P.S.R. ENGINEERING COLLEGE (AUTONOMOUS)

(An Autonomous Institution Affiliated to Anna University, Chennai)
Approved by AICTE, New Delhi, Accredited by NBA & NAAC with “A+” Grade,
Recognized under 12(B) of the UGC Act, 1956.
An ISO 9001:2015 Certified Institution
Sevalpatti (PO), Sivakasi – 626140, Tamilnadu

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT NOTES

COURSE CODE	: **231CS31**
COURSE NAME	: **COMPUTER ORGANIZATION AND ARCHITECTURE**
SEMESTER	: **III**
ACADEMIC YEAR	: **2024 - 25**
COURSE TUTOR	: **Dr.R.NATCHADALINGAM** Professor – CSE DEPT.

231CS31	**COMPUTER ORGANIZATION AND ARCHITECTURE**	**L**	**T**	**P**	**C**
		3	**0**	**0**	**3**
Programme:	B.E. Computer Science and Engineering	**Sem:**	**3**	**Category:**	**PC**
Pre-/Co-requisite:	191EC35 – Digital Electronics and Microprocessors				

Aim:	To understand the organization of a computer, hardware-software interface, and to discuss the basic structure of a digital computer.
Course Outcomes: The students will be able to	
CO1:	Outline the fundamental organization of a computer system.
CO2:	Explain the fundamentals concepts of pipeline processing.
CO3:	Identify the role of the control unit in instruction execution
CO4:	Compare the performance of various memory systems.
CO5:	Illustrate the various types of I/O devices and interfaces.
CO6:	Summarize about the multi-core architecture and processors.

BASIC STRUCTURE OF COMPUTERS	**9**
Functional units -Basic operational concepts - Bus structures - Performance and metrics – Instruction Set Architecture - Addressing modes - RISC - CISC. Fixed point and floating-point operations.	
PROCESSOR AND CONTROL UNIT	**9**
Execution of a complete instruction - Multiple bus organization - Hardwired control – Micro-programmed control - Nano programming. Pipelining – Basic concepts - Hazards - Types - Influence on instruction sets – Data path and control considerations - Performance considerations - Exception handling.	
MEMORY SYSTEMS	**9**
Basic concepts - Semiconductor RAM - ROM - Speed - Size and cost - Cache Memories – Improving cache performance - Virtual memory - Memory management requirements - Associative memories – Performance Considerations.	
I/O SYSTEMS	**9**
Accessing I/O devices - Programmed Input/output - Interrupts – Direct Memory Access - Buses – Interface circuits - Standard I/O Interfaces (PCI, SCSI, USB).	
MULTI-CORE ARCHITECTURES	**9**
Introduction to Multithreading – Software and hardware multithreading – SMT and CMP architectures – Design issues – Case study – Intel Core i7. Heterogeneous multi-core processors – Case study: IBM Cell Processor.	
Total Periods:	**45**
Text Books:	
1. Carl Hamacher, Zvonko Vranesic and Safwat Zaky, "*Computer Organization*", *6/e*, McGraw-Hill Inc, 2012. 2. John L. Hennessey and David A. Patterson, "*Computer Architecture – A quantitative approach*", Morgan Kaufmann / Elsevier Publishers, 5/e, 2011.	

Reference Books:

1. David A. Patterson and John L. Hennessy, "*Computer Organization and Design: The Hardware / Software interface*", 5/e, Elsevier, 2013.
2. William Stallings, "*Computer Organization and Architecture – Designing for Performance*", 8/e, Pearson Education, 2010.
3. John P.Hayes, "*Computer Architecture and Organization*", Indian/e, Tata McGraw Hill, 2017.
4. V.P. Heuring, H.F. Jordan, "*Computer Systems Design and Architecture*", 2/e, Pearson Education, 2004.
5. David E. Culler, Jaswinder Pal Singh, "*Parallel computing architecture: A hardware/software approach*", Morgan Kaufmann /Elsevier Publishers, 1999.
6. https://archive.nptel.ac.in/courses/106/105/106105163/

WEB LINKS:

https://www.youtube.com/playlist?list=PLWPirh4EWFpF0FVeBgL75d1RlASn4sGoz

https://www.youtube.com/watch?v=uHqbE3JiDPY&list=PLWPirh4EWFpF0FVeBgL75d1RlASn4sGoz&index=2

https://www.youtube.com/watch?v=CO0ix61lN8k&list=PLWPirh4EWFpF0FVeBgL75d1RlASn4sGoz&index=4

231CS31	**COMPUTER ORGANIZATION AND ARCHITECTURE**	**L**	**T**	**P**	**C**
		3	**0**	**0**	**3**
Programme:	B.E. Computer Science and Engineering	**Sem:**	**3**	**Category:**	**PC**
Pre-/Co-requisite:	191EC35 – Digital Electronics and Microprocessors				
Aim:	To understand the organization of a computer, hardware-software interface, and to discuss the basic structure of a digital computer.				

Course Outcomes: The students will be able to

CO1:	Outline the fundamental organization of a computer system.
CO2:	Explain the fundamentals concepts of pipeline processing.
CO3:	Identify the role of the control unit in instruction execution
CO4:	Compare the performance of various memory systems.
CO5:	Illustrate the various types of I/O devices and interfaces.
CO6:	Summarize about the multi-core architecture and processors.

BASIC STRUCTURE OF COMPUTERS	**9**
Functional units -Basic operational concepts - Bus structures - Performance and metrics – Instruction Set Architecture - Addressing modes - RISC - CISC. Fixed point and floating point operations.	
PROCESSOR AND CONTROL UNIT	**9**
Execution of a complete instruction - Multiple bus organization - Hardwired control – Micro-programmed control - Nano programming. Pipelining – Basic concepts - Hazards - Types - Influence on instruction sets – Data path and control considerations - Performance considerations - Exception handling.	
MEMORY SYSTEMS	**9**
Basic concepts - Semiconductor RAM - ROM - Speed - Size and cost - Cache Memories – Improving cache performance - Virtual memory - Memory management requirements - Associative memories – Performance Considerations.	
I/O SYSTEMS	**9**
Accessing I/O devices - Programmed Input/output - Interrupts – Direct Memory Access - Buses – Interface circuits - Standard I/O Interfaces (PCI, SCSI, USB).	
MULTI-CORE ARCHITECTURES	**9**
Introduction to Multithreading – Software and hardware multithreading – SMT and CMP architectures – Design issues – Case study – Intel Core i7. Heterogeneous multi-core processors – Case study: IBM Cell Processor.	
Total Periods:	**45**

Text Book:

1. Carl Hamacher, Zvonko Vranesic and Safwat Zaky, "*Computer Organization*", *6/e*, McGraw-Hill Inc, 2012.
2. John L. Hennessey and David A. Patterson, "*Computer Architecture – A quantitative approach*", Morgan Kaufmann / Elsevier Publishers, 5/e, 2011.

Introduction

Computer Types

Since their introduction in the 1940s, digital computers have evolved into many different types that vary widely in size, cost, computational power, and intended use. Modern computers can be divided roughly into four general categories:

- Embedded computers are integrated into a larger device or system in order to automatically monitor and control a physical process or environment. They are used for a specific purpose rather than for general processing tasks. Typical applications include industrial and home automation, appliances, telecommunication products, and vehicles. Users may not even be aware of the role that computers play in such systems.
- Personal computers have achieved widespread use in homes, educational institutions, and business and engineering office settings, primarily for dedicated individual use. They support a variety of applications such as general computation, document preparation, computer-aided design, audiovisual entertainment, interpersonal communication, and Internet browsing. A number of classifications are used for personal computers. Desktop computers serve general needs and fit within a typical personal workspace. Workstation computers offer higher computational capacity and more powerful graphical display capabilities for engineering and scientific work. Finally, Portable and Notebook computers provide the basic features of a personal computer in a smaller lightweight package. They can operate on batteries to provide mobility.

- Servers and Enterprise systems are large computers that are meant to be shared by a potentially large number of users who access them from some form of personal computer over a public or private network. Such computers may host large databases and provide information processing for a government agency or a commercial organization.

- Supercomputers and Grid computers normally offer the highest performance. They are the most expensive and physically the largest category of computers. Supercomputers are used for the highly demanding computations needed in weather forecasting, engineering design and simulation, and scientific work. They are expensive. Grid computers provide a more cost-effective alternative. They combine many personal computers and disk storage units in a physically distributed high-speed network, called a grid, which is managed as a coordinated computing resource. By evenly distributing the computational workload across the grid, it is possible to achieve high performance on large applications ranging from numerical computation to information searching.

There is an emerging trend in access to computing facilities, known as cloud computing. Personal computer users access widely distributed computing and storage server resources for individual, independent, computing needs. The Internet provides the necessary communication facility. Cloud hardware and software service providers operate as a utility, charging on a pay-as-you-use basis.

The structure of von Neumann's earlier proposal, which is worth quoting at this point:

First: Because the device is primarily a computer, it will have to perform the elementary operations of arithmetic most frequently. At any rate a central arithmetical part of the device will probably have to exist, and this constitutes the first specific part: CA.

Second: The logical control of the device, that is, the proper sequencing of its operations, can be most efficiently carried out by a central control organ. By central control and the organs which perform it form the second specific part: CC

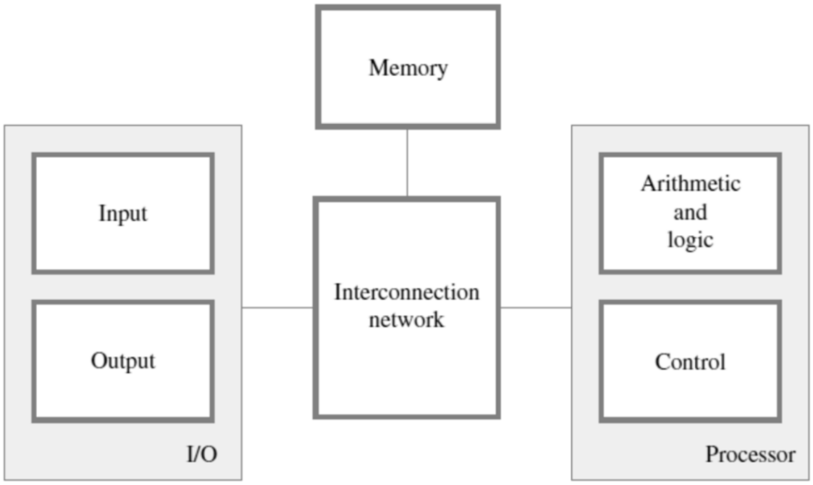
Third: Any device which is to carry out long and complicated sequences of operations (specifically of calculations) must have a considerable memory . . . At any rate, the total memory constitutes the third specific part of the device: M.

Fourth: The device must have organs to transfer . . . information from R into its specific parts C and M. These organs form its input, the fourth specific part: I

Fifth: The device must have organs to transfer . . . from its specific parts C and M into R. These organs form its output, the fifth specific part:

1.**FUNCTIONAL UNITS**:

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate their actions.



We refer to the arithmetic and logic circuits, in conjunction with the main control circuits, as the processor. Input and output equipment is often collectively referred to as the input-output (I/O) unit. It is convenient to categorize this information as either instructions or data. Instructions, or machine instructions, are explicit commands that

- Govern the transfer of information within a computer as well as between the computer and its I/O devices

- Specify the arithmetic and logic operations to be performed

A program is a list of instructions which performs a task. Programs are stored in memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it. Data are numbers and characters that are used as operands by the instructions. Data are also stored in the memory.

The instructions and data handled by a computer must be encoded in a suitable format. Most present-day hardware employs digital circuits that have only two stable states. Each instruction, number, or character is encoded as a string of binary digits called bits, each having one of two possible values, 0 or 1, represented by the two stable states. Numbers are usually represented in positional binary notation. Alphanumeric characters are also expressed in terms of binary codes.

Input Unit

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor. Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input. Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers.

Memory Unit

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory

Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called words. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits. To provide easy access to any word in the memory, a distinct address is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.

Instructions and data can be written into or read from the memory under the control of the processor. It is essential to be able to access any word location in the memory as quickly as possible. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The time required to access one word is called the memory access time. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units.

Cache Memory

As an adjunct to the main memory, a smaller, faster RAM unit, called a cache, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated- circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

At the start of program execution, the cache is empty. All program instructions and any required data are stored in the main memory. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Now, suppose a number of instructions are executed repeatedly as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. Similarly, if the same data locations are accessed repeatedly while copies of their contents are available in the cache, they can be fetched quickly.

Secondary Storage

Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondary storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. A wide selection of secondary storage devices is available, including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit

Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried- out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use. When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

Output Unit

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a printer. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor.

Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name input/output (I/O) unit in many cases.

Control Unit

The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states. I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred. Control circuits are responsible for generating the timing signals that govern the transfers and determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals. It is reasonable to think of a control unit as a well-defined, physically separate unit that interacts with other parts of the computer. In practice, however, this is seldom the case. Much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in memory.
- Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities in the computer are directed by the control unit.

BASIC OPERATIONAL CONCEPTS

The activity in a computer is governed by instructions. To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.

A typical instruction might be

Load R2, LOC

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2. The original contents of location LOC are preserved, whereas those of register R2 are overwritten. Execution of this instruction requires several steps. First, the instruction is fetched from the memory into the processor. Next, the operation to be performed is determined by the control unit. The operand at LOC is then fetched from the memory into the processor. Finally, the operand is stored in register R2.

After operands have been loaded from memory into processor registers, arithmetic or logic operations can be performed on them. For example, the instruction

Add R4, R2, R3

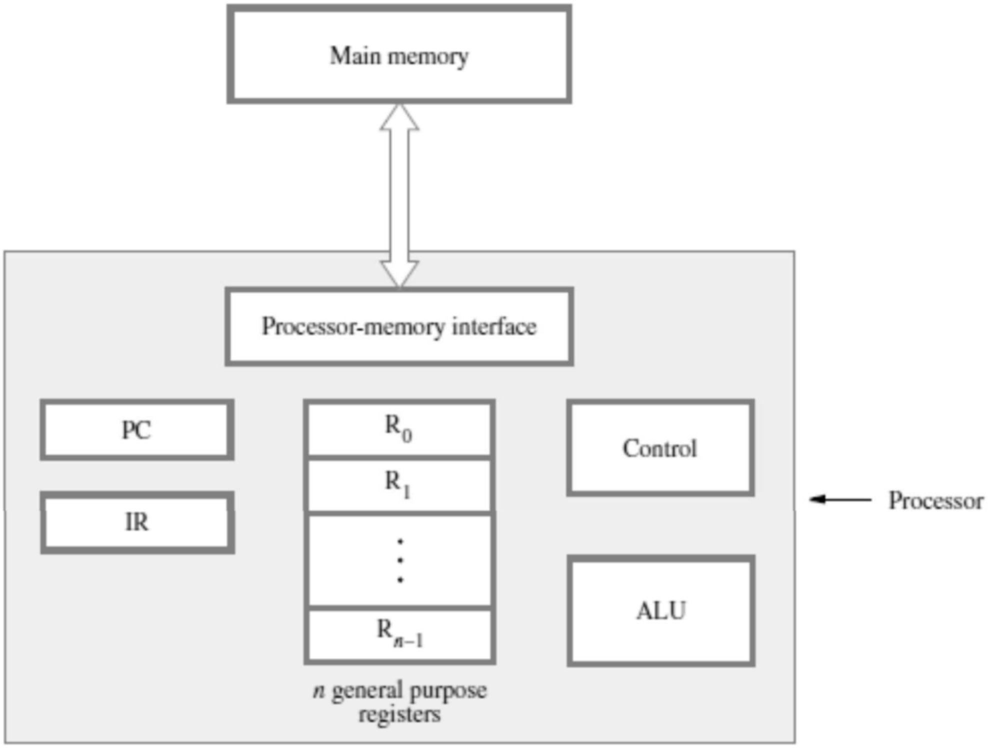
adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

Store R4, LOC

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved. For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location to the memory unit and asserting the appropriate control signals. The data are then transferred to or from the memory.

The figure shows how the memory and the processor can be connected. It also shows some components of the processor that have not been discussed yet. The interconnections between these components are not shown explicitly since we will only discuss their functional characteristics.



In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The instruction register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction. The program counter (PC) is another specialized register. It contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. It is customary to say that the PC points to the next instruction that is to be fetched from the memory. In addition to the IR and PC, Figure shows general-purpose registers R0 through Rn−1, often called processor registers. They serve a variety of functions, including holding operands that have been loaded from the memory for processing.

The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor. If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal. The interface waits for the word to be retrieved, then transfers it to the appropriate processor register. If a ord is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

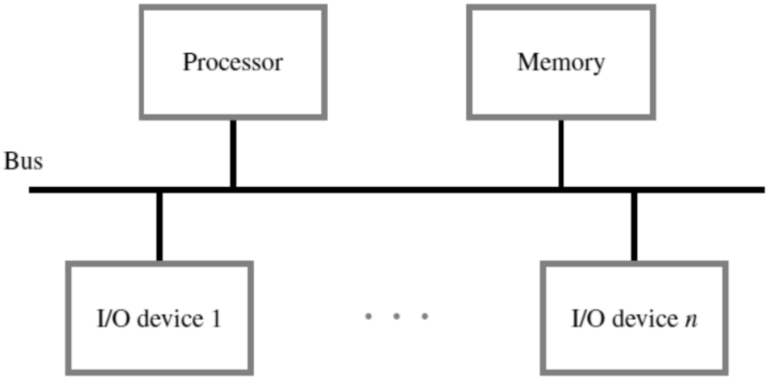
A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage through the input unit. Execution of the program begins when the PC is set to point to the first instruction of the program. The contents of the PC are transferred to the memory along with a Read control signal. When the addressed word (in this case, the first instruction of the program) has been fetched from the memory it is loaded into register IR. At this point, the instruction is ready to be interpreted and executed.

Instructions such as Load, Store, and Add perform data transfer and arithmetic operations. If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register. After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register. If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated. At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices. Thus, some machine instructions are provided for the purpose of handling I/O transfers. Normal execution of a program may be preempted if some device requires urgent service. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an interrupt signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an interrupt-service routine. Because such diversions may alter the internal state of the processor, its state must be saved in the memory before servicing the interrupt request. Normally, the information that is saved includes the contents of the PC, the contents of the general-purpose registers, and some control information. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

BUS STRUCTURES

One of the basic features of a computer is its ability to transfer data to and from I/O devices. An interconnection network is used to transfer data among the processor, memory, and I/O devices. We describe below a commonly used interconnection network called a bus. The bus shown in Figure is a simple structure that implements the interconnection network in Figure. Only one source/destination pair of units can use this bus to transfer data at any one time.



The bus consists of three sets of lines used to carry address, data, and control signals. I/O device interfaces are connected to these lines, as shown in Figure for an input device. Each I/O device is assigned a unique set of addresses for the registers in its interface. When the processor places a particular address on the address lines, it is examined by the address decoders of all devices on the bus. The device that recognizes this address responds to the commands issued on the control lines. The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines.

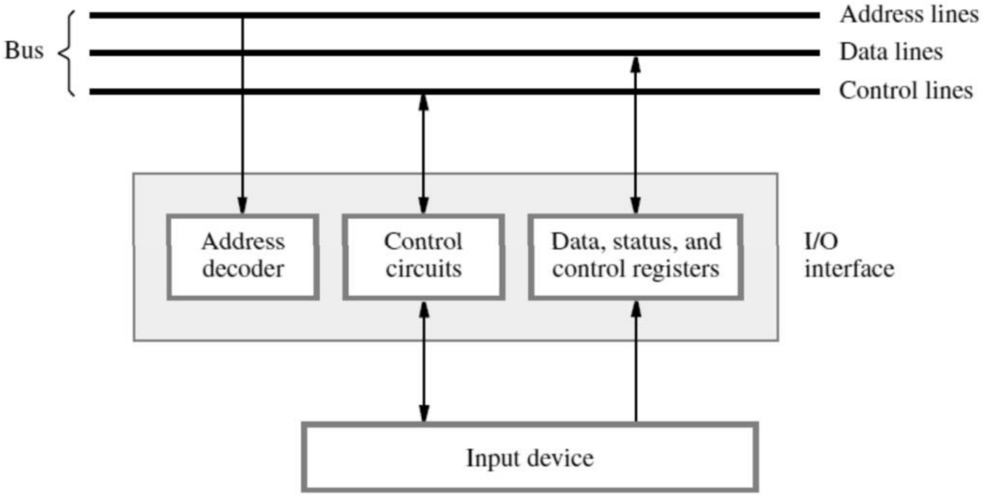
When I/O devices and the memory share the same address space, the arrangement is called memory mapped I/O. Any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if the input device in Figure is a keyboard and if DATAIN is its data register, the instruction

Load R2, DATAIN

reads the data from DATAIN and stores them into processor register R2. Similarly, the instruction

Store R2, DATAOUT

sends the contents of register R2 to location DATAOUT, which may be the data register of a display device interface. The status and control registers contain information relevant to the operation of the I/O device. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.



Bus Operation

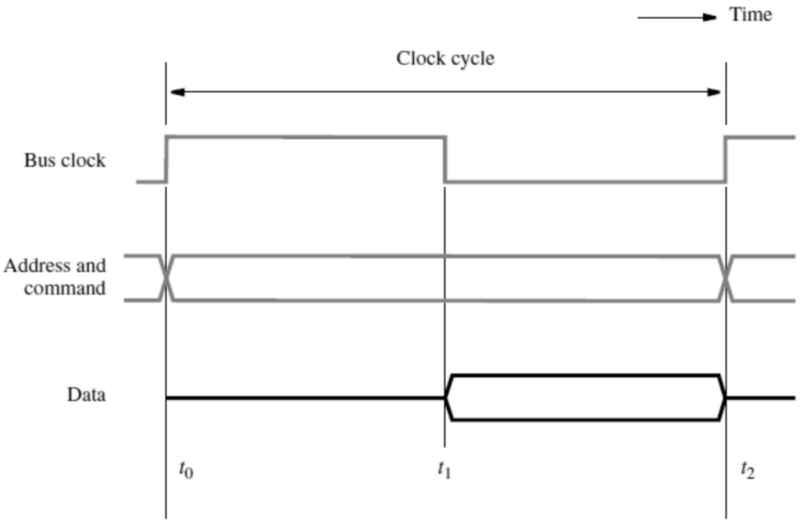
A bus requires a set of rules, often called a bus protocol, that govern how the bus is used by various devices. The bus protocol determines when a device may place information on the bus, when it may load the data on the bus into one of its registers, and so on. These rules are implemented by control signals that indicate what and when actions are to be taken.

One control line, usually labelled R/W, specifies whether a Read or a Write operation is to be performed. As the label suggests, it specifies Read when set to 1 and Write when set to 0. When several data sizes are possible, such as byte, halfword, or word, the required size is indicated by other control lines. The bus control lines also carry timing information. They specify the times at which the processor and the I/O devices may place data on or receive data from the data lines. A variety of schemes have been devised for the timing of data transfers over a bus. These can be broadly classified as either synchronous or asynchronous schemes.

In any data transfer operation, one device plays the role of a master. This is the device that initiates data transfers by issuing Read or Write commands on the bus. Normally, the processor acts as the master, but other devices may also become masters. The device addressed by the master is referred to as a slave.

Synchronous Bus

On a synchronous bus, all devices derive timing information from a control line called the bus clock, shown at the top of Figure 7.3. The signal on this line has two phases: a high level followed by a low level. The two phases constitute a clock cycle. The first half of the cycle between the low-to-high and high-to-low transitions is often referred to as a clock pulse. The address and data lines in Figure are shown as if they are carrying both high and low signal levels at the same time. This is a common convention for indicating that some lines are high and some low, depending on the particular address or data values being transmitted. The crossing points indicate the times at which these patterns change. A signal line at a level half- way between the low and high signal levels indicates periods during which the signal is unreliable, and must be ignored by all devices.



Let us consider the sequence of signal events during an input (Read) operation. At time t0, the master places the device address on the address lines and sends a command on the control lines indicating a Read operation. The command may also specify the length of the operand to be read. Information travels over the bus at a speed determined by its physical and electrical characteristics. The clock pulse width, t1 − t0, must be longer than the maximum propagation delay over the bus. Also, it must be long enough to allow all devices to decode the address and control signals, so that the addressed device (the slave) can respond at time t1 by placing the requested input data on the data lines. At the end of the clock cycle, at time t2, the master loads the data on the data lines into one of its registers. To be loaded correctly into a register, data must be available for a period greater than the setup time of the register (see Appendix A). Hence, the period t2 − t1 must be greater than the maximum propagation time on the bus plus the setup time of the master's register. A similar procedure is followed for a Write operation. The master placed the output data on the data lines when it transmits the address and command information. At t2, the addressed device loads the data into its data register.

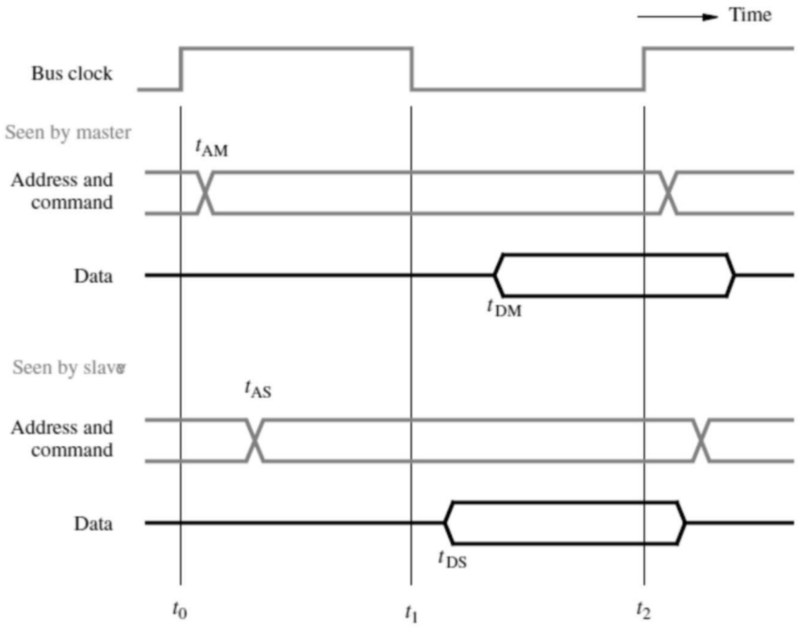
The timing diagram in Figure is an idealized representation of the actions that take place on the bus lines. The exact times at which signals change are somewhat different from those shown, because of propagation delays on bus wires and in the circuits of the devices. Figure 7.4 gives a more realistic picture of what happens. It shows two views of each signal, except the clock. Because signals take time to travel from one device to another, a given signal transition is seen by different devices at different times. The top view shows the signals as seen by the master and the bottom view as seen by the slave. We assume that the clock changes are seen at the same time by all devices connected to the bus. System designers spend considerable effort to ensure that the clock signal satisfies this requirement.

The master sends the address and command signals on the rising edge of the clock at the beginning of the clock cycle (at t0). However, these signals do not actually appear on the bus until tAM, largely due to the delay in the electronic circuit output from the master to the bus lines. A short while later, at tAS, the signals reach the slave. The slave decodes the address, and at t1 sends the requested data. Here again, the data signals do not appear on the bus until tDS. They travel toward the master and arrive at tDM. At t2, the master loads the data into its register. Hence the period t2 − tDM must be greater than the setup time of that register. The data must continue to be valid after t2 for a period equal to the hold time requirement of the register (see Appendix A for hold time). Timing diagrams often show only the simplified picture in Figure, particularly when the intent is to give the basic idea of how data are transferred. But, actual signals will always involve delays as shown in Figure.

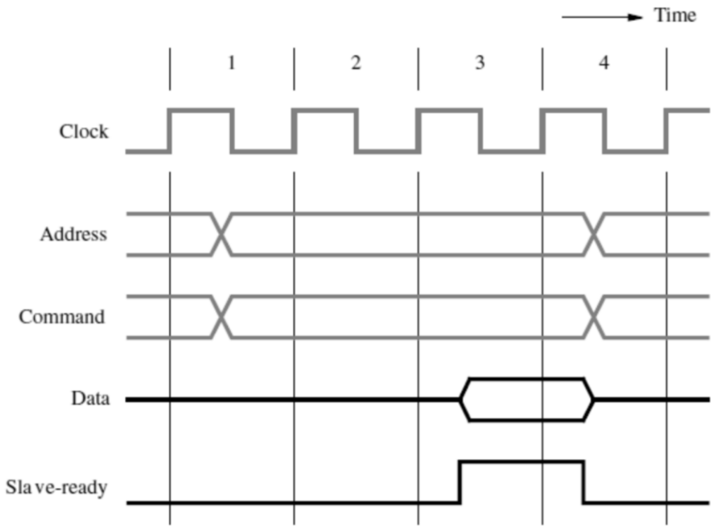
Multiple-Cycle Data Transfer

The scheme described above results in a simple design for the device interface. However, it has some limitations. Because a transfer has to be completed within one clock cycle, the clock period, t2 − t0, must be chosen to accommodate the longest delays on the bus and the slowest device interface. This forces all devices to operate at the speed of the slowest device. Also, the processor has no way of determining whether the addressed device has actually responded. At t2, it simply assumes that the input data are available on the data lines in a Read operation, or that the output data have been received by the I/O device in a Write operation. If, because of a malfunction, a device does not operate correctly, the error will not be detected.

To overcome these limitations, most buses incorporate control signals that represent a response from the device. These signals inform the master that the slave has recognized its address and that it is ready to participate in a data transfer operation. They also make it possible to adjust the duration of the data transfer period to match the response speeds of different devices. This is often accomplished by allowing a complete data transfer operation to span several clock cycles. Then, the number of clock cycles involved can vary from one device to another.



An example of this approach is shown in Figure. During clock cycle 1, the master sends address and command information on the bus, requesting a Read operation. The slave receives this information and decodes it.

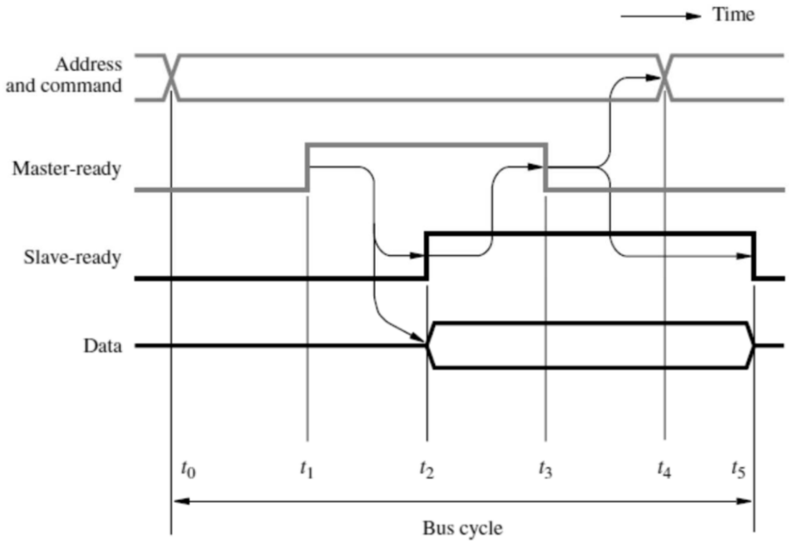


It begins to access the requested data on the active edge of the clock at the beginning of clock cycle 2. We have assumed that due to the delay involved in getting the data, the slave cannot respond immediately. The data becomes ready and are placed on the bus during clock cycle 3. The slave asserts a control signal called Slave-ready at the same time. The master, which has been waiting for this signal, loads the data into its register at the end of the clock cycle. The slave removes its data signals from the bus and returns its Slave-ready signal to the low level at the end of cycle 3. The bus transfer operation is now complete, and the master may send a new address and command signals to start a new transfer in clock cycle 4.

The Slave-ready signal is an acknowledgment from the slave to the master, confirming that the requested data have been placed on the bus. It also allows the duration of a bus transfer to change from one device to another. In the example in Figure, the slave responds in cycle 3. A different device may respond in an earlier or a later cycle. If the addressed device does not respond at all, the master waits for some predefined maximum number of clock cycles, then aborts the operation. This could be the result of an incorrect address or a device malfunction.

Asynchronous Bus

An alternative scheme for controlling data transfers on a bus is based on the use of a handshake protocol between the master and the slave. A handshake is an exchange of command and response signals between the master and the slave. It is a generalization of the way the Slave-ready signal is used in Figure. A control line called Master-ready is asserted by the master to indicate that it is ready to start a data transfer. The Slave responds by asserting Slave- ready.



A data transfer controlled by a handshake protocol proceeds as follows. The master places the address and command information on the bus. Then it indicates to all devices that it has done so by activating the Master-ready line. This causes all devices to decode the address. The selected slave performs the required operation and informs the processor that it has done so by activating the Slave-ready line. The master waits for Slave-ready to become asserted before it removes its signals from the bus. In the case of a Read operation, it also loads the data into one of its registers.

An example of the timing of an input data transfer using the handshake protocol is given in Figure, which depicts the following sequence of events:

t0—The master places the address and command information on the bus, and all devices on the bus decode this information.

t1—The master sets the Master-ready line to 1 to inform the devices that the address and command information is ready. The delay t1 − t0 is intended to allow for any skew that may occur on the bus. Skew occurs when two signals transmitted simultaneously from one source arrive at the destination at different times. This happens because different lines of the bus may have different propagation speeds. Thus, to guarantee that the Master-ready signal does not arrive at any device ahead of the address and command information, the delay t1 − t0 should be longer than the maximum possible bus skew. (Note that bus skew is a part of the maximum propagation delay in the synchronous case.) Sufficient time should be allowed for the device interface circuitry to decode the address. The delay needed can be included in the period t1 − t0.

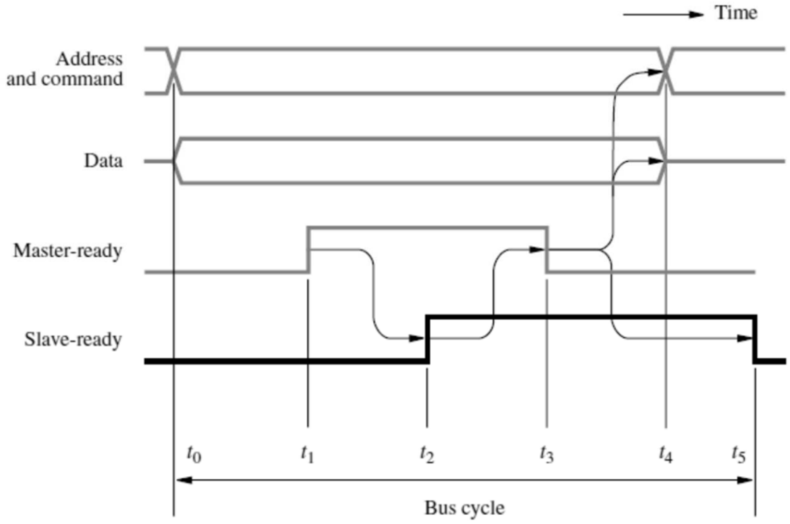
t2—The selected slave, having decoded the address and command information, performs the required input operation by placing its data on the data lines. At the same time, it sets the Slave-ready signal to 1. If extra delays are introduced by the interface circuitry before it places the data on the bus, the slave must delay the Slave-ready signal accordingly. The period t2 − t1 depends on the distance between the master and the slave and on the delays introduced by the slave's circuitry.

t3—The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. The master must allow for bus skew. It must also allow for the setup time needed by its register. After a delay equivalent to the maximum bus skew and the minimum setup time, the master loads the data into its register. Then, it drops the Master-ready signal, indicating that it has received the data.

t4—The master removes the address and command information from the bus. The delay between t3 and t4 is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some device on the bus, starts to change while the Master-ready signal is still equal to 1.

t5—When the device interface receives the 1-to-0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer.

The timing for an output operation, illustrated in Figure, is essentially the same as for an input operation. In this case, the master places the output data on the data lines at the same time that it transmits the address and command information. The selected slave loads the data into its data register when it receives the Master-ready signal and indicates that it has done so by setting the Slave-ready signal to 1. The remainder of the cycle is similar to the input operation.



The handshake signals in Figures 7.6 and 7.7 are said to be fully interlocked, because a change in one signal is always in response to a change in the other. Hence, this scheme is known as a full handshake. It provides the highest degree of flexibility and reliability.

Discussion

Many variations of the bus protocols just described are found in commercial computers. The choice of a particular design involves trade-offs among factors such as:

- Simplicity of the device interface
- Ability to accommodate device interfaces that introduce different amounts of delay
- Total time required for a bus transfer
- Ability to detect errors resulting from addressing a non-existent device or from an interface malfunction

The main advantage of the asynchronous bus is that the handshake protocol eliminates the need for distribution of a single clock signal whose edges should be seen by all devices at about the same time. This simplifies timing design. Delays, whether introduced by the interface circuits or by propagation over the bus wires, are readily accommodated. These delays are likely to differ from one device to another, but the timing of data transfers adjusts automatically. For a synchronous bus, clock circuitry must be designed carefully to ensure proper timing, and delays must be kept within strict bounds.

The rate of data transfer on an asynchronous bus controlled by the handshake protocol is limited by the fact that each transfer involves two round-trip delays (four end-to-end delays). This can be seen in Figures 7.6 and 7.7 as each transition on Slave-ready must wait for the arrival of a transition on Master-ready, and vice versa. On synchronous buses, the clock period need only accommodate one round trip delay. Hence, faster transfer rates can be achieved. To accommodate a slow device, additional clock cycles are used, as described above. Most of today's high-speed buses use the synchronous approach.

Electrical Considerations

A bus is an interconnection medium to which several devices may be connected. It is essential to ensure that only one device can place data on the bus at any given time. A logic gate that places data on the bus is called a bus driver. All devices connected to the bus, except the one that is currently sending data, must have their bus drivers turned off. A special type of logic gate, known as a tri-state gate, is used for this purpose. A tri-state gate has a control input that is used to turn the gate on or off. When turned on, or enabled, it drives the bus with 1 or 0, corresponding to the value of its input signal. When turned off, or disabled, it is effectively disconnected from the bus. From an electrical point of view, its output goes into a high- impedance state that does not affect the signal on the bus.

PERFORMANCE AND METRICS

Performance

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes programs is affected by the design of its instruction set, its hardware and its software, including the operating system, and the technology in which the hardware is implemented. Because programs are usually written in a high-level language, performance is also affected by the compiler that translates programs into machine language. An overview of how performance is affected by technology, as well as processors and system organization.

Technology

The technology ofVery Large Scale Integration (VLSI) that is used to fabricate the electronic circuits for a processor on a single chip is a critical factor in the speed of execution of machine instructions. The speed of switching between the 0 and 1 states in logic circuits is largely determined by the size of the transistors that implement the circuits. Smaller transistors switch faster. Advances in fabrication technology over several decades have reduced transistor sizes dramatically. This has two advantages: instructions can be executed faster, and more transistors can be placed on a chip, leading to more logic functionality and more memory storage capacity.

Parallelism

Performance can be increased by performing a number of operations in parallel. Parallelism can be implemented on many different levels..

Instruction-level Parallelism

The simplest way to execute a sequence of instructions in a processor is to complete all steps of the current instruction before starting the steps of the next instruction. If we overlap the execution of the steps of successive instructions, total execution time will be reduced. For example, the next instruction could be fetched from memory at the same time that an arithmetic operation is being performed on the register operands of the current instruction. This form of parallelism is called pipelining.

Multicore Processors

Multiple processing units can be fabricated on a single chip. In technical literature, the term core is used for each of these processors. The term processor is then used for the complete chip. Hence, we have the terminology dual-core, quad-core, and octo-core processors for chips that have two, four, and eight cores, respectively.

Multiprocessors

Computer systems may contain many processors, each possibly containing multiple cores. Such systems are called multiprocessors. These systems either execute a number of different application tasks in parallel, or they execute subtasks of a single large task in parallel. All processors usually have access to all of the memory in such systems, and the term shared- memory multiprocessor is often used to make this clear. The high performance of these systems comes with much higher complexity and cost, arising from the use of multiple processors and memory units, along with more complex interconnection networks.

In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power. The computers normally have access only to their own memory units. When the tasks they are executing need to share data, they do so by exchanging messages over a communication network. This property distinguishes them from shared-memory multiprocessors, leading to the name message passing multi-computers.

METRICS

Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program execution time is measured in seconds per program. However, time can be defined in different ways, depending on what we count. Th e most straightforward definition of time is called wall clock time, response time, or elapsed time. Th ese terms mean the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything.

Computers are often shared, however, and a processor may work on several programs simultaneously. In such cases, the system may try to optimize throughput rather than attempt to minimize the elapsed time for one program. Hence, we oft en want to distinguish between the elapsed time and the time over which the processor is working on our behalf. CPU execution time or simply CPU time, which recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called user CPU time, and the CPU time spent in the operating system performing tasks on behalf of the program, called system CPU time. Differentiating between system and user CPU time is difficult to do accurately, because it is oft en hard to assign responsibility for operating system activities to one user program rather than another and because of the functionality differences among operating systems. For consistency, we maintain a distinction between performance based on elapsed time and that based on CPU execution time. We will use the term system performance to refer to elapsed time on an unloaded system and CPU performance to refer to user CPU time.

Although as computer users we care about time, when we examine the details of a computer it's convenient to think about performance in other metrics. In particular, computer designers may want to think about a computer by using a measure that relates to how fast the hardware can perform basic functions. Almost all computers are constructed using a clock that determines when events take place in the hardware. Th ese discrete time intervals are called clock cycles (or ticks, clock ticks, clock periods, clocks, cycles). Designers refer to the length of a clock period both as the time for a complete clock cycle (e.g., 250 picoseconds, or 250 ps) and as the clock rate (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period. In the next subsection, we will formalize the relationship between the clock cycles of the hardware designer and the seconds of the computer user.

1. Suppose we know that an application that uses both personal mobile devices and the Cloud is limited by network performance. For the following changes, state whether only the throughput improves, both response time and throughput improve, or neither improves.

   a. An extra network channel is added between the PMD and the Cloud, increasing the total network throughput and reducing the delay to obtain network access (since there are now two channels).

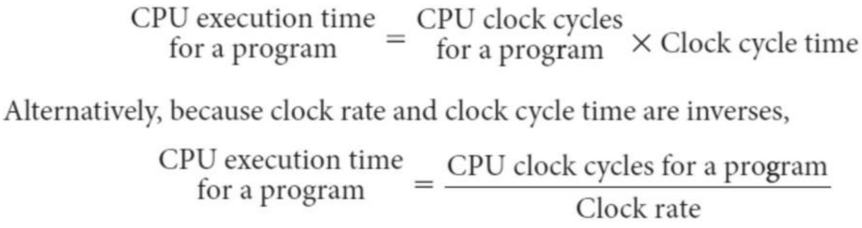
   b. The networking software is improved, thereby reducing the network communication delay, but not increasing throughput.

   c. More memory is added to the computer.

2. Computer C's performance is 4 times as fast as the performance of computer B, which runs a given application in 28 seconds.

CPU Performance and Its Factors

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could determine the effect of a design change on the performance as experienced by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:



$$\begin{matrix}\text{CPU execution time}\\\text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles}\\\text{for a program}\end{matrix} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\begin{matrix}\text{CPU execution time}\\\text{for a program}\end{matrix} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by reducing the number of clock cycles required for a program or the length of the clock cycle. As we will see in later chapters, the designer often faces a trade-off between the number of clock cycles needed for a program and the length of each cycle. Many techniques that decrease the number of clock cycles may also increase the clock cycle time.

Instruction Performance

The performance equations above did not include any reference to the number of instructions needed for the program. However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program. One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \begin{array}{c}\text{Average clock cycles}\\ \text{per instruction}\end{array}$$

Th term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI. Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program. CPI provides one way of comparing two different implementations of the same instruction set architecture, since the number of instructions executed for a program will, of course, be the same.

Classic CPU Performance Equation

We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

CPU time = Instruction count X CPI X Clock cycle time

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

These formulas are particularly useful because they separate the three key factors that affect performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters. The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. Clock cycle Also called tick, clock tick, clock period, clock, or cycle. The time for one clock period, usually of the processor clock, which runs at a constant rate. Clock period - The length of each clock cycle.

The following table summarizes how these components affect the factors in the CPU performance equation.

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more divides, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

Memory Locations and Addresses

We will first consider how the memory of a computer is organized. The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or addresses for each location. We now have three basic information quantities to deal with: bit, byte, and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. The most practical assignment is to have successive addresses refer to successive byte locations in the memory. This is the assignment used in most modern computers. The term byte-addressable memory is used for this assignment. Byte locations have addresses 0, 1, 2, ... There are two ways that byte addresses can be assigned across words, as shown in Figure.

The name big-endian is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name little-endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word. In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, . . . , as shown in Figure. We say that the word locations have aligned addresses if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2.

Memory Operations

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, Read and Write.

The Read operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

INSTRUCTIONS AND INSTRUCTION SEQUENCING

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

We begin by discussing instructions for the first two types of operations.

Register Transfer Notation

We need to describe the transfer of information from one location in a computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify such locations symbolically with convenient names. For example, names that represent the addresses of memory locations may be LOC, PLACE, A, or VAR2. Predefined names for the processor registers may be R0 or R5. Registers in the I/O subsystem may be identified by names such as DATAIN or OUTSTATUS. To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name. Thus, the expression

$$R2 \leftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R2.

As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as

$$R4 \leftarrow [R2] + [R3]$$

This type of notation is known as Register Transfer Notation (RTN). Note that the righthand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location. In computer jargon, the words “transfer” and “move” are commonly used to mean “copy.” Transferring data from a source location A to a destination location B means that the contents of location A are read and then written into location B. In this operation, only the contents of the destination will change. The contents of the source will stay the same.

Assembly-Language Notation

We need another type of notation to represent machine instructions and programs. For this, we use assembly language. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, is specified by the statement

Load R2, LOC

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are loaded into a processor register. The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

Add R4, R2, R3

In this case, R2 and R3 register hold the source operands, while R4 is the destination.

An instruction specifies an operation to be performed and the operands involved. In the above examples, we used the English words Load and Add to denote the required operations. In the assembly-language instructions of actual (commercial) processors, such operations are defined by using mnemonics, which are typically abbreviations of the words describing the operations. For example, the operation Load may be written as LD, while the operation Store, which transfers a word from a processor register to the memory, may be written as STR or ST. Assembly languages for different processors often use different mnemonics for a given operation. To avoid the need for details of a particular assembly language at this early stage, we will continue the presentation in this chapter by using English words rather than processor-specific mnemonics.

RISC and CISC Instruction Sets

One of the most important characteristics that distinguish different computers is the nature of their instructions. There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. This approach is conducive to an implementation of the processing unit in which the various operations needed to process a sequence of instructions are performed in "pipelined" fashion to overlap activity and reduce total execution time of a program. The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called Reduced Instruction Set Computers (RISC).

An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations. This approach was prevalent prior to the introduction of the RISC approach in the 1970s. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called Complex Instruction Set Computers (CISC).

We will start our presentation by concentrating on RISC-style instruction sets because they are simpler and therefore easier to understand.

Introduction to RISC Instruction Sets

Two key characteristics of RISC instruction sets are:

- Each instruction fits in a single word.
- A load/store architecture is used, in which

– Memory operands are accessed only using Load and Store instructions.

– All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer. Processor registers do not contain valid operands at that time. If operands are expected to be in processor registers before they can be used by an instruction, then it is necessary to first bring these operands into the registers. This task is done by Load instructions which copy the contents of a memory location into a processor register. Load instructions are of the form

Load destination, source

or more specifically

Load processor_register, memory_location

The memory location can be specified in several ways. The term addressing modes is used to refer to the different ways in which this may be accomplished. Let us now consider a typical arithmetic operation. The operation of adding two numbers is a fundamental capability in any computer. The statement

$$C = A + B$$

in a high-level language program instructs the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. For simplicity, we will refer to the addresses of these locations as A, B, and C, respectively. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action

$$C \leftarrow [A] + [B]$$

to take place in the computer. To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C. The required action can be accomplished by a sequence of simple machine instructions. We choose to use registers R2, R3, and R4 to perform the task with four instructions:

Load R2, A

Load R3, B

Add R4, R2, R3

Store R4, C

We say that Add is a three-operand, or a three-address, instruction of the form

Add destination, source1, source2

The Store instruction is of the form

Store source, destination

where the source is a processor register and the destination is a memory location. Observe that in the Store instruction the source and destination are specified in the reverse order from the Load instruction; this is a commonly used convention. Note that we can accomplish the desired addition by using only two registers, R2 and R3, if one of the source registers is also used as the destination for the result. In this case the addition would be performed as

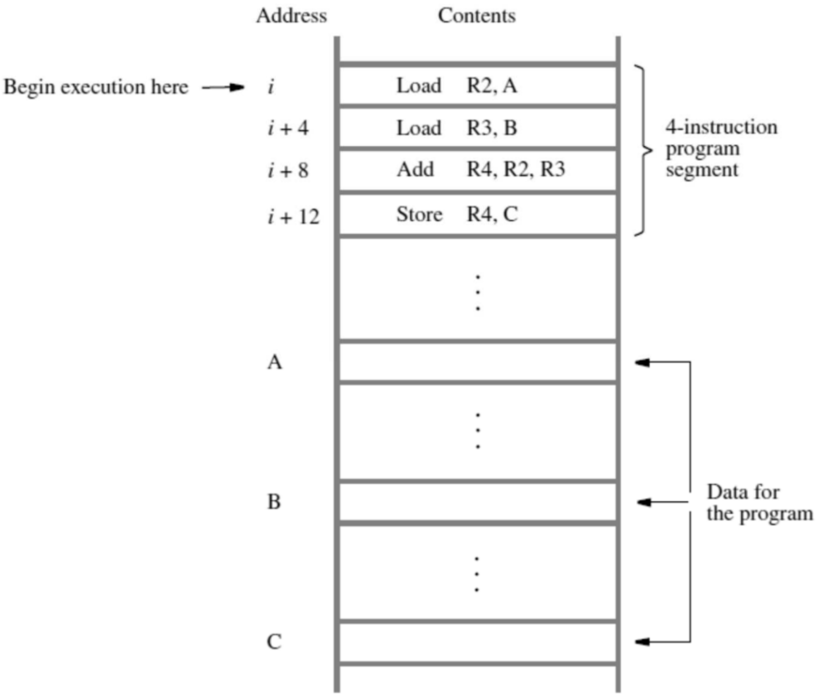
Add R3, R2, R3

and the last instruction would become

Store R3, C

INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING

We used the task C = A + B, implemented as C←[A] + [B], as an example. Figure shows a possible program segment for this task as it appears in the memory of a computer. We assume that the word length is 32 bits and the memory is byte-addressable. The four instructions of the program are in successive word locations, starting at location i. Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses i + 4, i + 8, and i + 12. For simplicity, we assume that a desired memory address can be directly specified in Load and Store instructions, although this is not possible if a full 32-bit address is involved.



Let us consider how this program is executed. The processor contains a register called the program counter (PC), which holds the address of the next instruction to be executed. To begin executing a program, the address of its first instruction (i in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight- line sequencing. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location i + 12 is executed, the PC contains the value i + 16, which is the address of the first instruction of the next program segment.

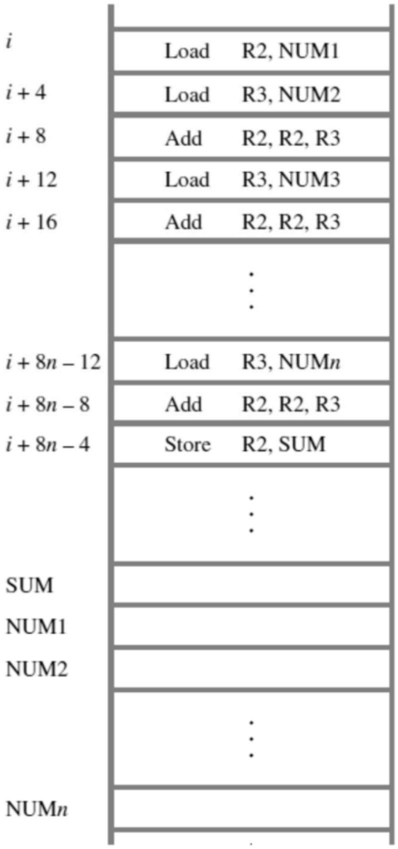
Executing a given instruction is a two-phase procedure. In the first phase, called instruction fetch, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the instruction register (IR) in the processor. At the start of the second phase, called instruction execute, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

Branching

Consider the task of adding a list of n numbers. The program outlined in Figure is a generalization of the program in Figure. The addresses of the memory locations containing the n numbers are symbolically given as NUM1, NUM2, . . . , NUMn, and separate Load and Add instructions are used to add each number to the contents of register R2. After all the numbers have been added, the result is placed in memory location SUM. Instead of using a long list of Load and Add instructions, as in Figure, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. Figure shows the structure of the desired program. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at LOOP location and ends at the instruction Branch if_[R2]>0. During each pass through this loop, the address of the next list entry is determined, and that entry is loaded into R5 and added to R3. The address of an operand can be specified in various ways. For now, we are concentrating on how to create and control a program loop. Assume that the number of entries in the list, n, is stored in memory location N, as shown. Register R2 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R2 at the beginning of the program. Then, within the body of the loop, the instruction

Subtract R2, R2, #1

reduces the contents of R2 by 1 each time through the loop. (We will explain the significance of the number sign '#' in Section 2.4.1.) Execution of the loop is repeated as long as the contents of R2 are greater than zero.

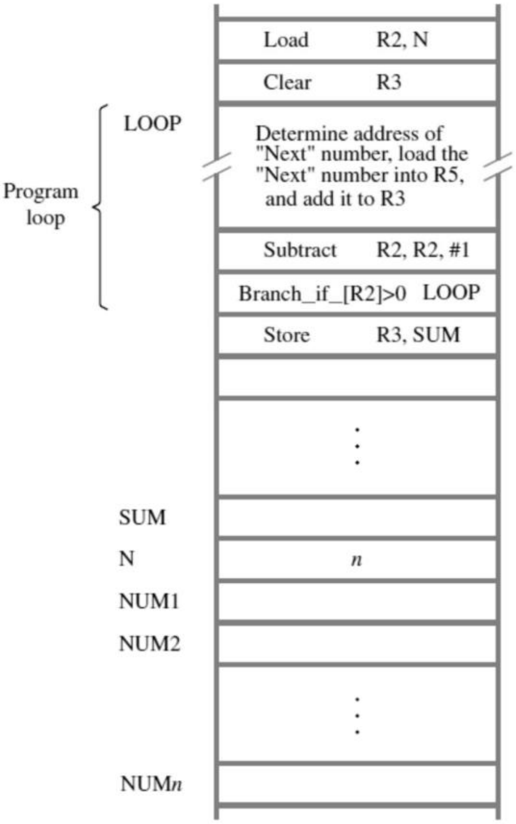


We now introduce branch instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the branch target, instead of the instruction at the location that follows the branch instruction in sequential address order. A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

In the program in Figure, the instruction

Branch if_[R2]>0 LOOP

is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R3. At the end of the nth pass through the loop, the Subtract instruction produces a value of zero in R2, and, hence, branching does not occur. Instead, the Store instruction is fetched and executed. It moves the final result from R3 into memory location SUM.



The ability to test conditions and subsequently choose one of a set of alternative ways to continue computation has many more applications than just loop control. Such a capability is found in the instruction sets of all computers and is fundamental to the programming of most nontrivial tasks.

One way of implementing conditional branch instructions is to compare the contents of two registers and then branch to the target instruction if the comparison meets the specified requirement. For example, the instruction that implements the action

Branch_if_[R4]>[R5] LOOP

may be written in generic assembly language as

Branch_greater_than R4, R5, LOOP

or using an actual mnemonic as

BGT R4, R5, LOOP

It compares the contents of registers R4 and R5, without changing the contents of either register. Then, it causes a branch to LOOPif the contents of R4 are greater than the contents of R5.

Generating Memory Addresses

Let us return to Figure 2.6. The purpose of the instruction block starting at LOOP is to add successive numbers from the list during each pass through the loop. Hence, the Load instruction in that block must refer to a different address during each pass. How are the addresses specified? The memory operand address cannot be given directly in a single Load instruction in the loop. Otherwise, it would need to be modified on each pass through the loop. As one possibility, suppose that a processor register, Ri, is used to hold the memory address of an operand. If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability.

This situation, and many others like it, give rise to the need for flexible ways to specify the address of an operand. The instruction set of a computer typically provides a number of such methods, called addressing modes. While the details differ from one computer to another, the underlying concepts are the same.

.

2.INSTRUCTION SET ARCHITECTURE

In this lesson, you will be described about assembly language for representing machine instructions, data, and programs and Addressing methods for accessing register and memory operands

One key difference is that CISC instruction sets are not constrained to the load/store architecture, in which arithmetic and logic operations can be performed only on operands that are in processor registers. Another key difference is that instructions do not necessarily have to fit into a single word. Some instructions may occupy a single word, but others may span multiple words.

Instructions in modern CISC processors typically do not use a three-address format. Most arithmetic and logic instructions use the two-address format Operation destination, source An Add instruction of this type is

Add B, A

which performs the operation B ← [A] + [B] on memory operands. When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that memory location B is both a source and a destination. Consider again the task of adding two numbers

$C = A + B$

where all three operands may be in memory locations. Obviously, this cannot be done with a single two-address instruction. The task can be performed by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

Move C, B

which performs the operation C←[B], leaving the contents of location B unchanged. The operation C←[A] + [B] can now be performed by the two-instruction sequence

Move C, B

Add C, A

Observe that by using this sequence of instructions the contents of neither A nor B locations are overwritten.

In some CISC processors one operand may be in the memory but the other must be in a register. In this case, the instruction sequence for the required task would be Move Ri, A

Add Ri, B

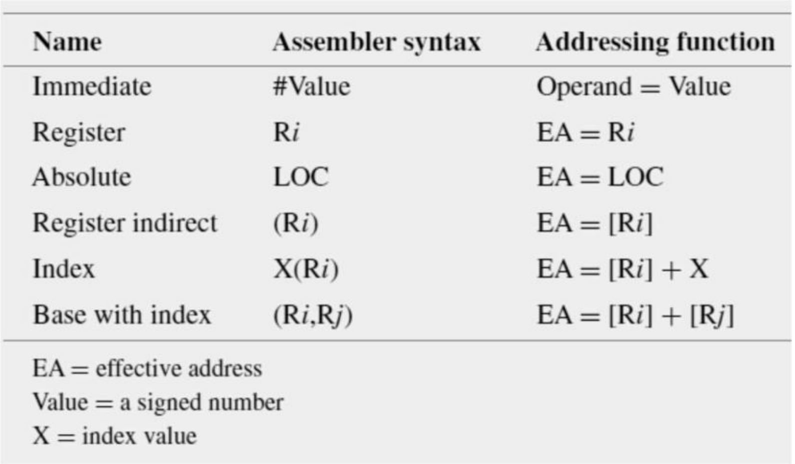
Move C, Ri

The general form of the Move instruction is Move destination, source where both the source and destination may be either a memory location or a processor register. The Move instruction includes the functionality of the Load and Store instructions we used previously in the discussion of RISC-style processors. In the Load instruction, the source is a memory location and the destination is a processor register. In the Store instruction, the source is a register and the destination is a memory location. While Load and Store instructions are restricted to moving operands between memory and processor registers, the Move instruction has a wider scope. It can be used to move immediate operands and to transfer operands between two memory locations or between two registers.

ADDRESSING MODES

We have now seen some simple examples of assembly-language programs. In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways that reflect the nature of the information and how it is used. Programmers use data structures such as lists and arrays for organizing the data used in computations.

