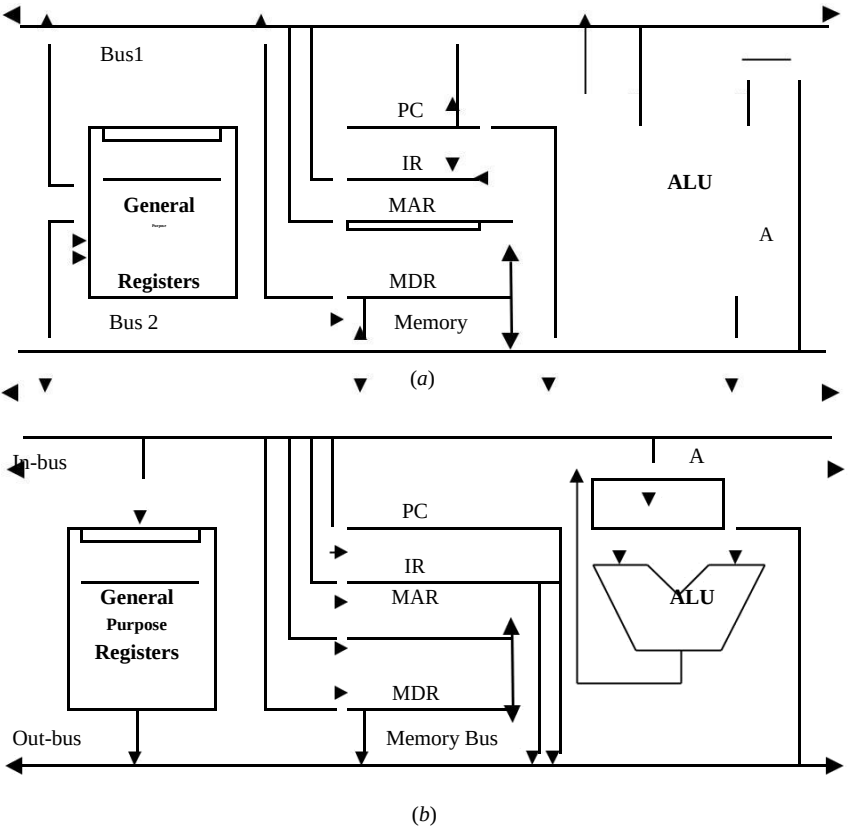

Figure 5.4 Two-bus organizations. (a) An Example of Two-Bus Datapath. (b) Another

Example of Two-Bus Datapath with in-bus and out-bus

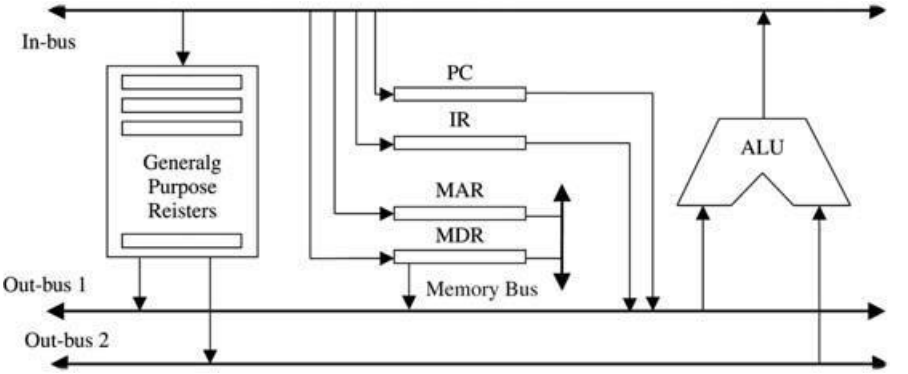

Figure 5.5: Three-bus datapath

the destination bus may move data into a register (in-bus). Each of the two out-buses is connected to an ALU input point. The output of the ALU is connected directly to the in-bus. As can be expected, the more buses we have, the more data we can move within a single clock cycle. However, increasing the number of buses will also increase the complexity of the hardware. Figure 5.5 shows an example of a three-bus datapath.

CPU Instruction Cycle

The sequence of operations performed by the CPU during its execution of instructions is presented in Fig. 5.6. As long as there are instructions to execute, the next instruction is fetched from main memory. The instruction is executed based on the operation specified in the opcode field of the instruction. At the completion of the instruction execution, a test is made to determine whether an interrupt has occurred. An interrupt handling routine needs to be invoked in case of an interrupt.

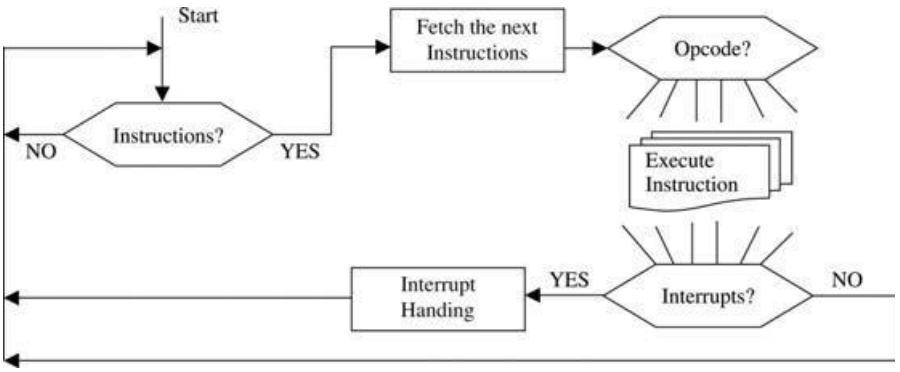

Figure 5.6: CPU functions

The basic actions during fetching an instruction, executing an instruction, or hand-ling an interrupt are defined by a sequence of micro-operations. A group of control signals must be enabled in a prescribed sequence to trigger the execution of a micro-operation. In this section, we show the micro-operations that implement instruction fetch, execution of simple arithmetic instructions, and interrupt handling.

Fetch Instructions

The sequence of events in fetching an instruction can be summarized as follows:

- The contents of the PC are loaded into the MAR.
- The value in the PC is incremented. (This operation can be done in parallel with a memory access.)
- As a result of a memory read operation, the instruction is loaded into the MDR.
- The contents of the MDR are loaded into the IR.

Let us consider the one-bus datapath organization shown in Fig. 5.3. We will see that the fetch operation can be accomplished in three steps as shown in the table below, where t_0 , t_1 , t_2 . Note that multiple operations separated by ";" imply that they are accomplished in parallel.

Step	Micro-operation
t_0	MAR (PC); A(PC)
t_1	MDR Mem[MAR]; PC(A) þ 4
t_2	IR (MDR)

Using the three-bus datapath shown in Figure 5.5, the following table shows the steps needed.

Step	Micro-operation
t_0	MAR (PC); PC(PC) þ 4
t_1	MDR Mem[MAR]
t_2	IR (MDR)

Execute Simple Arithmetic Operation

Add R_1 , R_2 , R_0 This instruction adds the contents of source registers R_1 and R_2 , and stores the results in destination register R_0 . This addition can be executed as follows:

- The registers R_0 , R_1 , R_2 , are extracted from the IR.
- The contents of R_1 and R_2 are passed to the ALU for addition.
- The output of the ALU is transferred to R_0 .

Using the one-bus datapath shown in Figure 5.3, this addition will take three steps as shown in the following table, where t_0 , t_1 , t_2 .

Step	Micro-operation
t_0	A (R_1)
t_1	B (R_2)
t_2	R_0 (A) þ (B)

Using the two-bus datapath shown in Figure 5.4a, this addition will take two steps as shown in the following table, where t_0 , t_1 .

Step	Micro-operation
t_0	$A(R_1)$ þ (R_2)
t_1	$R_0(A)$

Using the two-bus datapath with in-bus and out-bus shown in Figure 5.4b, this addition will take two steps as shown below, where t_0 , t_1 .

Step	Micro-operation
t_0	A (R_1)
t_1	R_0 (A) þ (R_2)

Using the three-bus datapath shown in Figure 5.5, this addition will take only one step as shown in the following table.

Step	Micro-operation
t_0	$R_0(R_1)$ þ (R_2)

Add X, R_0 This instruction adds the contents of memory location X to register R_0 and stores the result in R_0 . This addition can be executed as follows:

- The memory location X is extracted from IR and loaded into MAR.
- As a result of memory read operation, the contents of X are loaded into MDR.
- The contents of MDR are added to the contents of R_0 .

Using the one-bus datapath shown in Figure 5.3, this addition will take five steps as shown below, where t_0 , t_1 , t_2 , t_3 , t_4 .

Step	Micro-operation
t_0	MAR X
t_1	MDR Mem[MAR]
t_2	A (R_0)
t_3	B (MDR)
t_4	R_0 (A) þ (B)

Using the two-bus datapath shown in Figure 5.4a, this addition will take four steps as shown below, where t_0 , t_1 , t_2 , t_3.

Step	Micro-operation
t_0	MAR X
t_1	MDR Mem[MAR]
t_2	A (R_0) þ (MDR)
t_3	R_0 (A)

Using the two-bus datapath with in-bus and out-bus shown in Figure 5.4b, this addition will take four steps as shown below, where t_0 , t_1 , t_2 , t_3 .

Step	Micro-operation
t_0	MAR X
t_1	MDR Mem[MAR]
t_2	A (R_0)
t_3	R_0 (A) þ (MDR)

Using the three-bus datapath shown in Figure 5.5, this addition will take three steps as shown below, where t_0 , t_1 , t_2 .

Step	Micro-operation
t_0	MAR X
t_1	MDR Mem[MAR]
t_2	R_0 R_0 þ (MDR)

Interrupt Handling

After the execution of an instruction, a test is performed to check for pending inter-rupts. If there is an interrupt request waiting, the following steps take place:

- The contents of PC are loaded into MDR (to be saved).
- The MAR is loaded with the address at which the PC contents are to be saved.
- The PC is loaded with the address of the first instruction of the interrupt hand-ling routine.

