Lecture Notes

on

COMPUTER ORGANIZATION

.

CONTENTS:

1. Functional units
    - Basic operational concepts – Bus structures – Performance and metrics
2. Instructions and instruction sequencing
3. Instruction set architecture
    - Addressing modes
4. Basic I/O Operation

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 have a high cost. Grid computers provide a more cost-effective alternative. They combine a large number of 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 the central control and the organs which perform it form the second specific part: CC

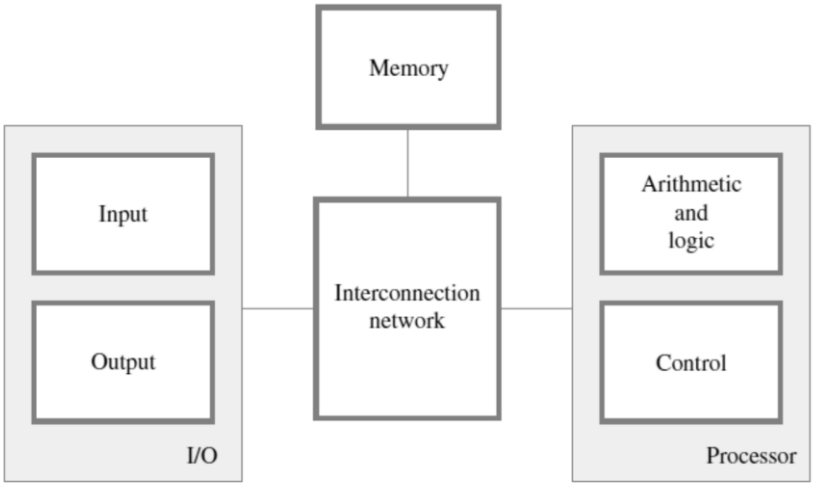
Third: Any device which is to carry out long and complicated sequences of operations (specifically of calculations) must have a considerable memory . . . At any rate, the total memory constitutes the third specific part of the device: M.

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: O.

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 the 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. Awide 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 the 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.

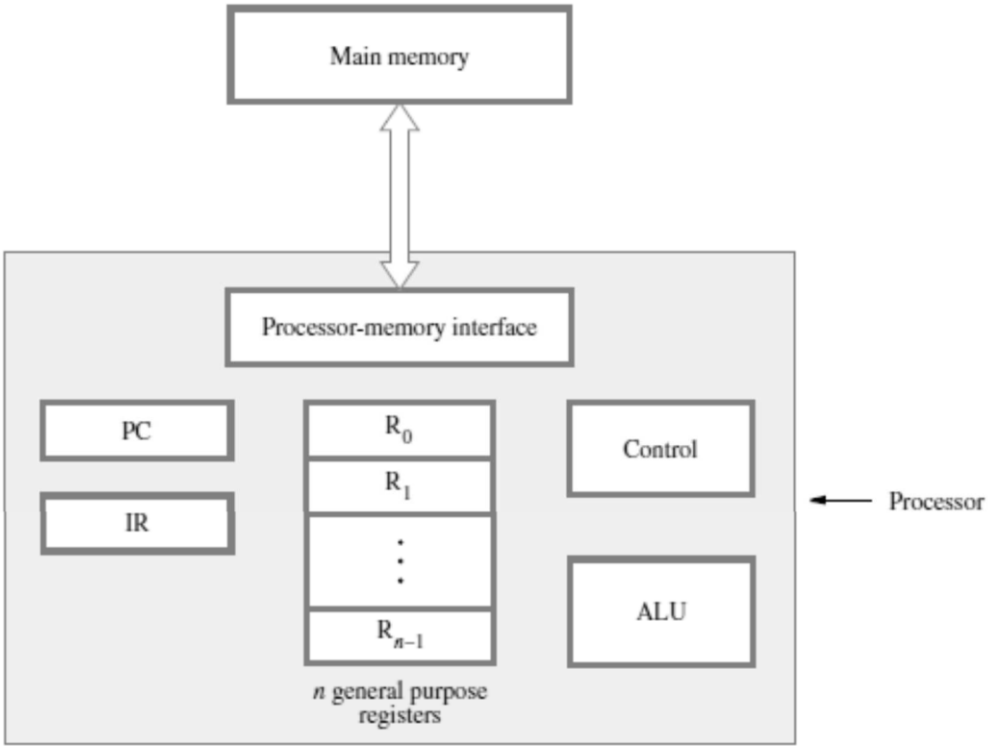
After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

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.

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.

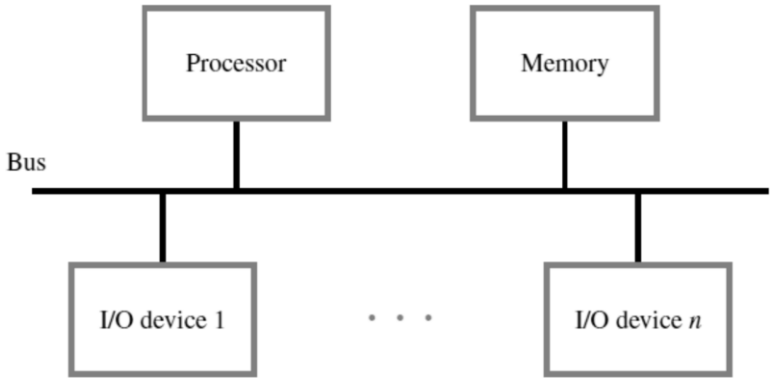
Instructions such as Load, Store, and Add perform data transfer and arithmetic operations. If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it 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.

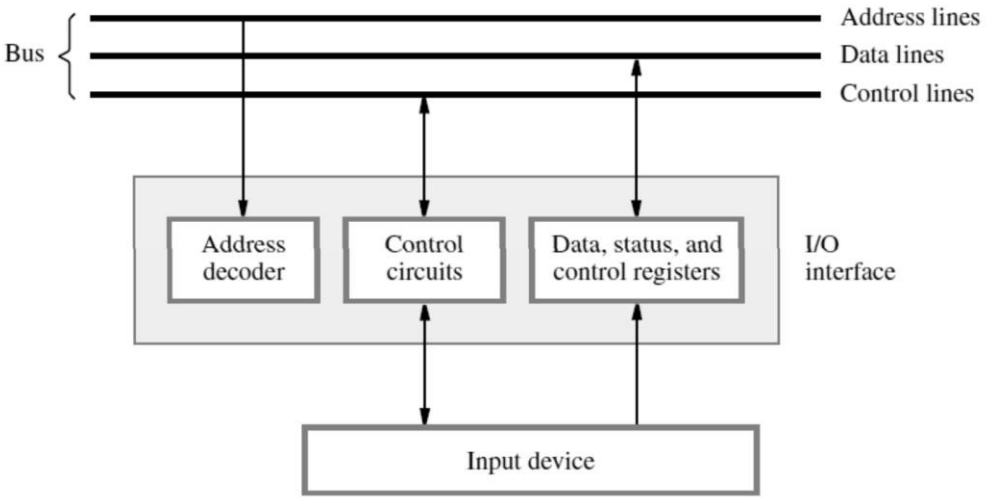
When I/O devices and the memory share the same address space, the arrangement is called memory-mapped I/O. Any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if the input device in Figure is a keyboard and if DATAIN is its data register,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

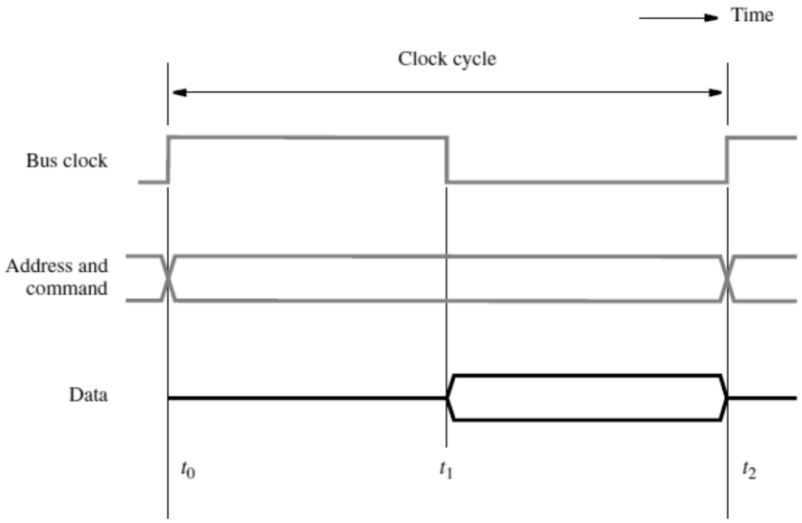
One control line, usually labelled R/W, specifies whether a Read or a Write operation is to be performed. As the label suggests, it specifies Read when set to 1 and Write when set to 0. When several data sizes are possible, such as byte, halfword, or word, the required size is indicated by other control 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 places the output data on the data lines when it transmits the address and command information. At time 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 state 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 actually 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.

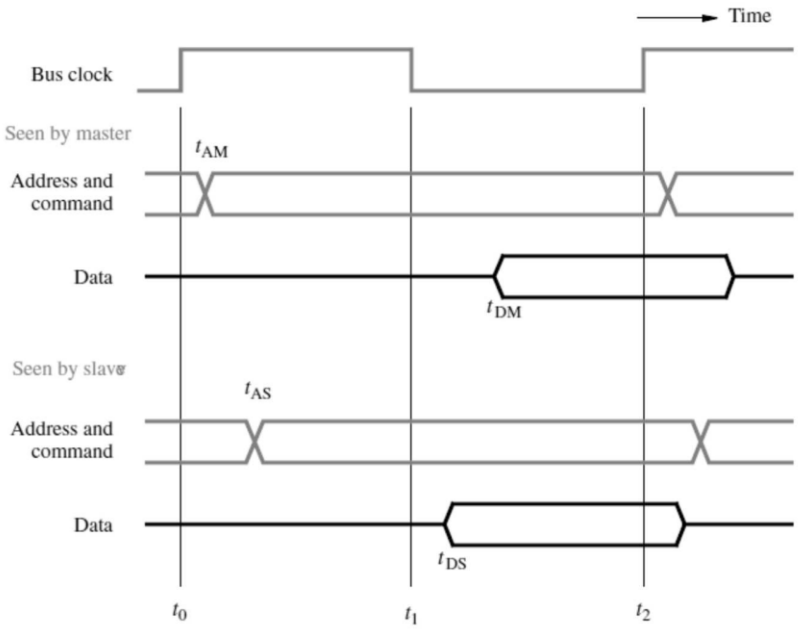
The master sends the address and command signals on the rising edge of the clock at the beginning of the clock cycle (at t0). However, these signals do not actually appear on the bus until tAM, largely due to the delay in the electronic circuit output from the master to the bus lines. A short while later, 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

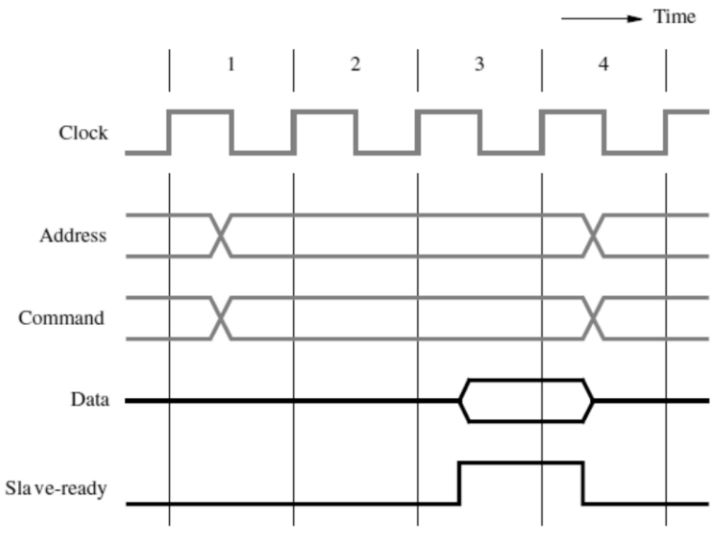
The scheme described above results in a simple design for the device interface. However, it has some limitations. Because a transfer has to be completed within one clock cycle, the clock period, t2 − t0, must be chosen to accommodate the longest delays on the bus and the slowest device interface. This 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.



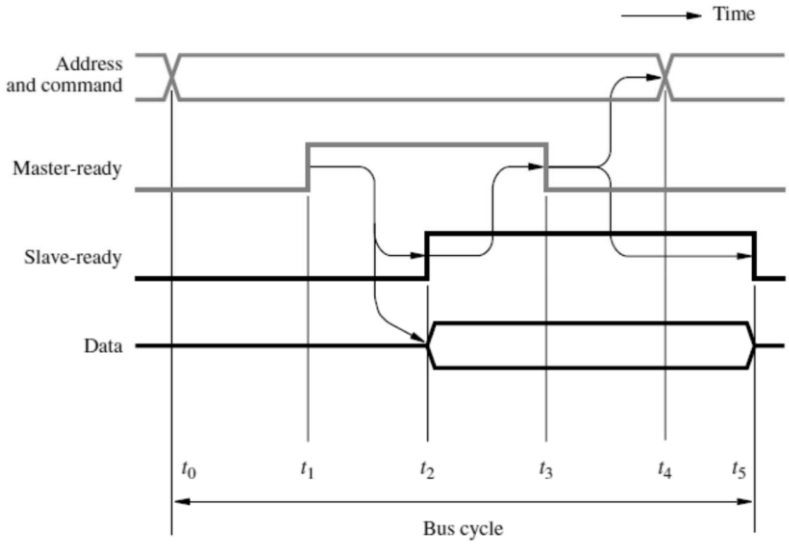
It begins to access the requested data on the active edge of the clock at the beginning of clock cycle 2. We have assumed that due to the delay involved in getting the data, the slave cannot respond immediately. The data become ready and are placed on the bus during clock

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 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.

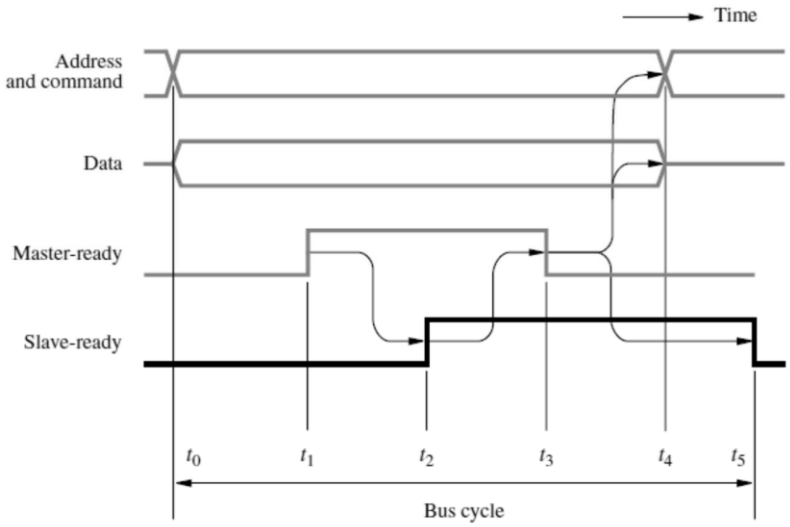
t2—The selected slave, having decoded the address and command information, performs the required input operation by placing its data on the data lines. At the same time, it sets the Slave-ready signal to 1. If extra delays are introduced by the interface circuitry before it places the data on the bus, 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 processor 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. Th e 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 oft en examine performance using different metrics. If we could relate these diff erent 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:

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

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

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

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 oft en 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

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

Th term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is oft en 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 effi ciency 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.

2. INSTRUCTIONS AND INSTRUCTION SEQUENCING

Objectives: you will learn about the Machine instructions and program execution, including branching and subroutine call and return operations.

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, registers R2 and R3 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

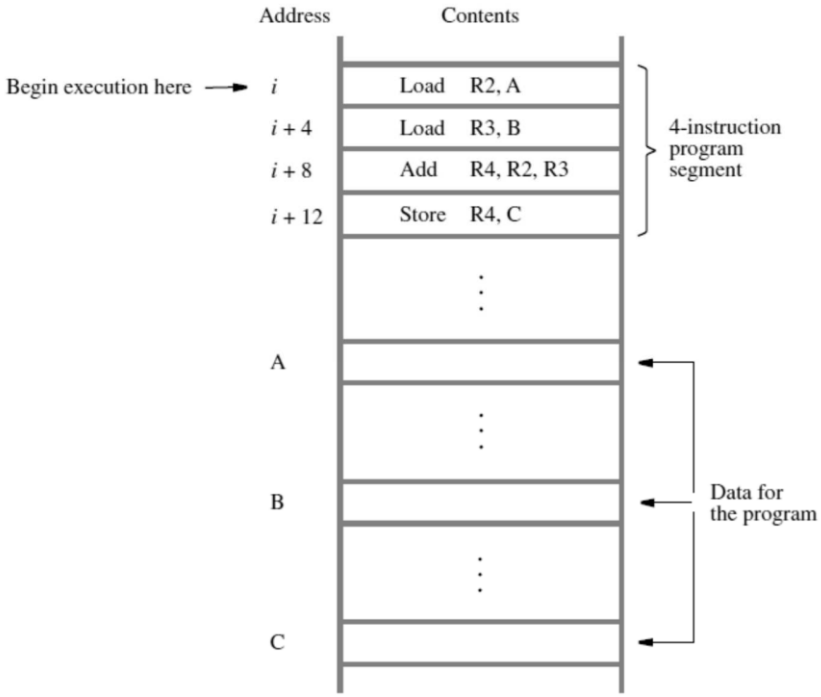
Add R3, R2, R3

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.

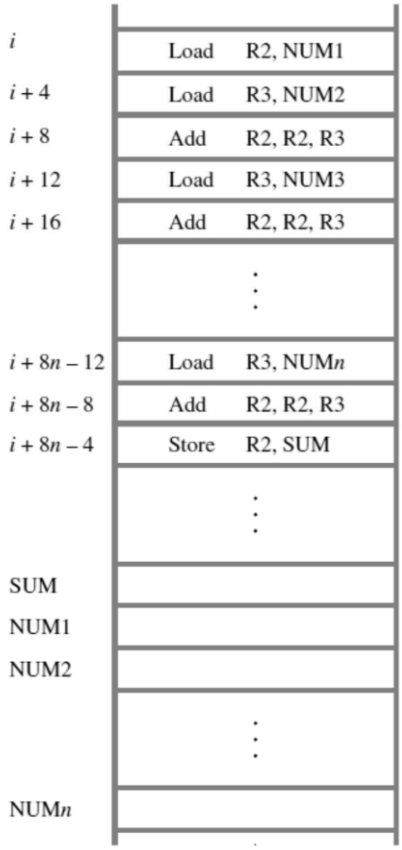
Branching

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 location LOOP 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 concentrate 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.



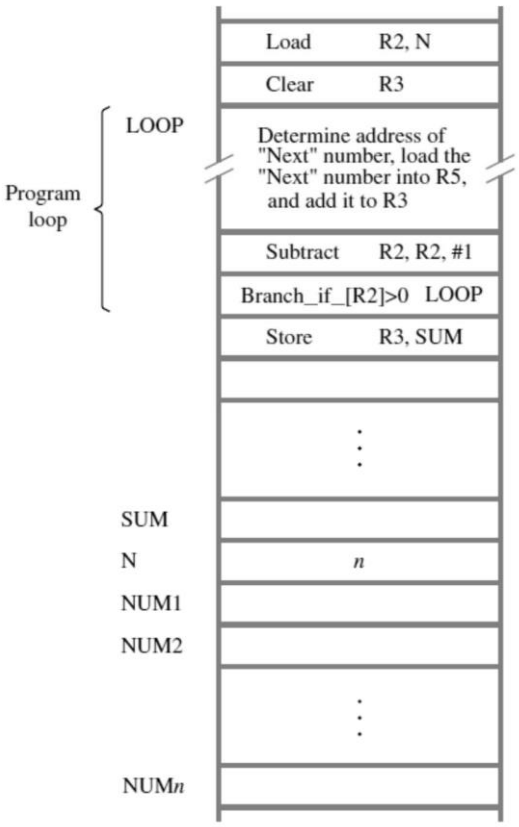
We now introduce branch instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the branch target, instead of the instruction at the location that follows the branch instruction in 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 capability 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.

.

3. 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

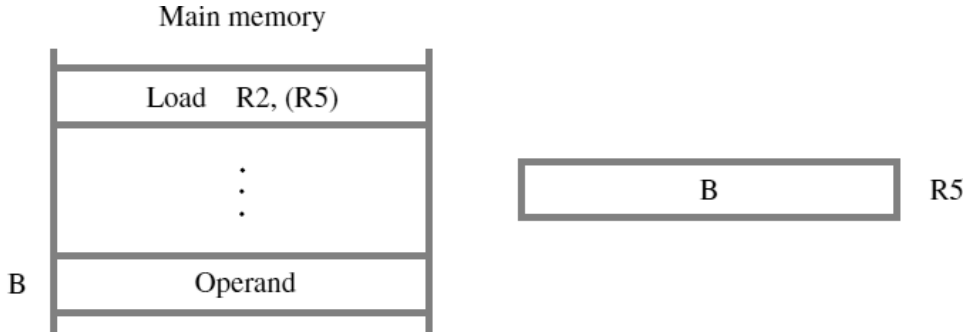
In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an effective address (EA) can be derived by the processor when the instruction is executed. The effective address is then used to access the operand.