Programs are normally written in a high-level language, which enables the programmer to conveniently describe the operations to be performed on various data structures. When translating a high-level language program into assembly language, the compiler generates appropriate sequences of low-level instructions that implement the desired operations. The different ways for specifying the locations of instruction operands are known as addressing modes. In this section we present the basic addressing modes found in RISC-style processors. A summary is provided in Table, which also includes the assembler syntax we will use for each mode.



Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute	LOC	EA = LOC
Register indirect	(R*i*)	EA = [R*i*]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]

EA = effective address
Value = a signed number
X = index value

Implementation of Variables and Constants

Variables are found in almost every computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. This value can be changed as needed using appropriate instructions.

The program in Figure uses only two addressing modes to access variables. We access an operand by specifying the name of the register or the address of the memory location where the operand is located. The precise definitions of these two modes are:

Register mode—The operand is the contents of a processor register; the name of the register is given in the instruction.

Absolute mode—The operand is in a memory location; the address of this location is given explicitly in the instruction.

Since in a RISC-style processor an instruction must fit in a single word, the number of bits that can be used to give an absolute address is limited, typically to 16 bits if the word length is 32 bits. To generate a 32-bit address, the 16-bit value is usually extended to 32 bits by replicating bit b15 into bit positions b31−16 (as in sign extension). This means that an absolute address can be specified in this manner for only a limited range of the full address space. To keep our examples simple, we will assume for now that all addresses of memory locations involved in a program can be specified in 16 bits.

The instruction

Add R4, R2, R3

uses the Register mode for all three operands. Registers R2 and R3 hold the two source operands, while R4 is the destination. The Absolute mode can represent global variables in a program. A declaration such as

Integer NUM1, NUM2, SUM;

in a high-level language program will cause the compiler to allocate a memory location to each of the variables NUM1, NUM2, and SUM. Whenever they are referenced later in the program, the compiler can generate assembly-language instructions that use the Absolute mode to access these variables.

The Absolute mode is used in the instruction

Load R2, NUM1

which loads the value in the memory location NUM1 into register R2.

Constants representing data or addresses are also found in almost every computer program. Such constants can be represented in assembly language using the Immediate addressing mode.

Immediate mode—The operand is given explicitly in the instruction. For example, the instruction

Add R4, R6, 200immediate

adds the value 200 to the contents of register R6, and places the result into register R4. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the number sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

Add R4, R6, #200

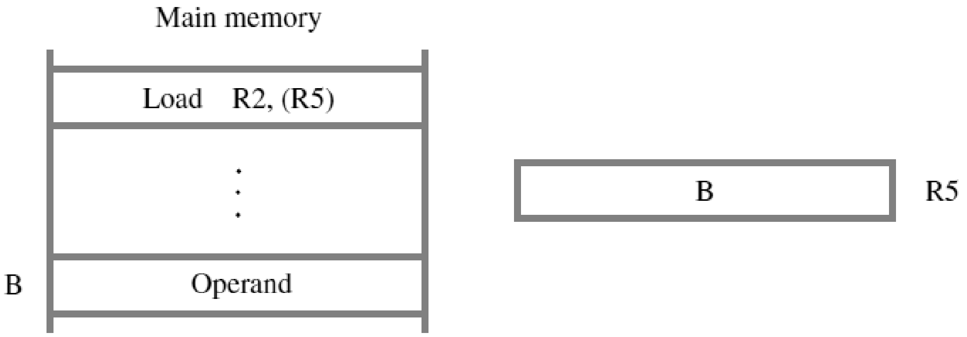
In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an effective address (EA) can be derived by the processor when the instruction is executed. The effective address is then used to access the operand.

Indirection and Pointers

The program in Figure 2.6 requires a capability for modifying the address of the memory operand during each pass through the loop. A good way to provide this capability is to use a processor register to hold the address of the operand. The contents of the register are then changed (incremented) during each pass to provide the address of the next number in the list that has to be accessed. The register acts as a pointer to the list, and we say that an item in the list is accessed indirectly by using the address in the register. The desired capability is provided by the indirect addressing mode.

*Indirect mo*de—The effective address of the operand is the contents of a register that is specified in the instruction.

We denote indirection by placing the name of the register given in the instruction in parentheses as illustrated in Figure and Table.

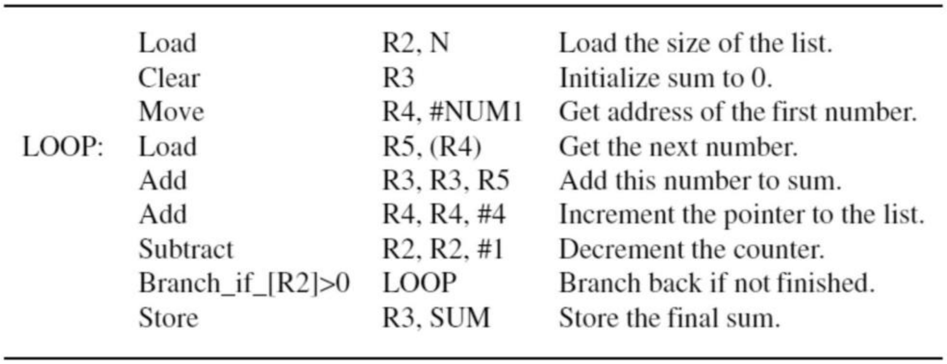


To execute the Load instruction in Figure, the processor uses the value B, which is in register R5, as the effective address of the operand. It requests a Read operation to fetch the contents of location B in the memory. The value from the memory is the desired operand, which the processor loads into register R2. Indirect addressing through a memory location is also possible, but it is found only in CISC-style processors.

Indirection and the use of pointers are important and powerful concepts in programming. They permit the same code to be used to operate on different data. For example, register R5 in Figure serves as a pointer for the Load instruction to load an operand from the memory into register R2. At one time, R5 may point to location B in memory. Later, the program may change the contents of R5 to point to a different location, in which case the same Load instruction will load the value from that location into R2. Thus, a program segment that includes this Load instruction is conveniently reused with only a change in the pointer value.

Let us now return to the program in Figure 2.6 for adding a list of numbers. Indirect addressing can be used to access successive numbers in the list, resulting in the program shown in Figure. Register R4 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R4. The initialization section of the program loads the counter value n from memory location N into R2. Then, it uses the Clear instruction to clear R3 to 0. The next instruction uses the Immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R4. Observe that we cannot use the Load instruction to load the desired immediate value, because the Load instruction can operate only on memory source operands. Instead, we use the Move instruction

Move R4, #NUM1



```
        Load               R2, N          Load the size of the list.
        Clear              R3             Initialize sum to 0.
        Move               R4, #NUM1      Get address of the first number.
LOOP:   Load               R5, (R4)       Get the next number.
        Add                R3, R3, R5     Add this number to sum.
        Add                R4, R4, #4     Increment the pointer to the list.
        Subtract           R2, R2, #1     Decrement the counter.
        Branch_if_[R2]>0   LOOP           Branch back if not finished.
        Store              R3, SUM        Store the final sum.
```

In many RISC-type processors, one general-purpose register is dedicated to holding a constant value zero. Usually, this is register R0. Its contents cannot be changed by a program instruction. We will assume that R0 is used in this manner in our discussion of RISC-style processors. Then, the above Move instruction can be implemented as

Add R4, R0, #NUM1

It is often the case that Move is provided as a pseudo-instruction for the convenience of programmers, but it is actually implemented using the Add instruction.

The first three instructions in the loop in Figure implement the unspecified instruction block starting at LOOP in Figure 2.6. The first time through the loop, the instruction

Load R5, (R4)

fetches the operand at location NUM1 and loads it into R5. The first Add instruction adds this number to the sum in register R3. The second Add instruction adds 4 to the contents of the pointer R4, so that it will contain the address value NUM2 when the Load instruction is executed in the second pass through the loop.

As another example of pointers, consider the C-language statement

*A = *B;*

where B is a pointer variable and the '*' symbol is the operator for indirect accesses. This statement causes the contents of the memory location pointed to by B to be loaded into memory location A. The statement may be compiled into

Load R2, B

Load R3, (R2)

Store R3, A

Indirect addressing through registers is used extensively. The program in Figure shows the flexibility it provides.

Indexing and Arrays

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays. Index mode—The effective address of the operand is generated by adding a constant value to the contents of a register. For convenience, we will refer to the register used in this mode as the index register. Typically, this is just a general-purpose register. We indicate the Index mode symbolically as

X(Ri)

where X denotes a constant signed integer value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by

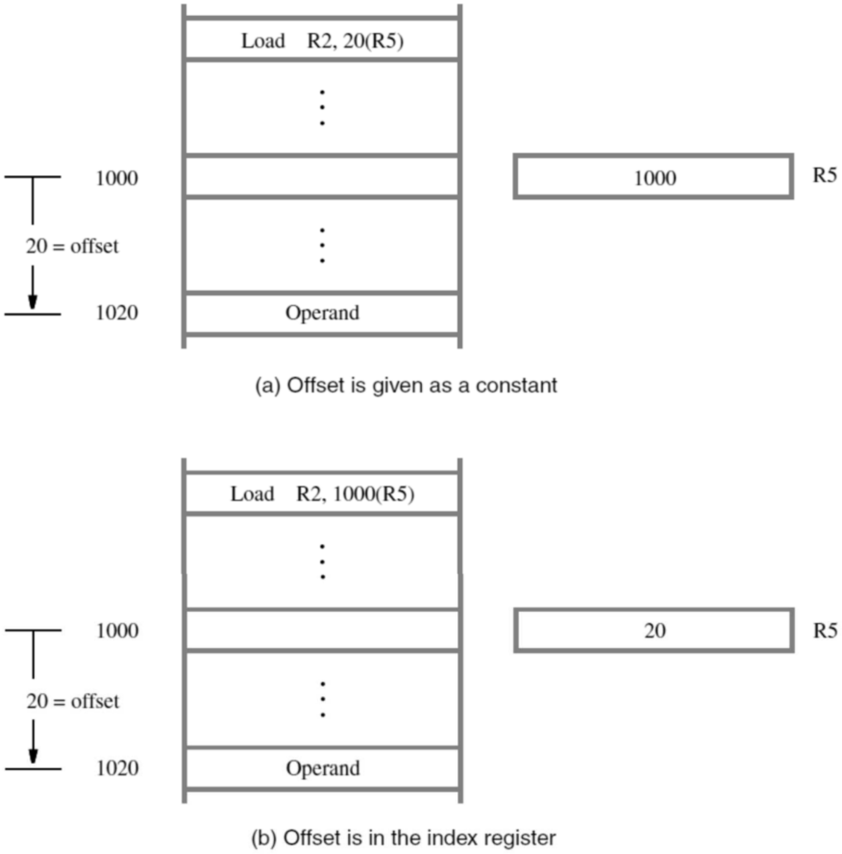
EA = X + [Ri]

The contents of the register are not changed in the process of generating the effective address. In

an assembly-language program, whenever a constant such as the value X is needed,

it may be given either as an explicit number or as a symbolic name representing a numerical value. When the instruction is translated into machine code, the constant X is given as a part of the instruction and is restricted to fewer bits than the word length of the computer. Since X is a signed integer, it must be sign-extended to the register length before being added to the contents of the register.

Figure illustrates two ways of using the Index mode. In Figure a, the index register, R5, contains the address of a memory location, and the value X defines an offset (also called a displacement) from this address to the location where the operand is found. An alternative use is illustrated in Figure b. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is held in a register.

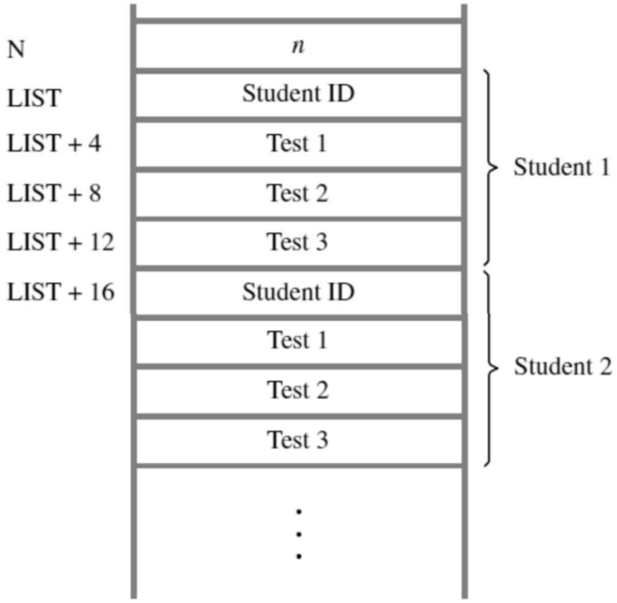


(a) Offset is given as a constant

(b) Offset is in the index register

To see the usefulness of indexed addressing, consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores, beginning at location LIST, is structured as shown in Figure. A four-word memory block comprises a record that stores the relevant information for each student. Each record consists of the student's identification number (ID), followed by the scores the student earned on three tests. There are n students in the class, and the value n is stored in location N immediately in front of the list. The addresses given in the figure for the student IDs and test scores assume that the memory is byte addressable and that the word length is 32 bits.

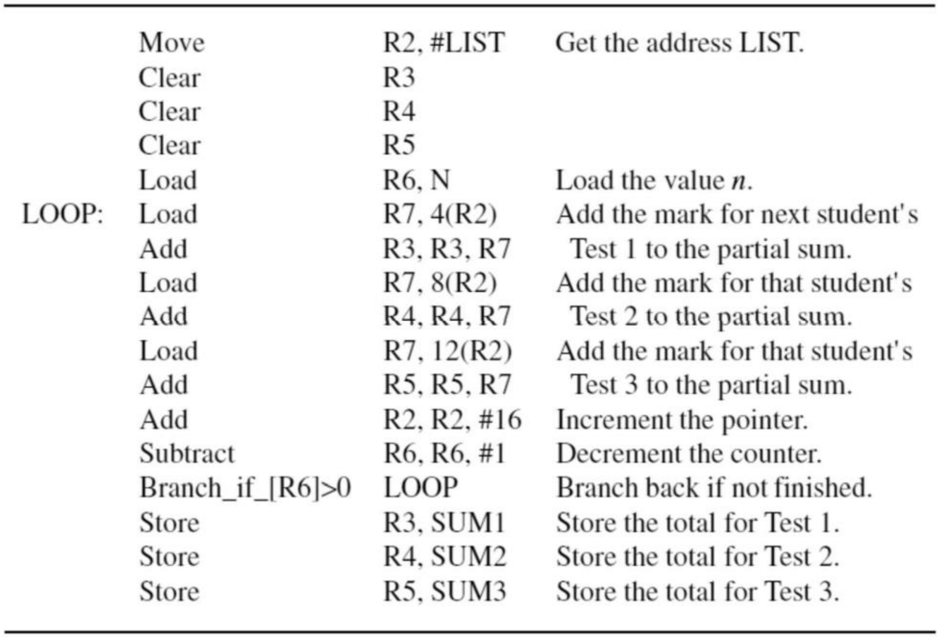
We should note that the list in Figure represents a two-dimensional array having n rows and four columns. Each row contains the entries for one student, and the columns give the IDs and test scores.



Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUM1, SUM2, and SUM3. A possible program for this task is given in Figure. In the body of the loop, the program uses the Index addressing mode in the manner depicted in Figure a to access each of the three scores in a student's record. Register R2 is used as the index register. Before the loop is entered, R2 is set to point to the ID location of the first student record which is the address LIST.

On the first pass through the loop, test scores of the first student are added to the running sums held in registers R3, R4, and R5, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R2), 8(R2), and 12(R2). The index register R2 is then incremented by 16 to point to the ID location of the second student. Register R6, initialized to contain the value n, is decremented by 1 at the end of each pass through the loop. When the contents of R6 reach 0, all student records have been accessed, and the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R3, R4, and R5, into memory locations SUM1, SUM2, and SUM3, respectively.

It should be emphasized that the contents of the index register, R2, are not changed when it is used in the Index addressing mode to access the scores. The contents of R2 are changed only by the last Add instruction in the loop, to move from one student record to the next.



```
        Move             R2, #LIST     Get the address LIST.
        Clear            R3
        Clear            R4
        Clear            R5
        Load             R6, N         Load the value n.
LOOP:   Load             R7, 4(R2)     Add the mark for next student's
        Add              R3, R3, R7      Test 1 to the partial sum.
        Load             R7, 8(R2)     Add the mark for that student's
        Add              R4, R4, R7      Test 2 to the partial sum.
        Load             R7, 12(R2)    Add the mark for that student's
        Add              R5, R5, R7      Test 3 to the partial sum.
        Add              R2, R2, #16   Increment the pointer.
        Subtract         R6, R6, #1    Decrement the counter.
        Branch_if_[R6]>0 LOOP          Branch back if not finished.
        Store            R3, SUM1      Store the total for Test 1.
        Store            R4, SUM2      Store the total for Test 2.
        Store            R5, SUM3      Store the total for Test 3.
```

We have introduced the most basic form of indexed addressing that uses a register Ri and a constant offset X. Several variations of this basic form provide for efficient access to memory operands in practical programming situations (although they may not be included in some processors). For example, a second register Rj may be used to contain the offset X, in which case we can write the Index mode as *(Ri,Rj)*

The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the base register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Yet another version of the Index mode uses two registers plus a constant, which can be denoted as *X(Ri,Rj)*.In this case, the effective address is the sum of the constant and the contents of registers Ri and Rj. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri,Rj) part of the addressing mode.

Finally, we should note that in the basic Index mode *X(Ri)*

if the contents of the register are equal to zero, then the effective address is just equal to the sign-extended value of X. This has the same effect as the Absolute mode. If register R0 always contains the value zero, then the Absolute mode is implemented simply as X(R0)

2.BASIC I/O OPERATION

In this chapter you will learn about:

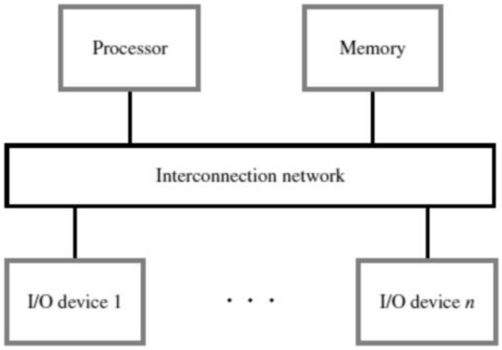
- Transferring data between a processor and input/output (I/O) devices
- The programmer’s view of I/O transfers
- How program-controlled I/O is performed using polling

One of the basic features of a computer is its ability to exchange data with other devices. This communication capability enables a human operator, for example, to use a keyboard and a display screen to process text and graphics. We make extensive use of computers to communicate with other computers over the Internet and access information around the globe. In other applications, computers are less visible but equally important. They are an integral part of home appliances, manufacturing equipment, transportation systems, banking, and point-of- sale terminals. In such applications, input to a computer may come from a sensor switch, a digital camera, a microphone, or a fire alarm. Output may be a sound signal sent to a speaker, or a digitally coded command that changes the speed of a motor, opens a valve, or causes a robot to move in a specified manner. In short, computers should have the ability to exchange digital and analog information with a wide range of devices in many different environments.

In this chapter we will consider the input/output (I/O) capability of computers as seen from the programmer’s point of view. We will present only basic I/O operations, which are provided in all computers. This knowledge will enable the reader to perform interesting and useful exercises on equipment found in a typical teaching laboratory environment.

Accessing I/O Devices

The components of a computer system communicate with each other through an interconnection network, as shown in Figure. The interconnection network consists of circuits needed to transfer information between the processor, the memory unit, and a number of I/O devices.



We described the concept of an address space and how the processor may access individual memory locations within such an address space. Load and Store instructions use addressing modes to generate effective addresses that identify the desired locations. This idea of using addresses to access various locations in the memory can be extended to deal with the I/O devices as well. For this purpose, each I/O device must appear to the processor as consisting of some addressable locations, just like the memory. Some addresses in the address space of the processor are assigned to these I/O locations, rather than to the main memory. These locations are usually implemented as bit storage circuits (flip-flops) organized in the form of registers. It is customary to refer to them as I/O registers. Since the I/O devices and the memory share the same address space, this arrangement is called memory mapped I/O. It is used in most computers.

With memory mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address of a register in an input device, the instruction

```
Load R2, DATAIN
```

reads the data from the DATAIN register and loads them into processor register R2. Similarly, the instruction

```
Store R2, DATAOUT
```

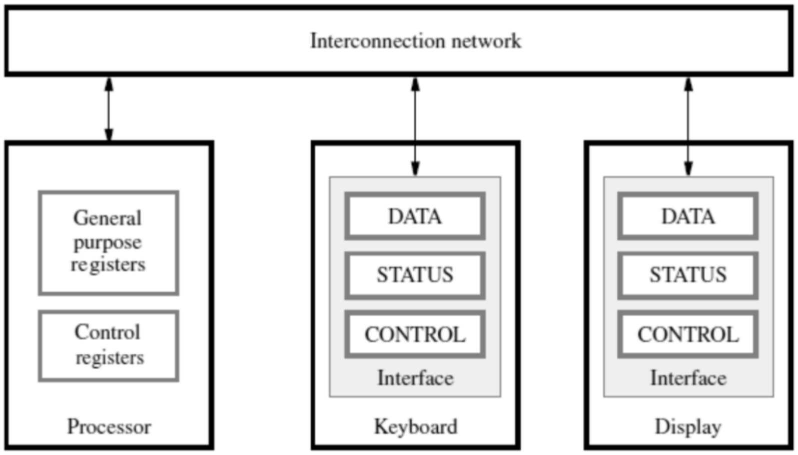
sends the contents of register R2 to location DATAOUT, which is a register in an output device.

I/O Device Interface

An I/O device is connected to the interconnection network by using a circuit, called the device interface, which provides the means for data transfer and for the exchange of status and control information needed to facilitate the data transfers and govern the operation of the device. The interface includes some registers that can be accessed by the processor. One register may serve as a buffer for data transfers, another may hold information about the current status of the device, and yet another may store the information that controls the operational behavior of the device. These data, status, and control registers are accessed by program instructions as if they were memory locations. Typical transfers of information are between I/O registers and the registers in the processor. Figure illustrates how the keyboard and display devices are connected to the processor from the software point of view.

Program-Controlled I/O

Let us begin the discussion of input/output issues by looking at two essential I/O devices for human-computer interaction, keyboard and display. Consider a task that reads characters typed on a keyboard, stores these data in the memory, and displays the same characters on a display screen. A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action. This method is known as program controlled I/O.



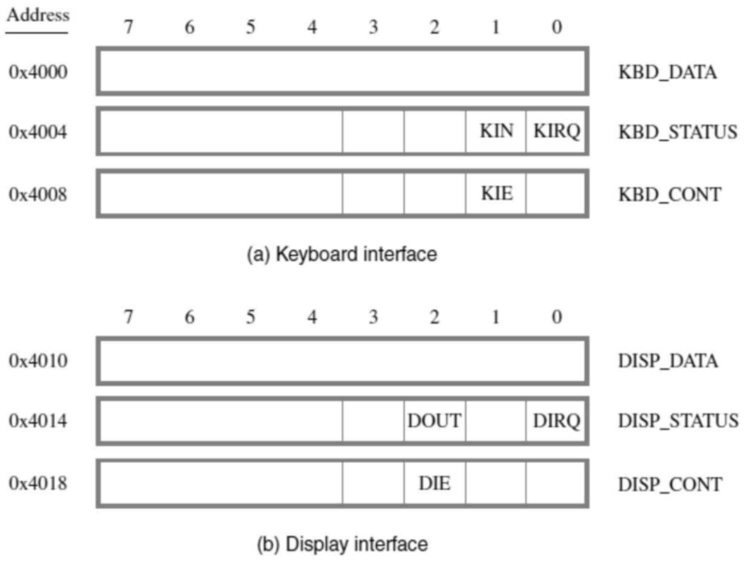
In addition to transferring each character from the keyboard into the memory, and then to the display, it is necessary to ensure that this happens at the right time. An input character must be read in response to a key being pressed. For output, a character must be sent to the display only when the display device is able to accept it. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted to and displayed on the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can execute billions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

One solution to this problem involves a signaling protocol. On output, the processor sends the first character and then waits for a signal from the display that the next character can be sent. It then sends the second character, and so on. An input character is obtained from the keyboard in a similar way. The processor waits for a signal to indicate that a key has been pressed and that a binary code that represents the corresponding character is available in an I/O register associated with the keyboard. Then the processor proceeds to read that code.

The keyboard includes a circuit that responds to a key being pressed by producing the code for the corresponding character that can be used by the computer. We will assume that ASCII code (presented in Table) is used, in which each character code occupies one byte. Let KBD_DA Table the address label of an 8-bit register that holds the generated character. Also, let a signal indicating that a key has been pressed be provided by setting to 1 a flip-flop called KIN, which is a part of an eight-bit status register, KBD_STATUS. The processor can read the status flag KIN to determine when a character code has been placed in KBD_DATA. When the processor reads the status flag to determine its state, we say that the processor polls the I/O device.

The display includes an 8-bit register, which we will call DISP_DATA, used to receive characters from the processor. It also must be able to indicate that it is ready to receive the next character; this can be done by using a status flag called DOUT, which is one bit in a status register, DISP_STATUS.

Figure illustrates how these registers may be organized. The interface for each device also includes a control register. We have identified only a few bits in the registers, those that are pertinent to the discussion in this chapter. Other bits can be used for other purposes, or perhaps simply ignored. If the registers in I/O interfaces are to be accessed as if they are memory locations, each register must be assigned a specific address that will be recognized by the interface circuit. In Figure, we assigned hexadecimal numbers 4000 and 4010 as base addresses for the keyboard and display, respectively. These are the addresses of the data registers. The addresses of the status registers are four bytes higher, and the control registers are eight bytes higher. This makes all addresses word-aligned in a 32-bit word computer, which is usually done in practice. Assigning the addresses to registers in this manner makes the I/O registers accessible in a program executed by the processor. This is the programmer’s view of the device.



(a) Keyboard interface

(b) Display interface

A program is needed to perform the task of reading the characters produced by the keyboard, storing these characters in the memory, and sending them to the display. To perform I/O transfers, the processor must execute machine instructions that check the state of the status flags and transfer data between the processor and the I/O devices.

Let us consider the details of the input process. When a key is pressed, the keyboard circuit places the ASCII-encoded character into the KBD_DATA register. At the same time the circuit sets the KIN flag to 1. Meanwhile, the processor is executing the I/O program which continuously checks the state of the KIN flag. When it detects that KIN is set to 1, it transfers the contents of KBD_DATA into a processor register. Once the contents of KBD_DATA are read, KIN must be cleared to 0, which is usually done automatically by the interface circuit. If a second character is entered at the keyboard, KIN is again set to 1 and the process repeats. The desired action can be achieved by performing the operations:

```
READWAIT    Read the KIN flag
            Branch to READWAIT if KIN = 0
            Transfer data from KBD_DATA to R5
```

which reads the character into processor register R5.

An analogous process takes place when characters are transferred from the processor to the display. When DOUT is equal to 1, the display is ready to receive a character. Under program control, the processor monitors DOUT, and when DOUT is equal to 1, the processor transfers an ASCII-encoded character to DISP_DATA. The transfer of a character to DISP_DATA clears DOUT to 0. When the display device is ready to receive a second character, DOUT is again set to 1. This can be achieved by performing the operations:

```
WRITEWAIT Read the DOUT flag
          Branch to WRITEWAIT if DOUT = 0
          Transfer data from R5 to DISP_DATA
```

The wait loop is executed repeatedly until the status flagDOUTis set to 1 by the display when it is free to receive a character. Then, the character from R5 is transferred to DISP_DATA to be displayed, which also clears DOUT to 0. We assume that the initial state of KIN is 0 and the initial state of DOUT is 1. This initialization is normally performed by the device control circuits when power is turned on.

In computers that use memory mapped I/O, in which some addresses are used to refer to registers in I/O interfaces, data can be transferred between these registers and the processor using instructions such as Load, Store, and Move. For example, the contents of the keyboard character buffer KBD_DATA can be transferred to register R5 in the processor by the instruction

```
LoadByte R5, KBD_DATA
```

Similarly, the contents of register R5 can be transferred to DISP_DATA by the instruction

```
StoreByte R5, DISP_DATA
```

The LoadByte and StoreByte operation codes signify that the operand size is a byte, to distinguish them from the Load and Store operation codes that we have used for word operands.

The Read operation described above may be implemented by the RISC-style instructions:

```
READWAIT:  LoadByte R4, KBD_STATUS
           And R4, R4, #2
           Branch_if_[R4]=0 READWAIT
           LoadByte R5, KBD_DATA
```

The And instruction is used to test the KIN flag, which is bit b1 of the status information in R4 that was read from the KBD_STATUS register. As long as b1 = 0, the result of the AND operation leaves the value in R4 equal to zero, and the READWAIT loop continues to be executed. Similarly, theWrite operation may be implemented as:

```
WRITEWAIT:  LoadByte R4, DISP_STATUS
            And R4, R4, #4
            Branch_if_[R4]=0 WRITEWAIT
            StoreByte R5, DISP_DATA
```

Observe that the And instruction in this case uses the immediate value 4 to test the display’s status bit, b2.

1. FUNDAMENTAL CONCEPTS

A typical computing task consists of a series of operations specified by a sequence of machine-language instructions that constitute a program. The processor fetches one instruction at a time and performs the operation specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered. The processor uses the program counter, PC, to keep track of the address of the next instruction to be fetched and executed. After fetching an instruction, the contents of the PC are updated to point to the next instruction in sequence. A branch instruction may cause a different value to be loaded into the PC.

When an instruction is fetched, it is placed in the instruction register, IR, from where it is interpreted, or decoded, by the processor's control circuitry. The IR holds the instruction until its execution is completed. Consider a 32-bit computer in which each instruction is contained in one word in the memory, as in RISC-style instruction set architecture. To execute an instruction, the processor has to perform the following steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR. In register transfer notation, the required action is

$$IR \leftarrow [[PC]]$$

2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is

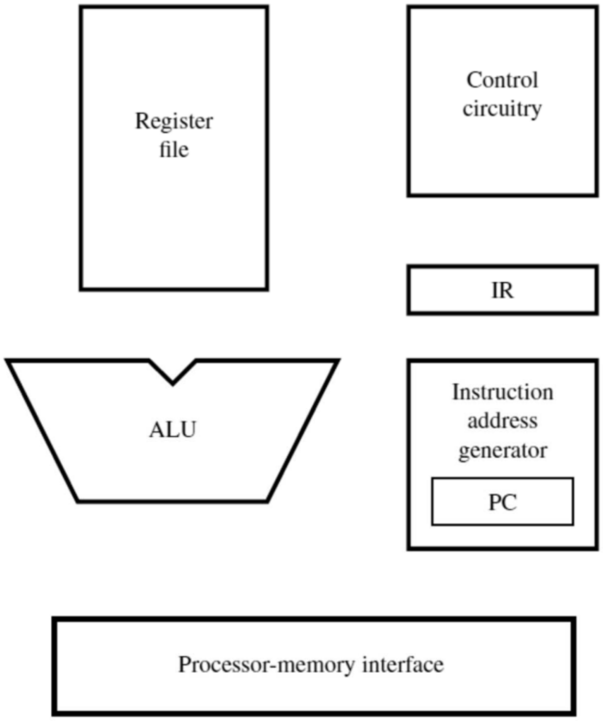
$$PC \leftarrow [PC] + 4$$

3. Carry out the operation specified by the instruction in the IR.

Fetching an instruction and loading it into the IR is usually referred to as the instruction fetch phase. Performing the operation specified in the instruction constitutes the instruction execution phase. With few exceptions, the operation specified by an instruction can be carried out by performing one or more of the following actions:

- Read the contents of a given memory location and load them into a processor register.
- Read data from one or more processor registers.
- Perform an arithmetic or logic operation and place the result into a processor register.
- Store data from a processor register into a given memory location.

The hardware components needed to perform these actions are shown in Figure. The processor communicates with the memory through the processor-memory interface, which transfers data from and to the memory during Read and Write operations. The instruction address generator updates the contents of the PC after every instruction is fetched. The register file is a memory unit whose storage locations are organized to form the processor's general-purpose registers. During execution, the contents of the registers named in an instruction that performs an arithmetic or logic operation are sent to the arithmetic and logic unit (ALU), which performs the required computation. The results of the computation are stored in a register in the register file.



EXECUTION OF A COMPLETE INSTRUCTION

Atypical computation operates on data stored in registers. These data are processed by combinational circuits, such as adders, and the results are placed into a register. A clock signal is used to control the timing of data transfers. The registers comprise edge-triggered flip-flops into which new data are loaded at the active edge of the clock. In this chapter, we assume that the rising edge of the clock is the active edge. The clock period, which is the time between two successive rising edges, must be long enough to allow the combinational circuit to produce the correct result. Let us now examine the actions involved in fetching and executing instructions. We illustrate these actions using a few representative RISC-style instructions.

SINGLE BUS ORGANIZATION

ALU and all the registers are interconnected via a Single Common Bus. Data & address lines of the external memory-bus is connected to the internal processor-bus via MDR.

and MAR respectively. (MDR à Memory Data Register, MAR à Memory Address Register). MDR has 2 inputs and 2 outputs. Data may be loaded into MDR either from memory-bus (external) or from processor-bus (internal).

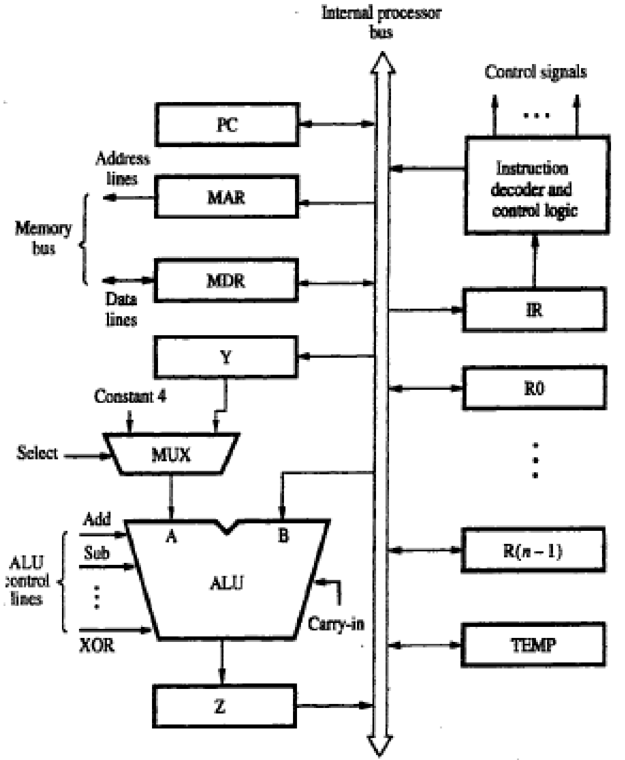


Figure 7.1 Single-bus organization of the datapath inside a processor.

MAR‟s input is connected to internal-bus; MAR‟s output is connected toexternal- bus. Instruction Decoder & Control Unit is responsible for issuing the control-signals to all the units inside the processor. We implement the actions specified by the instruction (loaded in the IR). Register R0 through R(n-1) are the Processor Registers. The programmer can access these registers for general-purpose use. Only processor can access 3 registers Y, Z & Temp for temporary storage during program-execution. The programmer cannot access these 3 registers. In ALU,1) “A‟ input gets the operand from the output of the multiplexer (MUX). 2) “B‟ input gets the operand directly from the processor-bus. There are 2 options provided for “A‟ input of the ALU. MUX is used to select one of the 2 inputs. MUX selects either output of Y or constant-value 4(which is used to increment PC content). An instruction is executed by performing one or more of the following operations:

1) Transfer a word of data from one register to another or to the ALU.

2) Perform arithmetic or a logic operation and store the result in a register.

3) Fetch the contents of a given memory-location and load them into a register.

4) Store a word of data from a register into a given memory-location.

Disadvantage: Only one data-word can be transferred over the bus in a clock cycle.
Solution: Provide multiple internal paths. Multiple paths allow several data-transfers to take place in parallel.

REGISTER TRANSFERS

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control signals are used: Riin & Riout. These are called Gating Signals. Riin=1 = data on bus is loaded into Ri. Riout=1 as content of Ri is placed on bus. Riout=0, makes bus can be used for transferring data from other registers.For example, Move R1, R2; This transfers the contents of register R1 to register R2. This can be accomplished as follows:

1) Enable the output of registers R1 by setting R1out to 1 (Figure 7.2). This places the contents of R1 on the processor-bus.

2) Enable the input of register R2 by setting R2out to 1. This loads data from the processor-bus into register R4.

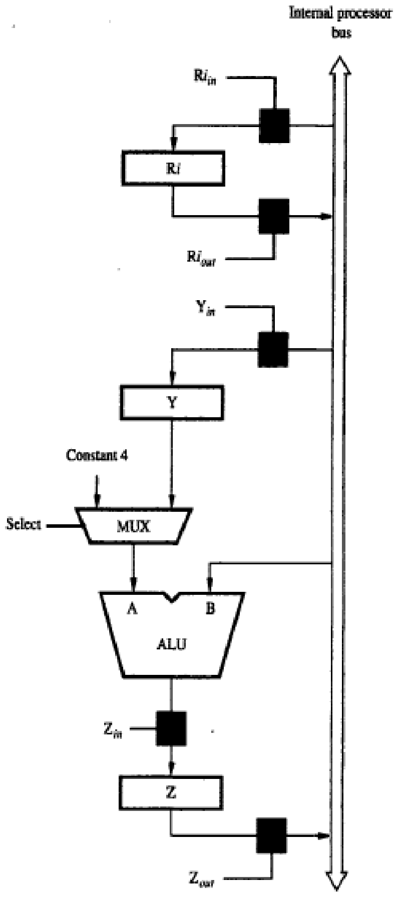


Figure 7.2 Input and output gating for the registers in Figure 7.1.

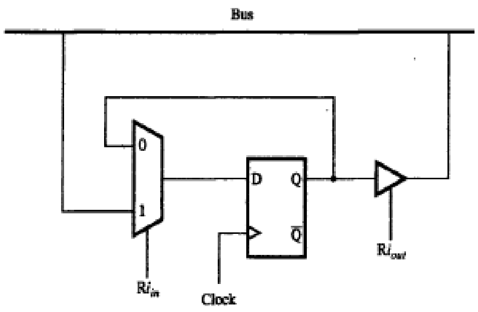


Figure 7.3 Input and output gating for one register bit.

All operations and data transfers within the processor take place within time-periods defined by the processor-clock. The control signals that govern a particular transfer are asserted at the start of the clock cycle.

Input & Output Gating for one Register Bit

A 2-input multiplexer is used to select the data applied to the input of an edge- triggered D flip-flop. Riin=1 makes mux selects data on bus. This data will be loaded into flip- flop at rising-edge of clock. Riin=0 makes mux feeds back the value currently stored in flip- flop (Figure). Q output of flip-flop is connected to bus via a tri-state gate. Riout=0 makes gate's output is in the high-impedance state. Riout=1 makesthe gate drives the bus to 0 or 1, depending on the value of Q.

PERFORMING ARITHMETIC OR LOGIC OPERATION

The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs. One of the operands is output of MUX and, the other operand is obtained directly from processor-bus. The result (produced by the ALU) is stored temporarily in register Z. The sequence of operations for [R3]□[R1]+[R2] is as follows:

1) R1out, Yin

2) R2out, SelectY, Add, Zin

3) Zout, R3in

Instruction execution proceeds as follows:

Step 1 --> Contents from register R1 are loaded into register Y.

Step2 --> Contents from Y and from register R2 are applied to the A and B inputs of ALU; Addition is performed & Result is stored in the Z register.

Step 3 --> The contents of Z register is stored in the R3 register.

The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

CONTROL-SIGNALS OF MDR

The MDR register has 4 control-signals (Figure). MDRin & MDRout control the connection to the internal processor data bus & MDRinE & MDRoutE control the connection to the memory Data bus. MAR register has 2 control-signals. MARin controls the connection to the internal processor address bus & MARout controls the connection to the memory address bus.

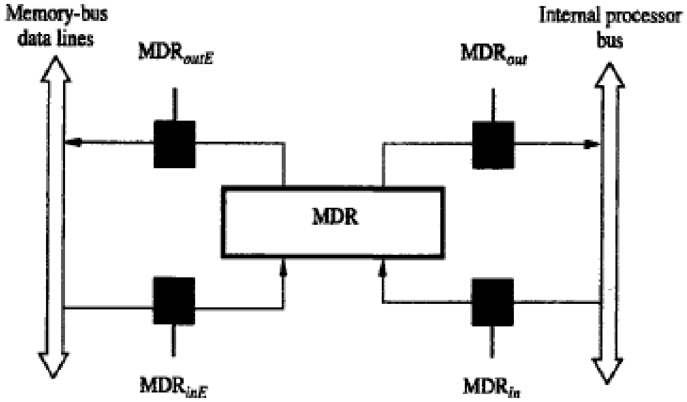
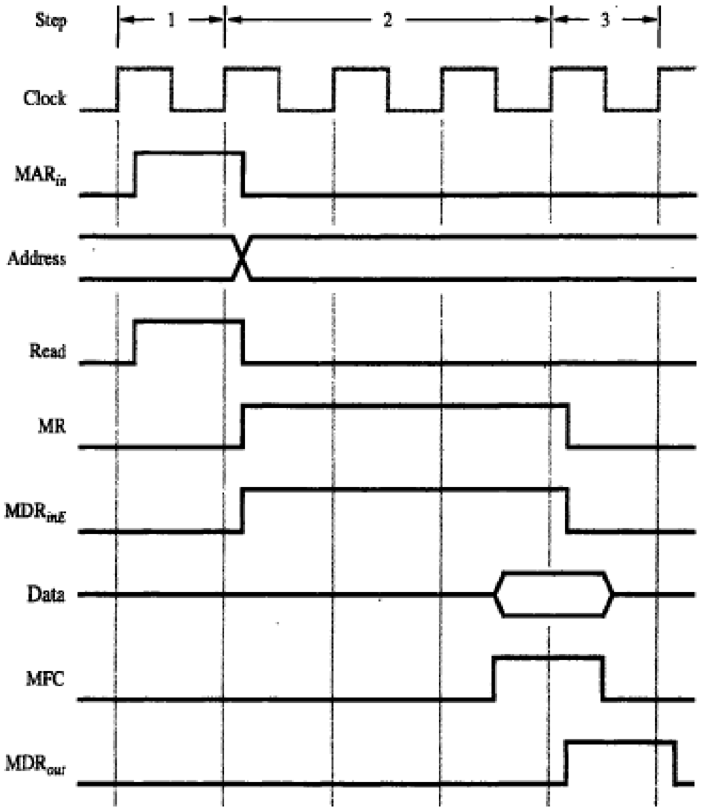


Figure 7.4 Connection and control signals for register MDR.

FETCHING A WORD FROM MEMORY

To fetch instruction/data from memory, processor transfers required address to MAR. At the same time, processor issues Read signal on control-lines of memory-bus. When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers. The response time of each memory access varies (based on cache miss, memory-mapped I/O). To accommodate this, MFC is used. (MFC makes Memory Function Completed). MFC is a signal sent from addressed-device to the processor. MFC informs the processor that the requested operation has been completed by addressed-device.



Consider the instruction Move (R1),R2. The sequence of steps is (Figure): R1out, MARin, Read ;desired address is loaded into MAR & Read command is issued. MDRinE, WMFC; load MDR from memory-bus & Wait for MFC response from memory. MDRout, R2in; load R2 from MDR where WMFC=control-signal that causes processor's control. circuitry to wait for arrival of MFC signal.

Storing a Word in Memory

Consider the instruction Move R2,(R1). This requires the following sequence: R1out, MARin; desired address is loaded into MAR. R2out, MDRin, Write; data to be written are loaded into MDR & Write command is issued. MDRoutE, WMFC ;load data into memory location pointed by R1 from MDR.

EXECUTION OF A COMPLETE INSTRUCTION

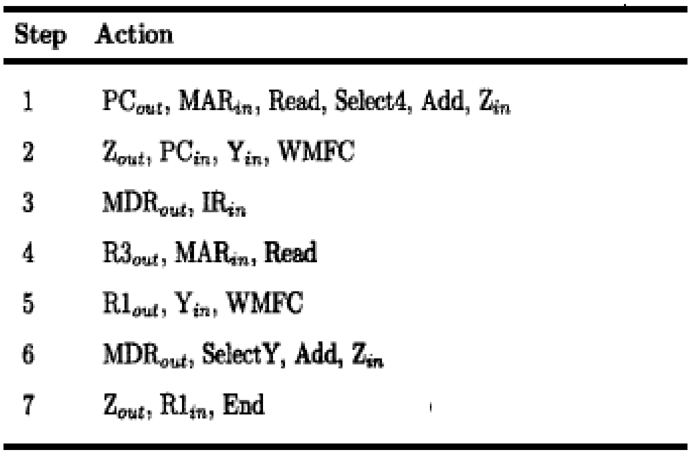
Consider the instruction Add (R3),R1 which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:

1) Fetch the instructions.

2) Fetch the first operand.

3) Perform the addition &

4) Load the result into R1.



Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1

Instruction execution proceeds as follows:

Step1: The instruction-fetch operation is initiated by loading contents of PC into MAR & sending a Read request to memory. The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC‟s content), and the result is stored in Z.

Step2: Updated value in Z is moved to PC. This completes the PC increment operation and PC will now point to next instruction.

Step3: Fetched instruction is moved into MDR and then to IR. The step 1 through 3 constitutes the Fetch Phase. At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control-signals for steps 4 through 7. The step 4 through 7 constitutes the Execution Phase.

Step4: Contents of R3 are loaded into MAR & a memory read signal is issued.

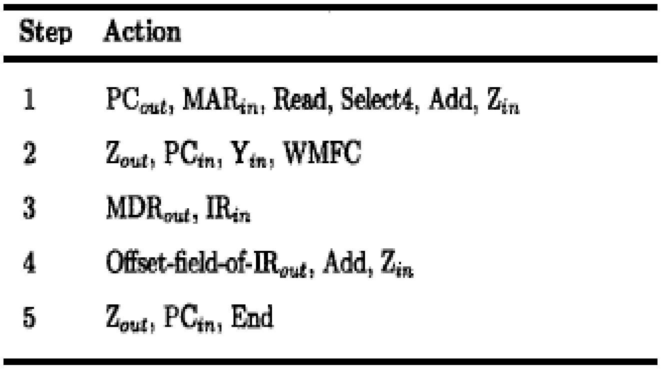
Step5: Contents of R1 are transferred to Y to prepare for addition.

Step6: When Read operation is completed, memory-operand is available in MDR, and the addition is performed.

Step7: Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

BRANCHING INSTRUCTIONS

Control sequence for an unconditional branch instruction is as follows: Instruction execution proceeds as follows:



Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7 Control sequence for an unconditional Branch instruction.

Step 1-3: The processing starts & the fetch phase ends in step3.

Step 4: The offset-value is extracted from IR by instruction-decoding circuit. Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.

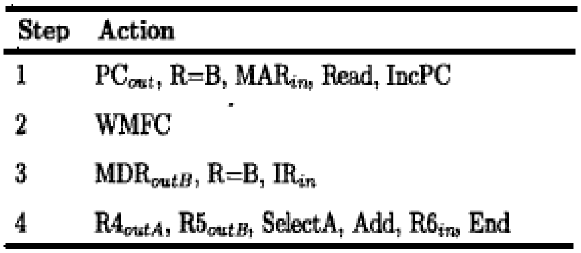
Step 5: The result, which is the branch-address, is loaded into the PC.

The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address. The branch target address is usually obtained by adding the offset in the contents of PC. The offset X is usually the difference between the branch target-address and the address immediately following the branch instruction.

In case of conditional branch, we have to check the status of the condition-codes before loading a new value into the PC. e.g.: Offset-field-of-IRout, Add, Zin, If N=0 then End If N=0, processor returns to step 1 immediately after step 4. If N=1, step 5 is performed load a new value into PC.

MULTIPLE BUS ORGANIZATION

The disadvantage of Single-bus organization is only one data-word can be transferred over the bus in a clock cycle. This increases the steps required to complete the execution of the instruction. The solution to reduce the number of steps, most processors provide multiple internal-paths. Multiple paths enable several transfers to take place in parallel.



Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Figure 7.9 Control sequence for the instruction Add R4,R5,R6

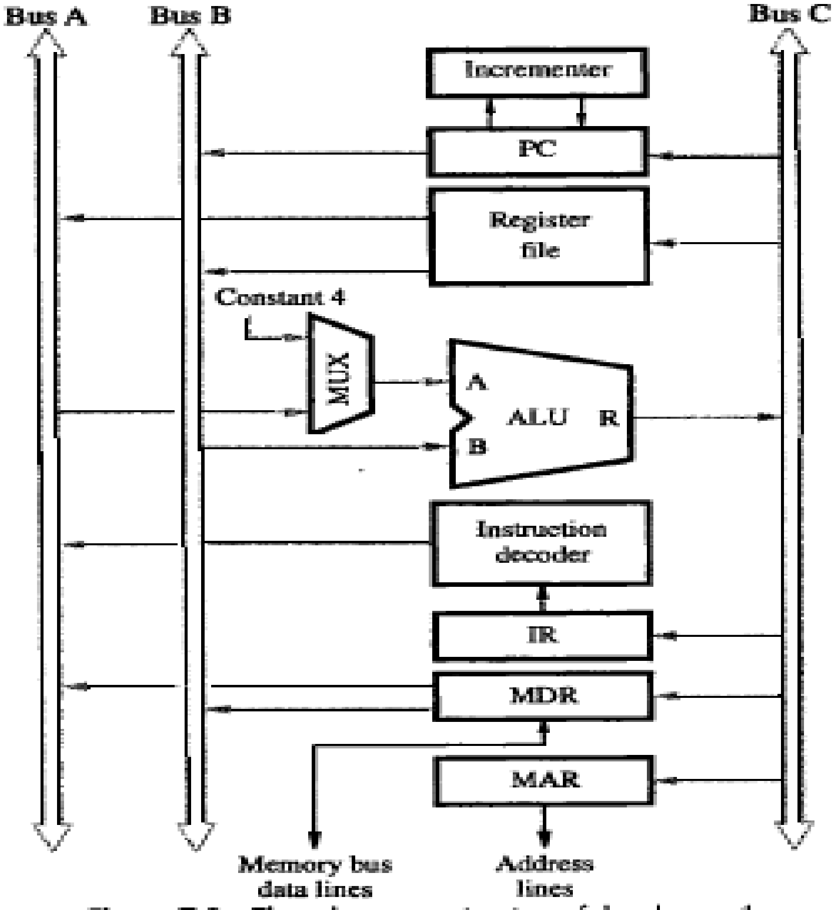


Figure 7.8 Three-bus organization of the datapath.

As shown in figure, three buses can be used to connect registers and the ALU of the processor. All general-purpose registers are grouped into a single block called the Register

File. Register-file has 3 ports:

1) Two output-ports allow the contents of 2 different registers to be simultaneously placed on buses A & B.

2) Third input-port allows data on bus C to be loaded into a third register during the same clock-cycle.

Buses A and B are used to transfer source-operands to A & B inputs of ALU. The result is transferred to destination over bus C. Incrementer Unit is used to increment PC by 4. Instruction execution proceeds as follows:

Step 1: Contents of PC are passed through ALU using R=B control-signal & loaded into MAR to start memory Read operation. At the same time, PC is incremented by 4.

Step2: Processor waits for MFC signal from memory.

Step3: Processor loads requested data into MDR, and then transfers them to IR.

Step4: The instruction is decoded and add operation takes place in a single step.

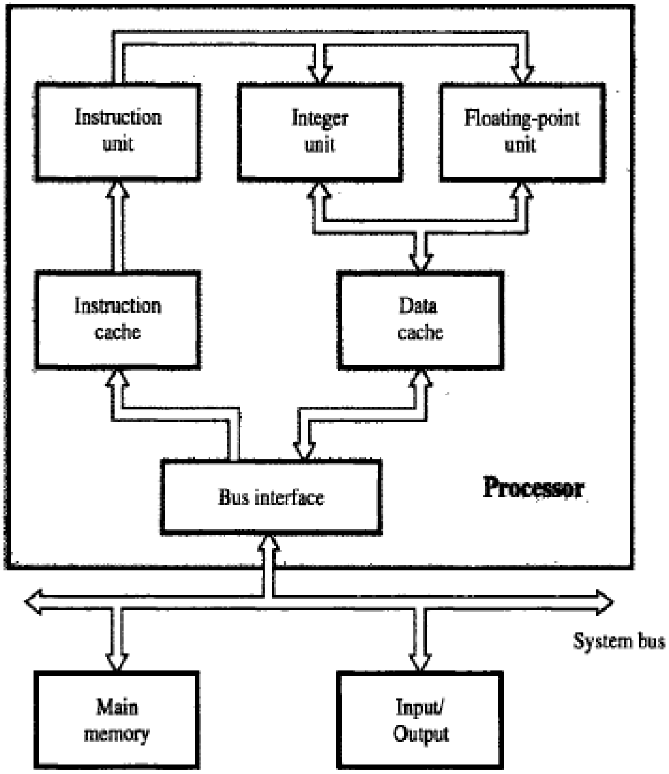


Figure 7.14 Block diagram of a complete processor.

COMPLETE PROCESSOR

This has separate processing units to deal with integer data and floating-point data. The integer unit has to process integer data. (Figure). Floating unit has to process floating point data. Data-Cache is inserted between these processing-units & main-memory. The integer and floating unit gets data from data cache. Instruction-Unit fetches instructions from an instruction-cache or from main-memory when desired instructions are not already in cache.

Processor is connected to system-bus & hence to the rest of the computer by means of a Bus Interface. Using separate caches for instructions & data is common practice in many processors today. A processor may include several units of each type to increase the potential for concurrent operations. The 80486 processor has 8-kbytes single cache for both instruction and data. Whereas the Pentium processor has two separate 8 kbytes caches for instruction and data.

Note: To execute instructions, the processor must have some means of generating the control-signals. There are two approaches for this purpose:

1) Hardwired control and

2) Microprogrammed control.

HARDWIRED CONTROL

Hardwired control is a method of control unit design (Figure). The control-signals are generated by using logic circuits such as gates, flip-flops, decoders etc. Decoder / Encoder Block is a combinational circuit that generates required control-outputs depending on state of all its inputs. Instruction decoder decodes the instruction loaded in the IR. If IR is an 8-bit register, then the instruction decoder generates 28(256 lines); one for each instruction. It consists of separate output-lines INS1 through INSm for each machine instruction. According to code in the IR, one of the output-lines INS1 through INSm is set to 1, and all other lines are set to 0. Step-Decoder provides a separate signal line for each step in the control sequence. Encoder gets the input from instruction decoder, step decoder, external inputs and condition codes. It uses all these inputs to generate individual control-signals: Yin, PCout, Add, End and so on. For example (Figure 7.12), Zin=T1+T6.ADD+T4.BR; This signal is asserted during time-slot T1 for all instructions during T6 for Add instruction. During T4 for unconditional branch instruction, when RUN=1, the counter is incremented by 1 at the end of every clock cycle. When RUN=0, counter stops counting. After execution of each instruction, end signal is generated. End signal resets step counter. Sequence of operations carried out by this machine is determined by wiring of logic circuits, hence the name “hardwired”.

***Advantage**:* Can operate at high speed.

Disadvantages: 1) Since no. of instructions/control-lines is often in hundreds, the complexity of control unit is very high.

2) It is costly and difficult to design.

3) The control unit is inflexible because it is difficult to change the design.

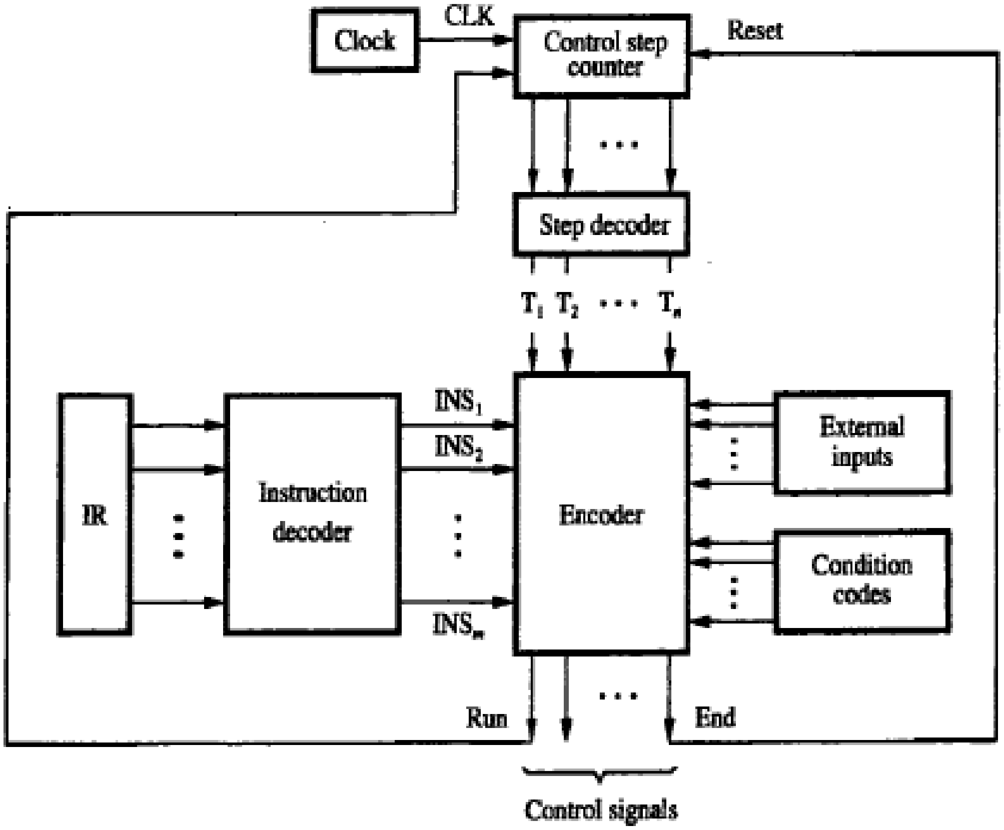


Figure 7.11 Separation of the decoding and encoding functions.