The contents of MDR (old value of the PC) are stored in memory. The following table shows the sequence of events, where t_1 , t_2 , t_3 .

Step	Micro-operation
t_1	MDR (PC)
t_2	MAR address1 (where to save old PC);
	PC address2 (interrupt handling routine)
t_3	Mem[MAR] (MDR)

Control Unit

The control unit is the main component that directs the system operations by sending control signals to the datapath. These signals control the flow of data within the CPU and between the CPU and external units such as memory and I/O. Control buses generally carry signals between the control unit and other computer components in a clock-driven manner. The system clock produces a continuous sequence of pulses in a specified duration and frequency. A sequence of steps t_0 , t_1 , t_2 , . . . ,

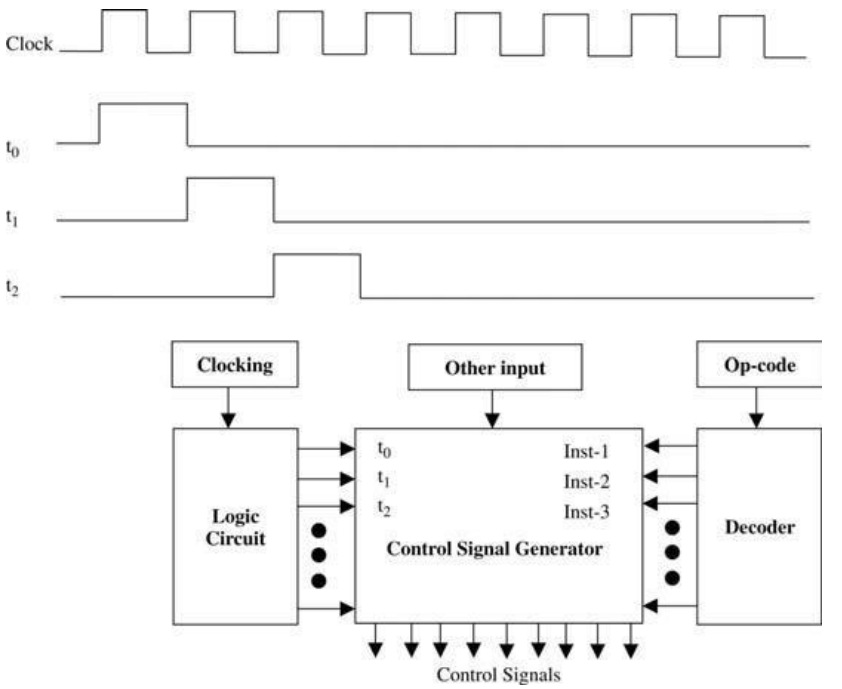

Figure 5.7: Timing of control signals

(t_0 , t_1 , t_2 , $\cdots$) are used to execute a certain instruction. The op-code field of a fetched instruction is decoded to provide the control signal generator with information about the instruction to be executed. Step information generated by a logic circuit module is used with other inputs to generate control signals. The signal generator can be specified simply by a set of Boolean equations for its output in terms of its inputs. Figure 5.7 shows a block diagram that describes how timing is used in generating control signals.

There are mainly two different types of control units: micro-programmed and hardwired. In micro-programmed control, the control signals associated with operations are stored in special memory units inaccessible by the programmer as control words. A control word is a microinstruction that specifies one or more micro-operations. A sequence of microinstructions is called a microprogram, which is stored in a ROM or RAM called a control memory CM.

In hardwired control, fixed logic circuits that correspond directly to the Boolean expressions are used to generate the control signals.

Clearly hardwired control is faster than micro-programmed control. However, hardwired control could be very expensive and complicated for complex systems. Hardwired control is more economical for small control units.
It should also be noted that micro-programmed control could adapt easily to changes in the system design. We can easily add new instructions without changing hardware. Hardwired control will require a redesign of the entire systems in the case of any change.

Hardwired Implementation
In hardwired control, a direct implementation is accomplished using logic circuits. For each control line, one must find the Boolean expression in terms of the input to the control signal generator as shown in Figure 5.7. Let us explain the

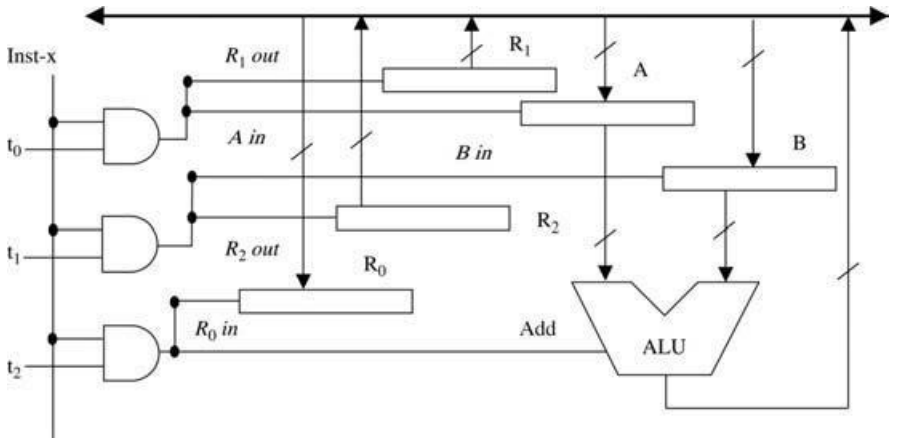

Figure 5.9: Signals generated to execute Inst-x on one-bus datapath during time period t_0, t_1, t_2

implementation using a simple example. Assume that the instruction set of a machine has the three instructions: Inst-x, Inst-y, and Inst-z; and A, B, C, D, E, F, G, and H are control lines. The following table shows the control lines that should be activated for the three instructions at the three steps t_0 , t_1 , and t_2 .

Step	Inst-x	Inst-y	Inst-z
t_0	D, B, E	F, H, G	E, H
t_1	C, A, H	G	D, A, C

t_2	G, C	B, C

The Boolean expressions for control lines A, B, and C can be obtained as follows:

A

B

C

Figure 5.10 shows the logic circuits for these control lines. Boolean expressions for the rest of the control lines can be obtained in a similar way. Figure 5.11 shows the state diagram in the execution cycle of these instructions.

Micro-programmed Control Unit

The idea of micro-programmed control units was introduced by M. V. Wilkes in the early 1950s. Microprogramming was motivated by the desire to reduce the complexities involved with hardwired control. As we studied earlier, an instruction is

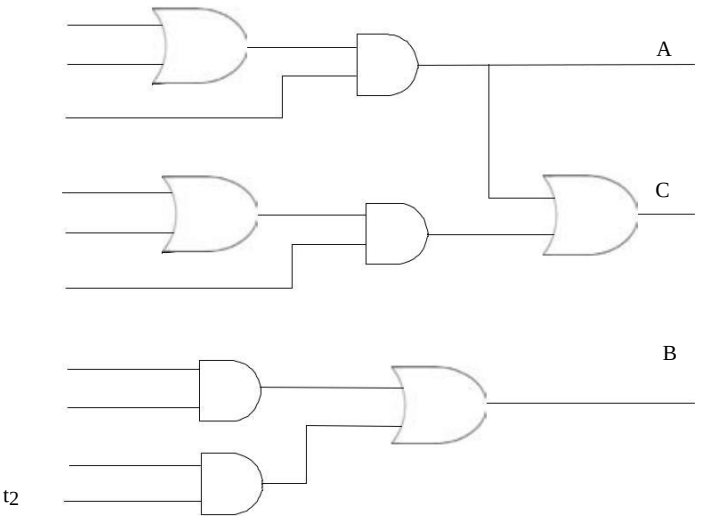

Figure 5.10 Logic circuits for control lines A, B, and C

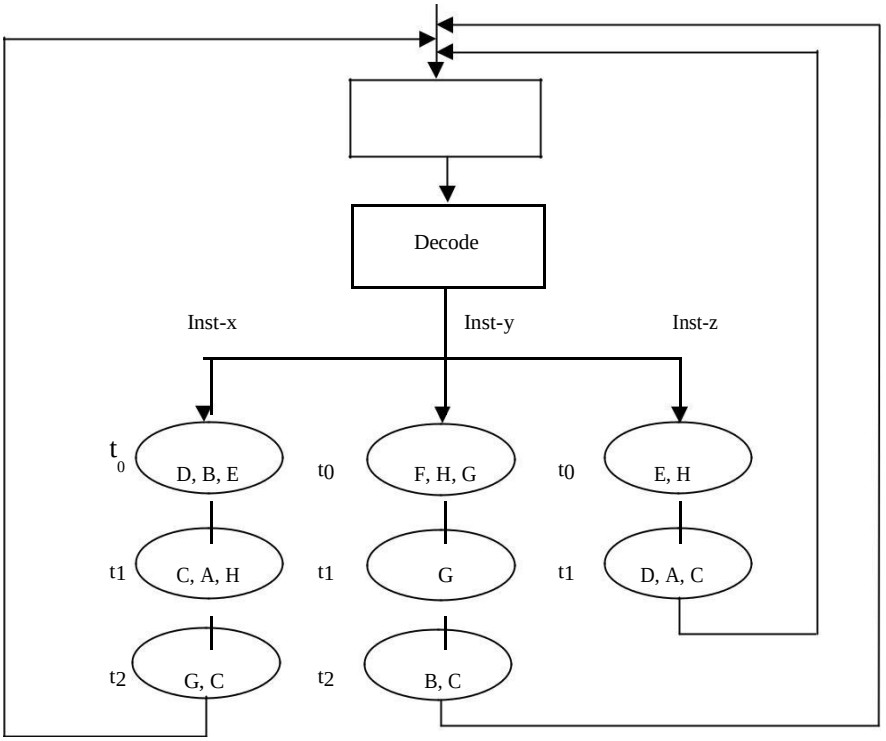

Figure 5.11xecution state diagram

implemented using a set of micro-operations. Associated with each microoperation is a set of control lines that must be activated to carry out the corresponding micro-operation. The idea of micro-programmed control is to store the control signals associated with the implementation of a certain instruction as a microprogram in a special memory called a control memory (CM). A microprogram consists of a sequence of microinstructions. A microinstruction is a vector of bits, where each bit is a control signal, condition code, or the address of the next microinstruction. Microinstructions are fetched from CM the same way program instructions are fetched from main memory (Fig. 5.12).