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

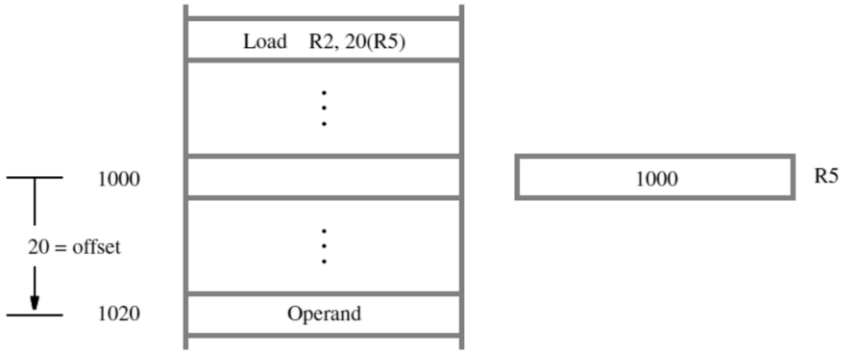
EA = X + [Ri]

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

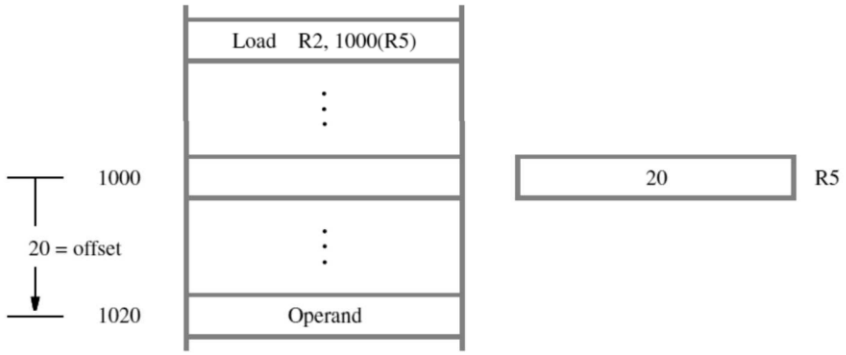
In an assembly-language program, whenever a constant such as the value X is needed, it may be given either as an explicit number or as a symbolic name representing a numerical value. When the instruction is translated into machine code, the constant X is given as a part of the instruction and is restricted 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.



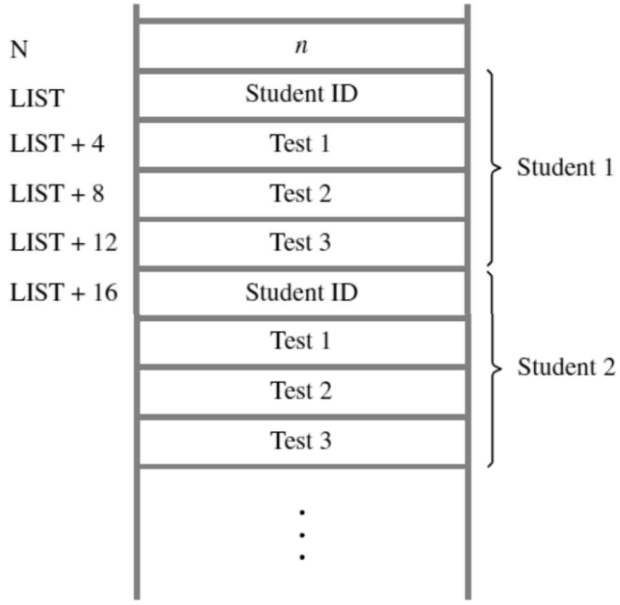
(a) Offset is given as a constant



(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 constantXand 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)

4. BASIC I/O OPERATION

In this chapter you will learn about:

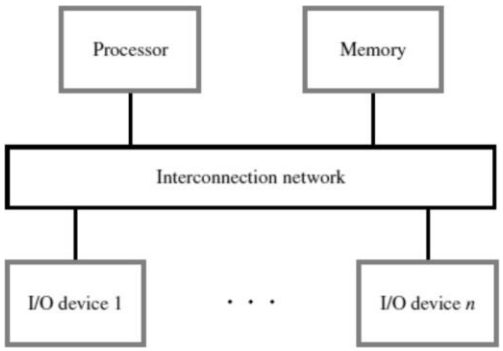
- Transferring data between a processor and input/output (I/O) devices
- The programmer's view of I/O transfers
- How program-controlled I/O is performed using polling

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.

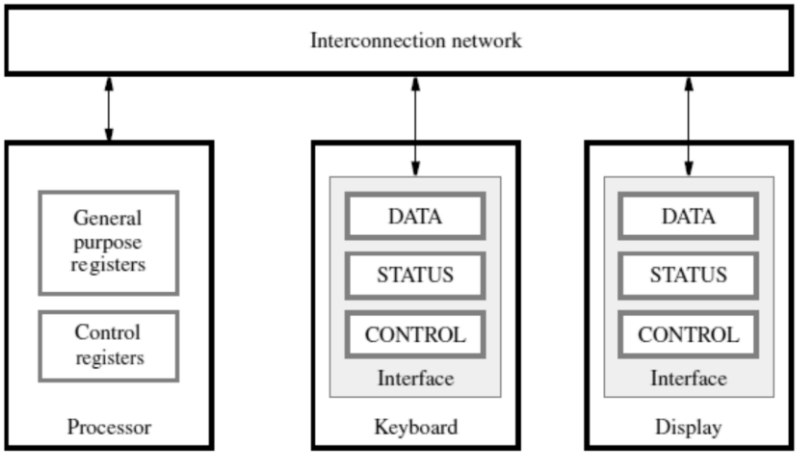
I/O Device Interface

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.

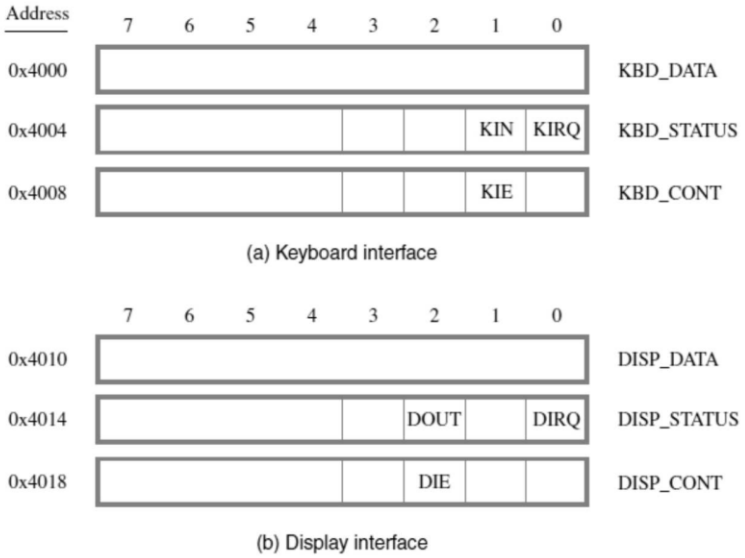
One solution to this problem involves a signalling protocol. On output, the processor sends the first character and then waits for a signal from the display that the next character can be sent. It then sends the second character, and so on. An input character is obtained from the keyboard in a similar way. The processor waits for a signal indicating 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_DATAbe 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 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_DATAregister. 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.

CONCLUSION:

This chapter has introduced basic concepts about the structure of computers and their operation. Machine instructions and programs have been described briefly. Much of the terminology needed to deal with these subjects has been defined. It introduced the representation and execution of instructions and programs at the assembly and machine level as seen by the programmer. The discussion emphasized the basic principles of addressing techniques and instruction sequencing. The programming examples illustrated the basic types of operations implemented by the instruction set of any modern computer. Commonly used addressing modes were introduced. In the discussion in this chapter, we provided the contrast

between two different approaches to the design of machine instruction sets—the RISC and CISC approaches.

In this chapter, we discussed two basic approaches to I/O transfers. The simplest technique is programmed I/O, in which the processor performs all of the necessary functions under direct control of program instructions. The second approach is based on the use of interrupts; this mechanism makes it possible to interrupt the normal execution of programs in order to service higher-priority requests that require more urgent attention.

COMPUTER ORGANIZATION

(Common to CSE and IT)

unit-3

THE MEMORY SYSTEM

- Basic concepts
- Semiconductor RAM
- ROM
- Speed ,Size and cost
- Cache memories
- Improving cache performance
- Virtual memory
- Mem
- Associative memories
- Secondary storage devices

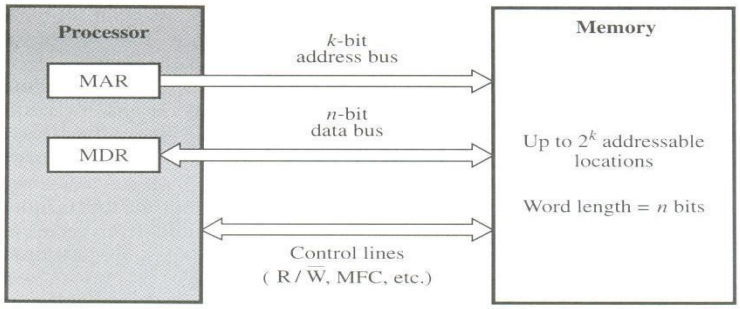
Performance consideration is here

BASIC CONCEPTS

The maximum size of the memory that can be used in any computer is determined by the addressing scheme.

Address	Memory Locations
16 Bit	2^{16} = 64 K
32 Bit	2^{32} = 4G (Giga)
40 Bit	2^{40} = IT (Tera)

Fig: Connection of Memory to Processor:



- If MAR is k bits long and MDR is n bits long, then the memory may contain upto 2^K addressable locations and the n-bits of data are transferred between the memory and processor.
- This transfer takes place over the processor bus.
- The processor bus has,
    - Address Line
    - Data Line
    - Control Line (R/W, MFC – Memory Function Completed)
- The control line is used for co-ordinating data transfer.
- The processor reads the data from the memory by loading the address of the required memory location into MAR and setting the R/W line to 1.
- The memory responds by placing the data from the addressed location onto the data lines and confirms this action by asserting MFC signal.
- Upon receipt of MFC signal, the processor loads the data onto the data lines into MDR register.
- The processor writes the data into the memory location by loading the address of this location into MAR and loading the data into MDR sets the R/W line to 0.

Memory Access Time → It is the time that elapses between the intiation of an

Operation and the completion of that operation.

Memory Cycle Time → It is the minimum time delay that required between the initiation of the two successive memory operations.

RAM (Random Access Memory):

In RAM, if any location that can be accessed for a Read/Write operation in fixed amount of time, it is independent of the location‟s address.

Cache Memory:

- It is a small, fast memory that is inserted between the larger slower main memory and the processor.
- It holds the currently active segments of a pgm and their data.

Virtual memory:

- The address generated by the processor does not directly specify the physical locations in the memory.
- The address generated by the processor is referred to as a virtual / logical address.
- The virtual address space is mapped onto the physical memory where data are actually stored.
- The mapping function is implemented by a special memory control circuit is often called the memory management unit.
- Only the active portion of the address space is mapped into locations in the physical memory.
- The remaining virtual addresses are mapped onto the bulk storage devices used, which are usually magnetic disk.
- As the active portion of the virtual address space changes during program execution, the memory management unit changes the mapping function and transfers the data between disk and memory.
- Thus, during every memory cycle, an address processing mechanism determines whether the addressed in function is in the physical memory unit.
- If it is, then the proper word is accessed and execution proceeds.
- If it is not, a page of words containing the desired word is transferred from disk to memory.
- This page displaces some page in the memory that is currently inactive.

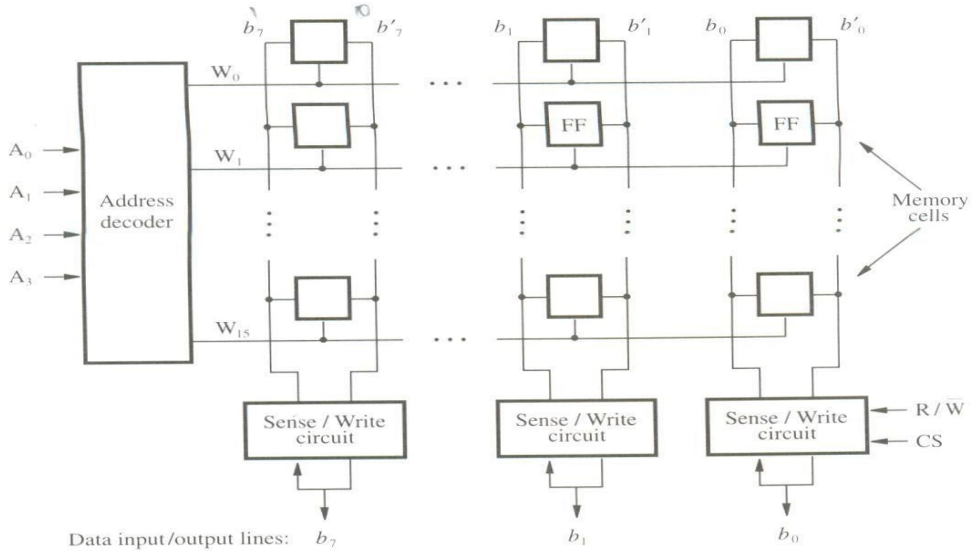
SEMI CONDUCTOR RAM MEMORIES:

- Semi-Conductor memories are available is a wide range of speeds.
- Their cycle time ranges from 100ns to 10ns

INTERNAL ORGANIZATION OF MEMORY CHIPS:

- Memory cells are usually organized in the form of array, in which each cell is capable of storing one bit of in formation.
- Each row of cells constitute a memory word and all cells of a row are connected to a common line called as **word line**.
- The cells in each column are connected to Sense / Write circuit by two bit lines.
- The Sense / Write circuits are connected to data input or output lines of the chip.
- During a write operation, the sense / write circuit receive input information and store it in the cells of the selected word.

Fig: Organization of bit cells in a memory chip



- The data input and data output of each senses / write ckt are connected to a single bidirectional data line that can be connected to a data bus of the cptr.

  R / W → Specifies the required operation.

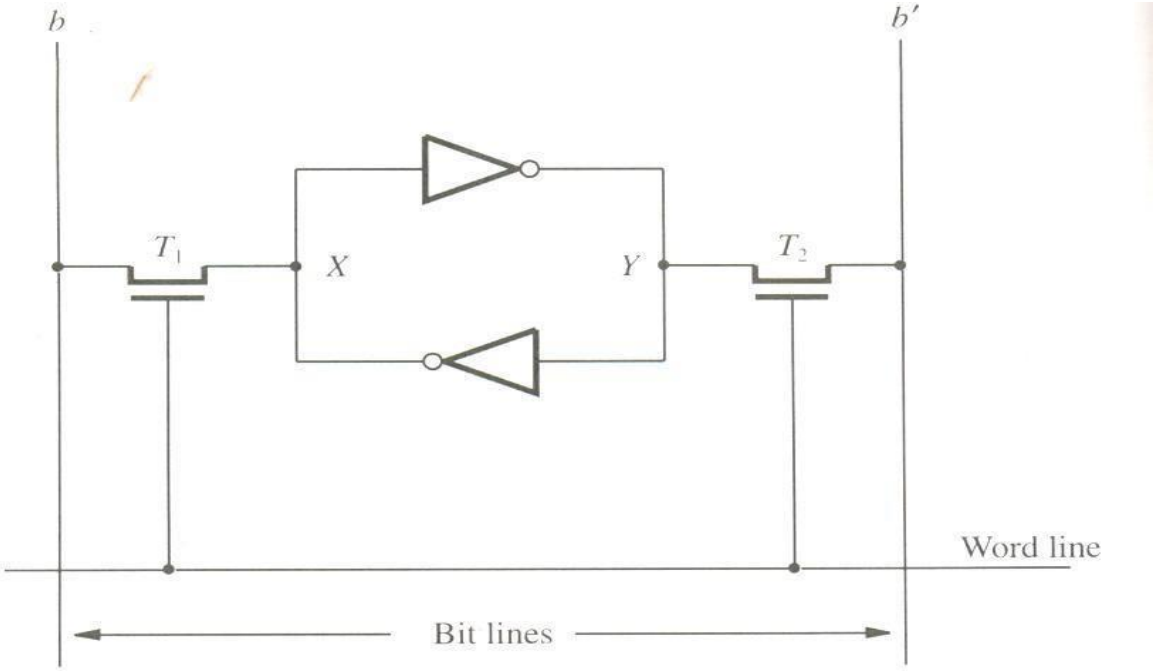
  CS → **Chip Select** input selects a given chip in the multi-chip memory system

Bit Organization	**Requirement of external connection for address, data and control lines**
128 (16x8)	14
(1024) 128x8(1k)	19

Static Memories:

Memories that consists of circuits capable of retaining their state as long as power is applied are known as **static memory.**

Fig:Static RAM cell



- Two inverters are cross connected to form a batch
- The batch is connected to two bit lines by transistors T_1 and T_2.
- These transistors act as switches that can be opened / closed under the control of the word line.
- When the wordline is at ground level, the transistors are turned off and the latch retain its state.

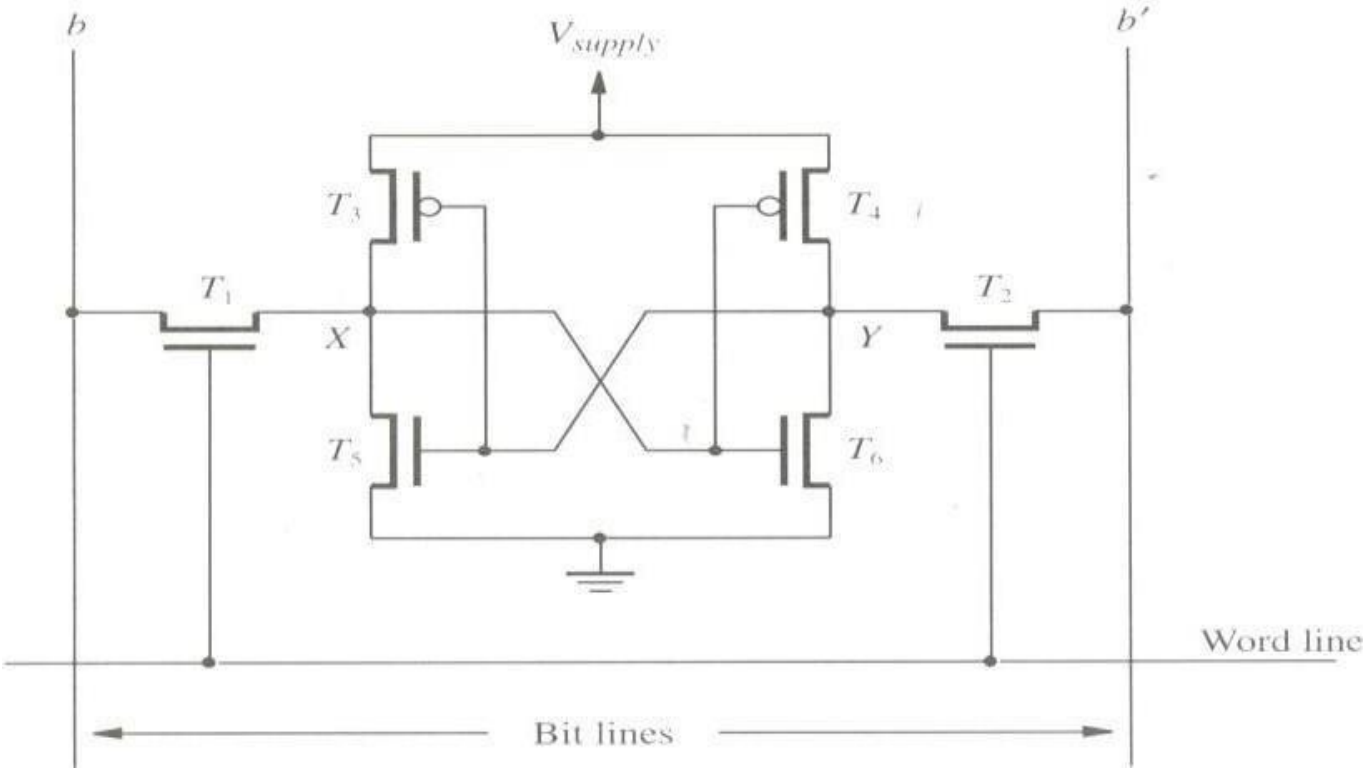
Read Operation:

- In order to read the state of the SRAM cell, the word line is activated to close switches T_1 and T_2.
- If the cell is in state 1, the signal on bit line b is high and the signal on the bit line b is low.Thus b and b are complement of each other.
- Sense / write circuit at the end of the bit line monitors the state of b and b‟ and set the output accordingly.

Write Operation:

- The state of the cell is set by placing the appropriate value on bit line b and its complement on b and then activating the word line. This forces the cell into the corresponding state.
- The required signal on the bit lines are generated by Sense / Write circuit.

Fig:CMOS cell (Complementary Metal oxide Semi Conductor):



- Transistor pairs (T_3, T_5) and (T_4, T_6) form the inverters in the latch.
- In state 1, the voltage at point X is high by having T_5, T_6 on and T_4, T_5 are OFF.
- Thus T_1, and T_2 returned ON (Closed), bit line b and b will have high and low signals respectively.
- The CMOS requires 5V (in older version) or 3.3.V (in new version) of power supply voltage.
- The continuous power is needed for the cell to retain its state

Merit :

- It has low power consumption because the current flows in the cell only when the cell is being activated accessed.
- Static RAM‟s can be accessed quickly. It access time is few nanoseconds.

Demerit:

- SRAM‟s are said to be volatile memories because their contents are lost when the power is interrupted.

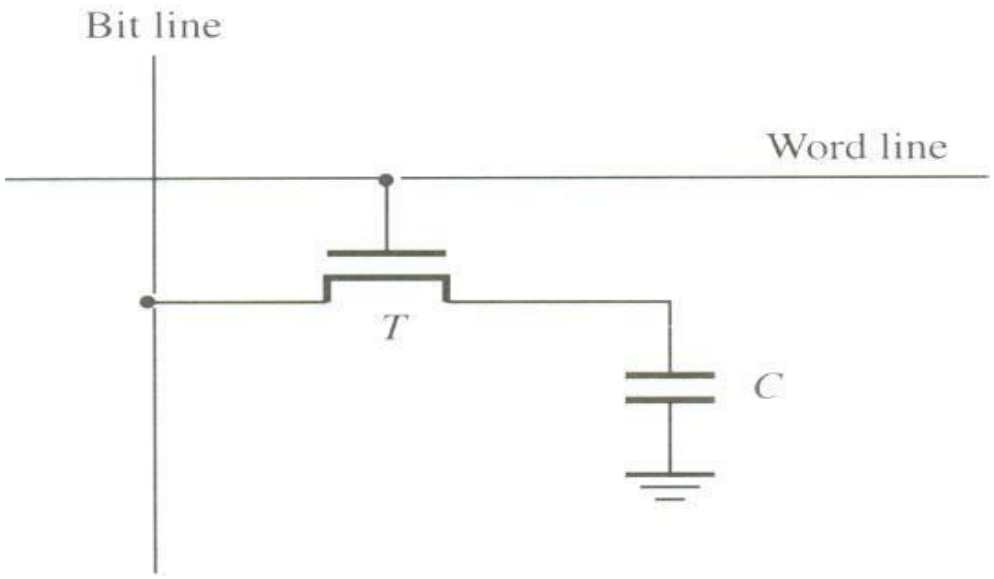
Asynchronous DRAMS:-

- Less expensive RAM‟s can be implemented if simplex calls are used such cells cannot retain their state indefinitely. Hence they are called **Dynamic RAM's (DRAM).**
- The information stored in a dynamic memory cell in the form of a charge on a capacitor and this charge can be maintained only for tens of Milliseconds.
- The contents must be periodically refreshed by restoring by restoring this

capacitor charge to its full value.