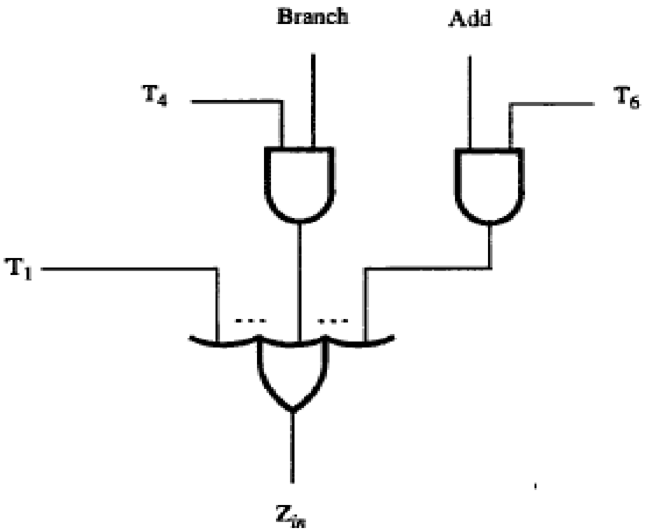
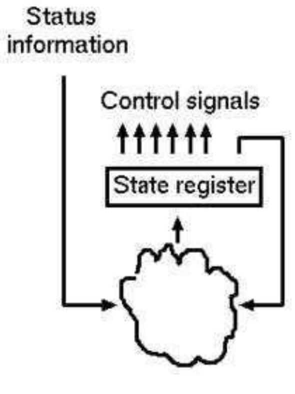
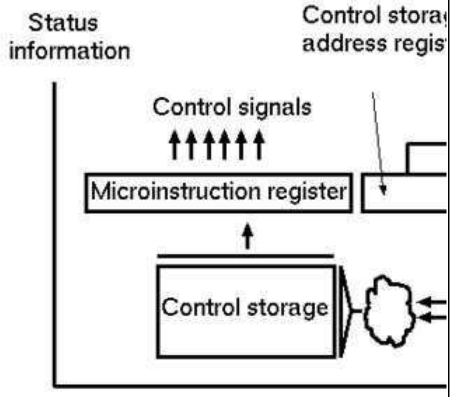


Figure 7.12 Generation of the Z_{in} control signal

HARDWIRED CONTROL VS MICROPROGRAMMED CONTROL

Attribute	**Hardwired Control**	**Microprogrammed Control**
Definition	Hardwired control is a control mechanism to generate control-signals by using gates, flip- flops, decoders, and other digital circuits.	Micro programmed control is a control mechanism to generate control-signals by using a memory called control store (CS), which contains the control-signals.
Speed	Fast	Slow
Control functions	Implemented in hardware.	Implemented in software.
Flexibility	Not flexible to accommodate new system specifications or new instructions.	More flexible, to accommodate new system specification or new instructions redesign is required.
Ability to handle large or complex instruction sets	Difficult.	Easier.
Ability to support Operating systems& diagnostic features	Very difficult.	Easy.
Design process	Complicated.	Orderly and systematic.
Applications	Mostly RISC microprocessors.	Mainframes, some microprocessors.
Instruction set size	Usually under 100 instructions.	Usually over 100 instructions.
ROM size	-	2K to 10K by 20-400 bit microinstructions.
Chip area efficiency	Uses least area.	Uses more area.
Diagram		 Status information Control stora address regis Control signals Microinstruction register Control storage

MICROPROGRAMMED CONTROL

Microprogramming is a method of control unit design (Figure). Control-signals are generated by a program similar to machine language programs. Control Word(CW) is a word whose individual bits represent various control-signals (like Add, PCin). Each of the control-steps in control sequence of an instruction defines a unique combination of 1s & 0s in CW. Individual control-words in microroutine are referred to as microinstructions (Figure).

A sequence of CWs corresponding to control-sequence of a machine instruction constitutes the microroutine. The microroutines for all instructions in the instruction-set of a computer are stored in a special memory called the Control Store (CS). Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS. µPC is used to read CWs sequentially from CS. (µPC□ Microprogram Counter). Every time new instruction is loaded into IR, o/p of Starting Address Generator is loaded into µPC. Then, µPC is automatically incremented by clock; causing successive microinstructions to be read from CS. Hence, control signals are delivered to various parts of processor in correct sequence.

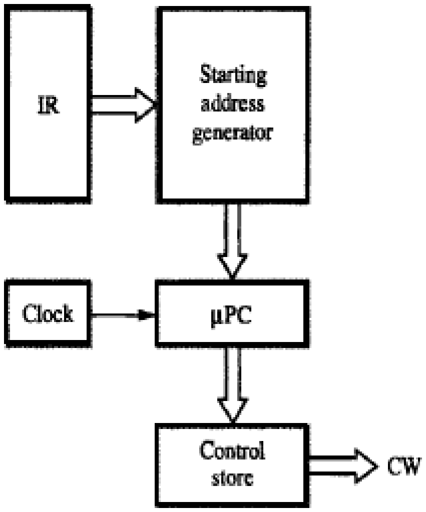
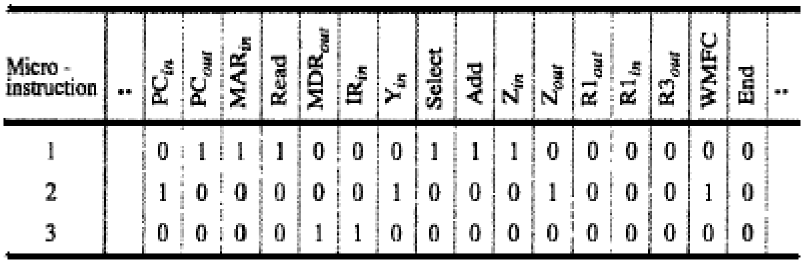


Figure 7.16 Basic organization of a microprogrammed control unit.



Micro - instruction	••	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	••
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	

Figure 7.15 An example of microinstructions for Figure 7.6.

Advantages

• It simplifies the design of control unit. Thus it is both cheaper and less error prone implement.

• Control functions are implemented in software rather than hardware.

• The design process is orderly and systematic.

• More flexibility can be changed to accommodate new system specifications or to correct the design errors quickly and cheaply.

• Complex functions such as floating point arithmetic can be realized efficiently.

Disadvantages

• A microprogrammed control unit is somewhat slower than the hardwired control unit, because time is required to access the microinstructions from CM.

• The flexibility is achieved at some extra hardware cost due to the control memory and its access circuitry.

Organization Of Microprogrammed Control Unit To Support Conditional Branching

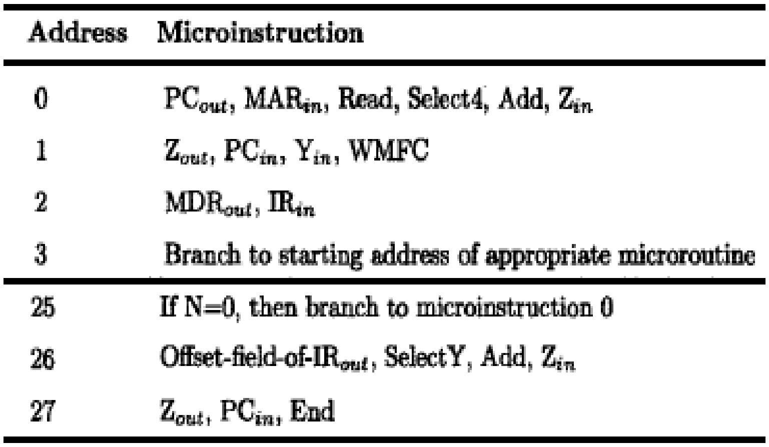
Drawback of previous Microprogram control

□ It cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.

Solution:

□ Use conditional branch microinstruction.

In case of conditional branching, microinstructions specify which of the external inputs, condition- codes should be checked as a condition for branching to take place. Starting and Branch Address Generator Block loads a new address into µPC when a microinstruction instructs it to do so (Figure). To allow implementation of a conditional branch, inputs to this block consist of external inputs and condition-codes & contents of IR.



Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17 Microroutine for the instruction Branch < 0.

µPC is increased every time a new microinstruction is fetched from microprogram memory except in following situations:

1) When a new instruction is loaded into IR, µPC is loaded with starting address of micro routine for that instruction.

2) When a Branch microinstruction is encountered and branch condition is satisfied, µPC is loaded with branch-address.

3) When an End microinstruction is encountered, µPC is loaded with address of first CW in micro routine for instruction fetch cycle.

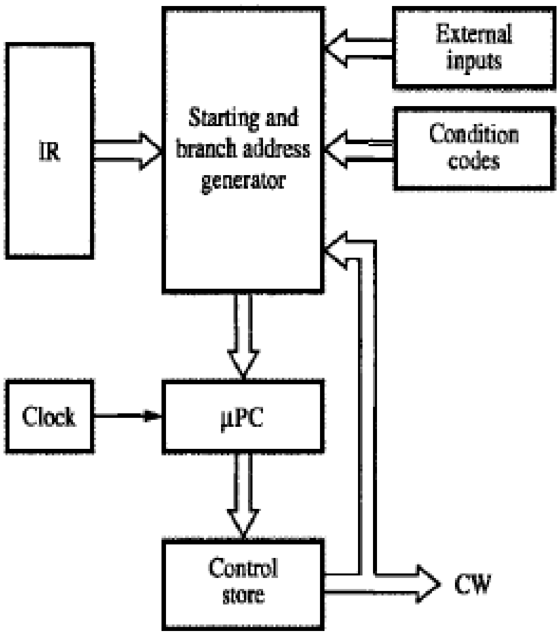


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

Microinstructions

A simple way to structure microinstructions is to assign one bit position to each control-signal required in the CPU. There are 42 signals and hence each microinstruction will have 42 bits.

Drawbacks of microprogrammed control:

1) Assigning individual bits to each control-signal results in long microinstructions because the number of required signals is usually large.

2) Available bit-space is poorly used because only a few bits are set to 1 in any given microinstruction.

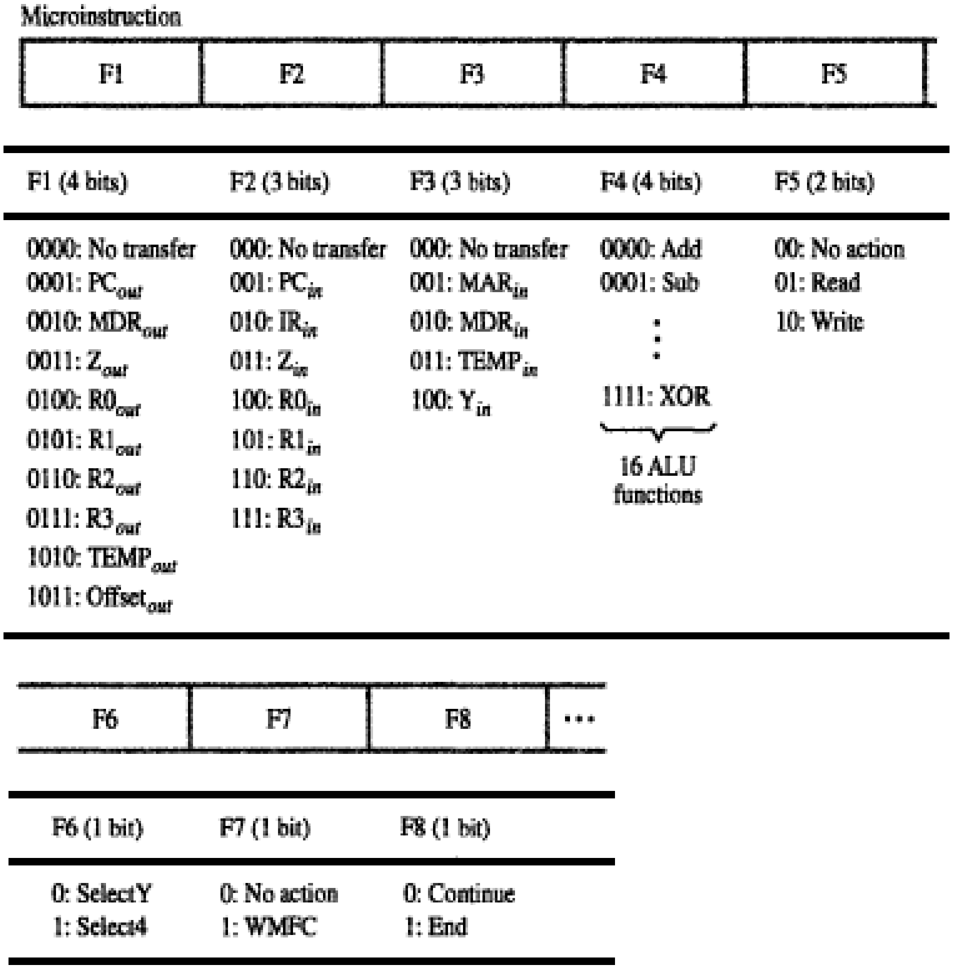
Solution: Signals can be grouped because

1) Most signals are not needed simultaneously.

2) Many signals are mutually exclusive. E.g. only 1 function of ALU can be activated at a time. For ex: Gating signals: IN and OUT signals (Figure); Control-signals: Read, Write; ALU signals: Add, Sub, Mul, Div, Mod.

Grouping control-signals into fields requires a little more hardware because decoding-circuits must be used to decode bit patterns of each field into individual control-signals.

Advantage: This method results in a smaller control-store (only 20 bits are needed to store the patterns for the 42 signals)



Microinstruction

F1	F2	F3	F4	F5

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$		
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	1111: XOR	
0101: $R1_{out}$	101: $R1_{in}$		16 ALU functions	
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

F6	F7	F8	•••

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY	0: No action	0: Continue
1: Select4	1: WMFC	1: End

Figure 7.19 An example of a partial format for field-encoded microinstructions.

Techniques of Grouping of Control-Signals

The grouping of control-signal can be done either by using

1) Vertical organization &

2) Horizontal organisation.

Vertical Organization	Horizontal Organization
Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a vertical organization.	The minimally encoded scheme in which many resources can be controlled with a single microinstuction is called a horizontal organization.
Slower operating-speeds.	Useful when higher operating-speed is desired.
Short formats.	Long formats.
Limited ability to express parallel microoperations.	Ability to express a high degree of parallelism.
Considerable encoding of the control information.	Little encoding of the control information.

Microprogram Sequencing

The task of microprogram sequencing is done by microprogram sequencer. Two important factors must be considered while designing the microprogram sequencer,

1) The size of the microinstruction &

2) The address generation time.

The size of the microinstruction should be minimum so that the size of control memory required to store microinstructions is also less. This reduces the cost of control memory. With less address generation time, microinstruction can be executed in less time resulting better throughout. During execution of a microprogram the address of the next microinstruction to be executed has 3 sources:

1) Determined by instruction register.

2) Next sequential address &

3) Branch.

Microinstructions can be shared using microinstruction branching.

Disadvantage of microprogrammed branching:

1) Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control-store.

2) Execution time is longer because it takes more time to carry out the required branches.

Consider the instruction Add src,Rdst ;which adds the source-operand to the contents of Rdst and places the sum in Rdst. Let source-operand can be specified in following addressing modes

a) Indexed b) Autoincrement c) Autodecrement d) Register indirect & e) Register direct

Each box in the chart corresponds to a microinstruction that controls the transfers and operations indicated within the box. The microinstruction is located at the address indicated by the octal number (001,002).

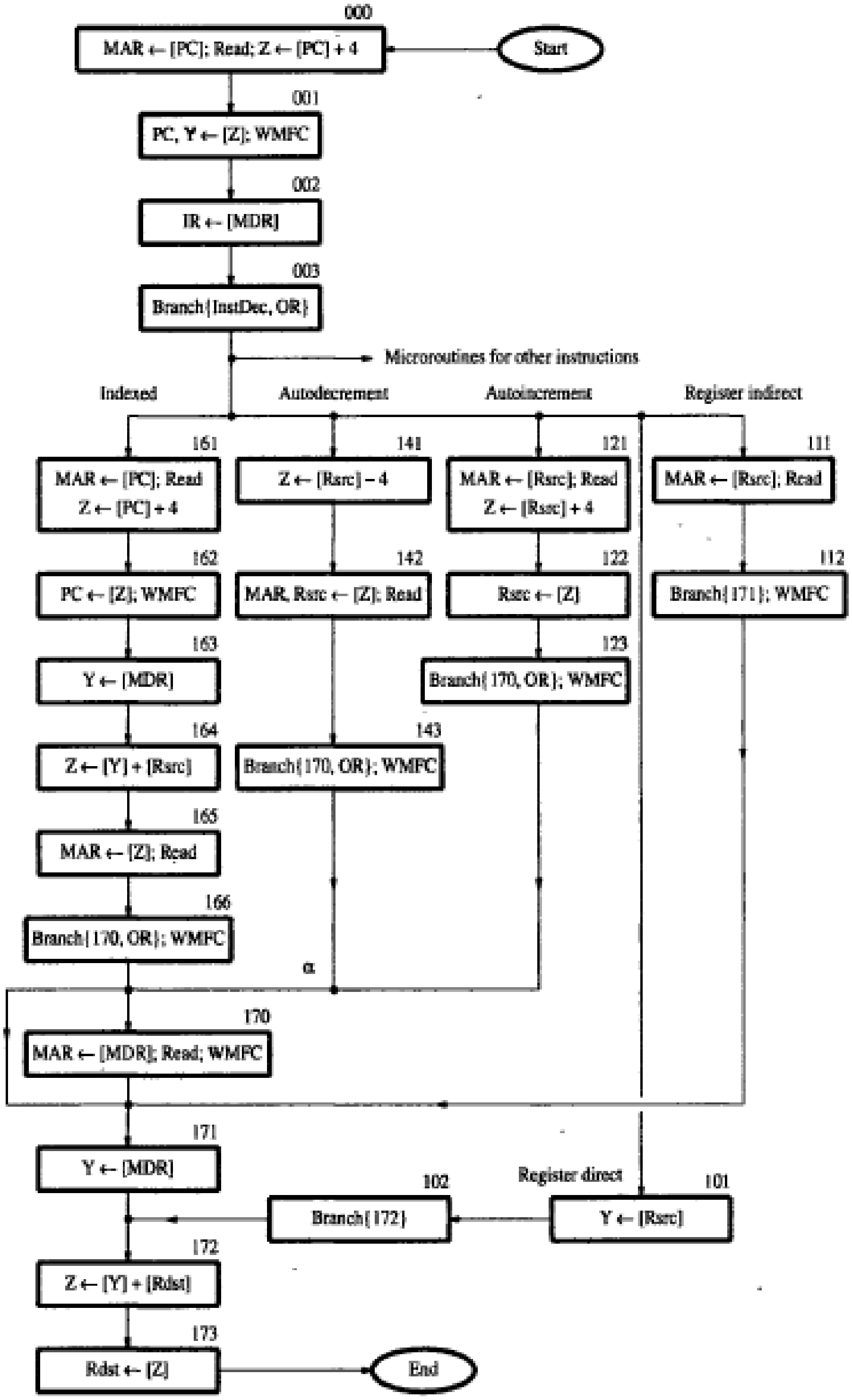


Figure 7.20 Flowchart of a microprogram for the Add src,Rdst instruction.

Branch Address Modification Using Bit-Oring

The branch address is determined by ORing particular bit or bits with the current address of microinstruction. Eg: If the current address is 170 and branch address is 171 then the branch address can be generated by ORing 01(bit 1), with the current address. Consider the point labelled in the figure. At this point, it is necessary to choose between direct and indirect addressing modes. If indirect mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory. If direct mode is specified, this fetch must be bypassed by branching immediately to location 171. The most efficient way to bypass microinstruction 170 is to have bit-ORing of current address 170 & branch address 171.

Wide Branch Addressing

The instruction-decoder (InstDec) generates the starting-address of the microroutine that implements the instruction that has just been loaded into the IR. Here, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101. (However, this address cannot be loaded as is into the μPC). The source operand can be specified in any of several addressing-modes. The bit-ORing technique can be used to modify the starting-address generated by the instruction-decoder to reach the appropriate path.

Use of WMFC

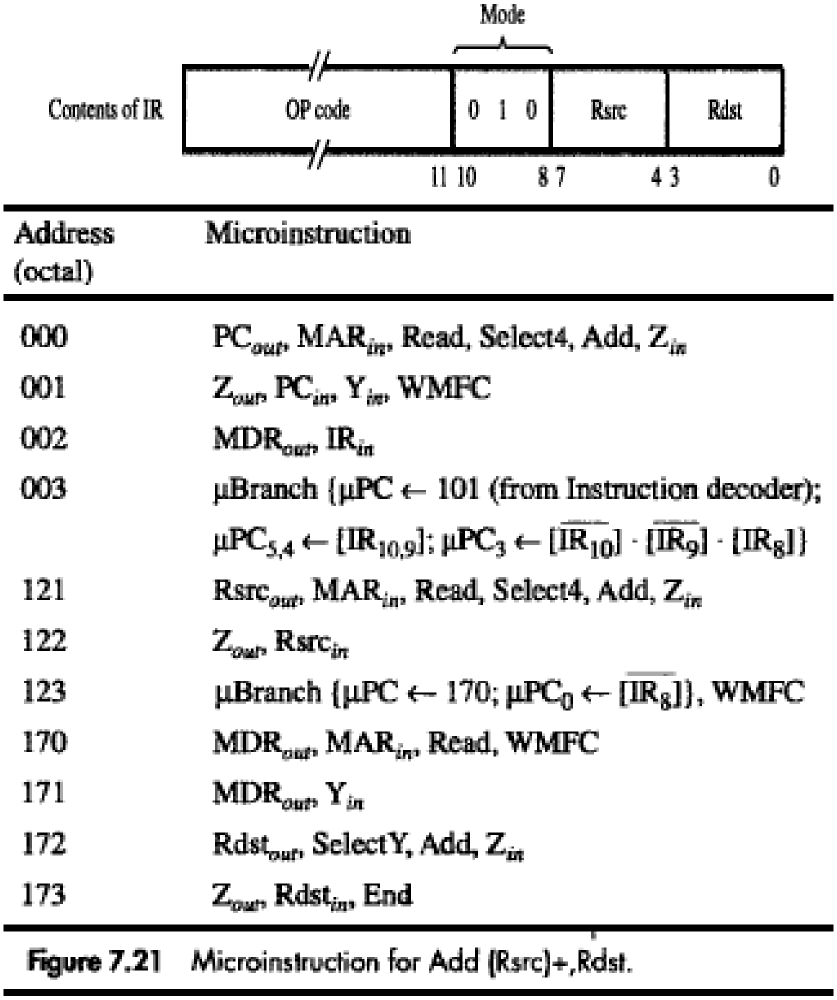
WMFC signal is issued at location 112 which causes a branch to the microinstruction in location 171. WMFC signal means that the microinstruction may take several clock cycles to complete. If the branch is allowed to happen in the first clock cycle, the microinstruction at location 171 would be fetched and executed prematurely. To avoid this problem, WMFC signal must inhibit any change in the contents of the μPC during the waiting period.

Detailed Examination of Add (Rsrc)+,Rdst

Consider Add (Rsrc)+,Rdst; which adds Rsrc content to Rdst content, then stores the sum in Rdst and finally increments Rsrc by 4 (i.e. auto-increment mode). In bit 10 and 9, bit- patterns 11, 10, 01 and 00 denote indexed, auto-decrement, auto-increment and register modes respectively. For each of these modes, bit 8 is used to specify the indirect version. The processor has 16 registers that can be used for addressing purposes; each specified using a 4- bit-code (Figure 7.21). There are 2 stages of decoding:

1) The microinstruction field must be decoded to determine that an Rsrc or Rdst register is involved.

2) The decoded output is then used to gate the contents of the Rsrc or Rdst fields in the IR into a second decoder, which produces the gating-signals for the actual registers R0 to R15.



Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {μPC ← 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {μPC ← 170; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21 Microinstruction for Add (Rsrc)+,Rdst.

Microinstructions With Next-Address Fields

Drawback of previous organization:

□ The microprogram requires several branch microinstructions which perform no useful operation. Thus, they detract from the operating-speed of the computer.

Solution:

□ Include an address-field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched. (Thus, every microinstruction becomes a branch microinstruction).

The flexibility of this approach comes at the expense of additional bits for the address-field (Figure). Advantage: Separate branch microinstructions are virtually eliminated. (Figure). *Disadvantage:* Additional bits for the address field (around 1/6).

• There is no need for a counter to keep track of sequential address. Hence, μPC is replaced with μAR.

• The next-address bits are fed through the OR gate to the μAR, so that the address can be modified on the basis of the data in the IR, external inputs and condition-codes.

• The decoding circuits generate the starting-address of a given microroutine on the basis of the opcode in the IR. (μAR Microinstruction Address Register).

Prefetching Microinstructions

Disadvantage of Microprogrammed Control: Slower operating-speed because of the time it takes to fetch microinstructions from the control-store.

Solution: Faster operation is achieved if the next microinstruction is pre-fetched while the current one is being executed.

Emulation

• The main function of microprogrammed control is to provide a means for simple, flexible and relatively inexpensive execution of machine instruction.

• Its flexibility in using a machine's resources allows diverse classes of instructions to be implemented.

• Suppose we add to the instruction-repository of a given computer M1, an entirely new set of instructions that is in fact the instruction-set of a different computer M2.

• Programs written in the machine language of M2 can be then be run on computer M1 i.e. M1 emulates M2.

• Emulation allows us to replace obsolete equipment with more up-to-date machines.

• If the replacement computer fully emulates the original one, then no software changes have to be made to run existing programs.

• Emulation is easiest when the machines involved have similar architectures.

2. COMPUTER ARITHMETIC

Addition and subtraction of two numbers are basic operations at the machine- instruction level in all computers. These operations, as well as other arithmetic and logic operations, are implemented in the arithmetic and logic unit (ALU) of the processor. In this chapter, we present the logic circuits used to implement arithmetic operations. The time needed to perform addition or subtraction affects the processor’s performance. Multiply and divide operations, which require more complex circuitry than either addition or subtraction operations, also affect performance. We present some of the techniques used in modern computers to perform arithmetic operations at high speed. Operations on floating-point numbers are also described.

ADDITION AND SUBTRACTION OF SIGNED NUMBERS

Figure shows the truth table for the sum and carry-out functions for adding equally weighted bits xi and yi in two numbers X and Y . The figure also shows logic expressions for these functions, along with an example of addition of the 4-bit unsigned numbers 7 and 6. Note that each stage of the addition process must accommodate a carry-in bit. We use ci to represent the carry-in to stage i, which is the same as the carry-out from stage (i − 1). The logic expression for si in Figure 9.1 can be implemented with a 3-input XOR gate, used in Figure a as part of the logic required for a single stage of binary addition. The carry-out function, ci+1, is implemented with an AND-OR circuit, as shown. A convenient symbol for the complete circuit for a single stage of addition, called a full adder (FA), is also shown in the figure.

A cascaded connection of n full-adder blocks can be used to add two n-bit numbers, as shown in Figure b. Since the carries must propagate, or ripple, through this cascade, the configuration is called a ripple-carry adder. The carry-in, c0, into the least-significant-bit (LSB) position provides a convenient means of adding 1 to a number. For instance, forming the 2’s-complement of a number involves adding 1 to the 1’s-complement of the number. The carry signals are also useful for interconnecting k adders to form an adder capable of handling input numbers that are kn bits long, as shown in Figure c.

Addition/Subtraction Logic Unit

The n-bit adder in Figure 9.2b can be used to add 2’s-complement numbers X and Y , where the xn−1 and yn−1 bits are the sign bits. The carry-out bit cn is not part of the answer. It occurs when the signs of the two operands are the same, but the sign of the result is different. Therefore, a circuit to detect overflow can be added to the n-bit adder by implementing the logic expression

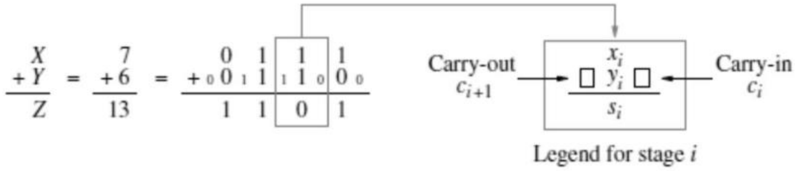
$$\text{Overflow} = x_{n-1}y_{n-1}\overline{s}_{n-1} + \overline{x}_{n-1}\overline{y}_{n-1}s_{n-1}$$

It can also be shown that overflow occurs when the carry bits cn and cn−1 are different. Therefore, a simpler circuit for detecting overflow can be obtained by implementing the expression cn ⊕ cn−1 with an XOR gate.

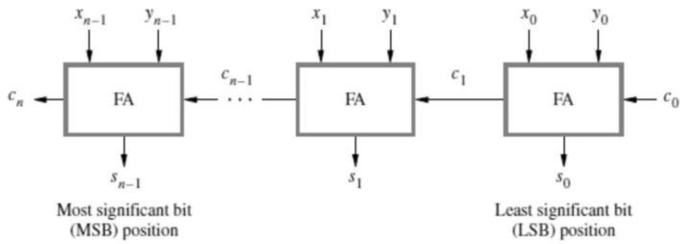
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x}_i\overline{y}_ic_i + \overline{x}_iy_i\overline{c}_i + x_i\overline{y}_i\overline{c}_i + x_iy_ic_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_ic_i + x_ic_i + x_iy_i$$

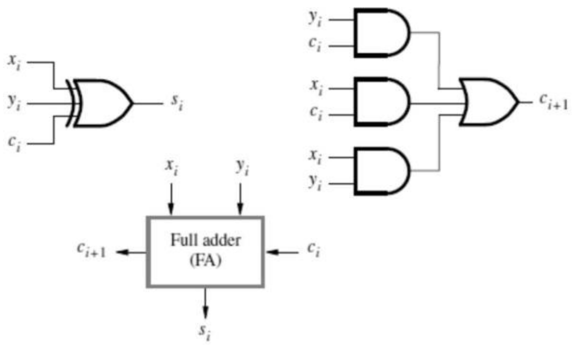
Example:



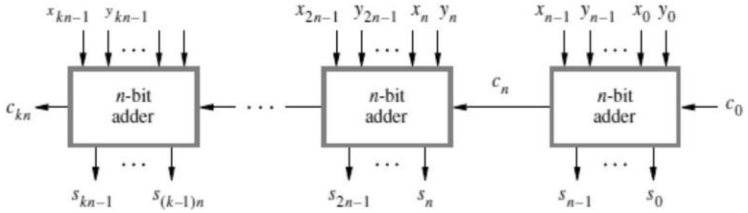
In order to perform the subtraction operation X − Y on 2’s-complement numbers X and Y , we form the 2’s-complement of Y and add it to X . The logic circuit shown in Figure can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying Y unchanged to one of the adder inputs along with a carry-in signal, c0, of 0. When the Add/Sub control line is set to 1, the Y number is 1’s-complemented (that is, bit-complemented) by the XOR gates and c0 is set to 1 to complete the 2’s-complementation of Y . Recall that 2’s-complementing a negative number is done in exactly the same manner as for a positive number. An XOR gate can be added to Figure to detect the overflow condition cn ⊕ cn−1.



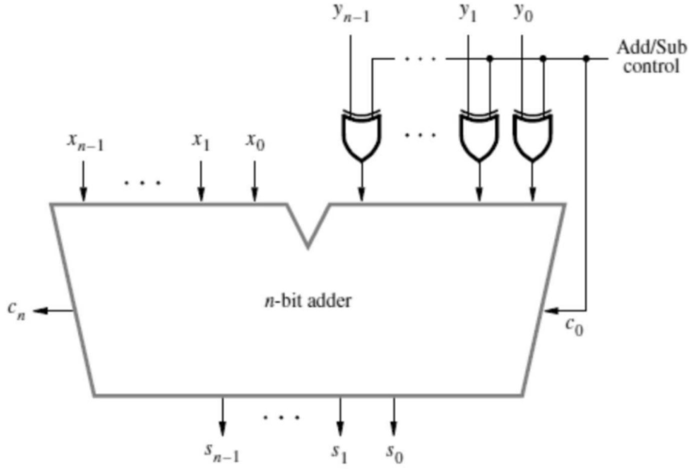
(b) An *n*-bit ripple-carry adder



(a) Logic for a single stage



(c) Cascade of k n-bit adders



Design of Fast Adders

If an n-bit ripple-carry adder is used in the addition/subtraction circuit, it may have too much delay in developing its outputs, s0 through sn−1 and cn. Whether or not the delay incurred is acceptable can be decided only in the context of the speed of other processor components and the data transfer times of registers and cache memories. The delay through a network of logic gates depends on the integrated circuit electronic technology used in fabricating the network and on the number of gates in the paths from inputs to outputs. The delay through any combinational circuit constructed from gates in a particular technology is determined by adding up the number of logic-gate delays along the longest signal propagation path through the circuit. In the case of the n-bit ripple-carry adder, the longest path is from inputs x0, y0, and c0 at the LSB position to outputs cn and sn−1 at the most-significant-bit (MSB) position.

Using the implementation indicated in Figure a, cn−1 is available in 2(n−1) gate delays, and sn−1 is correct oneXORgate delay later. The final carry-out, cn, is available after 2n gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit shown in Figure 9.3, all sum bits are available in 2n gate delays, including the delay through the XOR gates on the Y input. Using the implementation cn ⊕ cn−1 for overflow, this indicator is available after 2n + 2 gate delays.

Two approaches can be taken to reduce delay in adders. The first approach is to use the fastest possible electronic technology. The second approach is to use a logic gate network called a carry-lookahead network, which is described in the ***Previous VLSI design Notes***.

MULTIPLICATION ALGORITHM

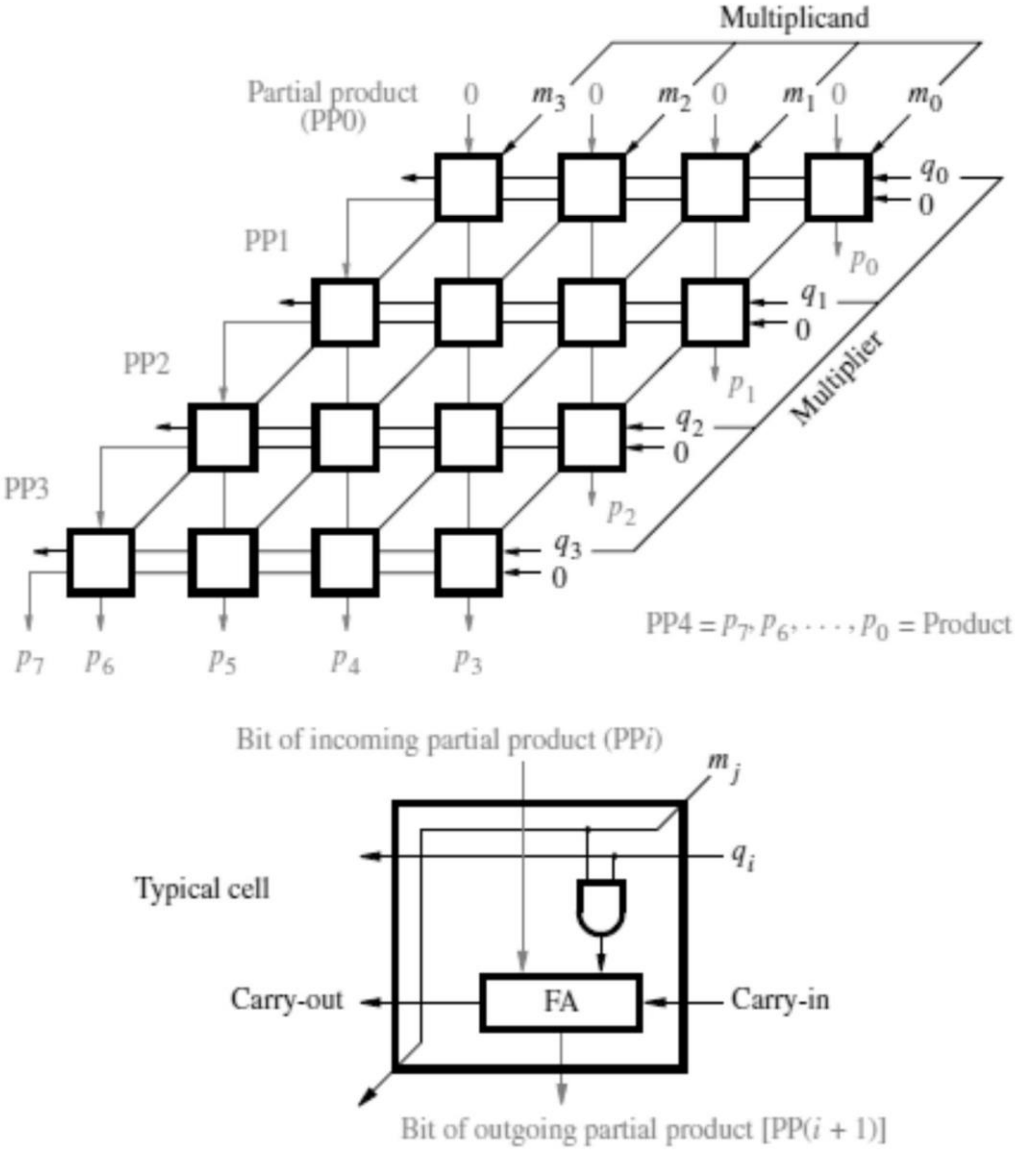
The usual algorithm for multiplying integers by hand is illustrated for the binary system. The product of two, unsigned, n-digit numbers can be accommodated in 2n digits, so the product of the two 4-bit numbers in this example is accommodated in 8bits, as shown. In the binary system, multiplication of the multiplicand by one bit of the multiplier is easy. If the multiplier bit is 1, the multiplicand is entered in the appropriate shifted position. If the multiplier bit is 0, then 0s are entered, as in the third row of the example. The product is computed one bit at a time by adding the bit columns from right to left and propagating carry values between columns.

Array Multiplier for Unsigned Numbers

Binary multiplication of unsigned operands can be implemented in a combinational, two-dimensional, logic array, as shown in Figure b for the 4-bit operand case. The main component in each cell is a full adder, FA. The AND gate in each cell determines whether a multiplicand bit, mj , is added to the incoming partial-product bit, based on the value of the multiplier bit, qi . Each row i, where 0 ≤ i ≤ 3, adds the multiplicand (appropriately shifted) to the incoming partial product, PPi, to generate the outgoing partial product, PP(i + 1), if qi = 1. If qi = 0, PPi is passed vertically downward unchanged. PP0 is all 0s, and PP4 is the desired product. The multiplicand is shifted to the left one position per row by the diagonal signal path. We note that the row-by-row addition done in the array circuit differs from the usual hand addition described previously, which is done column-by-column.

```
          1 1 0 1      (13)  Multiplicand M
        × 1 0 1 1      (11)  Multiplier Q
        ---------
          1 1 0 1
        1 1 0 1
      0 0 0 0
    1 1 0 1
  ---------------
  1 0 0 0 1 1 1 1      (143)  Product P
```

(a) Manual multiplication algorithm

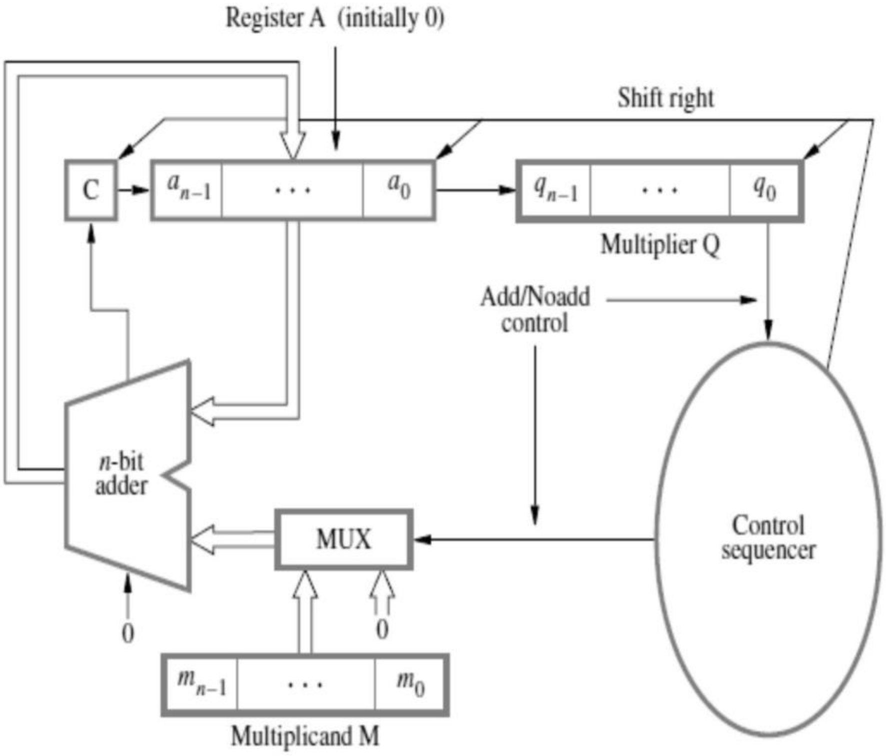


(b) Array implementation

The worst-case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array. This critical path consists of the staircase pattern that includes the two cells at the right end of each row, followed by all the cells in the bottom row. Assuming that there are two gate delays from the inputs to the outputs of a full-adder block, FA, the critical path has a total of $6(n - 1) - 1$ gate delays, including the initial AND gate delay in all cells, for an n × n array. (See Problem 9.8.) In the first row of the array, no full adders are needed, because the incoming partial product PP0 is zero. This has been taken into account in developing the delay expression.

Sequential Circuit Multiplier

The combinational array multiplier just described uses a large number of logic gates for multiplying numbers of practical size, such as 32- or 64-bit numbers. Multiplication of two n-bit numbers can also be performed in a sequential circuit that uses a single n-bit adder. The block diagram in Figure a shows the hardware arrangement for sequential multiplication. This circuit performs multiplication by using a single n-bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders in Figure b. Registers A and Q are shift registers, concatenated as shown. Together, they hold partial product PPi while multiplier bit qi generates the signal Add/Noadd. This signal causes the multiplexer MUX to select 0 when qi = 0, or to select the multiplicand M when qi = 1, to be added to PPi to generate PP(i + 1). The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, PP0, of n 0s in register A. The carry-out from the adder is stored in flip- flop C, shown at the left end of register A. At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit qi appears at the LSB position of Q to generate the Add/Noadd signal at the correct time, starting with q0 during the first cycle, q1 during the second cycle, and so on. After they are used, the multiplier bits are discarded by the right-shift operation. Note that the carry-out from the adder is the leftmost bit of PP(i + 1), and it must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low-order half is in register Q. The multiplication example of Figure 9.6a is shown in Figure b as it would be performed by this hardware arrangement.



(a) Register configuration

M: 1 1 0 1

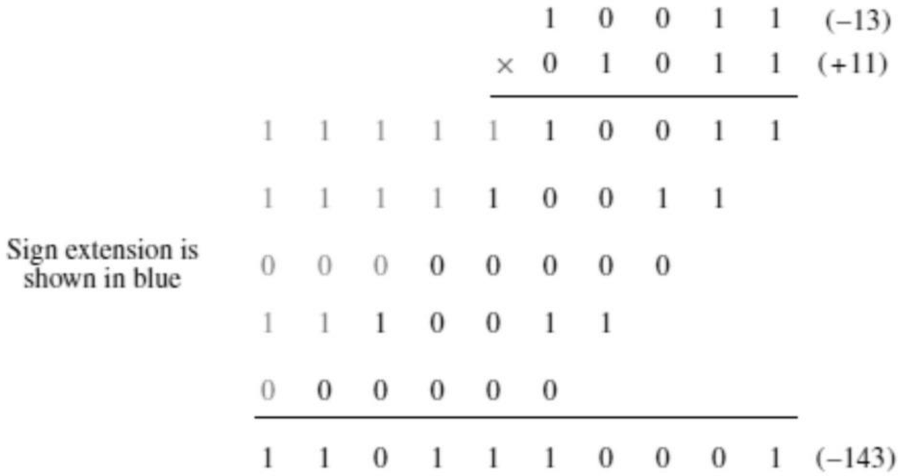
C	A	Q		
0	0 0 0 0	1 0 1 1		Initial configuration
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

Product

(b) Multiplication example

Multiplication of Signed Numbers

We now discuss multiplication of 2’s-complement operands, generating a double- length product. The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits. First, consider the case of a positive multiplier and a negative multiplicand. When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend. Figure shows an example in which a 5-bit signed operand, −13, is the multiplicand. It is multiplied by +11 to get the 10-bit product, −143. The sign extension of the multiplicand is shown in blue. The hardware discussed earlier can be used for negative multiplicands if it is augmented to provide for sign extension of the partial products.



```
                    1 0 0 1 1   (−13)
                  × 0 1 0 1 1   (+11)
                  -----------
          1 1 1 1 1 1 0 0 1 1
          1 1 1 1 1 0 0 1 1
          0 0 0 0 0 0 0 0
          1 1 1 0 0 1 1
          0 0 0 0 0 0
          -------------------
          1 1 0 1 1 1 0 0 0 1   (−143)
```

Sign extension is shown in blue

For a negative multiplier, a straightforward solution is to form the 2’s-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. A technique that works equally well for both negative and positive multipliers, called the Booth algorithm.

The Booth Algorithm

The Booth algorithm [1] generates a 2n-bit product and treats both positive and negative 2’scomplement n-bit operands uniformly. To understand the basis of this algorithm, consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110. To derive the product, we could add four appropriately shifted versions of the multiplicand, as in the standard procedure. However, we can reduce the number of required operations by regarding this multiplier as the difference between two numbers:

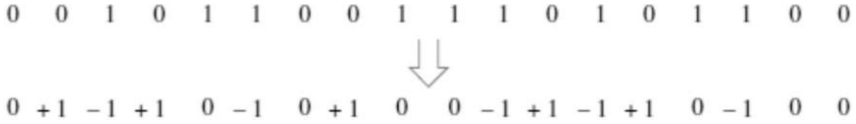
```
    0100000    (32)
 -  0000010     (2)
 ------------------
    0011110    (30)
```

This suggests that the product can be generated by adding 25 times the multiplicand to the 2's-complement of 21 times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as 0 +1 0 0 0 −1 0. In general, in the Booth algorithm, −1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left. Figure illustrates the normal and the Booth algorithms for the example just discussed. The Booth algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is considered a block. Figure shows another example of recoding a multiplier. The case when the least significant bit of the multiplier is 1 is handled by assuming that an implied 0 lies to its right. The Booth algorithm can also be used directly for negative multipliers, as shown in Figure.

```
                     0  1  0  1  1  0  1
                     0  0 +1 +1 +1 +1  0
                     -------------------
                     0  0  0  0  0  0  0
                  0  1  0  1  1  0  1
               0  1  0  1  1  0  1
            0  1  0  1  1  0  1
         0  1  0  1  1  0  1
      0  0  0  0  0  0  0
   0  0  0  0  0  0  0
   -------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0

                     0  1  0  1  1  0  1
                     0 +1  0  0  0 -1  0
                     -------------------
0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  1  1  0  1  0  0  1  1      ←  2's complement of
0  0  0  0  0  0  0  0  0  0  0  0             the multiplicand
0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  1  0  1  1  0  1
0  0  0  0  0  0  0  0
----------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

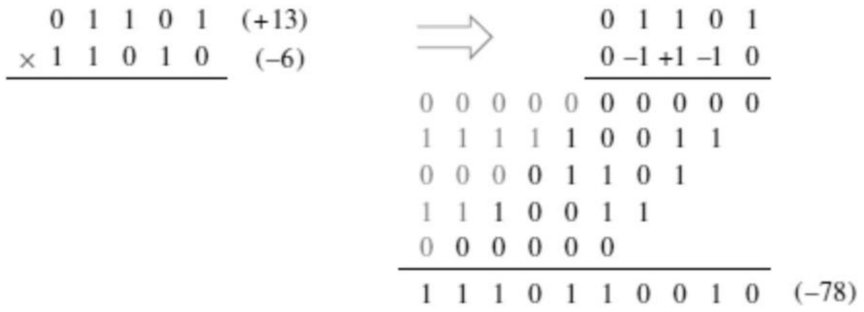
Normal and Booth multiplication schemes

To demonstrate the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2's-complement system.



```
0  0  1  0  1  1  0  0  1  1  1  0  1  0  1  1  0  0
                        ⇓
0 +1 -1 +1  0 -1  0 +1  0  0 -1 +1 -1 +1  0 -1  0  0
```

Booth recoding of a multiplier



```
  0 1 1 0 1   (+13)                  0 1 1 0 1
× 1 1 0 1 0   (-6)      ⟹           0 -1 +1 -1 0
-------------                      ---------------
                        0 0 0 0 0 0 0 0 0 0
                        1 1 1 1 1 0 0 1 1
                        0 0 0 0 1 1 0 1
                        1 1 1 0 0 1 1
                        0 0 0 0 0 0
                        -------------------
                        1 1 1 0 1 1 0 0 1 0   (-78)
```

Booth multiplication with a negative multiplier

Suppose that the leftmost 0 of a negative number, X, is at bit position k, that is,

$$X = 11\ldots10x_{k-1}\ldots x_0$$

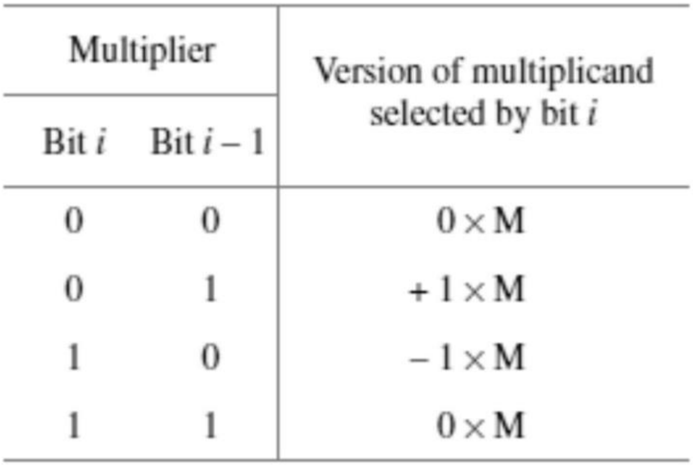
Then the value of X is given by

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \cdots + x_0 \times 2^0$$

The correctness of this expression for $V(X)$ is shown by observing that if X is formed as the sum of two numbers, as follows,

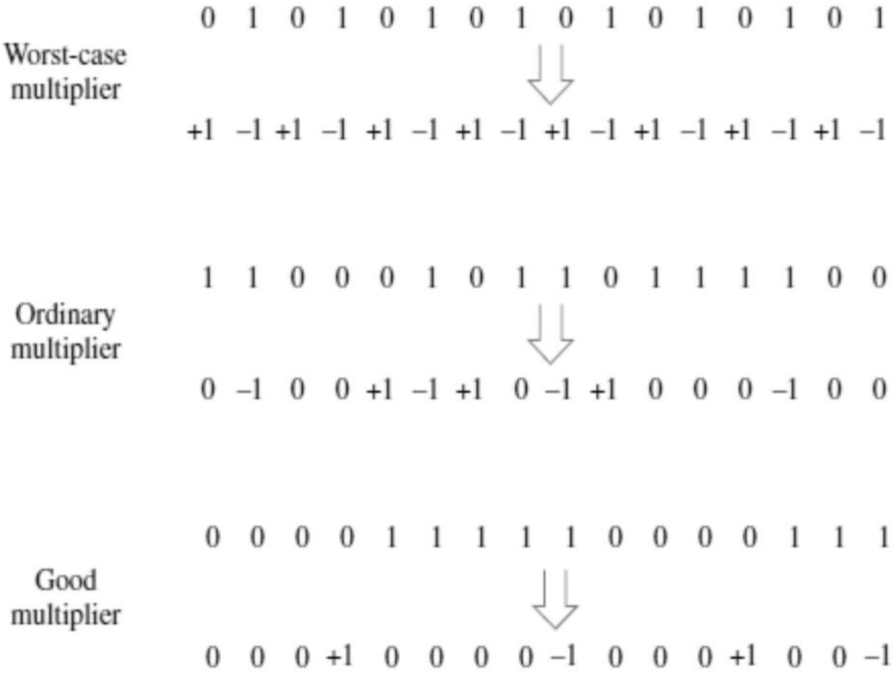
$$\begin{array}{rl} & 11\ldots100000\ldots0 \\ + & 00\ldots00x_{k-1}\ldots x_0 \\ \hline X = & 11\ldots10x_{k-1}\ldots x_0 \end{array}$$

then the upper number is the 2's-complement representation of −2k+1. The recoded multiplier now consists of the part corresponding to the lower number, with −1 added in position k + 1. For example, the multiplier 110110 is recoded as 0 −1 +1 0 −1 0. The Booth technique for recoding multipliers is summarized in Figure. The transformation 011 . . . 110⇒+1 0 0. . . 0 −1 0 is called skipping over 1s. This term is derived from the case in which the multiplier has its 1s grouped into a few contiguous blocks. Only a few versions of the shifted multiplicand (the summands) need to be added to generate the product, thus speeding up the multiplication operation. However, in the worst case—that of alternating 1s and 0s in the multiplier—each bit of the multiplier selects a summand. In fact, this results in more summands than if the Booth algorithm were not used. A 16-bit worst-case multiplier, an ordinary multiplier, and a good multiplier are shown in Figure.



Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Booth multiplier recoding table



Booth recoded multipliers

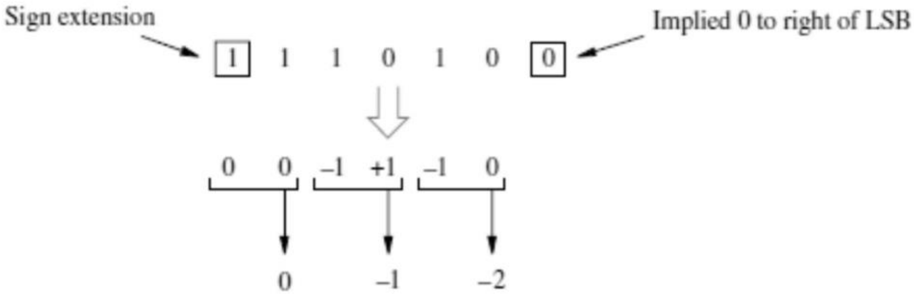
The Booth algorithm has two attractive features. First, it handles both positive and negative multipliers uniformly. Second, it achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s.

Fast Multiplication

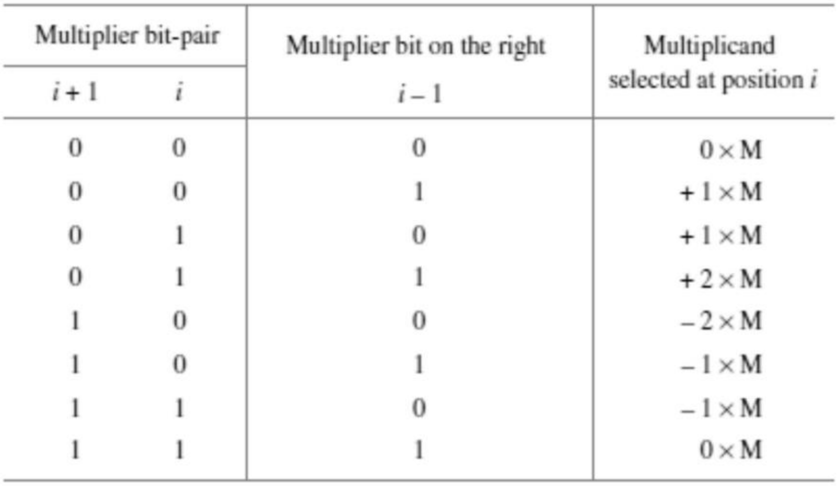
We now describe two techniques for speeding up the multiplication operation. The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is n/2 for n-bit operands. The second technique leads to adding the summands in parallel.

Bit-Pair Recoding of Multipliers

A technique called bit-pair recoding of the multiplier results in using at most one summand for each pair of bits in the multiplier. It is derived directly from the Booth algorithm. Group the Booth-recoded multiplier bits in pairs, and observe the following.



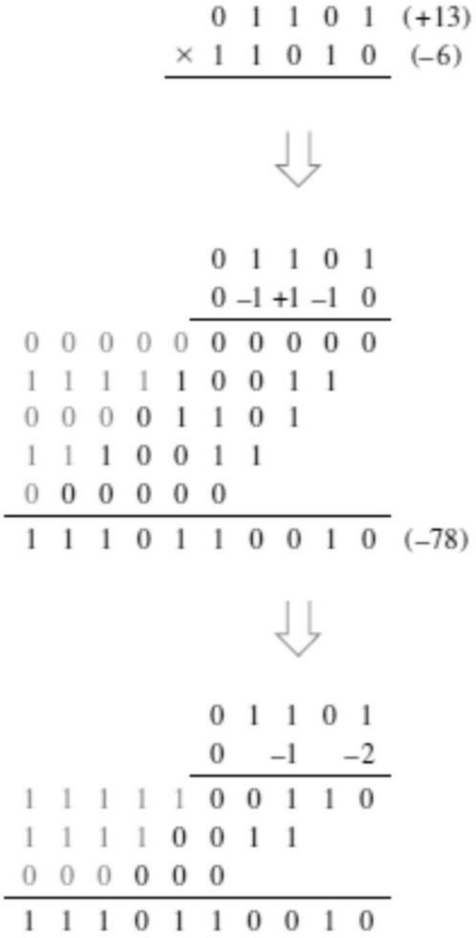
(a) Example of bit-pair recoding derived from Booth recoding



Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Table of multiplicand selection decisions

The pair (+1 −1) is equivalent to the pair (0 +1). That is, instead of adding −1 times the multiplicand M at shift position i to +1 ×M at position i + 1, the same result is obtained by adding +1 ×M at position i. Other examples are: (+1 0) is equivalent to (0 +2), (−1 +1) is equivalent to (0 −1), and so on. Thus, if the Booth-recoded multiplier is examined two bits at a time, starting from the right, it can be rewritten in a form that requires at most one version of the multiplicand to be added to the partial product for each pair of multiplier bits.



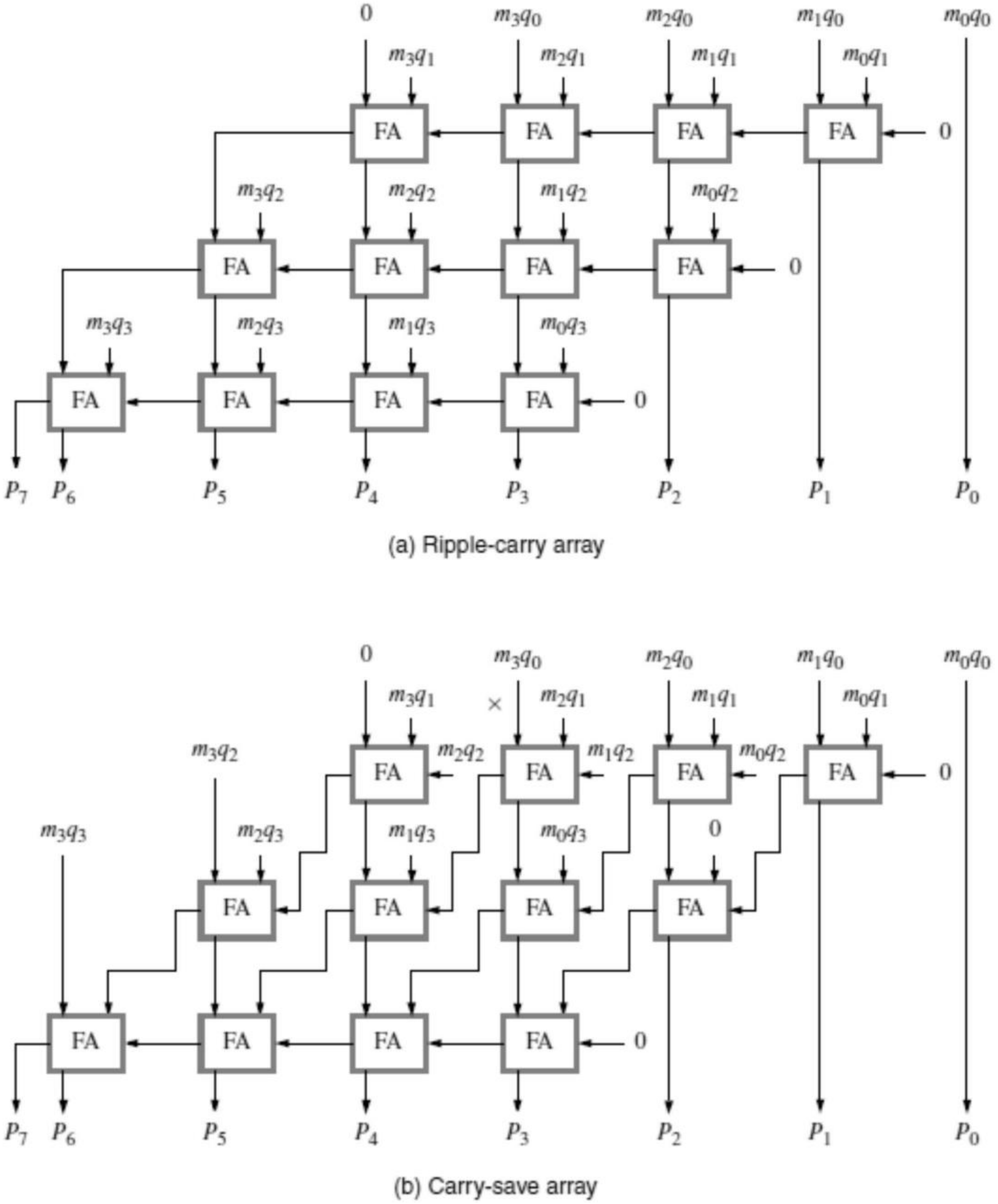
```
                 0  1  1  0  1   (+13)
               × 1  1  0  1  0   (−6)