When an instruction is fetched from memory, the op-code field of the instruc-tion will determine which microprogram is to be executed. In other words, the op-code is mapped to a microinstruction address in the control memory. The microinstruction processor uses that address

to fetch the first microinstruction in the microprogram. After fetching each microinstruction, the appropriate control lines will be enabled. Every control line that corresponds to a “1” bit should be turned on. Every control line that corresponds to a “0” bit should be left off. After completing the execution of one microinstruction, a new microinstruction will be fetched and executed. If the condition code bits indicate that a branch must be taken, the next microinstruction is specified in the address bits of the cur-rent microinstruction. Otherwise, the next microinstruction in the sequence will be fetched and executed.

When an instruction is fetched from memory, the op-code field of the instruc-tion will determine which microprogram is to be executed. In other words, the op-code is mapped to a microinstruction address in the control memory. The microinstruction processor uses that address to fetch the first microinstruction in the microprogram. After fetching each microinstruction, the appropriate control lines will be enabled. Every control line that corresponds to a “1” bit should be turned on. Every control line that corresponds to a “0” bit should be left off. After completing the execution of one microinstruction, a new microinstruction will be fetched and executed. If the condition code bits indicate that a branch must be taken, the next microinstruction is specified in the address bits of the cur-rent microinstruction. Otherwise, the next microinstruction in the sequence will be fetched and executed.

The length of a microinstruction is determined based on the number of micro-operations specified in the microinstructions, the way the control bits will be interpreted, and the way the address of the next microinstruction is obtained. A microinstruction may specify one or more micro-operations that will be activated simultaneously. The length of the microinstruction will increase as the number of parallel micro-operations per microinstruction increases. Furthermore, when each control bit in the microinstruction corresponds to exactly one control line, the length of microinstruction could get bigger. The length of a microinstruction could be reduced if control lines are coded in specific fields in the microinstruction. Decoders will be needed to map each field into the individual control lines. Clearly, using the decoders will reduce the number of control lines that can be

activated simultaneously. There is a tradeoff between the length of the microinstructions and the amount of parallelism. It is important that we reduce the length of microinstructions to reduce the cost and access time of the control memory. It may also be desirable that more micro-operations be performed in parallel and more control lines can be activated simultaneously.

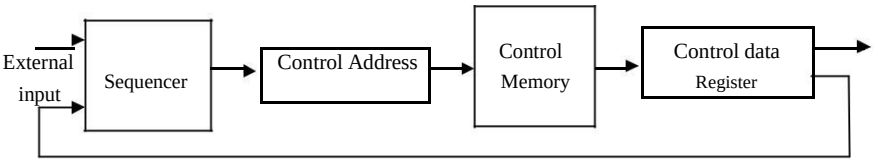

Figure 5.12 Fetching microinstructions (control words)

Horizontal Versus Vertical Microinstructions Microinstructions can be classified as horizontal or vertical. Individual bits in horizontal microinstructions correspond to individual control lines. Horizontal microinstructions are long and allow maximum parallelism since each bit controls a single control line. In vertical microinstructions, control lines are coded into specific fields within a microinstruction. Decoders are needed to map a field of k bits to 2^k possible combinations of control lines. For example, a 3-bit field in a microinstruction could be used to specify any one of eight possible lines. Because of the encoding, vertical microinstructions are much shorter than horizontal ones. Control lines encoded in the same field cannot be activated simultaneously. Therefore, vertical microinstructions allow only limited parallelism. It should be noted that no decoding is needed in horizontal microinstructions while decoding is necessary in the vertical case.

Example 3 Consider the three-bus datapath shown in Figure 5.5. In addition to the PC, IR, MAR, and MDR, assume that there are 16 general-purpose registers numbered $R_0 – R_{15}$. Also, assume that the ALU supports eight functions (add, sub-tract, multiply, divide, AND, OR, shift left, and shift right). Consider the add operation Add R_1 , R_2 , R_0 , which adds the contents of source registers R_1 , R_2 , and

store the results in destination register R_0 . In this example, we will study the format of the microinstruction under horizontal organization.
We will use horizontal microinstructions, in which there is a control bit for each control line.

The format of the microinstruction should have control bits for the following:

ALU operations

Registers that output to out-bus1 (source 1)

Registers that output to out-bus2 (source 2)

Registers that input from in-bus (destination)

Other operations that are not shown here

The following table shows the number of bits needed for ALU, Source 1, Source 2, and destination:

Purpose	Number of bits	Explanations
ALU	8 bits	8 functions
Source 1	20 bits	16 general-purpose registers þ 4 special-purpose registers
Source 2	16 bits	16 general-purpose registers
Destination	20 bits	16 general-purpose registers þ 4 special-purpose registers

5.13 is the microinstruction for Add R_1 , R_2 , R_0 on the three-bus datapath

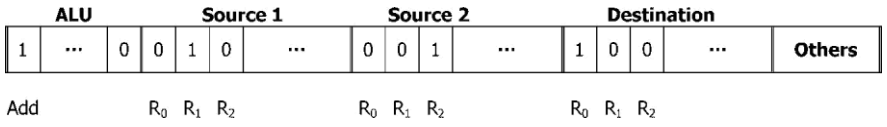

Figure 5.13: Microinstruction for Add R_1, R_2, R_0

CHAPTE 4
Memory System Design

Memory Hierarchy

A typical memory hierarchy starts with a small, expensive, and relatively fast unit, called the cache, followed by a larger, less expensive, and relatively slow main memory unit. Cache and main memory are built using solid-state semiconductor material (typically CMOS transistors). It is customary to call the fast memory level the primary memory. The solid-state memory is followed by larger, less expensive, and far slower magnetic memories that consist typically of the (hard) disk and the tape. It is customary to call the disk the secondary memory, while the tape is conventionally called the tertiary memory. The objective behind designing a memory hierarchy is to have a memory system that performs as if it consists entirely of the fastest unit and whose cost is dominated by the cost of the slowest unit.

The memory hierarchy can be characterized by a number of parameters. Among these parameters are the access type, capacity, cycle time, latency, bandwidth, and cost. The term access refers to the action that physically takes place during a read or write operation. The capacity of a memory level is usually measured in bytes. The cycle time is defined as the time elapsed from the start of a read operation to the start of a subsequent read. The latency is defined as the time interval between the request for information and the access to the first bit of that information. The bandwidth provides a measure of the number of bits per second that can be accessed. The cost of a memory level is usually specified as dollars per megabytes. Figure 6.1 depicts a typical memory hierarchy. Table 6.1 provides typical values of the memory hierarchy parameters.

The term random access refers to the fact that any access to any memory location takes the same fixed amount of time regardless of the actual memory location and/or the sequence of accesses that takes place. For example, if a write operation to memory location 100 takes 15 ns and if this operation is followed by a read operation to memory location 3000, then the latter operation will also take 15 ns.

This is to be compared to sequential access in which if access to location 100 takes 500 ns, and if a consecutive access to location 101 takes 505 ns, then it is expected that an access to location 300 may take 1500 ns. This is because the memory has to cycle through locations 100 to 300, with each location requiring 5 ns.

The effectiveness of a memory hierarchy depends on the principle of moving information into the fast memory infrequently and accessing it many times before replacing it with new information. This principle is possible due to a phenomenon called locality of reference; that is, within a given period of time, programs tend to reference a relatively confined area of memory repeatedly. There exist two forms of locality: spatial and temporal locality. Spatial locality refers to the

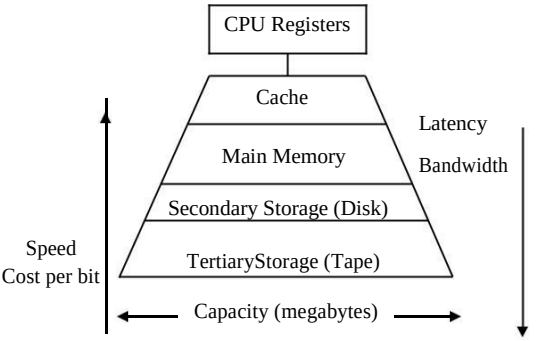

Figure 6.1 Typical memory hierarchy

TABLE 6.1 Memory Hierarchy Parameters

	Access type	Capacity	Latency	Bandwidth	Cost/MB
CPU registers	Random	64– 1024 bytes	1– 10 ns	System clock rate	High
Cache memory	Random	8– 512 KB	15– 20 ns	10– 20 MB/s	$500
Main memory	Random	16– 512 MB	30– 50 ns	1– 2 MB/s	$20 – 50
Disk memory	Direct	1– 20 GB	10– 30 ms	1– 2 MB/s	$0.25
Tape memory	Sequential	1– 20 TB	30– 10,000 ms	1– 2 MB/s	$0.025

phenomenon that when a given address has been referenced, it is most likely that addresses near it will be referenced within a short period of time, for example, consecutive instructions in a straightline program. Temporal locality, on the other hand, refers to the phenomenon that once a particular memory item has been referenced, it is most likely that it will be referenced next, for example, an instruction in a program loop.

The sequence of events that takes place when the processor makes a request for an item is as follows. First, the item is sought in the first memory level of the memory hierarchy. The probability of finding the requested item in the first level is called the hit ratio, h_1. The probability of not finding (missing) the requested item in the first level of the memory hierarchy is called the miss ratio, (1 2 h_1). When the requested item causes a "miss," it is sought in the next subsequent memory level. The probability of finding the requested item in the second memory level, the hit ratio of the second level, is h_2. The miss ratio of the second memory level is (1 h_2). The process is repeated until the item is found. Upon finding the requested item, it is brought and sent to the processor.