-

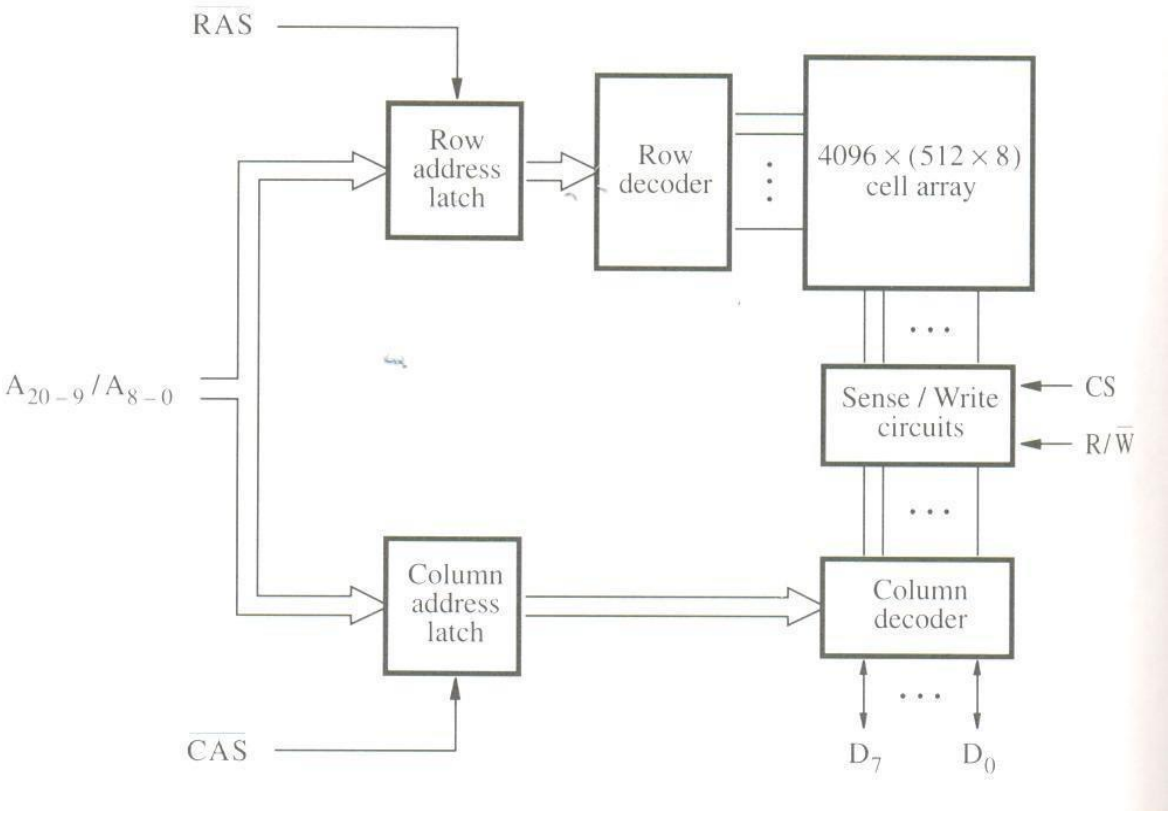
Fig:A single transistor dynamic Memory cell



- In order to store information in the cell, the transistor T is turned „on" & the appropriate voltage is applied to the bit line, which charges the capacitor.
- After the transistor is turned off, the capacitor begins to discharge which is caused by the capacitor"s own leakage resistance.
- Hence the information stored in the cell can be retrieved correctly before the threshold value of the capacitor drops down.
- During a read operation, the transistor is turned „on" & a sense amplifier connected to the bit line detects whether the charge on the capacitor is above the threshold value.

  If charge on capacitor > threshold value -> Bit line will have logic value „1".
  If charge on capacitor < threshold value -> Bit line will set to logic value „0".

Fig:Internal organization of a 2M X 8 dynamic Memory chip.



DESCRIPTION:

- The 4 bit cells in each row are divided into 512 groups of 8.
- 21 bit address is needed to access a byte in the memory(12 bit→To select a row,9 bit→Specify the group of 8 bits in the selected row).

  A_{8-0} →Row address of a byte.
  A_{20-9} →Column address of a byte.

- During Read/ Write operation ,the row address is applied first. It is loaded into the row address latch in response to a signal pulse on **Row Address Strobe(RAS)** input of the chip.
- When a Read operation is initiated, all cells on the selected row are read and refreshed.
- Shortly after the row address is loaded,the column address is applied to the address pins & loaded into **Column Address Strobe(CAS).**
- The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits are selected.
- R/W =1(read operation)→The output values of the selected circuits are transferred to the data lines D0 - D7.
- R/W =0(write operation)→The information on D0 - D7 are transferred to the selected circuits.

- RAS and CAS are active low so that they cause the latching of address when they change from high to low. This is because they are indicated by RAS & CAS.
- To ensure that the contents of a DRAM „s are maintained, each row of cells must be accessed periodically.
- Refresh operation usually perform this function automatically.
- A specialized memory controller circuit provides the necessary control signals RAS & CAS, that govern the timing.
- The processor must take into account the delay in the response of the memory. Such memories are referred to as **Asynchronous DRAM's.**

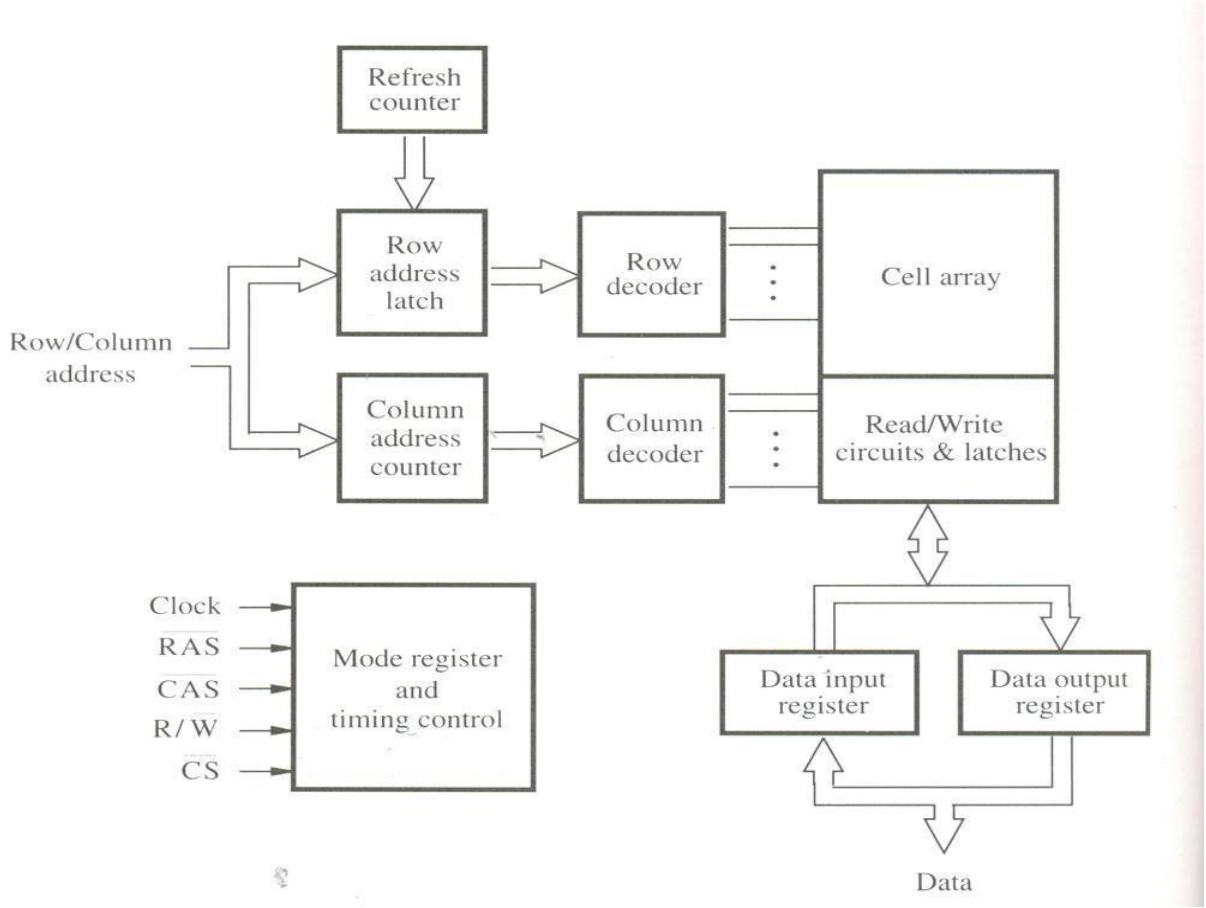
Fast Page Mode:

- Transferring the bytes in sequential order is achieved by applying the consecutive sequence of column address under the control of successive CAS signals.
- This scheme allows transferring a block of data at a faster rate. The block of transfer capability is called as **Fast Page Mode.**

Synchronous DRAM:

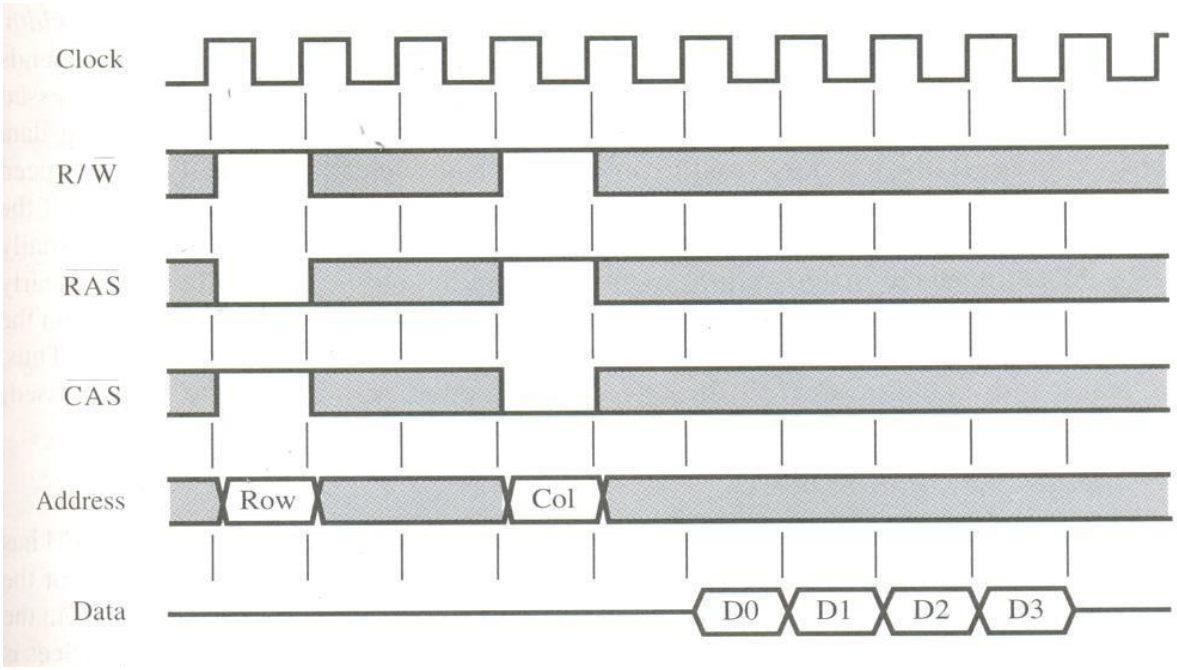
- Here the operations e directly synchronized with clock signal.
- The address and data connections are buffered by means of registers.
- The output of each sense amplifier is connected to a latch.
- A Read operation causes the contents of all cells in the selected row to be loaded in these latches.

Fig:Synchronous DRAM



- Data held in the latches that correspond to the selected columns are transferred into the data output register, thus becoming available on the data output pins.

Fig:Timing Diagram →Burst Read of Length 4 in an SDRAM



- First ,the row address is latched under control of RAS signal.
- The memory typically takes 2 or 3 clock cycles to activate the selected row.
- Then the column address is latched under the control of 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 3 sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

Latency & Bandwidth:

- A good indication of performance is given by two parameters.They are,
  - Latency
  - Bandwidth

Latency:

- It refers to the amount of time it takes to transfer a word of data to or from the memory.
- For a transfer of single word,the latency provides the complete indication of memory performance.
- For a block transfer,the latency denote the time it takes to transfer the first word of data.

Bandwidth:

- It is defined as the number of bits or bytes that can be transferred in one second.
- Bandwidth mainly depends upon the speed of access to the stored data & on the number of bits that can be accessed in parallel.

Double Data Rate SDRAM(DDR-SDRAM):

- The standard SDRAM performs all actions on the rising edge of the clock signal.
- The double data rate SDRAM transfer data on both the edges(loading edge, trailing edge).
- The Bandwidth of DDR-SDRAM is doubled for long burst transfer.
- To make it possible to access the data at high rate , the cell array is organized into two banks.
- Each bank can be accessed separately.
- Consecutive words of a given block are stored in different banks.
- Such interleaving of words allows simultaneous access to two words that are transferred on successive edge of the clock.

Larger Memories:

Dynamic Memory System:

- The physical implementation is done in the form of Memory Modules.
- If a large memory is built by placing DRAM chips directly on the main system printed circuit board that contains the processor ,often referred to as Motherboard;it will occupy large amount of space on the board.
- These packaging consideration have led to the development of larger memory units known as SIMM‟s & DIMM‟s .

  SIMM-Single Inline memory Module
  DIMM-Dual Inline memory Module

- SIMM & DIMM consists of several memory chips on a separate small board that plugs vertically into single socket on the motherboard.

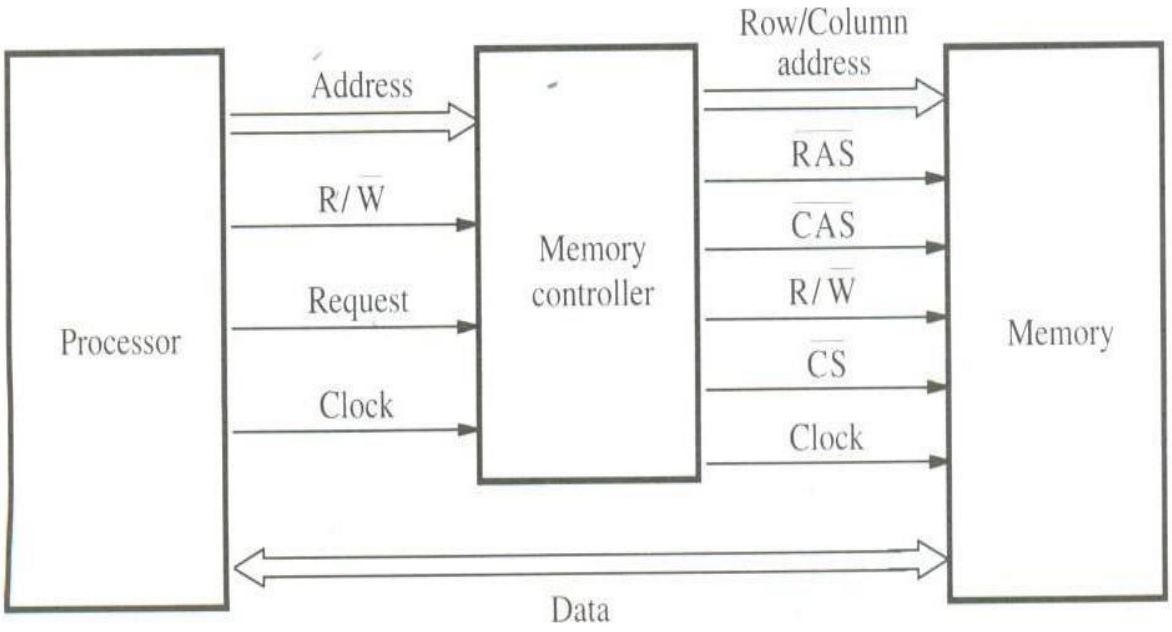
MEMORY SYSTEM CONSIDERATION:

- To reduce the number of pins, the dynamic memory chips use multiplexed address inputs.
- The address is divided into two parts.They are,

  - **High Order Address Bit**(Select a row in cell array & it is provided first and latched into memory chips under the control of RAS signal).
  - **Low Order Address Bit**(Selects a column and they are provided on same address pins and latched using CAS signals).

- The Multiplexing of address bit is usually done by **Memory Controller Circuit.**

Fig:Use of Memory Controller



- The Controller accepts a complete address & R/W signal from the processor, under the control of a Request signal which indicates that a memory access operation is needed.
- The Controller then forwards the row & column portions of the address to the memory and generates RAS &CAS signals.
- It also sends R/W &CS signals to the memory.
- The CS signal is usually active low, hence it is shown as CS.

Refresh

- All dynamic memories have to be refreshed.
- In DRAM ,the period for refreshing all rows is 16ms whereas 64ms in SDRAM.

Eg:Given a cell array of 8K(8192).

Clock cycle=4
Clock Rate=133MHZ
No of cycles to refresh all rows =8192*4
=32,768
Time needed to refresh all rows=$32768/133*10^6$
=$246*10^{-6}$ sec
=0.246sec
Refresh Overhead =0.246/64
Refresh Overhead =0.0038

Rambus Memory:

- The usage of wide bus is expensive.
-
-

Rambus developed the implementation of narrow bus.

Rambus technology is a fast signaling method used to transfer information between chips.

- Instead of using signals that have voltage levels of either 0 or V_{supply} to represent the logical values, the signals consists of much smaller voltage swings around a reference voltage $V_{ref.}$
- .The reference Voltage is about 2V and the two logical values are represented by 0.3V swings above and below $V_{ref..}$
- This type of signaling is generally is known as **Differential Signalling**.
- Rambus provides a complete specification for the design of communication links(**Special Interface circuits**) called as **Rambus Channel**.
- Rambus memory has a clock frequency of 400MHZ.
- The data are transmitted on both the edges of the clock so that the effective data transfer rate is 800MHZ.
- The circuitry needed to interface to the Rambus channel is included on the chip.Such chips are known as Rambus DRAM‟s(RDRAM).
- Rambus channel has,

    - 9 Data lines(1-8→Transfer the data,9^{th} line→Parity checking).
    - Control line
    - Power line

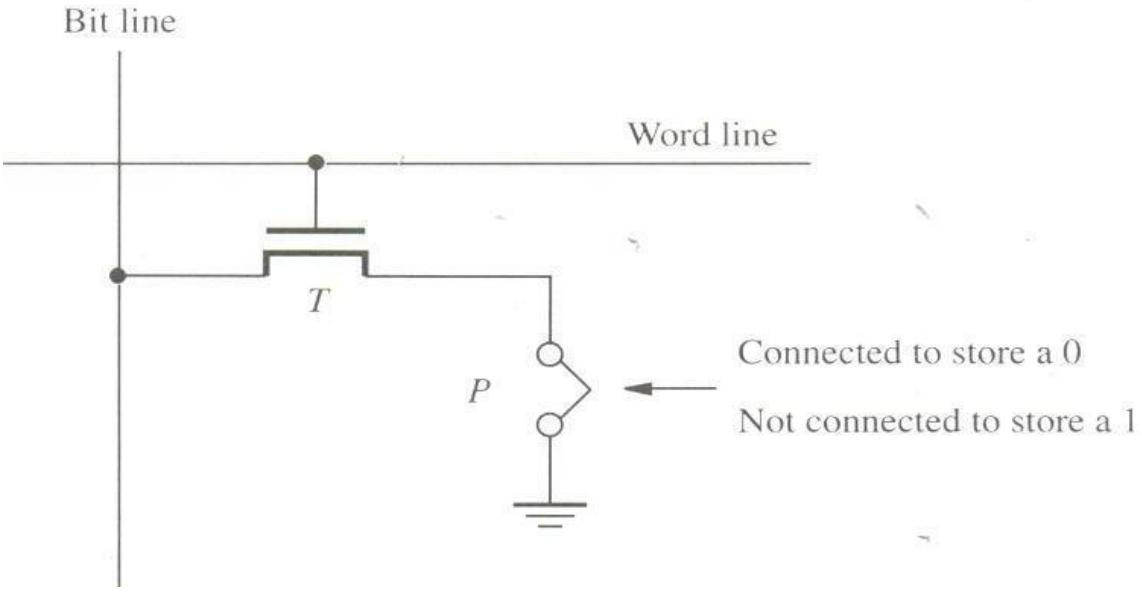
- A two channel rambus has 18 data lines which has no separate address lines.It is also called as **Direct RDRAM's.**
- Communication between processor or some other device that can serves as a master and RDRAM modules are serves as slaves ,is carried out by means of packets transmitted on the data lines.
- There are 3 types of packets.They are,

    - Request
    - Acknowledge
    - Data

READ ONLY MEMORY:

- Both SRAM and DRAM chips are volatile,which means that they lose the stored information if power is turned off.
- Many application requires Non-volatile memory (which retain the stored information if power is turned off).
- Eg:Operating System software has to be loaded from disk to memory which requires the program that boots the Operating System ie. It requires non-volatile memory.
- Non-volatile memory is used in embedded system.
- Since the normal operation involves only reading of stored data ,a memory of this type is called ROM.