                 0  1  1  0  1
                 0 −1 +1 −1  0
 0  0  0  0  0   0  0  0  0  0
 1  1  1  1  1   0  0  1  1
 0  0  0  0  1   1  0  1
 1  1  1  0  0   1  1
 0  0  0  0  0   0
 1  1  1  0  1   1  0  0  1  0   (−78)

                 0  1  1  0  1
                 0    −1    −2
 1  1  1  1  1   0  0  1  1  0
 1  1  1  1  0   0  1  1
 0  0  0  0  0   0
 1  1  1  0  1   1  0  0  1  0
```

Multiplication requiring only n/2 summands

Figure a shows an example of bit-pair recoding of the multiplier in Figure and Figure b shows a table of the multiplicand selection decisions for all possibilities. The multiplication operation in Figure is shown in Figure as it would be computed using bit-pair recoding of the multiplier.

Carry-Save Addition of Summands

Multiplication requires the addition of several summands. A technique called carry- save addition (CSA) can be used to speed up the process. Consider the 4 × 4 multiplication array shown in Figure 9.16a. This structure is in the form of the array shown in Figure, in which the first row consists of just theAND gates that produce the four inputs m3q0, m2q0, m1q0, and m0q0. Instead of letting the carries ripple along the rows, they can be “saved” and introduced into the next row, at the correct weighted positions, as shown in Figure 9.16b. This frees up an input to each of three full adders in the first row. These inputs can be used to introduce the third summand bits m2q2, m1q2, and m0q2.



(a) Ripple-carry array

(b) Carry-save array

Ripple-carry and carry-save arrays for a 4 × 4 multiplier

Now, two inputs of each of three full adders in the second row are fed by the sum and carry outputs from the first row. The third input is used to introduce the bits m2q3, m1q3, and m0q3 of the fourth summand. The high-order bits m3q2 and m3q3 of the third and fourth summands are introduced into the remaining free full-adder inputs at the left end in the second and third rows. The saved carry bits and the sum bits from the second row are now added in the third row, which is a ripple-carry adder, to produce the final product bits. The delay through the carry-save array is somewhat less than the delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.

DIVISION ALGORITHM

We discussed the multiplication of unsigned numbers by relating the way the multiplication operation is done manually to the way it is done in a logic circuit. We use the same approach here in discussing integer division. We discussed unsigned-number division in detail, and then make some general comments on the signed-number case.

Integer Division

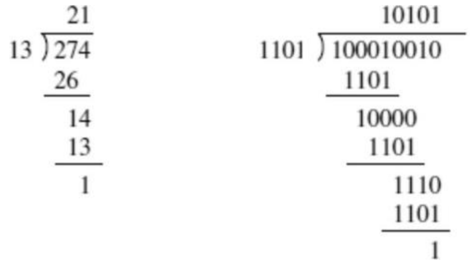
Figure shows examples of decimal division and binary division of the same values. Consider the decimal version first. The 2 in the quotient is determined by the following reasoning: First, we try to divide 13 into 2, and it does not work. Next, we try to divide 13 into

27. We go through the trial exercise of multiplying 13 by 2 to get 26, and, observing that 27 − 26 = 1 is less than 13, we enter 2 as the quotient and perform the required subtraction. The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once, and the remainder is 1. We can discuss binary division in a similar way, with the simplification that the only possibilities for the quotient bits are 0 and 1.

A circuit that implements division by this longhand method operates as follows: It positions the divisor appropriately with respect to the dividend and performs a subtraction. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed. If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction. This is called the restoring division algorithm.

Restoring Division

Figure shows a logic circuit arrangement that implements the restoring division algorithm just discussed. Note its similarity to the structure for multiplication shown in Figure. An n-bit positive divisor is loaded into register and an n-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2’s-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.



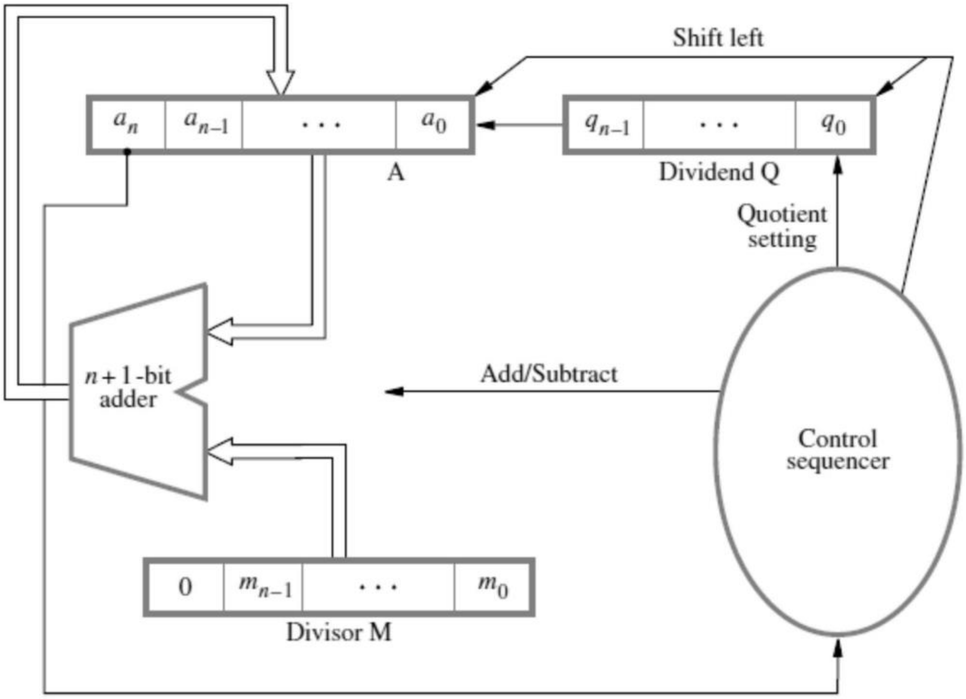
The following algorithm performs restoring division. Do the following three steps n times:

1. Shift A and Q left one bit position.

2. Subtract M from A, and place the answer back in A.

3. If the sign of A is 1, set q0 to 0 and add M back to A (that is, restore A); otherwise, set q0 to 1.

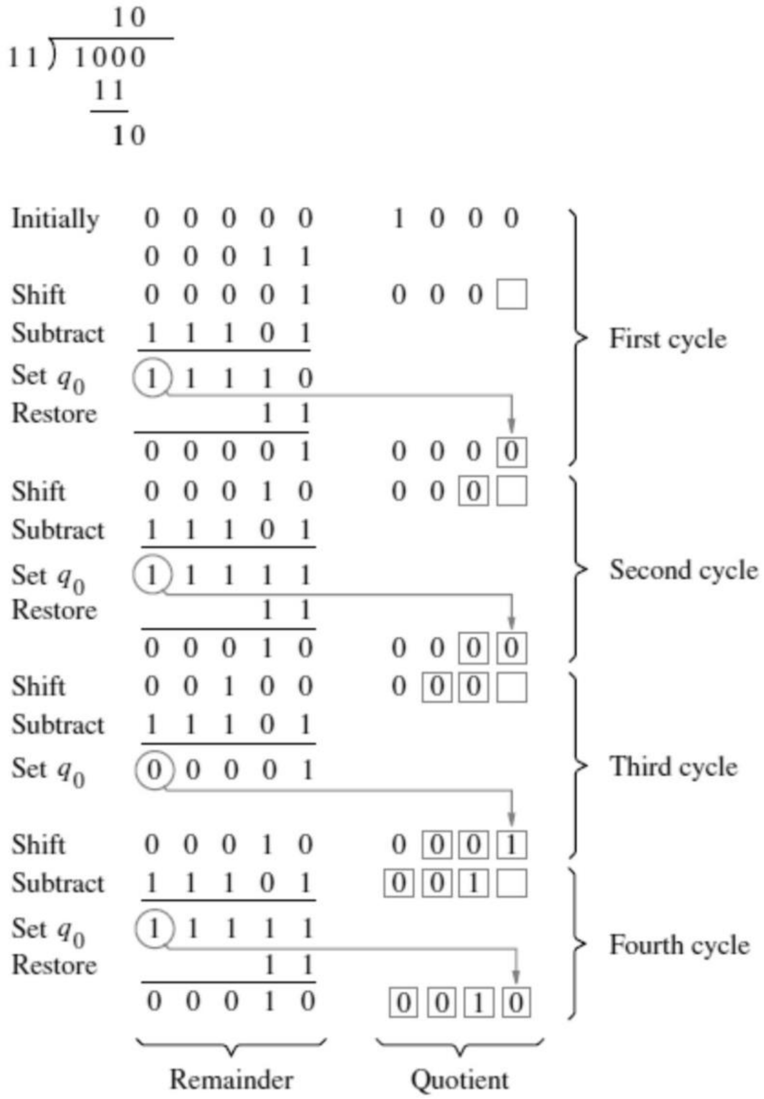
Figure shows a 4-bit example as it would be processed by the circuit in Figure.



Non-Restoring Division

The restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative.

Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform 2A− M. IfAis negative, we restore it by performing A+ M, and then we shift it left and subtract M. This is equivalent to performing 2A+ M. The q0 bit is appropriately set to 0 or 1 after the correct operation has been performed.



We can summarize this in the following algorithm for non-restoring division.

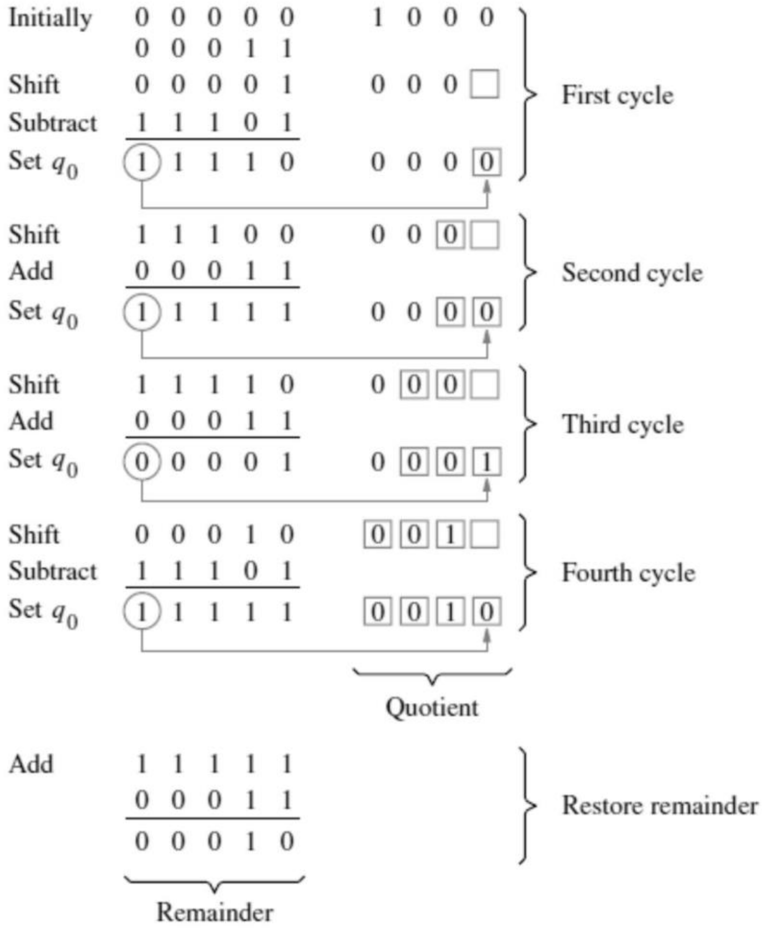
Stage 1: Do the following two steps n times:

1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.

2. Now, if the sign of A is 0, set q0 to 1; otherwise, set q0 to 0.

Stage 2: If the sign of A is 1, add M to A.

Stage 2 is needed to leave the proper positive remainder in A after the n cycles of Stage 1. The logic circuitry in Figure can also be used to perform this algorithm, except that the Restore operations are no longer needed. One Add or Subtract operation is performed in each of the n cycles of stage 1, plus a possible final addition in Stage 2. Figure shows how the division example in Figure is executed by the non-restoring division algorithm.



There are no simple algorithms for directly performing division on signed operands that are comparable to the algorithms for signed multiplication. In division, the operands can be preprocessed to change them into positive values. After using one of the algorithms just discussed, the signs of the quotient and the remainder are adjusted as necessary.

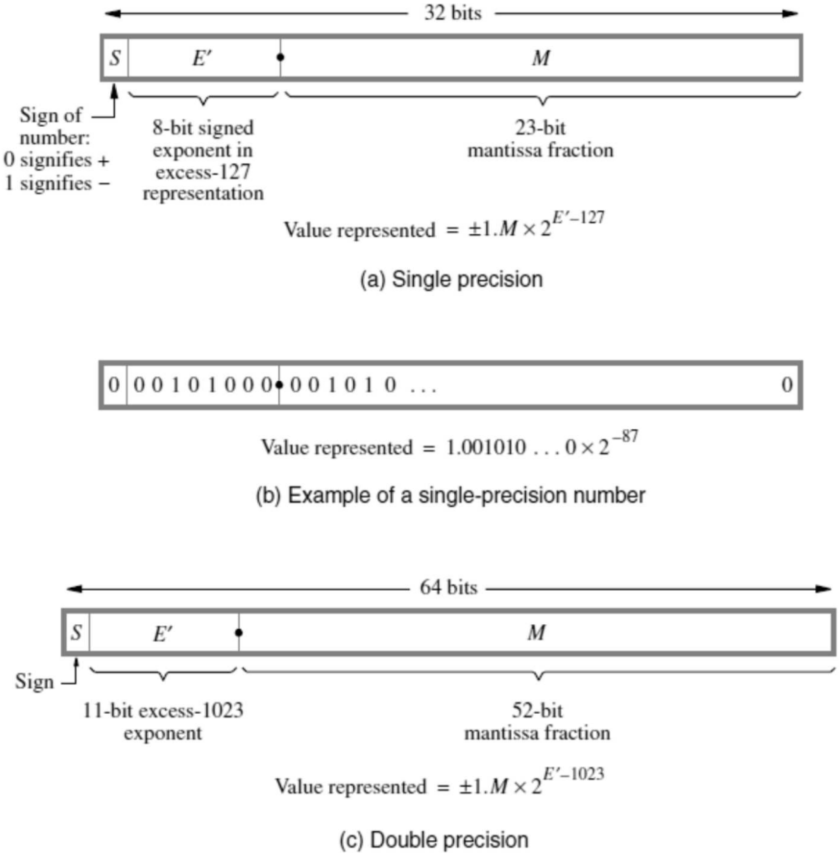
3.FLOATING POINT ARITHMETIC OPERATIONS

We know that the floating-point numbers and indicated how they can be represented in a 32-bit binary format. Now, we provide more detail on representation formats and arithmetic operations on floating-point numbers. The descriptions provided here are based on the 2008 version of IEEE (Institute of Electrical and Electronics Engineers) Standard.

DECIMAL ARITHMETIC OPERATIONS

Recalling from Chapter 1 that a binary floating-point number can be represented by

- A sign for the number
- Some significant bits
- A signed scale factor exponent for an implied base of 2


32 bits
S
E′
M
Sign of number: 0 signifies + 1 signifies −
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction
Value represented = $\pm 1.M \times 2^{E'-127}$
0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 ... 0
Value represented = $1.001010\ldots0 \times 2^{-87}$
64 bits
S
E′
M
Sign
11-bit excess-1023 exponent
52-bit mantissa fraction
Value represented = $\pm 1.M \times 2^{E'-1023}$

(a) Single precision

(b) Example of a single-precision number

(c) Double precision

The basic IEEE format is a 32-bit representation, shown in Figure a. The leftmost bit represents the sign, S, for the number. The next 8 bits, E , represent the signed exponent of the scale factor (with an implied base of 2), and the remaining 23 bits, M, are the fractional part of the significant bits. The full 24-bit string, B, of significant bits, called the mantissa, always has a leading 1, with the binary point immediately to its right. Therefore, the mantissa

$$B = 1.M = 1.b_{-1}b_{-2} \dots b_{-23}$$

has the value

$$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-23} \times 2^{-23}$$

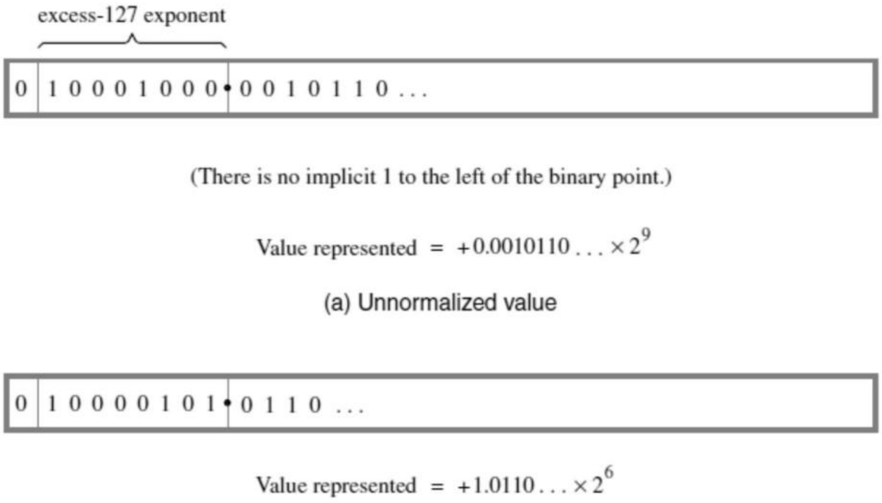
By convention, when the binary point is placed to the right of the first significant bit, the number is said to be normalized. Note that the base, 2, of the scale factor and the leading 1 of the mantissas are both fixed. They do not need to appear explicitly in the representation. Instead of the actual signed exponent, E, the value stored in the exponent field is an unsigned integer E = E + 127. This is called the excess-127 format. Thus, E is in the range 0 ≤ E ≤ 255. The end values of this range, 0 and 255, are used to represent special values, as described later. Therefore, the range of E for normal values is 1 ≤ E ≤ 254.

This means that the actual exponent, E, is in the range −126 ≤ E ≤ 127. The use of the excess-127 representation for exponents simplifies comparison of the relative sizes of two floating-point numbers. The 32-bit standard representation in Figure 9.26a is called a single-precision representation because it occupies a single 32-bit word. The scale factor has a range of 2−126 to 2+127, which is approximately equal to 10±38. The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value. An example of a single-precision floating-point number is shown in Figure b.

To provide more precision and range for floating-point numbers, the IEEE standard also specifies a double-precision format, as shown in Figure c. The double-precision format has increased exponent and mantissa ranges. The 11-bit excess-1023 exponent E has the range 1 ≤ E ≤ 2046 for normal values, with 0 and 2047 used to indicate special values, as before. Thus, the actual exponent E is in the range −1022 ≤ E ≤ 1023, providing scale factors of 2−1022 to 21023 (approximately 10±308). The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

A computer must provide at least single-precision representation to conform to the IEEE standard. Double-precision representation is optional. The standard also specifies certain optional extended versions of both of these formats. The extended versions provide increased precision and increased exponent range for the representation of intermediate values in a sequence of calculations. The use of extended formats helps to reduce the size of the accumulated round-off error in a sequence of calculations leading to a desired result.

For example, the dot product of two vectors of numbers involves accumulating a sum of products. The input vector components are given in a standard precision, either single or double, and the final answer (the dot product) is truncated to the same precision. All intermediate calculations should be done using extended precision to limit accumulation of errors. Extended formats also enhance the accuracy of evaluation of elementary functions such as sine, cosine, and so on. This is because they are usually evaluated by adding up a number of terms in a series representation. In addition to requiring the four basic arithmetic operations, the standard requires three additional operations to be provided: remainder, square root, and conversion between binary and decimal representations.


excess-127 exponent
0
1 0 0 0 1 0 0 0 • 0 0 1 0 1 1 0 . . .
(There is no implicit 1 to the left of the binary point.)
Value represented = +0.0010110 . . . × 2^9
0
1 0 0 0 0 1 0 1 • 0 1 1 0 . . .
Value represented = +1.0110 . . . × 2^6

(a) Unnormalized value

(b) Normalized version

We note two basic aspects of operating with floating-point numbers. First, if a number is not normalized, it can be put in normalized form by shifting the binary point and adjusting the exponent. Figure shows an unnormalized value, 0.0010110 . . . × 29, and its normalized version, 1.0110 . . . × 26. Since the scale factor is in the form 2i , shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent, respectively. Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated. In single precision, this means that its normalized representation requires an exponent less than −126 or greater than +127. In the first case, we say that underflow has occurred, and in the second case, we say that overflow has occurred.

Special Values

The end values 0 and 255 of the excess-127 exponent E are used to represent special values. When E = 0 and the mantissa fraction M is zero, the value 0 is represented. When E = 255 and M = 0, the value ∞ is represented, where ∞ is the result of dividing a normal number by zero. The sign bit is still used in these representations, so there are representations for ±0 and ±∞. When E = 0 and M = 0, denormal numbers are represented. Their value is ±0.M × 2−126. Therefore, they are smaller than the smallest normal number. There is no implied one to the left of the binary point, and M is any nonzero 23-bit fraction. The purpose of introducing denormal numbers is to allow for gradual underflow, providing an extension of the range of normal representable numbers. This is useful in dealing with very small numbers, which may be needed in certain situations. When E = 255 and M = 0, the value represented is called Not a Number (NaN). A NaN represents the result of performing an invalid operation such as 0/0 or √−1.

Exceptions

In conforming to the IEEE Standard, a processor must set exception flags if any of the following conditions arise when performing operations: underflow, overflow, divide by zero, inexact, invalid. We have already mentioned the first three. Inexact is the name for a result that requires rounding in order to be represented in one of the normal formats. An invalid exception occurs if operations such as 0/0 or √−1 are attempted. When an exception occurs, the result is set to one of the special values. If interrupts are enabled for any of the exception flags, system or user-defined routines are entered when the associated exception occurs. Alternatively, the application program can test for the occurrence of exceptions, as necessary, and decide how to proceed.

Arithmetic Operations on Floating-Point Numbers

In this section, we outline the general procedures for addition, subtraction, multiplication, and division of floating-point numbers. The rules given below apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations; for example, the possibility that overflow or underflow might occur is not discussed.

Furthermore, intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. Although we do not provide full details in specifying the rules, we consider some aspects of implementation, including rounding, in later sections. When adding or subtracting floating-point numbers, their mantissas must be shifted with respect to each other if their exponents differ. Consider a decimal example in which we wish to add 2.9400 × 102 to 4.3100 × 104. We rewrite 2.9400 × 102 as 0.0294 ×

104 and then perform addition of the mantissas to get 4.3394 × 104. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Multiply Rule

1. Add the exponents and subtract 127 to maintain the excess-127 representation.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide Rule

1. Subtract the exponents and add 127 to maintain the excess-127 representation.
2. Divide the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

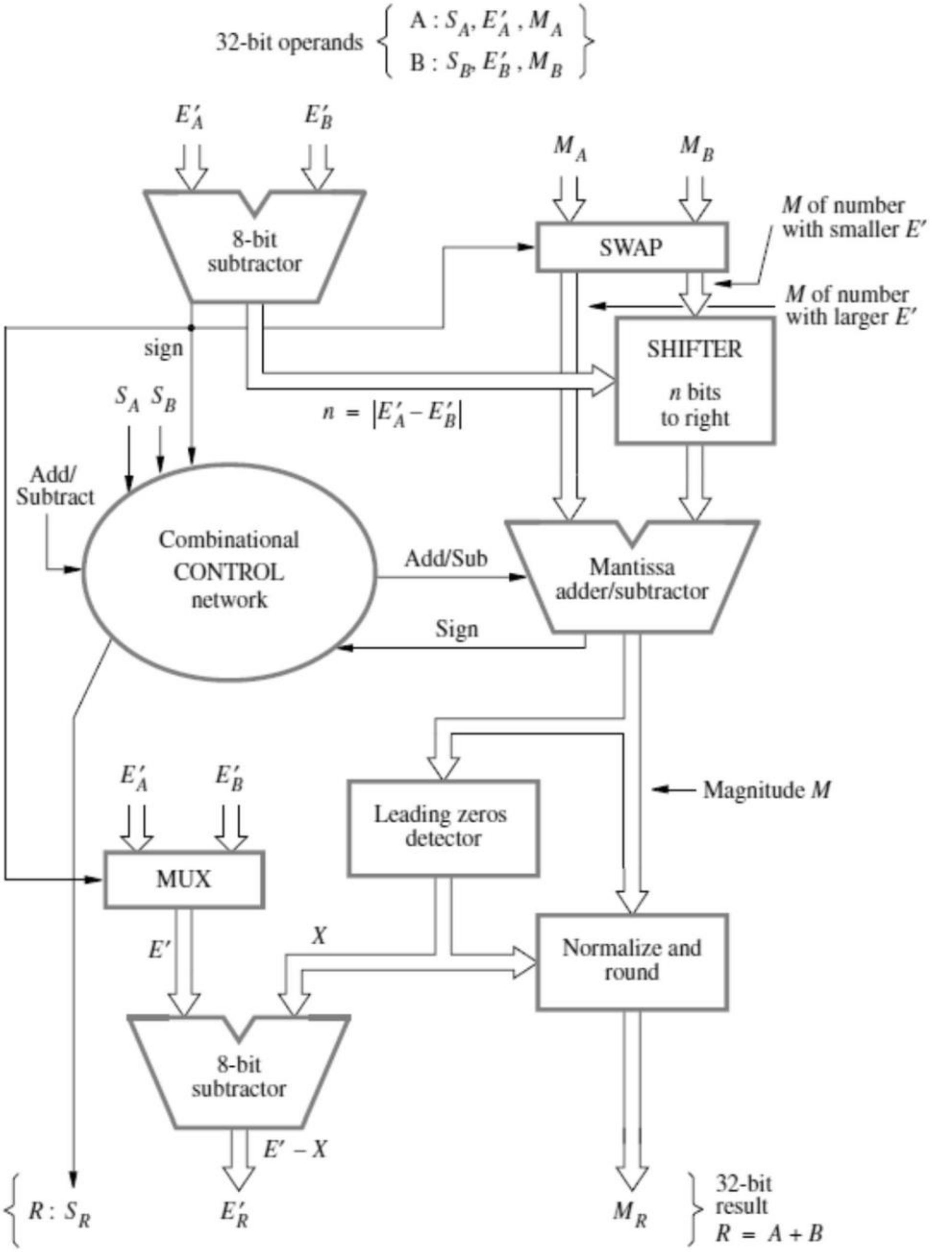
DECIMAL ARITHMETIC UNIT

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by software routines. In either case, the computer must be able to convert input and output from and to the user's decimal representation of numbers. In many general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware.

Implementing Floating-Point Operations

An example of the implementation of floating-point operations is shown in Figure. This is a block diagram of a hardware implementation for the addition and subtraction of 32-bit floating-point operands that have the format shown in Figure a. Following the Add/Subtract rule, we see that the first step is to compare exponents to determine how far to shift the mantissa of the number with the smaller exponent. The shift-count value, n, is determined by the 8-bit subtractor circuit in the upper left corner of the figure. The magnitude of the difference EA−

EB, or n, is sent to the SHIFTER unit. If n is larger than the number of significant bits of the operands, then the answer is essentially the larger operand (except for guard and sticky-bit considerations in rounding), and shortcuts can be taken in deriving the result.



The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. Therefore, in step 1, the sign is sent to the SWAP network in the upper right corner of Figure. If the sign is 0, then EA≥ EB and the mantissas MA and MB are sent straight through the SWAP network. This results in MB being sent to the SHIFTER, to be shifted n positions to the right. The other mantissa, MA, is sent directly to the mantissa adder/subtractor. If the sign is 1, then EA < EB and the mantissas are swapped before they are sent to the SHIFTER.

Step 2 is performed by the two-way multiplexer, MUX, near the bottom left corner of the figure. The exponent of the result, E, is tentatively determined as EA if EA≥ EB, or E B if EA < EB, based on the sign of the difference resulting from comparing exponents in step 1.

Step 3 involves the major component, the mantissa adder/subtractor in the middle of the figure. The CONTROL logic determines whether the mantissas are to be added or subtracted. This is decided by the signs of the operands (SA and SB) and the operation (Add or Subtract) that is to be performed on the operands. The CONTROL logic also determines the sign of the result, SR. For example, if A is negative (SA = 1), B is positive (SB = 0), and the operation is A − B, then the mantissas are added and the sign of the result is negative (SR = 1). On the other hand, if A and B are both positive and the operation is A − B, then the mantissas are subtracted. The sign of the result, SR, now depends on the mantissa subtraction operation. For instance, if EA > EB, then M = MA − (shifted MB) and the resulting number is positive. But if EB > EA, then M = MB − (shifted MA) and the result is negative. This example shows that the sign from the exponent comparison is also required as an input to the CONTROL network. When EA= EB and the mantissas are subtracted, the sign of the mantissa adder/subtractor output determines the sign of the result. The reader should now be able to construct the complete truth table for the CONTROL network.

Step 4 of the Add/Subtract rule consists of normalizing the result of step 3 by shifting M to the right or to the left, as appropriate. The number of leading zeros in M determines the number of bit shifts, X , to be applied to M. The normalized value is rounded to generate the 24-bit mantissa, MR, of the result. The value X is also subtracted from the tentative result exponent E to generate the true result exponent, ER. Note that only a single right shift might be needed to normalize the result. This would be the case if two mantissas of the form 1.xx . . . were added. The vector M would then have the form 1x.xx We have not given any details on the guard bits that must be carried along with intermediate mantissa values. In the IEEE standard, only a few bits are needed, to generate the 24-bit normalized mantissa of the result.

Let us consider the actual hardware that is needed to implement the blocks in Figure. The two 8-bit subtractors and the mantissa adder/subtractor can be implemented by combinational logic, as discussed earlier in this chapter. Because their outputs must be in sign- and-magnitude form, we must modify some of our earlier discussions. A combination of 1's- complement arithmetic and sign-and-magnitude representation is often used. Considerable flexibility is allowed in implementing the SHIFTER and the output normalization operation.

The operations can be implemented with shift registers. However, they can also be built as combinational logic units for high performance.

Introduction

The basic building blocks of a computer are introduced in preceding chapters. In this chapter, we discuss in detail the concept of pipelining, which is used in modern computers to achieve high performance. We begin by explaining the basics of pipelining and how it can lead to improved performance. Then we examine machine instruction features that facilitate pipelined execution, and we show that the choice of instructions and instruction sequencing can have a significant effect on performance. Pipelined organization requires sophisticated compilation techniques and optimizing compilers have been developed for this purpose.

Among other things, such compilers rearrange the sequence of operations to maximize the benefits of pipelined execution.

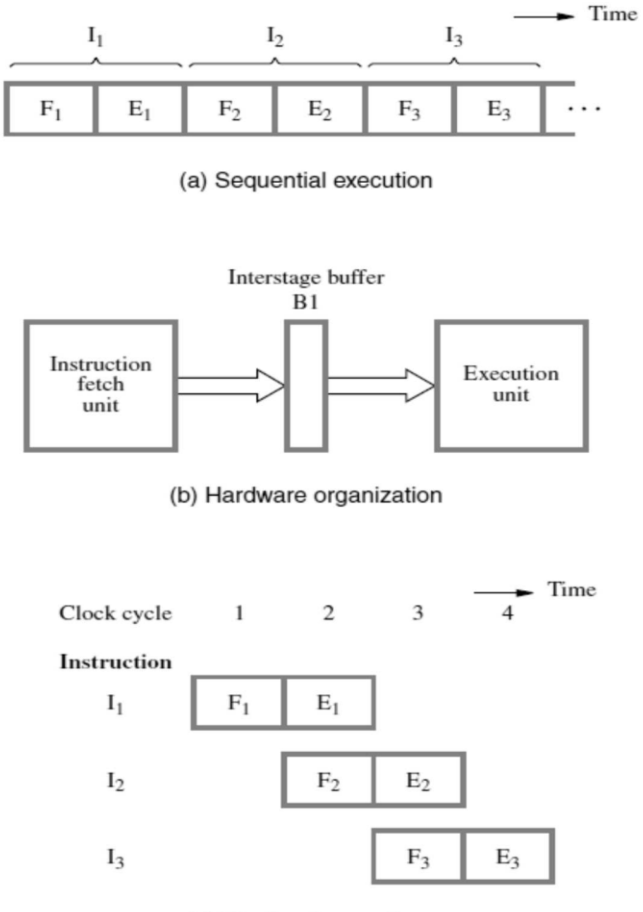
1.BASIC CONCEPTS

The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed. We have encountered concurrent activities several times before. The concept of multiprogramming and explained how it is possible for I/O transfers and computational activities to proceed simultaneously. DMA devices make this possible because they can perform I/O transfers independently once these transfers are initiated by the processor.

Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let Fi and Ei refer to the fetch and execute steps for instruction Ii . Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure a.

Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure b. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labelled “Execution unit.”



(a) Sequential execution

(b) Hardware organization

(c) Pipelined execution

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1c. In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2). Meanwhile, the execution unit performs the operation specified by instruction I1, which is available to it in buffer B1 (step E1). By the end of the second clock cycle, the execution of instruction I1 is completed and instruction I2 is available. Instruction I2 is stored in B1, replacing I1, which is no longer needed. Step E2 is performed by the execution unit during the third clock cycle, while instruction I3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1c can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure a.

In summary, the fetch and execute units in Figure b constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An inter-stage storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

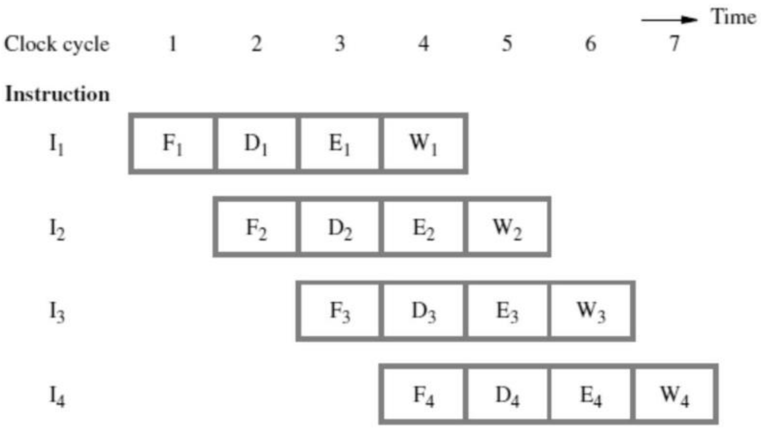
The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory.

D Decode: decode the instruction and fetch the source operand(s).

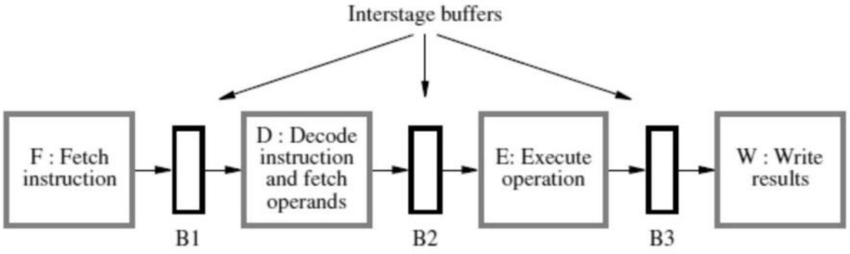
E Execute: perform the operation specified by the instruction.

W Write: store the result in the destination location.



Clock cycle	1	2	3	4	5	6	7
Instruction							
I_1	F_1	D_1	E_1	W_1			
I_2		F_2	D_2	E_2	W_2		
I_3			F_3	D_3	E_3	W_3	
I_4				F_4	D_4	E_4	W_4

(a) Instruction execution divided into four steps



(b) Hardware organization

The sequence of events for this case is shown in Figure a. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure b. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

- Buffer B1 holds instruction I3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer B2 holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I2 (stepW2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction I1.

Role of Cache Memory

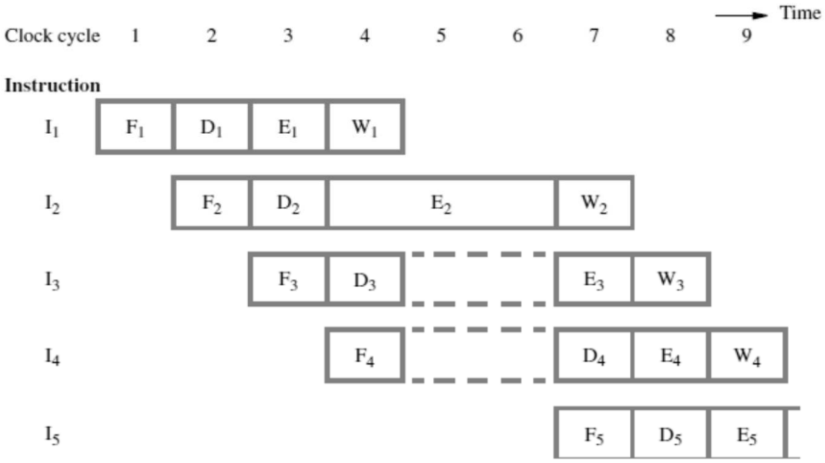
Each stage in a pipeline is expected to complete its operation in one clock cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage. If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving performance if the tasks being performed in different stages require about the same amount of time. This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure a. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each instruction fetch required access to the main memory, pipelining would be of little value.

The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

Pipeline Performance

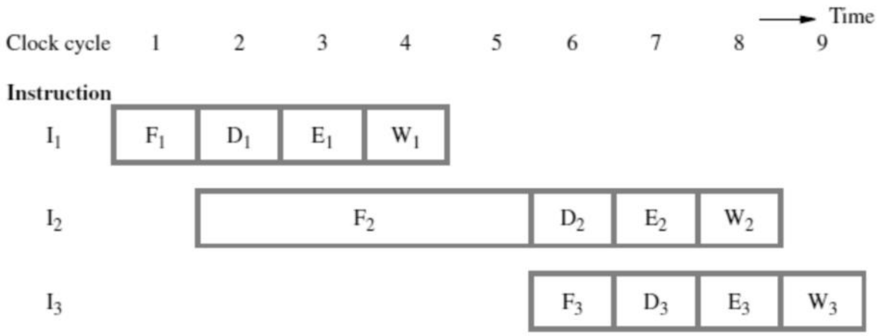
The pipelined processor in Figure completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation as depicted in Figure a could be sustained without interruption throughout program execution. Unfortunately, this is not the case.

For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four- stage pipeline of Figure b is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure shows an example in which the operation specified in instruction I2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D4 and F5 must be postponed as shown.

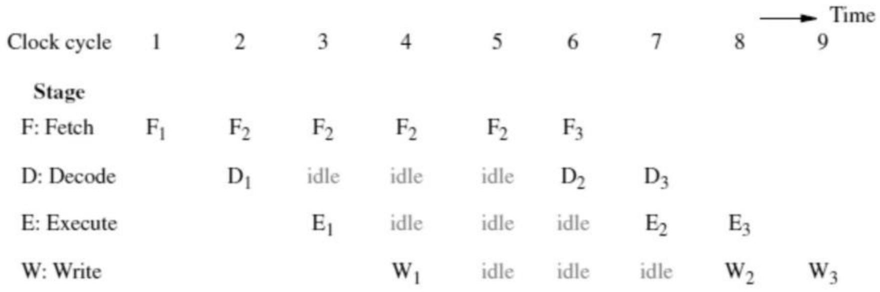


Effect of an execution operation taking more than one clock cycle

Pipelined operation in Figure is said to have been stalled for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a hazard. We have just seen an example of a data hazard. A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls. The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called control hazards or instruction hazards. The effect of a cache miss on pipelined operation is illustrated in Figure. Instruction I1 is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I2, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I2 to arrive. We assume that instruction I2 is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.



(a) Instruction execution steps in successive clock cycles



→ Time

Clock cycle	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

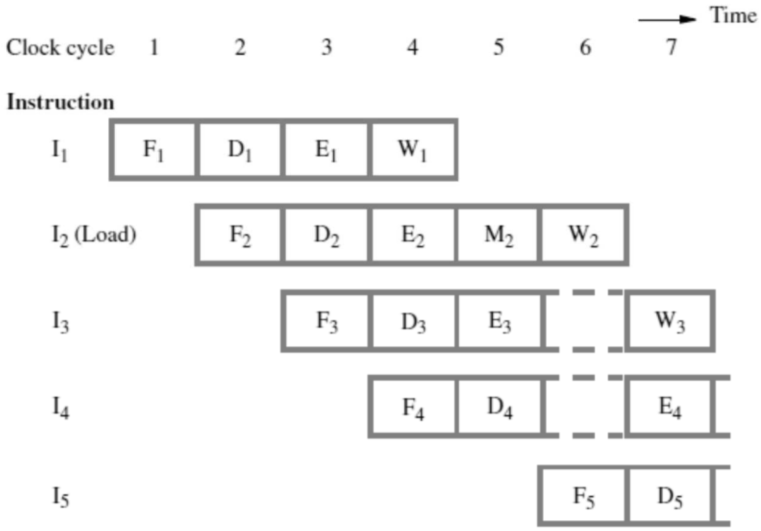
(b) Function performed by each processor stage in successive clock cycles

An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure b. This figure gives the function performed by each pipeline stage in each clock cycle. Note that the Decode unit is idle in cycles 3 through 5, the Execute unit is idle in cycles 4 through 6, and the Write unit is idle in cycles 5 through 7. Such idle periods are called stalls. They are also often referred to as bubbles in the pipeline. Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit.

A third type of hazard that may be encountered in pipelined operation is known as a structural hazard. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use separate instruction and data caches to avoid this delay. An example of a structural hazard is shown in Figure. This figure shows how the load instructions

Load X(R1),R2

can be accommodated in our example 4-stage pipeline. The memory address, X+[R1], is computed in stepE2 in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions I2 and I3 require access to the register file in cycle 6. Even though the instructions and their data are all available, the pipeline is stalled because one hardware resource, the register file, cannot handle two operations at once. If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled. In general, structural hazards are avoided by providing sufficient hardware resources on the processor chip.


Time
Clock cycle 1 2 3 4 5 6 7
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 (Load) F_2 D_2 E_2 M_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4
I_5 F_5 D_5

Effect of a Load instruction on pipeline timing

It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases, where throughput is measured by the rate at which instruction execution is completed. Any time one of the stages in the pipeline cannot complete its operation in one clock cycle, the pipeline stalls, and some degradation in performance occurs. Thus, the performance level of one instruction completion in each clock cycle is actually the upper limit for the throughput achievable in a pipelined processor organized as in Figure b. An important goal in designing processors is to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact. In the following sections we discuss various hazards, starting with data hazards, followed by control hazards. In each case we present some of the techniques used to mitigate their negative effect on performance. We discuss the issue of performance assessment in the following section in detail.

DATA HAZARDS

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure. We will now examine the issue of availability of data in some detail. Consider a program that contains two instructions, I1 followed by I2. When this program is executed in a pipeline, the execution of I2 can begin before the execution of I1 is completed. This means that the results generated by I1 may not be available for use by I2. We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that A=5, and consider the following two operations:

$$A \leftarrow 3 + A$$
$$B \leftarrow 4 \times A$$

When these operations are performed in the order given, the result is B = 32. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

$$A \leftarrow 5 \times C$$
$$B \leftarrow 20 + C$$

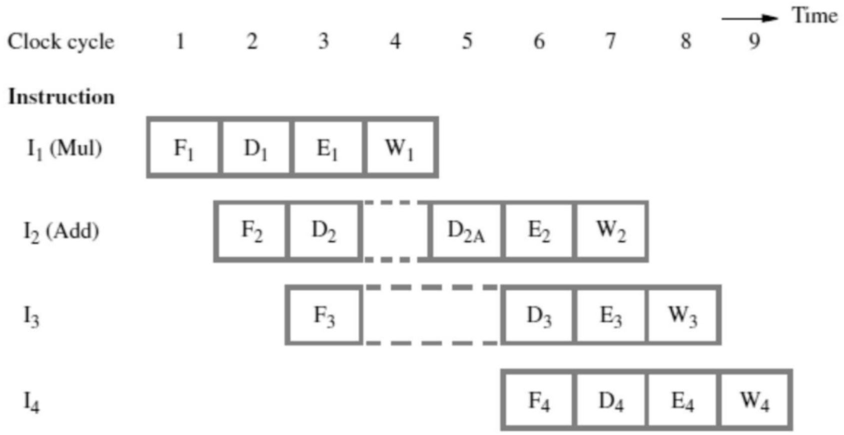
can be performed concurrently, because these operations are independent.

This example illustrates a basic constraint that must be enforced to guarantee correct results. When two operations depend on each other, they must be performed sequentially in the correct order. This rather obvious condition has far-reaching consequences. Understanding its implications is the key to understanding the variety of design alternatives and trade-offs encountered in pipelined computers. Consider the pipeline in Figure 2. The data dependency just described arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

Mul R2,R3,R4

Add R5,R4,R6

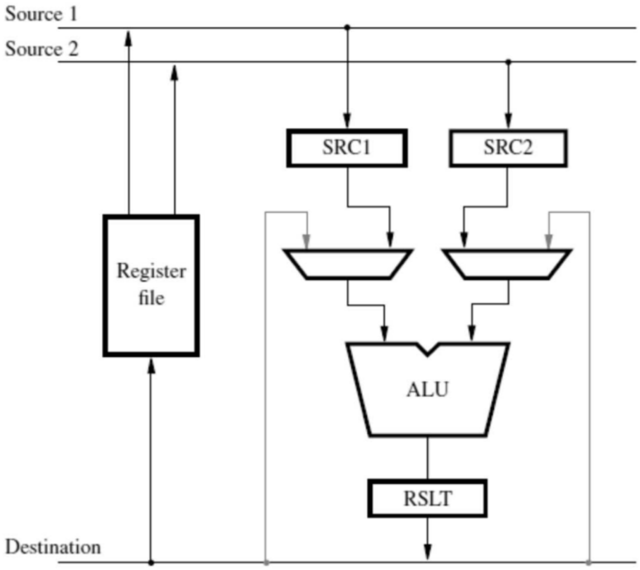
give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D2 must be delayed to clock cycle 5, and is shown as step D2A in the figure. Instruction I3 is fetched in cycle 3, but its decoding must be delayed because step D3 cannot precede D2. Hence, pipelined execution is stalled for two cycles.



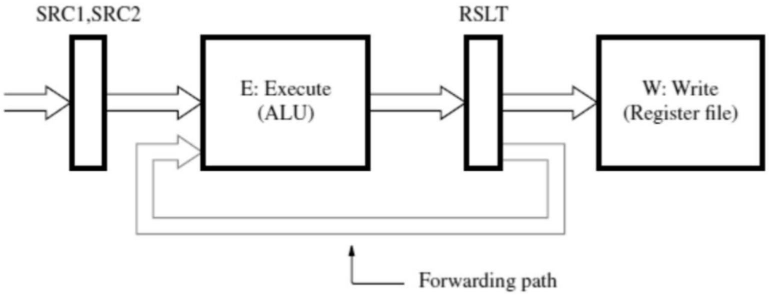
Operand Forwarding

The data hazard just described arises because one instruction, instruction I2 in Figure, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E1. Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I1 to be forwarded directly for use in step E2. Figure a shows a part of the processor datapath involving the ALU and the register file. This arrangement is similar to the three-bus structure in Figure, except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the interstage buffers needed for pipelined operation, as illustrated in Figure b. With reference to Figure b, registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3. The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

When the instructions in Figure are executed in the datapath of Figure, the operations performed in each clock cycle are as follows. After decoding instruction I2 and detecting the data dependency, a decision is made to use data forwarding. The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I1 is available in register RSLT, and because of the forwarding connection, it can be used in step E2. Hence, execution of I2 proceeds without interruption.



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Side Effects

The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I1 and as a source in I2. Sometimes instructions change the contents of a register other than the one named as the destination. An instruction that uses an autoincrement or autodecrement addressing mode is an example. In addition to storing new data in its destination location, the instruction changes the contents of a source register used to access one of its operands. All the precautions needed to handle data dependencies involving the destination location must also be applied to the registers affected by an autoincrement or autodecrement operation. When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect. For example, stack instructions, such as push and pop, produce similar side effects because they implicitly use the autoincrement and autodecrement addressing modes.

Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry. Suppose that registers R1 and R2 hold a double-precision integer number that we wish to add to another double precision number in registers R3 and R4. This may be accomplished as follows:

```
Add R1,R3
AddWithCarry R2,R4
```

An implicit dependency exists between these two instructions through the carry flag. This flag is set by the first instruction and used in the second instruction, which performs the operation

$$R4 \leftarrow [R2] + [R4] + \text{carry}$$

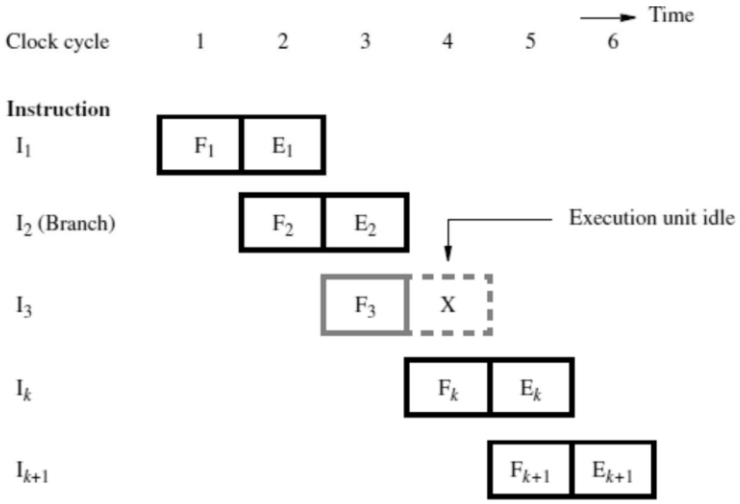
Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have few side effects. Ideally, only the contents of the destination location, either a register or a memory location, should be affected by any given instruction. Side effects, such as setting the condition code flags or updating the contents of an address pointer, should be kept to a minimum. However, it showed that the autoincrement and autodecrement addressing modes are potentially useful. Condition code flags are also needed for recording such information as the generation of a carry or the occurrence of overflow in an arithmetic operation. We show how such functions can be provided by other means that are consistent with a pipelined organization and with the requirements of optimizing compilers.

INSTRUCTION HAZARDS

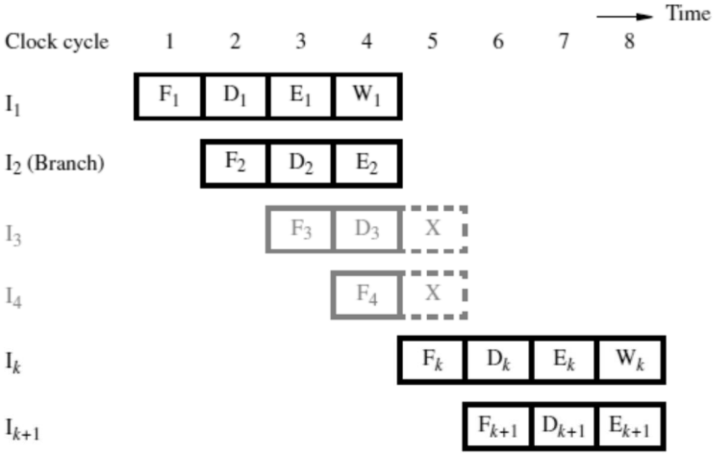
The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. Whenever this stream is interrupted, the pipeline stalls, as Figure illustrates for the case of a cache miss. A branch instruction may also cause the pipeline to stall. We will now examine the effect of branch instructions and the techniques that can be used for mitigating their impact.

Unconditional Branches

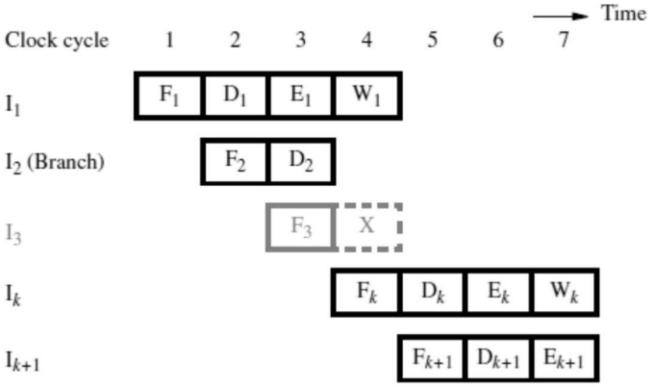
Figure shows a sequence of instructions being executed in a two-stage pipeline. Instructions I1 to I3 are stored at successive memory addresses, and I2 is a branch instruction. Let the branch target be instruction Ik . In clock cycle 3, the fetch operation for instruction I3 is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard I3, which has been incorrectly fetched, and fetch instruction Ik . In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.



The time lost as a result of a branch instruction is often referred to as the branch penalty. In Figure, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure a shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E2. Instructions I3 and I4 must be discarded, and the target instruction, Ik , is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles. Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched. With this additional hardware, both of these tasks can be performed in step D2, leading to the sequence of events shown in Figure b. In this case, the branch penalty is only one clock cycle.



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

Instruction Queue and Prefetching

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. Typically, the instruction queue can store several instructions. A separate unit, which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit. This leads to the organization shown in Figure. The dispatch unit also performs the decoding function.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

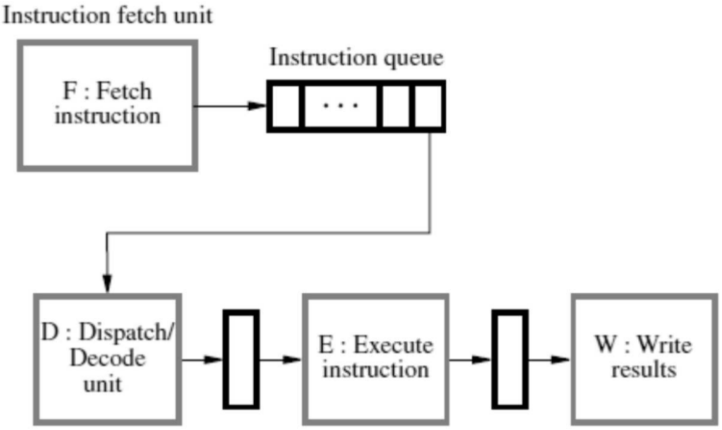
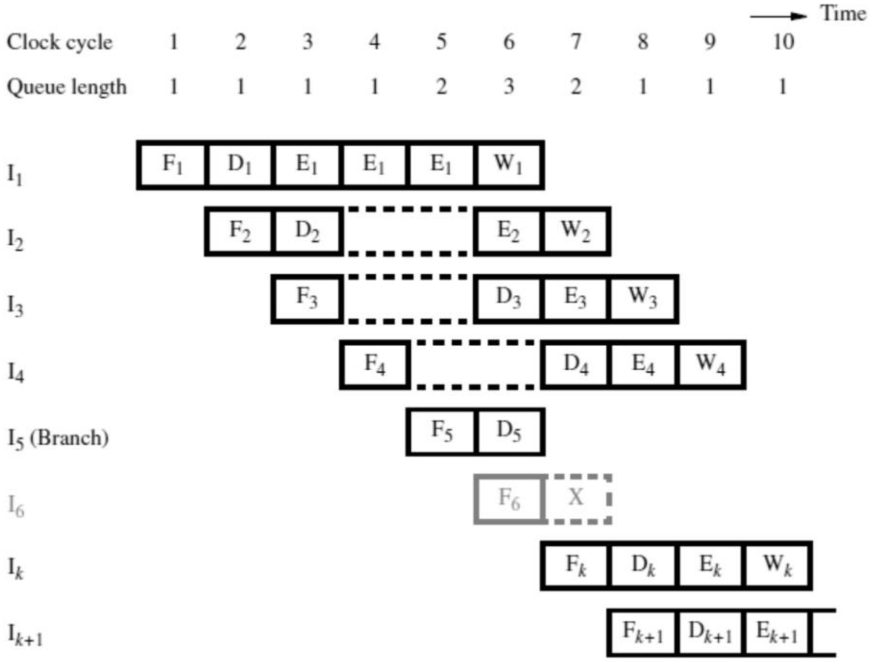


Figure illustrates how the queue length changes and how it affects the relationship between different pipeline stages. We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles. (There is both an F and a D step in each of these cycles.) Suppose that instruction I1 introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions, and the queue length rises to 3 in clock cycle 6. Instruction I5 is a branch instruction. Its target instruction, Ik , is fetched in cycle 7, and instruction I6 is discarded. The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I6. Instead, instruction I4 is dispatched from the queue to the decoding stage. After discarding I6, the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered. Now observe the sequence of instruction completions in Figure. Instructions I1, I2, I3, I4, and Ik complete execution in successive clock cycles. Hence, the branch instruction does not increase the overall execution time. This is because the instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as branch folding. Note that branch folding occurs only if at the time a branch instruction is encountered, at least one instruction is available in the queue other than the branch instruction. If only the branch instruction is in the queue, execution would proceed as in Figure b.

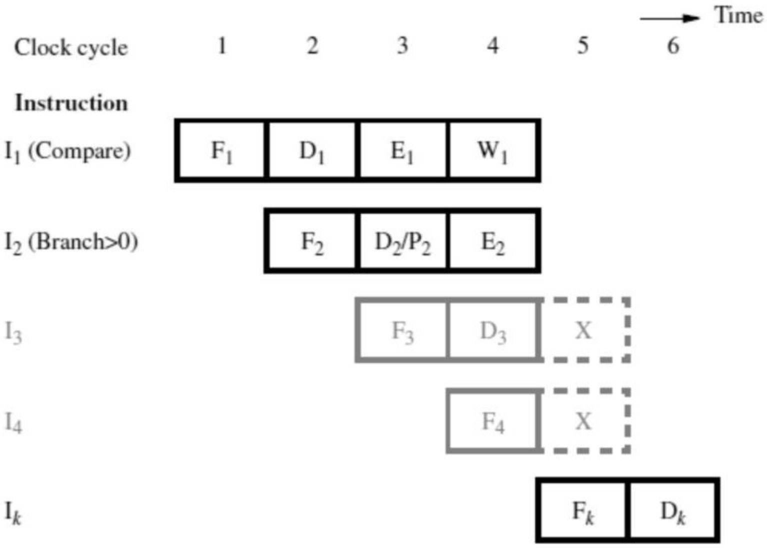
Therefore, it is desirable to arrange for the queue to be full most of the time, to ensure an adequate supply of instructions for processing. This can be achieved by increasing the rate at which the fetch unit reads instructions from the cache. In many processors, the width of the connection between the fetch unit and the instruction cache allows reading more than one instruction in each clock cycle. If the fetch unit replenishes the instruction queue quickly after a branch has occurred, the probability that branch folding will occur increases.


Time
Clock cycle 1 2 3 4 5 6 7 8 9 10
Queue length 1 1 1 1 2 3 2 1 1 1
I_1 F_1 D_1 E_1 E_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4
I_5 (Branch) F_5 D_5
I_6 F_6 X
I_k F_k D_k E_k W_k
I_{k+1} F_{k+1} D_{k+1} E_{k+1}

Having an instruction queue is also beneficial in dealing with cache misses. When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as the instruction queue is not empty. Meanwhile, the desired cache block is read from the main memory or from a secondary cache. When fetch operations are resumed, the instruction queue is refilled. If the queue does not become empty, a cache miss will have no effect on the rate of instruction execution. In summary, the instruction queue mitigates the impact of branch instructions on performance through the process of branch folding. It has a similar effect on stalls caused by cache misses. The effectiveness of this technique is enhanced when the instruction fetch unit is able to read more than one instruction at a time from the instruction cache.

Branch Prediction

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis. Speculative execution means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.



An incorrectly predicted branch is illustrated in Figure for a four-stage pipeline. The figure shows a Compare instruction followed by a Branch>0 instruction. Branch prediction takes place in cycle 3, while instruction I3 is being fetched. The fetch unit predicts that the branch will not be taken, and it continues to fetch instruction I4 as I3 enters the Decode stage. The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4. At this point, the instruction fetch unit realizes that the prediction was incorrect, and the two instructions in the execution pipe are purged. A new instruction, Ik , is fetched from the branch target address in clock cycle 5.

If branch outcomes were random, then half the branches would be taken. Then the simple approach of assuming that branches will not be taken would save the time lost to conditional branches 50 percent of the time. However, better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken, depending on the expected program behavior. For example, a branch instruction at the end of a loop causes a branch to the start of the loop for every pass through the loop except the last one. Hence, it is advantageous to assume that this branch will be taken and to have the instruction fetch unit start to fetch instructions at the branch target address. On the other hand, for a branch instruction at the beginning of a program loop, it is advantageous to assume that the branch will not be taken.

A decision on which way to predict the result of the branch may be made in hardware by observing whether the target address of the branch is lower than or higher than the address of the branch instruction. A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch instructions of some processors, such as SPARC, include a branch prediction bit, which is set to 0 or 1 by the compiler to indicate the desired behavior. The instruction fetch unit checks this bit to predict whether the branch will be taken or not taken.

With either of these schemes, the branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called static branch prediction. Another approach in which the prediction decision may change depending on execution history is called dynamic branch prediction. The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded. In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

INFLUENCE ON INSTRUCTION SETS

We have seen that some instructions are much better suited to pipelined execution than others. For example, instruction side effects can lead to undesirable data dependencies. In this section, we examine the relationship between pipelined execution and machine instruction features. We discuss two key aspects of machine instructions, addressing modes and condition code flags.

Addressing Modes

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, autoincrement, and autodecrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered. In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline.

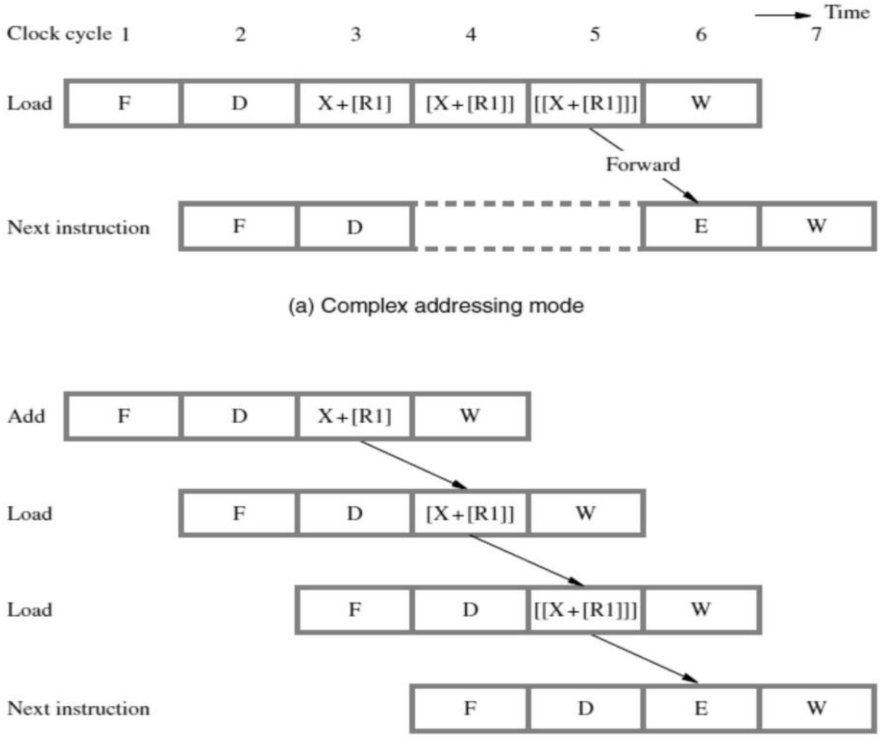
Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers. To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction Load X(R1),R2 takes five cycles to complete execution, as indicated in Figure. However, the instruction

Load (R1),R2

can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

Load (X(R1)),R2

may be executed as shown in Figure 8.16a, assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location X+[R1] in clock cycle 4 and then to read location [X+[R1]] in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.



(a) Complex addressing mode

(b) Simple addressing mode

To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

Add #X,R1,R2
Load (R2),R2
Load (R2),R2

The Add instruction performs the operation R2←X+[R1]. The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure b. This example indicates that, in a pipelined processor, complex addressing modes that involve several accesses to the memory do not necessarily lead to faster execution. The main advantage of such modes is that they reduce the number of instructions needed to perform a given task and thereby reduce the program space needed in the main memory. Their main disadvantage is that their long execution times cause the pipeline to stall, thus reducing its effectiveness. They require more complex hardware to decode and execute them. Also, they are not convenient for compilers to work with.

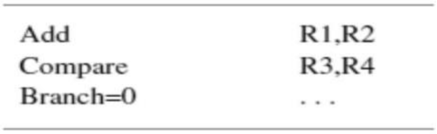
The instruction sets of modern processors are designed to take maximum advantage of pipelined hardware. Because complex addressing modes are not suitable for pipelined execution, they should be avoided. The addressing modes used in modern processors often have the following features:

- Access to an operand does not require more than one access to the memory.
- Only load and store instructions access memory operands.
- The addressing modes used do not have side effects.

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register. Memory is accessed in the following cycle. None of these modes has any side effects, with one possible exception. Some architectures, such as ARM, allow the address computed in the index mode to be written back into the index register. This is a side effect that would not be allowed under the guidelines above. Note also that relative addressing can be used; this is a special case of indexed addressing in which the program counter is used as the index register. The three features just listed were first emphasized as part of the concept of RISC processors. The SPARC processor architecture, which adheres to these guidelines.

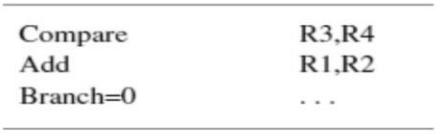
Condition Codes

The condition code flags either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.