In a memory hierarchy that consists of three levels, the average memory access time can be expressed as follows:

The average access time of a memory level is defined as the time required to access one word in that level. In this equation, t_1, t_2, t_3 represent, respectively, the access times of the three levels.

Cache Memory

Cache memory owes its introduction to Wilkes back in 1965. At that time, Wilkes distinguished between two types of main memory: The conventional and the slave memory.

In Wilkes terminology, a slave memory is a second level of unconventional high-speed memory, which nowadays corresponds to what is called cache memory (the term cache means a safe place for hiding or storing things).

The idea behind using a cache as the first level of the memory hierarchy is to keep the information expected to be used more frequently by the CPU in the cache

(a small high-speed memory that is near the CPU). The end result is that at any given time some active portion of the main memory is duplicated in the cache. Therefore, when the processor makes a request for a memory reference, the request is first sought in the cache. If the request corresponds to an element that is currently residing in the cache, we call that a cache hit. On the other hand, if the request corre-sponds to an element that is not currently in the cache, we call that a cache miss. A cache hit ratio, h_c, is defined as the probability of finding the requested element in the cache. A cache miss ratio (1 h_c) is defined as the probability of not finding the requested element in the cache.

In the case that the requested element is not found in the cache, then it has to be brought from a subsequent memory level in the memory hierarchy. Assuming that the element exists in the next memory level, that is, the main memory, then it has to be brought and placed in the cache. In expectation that the next requested element will be residing in the neighboring locality of the current requested element (spatial locality), then upon a cache miss what is actually brought to the main memory is a block of elements that contains the requested element. The advantage of transferring a block from the main memory to the cache will be most visible if it could be possible to transfer such a block using one main memory access time.

Such a possibility could be achieved by increasing the rate at which information can be transferred between the main memory and the cache. One possible technique that is used to increase the bandwidth is memory interleaving. To achieve best results, we can assume that the block brought from the main memory to the cache, upon a cache miss, consists of elements that are stored in different memory modules, that is, whereby consecutive memory addresses are stored in successive memory modules. Figure 6.2 illustrates the simple case of a main memory consisting of eight memory modules. It is assumed in this case that the block consists of 8 bytes.

Having introduced the basic idea leading to the use of a cache memory, we would like to assess the impact of temporal and spatial locality on the performance of the memory hierarchy. In order to make such an assessment, we will limit our

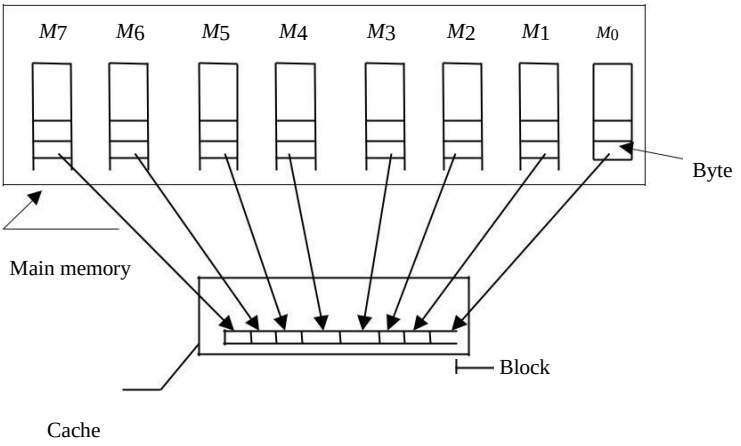

Figure 6.2 Memory interleaving using eight modules

deliberation to the simple case of a hierarchy consisting only of two levels, that is, the cache and the main memory. We assume that the main memory access time is t_m and the cache access time is t_c. We will measure the impact of locality in terms of the average access time, defined as the average time required to access an element (a word) requested by the processor in such a two-level hierarchy.

Impact of Temporal Locality

In this case, we assume that instructions in program loops, which are executed many times, for example, n times, once loaded into the cache, are used more than once before they are replaced by new instructions. The average access time, t_{av}, is given by

In deriving the above expression, it was assumed that the requested memory element has created a cache miss, thus leading to the transfer of a main memory block in time t_m. Following that, n accesses were made to the same requested element, each taking t_c. The above expression reveals that as the number of repeated accesses, n, increases, the average access time decreases, a desirable feature of the memory hierarchy.

Impact of Spatial Locality

In this case, it is assumed that the size of the block transferred from the main memory to the cache, upon a cache miss, is m elements. We also assume that due to spatial locality, all m elements were requested, one at a time, by the processor. Based on these assumptions, the average access time, t_{av}, is given by

In deriving the above expression, it was assumed that the requested memory element has created a cache miss, thus leading to the transfer of a main memory block, con-sisting of m elements, in time t_m. Following that, m accesses, each for one of the elements constituting the block, were made. The above expression reveals that as the number of elements in a block, m, increases, the average access time decreases, a desirable feature of the memory hierarchy.

Cache Memory Organization

There are three main different organization techniques used for cache memory. The three techniques are discussed below. These techniques differ in two main aspects:

The criterion used to place, in the cache, an incoming block from the main memory.

The criterion used to replace a cache block by an incoming block (on cache full).

Direct Mapping This is the simplest among the three techniques. Its simplicity stems from the fact that it places an incoming main memory block into a specific fixed cache block location. The placement is done based on a fixed relation between the incoming block number, i, the cache block number, j, and the number of cache blocks, N:

CHAPTER 5
Input–Output Design and Organization

Having considered the fundamental concepts related to instruction set design, assembly language programming, processor design, and memory design, we now turn our attention to the issues related to input – output (I/O) design and organization. It should be emphasized at the outset that I/O plays a crucial role in any modern computer system. Therefore, a clear understanding and appreciation of the fundamentals of I/O operations, devices, and interfaces are of great importance.

Input – output (I/O) devices vary substantially in their characteristics. One distinguishing factor among input devices (and also among output devices) is their data processing rate, defined as the average number of characters that can be processed by a device per second. For example, while the data processing rate of an input device such as the keyboard is about 10 characters (bytes)/second, a scanner can send data at a rate of about 200,000 characters/second. Similarly, while a laser printer can output data at a rate of about100,000 characters/second, a graphic display can output data at a rate of about 30,000,000 characters/second.

Striking a character on the keyboard of a computer will cause a character (in the form of an ASCII code) to be sent to the computer. The amount of time passed before the next character is sent to the computer will depend on the skill of the user and even sometimes on his/her speed of thinking. It is often the case that the user knows what he/she wants to input, but sometimes they need to think before touching the next button on the keyboard. Therefore, input from a keyboard is slow and burst in nature and it will be a waste of time for the computer to spend its valuable time waiting for input from slow input devices. A mechanism is therefore needed whereby a device will have to interrupt the processor asking for attention whenever it is ready. This is called interrupt-driven communication between the computer and I/O devices (see Section 8.3).

Consider the case of a disk. A typical disk should be capable of transferring data at rates exceeding several million bytes/second.

It would be a waste of time to transfer data byte by byte or even word by word. Therefore, it is always the case that data is transferred in the form of blocks, that is, entire programs. It is also necessary to provide a mechanism that allows a disk to transfer this huge volume of data without the intervention of the CPU. This will allow the CPU to perform other useful operation(s) while a huge amount of data is being transferred between the disk and the memory.

Basic Concepts

Figure 8.1 shows a simple arrangement for connecting the processor and the memory in a given computer system to an input device, for example, a keyboard and an output device such as a graphic display. A single bus consisting of the required address, data, and control lines is used to connect the system's components in Figure 8.1.

We are here concerned with the way the processor and the I/O devices exchange data. It has been indicated in the introduction part that there exists a big difference in the rate at which a processor can process information and those of input and output devices. One simple way to accommodate this speed difference is to have the input device, for example, a keyboard, deposit the character struck by the user in a register (input register), which indicates the avail-ability of that character to the processor. When the input character has been taken by the processor, this will be indicated to the input device in order to proceed and input the next character, and so on. Similarly, when the processor has a character to output (display), it deposits it in a specific register dedicated for communication with the graphic display (output register). When the character has been taken by the graphic display, this will be indicated to the processor such that it can proceed and output the next character, and so on. This simple way of communication between the processor and I/O devices, called I/O protocol, requires the availability of the input and output registers. In a typical computer system, there is a number of input registers, each belonging to a specific input device. There is also a number of output registers,

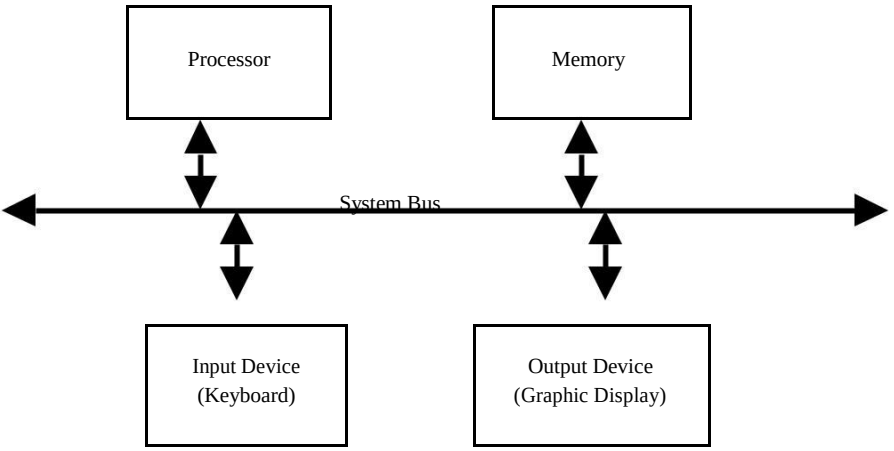

Figure: 8.1 A single bus system

each belonging to a specific output device. In addition, a mechanism according to which the processor can address those input and output registers must be adopted. More than one arrangement exists to satisfy the abovementioned requirements. Among these, two particular methods are explained below.