Fig:ROM cell



At Logic value '0' → Transistor(T) is connected to the ground point(P).
Transistor switch is closed & voltage on bitline nearly drops to zero.
At Logic value '1' → Transistor switch is open.
The bitline remains at high voltage.

- To read the state of the cell,the word line is activated.
- A Sense circuit at the end of the bitline generates the proper output value.

Types of ROM:

- Different types of non-volatile memory are,
  - PROM
  - EPROM
  - EEPROM
  - Flash Memory

PROM:-Programmable ROM:

- PROM allows the data to be loaded by the user.
- Programmability is achieved by inserting a „fuse" at point P in a ROM cell.
- Before it is programmed, the memory contains all 0"s
- The user can insert 1"s at the required location by burning out the fuse at these locations using high-current pulse.
- This process is irreversible.

Merit: It provides flexibility.

- It is faster.
- It is less expensive because they can be programmed directly by the user.
-

EPROM:-Erasable reprogrammable ROM:

- EPROM allows the stored data to be erased and new data to be loaded.
- In an EPROM cell, a connection to ground is always made at „P" and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned „off".
- This transistor can be programmed to behave as a permanently open switch, by injecting charge into it that becomes trapped inside.
- Erasure requires dissipating the charges trapped in the transistor of memory cells. This can be done by exposing the chip to ultra-violet light, so that EPROM chips are mounted in packages that have transparent windows.

Merits:

- It provides flexibility during the development phase of digital system.
- It is capable of retaining the stored information for a long time.

Demerits:

- The chip must be physically removed from the circuit for reprogramming and its entire contents are erased by UV light.

EEPROM:-Electrically Erasable ROM:

Merits:

- It can be both programmed and erased electrically.
- It allows the erasing of all cell contents selectively.

Demerits:

- It requires different voltage for erasing ,writing and reading the stored data.

Flash Memory:

- In EEPROM, it is possible to read & write the contents of a single cell.
- In Flash device, it is possible to read the contents of a single cell but it is only possible to write the entire contents of a block.
- Prior to writing,the previous contents of the block are erased.
- Eg.In MP3 player,the flash memory stores the data that represents sound.
- Single flash chips cannot provide sufficient storage capacity for embedded system application.
- There are 2 methods for implementing larger memory modules consisting of number of chips.They are,
  - Flash Cards
  - Flash Drives.

Merits:

- Flash drives have greater density which leads to higher capacity & low cost per bit.
- It requires single power supply voltage & consumes less power in their operation.

Flash Cards:

- One way of constructing larger module is to mount flash chips on a small card.
- Such flash card have standard interface.

- The card is simply plugged into a conveniently accessible slot.
- Its memory size are of 8,32,64MB.
- Eg:A minute of music can be stored in 1MB of memory. Hence 64MB flash cards can store an hour of music.

Flash Drives:

- Larger flash memory module can be developed by replacing the hard disk drive.
- The flash drives are designed to fully emulate the hard disk.
- The flash drives are solid state electronic devices that have no movable parts.

Merits:

- They have shorter seek and access time which results in faster response.
- They have low power consumption which makes them attractive for battery driven application.
- They are insensitive to vibration.

Demerit:

- The capacity of flash drive (<1GB) is less than hard disk(>1GB).
- It leads to higher cost perbit.
- Flash memory will deteriorate after it has been written a number of times(typically atleast 1 million times.)

SPEED,SIZE COST:

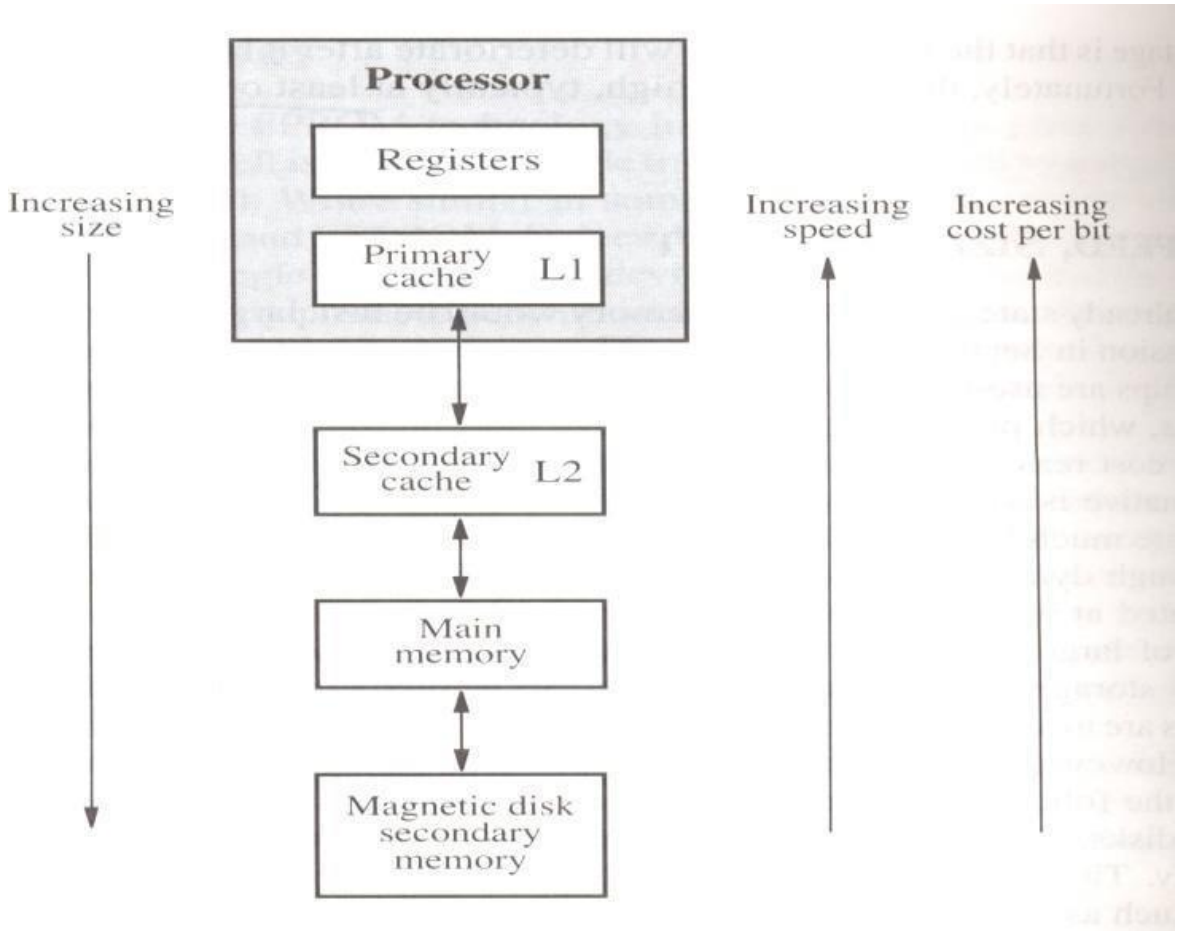
Characteristics	SRAM	DRAM	Magnetis Disk
speed	Very fast	slower	Much slower than DRAM
size	large	smaller	smaller
cost	Expensive	Less Expensive	Lower

Magnetic Disk:

- A 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

Fig:Memory Hierarchy



Types of Cache Memory:

- The Cache memory is of 2 types.They are,
  - Primary /Processor Cache(Level1 or L1 cache)
  - Secondary Cache(Level2 or L2 cache)

Primary Cache → It is always located on the processor chip.
Secondary Cache→It is placed between the primary cache and the rest of the memory.

- The main memory is implemented using the dynamic components(**SIMM,RIMM,DIMM**).
- The access time for main memory is about 10 times longer than the access time for L1 cache.

CACHE MEMORIES

- The effectiveness of cache mechanism is based on the property of „**Locality of reference'.**

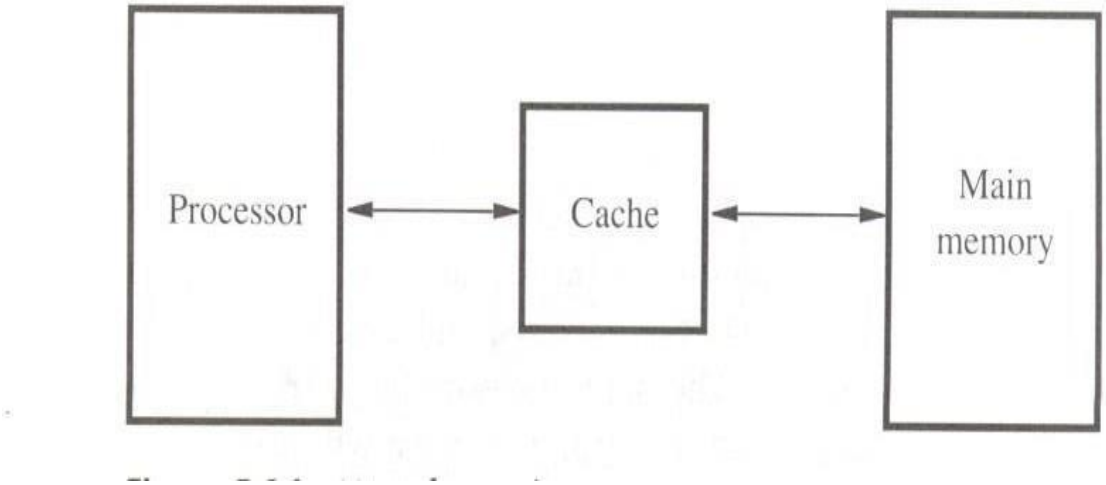
Locality of Reference:

- Many instructions in the localized areas of the program are executed repeatedly during some time period and remainder of the program is accessed relatively infrequently.

- It manifests itself in 2 ways.They are,
  - **Temporal**(The recently executed instruction are likely to be executed again very soon.)
  - **Spatial(**The instructions in close proximity to recently executed instruction are also likely to be executed soon.)
- If the active segment of the program is placed in cache memory, then the total execution time can be reduced significantly.

- The term Block refers to the set of contiguous address locations of some size.
- The cache line is used to refer to the cache block.

Fig:Use of Cache Memory



- The Cache memory stores a reasonable number of blocks at a given time but this number is small compared to the total number of blocks available in Main Memory.
- The correspondence between main memory block and the block in cache memory is specified by a mapping function.
- The Cache control hardware decide that which block should be removed to create space for the new block that contains the referenced word.
- The collection of rule for making this decision is called the **replacement algorithm.**
- The cache control circuit determines whether the requested word currently exists in the cache.
- If it exists, then Read/Write operation will take place on appropriate cache location. In this case **Read/Write hit** will occur.
- In a Read operation, the memory will not involve.
- The write operation is proceed in 2 ways.They are,
    - Write-through protocol
    - Write-back protocol

Write-through protocol:

- Here the cache location and the main memory locations are updated simultaneously.

Write-back protocol:

- This technique is to update only the cache location and to mark it as with associated flag bit called **dirty/modified bit.**
- The word in the main memory will be updated later,when the block containing this marked word is to be removed from the cache to make room for a new block.
- If the requested word currently not exists in the cache during read operation,then **read miss** will occur.

- To overcome the read miss **Load –through / Early restart protocol** is used.

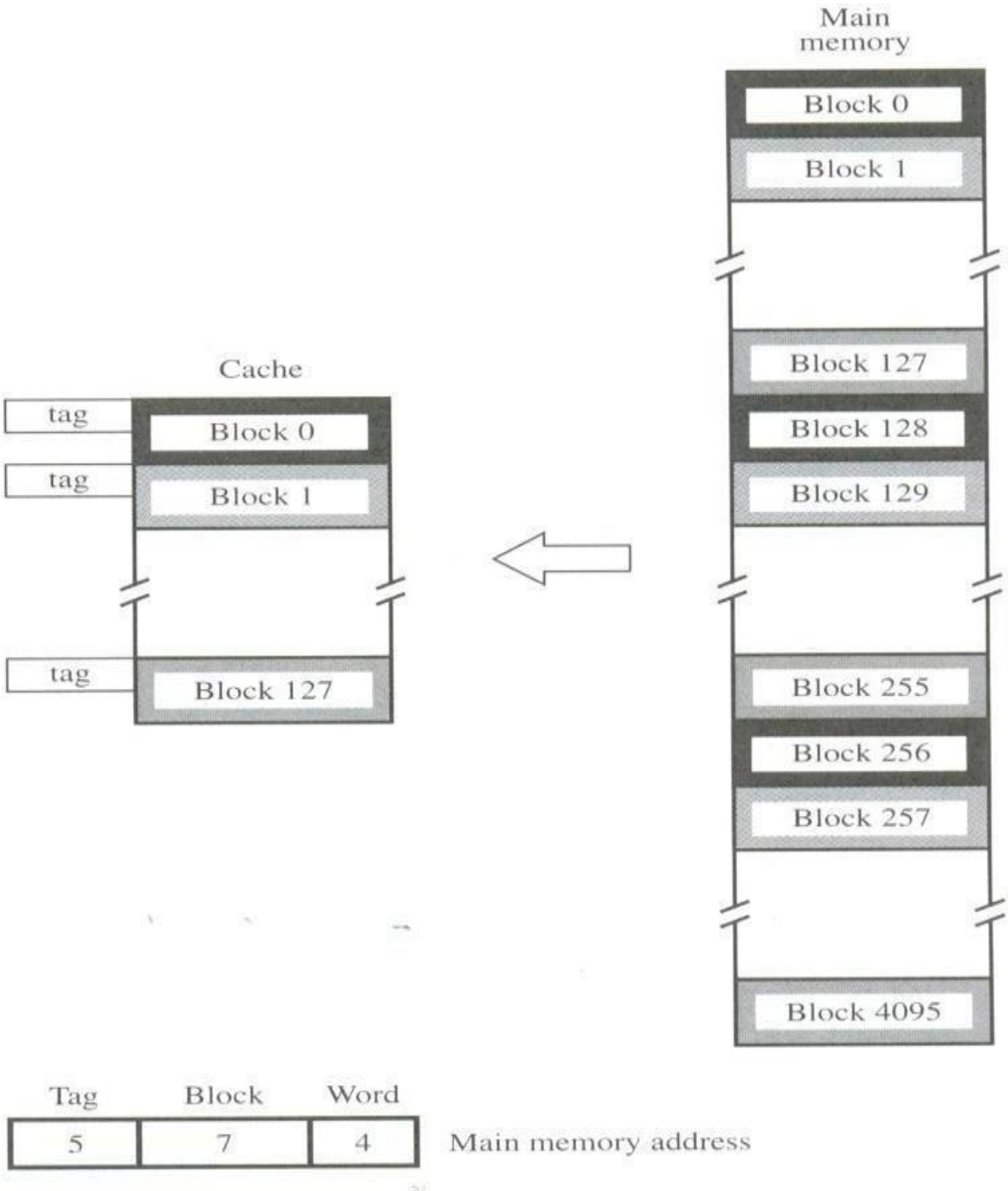
Read Miss:
The block of words that contains the requested word is copied from the main memory into cache.
Load –through:

- After the entire block is loaded into cache,the particular word requested is forwarded to the processor.
- If the requested word not exists in the cache during write operation,then **Write Miss** will occur.
- If Write through protocol is used,the information is written directly into main memory.
- If Write back protocol is used then block containing the addressed word is first brought intothe cache and then the desired word in the cache is over-written with the new information.

Mapping Function:
Direct Mapping:

- It is the simplest technique in which block j of the main memory maps onto block „j" modulo 128 of the cache.
- Thus whenever one of the main memory blocks 0,128,256 is loaded in the cache,it is stored in block 0.
- Block 1,129,257 are stored in cache block 1 and so on.
- The contention may arise when,
  - When the cache is full
  - When more than one memory block is mapped onto a given cache block position.
- The contention is resolved by allowing the new blocks to overwrite the currently resident block.
- Placement of block in the cache is determined from memory address.

Fig: Direct Mapped Cache



- The memory address is divided into 3 fields.They are,

  Low Order 4 bit field(word)→Selects one of 16 words in a block.
  7 bit cache block field→When new block enters cache,7 bit determines the cache position in which this block must be stored.
  5 bit Tag field→The high order 5 bits of the memory address of the block is stored in 5 tag bits associated with its location in the cache.
- As execution proceeds, the high order 5 bits of the address is compared with 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 be first read from the main memory and loaded into the cache.

Merit:

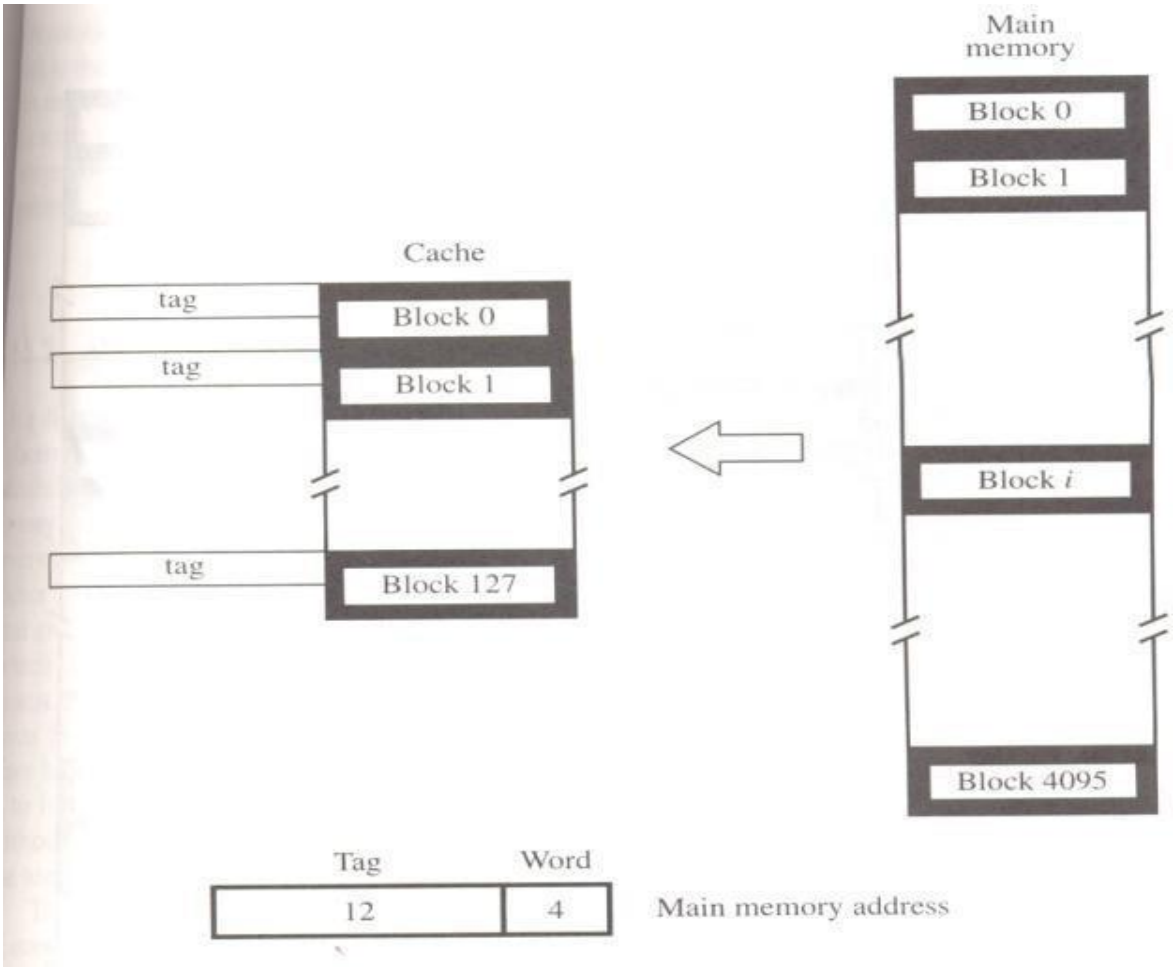
- It is easy to implement.

Demerit:

- It is not very flexible.

Associative Mapping:
In this method, the main memory block can be placed into any cache block position.

Fig:Associative Mapped Cache.



- 12 tag bits will identify a memory block when it is resolved 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 persent.This is called **associative mapping.**
- **I**t gives complete freedom in choosing the cache location.
- A new block that has to be brought into the cache has to replace(eject)an existing block if the cache is full.

- In this method,the memory has to determine whether a given block is in the cache.
- A search of this kind is called an **associative Search.**

Merit: It is more flexible than direct mapping technique.