```
Add          R1,R2
Compare      R3,R4
Branch=0     . . .
```

(a) A program fragment



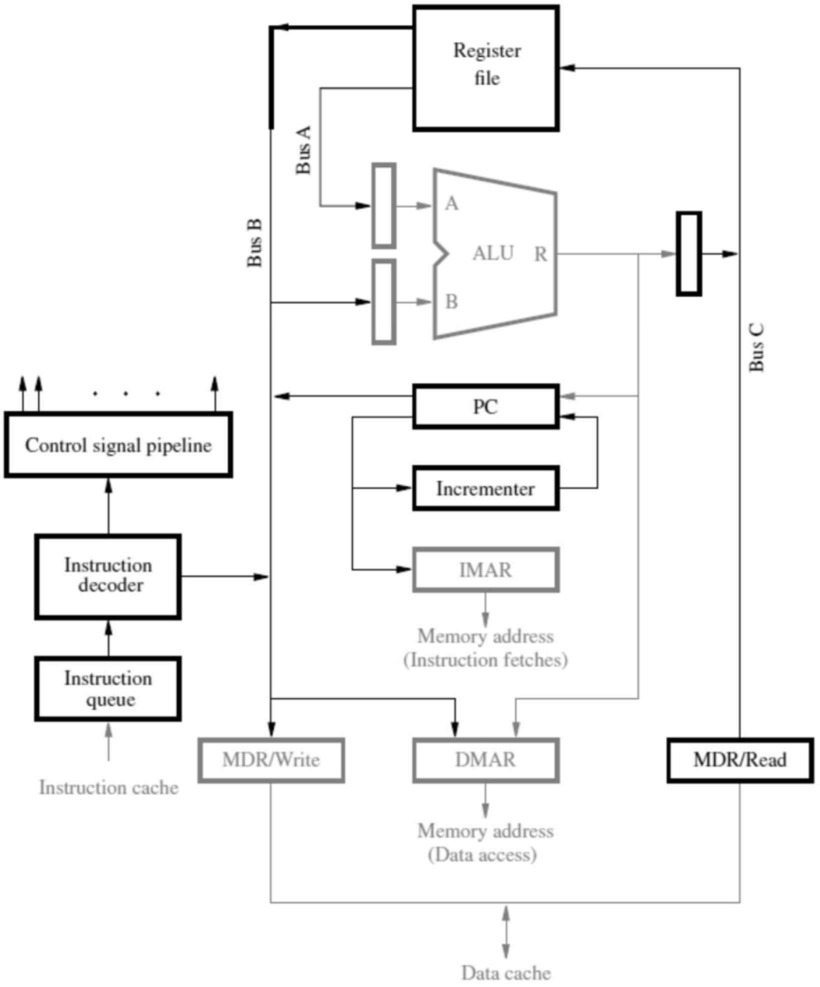
```
Compare      R3,R4
Add          R1,R2
Branch=0     . . .
```

(b) Instructions reordered

Consider the sequence of instructions in Figure a, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure. The execution time of the Branch instruction can be reduced by interchanging the Add and Compare instructions, as shown in Figure b. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes. These observations lead to two important conclusions about the way condition codes should be handled. First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility. Figure b shows the instructions reordered assuming that the condition code flags are affected only when this is explicitly stated as part of the instruction OP code. The SPARC and ARM architectures provide this flexibility.

DATA PATH AND CONTROL CONSIDERATIONS

Consider the three-bus structure from organization of the internal datapath of a processor suitable for pipelined execution, it can be modified as shown in Figure to support a 4-stage pipeline. The resources involved in stages F and E are shown in blue and those used in stages D and W in black. Operations in the data cache may happen during stage E or at a later stage, depending on the addressing mode and the implementation details.



Several important changes should be noted:

1. There are separate instructions and data caches that use separate address and data connections to the processor. This requires two versions of the MAR register, IMAR for accessing the instruction cache and DMAR for accessing the data cache.

2. The PC is connected directly to the IMAR, so that the contents of the PC can be transferred to IMAR at the same time that an independent ALU operation is taking place.

3. The data address in DMAR can be obtained directly from the register file or from the ALU to support the register indirect and indexed addressing modes.

4. Separate MDR registers are provided for read and write operations. Data can be transferred directly between these registers and the register file during load and store operations without the need to pass through the ALU.

5. Buffer registers have been introduced at the inputs and output of the ALU. These are registers SRC1, SRC2, and RSLT in Figure. Forwarding connections are not included in Figure. They may be added if desired.

6. The instruction register has been replaced with an instruction queue, which is loaded from the instruction cache.

7. The output of the instruction decoder is connected to the control signal pipeline. The need for buffering control signals and passing them from one stage to the next along with the instruction is discussed in Section 8.1. This pipeline holds the control signals in buffers B2 and B3.

The following operations can be performed independently in the processor of Figure:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

Because these operations do not use any shared resources, they can be performed simultaneously in any combination. The structure provides the flexibility required to implement the four-stage pipeline. For example, let I1, I2, I3, and I4 be a sequence of four instructions. As shown in Figure 8.2a, the following actions all happen during clock cycle 4:

- Write the result of instruction I1 into the register file
- Read the operands of instruction I2 from the register file
- Decode instruction I3
- Fetch instruction I4 and increment the PC.

SUPERSCALAR OPERATION

Pipelining makes it possible to execute instructions concurrently. Several instructions are present in the pipeline at the same time, but they are in different stages of their execution. While one instruction is performing an ALU operation, another instruction is being decoded and yet another is being fetched from the memory. Instructions enter the pipeline in strict program order. In the absence of hazards, one instruction enters the pipeline and one instruction completes execution in each clock cycle. This means that the maximum throughput of a pipelined processor is one instruction per clock cycle.

A more aggressive approach is to equip the processor with multiple processing units to handle several instructions in parallel in each processing stage. With this arrangement, several instructions start execution in the same clock cycle, and the processor is said to use multiple-issue. Such processors are capable of achieving an instruction execution throughput of more than one instruction per cycle. They are known as superscalar processors. Many modern high-performance processors use this approach. We introduced the idea of an instruction queue. We pointed out that to keep the instruction queue filled, a processor should be able to fetch more than one instruction at a time from the cache. For superscalar operation, this arrangement is essential. Multiple-issue operation requires a wider path to the cache and multiple execution units. Separate execution units are provided for integer and floating-point instructions.

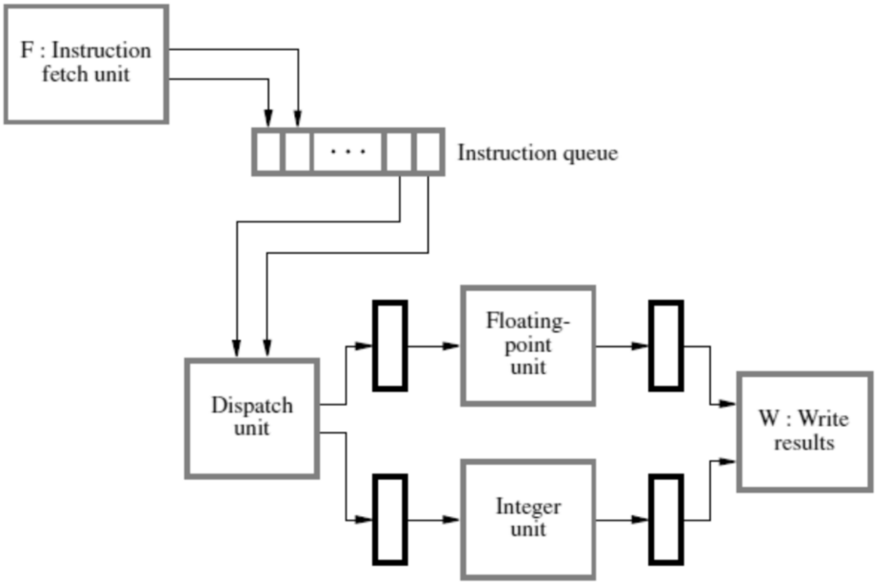
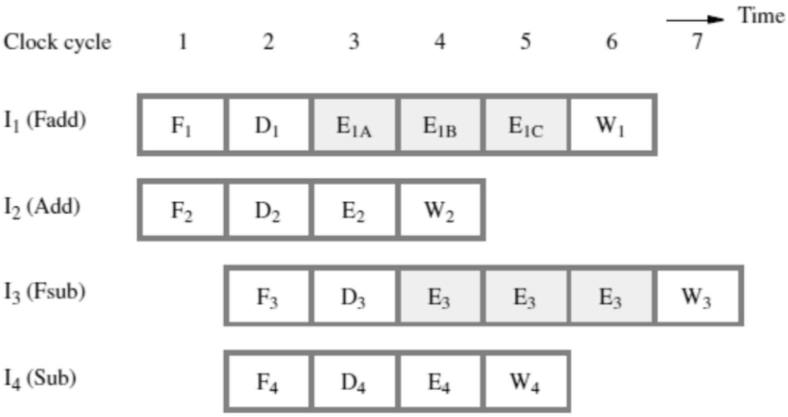


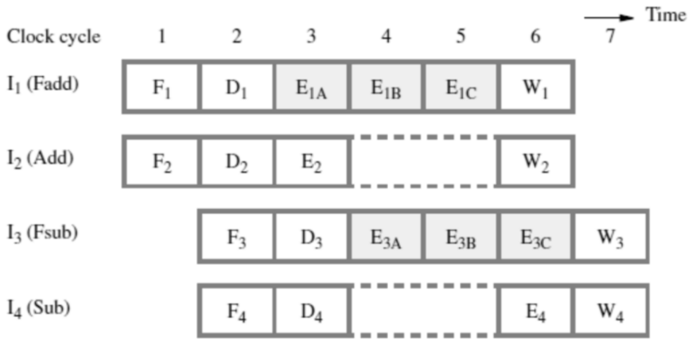
Figure shows an example of a processor with two execution units, one for integer and one for floating-point operations. The Instruction fetch unit is capable of reading two instructions at a time and storing them in the instruction queue. In each clock cycle, the Dispatch unit retrieves and decodes up to two instructions from the front of the queue. If there is one integer, one floating-point instruction, and no hazards, both instructions are dispatched in the same clock cycle.



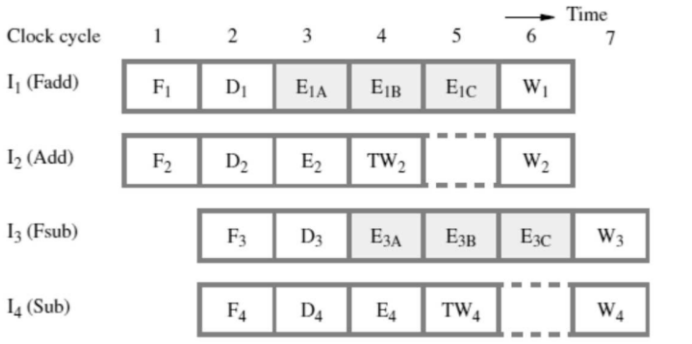
In a superscalar processor, the detrimental effect on performance of various hazards becomes even more pronounced. The compiler can avoid many hazards through judicious selection and ordering of instructions. For example, for the processor in Figure, the compiler should strive to interleave floating-point and integer instructions. This would enable the dispatch unit to keep both the integer and floating-point units busy most of the time. In general, high performance is achieved if the compiler is able to arrange program instructions to take maximum advantage of the available hardware units. Pipeline timing is shown in Figure. The blue shading indicates operations in the floating-point unit. The floating-point unit takes three clock cycles to complete the floating-point operation specified in I1. The integer unit completes execution of I2 in one clock cycle. We have also assumed that the floating-point unit is organized internally as a three-stage pipeline. Thus, it can still accept a new instruction in each clock cycle. Hence, instructions I3 and I4 enter the dispatch unit in cycle 3, and both are dispatched in cycle 4. The integer unit can receive a new instruction because instruction I2 has proceeded to the Write stage. Instruction I1 is still in the execution phase, but it has moved to the second stage of the internal pipeline in the floating-point unit. Therefore, instruction I3 can enter the first stage. Assuming that no hazards are encountered, the instructions complete execution as shown.

Out-of-Order Execution

In Figure, instructions are dispatched in the same order as they appear in the program. However, their execution is completed out of order. Does this lead to any problems? We have already discussed the issues arising from dependencies among instructions. For example, if instruction I2 depends on the result of I1, the execution of I2 will be delayed. As long as such dependencies are handled correctly, there is no reason to delay the execution of an instruction. However, a new complication arises when we consider the possibility of an instruction causing an exception. **Exceptions** may be caused by a bus error during an operand fetch or by an illegal operation, such as an attempt to divide by zero. The results of I2 are written back into the register file in cycle 4. If instruction I1 causes an exception, program execution is in an inconsistent state. The program counter points to the instruction in which the exception occurred. However, one or more of the succeeding instructions have been executed to completion. If such a situation is permitted, the processor is said to have **imprecise exceptions**.



(a) Delayed write



(b) Using temporary registers

To guarantee a consistent state when exceptions occur, the results of the execution of instructions must be written into the destination locations strictly in program order. This means we must delay stepW2 in Figure 8.20 until cycle 6. In turn, the integer execution unit must retain the result of instruction I2, and hence it cannot accept instruction I4 until cycle 6, as shown in Figure 8.21a. If an exception occurs during an instruction, all subsequent instructions that may have been partially executed are discarded. This is called a **precise exception**. It is easier to provide precise exceptions in the case of external interrupts. When an external interrupt is received, the Dispatch unit stops reading new instructions from the instruction queue, and the instructions remaining in the queue are discarded. All instructions whose

Execution is pending continue to completion. At this point, the processor and all its registers are in a consistent state, and interrupting processing can begin.

Execution Completion

It is desirable to use out-of-order execution, so that an execution unit is freed to execute other instructions as soon as possible. At the same time, instructions must be completed in program order to allow precise exceptions. These seemingly conflicting requirements are readily resolved if execution is allowed to proceed as shown in Figure, but the results are written into temporary registers. The contents of these registers are later transferred to the permanent registers in correct program order. This approach is illustrated in Figure b. Step TW is a write into a temporary register. Step W is the final step in which the contents of the temporary register are transferred into the appropriate permanent register. This step is often called the commitment step because the effect of the instruction cannot be reversed after that point. If an instruction causes an exception, the results of any subsequent instruction that has been executed would still be in temporary registers and can be safely discarded.

A temporary register assumes the role of the permanent register whose data it is holding and is given the same name. For example, if the destination register of I2 is R5, the temporary register used in step TW2 is treated as R5 during clock cycles 6 and 7. Its contents would be forwarded to any subsequent instruction that refers to R5 during that period. Because of this feature, this technique is called register renaming. Note that the temporary register is used only for instructions that follow I2 in program order. If an instruction that precedes I2 needs to read R5 in cycle 6 or 7, it would access the actual register R5, which still contains data that have not been modified by instruction I2. When out-of-order execution is allowed, a special control unit is needed to guarantee in-order commitment. This is called the commitment unit. It uses a queue called the reorder buffer to determine which instruction(s) should be committed next. Instructions are entered in the queue strictly in program order as they are dispatched for execution. When an instruction reaches the head of that queue and the execution of that instruction has been completed, the corresponding results are transferred from the temporary registers to the permanent registers and the instruction is removed from the queue. All resources that were assigned to the instruction, including the temporary registers, are released. The instruction is said to have been retired at this point. Because an instruction is retired only when it is at the head of the queue, all instructions that were dispatched before it must also have been retired. Hence, instructions may complete execution out of order, but they are retired in program order.

Dispatch Operation

We now return to the dispatch operation. When dispatching decisions are made, the dispatch unit must ensure that all the resources needed for the execution of an instruction are available. For example, since the results of an instruction may have to be written in a temporary register, the required register must be free, and it is reserved for use by that instruction as a part of the dispatch operation. A location in the reorder buffer must also be available for the instruction. When all the resources needed are assigned, including an appropriate execution unit, the instruction is dispatched. Should instructions be dispatched out of order? For example, if instruction I2 in Figure b is delayed because of a cache miss for a source operand, the integer unit will be busy in cycle 4, and I4 cannot be dispatched. Should I5 be dispatched instead? In principle this is possible, provided that a place is reserved in the reorder buffer for instruction I4 to ensure that all instructions are retired in the correct order. Dispatching instructions out of order requires considerable care. If I5 is dispatched while I4 is still waiting for some resource, we must ensure that there is no possibility of a deadlock occurring. A deadlock is a situation that can arise when two units, A and B, use a shared resource. Suppose that unit B cannot complete its task until unit A completes its task. At the same time, unit B has been assigned a resource that unit A needs. If this happens, neither unit can complete its task. Unit A is waiting for the resource it needs, which is being held by unit B. At the same time, unit B is waiting for unit A to finish before it can release that resource.

If instructions are dispatched out of order, a deadlock can arise as follows. Suppose that the processor has only one temporary register, and that when I5 is dispatched, that register is reserved for it. Instruction I4 cannot be dispatched because it is waiting for the temporary register, which, in turn, will not become free until instruction I5 is retired. Since instruction I5 cannot be retired before I4, we have a deadlock. To prevent deadlocks, the dispatcher must take many factors into account. Hence, issuing instructions out of order is likely to increase the complexity of the Dispatch unit significantly. It may also mean that more time is required to make dispatching decisions. For these reasons, most processors use only in-order dispatching. Thus, the program order of instructions is enforced at the time instructions are dispatched and again at the time instructions are retired. Between these two events, the execution of several instructions can proceed at their own speed, subject only to any interdependencies that may exist among instructions.

PERFORMANCE CONSIDERATIONS

We pointed out in Section 1.6 that the execution time, T , of a program that has a dynamic instruction count N is given by

$$T = (N \times S) / R$$

where S is the average number of clock cycles it takes to fetch and execute one instruction, and R is the clock rate. This simple model assumes that instructions are executed one after the other, with no overlap. A useful performance indicator is the instruction throughput, which is the number of instructions executed per second. For sequential execution, the throughput, Ps is given by

$$Ps = R/S$$

We examine the extent to which pipelining increases instruction throughput. The only real measure of performance is the total execution time of a program. Higher instruction throughput will not necessarily lead to higher performance if a larger number of instructions is needed to implement the desired task. For this reason, the SPEC ratings described in Chapter 1 provide a much better indicator when comparing two processors.

The four-stage pipeline may increase instruction throughput by a factor of four. In general, an n-stage pipeline has the potential to increase throughput n times. Thus, it would appear that the higher the value of n, the larger the performance gain. This leads to two questions:

- How much of this potential increase in instruction throughput can be realized in practice?
- What is a good value for n?

Any time a pipeline is stalled, the instruction throughput is reduced. Hence, the performance of a pipeline is highly influenced by factors such as branch and cache miss penalties. First, we discuss the effect of these factors on performance, and then we return to the question of how many pipeline stages should be used.

Effect of Instruction Hazards

The effects of various hazards have been examined qualitatively in the previous sections. We now assess the impact of cache misses and branch penalties in quantitative terms. Consider a processor that uses the four-stage pipeline. The clock rate, hence the time allocated to each step in the pipeline, is determined by the longest step. Let the delay through the ALU be the critical parameter. This is the time needed to add two integers. Thus, if the ALU delay is 2 ns, a clock of 500 MHz can be used. The on-chip instruction and data caches for this processor should also be designed to have an access time of 2 ns. Under ideal conditions, this pipelined processor will have an instruction throughput, Pp, given by

$$Pp = R = 500\ MIPS\ (million\ instructions\ per\ second)$$

To evaluate the effect of cache misses, we use the same parameters. The cache miss penalty, Mp, in that system is computed to be 17 clock cycles. Let TI be the time between two successive instruction completions. For sequential execution, TI = S. However, in the absence of hazards, a pipelined processor completes the execution of one instruction each clock cycle, thus, TI = 1 cycle. A cache miss stalls the pipeline by an amount equal to the cache miss penalty. This means that the value of TI increases by an amount equal to the cache miss penalty for the instruction in which the miss occurs. A cache miss can occur for either instructions or data. Consider a computer that has a shared cache for both instructions and data, and let d be the percentage of instructions that refer to data operands in the memory. The average increase in the value of TI as a result of cache misses is given by

$$\delta miss = ((1 - hi\ ) + d(1 - hd\ )) \times Mp$$

where hi and hd are the hit ratios for instructions and data, respectively. Assume that 30 percent of the instructions access data in memory. With a 95-percent instruction hit rate and a 90- percent data hit rate, δmiss is given by

$$\delta miss = (0.05 + 0.3 \times 0.1) \times 17 = 1.36\ cycles$$

Taking this delay into account, the processor's throughput would be

$$Pp = R/TI = R/(1 + \delta miss) = 0.42R$$

Note that with R expressed in MHz, the throughput is obtained directly in millions of instructions per second. For R = 500 MHz, Pp = 210 MIPS. Let us compare this value to the throughput obtainable without pipelining. A processor that uses sequential execution requires four cycles per instruction. Its throughput would be

$$Ps = R\ /\ (4 + \delta miss) = 0.19R$$

For R = 500 MHz, Ps = 95 MIPS. Clearly, pipelining leads to significantly higher throughput. But the performance gain of 0.42/0.19 = 2.2 is only slightly better than one-half the ideal case. Reducing the cache miss penalty is particularly worthwhile in a pipelined processor. This can be achieved by introducing a secondary cache between the primary, on-chip cache and the memory. Assume that the time needed to transfer an 8-word block from the secondary cache is 10 ns. Hence, a miss in the primary cache for which the required block is found in the secondary cache introduces a penalty, Ms, of 5 cycles. In the case of a miss in the secondary cache, the full 17-cycle penalty (Mp) is still incurred. Hence, assuming a hit rate hs of 94 percent in the secondary cache, the average increase in TI is

$$\delta miss = ((1 - hi\ ) + d(1 - hd\ )) \times (hs \times Ms + (1 - hs) \times Mp) = 0.46\ cycle$$

The instruction throughput in this case is 0.68R, or 340 MIPS. An equivalent nonpipelined processor would have a throughput of 0.22R, or 110 MIPS. Thus, pipelining provides a performance gain of 0.68/0.22 = 3.1. The values of 1.36 and 0.46 are, in fact, somewhat pessimistic, because we have assumed that every time a data miss occurs, the entire miss penalty is incurred. This is the case only if the instruction immediately following the instruction that references memory is delayed while the processor waits for the memory access to be completed. However, an optimizing compiler attempts to increase the distance between two instructions that create a dependency by placing other instructions between them whenever possible. Also, in a processor that uses an instruction queue, the cache miss penalty during instruction fetches may have a much reduced effect as the processor is able to dispatch instructions from the queue.

Number of Pipeline Stages

The fact that an n-stage pipeline may increase instruction throughput by a factor of *n* suggests that we should use a large number of stages. However, as the number of pipeline stages increases, so does the probability of the pipeline being stalled, because more instructions are being executed concurrently. Thus, dependencies between instructions that are far apart may still cause the pipeline to stall. Also, branch penalties may become more significant. For these reasons, the gain from increasing the value of n begins to diminish, and the associated cost is not justified.

Another important factor is the inherent delay in the basic operations performed by the processor. The most important among these is the ALU delay. In many processors, the cycle time of the processor clock is chosen such that one ALU operation can be completed in one cycle. Other operations are divided into steps that take about the same time as an add operation. It is also possible to use a pipelined ALU. For example, the ALU of the Compaq Alpha 21064 processor consists of a two-stage pipeline, in which each stage completes its operation in 5 ns. Many pipelined processors use four to six stages. Others divide instruction execution into smaller steps and use more pipeline stages and a faster clock. For example, the UltraSPARC II uses a 9-stage pipeline and Intel's Pentium Pro uses a 12-stage pipeline. The latest Intel processor, Pentium 4, has a 20-stage pipeline and uses a clock speed in the range 1.3 to 1.5 GHz. For fast operations, there are two pipeline stages in one clock cycle.

HANDLING EXCEPTIONS

The general actions needed to switch from User mode to the appropriate exception mode. The actions vary in detail, depending upon the exception and the exception mode entered. Here, we consider some of those details.

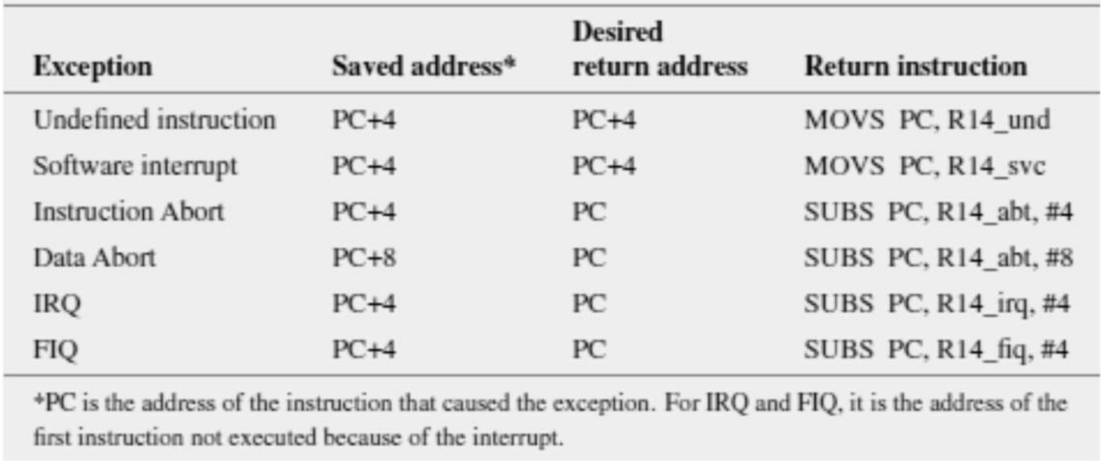
Pipelined Execution, the Program Counter, and the Status Register

The ARM processor overlaps the fetching and execution of successive instructions in order to increase instruction throughput. This technique is called pipelined instruction execution. During pipelined execution of instructions, updating of the program counter is done as follows. Suppose that the processor fetches instruction I1 from address A. The contents of PC are incremented toA+4, then execution of I1 is begun. Before the execution of I1 is completed, the processor fetches instruction I2 from address A+4, then increments PC to A+8.

Now assume that at the end of execution of I1 the processor detects that an ordinary interrupt request (IRQ) has been received. The processor performs the actions to enter the IRQ exception mode to service the interrupt. It copies the contents of CPSR into SPSR_irq and copies the contents of PC, which are now A+8, into the link register R14_irq. Instruction I2, which has been fetched but not yet fully executed, is discarded. This is the instruction to which the interrupt-service routine must return. The interrupt-service routine must subtract 4 from R14_irq before using its contents as the return address. The saved copy of the Status register must also be restored. The required actions are carried out by the single instruction

SUBS PC, R14_irq, #4

which subtracts 4 from R14_irq and stores the result into PC. The suffix S in the OP code normally means “set condition codes.” But when the target register of the instruction is PC, the S suffix causes the processor to copy the contents of SPSR_irq into CPSR, thus completing the actions needed to return to the interrupted program.



Exception	Saved address*	Desired return address	Return instruction
Undefined instruction	PC+4	PC+4	MOVS PC, R14_und
Software interrupt	PC+4	PC+4	MOVS PC, R14_svc
Instruction Abort	PC+4	PC	SUBS PC, R14_abt, #4
Data Abort	PC+8	PC	SUBS PC, R14_abt, #8
IRQ	PC+4	PC	SUBS PC, R14_irq, #4
FIQ	PC+4	PC	SUBS PC, R14_fiq, #4

*PC is the address of the instruction that caused the exception. For IRQ and FIQ, it is the address of the first instruction not executed because of the interrupt.

In the case of a software interrupt triggered by execution of the SWI instruction, the value saved in R14_svc is the correct return address. Return from a software interrupt can be accomplished using the instruction

MOVS PC, R14_svc

that also copies the contents of SPSR_svc into CPSR. Table gives the correct return-address value and the instruction that can be used to return to the interrupted program for each of the exceptions in Table, except for powerup/reset, which abandons any currently executing program. Note that for a data access or instruction access violation, the return address is the address of the instruction that caused the exception, because it must be re-executed after the cause of the violation has been resolved.

Manipulating Status Register Bits

When the processor is running in a privileged mode, special Move instructions, MRS and MSR, can be used to transfer the contents of the current or saved processor status registers to or from a general-purpose register. For example,

MRS Rd, CPSR

copies the contents of CPSR into register Rd. Similarly,

MSR SPSR, Rm

copies the contents of register Rm into SPSR_mode. After status register contents have been loaded into a register, logic instructions can be used to manipulate individual bits. Then, the register contents can be copied back into the status register to effect the desired changes. For example, these steps can be used to set or clear interrupt-disable bits in an exception-service routine.

Nesting Exception-Service Routines

Recall that nesting of subroutines is facilitated by storing the contents of the link register in the stack frame associated with a subroutine that calls another subroutine. This action is not needed when an exception-service routine is interrupted by a higher-priority exception whose service routine runs in a different processor mode. This is because each mode has its own banked link register.

For example, suppose that an ordinary interrupt is being serviced by an IRQ-mode routine when an interrupt that requires fast servicing is received. The first routine is interrupted and the FIQ mode is entered to service the second interrupt. The return address for the program that was interrupted to service the IRQ interrupt remains unchanged in link register R14_irq.

The return address for the IRQ routine is stored in R14_fiq. Hence, the use of banked registers avoids overwriting saved return addresses, and these addresses do not need to be placed on the stack when nesting of exception routines occurs. However, if different exceptions are serviced in the same processor mode, then their return addresses will need to be saved if nesting is allowed.

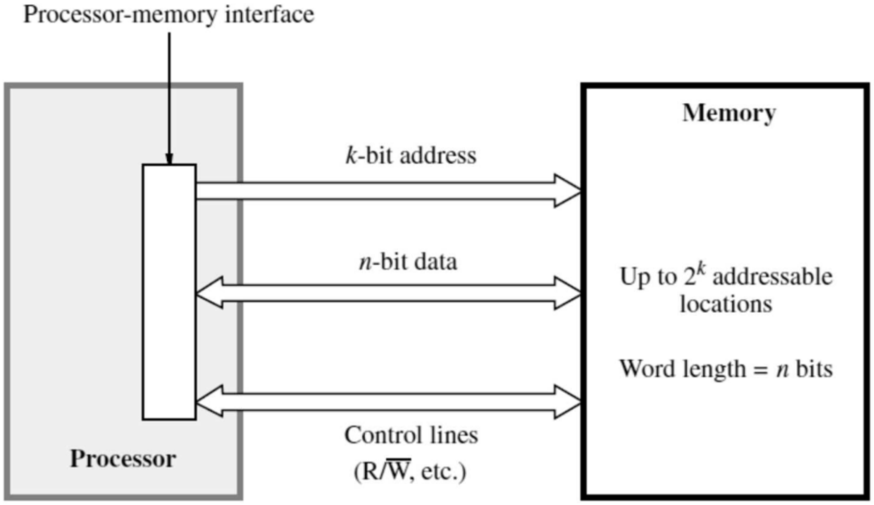
Introduction

Programs and the data they operate on are held in memory of the computer. In this chapter, we discuss how this vital part of the computer operates. By now, the reader appreciates that the execution speed of programs is highly dependent on the speed with which instructions and data can be transferred between the processor and the memory. It is also important to have sufficient memory to facilitate execution of large programs having large amounts of data.

Ideally, the memory would be fast, large, and inexpensive. Unfortunately, it is impossible to meet all three of these requirements simultaneously. Increased speed and size are achieved at increased cost.

BASIC CONCEPTS

The maximum size of the memory that can be used in any computer is determined by the addressing scheme. For example, a computer that generates 16-bit addresses is capable of addressing up to 216 = 64K (kilo) memory locations. Machines whose instructions generate 32-bit addresses can utilize a memory that contains up to 232 = 4G (giga) locations, whereas machines with 64-bit addresses can access up to 264 = 16E (exa) ≈ 16 × 1018 locations. The number of locations represents the size of the address space of the computer. The memory is usually designed to store and retrieve data in word-length quantities. Consider, for example, a byte-addressable computer whose instructions generate 32-bit addresses. When a 32-bit address is sent from the processor to the memory unit, the high order 30 bits determine which word will be accessed. If a byte quantity is specified, the low-order 2 bits of the address specify which byte location is involved.



The connection between the processor and its memory consists of address, data, and control lines, as shown in Figure. The processor uses the address lines to specify the memory location involved in a data transfer operation, and uses the data lines to transfer the data. At the same time, the control lines carry the command indicating a Read or a Write operation and whether a byte or a word is to be transferred. The control lines also provide the necessary timing information and are used by the memory to indicate when it has completed the requested operation. When the processor-memory interface receives the memory's response, it asserts the MFC signal shown in Figure. This is the processor's internal control signal that indicates that the requested memory operation has been completed. When asserted, the processor proceeds to the next step in its execution sequence.

A useful measure of the speed of memory units is the time that elapses between the initiation of an operation to transfer a word of data and the completion of that operation. This is referred to as the memory access time. Another important measure is the memory cycle time, which is the minimum time delay required between the initiation of two successive memory operations, for example, the time between two successive Read operations. The cycle time is usually slightly longer than the access time, depending on the implementation details of the memory unit. A memory unit is called a random-access memory (RAM) if the access time to any location is the same, independent of the location's address. This distinguishes such memory units from serial, or partly serial, access storage devices such as magnetic and optical disks. Access time of the latter devices depends on the address or position of the data. The technology for implementing computer memories uses semiconductor integrated circuits. The sections that follow present some basic facts about the internal structure and operation of such memories. We then discuss some of the techniques used to increase the effective speed and size of the memory.

Cache and Virtual Memory

The computer processor can usually process instructions and data faster than they can be fetched from the main memory. Hence, the memory access time is the bottleneck in the system. One way to reduce the memory access time is to use a cache memory. This is a small, fast memory inserted between the larger, slower main memory and the processor. It holds the currently active portions of a program and their data.

Virtual memory is another important concept related to memory organization. With this technique, only the active portions of a program are stored in the main memory, and the remainder is stored on the much larger secondary storage device. Sections of the program are transferred back and forth between the main memory and the secondary storage device in a manner that is transparent to the application program. As a result, the application program sees a memory that is much larger than the computer's physical main memory.

Block Transfers

The discussion above shows that data move frequently between the main memory and the cache and between the main memory and the disk. These transfers do not occur one word at a time. Data are always transferred in contiguous blocks involving tens, hundreds, or thousands of words. Data transfers between the main memory and high-speed devices such as a graphic display or an Ethernet interface also involve large blocks of data. Hence, a critical parameter for the performance of the main memory is its ability to read or write blocks of data at high speed. This is an important consideration that we will encounter repeatedly as we discuss memory technology and the organization of the memory system.

SEMICONDUCTOR RAM MEMORIES

Semiconductor random-access memories (RAMs) are available in a wide range of speeds. Their cycle times range from 100 ns to less than 10 ns. In this section, we discuss the main characteristics of these memories. We start by introducing the way that memory cells are organized inside a chip.

Internal Organization of Memory Chips

Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. A possible organization is illustrated in Figure. Each row of cells constitutes a memory word, and all cells of a row are connected to a common line referred to as the word line, which is driven by the address decoder on the chip. The cells in each column are connected to a Sense/Write circuit by two bit lines, and the Sense/Write circuits are connected to the data input/output lines of the chip. During a Read operation, these circuits sense, or read, the information stored in the cells selected by a word line and place this information on the output data lines. During a Write operation, the Sense/Write circuits receive input data and store them in the cells of the selected word.

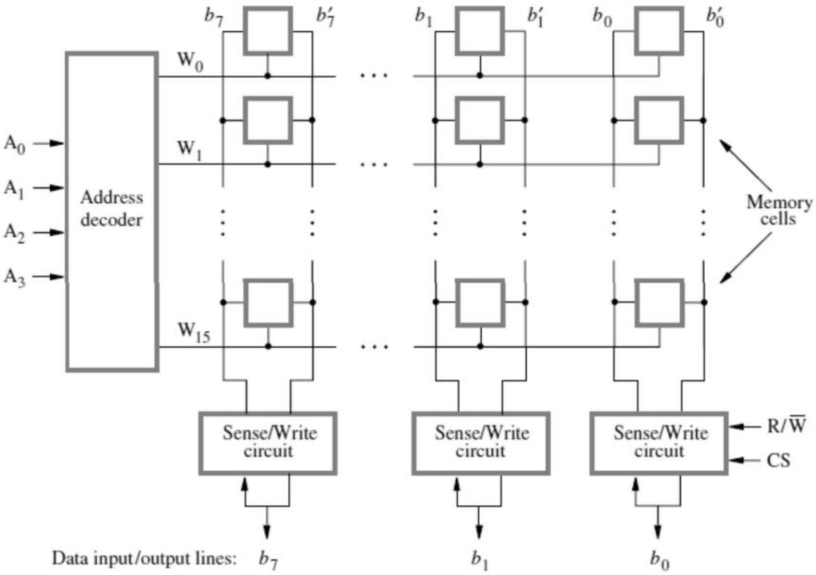
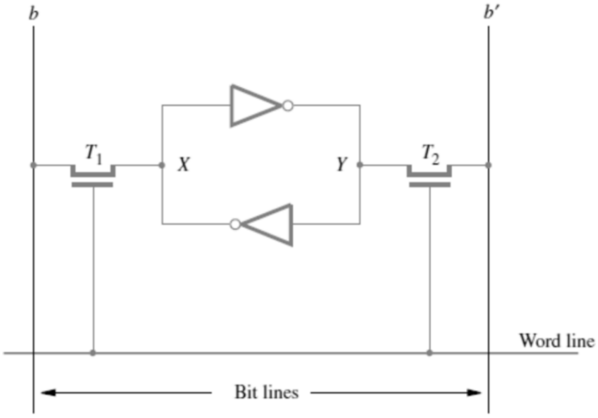


Figure is an example of a very small memory circuit consisting of 16 words of 8 bits each. This is referred to as a 16 × 8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that can be connected to the data lines of a computer. Two control lines, R/W and CS, are provided. The R/W (Read/Write) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

The memory circuit in Figure stores 128 bits and requires 14 external connections for address, data, and control lines. It also needs two lines for power supply and ground connections. Consider now a slightly larger memory circuit, one that has 1K (1024) memory cells. This circuit can be organized as a 128 × 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a 1K×1 format. In this case, a 10-bit address is needed, but there is only one data line, resulting in 15 external connections.

STATIC MEMORIES

Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memories. Figure illustrates how a static RAM (SRAM) cell may be implemented. Two inverters are cross connected to form a latch. The latch is connected to two bit lines by transistors T1 and T2. These transistors act as switches that can be opened or closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state. For example, if the logic value at point X is 1 and at point Y is 0, this state is maintained as long as the signal on the word line is at ground level. Assume that this state represents value 1.



Read Operation

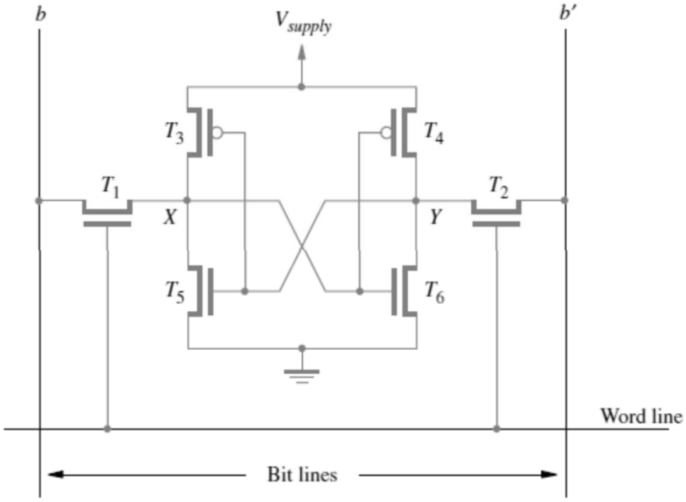
In order to read the state of the SRAM cell, the word line is activated to close switches T1 and T2. If the cell is in state 1, the signal on bit line b is high and the signal on bit line b’ is low. The opposite is true if the cell is in state 0. Thus, b and b’ are always complements of each other. The Sense/Write circuit at the end of the two bit lines monitors their state and sets the corresponding output accordingly.

Write Operation

During a Write operation, the Sense/Write circuit drives bit lines b and b’, instead of sensing their state. It places the appropriate value on bit line b and its complement on b’ and activates the word line. This forces the cell into the corresponding state, which the cell retains when the word line is deactivated.

CMOS Cell

A CMOS realization of the cell in Figure is given in Figure. Transistor pairs (T3, T5) and (T4, T6) form the inverters in the latch (see Appendix A). The state of the cell is read or written as just explained. For example, in state 1, the voltage at point X is maintained high by having transistors T3 and T6 on, while T4 and T5 are off. If T1 and T2 are turned on, bit lines b and bwill have high and low signals, respectively.



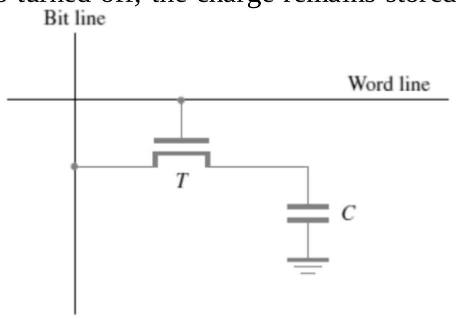
Continuous power is needed for the cell to retain its state. If power is interrupted, the cell’s contents are lost. When power is restored, the latch settles into a stable state, but not necessarily the same state the cell was in before the interruption. Hence, SRAMs are said to be volatile memories because their contents are lost when power is interrupted. A major advantage of CMOS SRAMs is their very low power consumption, because current flows in the cell only when the cell is being accessed. Otherwise, T1, T2, and one transistor in each inverter are turned off, ensuring that there is no continuous electrical path between Vsupply and ground.

Static RAMs can be accessed very quickly. Access times on the order of a few nanoseconds are found in commercially available chips. SRAMs are used in applications where speed is of critical concern.

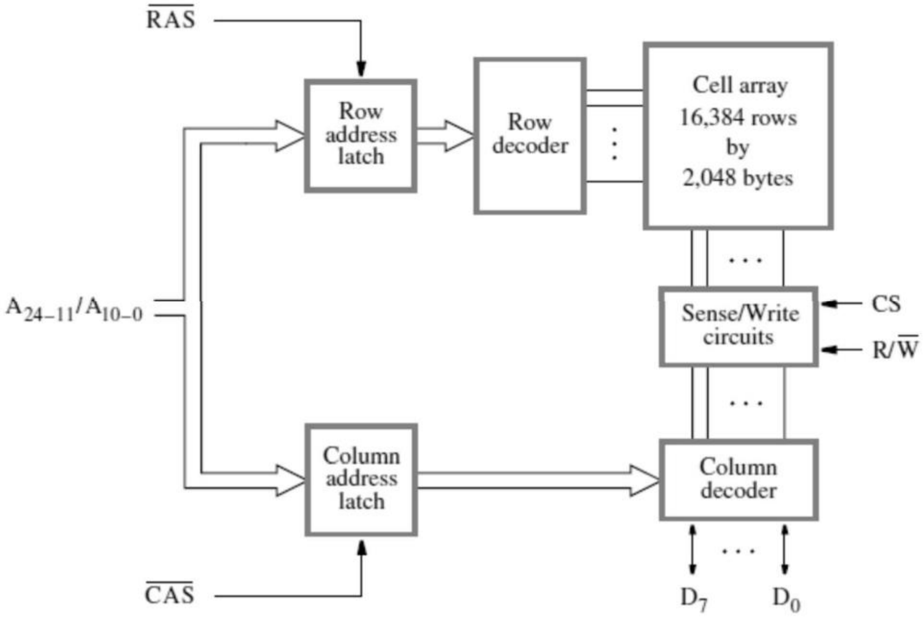
DYNAMIC RAMS

Static RAMs are fast, but their cells require several transistors. Less expensive and higher density RAMs can be implemented with simpler cells. But, these simpler cells do not retain their state for a long period, unless they are accessed frequently for Read or Write operations. Memories that use such cells are called dynamic RAMs (DRAMs). Information is stored in a dynamic memory cell in the form of a charge on a capacitor, but this charge can be maintained for only tens of milliseconds. Since the cell is required to store information for a much longer time, its contents must be periodically refreshed by restoring the capacitor charge to its full value. This occurs when the contents of the cell are read or when new information is written into it. An example of a dynamic memory cell that consists of a capacitor, C, and a transistor, T, is shown in Figure. To store information in this cell, transistor T is turned on and an appropriate voltage is applied to the bit line. This causes a known amount of charge to be stored in the capacitor.

After the transistor is turned off, the charge remains stored in the capacitor, but not for long. The capacitor begins to discharge. This is because the transistor continues to conduct a tiny amount of current, measured in picoamperes, after it is turned off. Hence, the information stored in the cell can be retrieved correctly only if it is read before the charge in the capacitor drops below some threshold value. During a Read operation, the transistor in a selected cell is turned on. A sense amplifier connected to the bit line detects whether the charge stored in the capacitor is above or below the threshold value. If the charge is above the threshold, the sense amplifier drives the bit line to the full voltage representing the logic value 1. As a result, the capacitor is recharged to the full charge corresponding to logic value 1. If the sense amplifier detects that the charge in the capacitor is below the threshold value, it pulls the bit line to ground level to discharge the capacitor fully. Thus, reading the contents of a cell automatically refreshes its contents. Since the word line is common to all cells in a row, all cells in a selected row are read and refreshed at the same time.



A256-Megabit DRAM chip, configured as 32M × 8, is shown in Figure. The cells are organized in the form of a 16K × 16K array. The 16,384 cells in each row are divided into 2,048 groups of 8, forming 2,048 bytes of data. Therefore, 14 address bits are needed to select a row, and another 11 bits are needed to specify a group of 8 bits in the selected row. In total, a 25-bit address is needed to access a byte in this memory. The high-order 14 bits and the low- order 11 bits of the address constitute the row and column addresses of a byte, respectively. To reduce the number of pins needed for external connections, the row and column addresses are multiplexed on 14 pins. During a Read or a Write operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on an input control line called the Row Address Strobe (RAS). This causes a Read operation to be initiated, in which all cells in the selected row are read and refreshed.



Shortly after the row address is loaded, the column address is applied to the address pins and loaded into the column address latch under control of a second control line called the Column Address Strobe (CAS). The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits is selected. If the R/W control signal indicates a Read operation, the output values of the selected circuits are transferred to the data lines, D7−0. For a Write operation, the information on theD7−0 lines is transferred to the selected circuits, then used to overwrite the contents of the selected cells in the corresponding 8 columns. We should note that in commercial DRAM chips, the RAS and CAS control signals are active when low. Hence, addresses are latched when these signals change from high to low. The signals are shown in diagrams as RAS and CAS to indicate this fact. The timing of the operation of the

DRAM described above is controlled by the RAS and CAS signals. These signals are generated by a memory controller circuit external to the chip when the processor issues a Read or a Write command. During a Read operation, the output data are transferred to the processor after a delay equivalent to the memory's access time. Such memories are referred to as asynchronous DRAMs. The memory controller is also responsible for refreshing the data stored in the memory chips, as we describe later.

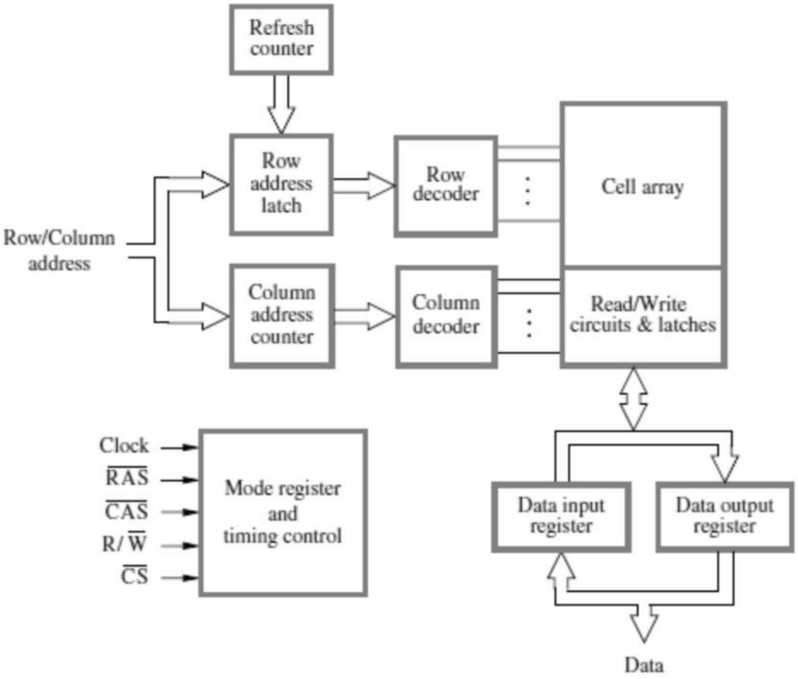
Fast Page Mode

When the DRAM in Figure is accessed, the contents of all 16,384 cells in the selected row are sensed, but only 8 bits are placed on the data lines, D7−0. This byte is selected by the column address, bitsA10−0. A simple addition to the circuit makes it possible to access the other bytes in the same row without having to reselect the row. Each sense amplifier also acts as a latch. When a row address is applied, the contents of all cells in the selected row are loaded into the corresponding latches. Then, it is only necessary to apply different column addresses to place the different bytes on the data lines.

This arrangement leads to a very useful feature. All bytes in the selected row can be transferred in sequential order by applying a consecutive sequence of column addresses under the control of successive CAS signals. Thus, a block of data can be transferred at a much faster rate than can be achieved for transfers involving random addresses. The block transfer capability is referred to as the fast page mode feature. (A large block of data is often called a page.) It was pointed out earlier that the vast majority of main memory transactions involve block transfers. The faster rate attainable in the fast page mode makes dynamic RAMs particularly well suited to this environment.

SYNCHRONOUS DRAMS

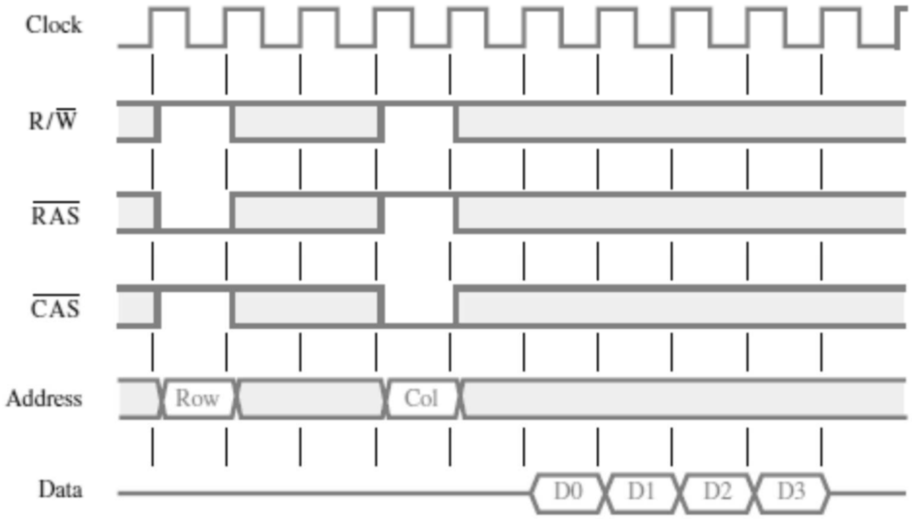
In the early 1990s, developments in memory technology resulted in DRAMs whose operation is synchronized with a clock signal. Such memories are known as synchronous DRAMs (SDRAMs). Their structure is shown in Figure. The cell array is the same as in asynchronous DRAMs. The distinguishing feature of an SDRAM is the use of a clock signal, the availability of which makes it possible to incorporate control circuitry on the chip that provides many useful features. For example, SDRAMs have built-in refresh circuitry, with a refresh counter to provide the addresses of the rows to be selected for refreshing. As a result, the dynamic nature of these memory chips is almost invisible to the user. The address and data connections of an SDRAM may be buffered by means of registers, as shown in the figure. Internally, the Sense/Write amplifiers function as latches, as in asynchronous DRAMs. A Read operation causes the contents of all cells in the selected row to be loaded into these latches. The data in the latches of the selected column are transferred into the data register, thus becoming available on the data output pins. The buffer registers are useful when transferring large blocks of data at very high speed. By isolating external connections from the chip's internal circuitry, it becomes possible to start a new access operation while data are being transferred to or from the registers.



SDRAMs have several different modes of operation, which can be selected by writing control information into a mode register. For example, burst operations of different lengths can be specified. It is not necessary to provide externally-generated pulses on the CAS line to select successive columns. The necessary control signals are generated internally using a column counter and the clock signal. New data are placed on the data lines at the rising edge of each clock pulse.

Figure shows a timing diagram for a typical burst read of length 4. First, the row address is latched under control of the RAS signal. The memory typically takes 5 or 6 clock cycles (we use 2 in the figure for simplicity) to activate the selected row. Then, the column address is latched under control of the CAS signal. After a delay of one clock cycle, the first set of data bits is placed on the data lines. The SDRAM automatically increments the column address to access the next three sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

Synchronous DRAMs can deliver data at a very high rate, because all the control signals needed are generated inside the chip. The initial commercial SDRAMs in the 1990s were designed for clock speeds of up to 133 MHz. As technology evolved, much faster SDRAM chips were developed. Today's SDRAMs operate with clock speeds that can exceed 1 GHz.



Latency and Bandwidth

Data transfers to and from the main memory often involve blocks of data. The speed of these transfers has a large impact on the performance of a computer system. The memory access time defined earlier is not sufficient for describing the memory's performance when transferring blocks of data. During block transfers, memory latency is the amount of time it takes to transfer the first word of a block. The time required to transfer a complete block depends also on the rate at which successive words can be transferred and on the size of the block. The time between successive words of a block is much shorter than the time needed to transfer the first word. For instance, in the timing diagram in Figure, the access cycle begins with the assertion of the RAS signal. The first word of data is transferred five clock cycles later. Thus, the latency is five clock cycles. If the clock rate is 500 MHz, then the latency is 10 ns. The remaining three words are transferred in consecutive clock cycles, at the rate of one word every 2 ns.

The example above illustrates that we need a parameter other than memory latency to describe the memory's performance during block transfers. A useful performance measure is the number of bits or bytes that can be transferred in one second. This measure is often referred to as the memory bandwidth. It depends on the speed of access to the stored data and on the number of bits that can be accessed in parallel. The rate at which data can be transferred to or from the memory depends on the bandwidth of the system interconnections. For this reason, the interconnections used always ensure that the bandwidth available for data transfers between the processor and the memory is very high.

Double-Data-Rate SDRAM

In the continuous quest for improved performance, faster versions of SDRAMs have been developed. In addition to faster circuits, new organizational and operational features make it possible to achieve high data rates during block transfers. The key idea is to take advantage of the fact that a large number of bits are accessed at the same time inside the chip when a row address is applied. Various techniques are used to transfer these bits quickly to the pins of the chip. To make the best use of the available clock speed, data are transferred externally on both the rising and falling edges of the clock. For this reason, memories that use this technique are called double-data-rate SDRAMs (DDR SDRAMs). Several versions of DDR chips have been developed. The earliest version is known as DDR. Later versions, called DDR2, DDR3, and DDR4, have enhanced capabilities. They offer increased storage capacity, lower power, and faster clock speeds. For example, DDR2 and DDR3 can operate at clock frequencies of 400 and 800 MHz, respectively. Therefore, they transfer data using the effective clock speeds of 800 and 1600 MHz, respectively.

Rambus Memory

The rate of transferring data between the memory and the processor is a function of both the bandwidth of the memory and the bandwidth of its connection to the processor. Rambus is a memory technology that achieves a high data transfer rate by providing a high- speed interface between the memory and the processor. One way for increasing the bandwidth of this connection is to use a wider data path. However, this requires more space and more pins, increasing system cost. The alternative is to use fewer wires with a higher clock speed. This is the approach taken by Rambus. Rambus technology competes directly with the DDR SDRAM technology. Each has certain advantages and disadvantages. Anontechnical consideration is that the specification of DDR SDRAM is an open standard that can be used free of charge. Rambus, on the other hand, is a proprietary scheme that must be licensed by chip manufacturers.

Refresh Overhead

A dynamic RAM cannot respond to read or write requests while an internal refresh operation is taking place. Such requests are delayed until the refresh cycle is completed. However, the time lost to accommodate refresh operations is very small. For example, consider an SDRAM in which each row needs to be refreshed once every 64 ms. Suppose that the minimum time between two row accesses is 50 ns and that refresh operations are arranged such that all rows of the chip are refreshed in 8K (8192) refresh cycles. Thus, it takes 8192 × 0.050

= 0.41 ms to refresh all rows. The refresh overhead is 0.41/64 = 0.0064, which is less than 1 percent of the total time available for accessing the memory.

Choice of Technology

The choice of an RAM chip for a given application depends on several factors. Foremost among these are the cost, speed, power dissipation, and size of the chip. Static RAMs are characterized by their very fast operation. However, their cost and bit density are adversely affected by the complexity of the circuit that realizes the basic cell. They are used mostly where a small but very fast memory is needed. Dynamic RAMs, on the other hand, have high bit densities and a low cost per bit. Synchronous DRAMs are the predominant choice for implementing the main memory.

RAM

Both static and dynamic RAM chips are volatile, which means that they retain information only while power is turned on. There are many applications requiring memory devices that retain the stored information when power is turned off. For example, the need to store a small program in such a memory, to be used to start the bootstrap process of loading the operating system from a hard disk into the main memory. The embedded applications are another important example. Many embedded applications do not use a hard disk and require non-volatile memories to store their software.

Different types of non-volatile memories have been developed. Generally, their contents can be read in the same way as for their volatile counterparts discussed above. But, a special writing process is needed to place the information into a non-volatile memory. Since its normal operation involves only reading the stored data, a memory of this type is called a read-only memory (ROM).

ROM

A memory is called a read-only memory, or ROM, when information can be written into it only once at the time of manufacture. Figure 8.11 shows a possible configuration for a ROM cell. A logic value 0 is stored in the cell if the transistor is connected to ground at point P; otherwise, a 1 is stored. The bit line is connected through a resistor to the power supply. To read the state of the cell, the word line is activated to close the transistor switch.