In the first arrangement, I/O devices are assigned particular addresses, isolated from the address space assigned to the memory. The execution of an input instruc-tion at an input device address will cause the character stored in the input register of that device to be transferred to a specific register in the CPU. Similarly, the execution of an output instruction at an output device address will cause the char-acter stored in a specific register in the CPU to be transferred to the output register of that output device. This arrangement, called shared I/O, is shown schematically in Figure 8.2. In this case, the address and data lines from the CPU can be shared between the memory and the I/O devices. A separate control line will have to be used. This is because of the need for executing input and output instructions. In a typical computer system, there exists more than one input and more than one output device. Therefore, there is a need to have address decoder circuitry for device identification. There is also a need for status registers for each input and output device. The status of an input device, whether it is ready to send data to the processor,

should be stored in the status register of that device. Similarly, the status of an output device, whether it is ready to receive data from the processor, should be stored in the status register of that device. Input (output) registers, status registers, and address decoder circuitry represent the main components of an I/O interface (module).

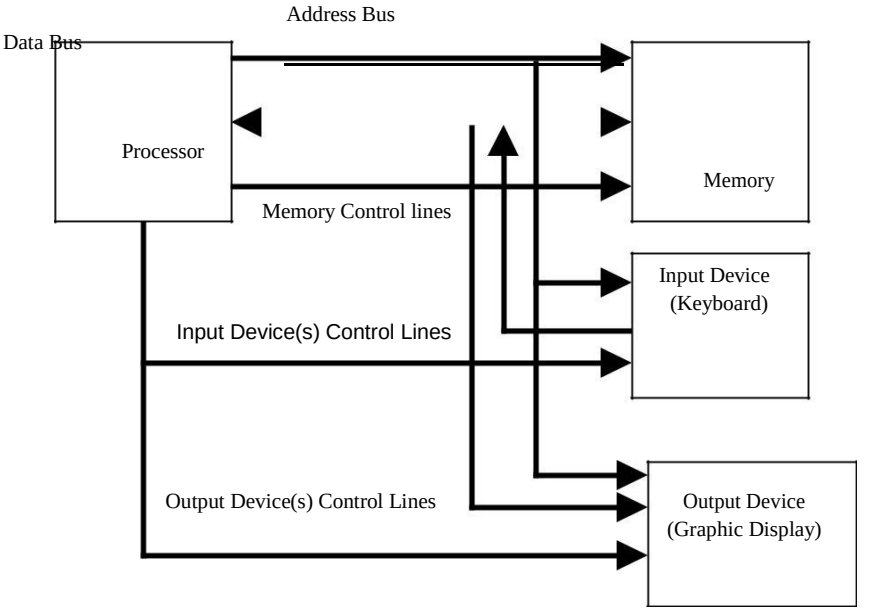

Figure 8.2 Shared I/O arrangement

The main advantage of the shared I/O arrangement is the separation between the memory address space and that of the I/O devices. Its main disadvantage is the need to have special input and output instructions in the processor instruction set. The shared I/O arrangement is mostly adopted by Intel.

The second possible I/O arrangement is to deal with input and output registers as if they are regular memory locations. In this case, a read operation from the address corresponding to the input register of an input device, for example, Read Device 6, is equivalent to performing an input operation from the input register in Device #6. Similarly, a write operation to the address corresponding to the

output register of an output device, for example, Write Device 9, is equivalent to performing an output operation into the output register in Device #9. This arrangement is called memory-mapped I/O. It is shown in Figure 8.3.

The main advantage of the memory-mapped I/O is the use of the read and write instructions of the processor to perform the input and output operations, respectively. It eliminates the need for introducing special I/O instructions. The main disadvantage of the memory-mapped I/O is the need to reserve a certain part of the memory address space for addressing I/O devices, that is, a reduction in the available memory address space. The memory-mapped I/O has been mostly adopted by Motorola.

Interrupt-Driven I/O

It is often necessary to have the normal flow of a program interrupted, for example, to react to abnormal events, such as power failure. An interrupt can also be used to acknowledge the completion of a particular course of action, such as a printer indicating to the computer that it has completed printing the character(s) in its input register and that it is ready to receive other character(s). An interrupt can also be used in time-sharing systems to allocate CPU time among different programs. The instruction sets of modern CPUs often include instruction(s) that mimic the actions of the hardware interrupts.

When the CPU is interrupted, it is required to discontinue its current activity, attend to the interrupting condition (serve the interrupt), and then resume its activity from wherever it stopped. Discontinuity of the processor's current activity requires finishing executing the current instruction, saving the processor status (mostly in the form of pushing register values onto a stack), and transferring control (jump) to what is called the interrupt service routine (ISR). The service offered to an interrupt will depend on the source of the interrupt. For example, if the interrupt is due to power failure, then the action taken will be to save the values of all processor registers and pointers such that resumption of correct operation can be guaranteed upon power return. In the case of an I/O interrupt, serving an interrupt means to perform the required data transfer. Upon finishing serving an interrupt, the processor should restore the original status by popping the relevant values from the stack. Once the processor returns to the normal state, it can enable sources of interrupt again.

One important point that was overlooked in the above scenario is the issue of serving multiple interrupts, for example, the occurrence of yet another interrupt while the processor is currently serving an interrupt. Response to the new interrupt will depend upon the priority of the newly arrived interrupt with respect to that of the interrupt being currently served. If the newly arrived interrupt has priority less than or equal to that of the currently served one, then it can wait until the processor finishes serving the current interrupt. If, on the other hand, the newly arrived interrupt has priority higher than that of the currently served interrupt, for example, power failure interrupt

occurring while serving an I/O interrupt, then the processor will have to push its status onto the stack and serve the higher priority interrupt. Correct handling of multiple interrupts in terms of storing and restoring the correct processor status is guaranteed due to the way the push and pop operations are performed.
For example, to serve the first interrupt, STATUS 1 will be pushed onto the stack. Upon receiving the second interrupt, STATUS 2 will be pushed onto the stack. Upon serving the second interrupt, STATUS 2 will be popped out of the stack and upon serving the first interrupt, STATUS 1 will be popped out of the stack.
It is possible to have the interrupting device identify itself to the processor by sending a code following the interrupt request. The code sent by a given I/O device can represent its I/O address or the memory address location of the start of the ISR for that device. This scheme is called vectored interrupt.

Interrupt Hardware

In the above discussion, we have assumed that the processor has recognized the occurrence of an interrupt before proceeding to serve it. Computers are provided with interrupt hardware capability in the form of specialized interrupt lines to the processor. These lines are used to send interrupt signals to the processor. In the case of I/O, there exists more than one I/O device. The processor should be provided with a mechanism that enables it to handle simultaneous interrupt requests and to recognize the interrupting device. Two basic schemes can be implemented to achieve this task. The first scheme is called daisy chain bus arbitration (DCBA) and the second is called independent source bus arbitration (ISBA).

Interrupt in Operating Systems

When an interrupt occurs, the operating system gains control. The operating system saves the state of the interrupted process, analyzes the interrupt, and passes control to the appropriate routine to handle the interrupt. There are several

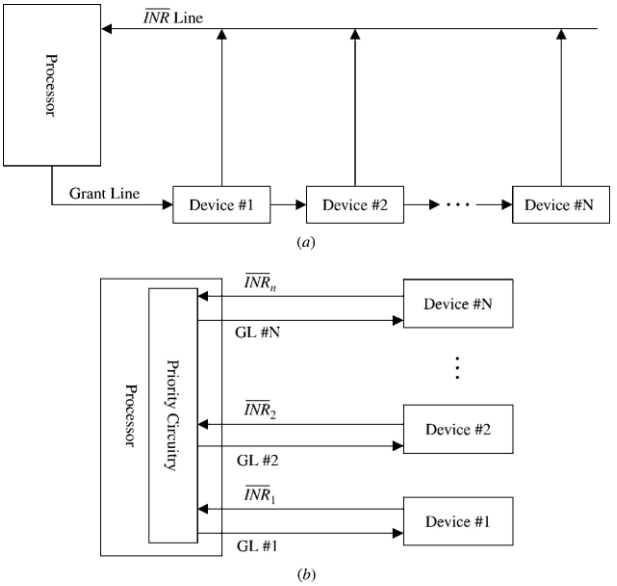

Figure 8.6 Interrupt hardware schemes. (a) Daisy chain interrupt arrangement (b) Independent interrupt arrangement

types of interrupts, including I/O interrupts. An I/O interrupt notifies the operating system that an I/O device has completed or suspended its operation and needs some service from the CPU. To process an interrupt, the context of the current process must be saved and the interrupt handling routine must be invoked. This process is called context switching. A process context has two parts: processor context and memory context. The processor context is the state of the CPU's registers including program counter (PC), program status words (PSWs), and other registers. The memory context is the state of the program's memory including the program and data. The interrupt handler is a routine that processes each different type of interrupt.

The operating system must provide programs with save area for their contexts. It also must provide an organized way for allocating and de-allocating memory for the interrupted process. When the interrupt handling routine finishes processing the interrupt, the CPU is dispatched to either the interrupted process, or to the highest priority ready process. This will depend on whether the interrupted process is preemptive or non-preemptive. If the process is non-preemptive, it gets the CPU again. First the con-text must be restored, then control is returned to the interrupts process.

Figure 8.7 Layered I/O software

Figure 8.7 shows the layers of software involved in I/O operations. First, the pro-gram issues an I/O request via an I/O call. The request is passed through to the I/O device. When the device completes the I/O, an interrupt is sent and the interrupt handler is invoked. Eventually, control is relinquished back to the process that initiated the I/O.

Direct Memory Access (DMA)

The main idea of direct memory access (DMA) is to enable peripheral devices to cut out the "middle man" role of the CPU in data transfer. It allows peripheral devices to transfer data directly from and to memory without the intervention of the CPU. Having peripheral devices access memory directly would allow the CPU to do other work, which would lead to improved performance, especially in the cases of large transfers.