-

Demerit:

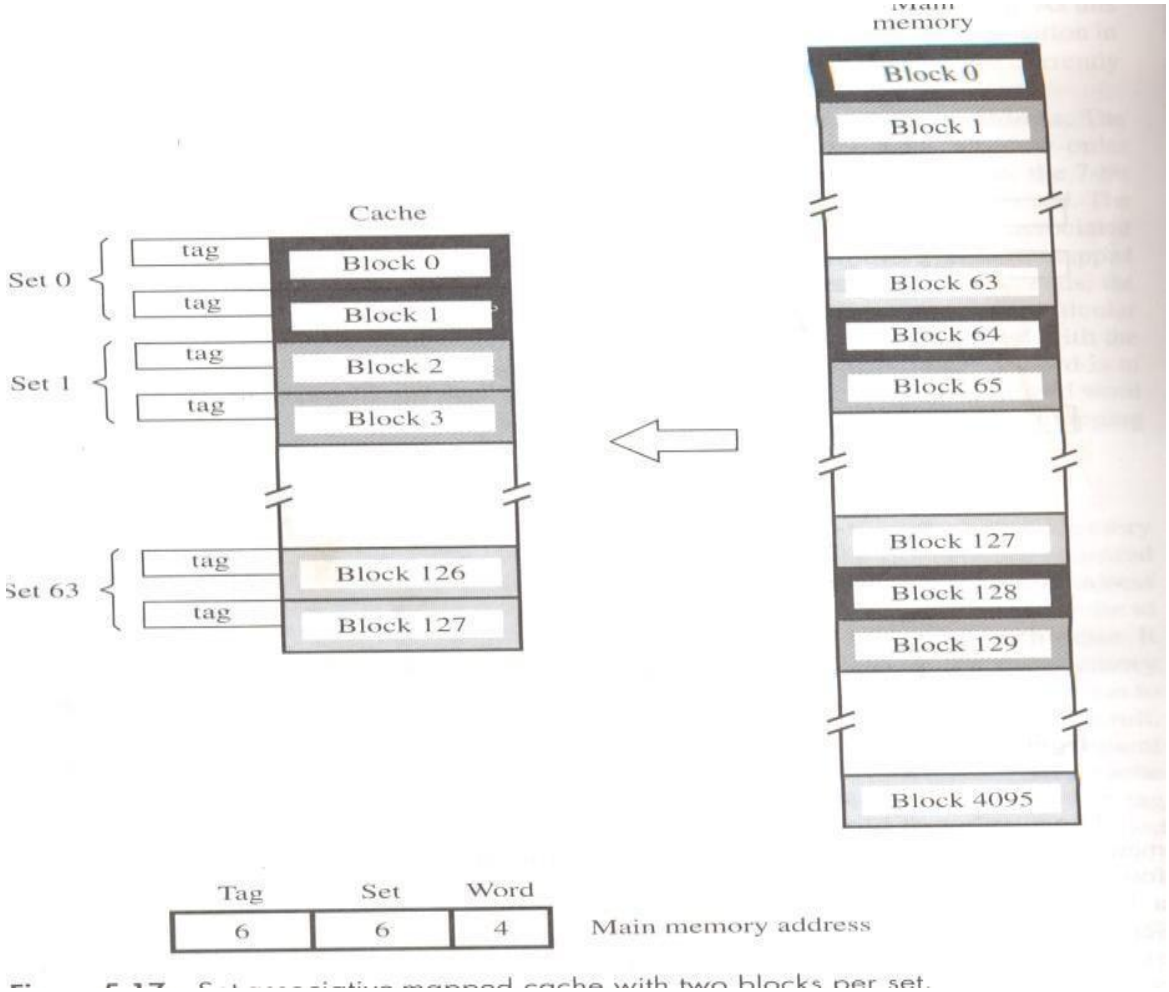
- Its cost is high.

Set-Associative Mapping:

- It is the combination of direct and associative mapping.
- 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 the specified set.
- In this case,the cache has two blocks per set,so the memory blocks 0,64,128…..... 4032 maps into cache set „0" and they can occupy either of the two block position within the set.
-

6 bit set field→Determines which set of cache contains the desired block .
6 bit tag field→The tag field of the address is compared to the tags of the two blocks of the set to clock if the desired block is present.

Fig: Set-Associative Mapping:



No of blocks per set	**no of set field**
2	6
3	5
8	4
128	no set field

- The cache which contains 1 block per set is called **direct Mapping.**
- A cache that has „k" blocks per set is called as „**k-way set associative cache**".
- Each block contains a control bit called a **valid bit**.
- The Valid bit indicates that whether the block contains valid data.
- The dirty bit indicates that whether the block has been modified during its cache residency.

  Valid bit=0→When power is initially applied to system

  Valid bit =1→When the block is loaded from main memory at first time.
- If the main memory block is updated by a source & if the block in the source is already exists in the cache,then the valid bit will be cleared to „0".
- If Processor & DMA uses the same copies of data then it is called as the **Cache Coherence Problem.**

Merit:

- The Contention problem of direct mapping is solved by having few choices for block placement.
- The hardware cost is decreased by reducing the size of associative search.

Replacement Algorithm:

- In direct mapping, the position of each block is pre-determined and there is noneed of replacement strategy.
- In associative and set assosaitive method the block positionis not pre-determined ; ie.. when the cache is full and if new blocks are brought into the cache,then the cache controller must decide
  - Therefore,when a block is to be over-written,it is sensible to over-write the one that has gone the longest time without being referenced.This block is called **Least recently Used(LRU) block** & the technique is called **LRU algorithm**.
  - The cache controller track the references to all blocks with the help of block counter.

Eg:

Consider 4 blocks/set in set associative cache,

- 2 bit counter can be used for each block.
- When a **'hit'** occurs,then block counter=0;The counter with values originally lower than the referenced one are incremented by 1 & all others remain unchanged.
- When a **'miss'** occurs & if the set is full,the blocks with the counter value 3 is removed,the new block is put in its place & its counter is set to „0" and other block counters are incremented by 1.

Merit:

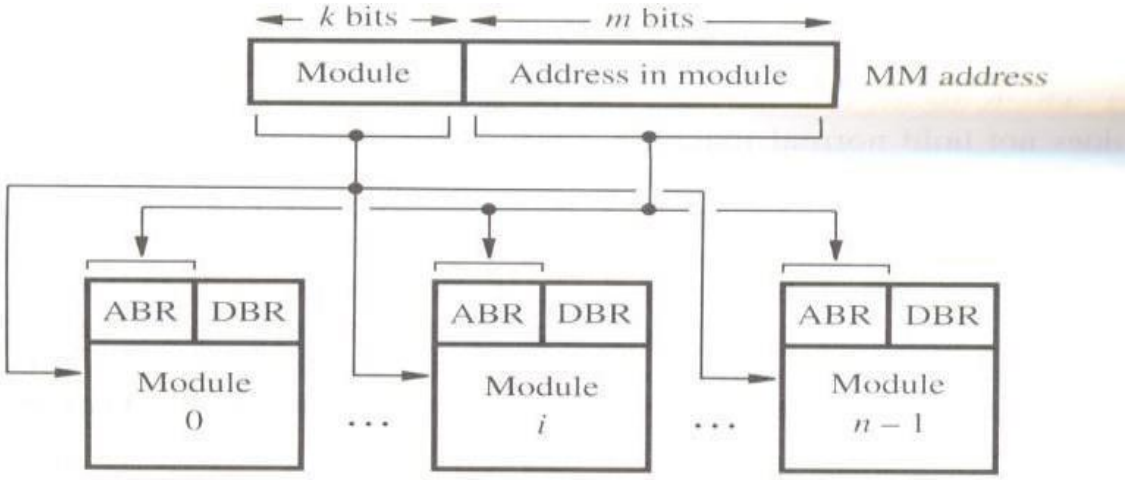
- The performance of LRU algorithm is improved by randomness in deciding which block is to be over-written.

PERFORMANCE CONSIDERATION:

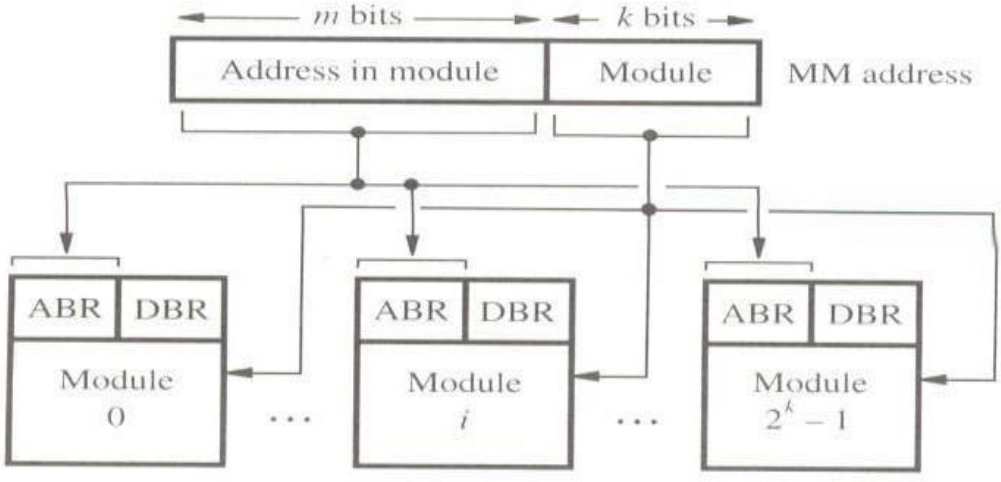
- Two Key factors in the commercial success are the performance & cost ie the best possible performance at low cost.
- A common measure of success is called the **Pricel Performance ratio**.
- Performance depends on how fast the machine instruction are brought to the processor and how fast they are executed.
- To achieve parallelism(ie. Both the slow and fast units are accessed in the same manner),**interleaving** is used.

Interleaving:

Fig:Consecutive words in a Module



(a) Consecutive words in a module

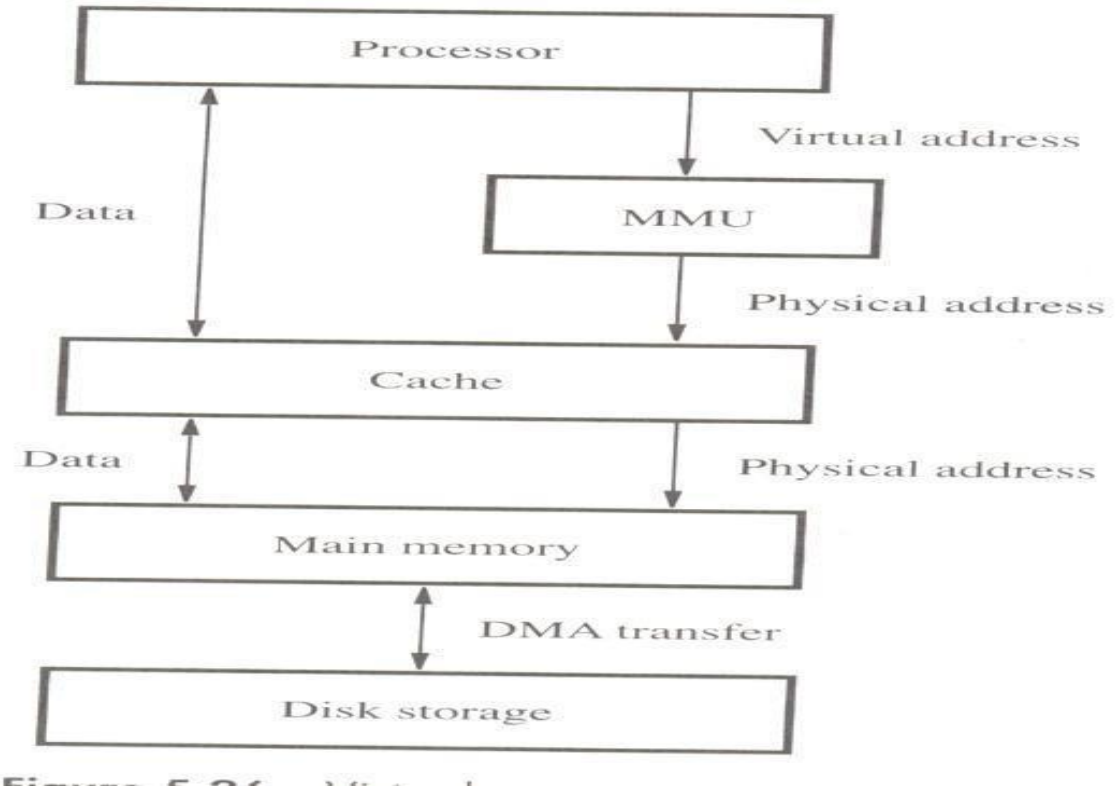


(b) Consecutive words in consecutive modules

VIRTUAL MEMORY:

- Techniques that automatically move program and data blocks into the physical main memory when they are required for execution is called the **Virtual Memory.**
- The binary address that the processor issues either for instruction or data are called the **virtual / Logical address**.
- The virtual address is translated into physical address by a combination of hardware and software components.This kind of address translation is done by **MMU**(Memory Management Unit).
- When the desired data are in the main memory ,these data are fetched /accessed immediately.
- If the data are not in the main memory,the MMU causes the Operating system to bring the data into memory from the disk.
- Transfer of data between disk and main memory is performed using DMA scheme.

Fig:Virtual Memory Organisation



Address Translation:

- In address translation,all programs and data are composed of fixed length units called **Pages.**
- The Page consists of a block of words that occupy contiguous locations in the main memory.
- The pages are commonly range from 2K to 16K bytes in length.
- The **cache bridge** speed up the gap between main memory and secondary storage and it is implemented in software techniques.
- Each virtual address generated by the processor contains **virtual Page number**(Low order bit) and **offset**(High order bit)

Virtual Page number+ Offset→Specifies the location of a particular byte (or word) within a page.

Page Table:

- It contains the information about the main memory address where the page is stored & the current status of the page.

Page Frame:

- An area in the main memory that holds one page is called the page frame.

Page Table Base Register:

- It contains the starting address of the page table.

Virtual Page Number+Page Table Base register→Gives the address of the corresponding entry in the page table.ie)it gives the starting address of the page if that page currently resides in memory.

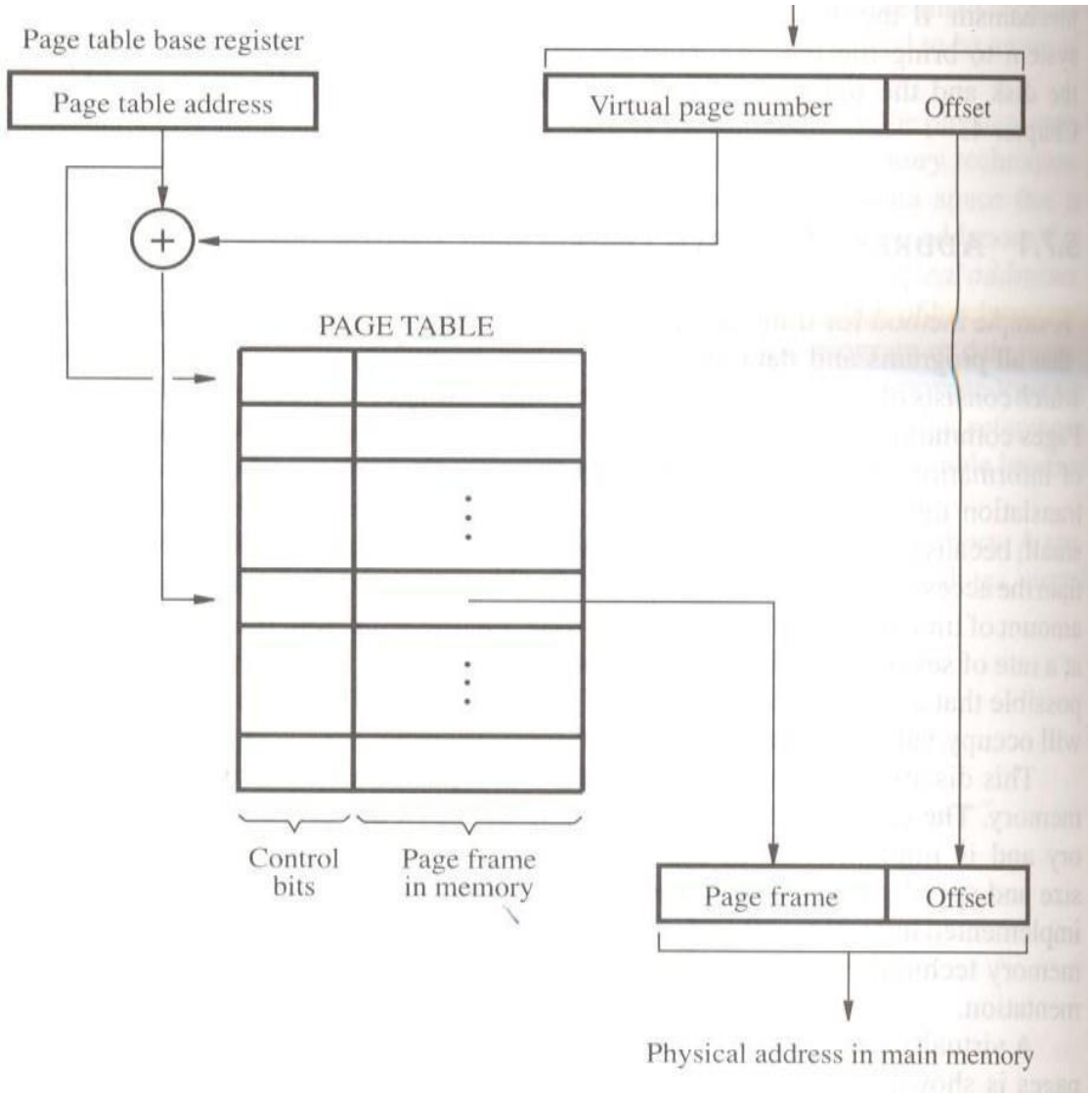
Control Bits in Page Table:

- The Control bits specifies the status of the page while it is in main memory.

Function:

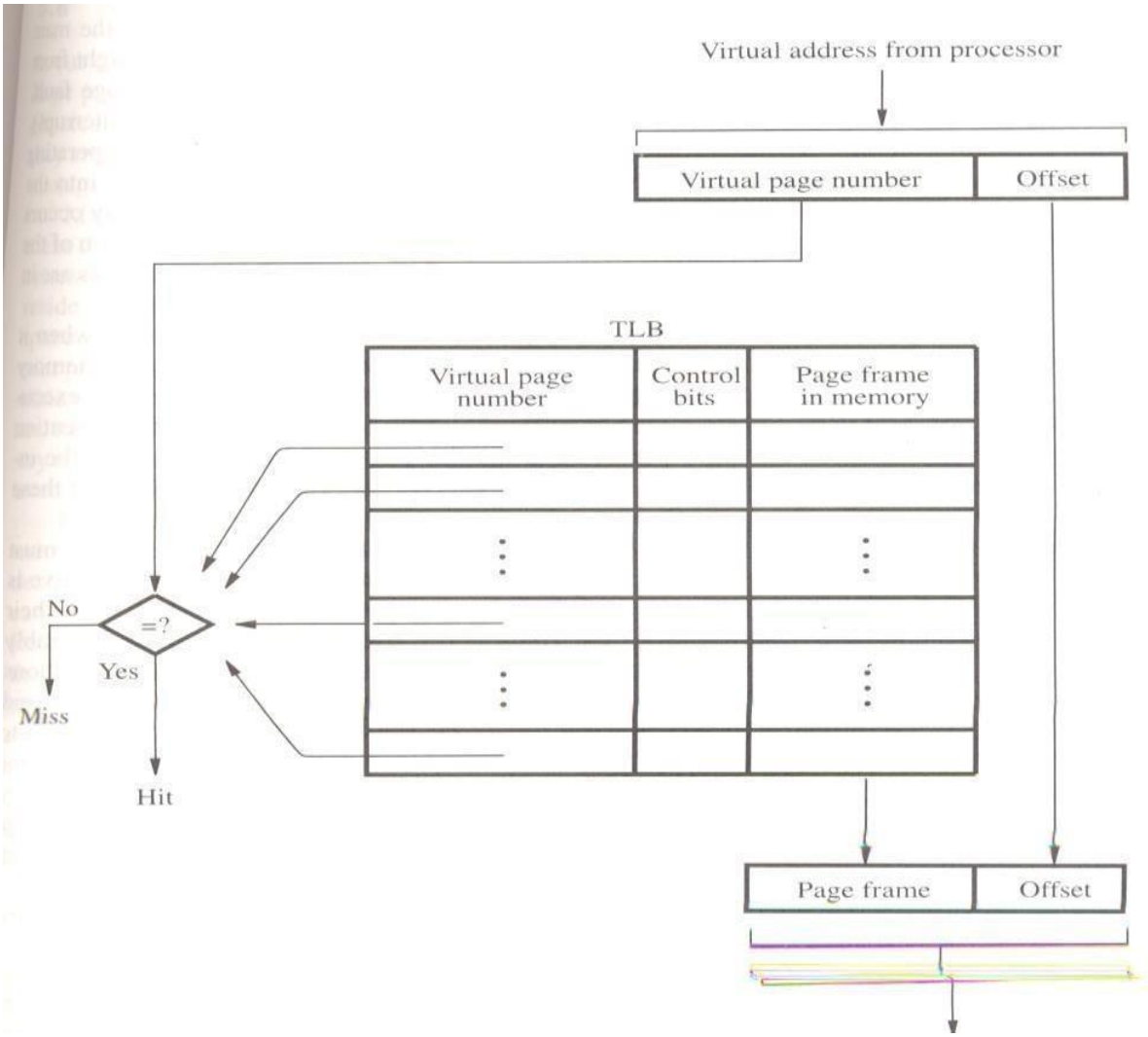
- The control bit indicates the validity of the page ie)it checks whether the page is actually loaded in the main memory.
- It also indicates that whether the page has been modified during its residency in the memory;this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page.

Fig:Virtual Memory Address Translation



- The Page table information is used by MMU for every read & write access.
- The Page table is placed in the main memory but a copy of the small portion of the page table is located within MMU.
- This small portion or small cache is called Translation **LookAside Buffer(TLB).**
- This portion consists of the page table enteries that corresponds to the most recently accessed pages and also contains the virtual address of the entry.

Fig:Use of Associative Mapped TLB



- When the operating system changes the contents of page table ,the control bit in TLB will invalidate the corresponding entry in the TLB.
- Given a virtual address,the MMU looks in TLB for the referenced page.
- If the page table entry for this page is found in TLB,the physical address is obtained immediately.
- If there is a miss in TLB,then the required entry is obtained from the page table in the main memory & TLB is updated.
- When a program generates an access request to a page that is not in the main memory ,then Page Fault will occur.
- The whole page must be broght from disk into memry before an access can proceed.

- When it detects a page fault,the MMU asks the operating system to generate an interrupt.
- The operating System suspend the execution of the task that caused the page fault and begin execution of another task whose pages are in main memory because the long delay occurs while page transfer takes place.
- When the task resumes,either the interrupted instruction must continue from the point of interruption or the instruction must be restarted.
- If a new page is brought from the disk when the main memory is full,it must replace one of the resident pages.In that case,it uses LRU algorithm which removes the least referenced Page.
- A modified page has to be written back to the disk before it is removed from the main memory. In that case,write –through protocol is used.

MEMORY MANAGEMENT REQUIREMENTS:

- Management routines are part of the Operating system.
- Assembling the OS routine into virtual address space is called „**System Space**".
- The virtual space in which the user application program reside is called the „**User Space'**.
- Each user space has a separate page table.
- The MMU uses the page table to determine the address of the table to be used in the translation process.
- Hence by changing the contents of this register, the OS can switch from one space to another.
- The process has two stages. They are,
  - User State
  - Supervisor state.