As a result, the voltage on the bit line drops to near zero if there is a connection between the transistor and ground. If there is no connection to ground, the bit line remains at the high voltage level, indicating a 1. A sense circuit at the end of the bit line generates the proper output value. The state of the connection to ground in each cell is determined when the chip is manufactured, using a mask with a pattern that represents the information to be stored.

PROM

Some ROM designs allow the data to be loaded by the user, thus providing a programmable ROM (PROM). Programmability is achieved by inserting a fuse at point P, in Figure. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible. PROMs provide flexibility and convenience not available with ROMs. The cost of preparing the masks needed for storing a particular information pattern makes ROMs cost effective only in large volumes. The alternative technology of PROMs provides a more convenient and considerably less expensive approach, because memory chips can be programmed directly by the user.

EPROM

Another type of ROM chip provides an even higher level of convenience. It allows the stored data to be erased and new data to be written into it. Such an erasable, reprogrammable ROM is usually called an EPROM. It provides considerable flexibility during the development phase of digital systems. Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs or PROMs while software is being developed.

In this way, memory changes and updates can be easily made. An EPROM cell has a structure similar to the ROM cell in Figure. However, the connection to ground at point P is made through a special transistor. The transistor is normally turned off, creating an open switch. It can be turned on by injecting charge into it that becomes trapped inside. Thus, an EPROM cell can be used to construct a memory in the same way as the previously discussed ROM cell. Erasure requires dissipating the charge trapped in the transistors that form the memory cells. This can be done by exposing the chip to ultraviolet light, which erases the entire contents of the chip. To make this possible, EPROM chips are mounted in packages that have transparent windows.

EEPROM

An EPROM must be physically removed from the circuit for reprogramming. Also, the stored information cannot be erased selectively. The entire contents of the chip are erased when exposed to ultraviolet light. Another type of erasable PROM can be programmed, erased, and reprogrammed electrically. Such a chip is called an electrically erasable PROM, or EEPROM. It does not have to be removed for erasure. Moreover, it is possible to erase the cell contents selectively. One disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data, which increases circuit complexity. However, this disadvantage is outweighed by the many advantages of EEPROMs. They have replaced EPROMs in practice.

Flash Memory

An approach similar to EEPROM technology has given rise to flash memory devices. A flash cell is based on a single transistor controlled by trapped charge, much like an EEPROM cell. Also like an EEPROM, it is possible to read the contents of a single cell. The key difference is that, in a flash device, it is only possible to write an entire block of cells. Prior to writing, the previous contents of the block are erased. Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and consume less power in their operation.

The low power consumption of flash memories makes them attractive for use in portable, battery-powered equipment. Typical applications include hand-held computers, cell phones, digital cameras, and MP3 music players. In hand-held computers and cell phones, a flash memory holds the software needed to operate the equipment, thus obviating the need for a disk drive. A flash memory is used in digital cameras to store picture data. In MP3 players, flash memories store the data that represent sound. Cell phones, digital cameras, and MP3 players are good examples of embedded systems. Single flash chips may not provide sufficient storage capacity for the applications. Larger memory modules consisting of a number of chips are used where needed. There are two popular choices for the implementation of such modules: flash cards and flash drives.

Flash Cards

One way of constructing a larger module is to mount flash chips on a small card. Such flash cards have a standard interface that makes them usable in a variety of products. A card is simply plugged into a conveniently accessible slot. Flash cards with a USB interface are widely used and are commonly known as memory keys. They come in a variety of memory sizes. Larger cards may hold as much as 32 Gbytes. A minute of music can be stored in about 1 Mbyte of memory, using the MP3 encoding format. Hence, a 32-Gbyte flash card can store approximately 500 hours of music.

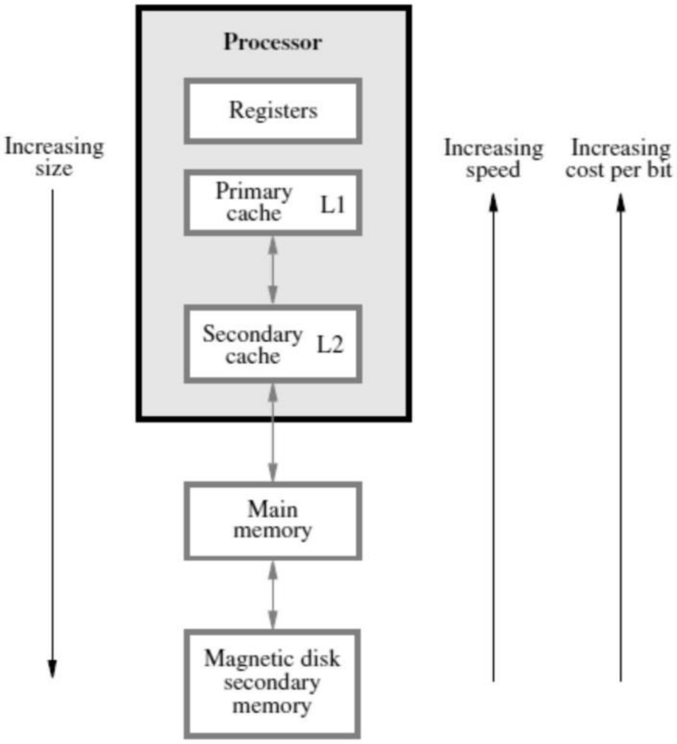
Flash Drives

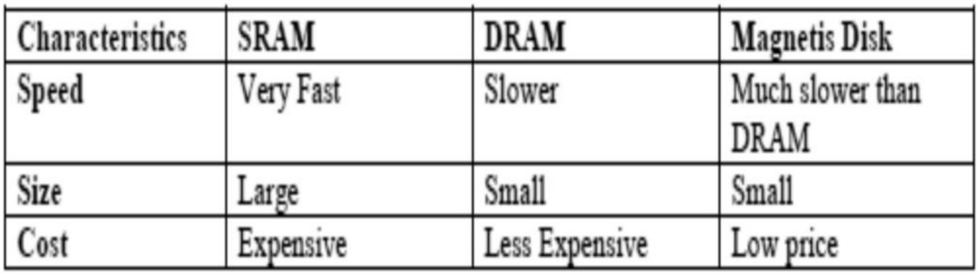
Larger flash memory modules have been developed to replace hard disk drives, and hence are called flash drives. They are designed to fully emulate hard disks, to the point that they can be fitted into standard disk drive bays. However, the storage capacity of flash drives is significantly lower. Currently, the capacity of flash drives is on the order of 64 to 128 Gbytes. In contrast, hard disks have capacities exceeding a terabyte. Also, disk drives have a very low cost per bit.

The fact that flash drives are solid state electronic devices with no moving parts provides important advantages over disk drives. They have shorter access times, which result in a faster response. They are insensitive to vibration and they have lower power consumption, which makes them attractive for portable, battery-driven applications.

SPEED, SIZE AND COST – REVIEW ABOUT MEMORY HIERARCHY

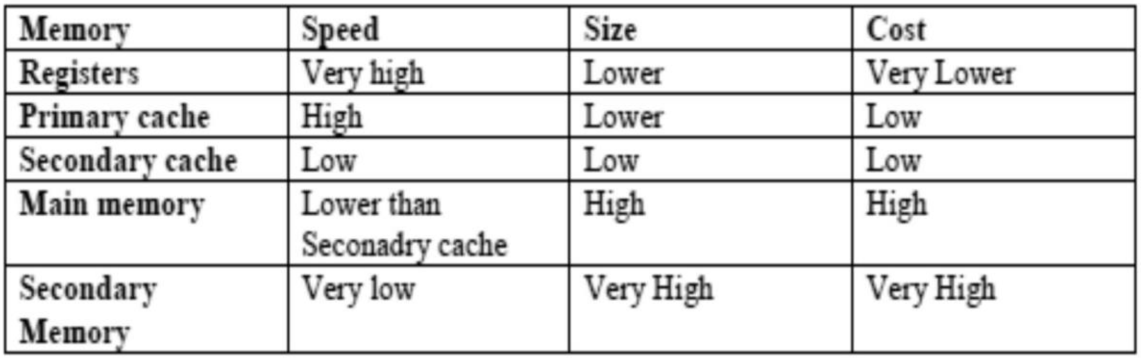
An ideal memory would be fast, large, and inexpensive. It is clear that a very fast memory can be implemented using static RAM chips. But, these chips are not suitable for implementing large memories, because their basic cells are larger and consume more power than dynamic RAM cells. Although dynamic memory units with gigabyte capacities can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data. A solution is provided by using secondary storage, mainly magnetic disks, to provide the required memory space. Disks are available at a reasonable cost, and they are used extensively in computer systems. However, they are much slower than semiconductor memory units.





Characteristics	SRAM	DRAM	Magnetis Disk
Speed	Very Fast	Slower	Much slower than DRAM
Size	Large	Small	Small
Cost	Expensive	Less Expensive	Low price

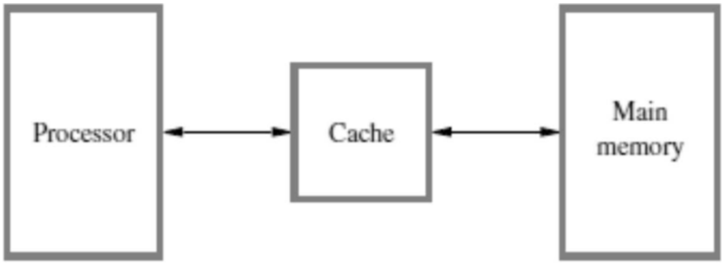
In summary, a very large amount of cost-effective storage can be provided by magnetic disks, and a large and considerably faster, yet affordable, main memory can be built with dynamic RAM technology. This leaves the more expensive and much faster static RAM technology to be used in smaller units where speed is of the essence, such as in cache memories. huge amount of cost effective storage can be provided by magnetic disk; The main memory can be built with DRAM which leaves SRAM‟s to be used in smaller units where speed is of essence



Memory	Speed	Size	Cost
Registers	Very high	Lower	Very Lower
Primary cache	High	Lower	Low
Secondary cache	Low	Low	Low
Main memory	Lower than Seconadry cache	High	High
Secondary Memory	Very low	Very High	Very High

CACHE MEMORIES

The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on the property of computer programs called locality of reference. Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other. The actual detailed pattern of instruction sequencing is not important—the point is that many instructions in localized areas of the program are executed repeatedly during some time period. This behaviour manifests itself in two ways: temporal and spatial. The first means that a recently executed instruction is likely to be executed again very soon. The spatial aspect means that instructions close to a recently executed instruction are also likely to be executed soon.



Conceptually, operation of a cache memory is very simple. The memory control circuitry is designed to take advantage of the property of locality of reference. Temporal locality suggests that whenever an information item, instruction or data, is first needed, this item should be brought into the cache, because it is likely to be needed again soon. Spatial locality suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that are located at adjacent addresses as well. The term cache block refers to a set of contiguous address locations of some size. Another term that is often used to refer to a cache block is a cache line. Consider the arrangement in Figure. When the processor issues a Read request, the contents of a block of memory words containing the location specified are transferred into the cache. Subsequently, when the program references any of the locations in this block, the desired contents are read directly from the cache. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function. When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache’s replacement algorithm.

Cache Hits

The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a read or write hit is said to have occurred. The main memory is not involved when there is a cache hit in a Read operation. For a Write operation, the system can proceed in one of two ways. In the first technique, called the write-through protocol, both the cache location and the main memory location are updated. The second technique is to update only the cache location and to mark the block containing it with an associated flag bit, often called the dirty or modified bit. The main memory location of the word is updated later, when the block containing this marked word is removed from the cache to make room for a new block. This technique is known as the write-back, or copy-back, protocol. The write-through protocol is simpler than the write-back protocol, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. The write-back protocol also involves unnecessary Write operations, because all words of the block are eventually written back, even if only a single word has been changed while the block was in the cache. The write-back protocol is used most often, to take advantage of the high speed with which data blocks can be transferred to memory chips.

Cache Misses

A Read operation for a word that is not in the cache constitutes a Read miss. It causes the block of words containing the requested word to be copied from the main memory into the cache. After the entire block is loaded into the cache, the particular word requested is forwarded to the processor. Alternatively, this word may be sent to the processor as soon as it is read from the main memory. The latter approach, which is called load-through, or early restart, reduces the processor’s waiting time somewhat, at the expense of more complex circuitry.

When a Write miss occurs in a computer that uses the write-through protocol, the information is written directly into the main memory. For the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information. Resource limitations in a pipelined processor can cause instruction execution to stall for one or more cycles. This can occur if a Load or Store instruction requests access to data in the memory at the same time that a subsequent instruction is being fetched. When this happens, instruction fetch is delayed until the data access operation is completed. To avoid stalling the pipeline, many processors use separate caches for instructions and data, making it possible for the two operations to proceed in parallel.

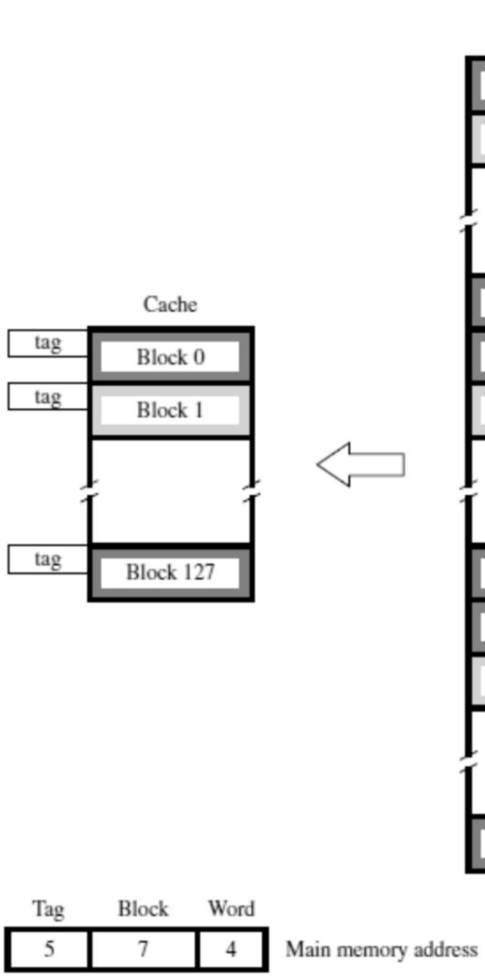
MAPPING FUNCTIONS

There are several possible methods for determining where memory blocks are placed in the cache. It is instructive to describe these methods using a specific small example. Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by a 16-bit address. The main memory has 64K words, which we will view as 4K blocks of 16 words each. For simplicity, we have assumed that consecutive addresses refer to consecutive words.

Direct Mapping

The simplest way to determine cache locations in which to store memory blocks is the direct-mapping technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure. Thus, whenever one of the main memory blocks 0, 128, 256, . . . is loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full. For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

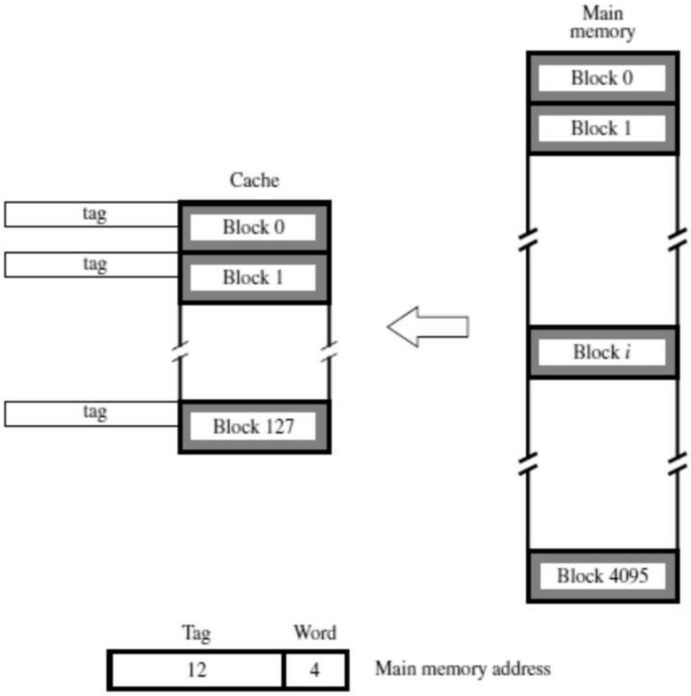
With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure. The low-order 4 bits select one of 16 words in a block. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The high-order 5 bits of the memory address of the block are stored in 5 tag bits associated with its location in the cache. The tag bits identify which of the 32 main memory blocks mapped into this cache position is currently resident in the cache. As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache. The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. The direct-mapping technique is easy to implement, but it is not very flexible.



Associative Mapping

Figure shows the most flexible mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the associative-mapping technique. It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced. Many replacement algorithms are possible. The complexity of an associative

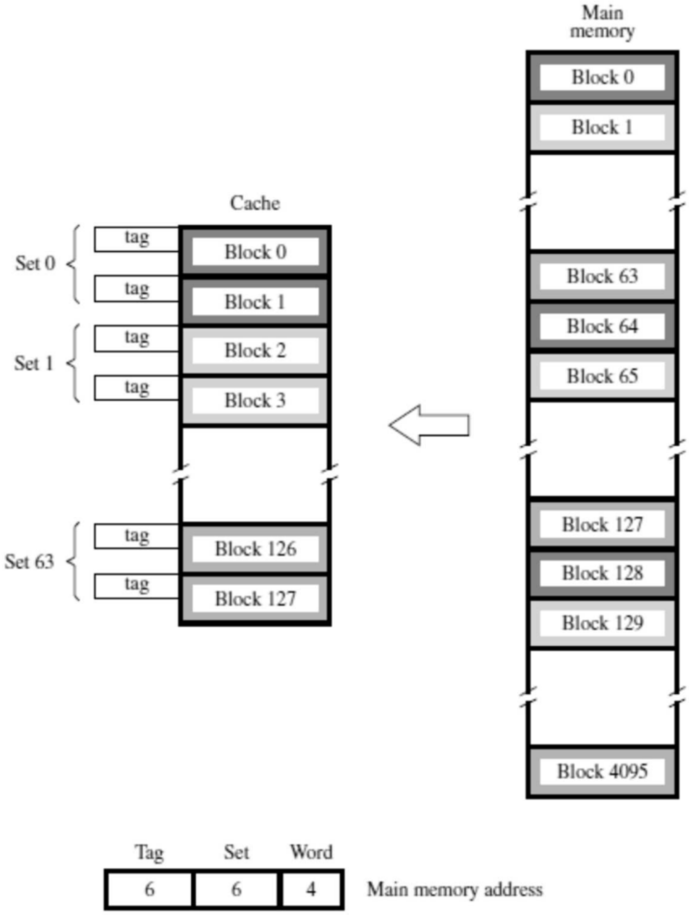
The cache is higher than that of a direct-mapped cache, because of the need to search all 128 tag patterns to determine whether a given block is in the cache. To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an associative search.



Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this set-associative-mapping technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping method. A cache that has k blocks per set is referred to as a k-way set-associative cache.



Stale Data

When power is first turned on, the cache contains no valid data. A control bit, usually called the valid bit, must be provided for each cache block to indicate whether the data in that block are valid. This bit should not be confused with the modified, or dirty, bit mentioned earlier. The valid bits of all cache blocks are set to 0 when power is initially applied to the system. Some valid bits may also be set to 0 when new programs or data are loaded from the disk into the main memory. Data transferred from the disk to the main memory using the DMA mechanism are usually loaded directly into the main memory, bypassing the cache. If the memory blocks being updated are currently in the cache, the valid bits of the corresponding cache blocks are set to 0. As program execution proceeds, the valid bit of a given cache block is set to 1 when a memory block is loaded into that location. The processor fetches data from a cache block only if its valid bit is equal to 1. The use of the valid bit in this manner ensures that the processor will not fetch stale data from the cache.

A similar precaution is needed in a system that uses the write-back protocol. Under this protocol, new data written into the cache are not written to the memory at the same time. Hence, data in the memory do not always reflect the changes that may have been made in the cached copy. It is important to ensure that such stale data in the memory are not transferred to the disk. One solution is to flush the cache, by forcing all dirty blocks to be written back to the memory before performing the transfer. The operating system can do this by issuing a command to the cache before initiating the DMA operation that transfers the data to the disk. Flushing the cache does not affect performance greatly, because such disk transfers do not occur often. The need to ensure that two different entities (the processor and the DMA subsystems in this case) use identical copies of the data is referred to as a cache-coherence problem.

REPLACEMENT ALGORITHMS

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced. The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the least recently used (LRU) block, and the technique is called the LRU replacement algorithm.

To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds. Suppose it is required to track the LRU block of a four-block set in a set-associative cache. A 2-bit counter can be used for each block. When a hit occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged. When a miss occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one. When a miss occurs and the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0. The other three block counters are incremented by one. It can be easily verified that the counter values of occupied blocks are always distinct.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache. Performance of the LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.

Several other replacement algorithms are also used in practice. An intuitively reasonable rule would be to remove the “oldest” block from a full set when a new block must be brought in. However, because this algorithm does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU algorithm in choosing the best blocks to remove. The simplest algorithm is to randomly choose the block to be overwritten. Interestingly enough, this simple algorithm has been found to be quite effective in practice.

IMPROVING CACHE PERFORMANCE

Two key factors in the commercial success of a computer are performance and cost; the best possible performance for a given cost is the objective. A common measure of success is the price/performance ratio. Performance depends on how fast machine instructions can be brought into the processor and how fast they can be executed. The main purpose of this hierarchy is to create a memory that the processor sees as having a short access time and a large capacity. When a cache is used, the processor is able to access instructions and data more quickly when the data from the referenced memory locations are in the cache. Therefore, the extent to which caches improve performance is dependent on how frequently the requested instructions and data are found in the cache. In this section, we examine this issue quantitatively.

Hit Rate and Miss Penalty

An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various levels of the hierarchy. Recall that a successful access to data in a cache is called a hit. The number of hits stated as a fraction of all attempted accesses is called the hit rate, and the miss rate is the number of misses stated as a fraction of attempted accesses. Ideally, the entire memory hierarchy would appear to the processor as a single memory unit that has the access time of the cache on the processor chip and the size of the magnetic disk. How close we get to this ideal depends largely on the hit rate at different levels of the hierarchy. High hit rates well over 0.9 are essential for high-performance computers.

Performance is adversely affected by the actions that need to be taken when a miss occurs. A performance penalty is incurred because of the extra time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. During that period, the processor is stalled waiting for instructions or data. The waiting time depends on the details of the operation of the cache. For example, it depends on whether or not the load-through approach is used. We refer to the total access time seen by the processor when a miss occurs as the miss penalty.

Consider a system with only one level of cache. In this case, the miss penalty consists almost entirely of the time to access a block of data in the main memory. Let h be the hit rate, M the miss penalty, and C the time to access information in the cache. Thus, the average access time experienced by the processor is

$$t_{avg} = hC + (1 - h) M$$

How can the hit rate be improved? One possibility is to make the cache larger, but this entails increased cost. Another possibility is to increase the cache block size while keeping the total cache size constant, to take advantage of spatial locality. If all items in a larger block are needed in a computation, then it is better to load these items into the cache in a single miss, rather than loading several smaller blocks as a result of several misses. The high data rate achievable during block transfers is the main reason for this advantage. But larger blocks are effective only up to a certain size, beyond which the improvement in the hit rate is offset by the fact that some items may not be referenced before the block is ejected (replaced). Also, larger blocks take longer to transfer, and hence increase the miss penalty. Since the performance of a computer is affected positively by increased hit rate and negatively by increased miss penalty, block size should be neither too small nor too large. In practice, block sizes in the range of 16 to 128 bytes are the most popular choices. Finally, we note that the miss penalty can be reduced if the load-through approach is used when loading new blocks into the cache. Then, instead of waiting for an entire block to be transferred, the processor resumes execution as soon as the required word is loaded into the cache.

Caches on the Processor Chip

When information is transferred between different chips, considerable delays occur in driver and receiver gates on the chips. Thus, it is best to implement the cache on the processor chip. Most processor chips include at least one L1 cache. Often there are two separate L1 caches, one for instructions and another for data. In high-performance processors, two levels of caches are normally used, separate L1 caches for instructions and data and a larger L2 cache. These caches are often implemented on the processor chip. In this case, the L1 caches must be very fast, as they determine the memory access time seen by the processor. The L2 cache can be slower, but it should be much larger than the L1 caches to ensure a high hit rate. Its speed is less critical because it only affects the miss penalty of the L1 caches. A typical computer may have L1 caches with capacities of tens of kilobytes and an L2 cache of hundreds of kilobytes or possibly several megabytes.

Other Enhancements

In addition to the main design issues just discussed, several other possibilities exist for enhancing performance. We discuss three of them in this section.

Write Buffer

When the write-through protocol is used, each Write operation results in writing a new value into the main memory. If the processor must wait for the memory function to be completed, as we have assumed until now, then the processor is slowed down by all Write requests. Yet the processor typically does not need immediate access to the result of a Write operation; so it is not necessary for it to wait for the Write request to be completed To improve performance, a Write buffer can be included for temporary storage of Write requests. The processor places each Write request into this buffer and continues execution of the next instruction. The Write requests stored in the Write buffer are sent to the main memory whenever the memory is not responding to Read requests. It is important that the Read requests be serviced quickly, because the processor usually cannot proceed before receiving the data being read from the memory. Hence, these requests are given priority overwrite requests.

The Write buffer may hold a number of Write requests. Thus, it is possible that a subsequent Read request may refer to data that are still in the Write buffer. To ensure correct operation, the addresses of data to be read from the memory are always compared with the addresses of the data in the Write buffer. In the case of a match, the data in the Write buffer are used. A similar situation occurs with the write-back protocol. In this case, Write commands issued by the processor are performed on the word in the cache. When a new block of data is to be brought into the cache as a result of a Read miss, it may replace an existing block that has some dirty data. The dirty block has to be written into the main memory. If the required write- back is performed first, then the processor has to wait for this operation to be completed before the new block is read into the cache. It is more prudent to read the new block first. The dirty block being ejected from the cache is temporarily stored in the Write buffer and held there while the new block is being read. Afterwards, the contents of the buffer are written into the main memory. Thus, theWrite buffer also works well for the write-back protocol.

Prefetching

In the previous discussion of the cache mechanism, we assumed that new data are brought into the cache when they are first needed. Following a Read miss, the processor has to pause until the new data arrive, thus incurring a miss penalty. To avoid stalling the processor, it is possible to prefetch the data into the cache before they are needed. The simplest way to do this is through software. A special prefetch instruction may be provided in the instruction set of the processor. Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a Read miss. Aprefetch instruction is inserted in a program to cause the data to be loaded in the cache shortly before they are needed in the program. Then, the processor will not have to wait for the referenced data as in the case of a Read miss. The hope is that prefetching will take place while the processor is busy executing instructions that do not result in a Read miss, thus allowing accesses to the main memory to be overlapped with computation in the processor.

Prefetch instructions can be inserted into a program either by the programmer or by the compiler. Compilers are able to insert these instructions with good success for many applications. Software prefetching entails a certain overhead because inclusion of prefetch instructions increases the length of programs. Moreover, some prefetches may load into the cache data that will not be used by the instructions that follow. This can happen if the prefetched data are ejected from the cache by a Read miss involving other data. However, the overall effect of software prefetching on performance is positive, and many processors have machine instructions to support this feature. Prefetching can also be done in hardware, using circuitry that attempts to discover a pattern in memory references and prefetches data according to this pattern.

Lockup-Free Cache

Software prefetching does not work well if it interferes significantly with the normal execution of instructions. This is the case if the action of prefetching stops other accesses to the cache until the prefetch is completed. While servicing a miss, the cache is said to be locked. This problem can be solved by modifying the basic cache structure to allow the processor to access the cache while a miss is being serviced. In this case, it is possible to have more than one outstanding miss, and the hardware must accommodate such occurrences.

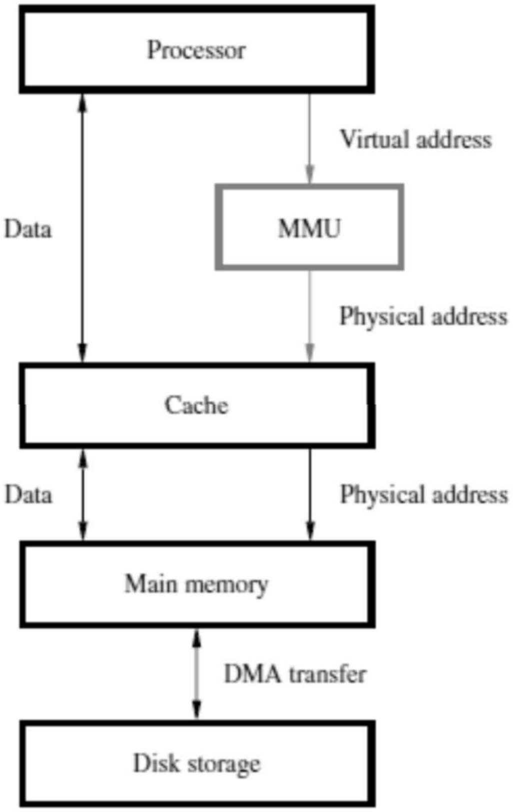
A cache that can support multiple outstanding misses is called lockup-free. Such a cache must include circuitry that keeps track of all outstanding misses. This may be done with special registers that hold the pertinent information about these misses. We have used software prefetching to motivate the need for a cache that is not locked by a Read miss. A much more important reason is that in a pipelined processor, which overlaps the execution of several instructions, a Read miss caused by one instruction could stall the execution of other instructions. A lockup-free cache reduces the likelihood of such stalls.

VIRTUAL MEMORY

In most modern computer systems, the physical main memory is not as large as the address space of the processor. For example, a processor that issues 32-bit addresses has an addressable space of 4G bytes. The size of the main memory in a typical computer with a 32- bit processor may range from 1G to 4G bytes. If a program does not completely fit into the main memory, the parts of it not currently being executed are stored on a secondary storage device, typically a magnetic disk. As these parts are needed for execution, they must first be brought into the main memory, possibly replacing other parts that are already in the memory. These actions are performed automatically by the operating system, using a scheme known as virtual memory. Application programmers need not be aware of the limitations imposed by the available main memory. They prepare programs using the entire address space of the processor.

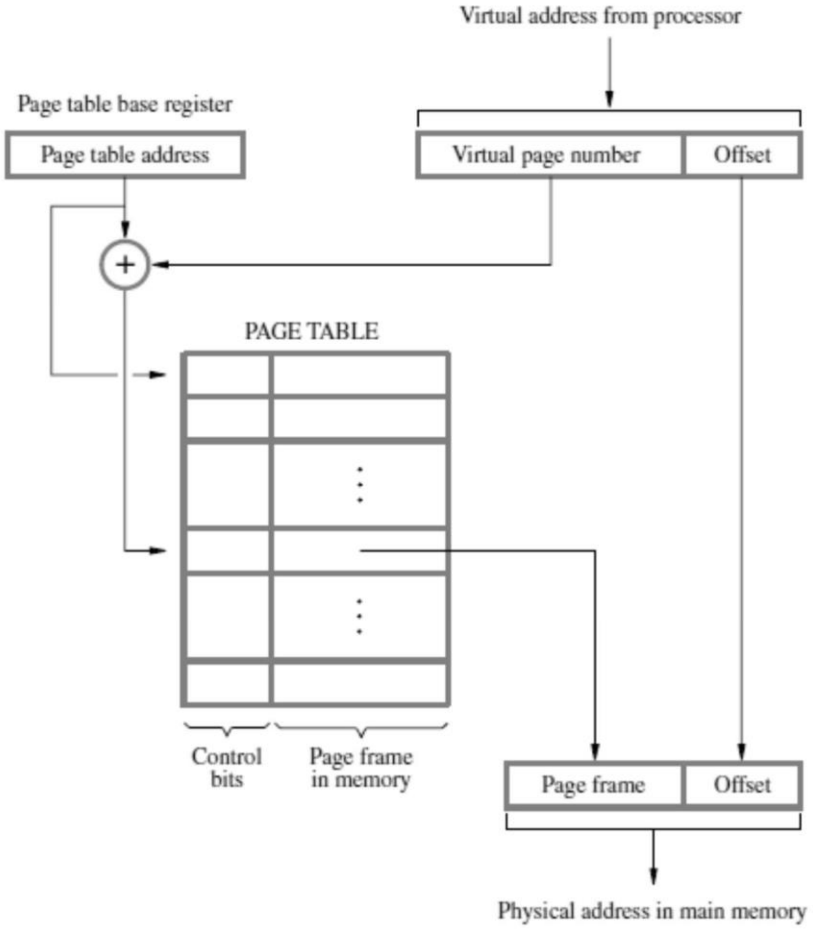
Under a virtual memory system, programs, and hence the processor, reference instructions and data in an address space that is independent of the available physical main memory space. The binary addresses that the processor issues for either instructions or data are called virtual or logical addresses. These addresses are translated into physical addresses by a combination of hardware and software actions. If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. Otherwise, the contents of the referenced address must be brought into a suitable location in the memory before they can be used.

Figure shows a typical organization that implements virtual memory. A special hardware unit, called the Memory Management Unit (MMU), keeps track of which parts of the virtual address space are in the physical memory. When the desired data or instructions are in the main memory, the MMU translates the virtual address into the corresponding physical address. Then, the requested memory access proceeds in the usual manner. If the data are not in the main memory, the MMU causes the operating system to transfer the data from the disk to the memory. Such transfers are performed using the DMA scheme.



Address Translation

A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called pages, each of which consists of a block of words that occupy contiguous locations in the main memory. Pages commonly range from 2K to 16K bytes in length. They constitute the basic unit of information that is transferred between the main memory and the disk whenever the MMU determines that a transfer is required. Pages should not be too small, because the access time of a magnetic disk is much longer (several milliseconds) than the access time of the main memory. The reason for this is that it takes a considerable amount of time to locate the data on the disk, but once located, the data can be transferred at a rate of several megabytes per second. On the other hand, if pages are too large, it is possible that a substantial portion of a page may not be used, yet this unnecessary data will occupy valuable space in the main memory. This discussion clearly parallels the concepts introduced on cache memory. The cache bridges the speed gap between the processor and the main memory and is implemented in hardware. The virtual-memory mechanism bridges the size and speed gaps between the main memory and secondary storage and is usually implemented in part by software techniques. Conceptually, cache techniques and virtual-memory techniques are very similar. They differ mainly in the details of their implementation.



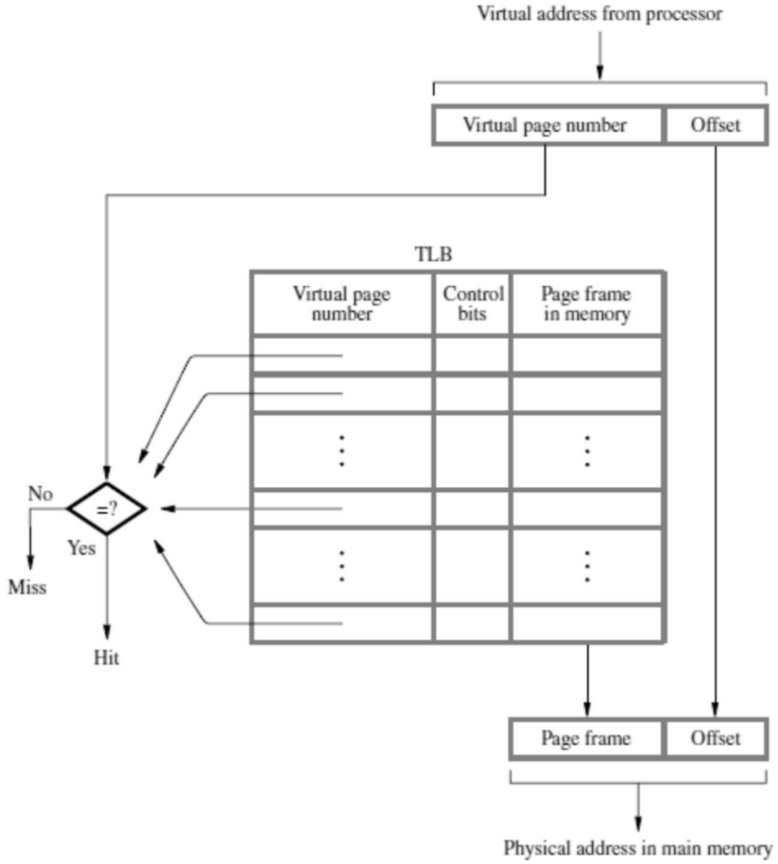
A virtual-memory address-translation method based on the concept of fixed-length pages is shown schematically in Figure. Each virtual address generated by the processor, whether it is for an instruction fetch or an operand load/store operation, is interpreted- as a virtual page number (high-order bits) followed by an offset (low-order bits) that specifies the location of a particular byte (or word) within a page. Information about the main memory location of each page is kept in a page table. This information includes the main memory address where the page is stored and the current status of the page. An area in the main memory that can hold one page is called a page frame. The starting address of the page table is kept in a page table base register. By adding the virtual page number to the contents of this register, the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the main memory.

Each entry in the page table also includes some control bits that describe the status of the page while it is in the main memory. One bit indicates the validity of the page, that is, whether the page is actually loaded in the main memory. It allows the operating system to invalidate the page without actually removing it. Another bit indicates whether the page has been modified during its residency in the memory. As in cache memories, this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page. Other control bits indicate various restrictions that may be imposed on accessing the page. For example, a program may be given full read and write permission, or it may be restricted to read accesses only.

Translation Lookaside Buffer

The page table information is used by the MMU for every read and write access. Ideally, the page table should be situated within the MMU. Unfortunately, the page table may be rather large. Since the MMU is normally implemented as part of the processor chip, it is impossible to include the complete table within the MMU. Instead, a copy of only a small portion of the table is accommodated within the MMU, and the complete table is kept in the main memory. The portion maintained within the MMU consists of the entries corresponding to the most recently accessed pages. They are stored in a small table, usually called the Translation Lookaside Buffer (TLB). The TLB functions as a cache for the page table in the main memory. Each entry in the TLB includes a copy of the information in the corresponding entry in the page table. In addition, it includes the virtual address of the page, which is needed to search the TLB for a particular page. Figure shows a possible organization of a TLB that uses the associative- mapping technique. Set-associated mapped TLBs are also found in commercial products.

Address translation proceeds as follows. Given a virtual address, the MMU looks in the TLB for the referenced page. If the page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated. It is essential to ensure that the contents of the TLB are always the same as the contents of page tables in the memory. When the operating system changes the contents of a page table, it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose. When an entry is invalidated, the TLB acquires the new information from the page table in the memory as part of the MMU's normal response to access misses.



Page Faults

When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred. The entire page must be brought from the disk into the memory before access can proceed. When it detects a page fault, the MMU asks the operating system to intervene by raising an exception (interrupt). Processing of the program that generated the page fault is interrupted, and control is transferred to the operating system. The operating system copies the requested page from the disk into the main memory. Since this process involves a long delay, the operating system may begin execution of another program whose pages are in the main memory. When page transfer is completed, the execution of the interrupted program is resumed.

When the MMU raises an interruption to indicate a page fault, the instruction that requested the memory access may have been partially executed. It is essential to ensure that the interrupted program continues correctly when it resumes execution. There are two options. Either the execution of the interrupted instruction continues from the point of interruption, or the instruction must be restarted. The design of a particular processor dictates which of these two options is used. If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. The problem of choosing which page to remove is just as critical here as it is in a cache, and the observation that programs spend most of their time in a few localized areas also applies. Because main memories are considerably larger than cache memories, it should be possible to keep relatively larger portions of a program in the main memory. This reduces the frequency of transfers to and from the disk. Concepts similar to the LRU replacement algorithm can be applied to page replacement, and the control bits in the page table entries can be used to record usage history. One simple scheme is based on a control bit that is set to 1 whenever the corresponding page is referenced (accessed). The operating system periodically clears this bit in all page table entries, thus providing a simple way of determining which pages have not been used recently.

A modified page has to be written back to the disk before it is removed from the main memory. It is important to note that the write-through protocol, which is useful in the framework of cache memories, is not suitable for virtual memory. The access time of the disk is so long that it does not make sense to access it frequently to write small amounts of data. Looking up entries in the TLB introduces some delay, slowing down the operation of the MMU. Here again we can take advantage of the property of locality of reference. It is likely that many successive TLB translations involve addresses on the same program page. This is particularly likely when fetching instructions. Thus, address translation time can be reduced by keeping the most recently used TLB entries in a few special registers that can be accessed quickly.

MEMORY MANAGEMENT REQUIREMENTS

In our discussion of virtual-memory concepts, we have tacitly assumed that only one large program is being executed. If all of the program does not fit into the available physical memory, parts of it (pages) are moved from the disk into the main memory when they are to be executed. Although we have alluded to software routines that are needed to manage this movement of program segments, we have not been specific about the details. Memory management routines are part of the operating system of the computer. It is convenient to assemble the operating system routines into a virtual address space, called the system space, that is separate from the virtual space in which user application programs reside. The latter space is called the user space. In fact, there may be a number of user spaces, one for each user. This is arranged by providing a separate page table for each user program. The MMU uses a page table base register to determine the address of the table to be used in the translation process. Hence, by changing the contents of this register, the operating system can switch from one space to another. The physical main memory is thus shared by the active pages of the system space and several user spaces. However, only the pages that belong to one of these spaces are accessible at any given time.

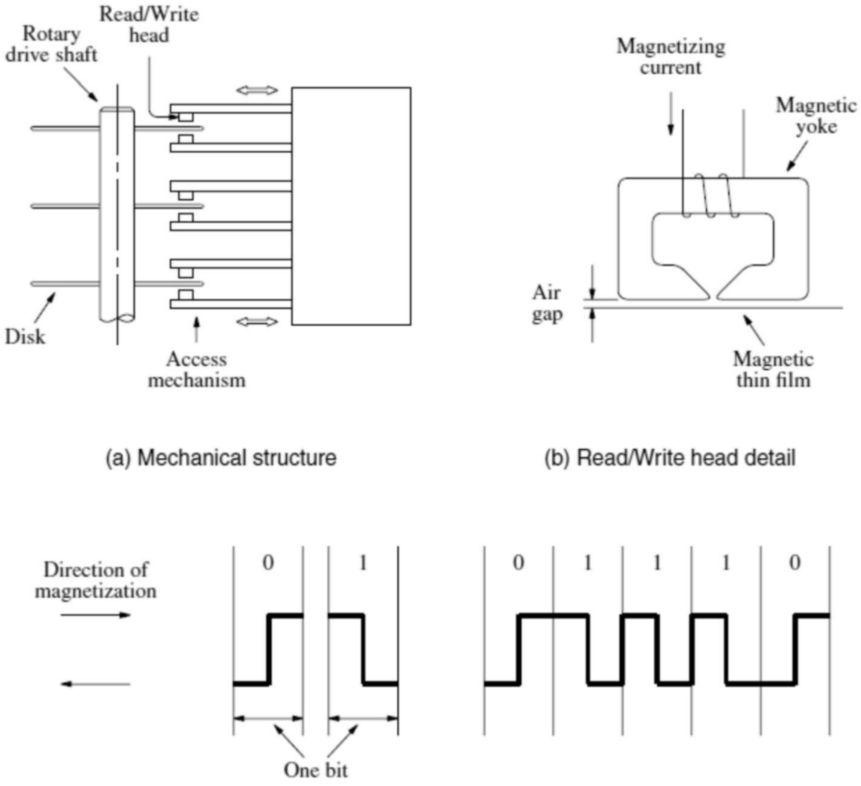
In any computer system in which independent user programs coexist in the main memory, the notion of protection must be addressed. No program should be allowed to destroy either the data or instructions of other programs in the memory. The needed protection can be provided in several ways. Let us first consider the most basic form of protection. Most processors can operate in one of two modes, the supervisor mode and the user mode. The processor is usually placed in the supervisor mode when operating system routines are being executed and in the user mode to execute user programs. In the user mode, some machine instructions cannot be executed. These are privileged instructions. They include instructions that modify the page table base register, which can only be executed while the processor is in the supervisor mode. Since a user program is executed in the user mode, it is prevented from accessing the page tables of other users or of the system space. It is sometimes desirable for one application program to have access to certain pages belonging to another program. The operating system can arrange this by causing these pages to appear in both spaces. The shared pages will therefore have entries in two different page tables. The control bits in each table entry can be set to control the access privileges granted to each program. For example, one program may be allowed to read and write a given page, while the other program may be given only read access.

ASSOCIATIVE MEMORIES & SECONDARY STORAGE DEVICES

The semiconductor memories discussed in the previous sections cannot be used to provide all of the storage capability needed in computers. Their main limitation is the cost per bit of stored information. The large storage requirements of most computer systems are economically realized in the form of magnetic and optical disks, which are usually referred to as secondary storage devices.

Magnetic Hard Disks

The storage medium in a magnetic-disk system consists of one or more disk platters mounted on a common spindle. A thin magnetic film is deposited on each platter, usually on both sides. The assembly is placed in a drive that causes it to rotate at a constant speed. The magnetized surfaces move in close proximity to read/write heads, as shown in Figure a. Data are stored on concentric tracks, and the read/write heads move radially to access different tracks.



(a) Mechanical structure

(b) Read/Write head detail

(c) Bit representation by phase encoding

Each read/write head consists of a magnetic yoke and a magnetizing coil, as indicated in Figure b. Digital information can be stored on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area immediately underneath the head to switch to a direction parallel to the applied field. The same head can be used for reading the stored information. In this case, changes in the magnetic field in the vicinity of the head caused by the movement of the film relative to the yoke induce a voltage in the coil, which now serves as a sense coil. The polarity of this voltage is monitored by the control circuitry to determine the state of magnetization of the film. Only changes in the magnetic field under the head can be sensed during the Read operation. Therefore, if the binary states 0 and 1 are represented by two opposite states of magnetization, a voltage is induced in the head only at 0-to-1 and at 1-to-0 transitions in the bit stream. A long string of 0s or 1s causes an induced voltage only at the beginning and end of the string. Therefore, to determine the number of consecutive 0s or 1s stored, a clock must provide information for synchronization.

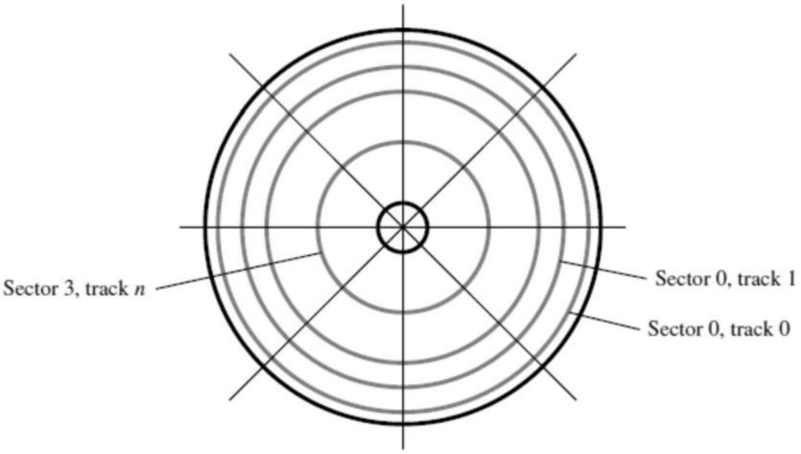
In some early designs, a clock was stored on a separate track, on which a change in magnetization is forced for each bit period. Using the clock signal as a reference, the data stored on other tracks can be read correctly. The modern approach is to combine the clocking information with the data. Several different techniques have been developed for such encoding. One simple scheme, depicted in Figure c, is known as phase encoding or Manchester encoding. In this scheme, changes in magnetization occur for each data bit, as shown in the figure. Clocking information is provided by the change in magnetization at the midpoint of each bit period. The drawback of Manchester encoding is its poor bit-storage density. The space required to represent each bit must be large enough to accommodate two changes in magnetization. We use the Manchester encoding example to illustrate how a self-clocking scheme may be implemented, because it is easy to understand. Other, more compact codes have been developed. They are much more efficient and provide better storage density. They also require more complex control circuitry. The discussion of such codes is beyond the scope of this book.

Read/write heads must be maintained at a very small distance from the moving disk surfaces in order to achieve high bit densities and reliable Read and Write operations. When the disks are moving at their steady rate, air pressure develops between the disk surface and the head and forces the head away from the surface. This force is counterbalanced by a spring- loaded mounting arrangement that presses the head toward the surface. The flexible spring connection between the head and its arm mounting permits the head to fly at the desired distance away from the surface in spite of any small variations in the flatness of the surface. In most modern disk units, the disks and the read/write heads are placed in a sealed, air-filtered enclosure. This approach is known as Winchester technology. In such units, the read/write heads can operate closer to the magnetized track surfaces, because dust particles, which are a problem in unsealed assemblies, are absent. The closer the heads are to a track surface, the more densely the data can be packed along the track, and the closer the tracks can be to each other. Thus, Winchester disks have a larger capacity for a given physical size compared to unsealed units. Another advantage of Winchester technology is that data integrity tends to be greater in sealed units, where the storage medium is not exposed to contaminating elements.

The read/write heads of a disk system are movable. There is one head per surface. All heads are mounted on a comb-like arm that can move radially across the stack of disks to provide access to individual tracks, as shown in Figure a. To read or write data on a given track, the read/write heads must first be positioned over that track. The disk system consists of three key parts. One part is the assembly of disk platters, which is usually referred to as the disk. The second part comprises the electromechanical mechanism that spins the disk and moves the read/write heads; it is called the disk drive. The third part is the disk controller, which is the electronic circuitry that controls the operation of the system. The disk controller may be implemented as a separate module, or it may be incorporated into the enclosure that contains the entire disk system. We should note that the term disk is often used to refer to the combined package of the disk drive and the disk it contains. We will do so in the sections that follow, when there is no ambiguity in the meaning of the term.

Organization and Accessing of Data on a Disk

The organization of data on a disk is illustrated in Figure. Each surface is divided into concentric tracks, and each track is divided into sectors. The set of corresponding tracks on all surfaces of a stack of disks forms a logical cylinder. All tracks of a cylinder can be accessed without moving the read/write heads. Data are accessed by specifying the surface number, the track number, and the sector number. Read and Write operations always start at sector boundaries.



Data bits are stored serially on each track. Each sector may contain 512 or more bytes. The data are preceded by a sector header that contains identification (addressing) information used to find the desired sector on the selected track. Following the data, there are additional bits that constitute an error-correcting code (ECC). The ECC bits are used to detect and correct errors that may have occurred in writing or reading the data bytes. There is a small inter-sector gap that enables the disk control circuitry to distinguish easily between two consecutive sectors.

An unformatted disk has no information on its tracks. The formatting process writes markers that divide the disk into tracks and sectors. During this process, the disk controller may discover some sectors or even whole tracks that are defective. The disk controller keeps a record of such defects and excludes them from use. The formatting information comprises sector headers, ECC bits, and inter-sector gaps. The capacity of a formatted disk, after accounting for the formatting information overhead, is the proper indicator of the disk's storage capability. After formatting, the disk is divided into logical partitions. Figure indicates that each track has the same number of sectors, which means that all tracks have the same storage capacity. In this case, the stored information is packed more densely on inner tracks than on outer tracks. It is also possible to increase the storage density by placing more sectors on the outer tracks, which have longer circumferences. This would be at the expense of more complicated access circuitry.

Access Time

There are two components involved in the time delay between the disk receiving an address and the beginning of the actual data transfer. The first, called the seek time, is the time required to move the read/write head to the proper track. This time depends on the initial position of the head relative to the track specified in the address. Average values are in the 5- to 8-ms range. The second component is the rotational delay, also called latency time, which is the time taken to reach the addressed sector after the read/write head is positioned over the correct track. On average, this is the time for half a rotation of the disk. The sum of these two delays is called the disk access time. If only a few sectors of data are accessed in a single operation, the access time is at least an order of magnitude longer than the time it takes to transfer the data.

Data Buffer/Cache

A disk drive is connected to the rest of a computer system using some standard interconnection scheme, such as SCSI or SATA. The interconnection hardware is usually capable of transferring data at much higher rates than the rate at which data can be read from disk tracks. An efficient way to deal with the possible differences in transfer rates is to include a data buffer in the disk unit. The buffer is a semiconductor memory, capable of storing a few megabytes of data. The requested data are transferred between the disk tracks and the buffer at a rate dependent on the rotational speed of the disk. Transfers between the data buffer and the main memory can then take place at the maximum rate allowed by the interconnect between them.

The data buffer in the disk controller can also be used to provide a caching mechanism for the disk. When a Read request arrives at the disk, the controller can first check to see if the desired data are already available in the buffer. If so, the data are transferred to the memory in microseconds instead of milliseconds. Otherwise, the data are read from a disk track in the usual way, stored in the buffer, then transferred to the memory. Because of locality of reference, a subsequent request is likely to refer to data that sequentially follow the data specified in the current request. In anticipation of future requests, the disk controller may read more data than needed and place them into the buffer. When used as a cache, the buffer is typically large enough to store entire tracks of data. So, a possible strategy is to begin transferring the contents of the track into the data buffer as soon as the read/write head is positioned over the desired track.

Disk Controller

Operation of a disk drive is controlled by a disk controller circuit, which also provides an interface between the disk drive and the rest of the computer system. One disk controller may be used to control more than one drive. A disk controller that communicates directly with the processor contains a number of registers that can be read and written by the operating system. Thus, communication between the OS and the disk controller is achieved in the same manner as with any I/O interface. The disk controller uses the DMA scheme to transfer data between the disk and the main memory. Actually, these transfers are from/to the data buffer, which is implemented as a part of the disk controller module. The OS initiates the transfers by issuing Read and Write requests, which entail loading the controller’s registers with the necessary addressing and control information. Typically, this information includes:

Main memory address—The address of the first main memory location of the block of words involved in the transfer.

Disk address—The location of the sector containing the beginning of the desired block of words.

Word count—The number of words in the block to be transferred. The disk address issued by the OS is a logical address. The corresponding physical address on the disk may be different. For example, bad sectors may be detected when the disk is formatted. The disk controller keeps track of such sectors and maintains the mapping between logical and physical addresses. Normally, a few spare sectors are kept on each track, or on another track in the same cylinder, to be used as substitutes for the bad sectors. On the disk drive side, the controller's major functions are:

Seek—Causes the disk drive to move the read/write head from its current position to the desired track.

Read—Initiates a Read operation, starting at the address specified in the disk address register. Data read serially from the disk are assembled into words and placed into the data buffer for transfer to the main memory. The number of words is determined by the word count register.

Write—Transfers data to the disk, using a control method similar to that for Read operations.

Error checking—Computes the error correcting code (ECC) value for the data read from a given sector and compares it with the corresponding ECC value read from the disk. In the case of a mismatch, it corrects the error if possible; otherwise, it raises an interrupt to inform the OS that an error has occurred. During a Write operation, the controller computes the ECC value for the data to be written and stores this value on the disk.