The DMA controller is a piece of hardware that controls one or more peripheral devices. It allows devices to transfer data to or from the system's memory without the help of the processor. In a typical DMA transfer, some event notifies the DMA controller that data needs to be transferred to or from memory. Both the DMA and CPU use memory bus and only one or the other can use the memory at the same time. The DMA controller then sends a request to the CPU asking its permission to use the bus. The CPU returns an acknowledgment to the DMA controller granting it bus access. The

DMA can now take control of the bus to independently conduct memory transfer. When the transfer is complete the DMA relinquishes its control of the bus to the CPU. Processors that support DMA provide one or more input signals that the bus requester can assert to gain control of the bus and one or more output signals that the CPU asserts to indicate it has relinquished the bus. Figure 8.10 shows how the DMA controller shares the CPU's memory bus.

Figure 8.1:DMA controller shares the CPU's memory bus

Direct memory access controllers require initialization by the CPU. Typical setup parameters include the address of the source area, the address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete. A DMA controller has an address register, a word count register, and a control register. The address register contains an address that specifies the memory location of the data to be transferred. It is typically possible to have the DMA controller automatically increment the address register after each word transfer, so that the next transfer will be from the next memory location. The word count register holds the number of words to be transferred. The word count is decremented by one after each word transfer. The control register specifies the transfer mode.

Direct memory access data transfer can be performed in burst mode or single-cycle mode.

In burst mode, the DMA controller keeps control of the bus until all the data has been transferred to (from) memory from (to) the peripheral device. This mode of transfer is needed for fast devices where data transfer cannot be stopped until the entire transfer is done. In single-cycle mode (cycle stealing), the DMA controller relinquishes the bus after each transfer of one data word. This minimizes the amount of time that the DMA controller keeps the CPU from controlling the bus, but it requires that the bus request/acknowledge sequence be performed for every single transfer. This overhead can result in a degradation of the performance. The single-cycle mode is preferred if the system cannot tolerate more than a few cycles of added interrupt latency or if the peripheral devices can buffer very large amounts of data, causing the DMA controller to tie up the bus for an excessive amount of time.

The following steps summarize the DMA operations:

DMA controller initiates data transfer.

Data is moved (increasing the address in memory, and reducing the count of words to be moved).

When word count reaches zero, the DMA informs the CPU of the termination by means of an interrupt.

The CPU regains access to the memory bus.

A DMA controller may have multiple channels. Each channel has associated with it an address register and a count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. While the transfer is taking place, the CPU is free to do other things. When the transfer is complete, the CPU is interrupted.

Direct memory access channels cannot be shared between device drivers. A device driver must be able to determine which DMA channel to use. Some devices have a fixed DMA channel, while others are more flexible, where the device driver can simply pick a free DMA channel to use.

Linux tracks the usage of the DMA channels using a vector of dma_chan data structures (one per DMA channel). The dma_chan data structure contains just two fields, a pointer to a string describing the owner of the DMA channel and a flag indicating if the DMA channel is allocated or not.

BUSES

A bus in computer terminology represents a physical connection used to carry a signal from one point to another. The signal carried by a bus may represent address, data, control signal, or power. Typically, a bus consists of a number of connections running together. Each connection is called a bus line. A bus line is normally identified by a number. Related groups of bus lines are usually identified by a name. For example, the group of bus lines 1 to 16 in a given computer system may be used to carry the address of memory locations, and therefore are identified as address lines. Depending on the signal carried, there exist at least four types of buses: address, data, control, and power buses. Data buses carry data, control buses carry control signals, and power buses carry the power-supply/ground voltage. The size (number of lines) of the address, data, and control bus varies from one system to another. Consider, for example, the bus connecting a CPU and memory in a given system, called the CPU bus. The size of the memory in that system is 512M-word and each word is 32 bits. In such system, the size of the address bus should be $\log_2(512\ 2^{20})$ ¼ 29 lines, the size of the data bus should be 32 lines, and at least one control line (R W) should exist in that system.

In addition to carrying control signals, a control bus can carry timing signals. These are signals used to determine the exact timing for data transfer to and from a bus; that is, they determine when a given computer system component, such as the processor, memory, or I/O devices, can place data on the bus and when they can receive data from the bus. A bus can be synchronous if data transfer over the bus is controlled by a bus clock. The clock acts as the timing reference for all bus signals. A bus is asynchronous if data transfer over the bus is based on the avail-ability of the data and not on a clock signal.

Data is transferred over an asynchronous bus using a technique called handshaking. The operations of synchronous and asynchronous buses are explained below.

To understand the difference between synchronous and asynchronous, let us consider the case when a master such as a CPU or DMA is the source of data to be transferred to a slave such as an I/O device. The following is a sequence of events involving the master and slave:

Master: send request to use the bus

Master: request is granted and bus is allocated to master

Master: place address/data on bus

Slave: slave is selected

Master: signal data transfer

Slave: take data

Master: free the bus

Synchronous Buses

In synchronous buses, the steps of data transfer take place at fixed clock cycles. Everything is synchronized to bus clock and clock signals are made available to both master and slave. The bus clock is a square wave signal. A cycle starts at one rising edge of the clock and ends at the next rising edge, which is the beginning of the next cycle. A transfer may take multiple bus cycles depending on the speed parameters of the bus and the two ends of the transfer.

One scenario would be that on the first clock cycle, the master puts an address on the address bus, puts data on the data bus, and asserts the appropriate control lines. Slave recognizes its address on the address bus on the first cycle and reads the new value from the bus in the second cycle.

Synchronous buses are simple and easily implemented. However, when connecting devices with varying speeds to a synchronous bus, the slowest device will deter-mine the speed of the bus. Also, the synchronous bus length could be limited to avoid clock-skewing problems.

Asynchronous Buses

There are no fixed clock cycles in asynchronous buses. Handshaking is used instead. Figure 8.11 shows the handshaking protocol. The master asserts the data-ready line

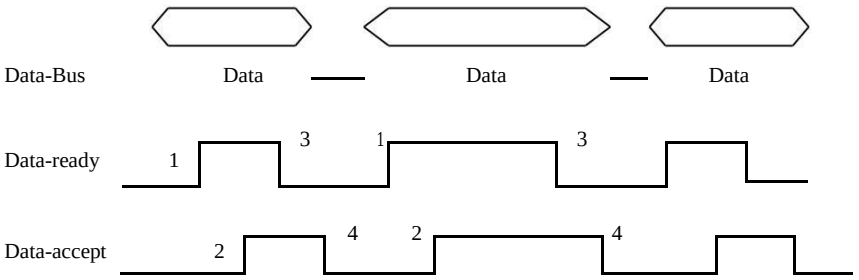

Figure 8.11 Asynchronous bus timing using handshaking protocol

(point 1 in the figure) until it sees a data-accept signal. When the slave sees a data-ready signal, it will assert the data-accept line (point 2 in the figure). The rising of the data-accept line will trigger the falling of the data-ready line and the removal of data from the bus. The falling of the data-ready line (point 3 in the figure) will trigger the falling of the data-accept line (point 4 in the figure). This handshaking, which is called fully interlocked, is repeated until the data is completely transferred. Asynchronous bus is appropriate for different speed devices.

INPUT –OUTPUT INTERFACES

An interface is a data path between two separate devices in a computer system. Interface to buses can be classified based on the number of bits that are transmitted at a given time to serial versus parallel ports. In a serial port, only 1 bit of data is transferred at a time. Mice and modems are usually connected to serial ports. A parallel port allows more than 1 bit of data to be processed at once. Printers are the most common peripheral devices connected to parallel ports. Table 8.4 shows a summary of the variety of buses and interfaces used in personal computers.

TABLE 8.4 Descriptions of Buses and Interfaces Used in Personal Computers

Bus/Interface	Description
PS/2	A type of port (or interface) that can be used to connect mice and keyboards to the computer. The PS/2 port is sometimes called the mouse port.
Industry standard architecture (ISA)	ISA was originally an 8-bit bus and later expanded to a 16-bit bus in 1984. In 1993, Intel and Microsoft introduced a plug and play ISA bus that allowed the computer to automatically detect and set up computer ISA peripherals such as a modem or sound card.
Extended industry standard architecture (EISA)	EISA is an enhanced form of ISA, which allows for 32-bit data transfers, while maintaining support for 8- and 16-bit expansion boards. However, its bus speed, like ISA, is only 8 MHz. EISA is not widely used, due to its high cost and complicated nature.
Micro channel architecture (MCA)	MCA was introduced by IBM in 1987. It offered several additional features over the ISA such as a 32-bit bus, automatically configured cards and bus mastering for greater efficiency. It is slightly superior to EISA, but not many expansion boards were ever made to fit MCA specifications.
VESA (Video electronics standards association) local bus (VLB)	The VESA, a nonprofit organization founded by NEC, released the VLB in 1992. It is a 32-bit bus that had direct access to the system memory at the speed of the processor, commonly the 486 CPU (33/40 MHz). VLB 2.0 was later released in 1994 and had a 64-bit bus and a bus speed of 50 MHz.
Peripheral component interconnect (PCI)	PCI was introduced by Intel in 1992, revised in 1993 to version 2.0, and later revised in 1995 to PCI 2.1. It is a 32-bit bus that is also available as a 64-bit bus today. Many modern expansion boards are connected to PCI slots.
Advanced graphic port (AGP)	AGP was introduced by Intel in 1997. AGP is a 32-bit bus designed for the high demands of 3D graphics. AGP has a direct line to memory, which allows 3D elements to be stored in the system memory instead of the video memory. AGP is geared towards data-intensive graphics cards, such as 3D accelerators; its design allows for data throughput at rates of 266 MB/s.