User State:

- In this state,the processor executes the user program.

Supervisor State:

- When the processor executes the operating system routines,the processor will be in supervisor state.

Privileged Instruction:

- In user state,the machine instructions cannot be executed.Hence a user program is prevented from accessing the page table of other user spaces or system spaces.
- The control bits in each entry can be set to control the access privileges granted to each program.
- Ie)One program may be allowed to read/write a given page,while the other programs may be given only red access.

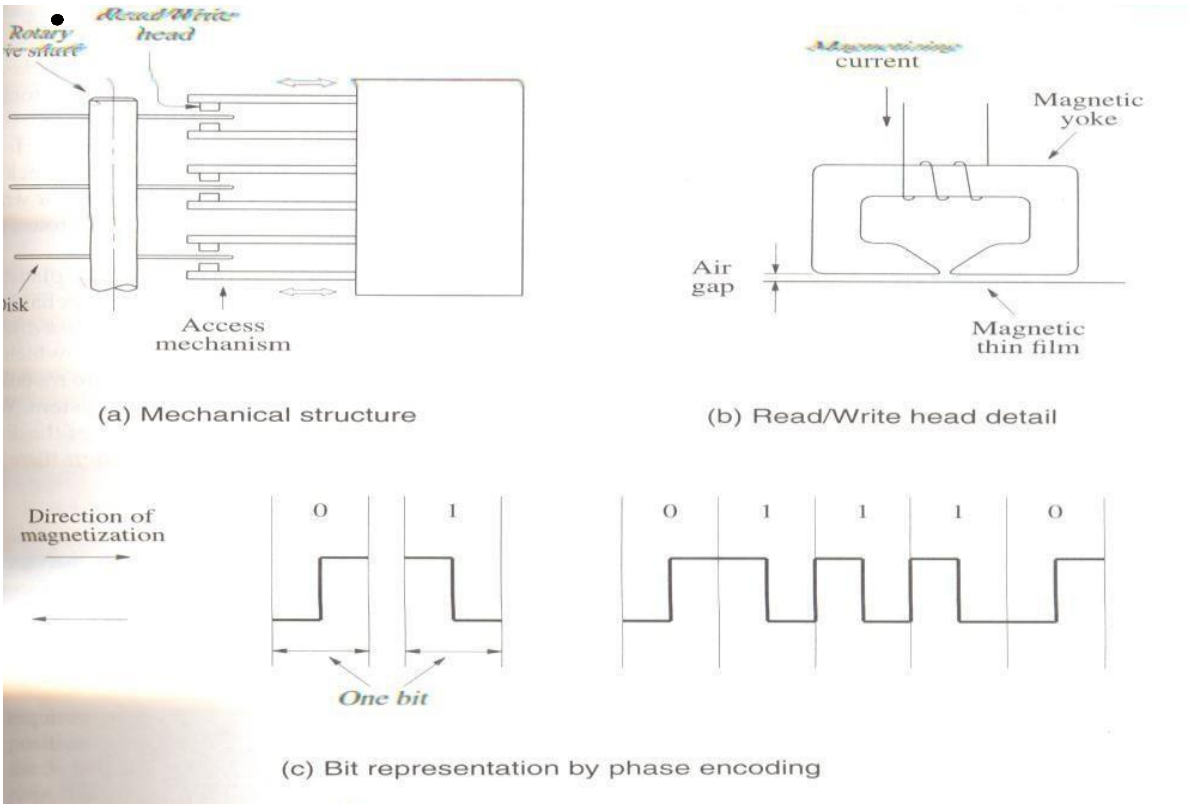
SECONDARY STORAGE:

- The Semi-conductor memories donot provide all the storage capability.
- The Secondary storage devices provide larger storage requirements.
- Some of the Secondary Storage devices are,
  - Magnetic Disk
  - Optical Disk
  - Magnetic Tapes.

Magnetic Disk:

- Magnetic Disk system consists o one or more disk mounted on a common spindle.
- A thin magnetic film is deposited on each disk, usually on both sides.
- The disk are placed in a rotary drive so that the magnetized surfaces move in close proximity to read /write heads.
- Each head consists of **magnetic yoke & magnetizing coil**.
- Digital information can be stored on the magnetic film by applying the current pulse of suitable polarity to the magnetizing coil.
- Only changes in the magnetic field under the head can be sensed during the Read operation.
- Therefore if the binary states 0 & 1 are represented by two opposite states of magnetization, a voltage is induced in the head only at 0-1 and at 1-0 transition in the bit stream.
- A consecutive (long string) of 0"s & 1"s are determined by using the clock which is mainly used for synchronization.
- Phase Encoding or Manchester Encoding is the technique to combine the clocking information with data.
- The Manchester Encoding describes that how the self-clocking scheme is implemented.

Fig:Mechanical Structure

- 

The Read/Write heads must be maintained at a very small distance from the moving disk surfaces in order to achieve high bit densities.

- When the disk are moving at their steady state, the air pressure develops between the disk surfaces & the head & it forces the head away from the surface.
- The flexible spring connection between head and its arm mounting permits the head to fly at the desired distance away from the surface.

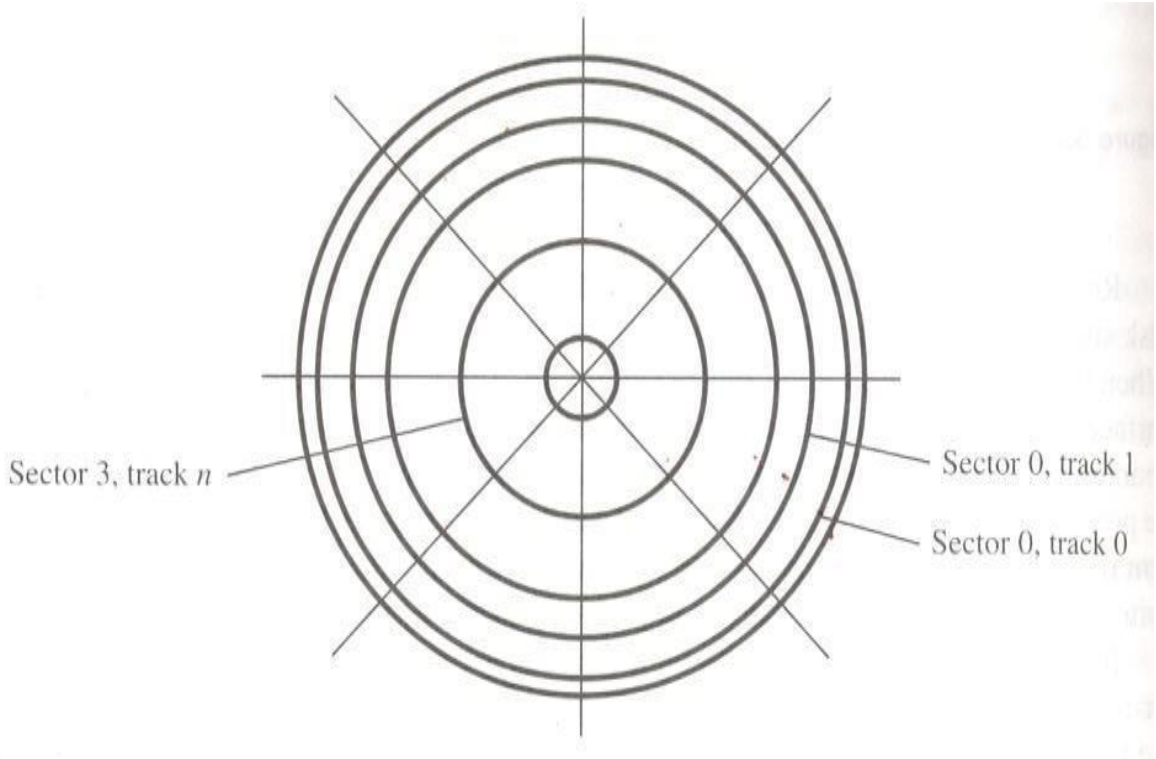
Wanchester Technology:

- Read/Write heads are placed in a sealed, air –filtered enclosure called the Wanchester Technology.
- In such units, the read/write heads can operate closure to magnetic track surfaces because the dust particles which are a problem in unsealed assemblies are absent.

Merits:

- It have a larger capacity for a given physical size.
- The data intensity is high because the storage medium is not exposed to contaminating elements.
- The read/write heads of a disk system are movable.
- The disk system has 3 parts.They are,
  - **Disk Platter**(Usually called Disk)
  - **Disk Drive(**spins the disk & moves Read/write heads)
  - **Disk Controller**(controls the operation of the system.)

Fig:Organizing & Accessing the data on disk



- Each surface is divided into concentric **tracks**.
- Each track is divided into **sectors**.
- The set of corresponding tracks on all surfaces of a stack of disk form a **logical cylinder.**

- The data are accessed by specifying **the surface number,track number and the sector number.**
- The Read/Write operation start at sector boundaries.
- Data bits are stored serially on each track.
- Each sector usually contains 512 bytes.

Sector header -> contains identification information.
It helps to find the desired sector on the selected track.
ECC (Error checking code)- used to detect and correct errors.

- An unformatted disk has no information on its tracks.
- The formatting process divides the disk physically into tracks and sectors and this process may discover some defective sectors on all tracks.
- The disk controller keeps a record of such defects.
- The disk is divided into logical partitions. They are,
  - Primary partition
  - Secondary partition
- In the diag, Each track has same number of sectors.
- So all tracks have same storage capacity.
- Thus the stored information is packed more densely on inner track than on outer track.

Access time

- There are 2 components involved in the time delay between receiving an address and the beginning of the actual data transfer. They are,
  - Seek time
  - Rotational delay / Latency

Seek time – Time required to move the read/write head to the proper track.
Latency – The amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed sector passes under the read/write head.
Seek time + Latency = Disk access time
Typical disk

One inch disk- weight=1 ounce, size -> comparable to match book
Capacity -> 1GB
Inch disk has the following parameter
Recording surface=20
Tracks=15000 tracks/surface
Sectors=400.
Each sector stores 512 bytes of data
Capacity of formatted disk=20x15000x400x512=$60x10^9$ =60GB
Seek time=3ms
Platter rotation=10000 rev/min
Latency=3ms
Internet transfer rate=34MB/s
Data Buffer / cache

- A disk drive that incorporates the required SCSI circuit is referred as SCSI drive.
- The SCSI can transfer data at higher rate than the disk tracks.
- An efficient method to deal with the possible difference in transfer rate between disk and SCSI bus is accomplished by including a data buffer.
- This buffer is a semiconductor memory.
- The data buffer can also provide cache mechanism for the disk (ie) when a read

request arrives at the disk, then controller first check if the data is available in the cache(buffer).

- If the data is available in the cache, it can be accessed and placed on SCSI bus . If it is not available then the data will be retrieved from the disk.

Disk Controller

- The disk controller acts as interface between disk drive and system bus.
- The disk controller uses DMA scheme to transfer data between disk and main memory.
- When the OS initiates the transfer by issuing Read/Write request, the controllers register will load the following information. They are,
- Main memory address(address of 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)

(number of words in the block to be transferred).

Sector header -> contains identification information.

It helps to find the desired sector on the selected track.

ECC (Error checking code)- used to detect and correct errors.

An unformatted disk has no information on its tracks.

The formatting process divides the disk physically into tracks and sectors and this process may discover some defective sectors on all tracks.

The disk controller keeps a record of such defects.

The disk is divided into logical partitions. They are,

Primary partition

Secondary partition

In the diag, Each track has same number of sectors.

So all tracks have same storage capacity.

Thus the stored information is packed more densely on inner track than on outer track.

Access time

There are 2 components involved in the time delay between receiving an address and the beginning of the actual data transfer. They are,

Seek time

Rotational delay / Latency

Seek time – Time required to move the read/write head to the proper track.

Latency – The amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed sector passes under the read/write head.

Seek time + Latency = Disk access time

Typical disk

One inch disk- weight=1 ounce, size -> comparable to match book

Capacity -> 1GB

inch disk has the following

parameterRecording surface=20

Tracks=15000 tracks/surface

Sectors=400.

Each sector stores 512 bytes of data

Capacity of formatted disk=$20x15000x400x512=60x10^9$ =60GB

Seek time=3ms

Platter rotation=10000 rev/min

Latency=3ms
Internet transfer rate=34MB/s

Data Buffer / cache

A disk drive that incorporates the required SCSI circuit is referred as SCSI drive.

The SCSI can transfer data at higher rate than the disk tracks.
An efficient method to deal with the possible difference in transfer rate between disk and SCSI bus is accomplished by including a data buffer.
This buffer is a semiconductor memory.
The data buffer can also provide cache mechanism for the disk (ie) when a read request arrives at the disk, then controller first check if the data is available in the cache(buffer).
If the data is available in the cache, it can be accessed and placed on SCSI bus . If it is not available then the data will be retrieved from the disk.

Disk Controller

The disk controller acts as interface between disk drive and system bus.
The disk controller uses DMA scheme to transfer data between disk and main memory.
When the OS initiates the transfer by issuing Read/Write request, the controllers register will load the following information. They are,
Main memory address(address of 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)
(number of words in the block to be transferred).

PART-A

Notes on
I/O Organization
in Computer Organization – Unit IV

CONTENTS:

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

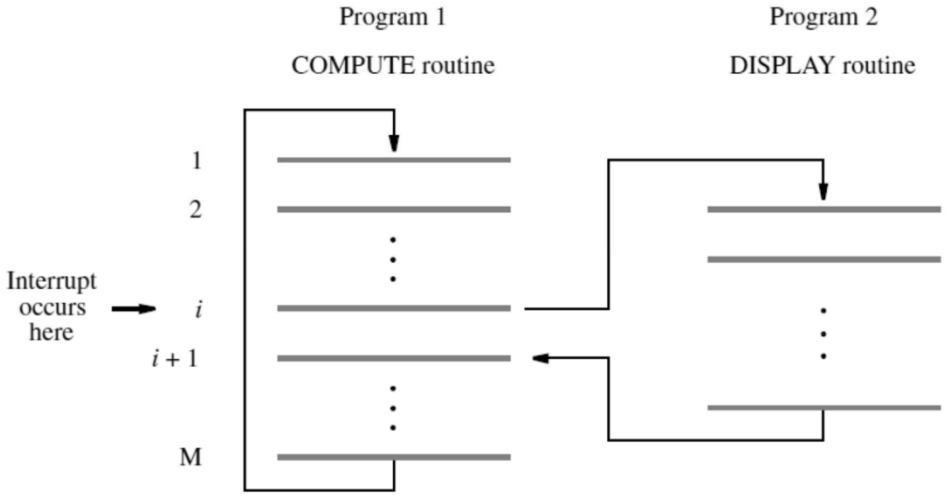
of home appliances, manufacturing equipment, transportation systems, banking, and point-of-sale terminals. In such applications, input to a computer may come from a sensor switch, a digital camera, a microphone, or a fire alarm. Output may be a sound signal sent to a speaker, or a digitally coded command 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.

1. 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.

2. 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 interrupts 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 prevented 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, interrupts are again enabled.

Before proceeding to study more complex aspects of interrupts, 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 interrupt request from a second device. Interrupt-service routines are typically short, and the delay they may cause is acceptable for most simple devices.

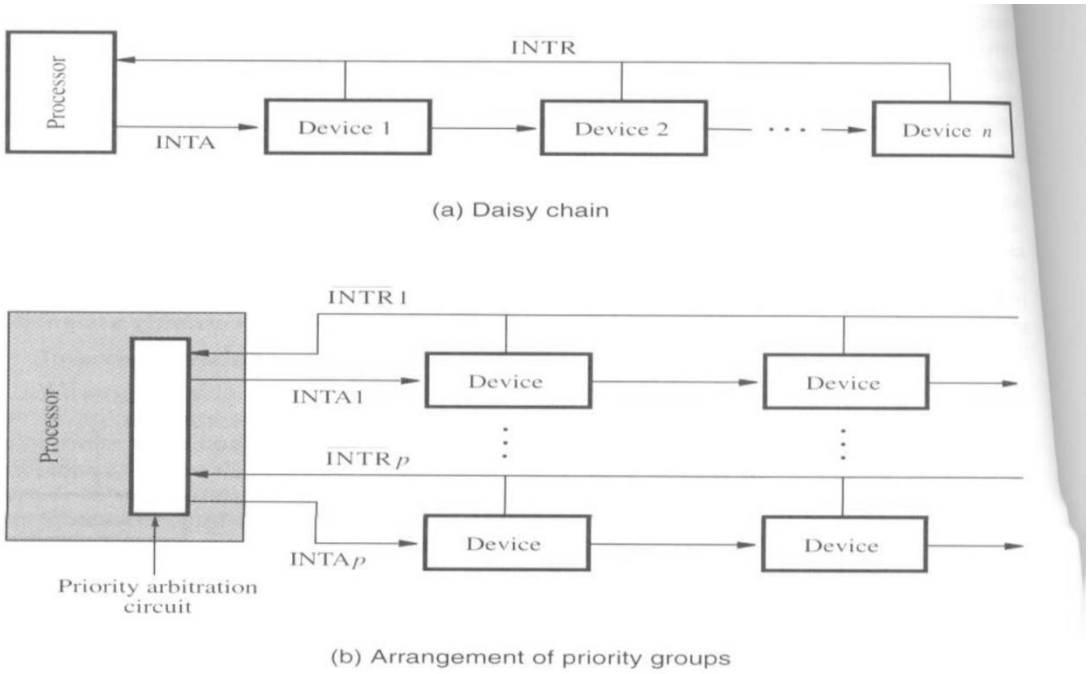
For some devices, however, a long delay in responding to an interrupt request may lead to erroneous operation. Consider, for example, a computer that keeps track of the time of day

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 .



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 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

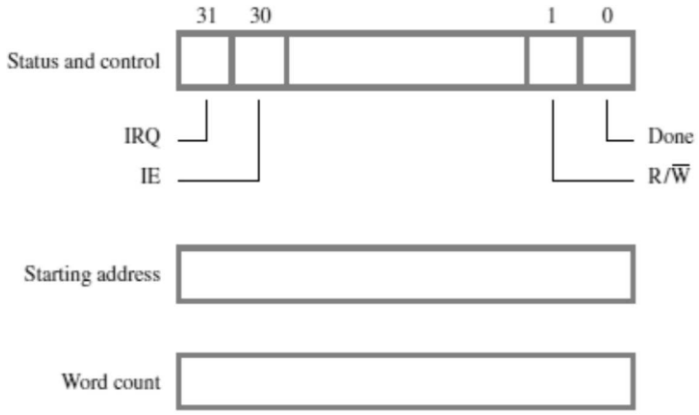
Load R2, DATAIN

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.

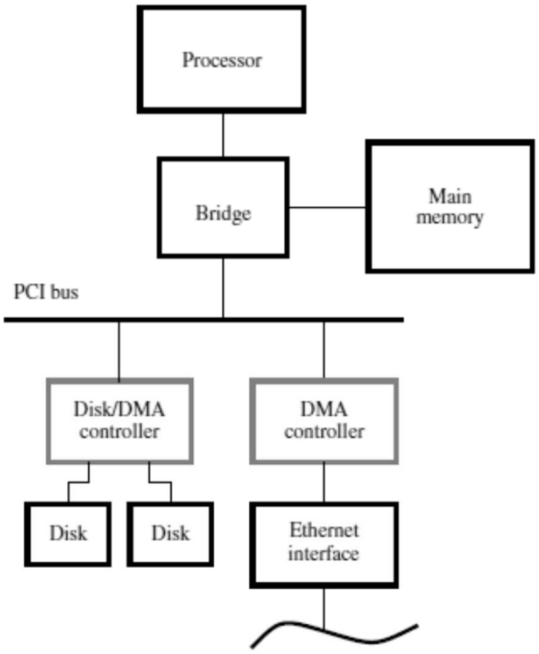
Although a DMA controller transfers data without intervention by the processor, its operation must be under the control of a program executed by the processor, usually an operating system routine. To initiate the transfer of a block of words, the processor sends to the DMA controller the starting address, 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:

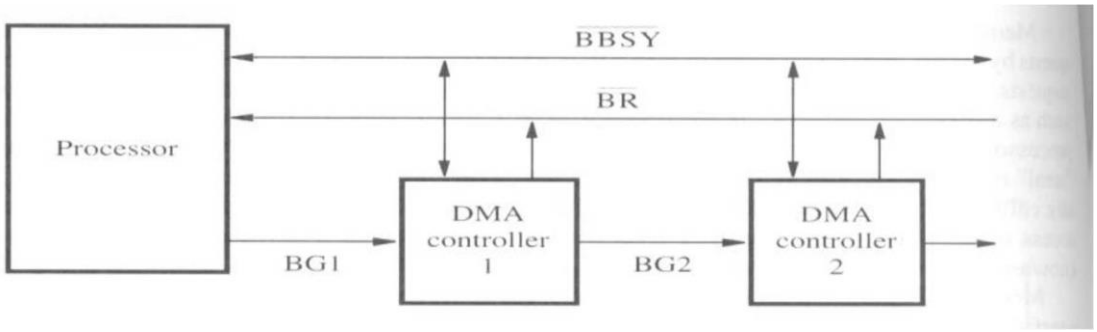
There are 2 approaches to bus arbitration. They are,

□ Centralized arbitration (A single bus arbiter performs arbitration)

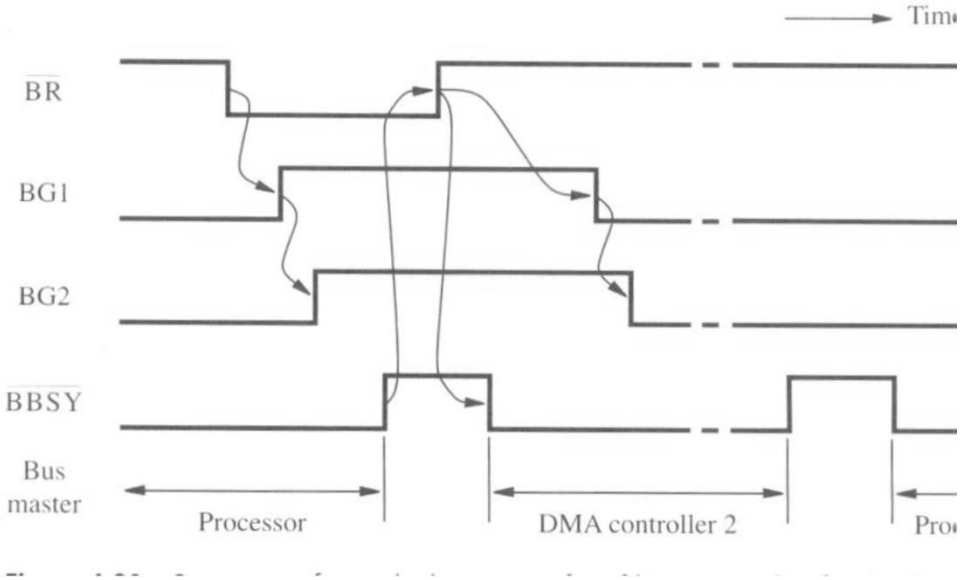
□ Distributed arbitration (all devices participate in the selection of next bus master).