Floppy Disks

The disks discussed above are known as hard or rigid disk units. *Floppy disks* are smaller, simpler, and cheaper disk units that consist of a flexible, removable, plastic *diskette* coated with magnetic material. The diskette is enclosed in a plastic jacket, which has an opening where the read/write head can be positioned. A hole in the centre of the diskette allows a spindle mechanism in the disk drive to position and rotate the diskette. The main feature of floppy disks is their low cost and shipping convenience. However, they have much smaller storage capacities, longer access times, and higher failure rates than hard disks. In recent years, they have largely been replaced by CDs, DVDs, and flash cards as portable storage media.

RAID Disk Arrays

Processor speeds have increased dramatically. At the same time, access times to disk drives are still on the order of milliseconds, because of the limitations of the mechanical motion involved. One way to reduce access time is to use multiple disks operating in parallel. They called it RAID, for Redundant Array of Inexpensive Disks. (Since all disks are now inexpensive, the acronym was later reinterpreted as Redundant Array of Independent Disks.) Using multiple disks also makes it possible to improve the reliability of the overall system. Different configurations were proposed, and many more have been developed since.

The basic configuration, known as RAID 0, is simple. A single large file is stored in several separate disk units by dividing the file into a number of smaller pieces and storing these pieces on different disks. This is called data striping. When the file is accessed for a Read operation, all disks access their portions of the data in parallel. As a result, the rate at which the data can be transferred is equal to the data rate of individual disks times the number of disks. However, access time, that is, the seek and rotational delay needed to locate the beginning of the data on each disk, is not reduced. Since each disk operates independently, access times vary. Individual pieces of the data are buffered, so that the complete file can be reassembled and transferred to the memory as a single entity.

Various RAID configurations form a hierarchy, with each level in the hierarchy providing additional features. For example, RAID 1 is intended to provide better reliability by storing identical copies of the data on two disks rather than just one. The two disks are said to be mirrors of each other. If one disk drive fails, all Read and Write operations are directed to its mirror drive. Other levels of the hierarchy achieve increased reliability through various parity-checking schemes, without requiring a full duplication of disks. Some also have error- recovery capability. The RAID concept has gained commercial acceptance. RAID systems are available from many manufacturers for use with a variety of operating systems.

Optical Disks

Storage devices can also be implemented using optical means. The familiar compact disk (CD), used in audio systems, was the first practical application of this technology. Soon after, the optical technology was adapted to the computer environment to provide a high capacity read-only storage medium known as a CD-ROM.

The first generation of CDs was developed in the mid-1980s by the Sony and Philips companies. The technology exploited the possibility of using a digital representation for analog sound signals. To provide high-quality sound recording and reproduction, 16-bit samples of the analog signal are taken at a rate of 44,100 samples per second. Initially, CDs were designed to hold up to 75 minutes, requiring a total of about 3 × 109 bits (3 gigabits) of storage. Since then, higher-capacity devices have been developed.

CD Technology

The optical technology that is used for CD systems makes use of the fact that laser light can be focused on a very small spot. A laser beam is directed onto a spinning disk, with tiny indentations arranged to form a long spiral track on its surface. The indentations reflect the focused beam toward a photodetector, which detects the stored binary patterns. The laser emits a coherent light beam that is sharply focused on the surface of the disk. Coherent light consists of synchronized waves that have the same wavelength. If a coherent light beam is combined with another beam of the same kind, and the two beams are in phase, the result is brighter beam. But, if the waves of the two beams are 180 degrees out of phase, they cancel each other. Thus, a photo detector can be used to detect the beams. It will see a bright spot in the first case and a dark spot in the second case. Across-section of a small portion of a CD is shown in Figure a. The bottom layer is made of transparent polycarbonate plastic, which serves as a clear glass base. The surface of this plastic is programmed to store data by identifying it with pits. The unintended parts are called lands. A thin layer of reflecting aluminium material is placed on top of a programmed disk. The aluminium is then covered by protective acrylic. Finally, the topmost layer is deposited and stamped with a label. The total thickness of the disk is 1.2 mm, almost all of it contributed by the polycarbonate plastic. The other layers are very thin. The laser source and the photodetector are positioned below the polycarbonate plastic. The emitted beam travels through the plastic layer, reflects off the aluminium layer, and travels back toward the photodetector. Note that from the laser side, the pits actually appear as bumps rising above the lands.

CD-Rewritable

The most flexible CDs are those that can be written multiple times by the user. They are known as CD-RWs (CD-ReWritables). The basic structure of CD-RWs is similar to the structure of CD-Rs. Instead of using an organic dye in the recording layer, an alloy of silver, indium, antimony, and tellurium is used. This alloy has interesting and useful behavior when it is heated and cooled. If it is heated above its melting point (500 degrees C) and then cooled down, it goes into an amorphous state in which it absorbs light. But, if it is heated only to about 200 degrees C and this temperature is maintained for an extended period, a process known as annealing takes place, which leaves the alloy in a crystalline state that allows light to pass through. If the crystalline state represents land area, pits can be created by heating selected spots past the melting point. The stored data can be erased using the annealing process, which returns the alloy to a uniform crystalline state. A reflective material is placed above the recording layer to reflect the light when the disk is read.

DVD Technology

The success of CD technology and the continuing quest for greater storage capability has led to the development of DVD (Digital Versatile Disk) technology. The first DVD standard was defined in 1996 by a consortium of companies, with the objective of being able to store a full-length movie on one side of a DVD disk. The physical size of a DVD disk is the same as that of CDs. The disk is 1.2 mm thick, and it is 120 mm in diameter. Its storage capacity is made much larger than that of CDs by several design changes:

- A red-light laser with a wavelength of 635 nm is used instead of the infrared light laser used in CDs, which has a wavelength of 780 nm. The shorter wavelength makes it possible to focus the light to a smaller spot.
- Pits are smaller, having a minimum length of 0.4 micron.
- Tracks are placed closer together; the distance between tracks is 0.74 micron.

Using these improvements leads to DVD capacity of 4.7 G bytes.

Magnetic Tape Systems

Magnetic tapes are suited for off-line storage of large amounts of data. They are typically used for backup purposes and for archival storage. Magnetic-tape recording uses the same principle as magnetic disks. The main difference is that the magnetic film is deposited on a very thin 0.5- or 0.25-inch-wide plastic tape. Seven or nine bits (corresponding to one character) are recorded in parallel across the width of the tape, perpendicular to the direction of motion. A separate read/write head is provided for each bit position on the tape, so that all bits of a character can be read or written in parallel. One of the character bits is used as a parity bit. Data on the tape are organized in the form of records separated by gaps. Tape motion is stopped only when a record gap is underneath the read/write heads. The record gaps are long enough to allow the tape to attain its normal speed before the beginning of the next record is reached. If a coding scheme such as that is used for recording data on the tape, record gaps are identified as areas where there is no change in magnetization. This allows record gaps to be detected independently of the recorded data. To help users organize large amounts of data, a group of related records is called a file. The beginning of a file is identified by a file mark. The file mark is a special single- or multiple-character record, usually preceded by a gap longer than the inter-record gap. The first record following a file mark can be used as a header or identifier for the file. This allows the user to search for a tape containing a large number of files for a particular file.

Introduction

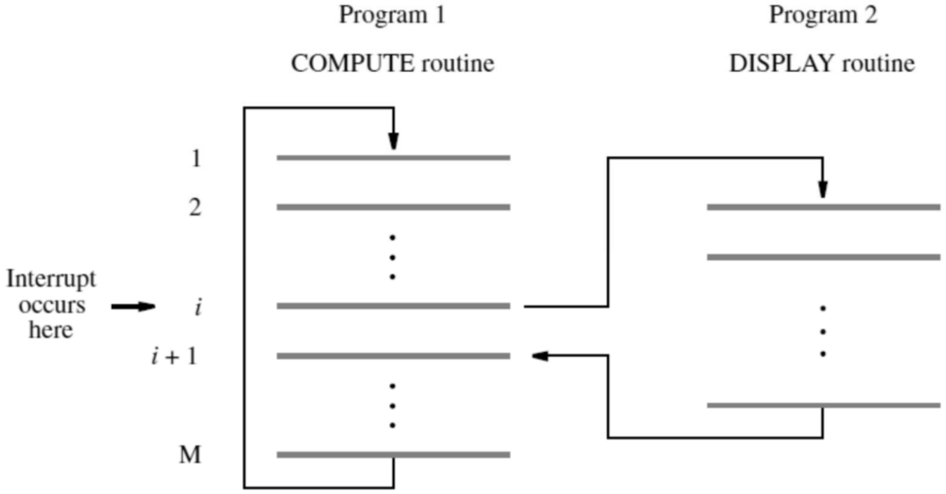
One of the basic features of a computer is its ability to exchange data with other devices. This communication capability enables a human operator, for example, to use a keyboard and a display screen to process text and graphics. We make extensive use of computers to communicate with other computers over the Internet and access information around the globe. In other applications, computers are less visible but equally important. They are an integral part of home appliances, manufacturing equipment, transportation systems, banking, and point-of-sale terminals. In such applications, input to a computer may come from a sensor switch, a digital camera, a microphone, or a fire alarm. Output may be a sound signal sent to a speaker, or a digitally coded command that changes the speed of a motor, opens a valve, or causes a robot to move in a specified manner. In short, computers should have the ability to exchange digital and analog information with a wide range of devices in many different environments.

ACCESSING I/O DEVICES & PROGRAMMED INPUT/OUTPUT

Unit 1/ Last topic presented these heading details as programmer's view of input/output data transfers that take place between the processor and the registers in I/O device interfaces.

INTERRUPTS

In the programmed I/O transfer, the program enters a wait loop in which it repeatedly tests the device status. During this period, the processor is not performing any useful computation. There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the processor when it becomes ready. It can do so by sending a hardware signal called an interrupt request to the processor. Since the processor is no longer required to continuously poll the status of I/O devices, it can use the waiting period to perform other useful tasks. Indeed, by using interrupts, such waiting periods can ideally be eliminated. The routine executed in response to an interrupt request is called the interrupt-service routine, which is the DISPLAY routine in our example. Interrupts bear considerable resemblance to subroutine calls.



Assume that an interrupt request arrives during execution of instruction *i* in Figure. The processor first completes execution of instruction i. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor returns to instruction i + 1. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction i + 1, must be put in temporary storage in a known location. A Return-from-interrupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction i + 1. The return address must be saved either in a designated general-purpose register or on the processor stack.

We should note that as part of handling interrupts, the processor must inform the device that its request has been recognized so that it may remove its interrupt-request signal. This can be accomplished by means of a special control signal, called interrupt acknowledge, which is sent to the device through the interconnection network. An alternative is to have the transfer of data between the processor and the I/O device interface accomplish the same purpose. The execution of an instruction in the interrupt-service routine that accesses the status or data register in the device interface implicitly informs the device that its interrupt request has been recognized.

So far, treatment of an interrupt-service routine is very similar to that of a subroutine. An important departure from this similarity should be noted. A subroutine performs a function required by the program from which it is called. As such, potential changes to status information and contents of registers are anticipated. However, an interrupt-service routine may not have any relation to the portion of the program being executed at the time the interrupt request is received. Therefore, before starting execution of the interrupt service routine, status information and contents of processor registers that may be altered in unanticipated ways during the execution of that routine must be saved. This saved information must be restored before execution of the interrupted program is resumed. In this way, the original program can continue execution without being affected in any way by the interruption, except for the time delay.

The task of saving and restoring information can be done automatically by the processor or by program instructions. Most modern processors save only the minimum amount of information needed to maintain the integrity of program execution. This is because the process of saving and restoring registers involves memory transfers that increase the total execution time, and hence represent execution overhead. Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt- service routine. This delay is called interrupt latency. In some applications, a long interrupt latency is unacceptable. For these reasons, the amount of information saved automatically by the processor when an interrupt request is accepted should be kept to a minimum. Typically, the processor saves only the contents of the program counter and the processor status register. Any additional information that needs to be saved must be saved by explicit instructions at the beginning of the interrupt-service routine and restored at the end of the routine. In some earlier processors, particularly those with a small number of registers, all registers are saved automatically by the processor hardware at the time an interrupt request is accepted. The data saved are restored to their respective registers as part of the execution of the Return-from- interrupt instruction.

Some computers provide two types of interrupts. One saves all register contents, and the other does not. A particular I/O device may use either type, depending upon its response time requirements. Another interesting approach is to provide duplicate sets of processor registers. In this case, a different set of registers can be used by the interrupt-service routine, thus eliminating the need to save and restore registers. The duplicate registers are sometimes called the shadow registers. An interrupt is more than a simple mechanism for coordinating I/O transfers. In a general sense, interrupts enable transfer of control from one program to another to be initiated by an event external to the computer. Execution of the interrupted program resumes after the execution of the interrupt-service routine has been completed. The concept of interrupts is used in operating systems and in many control applications where processing of certain routines must be accurately timed relative to external events. The latter type of application is referred to as real-time processing.

Enabling and Disabling Interrupts

The facilities provided in a computer must give the programmer complete control over the events that take place during program execution. The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and start the execution of another. Because interrupts can arrive at any time, they may alter the sequence of events from that envisaged by the programmer. Hence, the interruption of program execution must be carefully controlled. A fundamental facility found in all computers is the ability to enable and disable such interruptions as desired.

There are many situations in which the processor should ignore interrupt requests. For instance, the timer circuit should raise interrupt requests only when the COMPUTE routine is being executed. It should be prevented from doing so when some other task is being performed. In another case, it may be necessary to guarantee that a particular sequence of instructions is executed to the end without interruption because the interrupt-service routine may change some of the data used by the instructions in question. For these reasons, some means for enabling and disabling interruptions must be available to the programmer.

It is convenient to be able to enable and disable interrupts at both the processor and I/O device ends. The processor can either accept or ignore interrupt requests. An I/O device can either be allowed to raise interrupt requests or prevent them from doing so. A commonly used mechanism to achieve this is to use some control bits in registers that can be accessed by program instructions.

The processor has a status register (PS), which contains information about its current state of operation. Let one bit, IE, of this register be assigned for enabling/disabling interrupts. Then, the programmer can set or clear IE to cause the desired action. When IE = 1, interrupt requests from I/O devices are accepted and serviced by the processor. When IE = 0, the processor simply ignores all interrupt requests from I/O devices. The interface of an I/O device includes a control register that contains the information that governs the mode of operation of the device. One bit in this register may be dedicated to interrupt control. The I/O device is allowed to raise interrupt requests only when this bit is set to 1.

Let us now consider the specific case of a single interrupt request from one device. When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the processor has accepted its request. This means that the interrupt-request signal will be active during execution of the interrupt-service routine, perhaps until an instruction is reached that accesses the device in question. It is essential to ensure that this active request signal does not lead to successive interruptions, causing the system to enter an infinite loop from which it cannot recover. A good choice is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. The processor saves the contents of the program counter and the processor status register. After saving the contents of the PS register, with the IE bit equal to 1, the processor clears the IE bit in the PS register, thus disabling further interrupts. Then, it begins execution of the interrupt-service routine. When a Return-from-interrupt instruction is executed, the saved contents of the PS register are restored, setting the IE bit back to 1. Hence, interruptions are again enabled.

Before proceeding to study more complex aspects of interruptions, let us summarize the sequence of events involved in handling an interrupt request from a single device. Assuming that interrupts are enabled in both the processor and the device, the following is a typical scenario:

1. The device raises an interrupt request.

2. The processor interrupts the program currently being executed and saves the contents of the PC and PS registers.

3. Interrupts are disabled by clearing the IE bit in the PS to 0.

4. The action requested by the interrupt is performed by the interrupt-service routine, during which time the device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.

5. Upon completion of the interrupt-service routine, the saved contents of the PC and PS registers are restored (enabling interrupts by setting the IE bit to 1), and execution of the interrupted program is resumed.

Handling Multiple Devices

Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time. This gives rise to a number of questions:

1. How can the processor determine which device is requesting an interrupt?

2. Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case?

3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?

4. How should two or more simultaneous interrupt requests be handled?

The means by which these issues are handled vary from one computer to another, and the approach taken is an important consideration in determining the computer's suitability for a given application. When an interrupt request is received it is necessary to identify the particular device that raised the request. Furthermore, if two devices raise interrupt requests at the same time, it must be possible to break the tie and select one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced.

The information needed to determine whether a device is requesting an interrupt is available in its status register. When the device raises an interrupt request, it sets to 1 a bit in its status register, which we will call the IRQ bit. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all I/O devices in the system. The first device encountered with its IRQ bit set to 1 is the device that should be serviced. An appropriate subroutine is then called to provide the requested service. The polling scheme is easy to implement. Its main disadvantage is the time spent interrogating the IRQ bits of devices that may not be requesting any service. An alternative approach is to use vectored interrupts, which we describe next.

Vectored Interrupts

To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine. The term vectored interrupts refers to interrupt- handling schemes based on this approach.

A device requesting an interrupt can identify itself if it has its own interrupt-request signal, or if it can send a special code to the processor through the interconnection network. The processor's circuits determine the memory address of the required interrupt-service routine. A commonly used scheme is to allocate permanently an area in the memory to hold the addresses of interrupt-service routines. These addresses are usually referred to as interrupt vectors, and they are said to constitute the interrupt-vector table. For example, 128 bytes may be allocated to hold a table of 32 interrupt vectors. Typically, the interrupt vector table is in the lowest-address range. The interrupt-service routines may be located anywhere in the memory. When an interrupt request arrives, the information provided by the requesting device is used as a pointer into the interrupt-vector table, and the address in the corresponding interrupt vector is automatically loaded into the program counter.

Interrupt Nesting

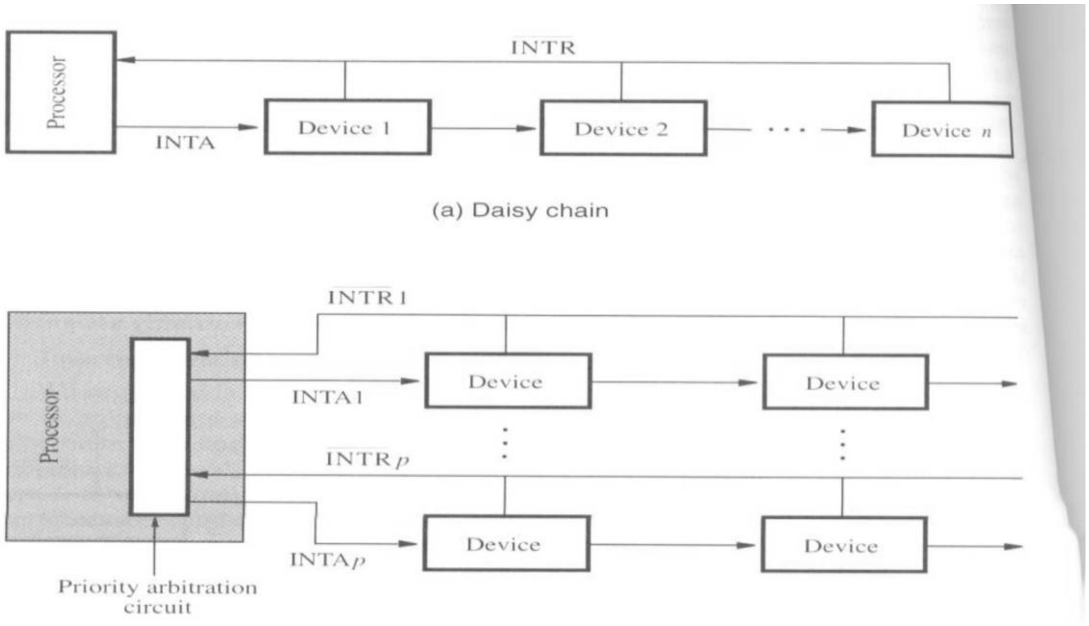
The interrupts should be disabled during the execution of an interrupt-service routine, to ensure that a request from one device will not cause more than one interruption. The same arrangement is often used when several devices are involved, in which case execution of a given interrupt-service routine, once started, always continues to completion before the processor accepts an interruption request from a second device. Interrupt-service routines are typically short, and the delay they may cause is acceptable for most simple devices.

For some devices, however, a long delay in responding to an interrupt request may lead to erroneous operation. Consider, for example, a computer that keeps track of the time of day using a real-time clock. This is a device that sends interrupt requests to the processor at regular intervals. For each of these requests, the processor executes a short interrupt-service routine to increment a set of counters in the memory that keep track of time in seconds, minutes, and so on. Proper operation requires that the delay in responding to an interrupt request from the real- time clock be small in comparison with the interval between two successive requests. To ensure that this requirement is satisfied in the presence of other interrupting devices, it may be necessary to accept an interrupt request from the clock during the execution of an interrupt- service routine for another device, i.e., to nest interrupts. This example suggests that I/O devices should be organized in a priority structure. An interrupt request from a high-priority device should be accepted while the processor is servicing a request from a lower-priority device.

A multiple-level priority organization means that during execution of an interrupt service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, we can assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts interrupts only from devices that have priorities higher than its own. At the time that execution of an interrupt-service routine for some device is started, the priority of the processor is raised to that of the device either automatically or with special instructions. This action disables interrupts from devices that have the same or lower level of priority. However, interrupt requests from higher-priority devices will continue to be accepted. The processor's priority can be encoded in a few bits of the processor status register. While this scheme is used in some processors, we will use a simpler scheme in later examples. Finally, we should point out that if nested interrupts are allowed, then each interrupt service routine must save on the stack the saved contents of the program counter and the status register. This has to be done before the interrupt- service routine enables nesting by setting the IE bit in the status register to 1.

Simultaneous Requests

We also need to consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which request to service first. Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. This is done in hardware, by using arbitration circuits.



(a) Daisy chain

(b) Arrangement of priority groups

Exceptions

An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin. So far, we have dealt only with interrupts caused by events associated with I/O data transfers. However, the interrupt mechanism is used in a number of other situations. The term exception is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception. We now describe a few other kinds of exceptions.

Recovery from Errors

Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include an error-checking code in the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the processor by raising an interrupt. The processor may also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program. For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero.

When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an exception-service routine, which takes appropriate action to recover from the error, if possible, or to inform the user about it. Recall that in the case of an

I/O interrupt, we assumed that the processor completes execution of the instruction in progress before accepting the interrupt. However, when an interrupt is caused by an error associated with the current instruction, that instruction cannot usually be completed, and the processor begins exception processing immediately.

Debugging

Another important type of exception is used as an aid in debugging programs. System software usually includes a program called a debugger, which helps the programmer find errors in a program. The debugger uses exceptions to provide two important facilities: trace mode and breakpoints.

Trace Mode

When the processor is in trace mode , an exception occurs after execution of every instance using the debugging program as the exception service routine. The debugging program examine the contents of registers, memory location etc. On return from the debugging program the next instance in the program being debugged is executed The trace exception is disabled during the execution of the debugging program.

Break point

Here the program being debugged is interrupted only at specific points selected by the user. An instance called the Trap (or) software interrupt is usually provided for this purpose. While debugging the user may interrupt the program execution after instance ‘I’ When the program is executed and reaches that point it examine the memory and register contents.

Privileged Exception

To protect the OS of a computer from being corrupted by user program certain instance can be executed only when the processor is in supervisor mode. These are called privileged exceptions. When the processor is in user mode, it will not execute instance (ie) when the processor is in supervisor mode , it will execute instance.

Use of Exceptions in Operating Systems

The operating system (OS) software coordinates the activities within a computer. It uses exceptions to communicate with and control the execution of user programs. It uses hardware interrupts to perform I/O operations.

DIRECT MEMORY ACCESS

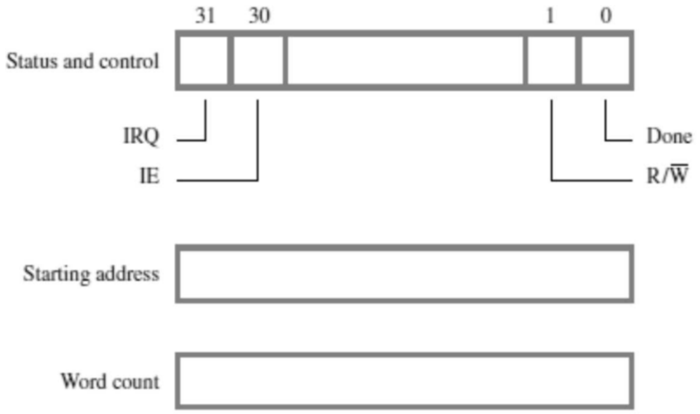
Blocks of data are often transferred between the main memory and I/O devices such as disks. This section discusses a technique for controlling such transfers without frequent, program-controlled intervention by the processor. The discussion concentrates on single-word or single-byte data transfers between the processor and I/O devices. Data are transferred from an I/O device to the memory by first reading them from the I/O device using an instruction such as

Load R2, DATAIN

which loads the data into a processor register. Then, the data read are stored into a memory location. The reverse process takes place for transferring data from the memory to an I/O device. An instruction to transfer input or output data is executed only after the processor determines that the I/O device is ready, either by polling its status register or by waiting for an interrupt request. In either case, considerable overhead is incurred, because several program instructions must be executed involving many memory accesses for each data word transferred. When transferring a block of data, instructions are needed to increment the memory address and keep track of the word count. The use of interrupts involves operating system routines which incur additional overhead to save and restore processor registers, the program counter, and other state information.

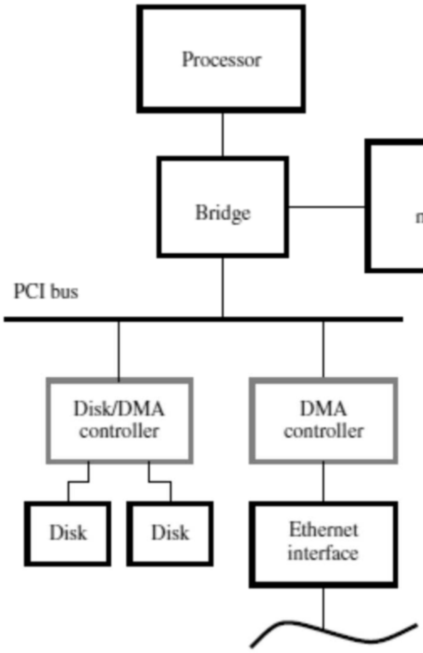
An alternative approach is used to transfer blocks of data directly between the main memory and I/O devices, such as disks. A special control unit is provided to manage the transfer, without continuous intervention by the processor. This approach is called direct memory access, or DMA. The unit that controls DMA transfers is referred to as a DMA controller. It may be part of the I/O device interface, or it may be a separate unit shared by a number of I/O devices. The DMA controller performs the functions that would normally be carried out by the processor when accessing the main memory. For each word transferred, it provides the memory address and generates all the control signals needed. It increments the memory address for successive words and keeps track of the number of transfers.

Although a DMA controller transfers data without intervention by the processor, its operation must be under the control of a program executed by the processor, usually an operating system routine. To initiate the transfer of a block of words, the processor sends to the DMA controller the starting address, the number of words in the block, and the direction of the transfer. The DMA controller then proceeds to perform the requested operation. When the entire block has been transferred, it informs the processor by raising an interrupt. Figure shows an example of the DMA controller registers that are accessed by the processor to initiate data transfer operations. Two registers are used for storing the starting address and the word count. The third register contains status and control flags. The R/W bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a Read operation, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a Write operation. Additional information is also transferred as may be required by the I/O device. For example, in the case of a disk, the processor provides the disk controller with information to identify where the data is located on the disk.



When the controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1. Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the IRQ bit to 1 when it has requested an interrupt. Figure shows how DMA controllers may be used in a computer system such as that in Figure. One DMA controller connects a high-speed Ethernet to the computer's I/O bus (a PCI bus in the case of Figure). The disk controller, which controls two disks, also has DMA capability and provides two DMA channels. It can perform two independent DMA operations, as if each disk had its own DMA controller. The registers needed to store the memory address, the word count, and so on, are duplicated, so that one set can be used with each disk.

To start a DMA transfer of a block of data from the main memory to one of the disks, an OS routine writes the address and word count information into the registers of the disk controller. The DMA controller proceeds independently to implement the specified operation. When the transfer is completed, this fact is recorded in the status and control register of the DMA channel by setting the Done bit. At the same time, if the IE bit is set, the controller sends an interrupt request to the processor and sets the IRQ bit. The status register may also be used to record other information, such as whether the transfer took place correctly or errors occurred.



Cycle Stealing

- Requests by DMA devices for using the bus are having higher priority than processor requests
- Top priority is given to high speed peripherals such as ,
  □ Disk
  □ High speed Network Interface and Graphics display device.
- Since the processor originates most memory access cycles, the DMA controller can be said to steal the memory cycles from the processor.
- This interviewing technique is called Cycle stealing.

Burst Mode

The DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as Burst/Block Mode

Bus Master

The device that is allowed to initiate data transfers on the bus at any given time is called the bus master.

Bus Arbitration

It is the process by which the next device to become the bus master is selected and the bus mastership is transferred to it.

Types:

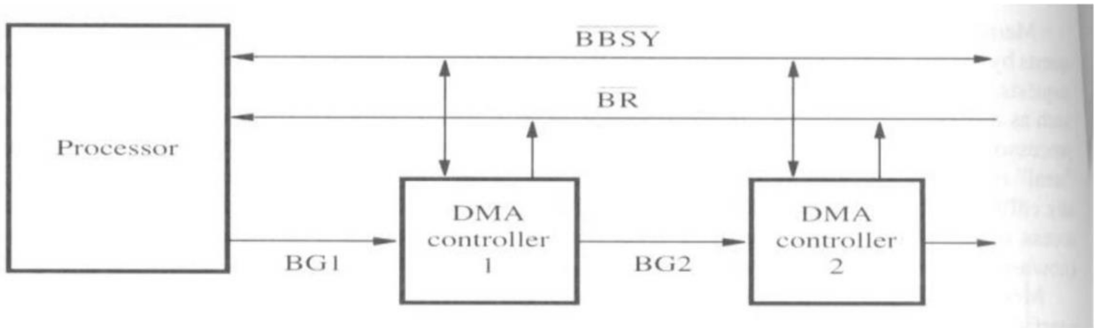
There are 2 approaches to bus arbitration. They are,

□ Centralized arbitration (A single bus arbiter performs arbitration)

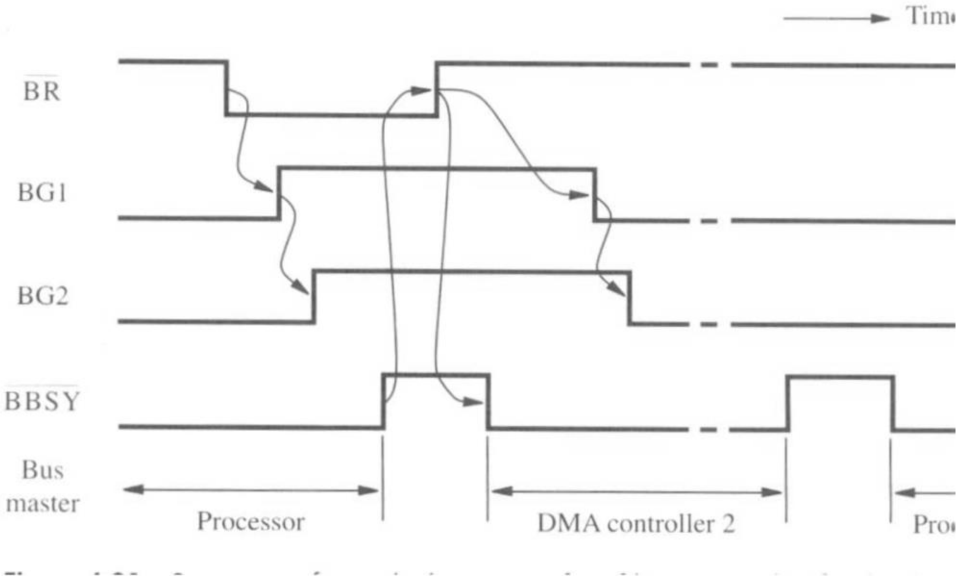
□ Distributed arbitration (all devices participate in the selection of next bus master).

Centralized Arbitration:

Here the processor is the bus master and it may grants bus mastership to one of its DMA controller. A DMA controller indicates that it needs to become the bus master by activating the Bus Request line (BR) which is an open drain line. The signal on BR is the logical OR of the bus request from all devices connected to it. When BR is activated the processor activates the Bus Grant Signal (BGI) and indicated the DMA controller that they may use the bus when it becomes free. This signal is connected to all devices using a daisy chain arrangement. If DMA requests the bus, it blocks the propagation of Grant Signal to other devices and it indicates to all devices that it is using the bus by activating open collector line, Bus Busy (BBSY).

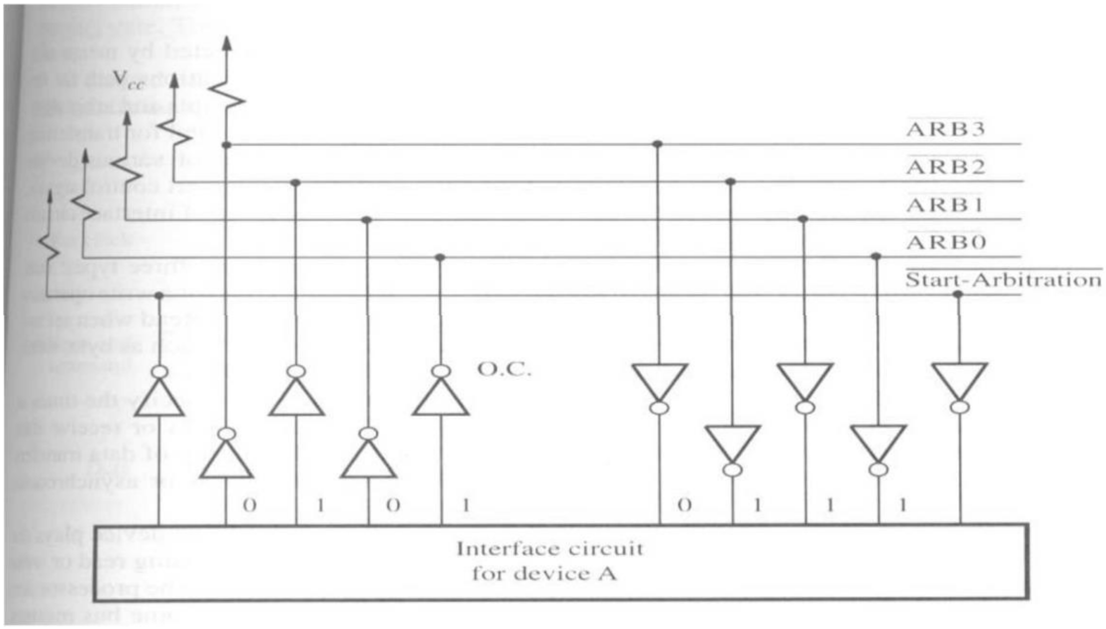


The timing diagram shows the sequence of events for the devices connected to the processor is shown. DMA controller 2 requests and acquires bus mastership and later releases the bus. During its tenure as bus master, it may perform one or more data transfer. After it releases the bus, the processor resources bus mastership



Distributed Arbitration:

It means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process. Each device on the bus is assigned a 4 bit id. When one or more devices request the bus, they assert the Start-Arbitration signal & place their 4 bit ID number on four open collector lines, ARB0 to ARB3. A winner is selected as a result of the interaction among the signals transmitted over these lines. The net outcome is that the code on the four lines represents the request that has the highest ID number. The drivers are of open collector type. Hence, if the i/p to one driver is equal to 1, the i/p to another driver connected to the same bus line is equal to „0‟(ie. bus the is in low-voltage state).



BUSES AND INTERFACE CIRCUITS

Unit 1/ First topic presented these heading details as the bus structure that implements the interconnection network used by various devices to transfer data at any one time and bus protocol, that govern how the bus is used by various devices.

INTERFACE CIRCUITS

The I/O interface of a device consists of the circuitry needed to connect that device to the bus. On one side of the interface are the bus lines for address, data, and control. On the other side are the connections needed to transfer data between the interface and the I/O device. This side is called a port, and it can be either a parallel or a serial port. A parallel port transfers multiple bits of data simultaneously to or from the device. A serial port sends and receives data one bit at a time. Communication with the processor is the same for both formats; the conversion from a parallel to a serial format and vice versa takes place inside the interface circuit.

Before we present specific circuit examples, let us recall the functions of an I/O interface. An I/O interface does the following:

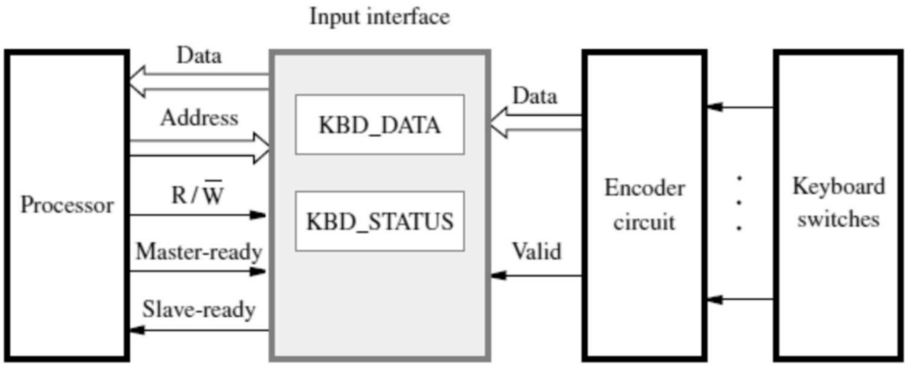
1. Provides a register for temporary storage of data
2. Includes a status register containing status information that can be accessed by the processor
3. Includes a control register that holds the information governing the behaviour of the interface
4. Contains address-decoding circuitry to determine when it is being addressed by the processor
5. Generates the required timing signals
6. Performs any format conversion that may be necessary to transfer data between the processor and the I/O device, such as parallel-to-serial conversion in the case of a serial port

Parallel Interface

We describe an interface circuit for an 8-bit input port that can be used for connecting a simple input device, such as a keyboard. Then, we describe an interface circuit for an 8-bit output port, which can be used with an output device such as a display. We assume that these interface circuits are connected to a 32-bit processor that uses memory-mapped I/O and the asynchronous bus protocol depicted.

Input Interface

Figure shows a circuit that can be used to connect a keyboard to a processor. The registers in this circuit correspond to those given in Figure. Assume that interrupts are not used, so there is no need for a control register. There are only two registers: a data register, KBD_DATA, and a status register, KBD_STATUS. The latter contains the keyboard status flag, KIN.



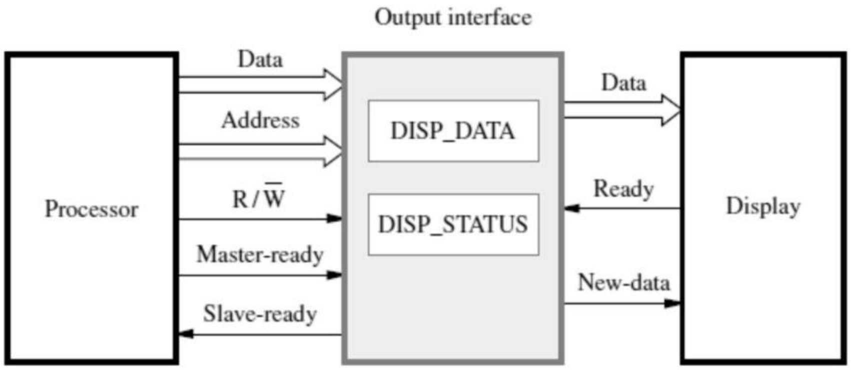
A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character. A difficulty with such mechanical pushbutton switches is that the contacts bounce when a key is pressed, resulting in the electrical connection being made then broken several times before the switch settles in the closed position. Although bouncing may last only one or two milliseconds, this is long enough for the computer to erroneously interpret a single pressing of a key as the key being pressed and released several times. The effect of bouncing can be eliminated using a simple debouncing circuit, which could be part of the keyboard hardware or may be incorporated in the encoder circuit. Alternatively, switch bouncing can be dealt with in software. The software detects that a key has been pressed when it observes that the keyboard status flag, KIN, has been set to 1. The I/O routine can then introduce sufficient delay before reading the contents of the input buffer, KBD_DATA, to ensure that bouncing has subsided. When debouncing is implemented in hardware, the I/O routine can read the input character as soon as it detects that KIN is equal to 1.

The output of the encoder in Figure consists of one byte of data representing the encoded character and one control signal called Valid. When a key is pressed, the Valid signal changes from 0 to 1, causing the ASCII code of the corresponding character to be loaded into the KBD_DATA register and the status flag KIN to be set to 1. The status flag is cleared to 0 when the processor reads the contents of the KBD_DATA register. The interface circuit is shown connected to an asynchronous bus on which transfers are controlled by the handshake signals Master-ready and Slave-ready, as in Figure. The bus has one other control line, R/W, which indicates a Read operation when equal to 1.

Output Interface

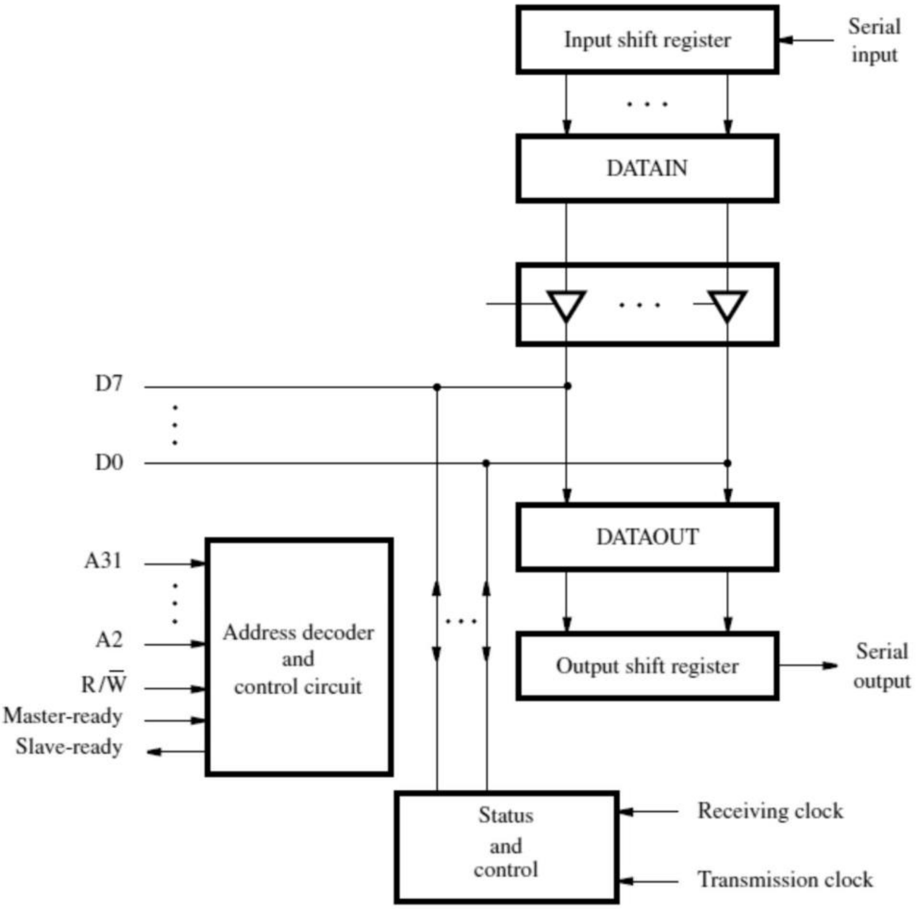
Let us now consider the output interface shown in Figure, which can be used to connect an output device such as a display. We have assumed that the display uses two handshake signals, New-data and Ready, in a manner similar to the handshake between the bus signals Master-ready and Slave-ready. When the display is ready to accept a character, it asserts its Ready signal, which causes the DOUT flag in the DISP_STATUS register to be set to 1. When the I/O routine checks DOUT and finds it equal to 1, it sends a character to DISP_DATA. This clears the DOUT flag to 0 and sets the New-data signal to 1. In response, the display returns Ready to 0 and accepts and displays the character in DISP_DATA. When it is ready to receive another character, it asserts Ready again, and the cycle repeats.

Figure shows an implementation of this interface. Its operation is similar to that of the input interface of Figure 7.11, except that it responds to both Read and Write operations. A Write operation in which A2 = 0 loads a byte of data into register DISP_DATA. A Read operation in which A2 = 1 reads the contents of the status register DISP_STATUS. In this case, only the DOUT flag, which is bit b2 of the status register, is sent by the interface. The remaining bits of DISP_STATUS are not used. The state of the status flag is determined by the handshake control circuit.



Serial Interface

A serial interface is used to connect the processor to I/O devices that transmit data one bit at a time. Data are transferred in a bit-serial fashion on the device side and in a bit-parallel fashion on the processor side. The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability. A block diagram of a typical serial interface is shown in Figure. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAOUT register are transferred to the output shift register, from which the bits are shifted out and sent to the I/O device.



The part of the interface that deals with the bus is the same as in the parallel interface described earlier. Two status flags, which we will refer to as SIN and SOUT, are maintained by the Status and control block. The SIN flag is set to 1 when new data are loaded into DATAIN from the shift register, and cleared to 0 when these data are read by the processor. The SOUT flag indicates whether the DATAOUT register is available. It is cleared to 0 when the processor writes new data into DATAOUT and set to 1 when data are transferred from DATAOUT to the output shift register. The double buffering used in the input and output paths in Figure is important. It is possible to implement DATAIN and DATAOUT themselves as shift registers, thus obviating the need for separate shift registers. However, this would impose awkward restrictions on the operation of the I/O device. After receiving one character from the serial line, the interface would not be able to start receiving the next character until the processor reads the contents of DATAIN. Thus, a pause would be needed between two characters to give the processor time to read the input data.

With double buffering, the transfer of the second character can begin as soon as the first character is loaded from the shift register into the DAT IN register. Thus, provided the processor reads the contents of DATAIN before the serial transfer of the second character is completed, the interface can receive a continuous stream of input data over the serial line. An analogous situation occurs in the output path of the interface. During serial transmission, the receiver needs to know when to shift each bit into its input shift register. Since there is no separate line to carry a clock signal from the transmitter to the receiver, the timing information needed must be embedded into the transmitted data using an encoding scheme. There are two basic approaches. The first is known as asynchronous transmission, because the receiver uses a clock that is not synchronized with the transmitter clock. In the second approach, the receiver is able to generate a clock that is synchronized with the transmitter clock. Hence it is called synchronous transmission. These approaches are described briefly below.

Asynchronous Transmission

This approach uses a technique called start-stop transmission. Data are organized in small groups of 6 to 8 bits, with a well-defined beginning and end. In a typical arrangement, alphanumeric characters encoded in 8 bits are transmitted as shown in Figure. The line connecting the transmitter and the receiver is in the 1 state when idle. A character is transmitted as a 0 bit, referred to as the Start bit, followed by 8 data bits and 1 or 2 Stop bits. The Stop bits have a logic value of 1. The 1-to-0 transition at the beginning of the Start bit alerts the receiver that data transmission is about to begin. Using its own clock, the receiver determines the position of the next 8 bits, which it loads into its input register.

The Stop bits following the transmitted character, which are equal to 1, ensure that the Start bit of the next character will be recognized. When transmission stops, the line remains in the 1 state until another character is transmitted. To ensure correct reception, the receiver needs to sample the incoming data as close to the center of each bit as possible. It does so by using a clock signal whose frequency, fR, is substantially higher than the transmission clock, fT . Typically, fR = 16fT . This means that 16 pulses of the local clock occur during each data bit interval. This clock is used to increment a modulo-16 counter, which is cleared to 0 when the leading edge of a Start bit is detected. The middle of the Start bit is reached at the count of 8. The state of the input line is sampled again at this point to confirm that it is a valid Start bit (a zero), and the counter is cleared to 0. From this point onward, the incoming data signal is sampled whenever the count reaches 16, which should be close to the middle of each incoming bit. Therefore, as long as fR/16 is sufficiently close to fT , the receiver will correctly load the bits of the incoming character.

Synchronous Transmission

In the start-stop scheme described above, the position of the 1-to-0 transition at the beginning of the start bit in Figure is the key to obtaining correct timing information. This scheme is useful only where the speed of transmission is sufficiently low and the conditions on the transmission link are such that the square waveforms shown in the figure maintain their shape. For higher speed a more reliable method is needed for the receiver to recover the timing information.

In synchronous transmission, the receiver generates a clock that is synchronized to that of the transmitter by observing successive 1-to-0 and 0-to-1 transitions in the received signal. It adjusts the position of the active edge of the clock to be in the center of the bit position. A variety of encoding schemes are used to ensure that enough signal transitions occur to enable the receiver to generate a synchronized clock and to maintain synchronization. Once synchronization is achieved, data transmission can continue indefinitely. Encoded data are usually transmitted in large blocks consisting of several hundreds or several thousands of bits. The beginning and end of each block are marked by appropriate codes, and data within a block are organized according to an agreed upon set of rules. Synchronous transmission enables very high data transfer rates

STANDARD I/O INTERFACES

A typical desktop or notebook computer has several ports that can be used to connect I/O devices, such as a mouse, a memory key, or a disk drive. Standard interfaces have been developed to enable I/O devices to use interfaces that are independent of any particular processor. For example, a memory key that has a USB connector can be used with any computer that has a USB port. In this section, we describe briefly some of the widely used interconnection standards. Most standards are developed by a collaborative effort among a number of companies. In many cases, the IEEE (Institute of Electrical and Electronics Engineers) develops these standards further and publishes them as IEEE Standards.

UNIVERSAL SERIAL BUS (USB)

The Universal Serial Bus (USB) [1] is the most widely used interconnection standard. A large variety of devices are available with a USB connector, including mice, memory keys, disk drives, printers, cameras, and many more. The commercial success of the USB is due to its simplicity and low cost. The original USB specification supports two speeds of operation, called low-speed (1.5 Megabits/s) and full-speed (12 Megabits/s). Later, USB 2, called High- Speed USB, was introduced. It enables data transfers at speeds up to 480 Megabits/s. As I/O devices continued to evolve with even higher speed requirements, USB 3 (called Superspeed) was developed. It supports data transfer rates up to 5 Gigabits/s. The USB has been designed to meet several key objectives:

- Provide a simple, low-cost, and easy to use interconnection system
- Accommodate a wide range of I/O devices and bit rates, including Internet connections, and audio and video applications
- Enhance user convenience through a "plug-and-play" mode of operation We will elaborate on some of these objectives before discussing the technical details of the USB.

Device Characteristics

The kinds of devices that may be connected to a computer cover a wide range of functionalities. The speed, volume, and timing constraints associated with data transfers to and from these devices vary significantly. In the case of a keyboard, one byte of data is generated every time a key is pressed, which may happen at any time. These data should be transferred to the computer promptly. Since the event of pressing a key is not synchronized to any other event in a computer system, the data generated by the keyboard are called asynchronous. Furthermore, the rate at which the data are generated is quite low. It is limited by the speed of the human operator to about 10 bytes per second, which is less than 100 bits per second.

A variety of simple devices that may be attached to a computer generate data of a similar nature—low speed and asynchronous. Computer mice and some of the controls and manipulators used with video games are good examples. Consider now a different source of data. Many computers have a microphone, either externally attached or built in. The sound picked up by the microphone produces an analog electrical signal, which must be converted into a digital form before it can be handled by the computer. This is accomplished by sampling the analog signal periodically. For each sample, an analog-to-digital (A/D) converter generates an n-bit number representing the magnitude of the sample. The number of bits, n, is selected based on the desired precision with which to represent each sample. Later, when these data are sent to a speaker, a digital to analog (D/A) converter is used to restore the original analog signal from the digital format.

A similar approach is used with video information from a camera. The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a data stream is called isochronous, meaning that successive events are separated by equal periods of time. A signal must be sampled quickly enough to track its highest-frequency components. In general, if the sampling rate is s samples per second, the maximum frequency component captured by the sampling process is s/2. For example, human speech can be captured adequately with a sampling rate of 8 kHz, which will record sound signals having frequencies up to 4 kHz. For higher-quality sound, as needed in a music system, higher sampling rates are used. A standard sampling rate for digital sound is 44.1 kHz. Each sample is represented by 4 bytes of data to accommodate the wide range in sound volume (dynamic range) that is necessary for high-quality sound reproduction. This yields a data rate of about 1.4 Megabits/s.