TABLE 8.4 Continued

Bus/Interface	Description
Universal serial bus (USB)	USB is an external bus developed by Intel, Compaq, DEC, IBM, Microsoft, NEC and Northern Telcom. It was released in 1996 with the Intel 430HX Triton II Mother Board. USB has the capability of transferring 12 Mbps, supporting up to 127 devices. Many devices can be connected to USB ports, which support plug and play.
FireWire (IEEE 1394)	FireWire is a type of external bus, which supports very fast transfer rates: 400 Mbps. Because of this, FireWire is suitable for connecting video devices, such as VCRs, to the computer.
Small computer system interface (SCSI)	SCSI is a type of parallel interface that is commonly used for mass storage devices. SCSI can transfer data at rates of 4 MB/s; in addition, there are several varieties of SCSI that support higher speeds: Fast SCSI (10 MB/s), Ultra SCSI and Fast Wide SCSI (20 MB/s), as well as Ultra Wide SCSI (40 MB/s).
Integrated drive electronics (IDE)	IDE is a commonly used interface for hard disk drives and CD-ROM drives. It is less expensive than SCSI, but offers slightly less in terms of performance.
Enhanced integrated drive electronics (EIDE)	EIDE is an improved version of IDE, which offers better performance than standard SCSI. It offers transfer rates between 4 and 16.6 MB/s.
PCI-X	PCI-X is a high performance bus that is designed to meet the increased I/O demands of technologies such as Fibre Channel, Gigabit Ethernet, and Ultra3 SCSI.
Communication and network riser (CNR)	CNR was introduced by Intel in 2000. It is a specification that supports audio, modem USB and local area networking interfaces of core logic chipsets.

SUMMARY

One of the major features in a computer system is its ability to exchange data with other devices and to allow the user to interact with the system. This chapter focused on the I/O system and the way the processor and the I/O devices exchange data in a computer system. The chapter described three ways of organizing I/O: programmed I/O, interrupt-driven I/O, and DMA. In programmed I/O, the CPU handles the transfers, which take place between registers and the devices. In interrupt-driven I/O, CPU handles data transfers and an I/O module is running concurrently. In DMA, data are transferred between memory and I/O devices without intervention of the CPU. We also studied two methods for synchronization: polling and interrupts. In polling, the processor polls the device while waiting for I/O to complete. Clearly processor cycles

are wasted in this method. Using interrupts, processors are free to switch to other tasks during I/O. Devices assert interrupts when I/O is complete. Interrupts incurs some delay penalty. Two examples of interrupt handling were covered: 80 86 family and ARM. The chapter also covered buses and interfaces. A wide variety of interfaces and buses used in personal computers are summarized.

CHAPTER 6
Pipelining Design Techniques

There exist two basic techniques to increase the instruction execution rate of a processor. These are to increase the clock rate, thus decreasing the instruction execution time, or alternatively to increase the number of instructions that can be executed simultaneously. Pipelining and instruction-level parallelism are examples of the latter technique. Pipelining owes its origin to car assembly lines. The idea is to have more than one instruction being processed by the processor at the same time. Similar to the assembly line, the success of a pipeline depends upon dividing the execution of an instruction among a number of subunits (stages), each performing part of the required operations. A possible division is to consider instruction fetch (F), instruction decode (D), operand fetch (F), instruction execution (E), and store of results (S) as the subtasks needed for the execution of an instruction. In this case, it is possible to have up to five instructions in the pipeline at the same time, thus reducing instruction execution latency. In this Chapter, we discuss the basic concepts involved in designing instruction pipelines. Performance measures of a pipeline are introduced. The main issues contributing to instruction pipeline hazards are discussed and some possible solutions are introduced. In addition, we introduce the concept of arithmetic pipelining together with the problems involved in designing such a pipeline. Our coverage concludes with a review of a recent pipeline processor.

GENERAL CONCEPTS

Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in sequence. Each subtask is performed by a given functional unit. The units are connected in a serial fashion and all of them operate simultaneously. The use of pipelining improves the performance compared to the traditional sequential execution of tasks. Figure 9.1 shows an illustration of the basic difference between executing four subtasks of a given instruction (in this case fetching F, decoding D, execution E, and writing the results W) using pipelining and sequential processing.

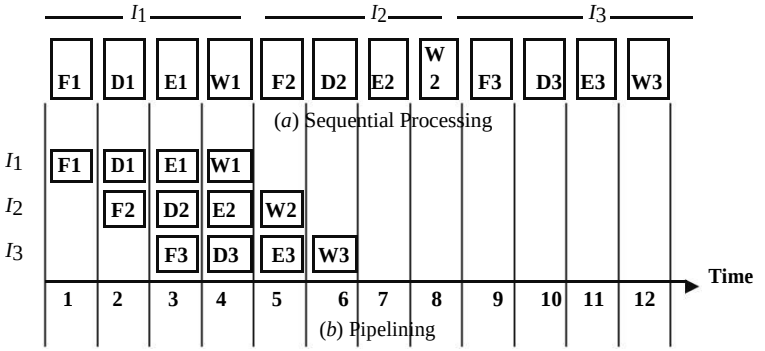

Figure 9.1 Pipelining versus sequential processing

It is clear from the figure that the total time required to process three instructions (I_1, I_2, I_3) is only six time units if four-stage pipelining is used as compared to 12 time units if sequential processing is used. A possible saving of up to 50% in the execution time of these three instructions is obtained. In order to formulate some performance measures for the goodness of a pipeline in processing a series of tasks, a space time chart (called the Gantt's chart) is used. The chart shows the succession of the subtasks in the pipe with respect to time. Figure 9.2 shows a Gantt's chart. In this chart, the vertical axis represents the subunits (four in this case) and the horizontal axis represents time (measured in terms of the time unit required for each unit to perform its task). In developing the Gantt's chart, we assume that the time (T) taken by each subunit to perform its task is the same; we call this the unit time.

As can be seen from the figure, 13 time units are needed to finish executing 10 instructions (I_1 to I_{10}). This is to be compared to 40 time units if sequential processing is used (ten instructions each requiring four time units).

In the following analysis, we provide three performance measures for the goodness of a pipeline. These are the Speed-up S(n), Throughput U(n), and Efficiency E(n). It should be noted that in this analysis we assume that the unit time T ¼ t units.

Speed-up S(n) Consider the execution of m tasks (instructions) using n-stages (units) pipeline. As can be seen, n þ m 1 time units are required

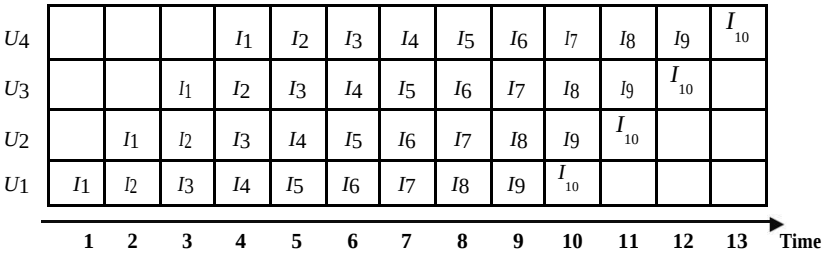

Figure 9.2 The space – time chart (Gantt chart)

INSTRUCTION PIPELINE

The simple analysis made in Section 9.1 ignores an important aspect that can affect the performance of a pipeline, that is, pipeline stall. A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle. Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units. Figure 9.3 illustrates the effect of having instruction I_2 incurring a cache miss (assuming the execution of ten instructions I_1 to I_{10}).

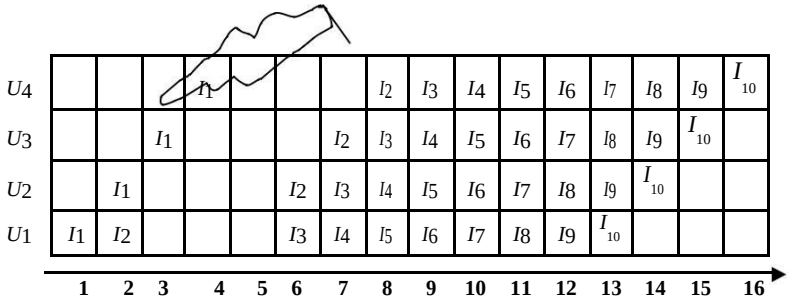

Figure 9.3 Effect of a cache miss on the pipeline

The figure shows that due to the extra time units needed for instruction I_2 to be fetched, the pipeline stalls, that is, fetching of instruction I_3 and subsequent instructions are delayed. Such situations create what is known as pipeline bubble (or pipe-line hazards). The creation of a pipeline bubble leads to wasted unit times, thus leading to an overall increase in the number of time units needed to finish executing a given number of instructions. The number of time units needed to execute the 10 instructions shown in Figure 9.3 is now 16 time units, compared to 13 time units if there were no cache misses.

Pipeline hazards can take place for a number of other reasons. Among these are instruction dependency and data dependency. These are explained below.

Methods Used to Prevent Fetching the Wrong Instruction or Operand

Use of NOP (No Operation) This method can be used in order to prevent the fetching of the wrong instruction, in case of instruction dependency, or fetching the wrong operand, in case of data dependency. Recall Example 1. In that example, the execution of a sequence of ten instructions I_1 –I_{10} on a pipeline consisting of four pipeline stages: IF, ID, IE, and IS were considered. In order to show the execution of these instructions in the pipeline, we have assumed that when the branch instruction is fetched, the pipeline stalls until the result of executing the branch instruction is stored. This assumption was needed in order to prevent fetching the wrong instruction after fetching the branch instruction. In real-life situations, a mechanism is needed to guarantee fetching the appropriate instruction at the appropriate time. Insertion of "NOP" instructions will help carrying out this task. A "NOP" is an instruction that has no effect on the status of the processor.