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).

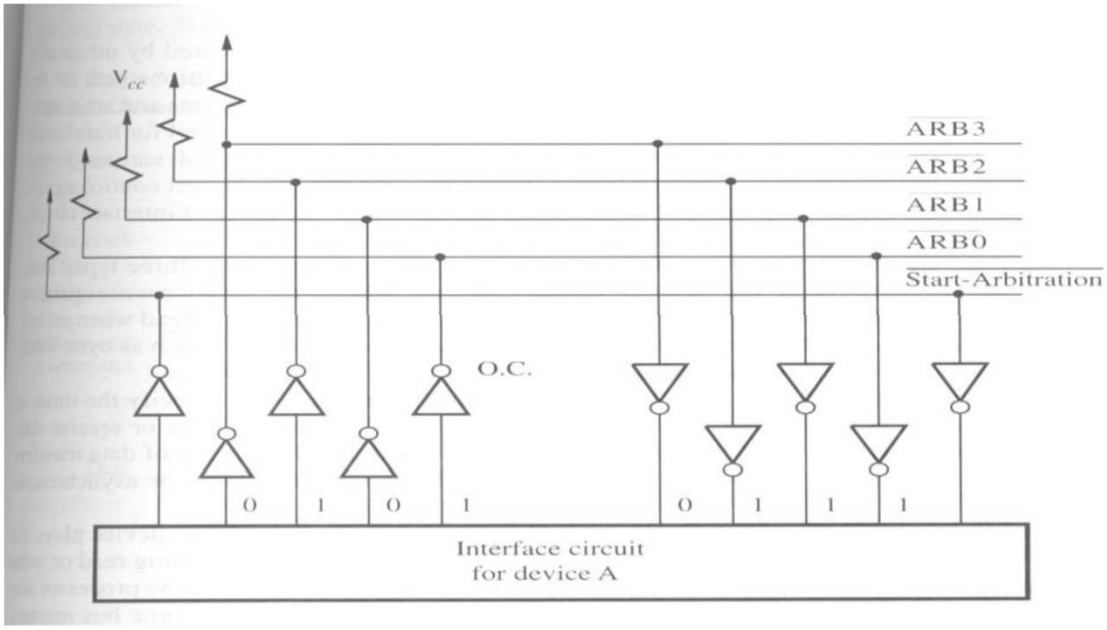


The timing diagram shows the sequence of events for the devices connected to the processor is shown. DMA controller 2 requests and acquires bus mastership and later releases the bus. During its tenure as bus master, it may perform one or more data transfer. After it releases the bus, the processor resources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



3. 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:

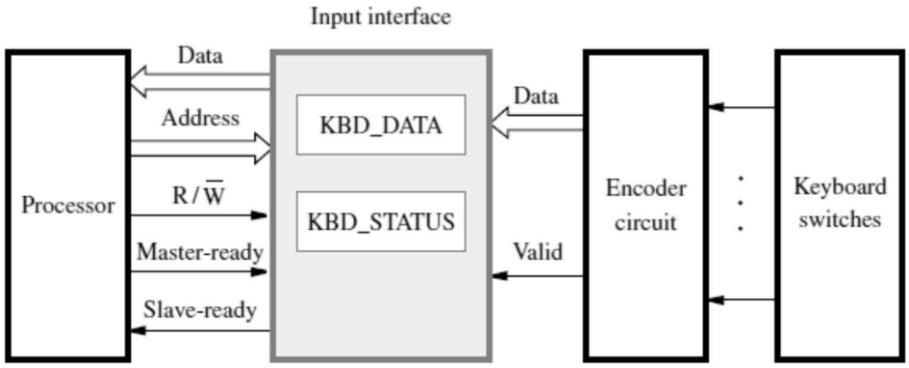
1. Provides a register for temporary storage of data
2. Includes a status register containing status information that can be accessed by the processor
3. Includes a control register that holds the information governing the behaviour of the interface
4. Contains address-decoding circuitry to determine 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.

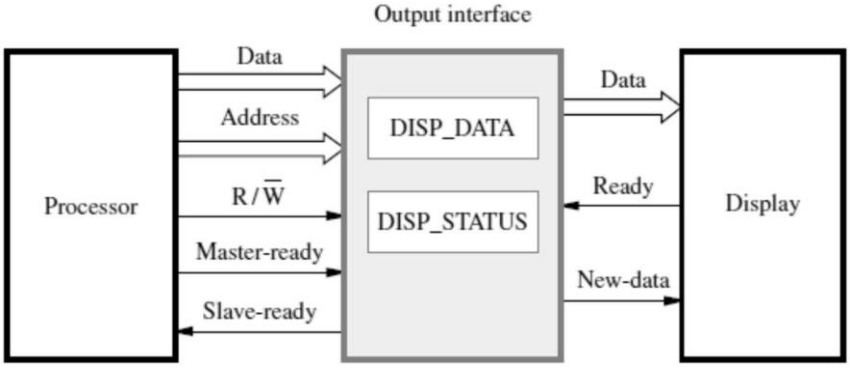
The output of the encoder in Figure consists of one byte of data representing the encoded character and one control signal called Valid. When a key is pressed, the Valid signal changes from 0 to 1, causing the ASCII code of the corresponding character to be loaded into the KBD_DATA register and the status 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.

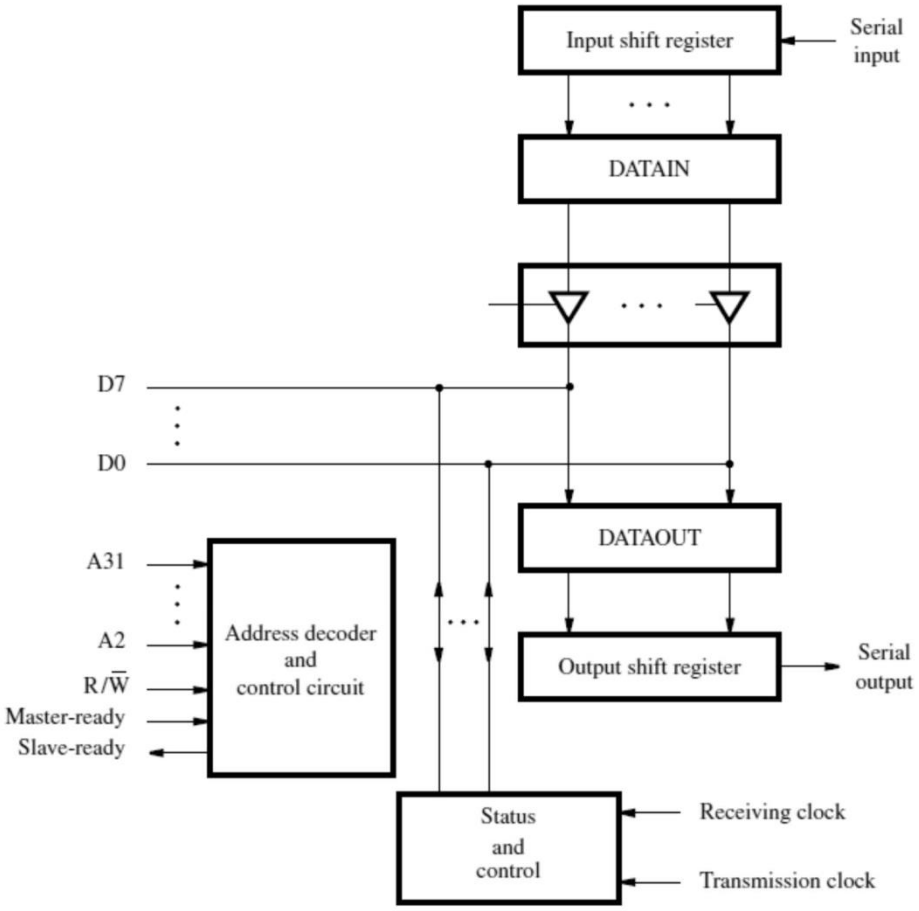
Figure shows an implementation of this interface. Its operation is similar to that of the input interface of Figure 7.11, except that it responds to both Read and Write operations. A Write operation in which A2 = 0 loads a byte of data into register DISP_DATA. A Read operation in which A2 = 1 reads the 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 functionality. 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

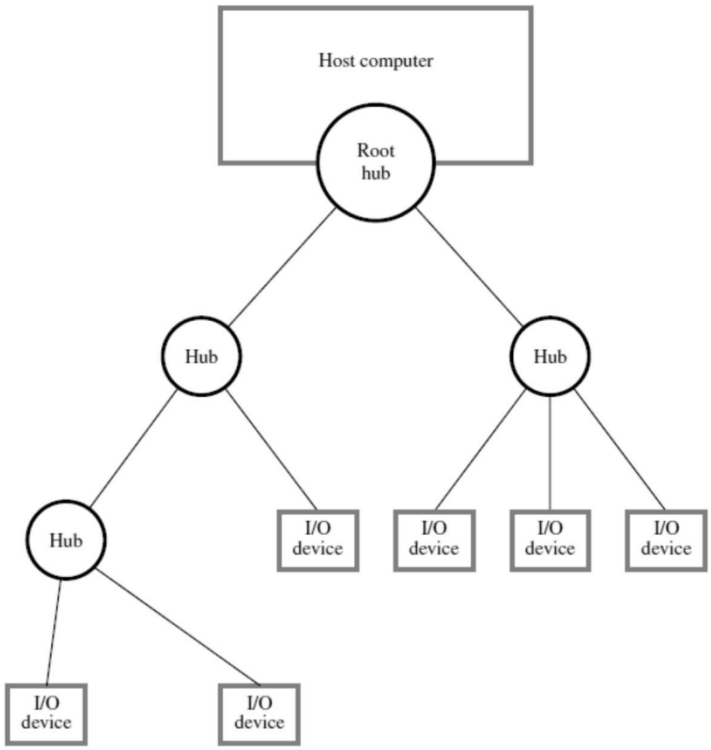
When an I/O device is connected to a computer, the operating system needs some information about it. It needs to know what type of device it is so that it can use the appropriate device driver. It also needs to know the addresses of the registers in the device's interface to be able to communicate with 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.

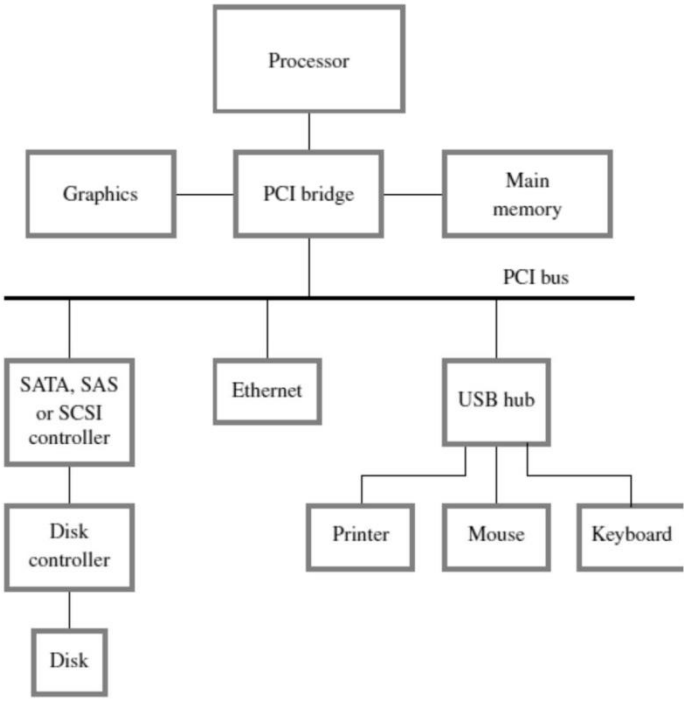
PCI BUS

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

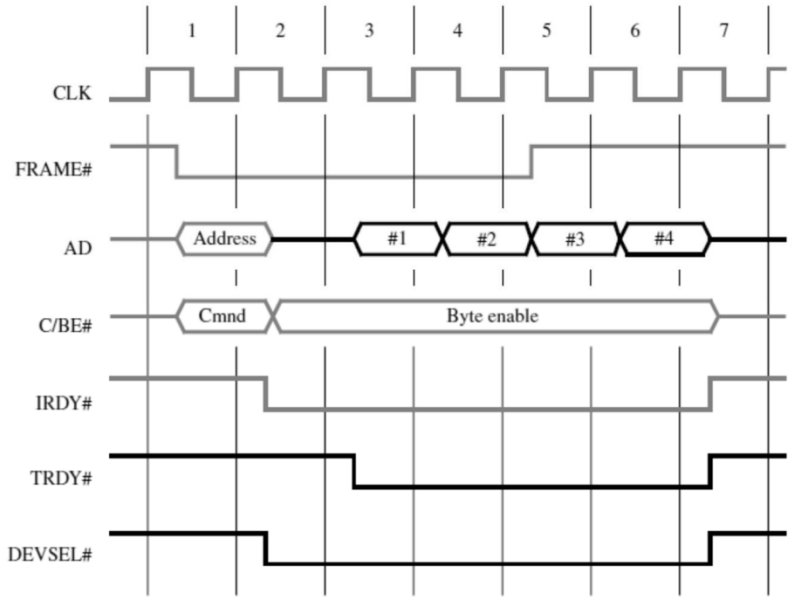
it to access successive address locations. A significant cost reduction can be realized in this manner, because the number of bus lines is an important factor affecting the cost of a computer system.

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 interrupt.

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.

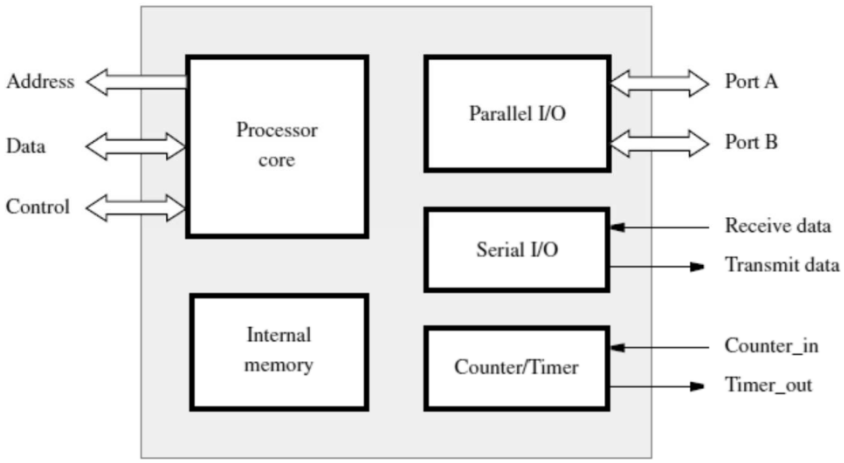
This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. Messages refer to more complex operations that may require several steps to complete, depending on the device. Neither the processor nor the SCSI controller need be aware 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.

4. 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 the 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.

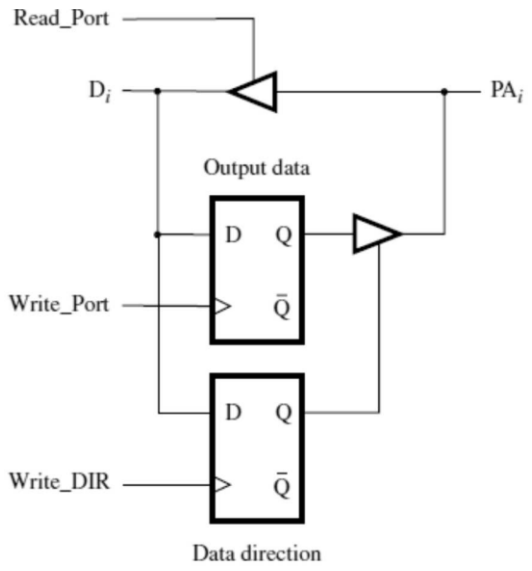
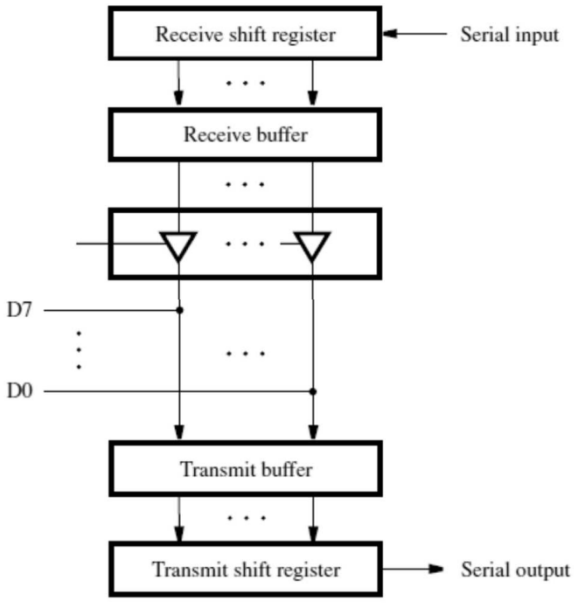


Figure shows only the part of the interface that controls the direction of data transfer. In the input data path there is no flip-flop to capture and hold the value of the data signal provided by a device connected to the corresponding pin. A versatile parallel interface may include two possibilities: 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.

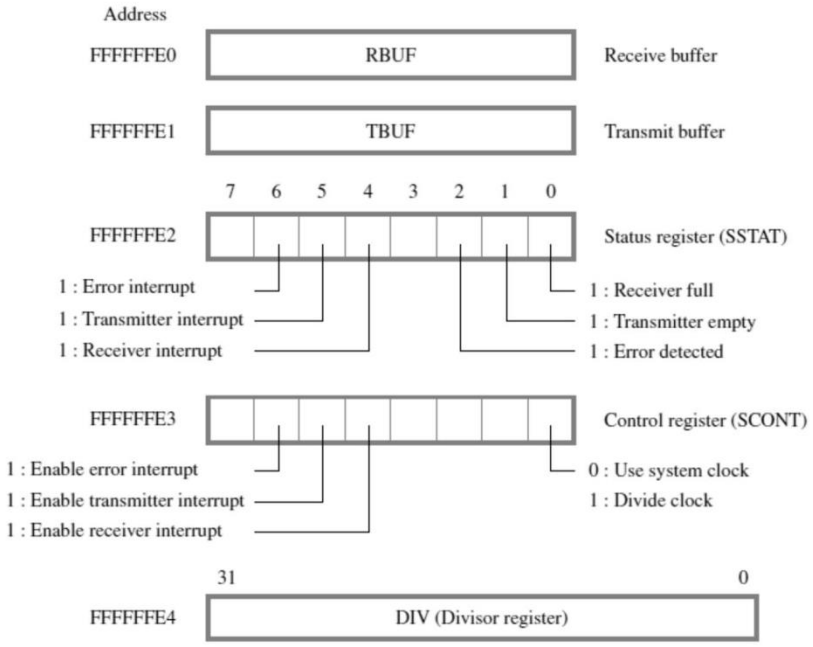
Serial I/O Interface

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.



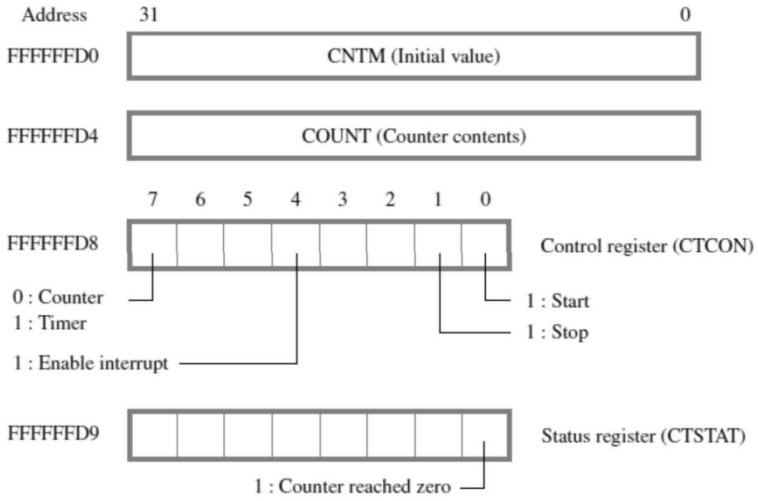
The control register, SCONT, is used to hold the interrupt-enable bits. Setting bits SCONT6−4 to 1 or 0 enables or disables the corresponding interrupts, respectively. This register also indicates how the transmit clock is generated. If SCONT0 = 0, then the transmit clock is the same as the system (processor) 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 interrupts raised by the I/O interfaces within the microcontroller. The XRQ input is used for interrupts raised by external devices. If the IRQ input is asserted and interrupts are enabled, the processor executes an interrupt-service routine that uses the polling method to determine the source(s) of the interrupt 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 interrupts. The IRQ interrupts are enabled if PSR6 = 1, and the XRQ interrupts are enabled if PSR7 = 1. When the processor accepts an interrupt, 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 running 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 execute 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 need 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 to the communications unit. The computer then continues performing other computations while the communications unit handles the details of sending and receiving messages.