An important requirement in dealing with sampled voice or music is to maintain precise timing in the sampling and replay processes. A high degree of jitter (variability in sample timing) is unacceptable. Hence, the data transfer mechanism between a computer and a music system must maintain consistent delays from one sample to the next. Otherwise, complex buffering and retiming circuitry would be needed. On the other hand, occasional errors or missed samples can be tolerated. They either go unnoticed by the listener or they may cause an unobtrusive click. No sophisticated mechanisms are needed to ensure perfectly correct data delivery. Data transfers for images and video have similar requirements but require much higher data transfer rates. To maintain the picture quality of commercial television, an image should be represented by about 160 kilobytes and transmitted 30 times per second. Together with control information, this yields a total bit rate of 44 Megabits/s. Higher-quality images, as in HDTV (High-Definition TV), require higher rates.

Large storage devices such as magnetic and optical disks present different requirements. These devices are part of the computer’s memory hierarchy. Their connection to the computer requires a data transfer bandwidth of at least 40 or 50 Megabits/s. Delays on the order of milliseconds are introduced by the movement of the mechanical components in the disk mechanism. Hence, a small additional delay introduced while transferring data to or from the computer is not important, and jitter is not an issue. However, the transfer mechanism must guarantee data correctness.

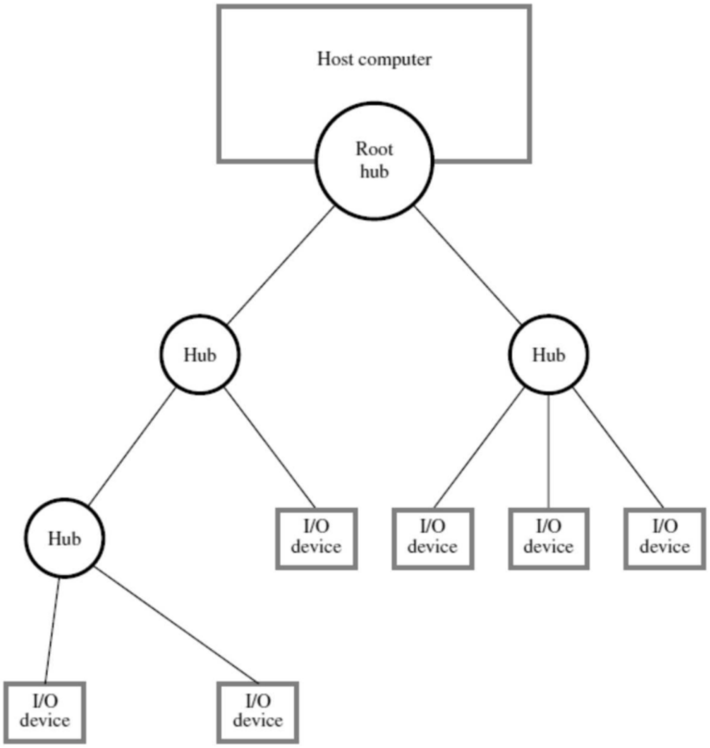
Plug-and-Play

When an I/O device is connected to a computer, the operating system needs some information about it. It needs to know what type of device it is so that it can use the appropriate device driver. It also needs to know the addresses of the registers in the device's interface to be able to communicate with it. The USB standard defines both the USB hardware and the software that communicates with it. Its plug-and-play feature means that when a new device is connected, the system detects its existence automatically. The software determines the kind of device and how to communicate with it, as well as any special requirements it might have. As a result, the user simply plugs in a USB device and begins to use it, without having to get involved in any of these details. The USB is also hot-pluggable, which means a device can be plugged into or removed from a USB port while power is turned on.

USB Architecture

The USB uses point-to-point connections and a serial transmission format. When multiple devices are connected, they are arranged in a tree structure as shown in Figure. Each node of the tree has a device called a hub, which acts as an intermediate transfer point between the host computer and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices: a mouse, a keyboard, a printer, an Internet connection, a camera, or a speaker. The tree structure makes it possible to connect many devices using simple point-to-point serial links. If I/O devices are allowed to send messages at any time, two messages may reach the hub at the same time and interfere with each other. For this reason, the USB operates strictly on the basis of polling. A device may send a message only in response to a poll message from the host processor. Hence, no two devices can send messages at the same time. This restriction allows hubs to be simple, low- cost devices.

Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the processor's address space. The root hub of the USB, which is attached to the processor, appears as a single device. The host software communicates with individual devices by sending information to the root hub, which it forwards to the appropriate device in the USB tree. When a device is first connected to a hub, or when it is powered on, it has the address 0. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected. When the host is informed that a new device has been connected, it reads the information in a special memory in the device's USB interface to learn about the device's capabilities. It then assigns the device a unique USB address and writes that address in one of the device’s interface registers. It is this initial connection procedure that gives the USB its plug-and-play capability.



Isochronous Traffic on USB

An important feature of the USB is its ability to support the transfer of isochronous data in a simple manner. As mentioned earlier, isochronous data need to be transferred at precisely timed regular intervals. To accommodate this type of traffic, the root hub transmits a uniquely recognizable sequence of bits over the USB tree every millisecond. This sequence of bits, called a Start of Frame character, acts as a marker indicating the beginning of isochronous data, which are transmitted after this character. Thus, digitized audio and video signals can be transferred in a regular and precisely timed manner.

Electrical Characteristics

USB connections consist of four wires, of which two carry power, +5 V and Ground, and two carry data. Thus, I/O devices that do not have large power requirements can be powered directly from the USB. This obviates the need for a separate power supply for simple devices such as a memory key or a mouse. wo methods are used to send data over a USB cable. When sending data at low speed, a high voltage relative to Ground is transmitted on one of the two data wires to represent a 0 and on the other to represent a 1. The Ground wire carries the return current in both cases. Such a scheme in which a signal is injected on a wire relative to ground is referred to as single-ended transmission. The speed at which data can be sent on any cable is limited by the amount of electrical noise present. The term noise refers to any signal that interferes with the desired data signal and hence could cause errors. Single-ended transmission is highly susceptible to noise. The voltage on the ground wire is common to all the devices connected to the computer. Signals sent by one device can cause small variations in the voltage on the ground wire, and hence can interfere with signals sent by another device. Interference can also be caused by one wire picking up noise from nearby wires.

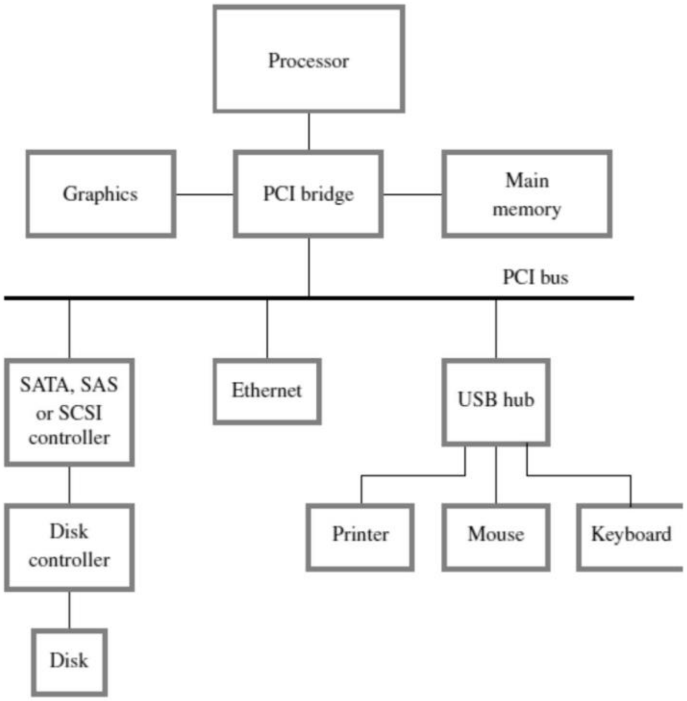
The High-Speed USB uses an alternative arrangement known as differential signalling. The data signal is injected between two data wires twisted together. The ground wire is not involved. The receiver senses the voltage difference between the two signal wires directly, without reference to ground. This arrangement is very effective in reducing the noise seen by the receiver, because any noise injected on one of the two wires of the twisted pair is also injected on the other. Since the receiver is sensitive only to the voltage difference between the two wires, the noise component is cancelled out. The ground wire acts as a shield for the data on the twisted pair against interference from nearby wires. Differential signaling allows much lower voltages and much higher speeds to be used compared to single-ended signaling.

PCI BUS

The PCI (Peripheral Component Interconnect) bus [3] was developed as a low-cost, processor-independent bus. It is housed on the motherboard of a computer and used to connect I/O interfaces for a wide variety of devices. A device connected to the PCI bus appears to the processor as if it is connected directly to the processor bus. Its interface registers are assigned addresses in the address space of the processor. We will start by describing how the PCI bus operates, then discuss some of its features.

Bus Structure

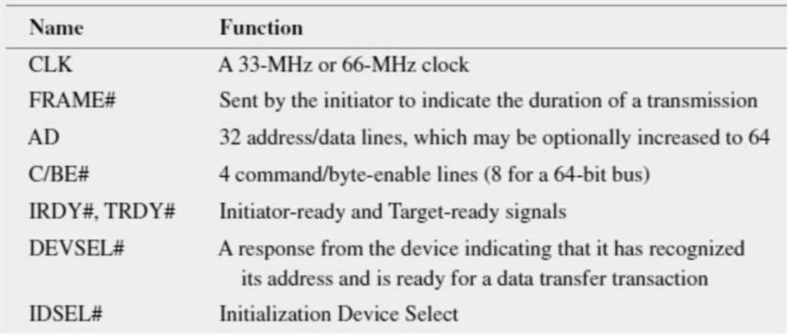
The use of the PCI bus in a computer system is illustrated in Figure. The PCI bus is connected to the processor bus via a controller called a bridge. The bridge has a special port for connecting the computer's main memory. It may also have another special highspeed port for connecting graphics devices. The bridge translates and relays commands and responses from one bus to the other and transfers data between them. For example, when the processor sends a Read request to an I/O device, the bridge forwards the command and address to the PCI bus. When the bridge receives the device's response, it forwards the data to the processor using the processor bus. I/O devices are connected to the PCI bus, possibly through ports that use standards such as Ethernet, USB, SATA, SCSI, or SAS. The PCI bus supports three independent address spaces: memory, I/O, and configuration.



The system designer may choose to use memory mapped I/O even with a processor that has a separate I/O address space. In fact, this is the approach recommended by the PCI standard for wider compatibility. The configuration space is intended to give the PCI its plug-and-play capability, as we will explain shortly. A4-bit command that accompanies the address identifies which of the three spaces is being used in a given data transfer operation. Data transfers on a computer bus often involve bursts of data rather than individual words. Words stored in successive memory locations are transferred directly between the memory and an I/O device such as a disk or an Ethernet connection. Data transfers are initiated by the interface of the I/O device, which acts as a bus master. The PCI bus is designed primarily to support multiple-word transfers. A Read or a Write operation involving a single word is simply treated as a burst of length one. The signaling convention on the PCI bus is similar to that used, with one important difference. The PCI bus uses the same lines to transfer both address and data. In Figure, we assumed that the master maintains the address information on the bus until the data transfer is completed. But, this is not necessary. The address is needed only long enough for the slave to be selected, freeing the lines for sending data in subsequent clock cycles. For transfers involving multiple words, the slave can store the address in an internal register and increment it to access successive address locations. A significant cost reduction can be realized in this manner, because the number of bus lines is an important factor affecting the cost of a computer system.

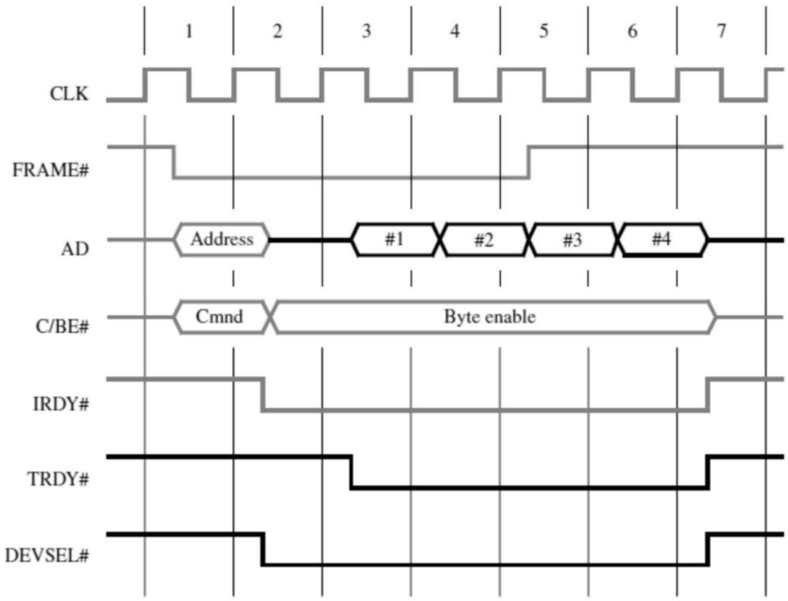
Data Transfer

To understand the operation of the bus and its various features, we will examine a typical bus transaction. The bus master, which is the device that initiates data transfers by issuing Read and Write commands, is called the initiator in PCI terminology. The addressed device that responds to these commands is called a target. The main bus signals used for transferring data are listed in Table. There are 32 or 64 lines that carry address and data using a synchronous signaling scheme similar to that of Figure. The target-ready, TRDY#, signal is equivalent to the Slave-ready signal in that figure. In addition, PCI uses an initiator-ready signal, IRDY#, to support burst transfers. We will describe these signals briefly, to provide the reader with an appreciation of the main features of the bus. A complete transfer operation on the PCI bus, involving an address and a burst of data, is called a transaction. Consider a bus transaction in which an initiator reads four consecutive 32-bit words from the memory. The sequence of events on the bus is illustrated in Figure. All signal transitions are triggered by the rising edge of the clock. As in the case of Figure, we show the signals changing later in the clock cycle to indicate the delays they encounter. A signal whose name ends with the symbol # is asserted when in the low-voltage state.



Name	Function
CLK	A 33-MHz or 66-MHz clock
FRAME#	Sent by the initiator to indicate the duration of a transmission
AD	32 address/data lines, which may be optionally increased to 64
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus)
IRDY#, TRDY#	Initiator-ready and Target-ready signals
DEVSEL#	A response from the device indicating that it has recognized its address and is ready for a data transfer transaction
IDSEL#	Initialization Device Select

The bus master, acting as the initiator, asserts FRAME# in clock cycle 1 to indicate the beginning of a transaction. At the same time, it sends the address on the AD lines and a command on the C/BE# lines. In this case, the command will indicate that a Read operation is requested and that the memory address space is being used.



In clock cycle 2, the initiator removes the address, disconnects its drivers from the AD lines, and asserts IRDY# to indicate that it is ready to receive data. The selected target asserts DEVSEL# to indicate that it has recognized its address and is ready to respond. At the same time, it enables its drivers on the AD lines, so that it can send data to the initiator in subsequent cycles. Clock cycle 2 is used to accommodate the delays involved in turning the AD lines around, as the initiator turns its drivers off and the target turns its drivers on. The target asserts TRDY# in clock cycle 3 and begins to send data. It maintains DEVSEL# in the asserted state until the end of the transaction.

We have assumed that the target is ready to send data in clock cycle 3. If not, it would delay asserting TRDY# until it is ready. The entire burst of data need not be sent in successive clock cycles. Either the initiator or the target may introduce a pause by deactivating its ready signal, then asserting it again when it is ready to resume the transfer of data. The C/BE# lines, which are used to send a bus command in clock cycle 1, are used for

a different purpose during the rest of the transaction. Each of these four lines is associated with one byte on the AD lines. The initiator asserts one or more of the C/BE# lines to indicate which byte lines are to be used for transferring data. The initiator uses the FRAME# signal to indicate the duration of the burst. It deactivates this signal during the second-last word of the transfer. In Figure, the initiator maintains FRAME# in the asserted state until clock cycle 5, the cycle in which it receives the third word. In response, the target sends one more word in clock cycle 6, then stops. After sending the fourth word, the target deactivates TRDY# and DEVSEL# and disconnects its drivers on the AD lines.

Device Configuration

When an I/O device is connected to a computer, several actions are needed to configure both the device interface and the software that communicates with it. Like USB, PCI has a plug-and-play capability that greatly simplifies this process. In fact, the plug-and-play feature was pioneered by the PCI standard. A PCI interface includes a small configuration ROM memory that stores information about the I/O device connected to it. The configuration ROMs of all devices are accessible in the configuration address space, where they are read by the PCI initialization software whenever the system is powered up or reset. By reading the information in the configuration ROM, the software determines whether the device is a printer, a camera, an Ethernet interface, or a disk controller. It can further learn about various device options and characteristics.

Devices connected to the PCI bus are not assigned permanent addresses that are built into their I/O interface hardware. Instead, device addresses are assigned by software during the initial configuration process. This means that when power is turned on, devices cannot be accessed using their addresses in the usual way, as they have not yet been assigned any address. A different mechanism is used to select I/O devices at that time.

The PCI bus may have up to 21 connectors for I/O device interface cards to be plugged. Each connector has a pin called Initialization Device Select (IDSEL#). This pin is connected to one of the upper 21 address/data lines, AD11 to AD31. A device interface responds to a configuration command if its IDSEL# input is asserted. The configuration software scans all 21 locations to identify where I/O device interfaces are present. For each location, it issues a configuration command using an address in which the AD line corresponding to that location is set to 1 and the remaining 20 lines are set to 0. If a device interface responds, it is assigned an address and that address is written into one of its registers designated for this purpose. Using the same addressing mechanism, the processor reads the device's configuration ROM and carries out any necessary initialization. It uses the low-order address bits, AD0 to AD10, to access locations within the configuration ROM. This automated process means that the user simply plugs in the interface board and turns on the power. The software does the rest.

The PCI bus has gained great popularity, particularly in the PC world. It is also used in many other computers, to benefit from the wide range of I/O devices for which a PCI interface is available. Both a 32-bit and a 64-bit configuration are available, using either a 33-MHz or 66-MHz clock. A high-performance variant known as PCI-X is also available. It is a 64-bit bus that runs at 133 MHz. Yet higher performance versions of PCI-X run at speeds up to 533 MHz.

SCSI BUS

The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI). The SCSI bus may be used to connect a variety of devices to a computer. It is particularly well-suited for use with disk drives. It is often found in installations such as institutional databases or email systems where many disks drives are used.

In the original specifications of the SCSI standard, devices are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates of up to 5 Megabytes/s. The standard has undergone many revisions, and its data transfer capability has increased rapidly. SCSI-2 and SCSI-3 have been defined, and each has several options. Data are transferred either 8 bits or 16 bits in parallel, using clock speeds of up to 80 MHz. There are also several options for the electrical signaling scheme used. The bus may use single- ended transmission, where each signal uses one wire, with a common ground return for all signals. In another option, differential signaling is used, with a pair of wires for each signal.

Data Transfer

Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus or to the PCI bus. A SCSI bus may be connected directly to the processor bus, or more likely to another standard I/O bus such as PCI, through a SCSI controller. Data and commands are transferred in the form of multi-byte messages called packets. To send commands or data to a device, the processor assembles the information in the memory then instructs the SCSI controller to transfer it to the device. Similarly, when data are read from a device, the controller transfers the data to the memory and then informs the processor by raising an interruption.

To illustrate the operation of the SCSI bus, let us consider how it may be used with a disk drive. Communication with a disk drive differs substantially from communication with the main memory. Data are stored on a disk in blocks called sectors, where each sector may contain several hundred bytes. When a data file is written on a disk, it is not always stored in contiguous sectors. Some sectors may already contain previously stored information; others may be defective and must be skipped. Hence, a Read or Write request may result in accessing several disk sectors that are not necessarily contiguous. Because of the constraints of the mechanical motion of the disk, there is a long delay, on the order of several milliseconds, before reaching the first sector to or from which data are to be transferred. Then, a burst of data are transferred at high speed. Another delay may ensue to reach the next sector, followed by a burst of data. A single Read or Write request may involve several such bursts. The SCSI protocol is designed to facilitate this mode of operation.

Let us examine a complete Read operation as an example. The following is a simplified high-level description, ignoring details and signaling conventions. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller contends for control of the SCSI bus.
2. When it wins the arbitration process, the SCSI controller sends a command to the disk controller, specifying the required Read operation.
3. The disk controller cannot start to transfer data immediately. It must first move the read head of the disk to the required sector. Hence, it sends a message to the SCSI controller indicating that it will temporarily suspend the connection between them. The SCSI bus is now free to be used by other devices.
4. The disk controller sends a command to the disk drive to move the read head to the first sector involved in the requested Read operation. It reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data, it requests control of the bus. After it wins arbitration, it re-establishes the connection with the SCSI controller, sends the contents of the data buffer, then suspends the connection again.
5. The process is repeated to read and transfer the contents of the second disk sector.
6. The SCSI controller transfers the requested data to the main memory and sends an interrupt to the processor indicating that the data are now available.

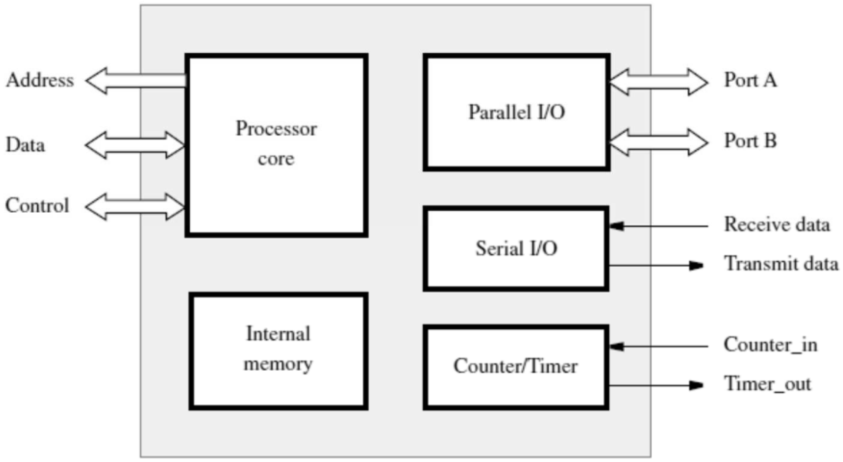
This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. Messages refer to more complex operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware of the details of the disk's operation and how it moves from one sector to the next.

The SCSI bus standard defines a wide range of control messages that can be used to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

I/O DEVICES AND PROCESSORS

Parallel I/O Interface

Embedded system applications require considerable flexibility in input/output interfaces. The nature of the devices involved and how they may be connected to the microcontroller can be appreciated by considering some components of the microwave oven shown in Figure. A sensor is needed to generate a signal with value 1 when the door is open. This signal is sent to the microcontroller on one of the pins of an input interface. The same is true for the keys on the microwave's front panel. Each of these simple devices produces one bit of information.



Output devices are controlled in a similar way. The magnetron is controlled by a single output line that turns it on or off. The same is true for the fan and the light. The speaker may also be connected via a single output line on which the processor sends a square wave signal having an appropriate tone frequency. A liquid-crystal display, on the other hand, requires several bits of data to be sent in parallel.

One of the objectives of the design of input/output interfaces for a microcontroller is to reduce the need for external circuitry as much as possible. The microcontroller is likely to be connected to simple devices, many of which require only one input or output signal line. In most cases, no encoding or decoding is needed. Each parallel port has an associated eight-bit data direction register, which can be used to configure individual data lines as either input or output. Figure illustrates the bidirectional control for one bit in port A. Port pin PAi is treated as an input if the data direction flip-flop contains a 0. In this case, activation of the control signal Read_Port places the logic value on the port pin onto the data line Di of the processor bus. The port pin serves as an output if the data direction flip-flop is set to 1. The value loaded into the output data flip-flop, under control of the Write_Port signal, is placed on the pin.

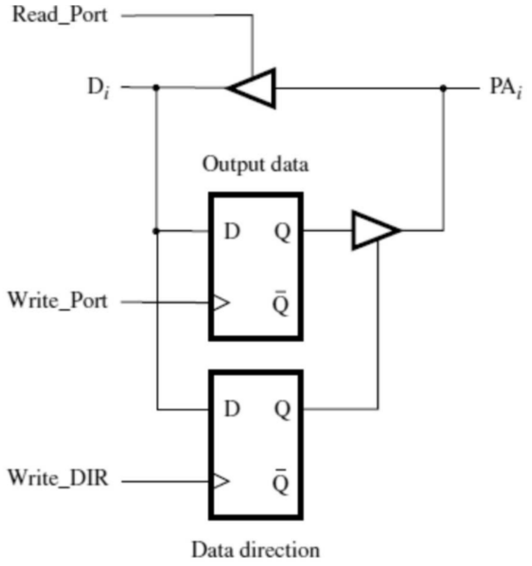
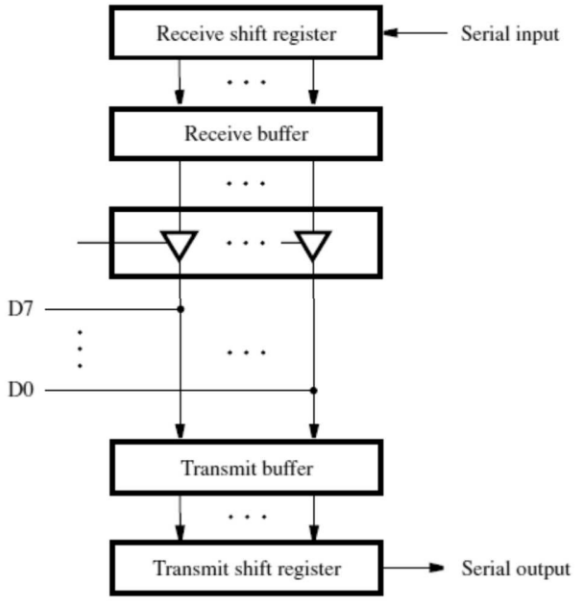


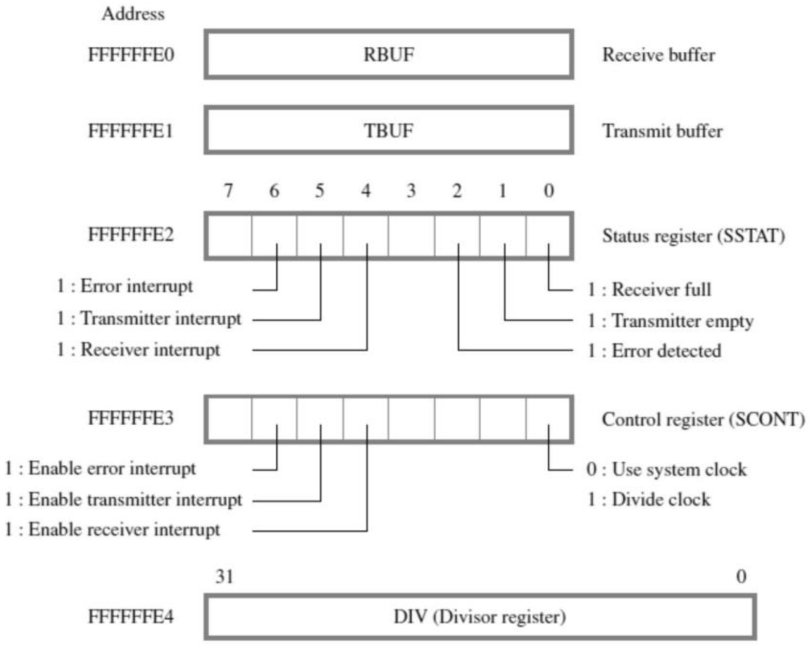
Figure shows only the part of the interface that controls the direction of data transfer. In the input data path there is no flip-flop to capture and hold the value of the data signal provided by a device connected to the corresponding pin. A versatile parallel interface may include two possibilities: one where input data are read directly from the pins, and the other where the input data are stored in a register as in the interface in Figure. The choice is made by setting a bit in the control register of the interface.

Serial I/O Interface

The serial interface provides the UART (Universal Asynchronous Receiver/Transmitter) capability to transfer data. Double buffering is used in both the transmit and receive paths, as shown in Figure. Such buffering is needed to handle bursts in I/O transfers correctly. Figure shows the addressable registers of the serial interface. Input data are read from the 8-bit Receive buffer, and output data are loaded into the 8-bit Transmit buffer. The status register, SSTAT, provides information about the current status of the receive and transmit units. Bit SSTAT0 is set to 1 when there are valid data in the receive buffer; it is cleared to 0 automatically upon a read access to the receive buffer. Bit SSTAT1 is set to 1 when the transmit buffer is empty and can be loaded with new data. These bits serve the same purpose as the status flags KIN and DOUT discussed in Section 3.1. Bit SSTAT2 is set to 1 if an error occurs during the receive process. For example, an error occurs if the character in the receive buffer is overwritten by a subsequently received character before the first character is read by the processor. The status register also contains the interrupt flags. Bit SSTAT4 is set to 1 when the receive buffer becomes full and the receiver interrupt is enabled. Similarly, SSTAT5 is set to 1 when the transmit buffer becomes empty and the transmitter interrupt is enabled. The serial interface raises an interrupt if either SSTAT4 or SSTAT5 is equal to 1. It also raises an interrupt if SSTAT6 = 1, which occurs if SSTAT2 = 1 and the error condition interrupt is enabled.



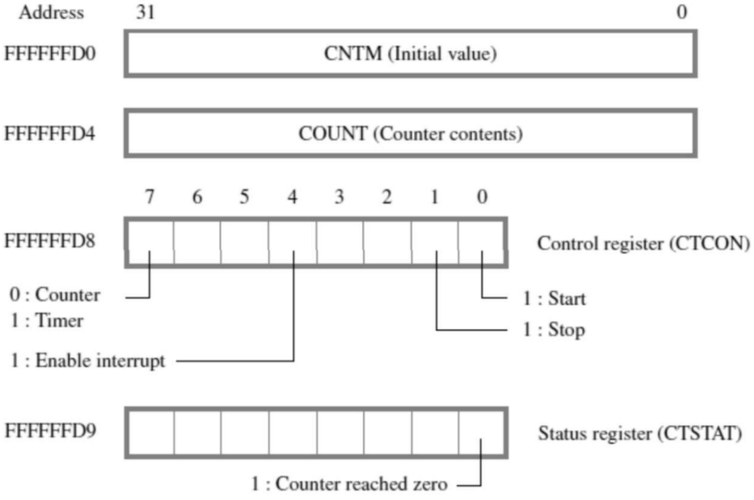
The control register, SCONT, is used to hold the interrupt-enable bits. Setting bits SCONT6−4 to 1 or 0 enables or disables the corresponding interrupts, respectively. This register also indicates how the transmit clock is generated. If SCONT0 = 0, then the transmit clock is the same as the system (processor) clock. If SCONT0 = 1, then a lower frequency transmit clock is obtained using a clock-dividing circuit.



The last register in the serial interface is the clock-divisor register, DIV. This 32-bit register is associated with a counter circuit that divides down the system clock signal to generate the serial transmission clock. The counter generates a clock signal whose frequency is equal to the frequency of the system clock divided by the contents of this register. The value loaded into this register is transferred into the counter, which then counts down using the system clock. When the count reaches zero, the counter is reloaded using the value in the DIV register.

Counter/Timer

A 32-bit down-counter circuit is provided for use as either a counter or a timer. The basic operation of the circuit involves loading a starting value into the counter, and then decrementing the counter contents using either the internal system clock or an external clock signal. The circuit can be programmed to raise an interrupt when the counter contents reach zero. Figure shows the registers associated with the counter/timer circuit. The counter/timer register, CNTM, can be loaded with an initial value, which is then transferred into the counter circuit. The current contents of the counter can be read by accessing memory address FFFFFFD4. The control register, CTCON, is used to specify the operating mode of the counter/timer circuit. It provides a mechanism for starting and stopping the counting process, and for enabling interrupts when the counter contents are decremented to 0. The status register, CTSTAT, reflects the state of the circuit.



Counter Mode

The counter mode is selected by setting bit CTCON7 to 0. The starting value is loaded into the counter by writing it into register CNTM. The counting process begins when bit CTCON0 is set to 1 by a program instruction. Once counting starts, bit CTCON0 is automatically cleared to 0. The counter is decremented by pulses on the Counter_in line. Upon reaching 0, the counter circuit sets the status flag CTSTAT0 to 1, and raises an interrupt if the corresponding interrupt-enable bit has been set to 1. The next clock pulse causes the counter to reload the starting value, which is held in register CNTM, and counting continues. The counting process is stopped by setting bit CTCON1 to 1.

Timer Mode

The timer mode is selected by setting bit CTCON7 to 1. This mode can be used to generate periodic interrupts. It is also suitable for generating a square-wave signal on the output line Timer_out in Figure. The process starts as explained above for the counter mode. As the counter counts down, the value on the output line is held constant. Upon reaching zero, the counter is reloaded automatically with the starting value, and the output signal on the line is inverted. Thus, the period of the output signal is twice the starting counter value multiplied by the period of the controlling clock pulse. In the timer mode, the counter is decremented by the system clock.

Interrupt-Control Mechanism

The processor in our example microcontroller has two interrupt-request inputs, IRQ and XRQ. The IRQ input is used for interruptions raised by the I/O interfaces within the microcontroller. The XRQ input is used for interruptions raised by external devices. If the IRQ input is asserted and interruptions are enabled, the processor executes an interrupt-service routine that uses the polling method to determine the source(s) of the interruption request. This is done by examining the flags in the status registers PSTAT, SSTAT, and CTSTAT. The XRQ interrupts have higher priority than the IRQ interrupts. The processor status register, PSR, has two bits for enabling interruptions. The IRQ interrupts are enabled if PSR6 = 1, and the XRQ interrupts are enabled if PSR7 = 1. When the processor accepts an interruption, it disables further interrupts at the same priority level by clearing the corresponding PSR bit before the interrupt service routine is executed. A vectored interrupt scheme is used, with the vectors for IRQ and XRQ interrupts in memory locations 0x20 and 0x24, respectively. Each vector contains the address of the first instruction of the corresponding interrupt-service routine. This address is automatically loaded into the program counter, PC.

The processor has a Link register, LR, which is used for subroutine linkage as explained. A subroutine Call instruction causes the updated contents of the program counter, which is the required return address, to be stored in LR prior to branching to the first instruction in the subroutine. There is another register, IRA, which saves the return address when an interrupt request is accepted. In this case, in addition to saving the return address in IRA, the contents of the processor status register, PSR, are saved in processor register IPSR. Return from a subroutine is performed by a ReturnS instruction, which transfers the contents of LR into PC. Return from an interrupt is performed by a ReturnI instruction, which transfers the contents of IRA and IPSR into PC and PSR, respectively. Since there is only one IRA and IPSR register, nested interrupts can be implemented by saving the contents of these registers on the stack using instructions in the interrupt-service routine. Note that if the interrupt-service routine calls a subroutine, then it must save the contents of LR, because an interrupt may occur when the processor is executing another subroutine.

PROCESSORS

This lesson describes three additional techniques for improving performance, namely multithreading, vector processing, and multiprocessing. They increase performance by improving the utilization of processing resources and by performing more operations in parallel.

Hardware Multithreading

Operating system (OS) software enables multitasking of different programs in the same processor by performing context switches among programs. A program, together with any information that describes its current state of execution, is regarded by the OS as an entity called a process. Information about the memory and other resources allocated by the OS is maintained with each process. Processes may be associated with applications such as Web- browsing, word-processing, and music-playing programs that a user has opened in a computer. Each process has a corresponding thread, which is an independent path of execution within a program. More precisely, the term thread is used to refer to a thread of control whose state consists of the contents of the program counter and other processor registers.

It is possible for multiple threads to execute portions of one program and run in parallel as if they correspond to separate programs. Two or more threads can be run on different processors, executing either the same part of a program on different data, or executing different parts of a program. Threads for different programs can also be executed on different processors. All threads that are part of a single program run in the same address space and are associated with the same process. In this section, we focus on multitasking where two or more programs run on the same processor and each program has a single thread. It describes the technique of time slicing, where the OS selects a process among those that are not presently blocked and allows this process to run for a short period of time. Only the thread corresponding to the selected process is active during the time slice. Context switching at the end of the time slice causes the OS to select a different process, whose corresponding thread becomes active during the next time slice. A timer interrupt invokes an interrupt-service routine in the OS to switch from one process to another.

To deal with multiple threads efficiently, a processor is implemented with several identical sets of registers, including multiple program counters. Each set of registers can be dedicated to a different thread. Thus, no time is wasted during a context switch to save and restore register contents. The processor is said to be using a technique called hardware multithreading. With multiple sets of registers, context switching is simple and fast. All that is necessary is to change a hardware pointer in the processor to use a different set of registers to fetch and execute subsequent instructions. Switching to a different thread can be completed within one clock cycle. The state of the previously active thread is preserved in its own set of registers.

Switching to a different thread may be triggered at any time by the occurrence of a specific event, rather than at the end of a fixed time interval. For example, a cache miss may occur when a Load or Store instruction is being executed for the active thread. Instead of stalling while the slower main memory is accessed to service the cache miss, a processor can quickly switch to a different thread and continue to fetch and execute other instructions. This is called coarse-grained multithreading because many instructions may be executed for one thread before an event such as a cache miss causes a switch to another thread. An alternative to switching between threads on specific events is to switch after every instruction is fetched. This is called fine-grained or interleaved multithreading. The intent is to increase the processor throughput. Each new instruction is independent of its predecessors from other threads. This should reduce the occurrence of stalls due to data dependencies. Thus, throughput may be increased by interleaving instructions from many threads, but it takes longer for a given thread to complete all of its instructions. A form of interleaved multithreading with only two threads is used in processors that implement the Intel IA-32 architecture described in Appendix E.

Vector (SIMD) Processing

Many computationally demanding applications involve programs that use loops to perform operations on vectors of data, where a vector is an array of elements such as integers or floating-point numbers. When a processor executes the instructions in such a loop, the operations are performed one at a time on individual vector elements. As a result, many instructions need to be executed to process all vector elements. A processor can be enhanced with multiple ALUs. In such a processor, it is possible to operate on multiple data elements in parallel using a single instruction. Such instructions are called single-instruction multiple-data (SIMD) instructions. They are also called vector instructions. These instructions can only be used when the operations performed in parallel are independent. This is known as data parallelism.

The data for vector instructions are held in vector registers, each of which can hold several data elements. The number of elements, L, in each vector register is called the vector length. It determines the number of operations that can be performed in parallel on multiple ALUs. If vector instructions are provided for different sizes of data elements using the same vector registers, L may vary. For example, the Intel IA-32 architecture has 128-bit vector registers that are used by instructions for vector lengths ranging from L = 2 up to L = 16, corresponding to integer data elements with sizes ranging from 64 bits down to 8 bits. Some typical examples of vector instructions are given below to illustrate how vector registers are used. We assume that the OP-code mnemonic includes a suffix S which specifies the size of each data element. This determines the number of elements, L, in a vector. For instructions that access the memory, the contents of a conventional register are used in the calculation of the effective address.

Graphics Processing Units (GPUs)

The increasing demands of processing for computer graphics has led to the development of specialized chips called graphics processing units (GPUs). The primary purpose of GPUs is to accelerate the large number of floating-point calculations needed in high- resolution three-dimensional graphics, such as in video games. Since the operations involved in these calculations are often independent, a large GPU chip contains hundreds of simple cores with floating-point ALUs to perform them in parallel.

AGPU chip and a dedicated memory for it are included on a video card. Such a card is plugged into an expansion slot of a host computer using an interconnection standard such as the PCIe standard. A small program is written for the processing cores in the GPU chip. A large number of cores execute this program in parallel. The cores execute the same instructions, but operate on different data elements. A separate controlling program runs in the general-purpose processor of the host computer and invokes the GPU program when necessary. Before initiating the GPU computation, the program in the host computer must first transfer the data needed by the GPU program from the main memory into the dedicated GPU memory. After the computation is completed, the resulting output data in the dedicated memory are transferred back to the main memory.

The processing cores in a GPU chip have a specialized instruction set and hardware architecture, which are different from those used in a general-purpose processor. An example is the Compute Unified Device Architecture (CUDA) that NVIDIA Corporation uses for the cores in its GPU chips. To facilitate writing programs that involve a general-purpose processor and a GPU, an extension to the C programming language, called CUDA C, has been developed by NVIDIA. This extension enables a single program to be written in C, with special keywords used to label the functions executed by the processing cores in a GPU chip. The compiler and related software tools automatically partition the final object program into the portions that are translated into machine instructions for the host computer and the GPU chip. Library routines are provided to allocate storage in the dedicated memory of a GPU-based video card and to transfer data between the main memory and the dedicated memory. An open standard called OpenCL has also been proposed by industry as a programming framework for systems that include GPU chips from any vendor.

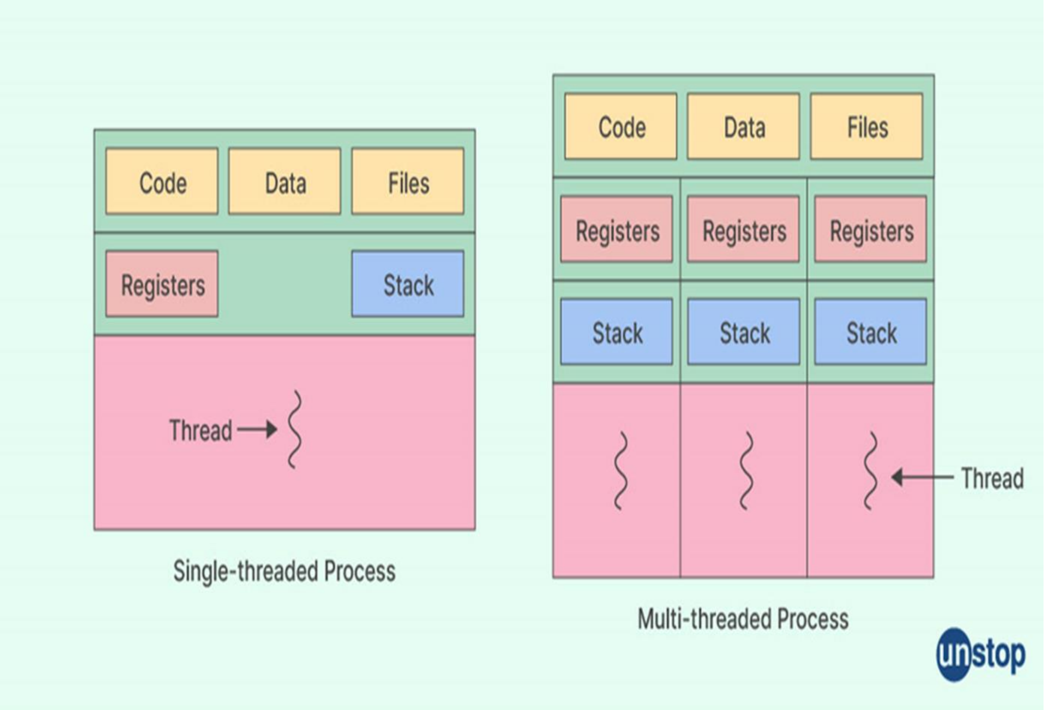
Shared-Memory Multiprocessors

A multiprocessor system consists of a number of processors capable of simultaneously executing independent tasks. The granularity of these tasks can vary considerably. A task may encompass a few instructions for one pass through a loop, or thousands of instructions executed in a subroutine. In a shared-memory multiprocessor, all processors have access to the same memory. Tasks running in different processors can access shared variables in the memory using the same addresses. The size of the shared memory is likely to be large. Implementing a large memory in a single module would create a bottleneck when many processors make requests to access the memory simultaneously. This problem is alleviated by distributing the memory across multiple modules so that simultaneous requests from different processors are more likely to access different memory modules, depending on the addresses of those requests. An interconnection network enables any processor to access any module that is a part of the shared memory. When memory modules are kept physically separate from the processors, all requests to access memory must pass through the network, which introduces latency. Figure shows such an arrangement. A system which has the same network latency for all accesses from the processors to the memory modules is called a Uniform Memory Access (UMA) multiprocessor. Although the latency is uniform, it may be large for a network that connects many processors and memory modules. For better performance, it is desirable to place a memory module close to each processor. The result is a collection of nodes, each consisting of a processor and a memory module.

Message-Passing Multi-computers

A different way of using multiple processors involves implementing each node in the system as a complete computer with its own memory. Other computers in the system do not have direct access to this memory. Data that needs to be shared are exchanged by sending messages from one computer to another. Such systems are called message-passing multicomputers. Parallel programs are written differently for message-passing multicomputers than for shared-memory multiprocessors. To share data between nodes, the program running in the computer that is the source of the data must send a message containing the data to the destination computer. The program running in the destination computer receives the message and copies the data into the memory of that node. To facilitate message passing, a special communications unit at each node is often responsible for the low-level details of formatting and interpreting messages that are sent and received, and for copying message data to and from the memory of the node. The computer in each node issues commands the communications unit. The computer then continues performing other computations while the communications unit handles the details of sending and receiving messages.

INTRODUCTION TO MULTITHREADING



A thread is a lightweight process that consumes fewer resources than the process. Each thread belongs to only one process, and there are no threads outside of a process.

Multithreading is a computer architecture technique that allows multiple threads to run simultaneously on a central processing unit (CPU) or a single core in a multi-core processor. The goal of multithreading is to increase a computer's performance and computing speed by optimizing CPU usage. Multithreading splits a large workload into multiple software threads, which are then processed in parallel by different CPU cores.

Multithreading allows a computer to handle multiple requests from a user or multiple users at once without requiring multiple copies of the program. It also allows the system to quickly switch to the next task, even when a process is waiting for data.

Multithreading is a software and hardware technology that allows a processor to execute multiple threads, or independent units of execution, simultaneously. This can improve the performance of an application and make it more responsive to the user. A program or operating system can use multithreading to handle multiple requests from a user or multiple users at once. Each request is tracked as a thread with its own identity.

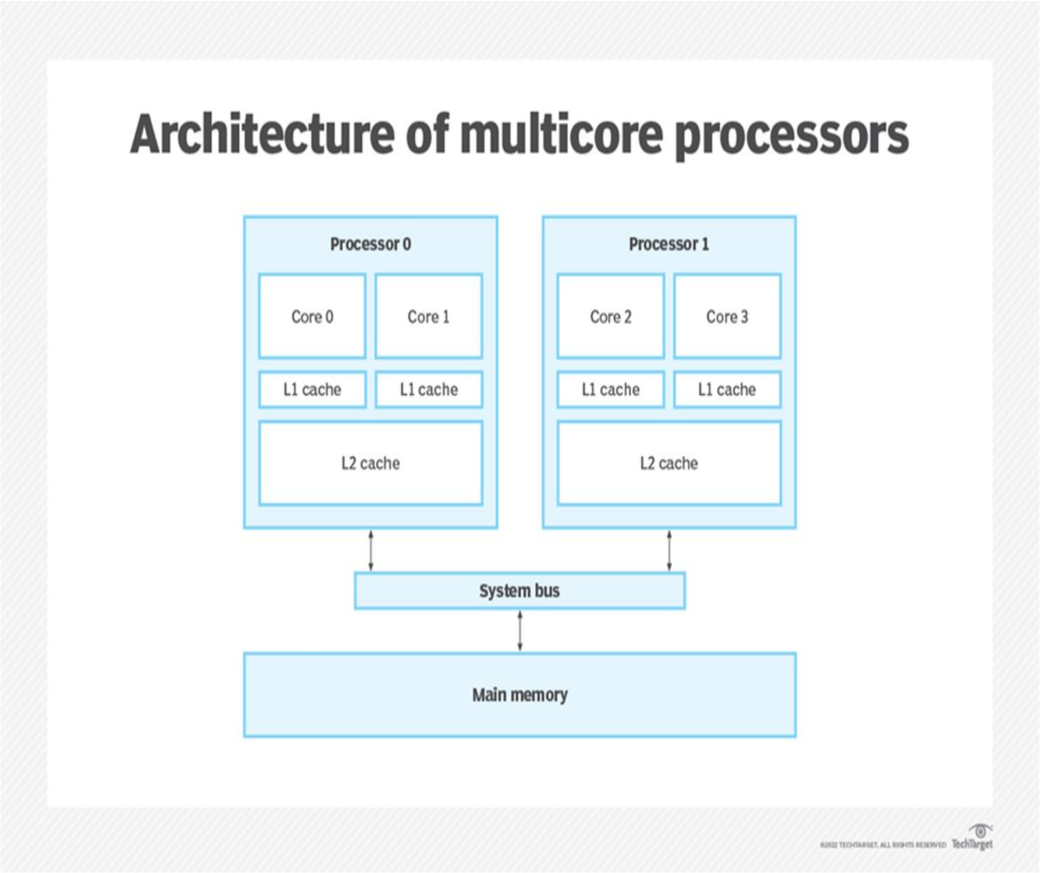
The processor executes pieces of different programs so quickly that it appears the programs are running simultaneously. Multithreading can improve performance by breaking processes into smaller threads that can be executed concurrently. For example, in a web browser, one thread can handle the user interface while another thread fetches data to display. Multithreading requires a fast central processing unit (CPU) and large memory capacity. Depending on the hardware, threads can run fully parallel if they are distributed to their own CPU core.

Multithreading differs from **multiprocessing**, which uses multiple complete processing units in one or more cores. Multithreading is useful for applications that consist of several processes and threads, such as video editing. Multithreading is sometimes combined with instruction-level parallelism and other techniques in systems with multiple multithreading CPUs or CPUs with multiple multithreading cores.

In computer architecture, software multithreading is a technique that allows a program to run multiple threads at the same time. Multithreading can improve performance and efficiency by keeping a processor busy, even if a program is waiting for memory or has a low instruction level parallelism (ILP).

Multithreading can introduce challenges like race conditions and deadlocks. It can also be difficult to design an application so that different threads can run concurrently without interfering with each other. Multithreading requires a fast central processing unit (CPU) speed and large memory capacities. Depending on the hardware, threads can run fully parallel if they are distributed to their own CPU core.

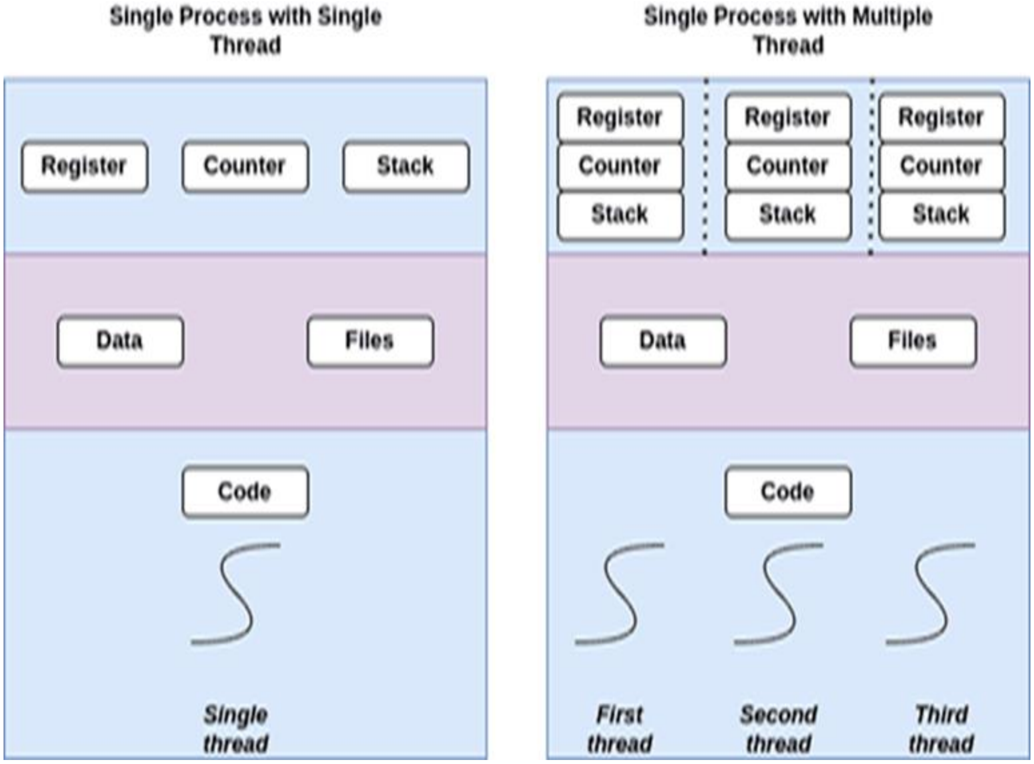
Hardware multithreading



Hardware multithreading is a computer architecture technique that allows a central processing unit (CPU) to run multiple threads of execution simultaneously. This is in contrast to a single-threaded processor, which can only execute one software thread at a time. In a multithreaded system, threads share resources like the CPU caches, computing units, and translation lookaside buffer (TLB). The goal of multithreading is to increase the utilization of a single core by using instruction-level and thread-level parallelism.

Software Multithreading

Software Multithreading can also refer to the ability of an operating system or program to allow multiple users to access the system at the same time. In this case, each user request is tracked as a thread with its own identity. Software Multithreading is different from multiprocessing, which uses multiple complete processing units in one or more cores. However, the two techniques can be combined in systems with multiple multithreading CPUs or CPUs with multiple multithreading.



SMT & CMP ARCHITECTURES

Simultaneous multithreading (SMT) and chip multiprocessors (CMP) are both computer architectures that allow a chip to achieve greater throughput. The main difference between the two is how many instructions a processor can issue in one cycle and how many threads are used:

SMT

A superscalar processor issues multiple instructions from multiple threads in a single cycle. This allows for better utilization of the processor's resources and provides latency tolerance.

CMP

A chip that integrates two or more processors, each executing threads independently. CMPs use simple single-thread processor cores to exploit moderate amounts of parallelism within each thread. Some other differences between SMT and CMP:

Flexibility: SMT processors are more flexible than CMP processors.

Underutilization: If an application can't be effectively decomposed into threads, CMPs will be underutilized.

Thermal properties: The thermal properties of both SMT and CMP are still poorly understood

DESIGN ISSUES: They determine the performance measures of each processor in a precise manner. The issue slots usage limitations and its issues also determine the performance.

- Large performance gap between MEMORY and PROCESSOR.
- Too many transistors on chip.
- More existing MT applications today.
- Multiprocessors on a single chip.
- Long network latency

DESIGN CHALLENGES

- Impact of fine-grained scheduling on single thread performance
- Reason for loss of throughput
- Larger register file needed to hold multiple contexts.
- Not affecting clock cycle time, especially in Instruction issue- (more candidate instructions need to be considered)
- Instruction completion- choosing which instructions to commit may be challenging
- Ensuring that cache and TLP conflicts generated by SMT do not degrade performance.

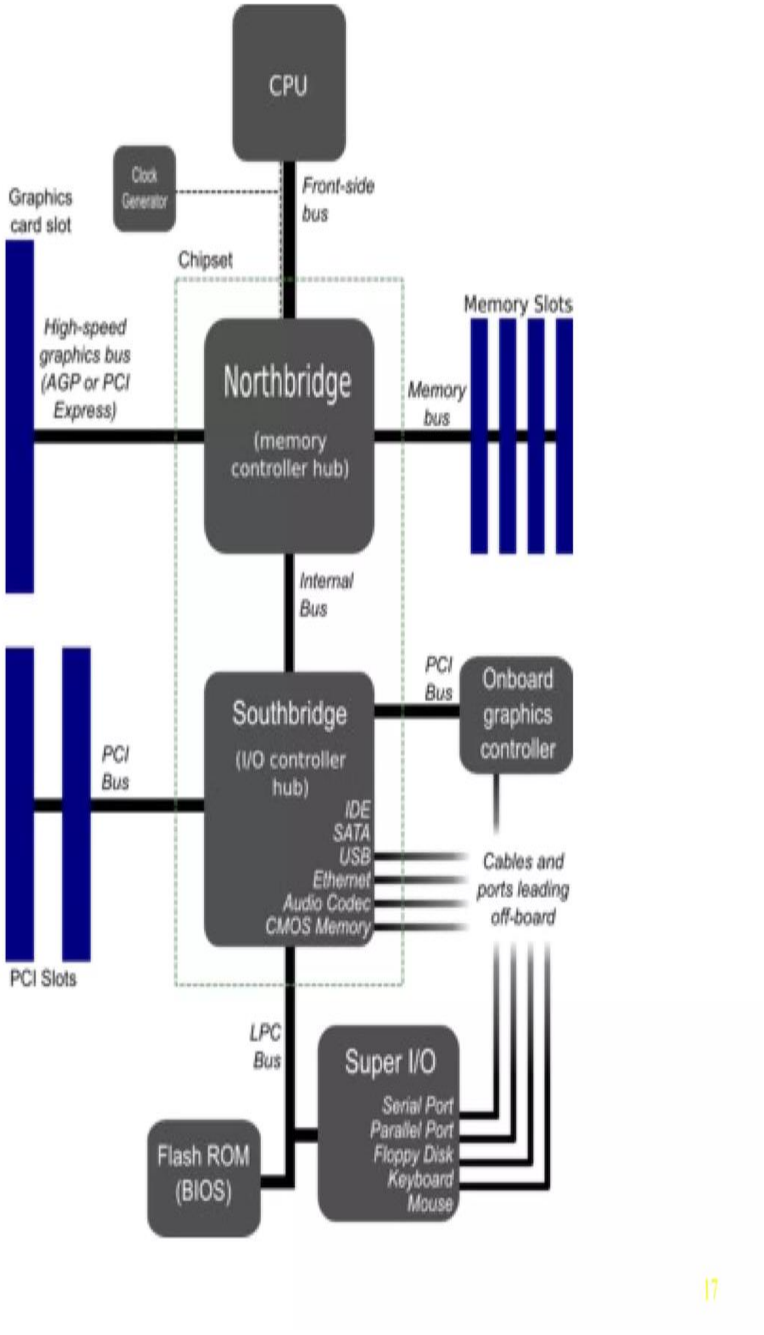
There are mainly two observations

> Potential performance overhead due to multithreading is small. Efficiency of current superscalar is low with the room for significant improvement. A SMT processor works well if Number of compute intensive threads does not exceed the number of threads supported in SMT.

Threads have highly different characteristics

> for e.g., 1 thread doing mostly integer operations and another doing mostly floating-point operations It does not work well if Threads try to utilize the same functional units and for assignment problems E.g., a dual core processor system, each processor having 2 threads simultaneously.

Intel Core i7 processor



Intel Core i7 processor

The Intel Core i7 processor is an example of a multicore processor, which can be used in heterogeneous multi-core architectures:

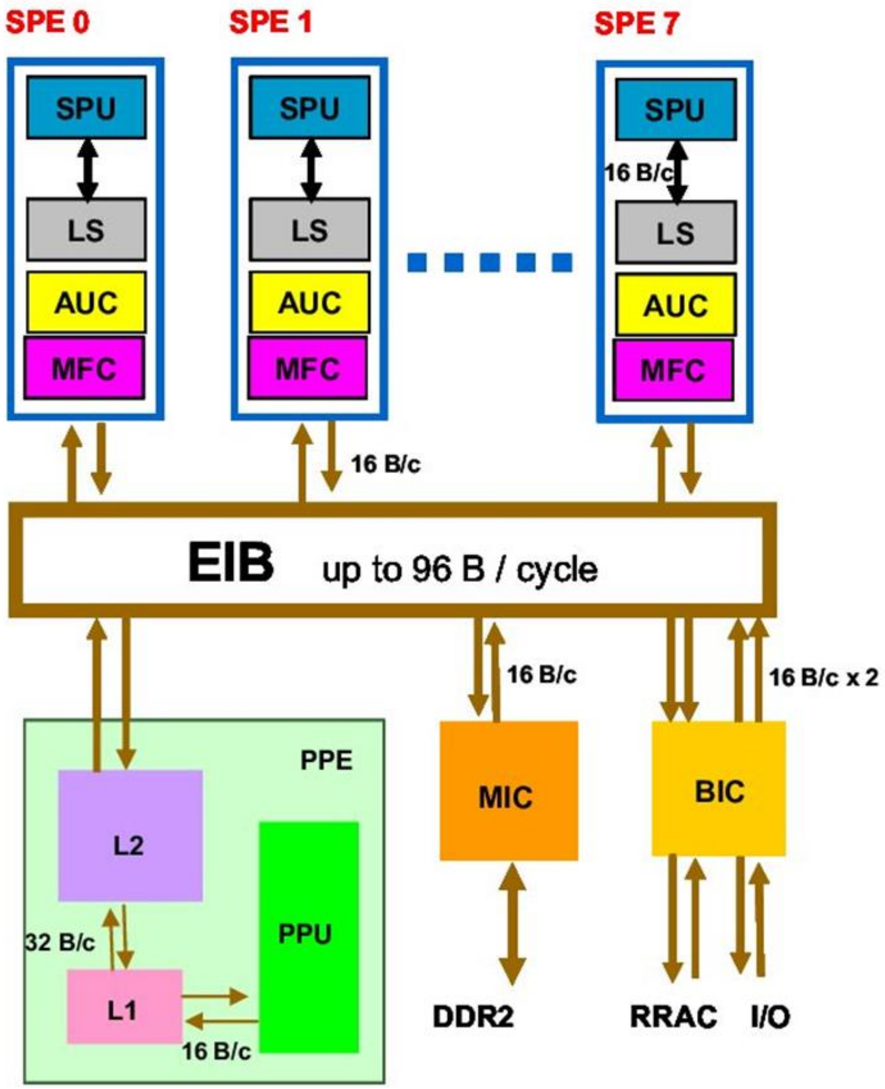
Multicore processor

The Core i7 is a multicore processor with an inclusive shared L3 cache that can be up to 12 MB in size. This shared cache helps to increase performance and reduce latency.

Heterogeneous multi-core architecture

A heterogeneous multi-core architecture combines different types of processing cores, such as CPUs, GPUs, and special-purpose accelerators. This allows the system to use only the cores that are best suited for the current task, which can lead to energy-efficient computation.

IBM CELL PROCESSOR



A chip with one PPC hyper-threaded core called **PPE** and eight specialized cores called **SPEs.** The challenge to be solved by the cell was to put all those cores together on a single chip. This was made possible by the use of a bus with outstanding performance.

The Cell processor can be split into four components**:**

1.External input and output structures, the main processor called the Power Processing Element (PPE)

2.Eight fully functional co-processors called the Synergistic Processing Elements, or SPEs,

3.A specialized high-bandwidth circular data bus connecting the PPE,

4.Input/output elements and the SPEs, called the Element Interconnect Bus or EIB.

POWER PC PROCESSOR ELEMENT:(PPE)

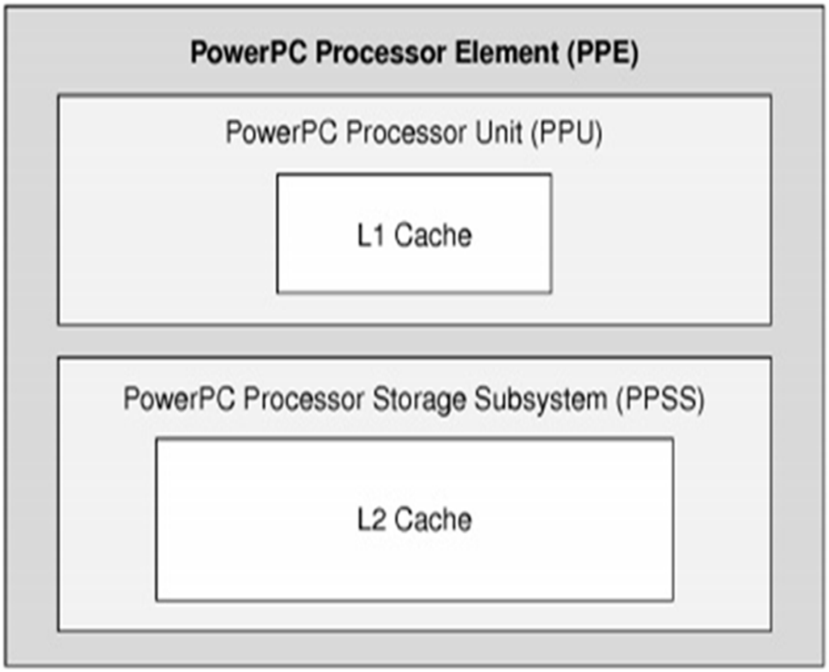
The PowerPC Processor Element, usually denoted as PPE is a dual-threaded PowerPC processor version 2.02. This 64-bit RISC processor also has the Vector/SIMD Multimedia Extension.The PPE's role is crucial in the Cell architecture since it is on the one hand running the OS, and on the other hand controlling all other resources, including the SPEs .

The PPE is made out of two main units:

1. The Power Processor Unit

2. The Power Processor Storage Subsystem (PPSS).

PPE Block diagram


PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache

PPU:

It is the processing part of the PPE and is composed of:

- ✓ A full set of 64-bit PowerPC registers.
- ✓ 32 128-bit vector multimedia registers.
- ✓ A 32KB L1 instruction cache.
- ✓ A 32KB L1 data cache

All the common components of a ppc processors with vector/SIMD extensions (instruction control unit, load and store unit, fixed-Point integer unit, floating-point unit, vector unit, branch unit, virtual memory management unit). The PPU is hyper-threaded and supports 2 simultaneous threads.

PPSS:

This handles all memory requests from the PPE and requests made to the PPE by other processors or I/O devices. It is composed of:

A unified 512-KB L2 instruction and data cache.

Various queues

A bus interface unit that handles bus arbitration and pacing on the Element Interconnect Bus

SYNERGISTIC PROCESSOR ELEMENTS: SPE

Each Cell chip has 8 Synergistic Processor Elements. They are 128-bit RISC processor which are specialized for data-rich, compute-intensive SIMD applications.

This consist of two main units.

1: The Synergistic Processor Unit (SPU)

2: The Memory Flow Controller (MFC)

Key Attributes of Cell

- ✓ Cell is multi-core
- ✓ Cell is a Flexible Architecture
- ✓ Cell is a Broadband Architecture
- ✓ Cell is a Real-Time Architecture
- ✓ Cell is a Security Enabled Architecture