CHAPTER 7
Reduced Instruction Set Computers (RISCs)

RISC/CISC EVOLUTION CYCLE

The term RISCs stands for Reduced Instruction Set Computers. It was originally introduced as a notion to mean architectures that can execute as fast as one instruction per clock cycle. RISC started as a notion in the mid-1970s and has eventually led to the development of the first RISC machine, the IBM 801 minicomputer. The launching of the RISC notion announces the start of a new paradigm in the design of computer architectures. This paradigm promotes simplicity in computer architecture design. In particular, it calls for going back to basics rather than providing extra hardware support for high-level languages. This paradigm shift relates to what is known as the semantic gap, a measure of the difference between the operations provided in the high-level languages (HLLs) and those provided in computer architectures.

It is recognized that the wider the semantic gap, the larger the number of undesirable consequences. These include (a) execution inefficiency, (b) excessive machine pro-gram size, and (c) increased compiler complexity. Because of these expected conse-quences, the conventional response of computer architects has been to add layers of complexity to newer architectures.

These include increasing the number and complexity of instructions together with increasing the number of addressing modes. The architectures resulting from the adoption of this "add more complexity" are now known as Complex Instruction Set Computers (CISCs). However, it soon became apparent that a complex instruction set has a number of disadvantages. These include a complex instruction decoding scheme, an increased size of the control unit, and increased logic delays. These drawbacks prompted a team of computer architects to adopt the principle of "less is actually more." A number of studies were then conducted to investigate the impact of complexity on performance. These are discussed below.

RISCs DESIGN PRINCIPLES

A computer with the minimum number of instructions has the disadvantage that a large number of instructions will have to be executed in realizing even a simple function. This will result in a speed disadvantage. On the other hand, a computer with an inflated number of instructions has the disadvantage of complex decoding and hence a speed disadvantage. It is then natural to believe that a computer with a carefully selected reduced set of instructions should strike a balance between the above two design alternatives. The question then becomes what constitutes a carefully selected reduced set of instructions? In order to arrive at an answer to this question, it is necessary to conduct in-depth studies on a number of aspects of computation. These aspects should include (a) operations that are most frequently performed during execution of typical (benchmark) programs, (b) operations that are most time consuming, and (c) the type of operands that are most frequently used.

A number of early studies were conducted in order to find out the typical breakdown of operations that are performed in executing benchmark programs. The estimated distribution of operations is shown in Table 10.1.

A careful look at the estimated percentage of operations performed reveals that assignment statements, conditional branches, and procedure calls constitute about 90% of the total operations performed, while other operations, however complex they may be, make up the remaining 10%.

TABLE 10.1 Estimated Distribution of Operations

Operations	Estimated percentage
Assignment statements	35
Loops	5
Procedure calls	15
Conditional branches	40
Unconditional branches	3
Others	2

In addition to the above findings, studies on time – performance characteristics of operations revealed that among all operations, procedure calls/return are the most time-consuming. With regards to the type of operands used during typical computation, it was noticed that the majority of references (no less than 60%) are made to simple scalar variables and that no less than 80% of scalars are local variables (to procedures).

The above observations about typical program behavior have led to the following conclusions:

Simple movement of data (represented by assignment statements), rather than complex operations, are substantial and should be optimized.

Conditional branches are predominant and therefore careful attention should be paid to the sequencing of instructions. This is particularly true when it is known that pipelining is indispensable to use.

Procedure calls/return are the most time-consuming operations and therefore a mechanism should be devised to make the communication of parameters among the calling and the called procedures cause the least number of instructions to execute.

A prime candidate for optimization is the mechanism for storing and accessing local scalar variables.

The above conclusions have led to the argument that instead of bringing the instruction set architecture closer to HLLs, it should be more appropriate to rather optimize the performance of the most time-consuming features of typical HLL programs. This is obviously a call for making the architecture simpler rather than complex. Remember that complex operations such as long division represent only a small portion (less than 2%) of the operations performed during a typical computation. One then should ask the question: how can we achieve that? The answer is by (a) keeping the most frequently accessed operands in CPU registers and (b) minimizing the register-to-memory operations.

The above two principles can be achieved using the following mechanisms:

Use a large number of registers to optimize operand referencing and reduce the processor memory traffic.

Optimize the design of instruction pipelines such that minimum compiler code generation can be achieved.

Use a simplified instruction set and leave out those complex and unnecessary instructions.

The following two approaches were identified to implement the above three mechanisms.

Software approach. Use the compiler to maximize register usage by allocating registers to those variables that are used the most in a given time period (this is the philosophy adopted in the Stanford MIPs machine).

Hardware approach. Use ample CPU registers so that more variables can be held in registers for larger periods of time (this is the philosophy adopted in the Berkeley RISC machine). The hardware approach necessitates the use of a new register organization, called overlapped register window.

RISCs VERSUS CISC

The choice of RISC versus CISC depends totally on the factors that must be considered by a computer designer. These factors include size, complexity, and speed. A RISC architecture has to execute more instructions to perform the same function performed by a CISC architecture. To compensate for this drawback, RISC architectures must use the chip area saved by not using complex instruction decoders in providing a large number of CPU registers, additional execution units, and instruction caches. The use of these resources leads to a reduction in the traffic between the processor and the memory. On the other hand, a CISC architecture with a richer and more complex instructions, will require a smaller number of instructions than its RISC counterpart.

However, a CISC architecture requires a complex decoding scheme and hence is subject to logic delays. It is therefore reason-able to consider that the RISC and CISC paradigms differ primarily in the strategy used to trade off different design factors. There is very little reason to believe that an idea that improves performance for a RISC architecture will fail to do the same thing in a CISC architecture and vice versa. For example, one key issue in RISC development is the use of optimizing the compiler to reduce the complexity of the hardware and to optimize the use of CPU registers. These same ideas should be applicable to CISC compilers. Increasing

TABLE 10.3 RISC Versus CISC Performance

Application	MIPS CPI (RISC)	VAX CPI (CISC)	CPI ratio	Instruction Ratio
Spice 2G6	1.80	8.02	4.44	2.48
Matrix300	3.06	13.81	4.51	2.37
Nasa 7	3.01	14.95	4.97	2.10
Espresso	1.06	5.40	5.09	1.70

the number of CPU registers could very much improve the performance of a CISC machine. This could be the reason behind not finding a pure commercially available RISC (or CISC) machine. It is not unusual to see a RISC machine with complex floating-point instructions (see the details of the SPARC architecture in the next sec-tion). It is equally expected to see CISC machines making use of the register win-dows RISC idea. In fact there have been studies indicating that a CISC machine such as the Motorola 680xx with a register window will achieve a 2 to 4 times decrease in the memory traffic. This is the same factor that can be achieved by a RISC architecture, such as the Berkeley RISC, due to the use of a register window.

It should, however, be noted that most processor developers (except for Intel and its associates) have opted for RISC processors. Computer system manufacturers such as Sun Microsystems are using RISC processors in their products. However, for compatibility with the PC-based market, such companies are still producing CISC-based products.

Tables 10.3 and 10.4 show a limited comparison between an example RISC and CISC machine in terms of performance and characteristics, respectively.

An elaborate comparison among a number of commercially available RISC and CISC machines is shown in Table 10.5.

It is worth mentioning at this point that the following set of common character-istics among RISC machines is observed:

Fixed-length instructions

Limited number of instructions (128 or less)

Limited set of simple addressing modes (minimum of two: indexed and PC-relative)

All operations are performed on registers; no memory operations

Only two memory operations: Load and Store

TABLE 10.4 RISC Versus CISC Characteristics

Characteristic	VAX-11 (CISC)	Berkeley RISC-1 (RISC)
Number of instructions	303	31
Instruction size (bits)	16-456	32
Addressing modes	22	3
No. general purpose registers	16	138

Pipelined instruction execution

Large number of general-purpose registers or the use of advanced compiler technology to optimize register usage

One instruction per clock cycle

Hardwired control unit design rather than microprogramming

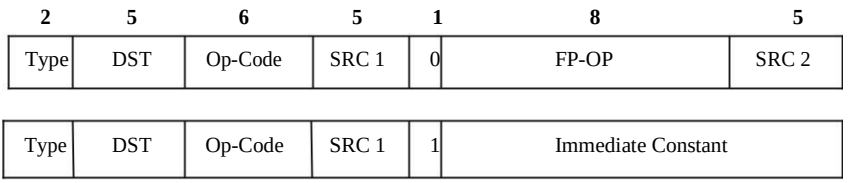

Figure 10.2 Three operand instructions formats used in RISC

3. Branch & Call: JMPX COND, (R_X)S; PC R_X þ S; where COND is a condition
4. Special Instructions: GETPSW R_d; R_d PSW

All arithmetic and logical instructions have three operands and have the form Destination : ¼ source1 op source2 (Fig. 10.2). The LOAD and STORE instructions may use

either of the indicated formats with DST being the register to be loaded or stored. The low order 19 bits of the instructions are used to determine the effective address.

Instructions load and store 8-, 16-, 32-, and 64-bit quantities into 32-bit registers. Two methods are provided for calling procedures. The CALL instruction uses a 30-bit PC relative offset (Fig. 10.3).

The JMP instruction uses any of the instruction formats used for arithmetic and logical operations and allows the return address to be put in any register.
RISC uses a three-address instruction format with the availability of some two-and one-address instructions. There are only two addressing modes. These are indexed mode and PC relative modes. The indexed mode can be used to synthesize three other modes. These are base-absolute (direct), register indirect, and indexed for linear byte array modes. RISC uses a static two-stage pipeline: fetch and execute.

The floating-point unit (FPU) contains thirty-two 32-bit registers to hold 32 single precision (32-bit) floating-point operands, 16 double-precision (64-bit) operands, or eight extended-precision (128-bit) operands. The FPU can execute about 20 floating-point instructions most of them in single-, double-, or extended-precision using the first instruction format used for arithmetic. In addition to instructions for loading and storing FPUs registers, the CPU can also test FPUs registers and branch conditionally on results. RISC employs a conventional MMU supporting a single paged 32-bit address space. The RISC four-bus organization is shown in Figure 10.4.