CONCLUSION:

In this chapter, we discussed two basic approaches to I/O transfers. The simplest technique is programmed I/O, in which the processor performs all of the necessary functions under direct control of program instructions. The second approach is based on the use of interrupts; this mechanism makes it possible to interrupt the normal execution of programs in order to service higher-priority requests that require more urgent attention. Although all computers have a mechanism for dealing with such situations, the complexity and sophistication of interrupt-handling schemes vary from one computer to another. Thus, we dealt with the I/O issues from the programmer's point of view.

This chapter introduced the I/O structure of a computer from a hardware point of view. I/O devices connected to a bus are used as examples to illustrate the synchronous and asynchronous schemes for transferring data. The architecture of interconnection networks for input and output devices has been a major area of development, driven by an ever-increasing need for transferring data at high speed, for reduced cost, and for features that enhance user

convenience such as plug-and play. Several I/O standards are described briefly in this chapter, illustrating the approaches used to meet these objectives. The current trend is to move away from parallel buses to serial point-to-point links. Serial links have lower cost and can transfer data at high speed.

Fundamental techniques such as pipelining and caches are important ways of improving performance and are widely used in computers. The additional techniques of multithreading, vector (SIMD) processing, and multiprocessing provide the potential for further improvements in performance by making more efficient use of processing resources and by performing more operations in parallel. These techniques have been incorporated into general-purpose multicore processor chips.

Notes on
Pipelining
Unit 5-
part-A

C.Ishaq Shareef
Assistant Professor,
CSE Department,
ASKWEC,

email:**c.shareef@ashokacollege.**

==========

CONTENTS:

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.

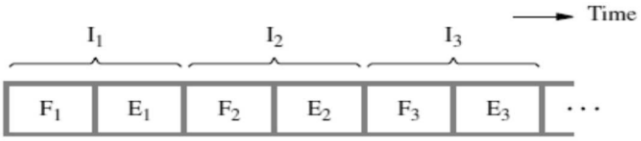
1. BASIC CONCEPTS

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.

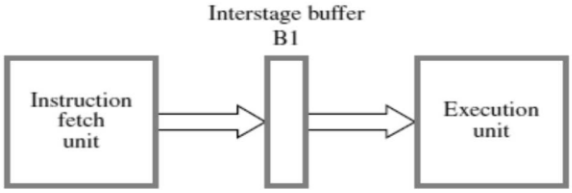
Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly 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.

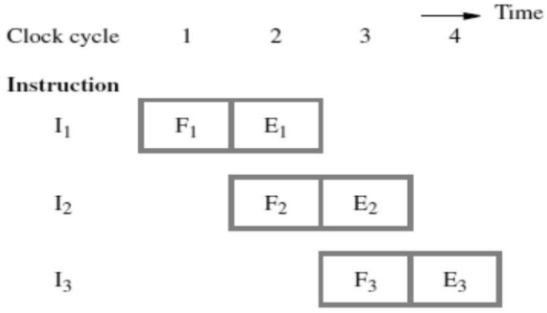
Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure b. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute 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.

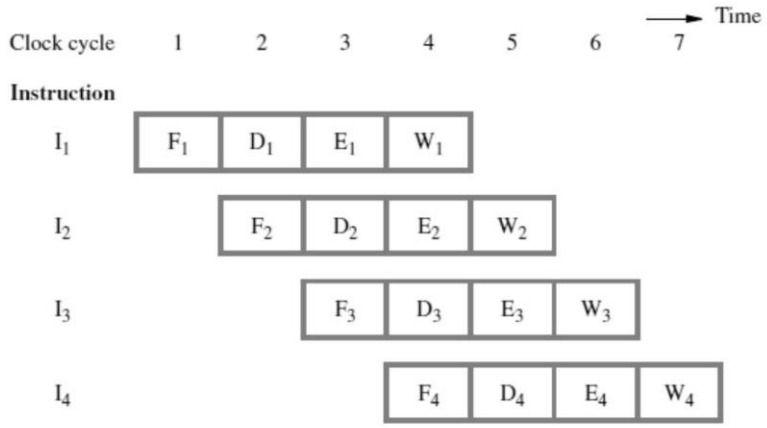
The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory.

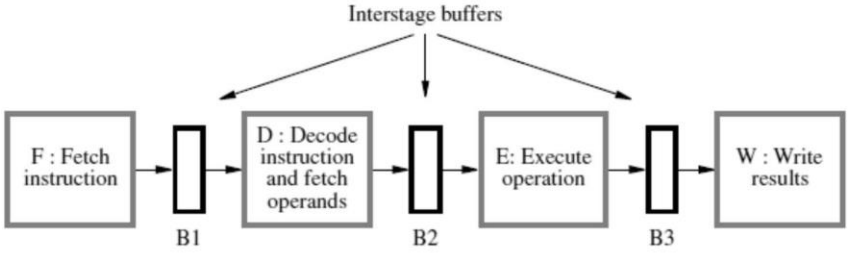
D Decode: decode the instruction and fetch the source operand(s).

E Execute: perform the operation specified by the instruction.

W Write: store the result in the destination location.



(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.

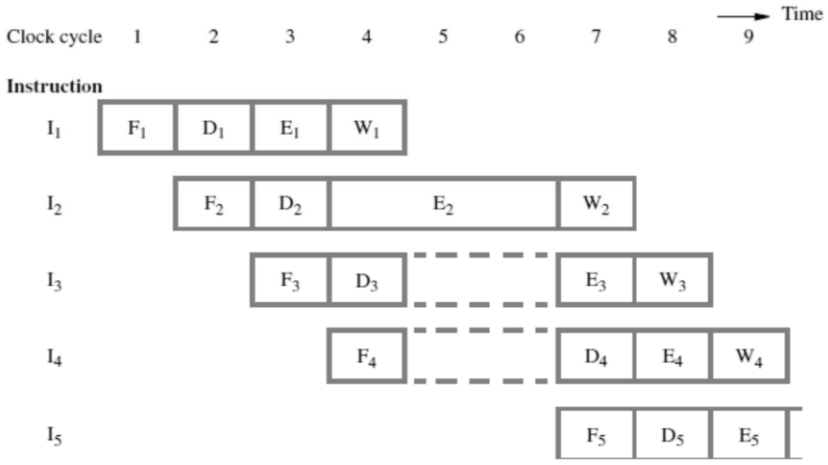
The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This makes it possible to divide instruction fetching 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.

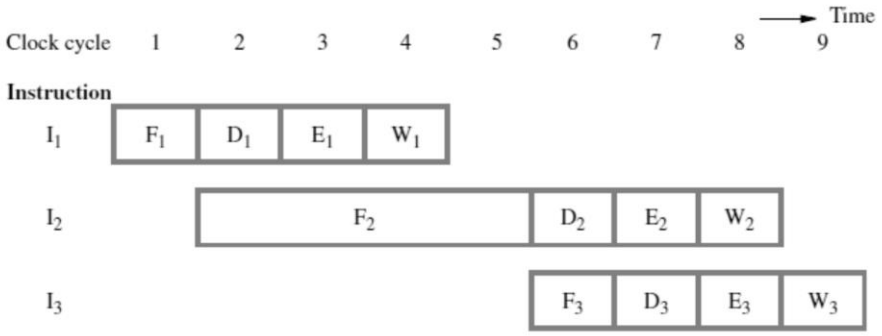
For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four-stage pipeline of Figure b is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. 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

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

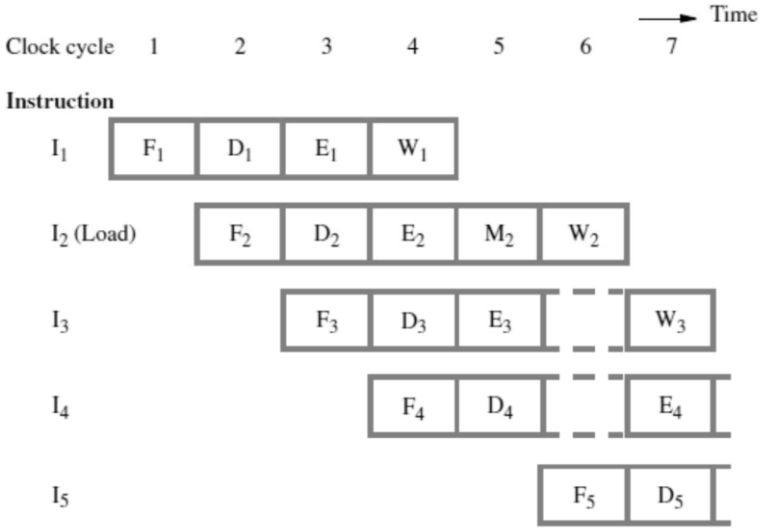
An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure b. This figure gives the function performed by each pipeline stage in each clock cycle. Note that the Decode unit is idle in cycles 3 through 5, the Execute unit is idle in cycles 4 through 6, 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 instruction

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.



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

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

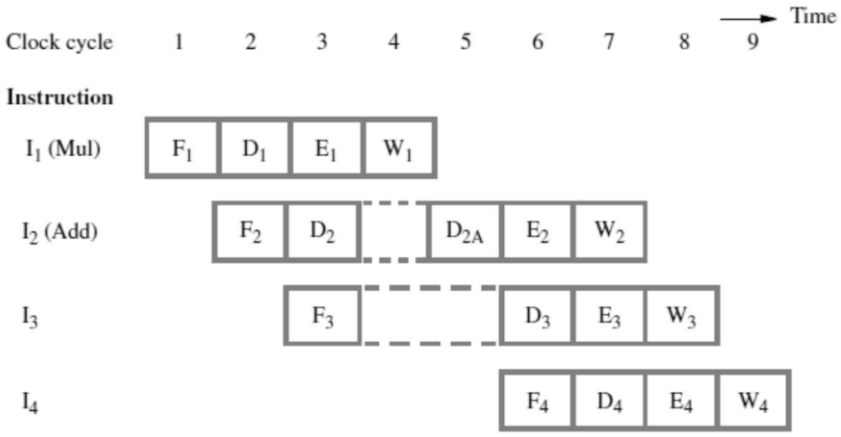
can be performed concurrently, because these operations are independent.

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

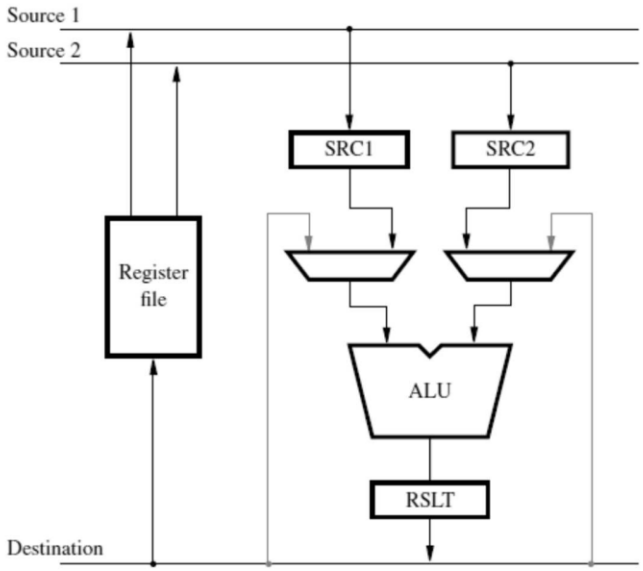
give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure. As the Decode 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.



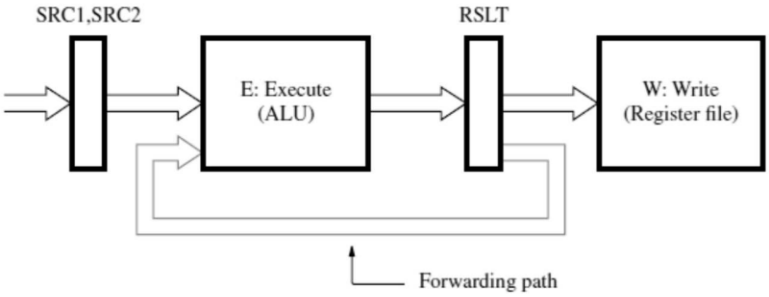
Operand Forwarding

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 an instruction changes 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

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

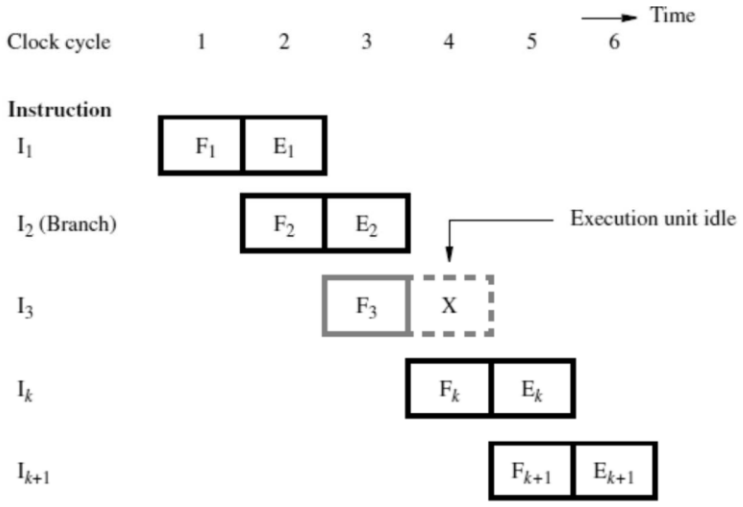
Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have few side effects. Ideally, only 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.

2. 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.

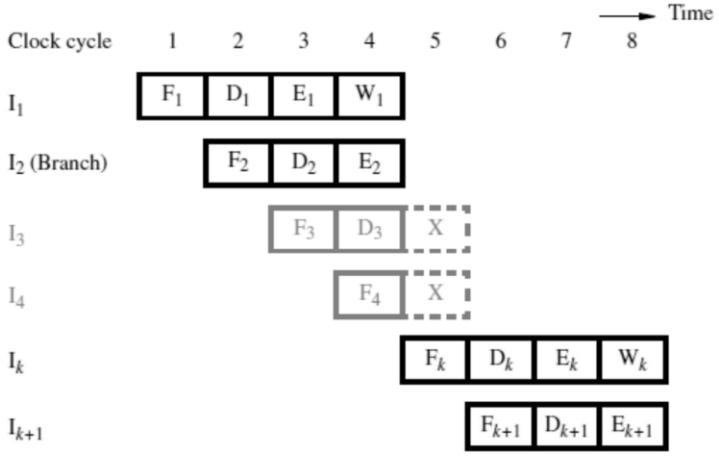
Unconditional Branches

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.

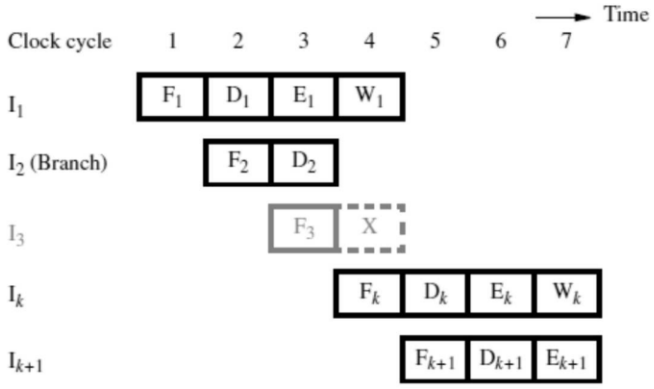


The time lost as a result of a branch instruction is often referred to as the branch penalty. In Figure, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure a shows the effect of a branch instruction on a four-stage pipeline. We have assumed 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.

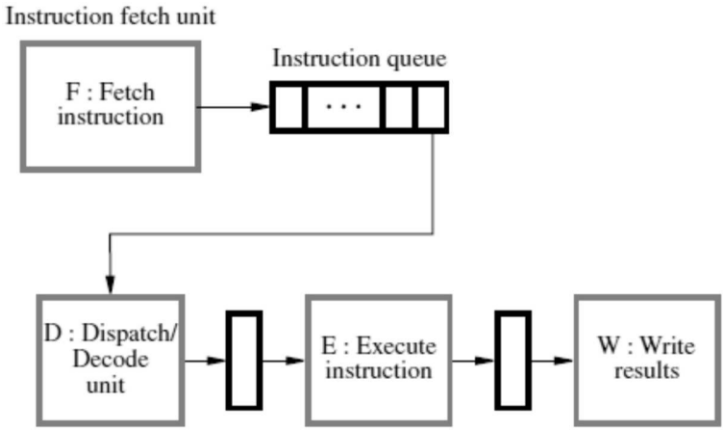
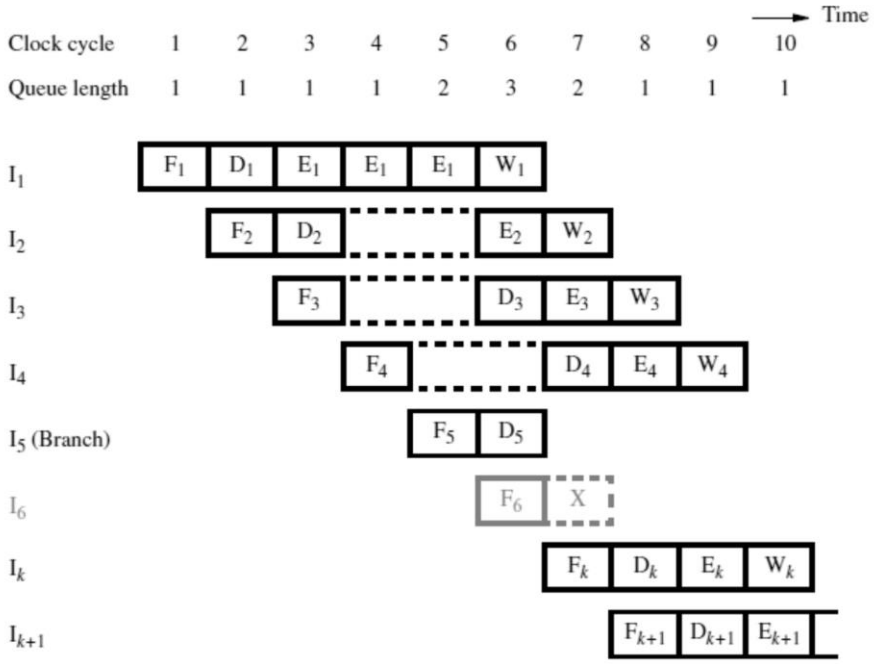


Figure illustrates how the queue length changes and how it affects the relationship between different pipeline stages. We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. 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.

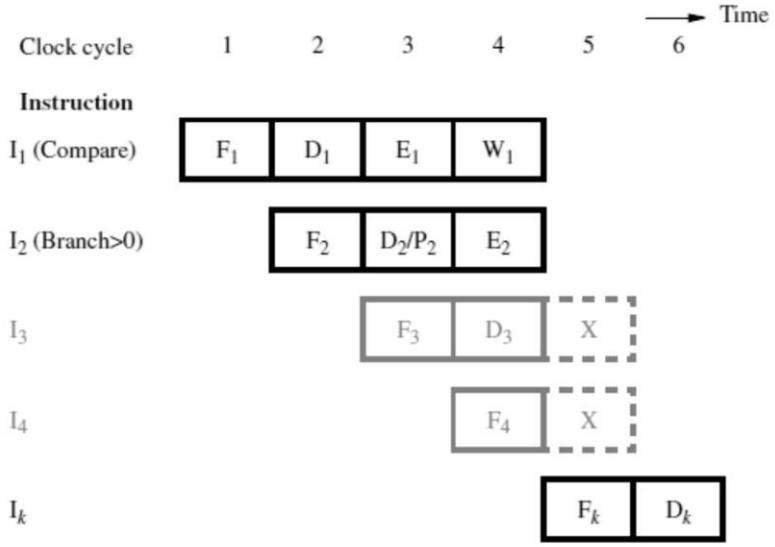


Having an instruction queue is also beneficial in dealing with cache misses. When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as the instruction queue is not empty. Meanwhile, the desired cache block is read from the main memory or from a secondary cache. 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.

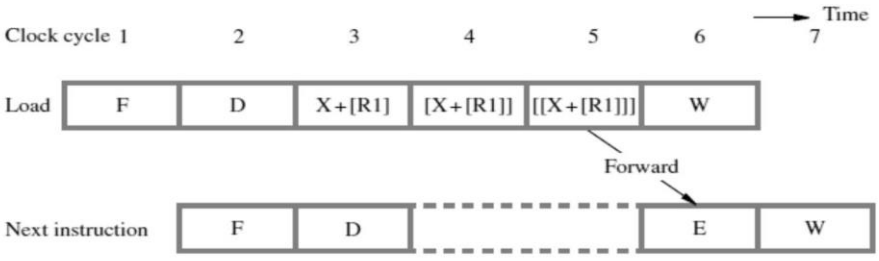
Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers. To compare various approaches, 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

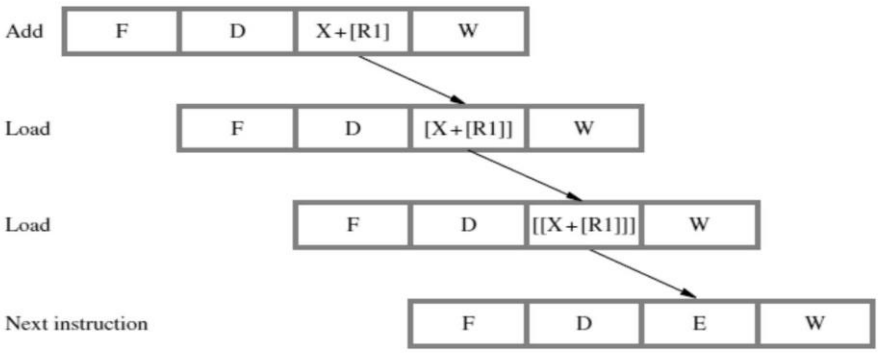
can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

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.

Unit-5

(part-B)

Large computer systems:-

forms of ~~pallas~~ ~~parallel~~ processing:- A/c to flynn's classification of computers are classified into 4 major groups

1) SISD (Single instruction stream single data stream)
2) SIMD (single instruction stream multiple data stream)
3) MIMD
4) MISD

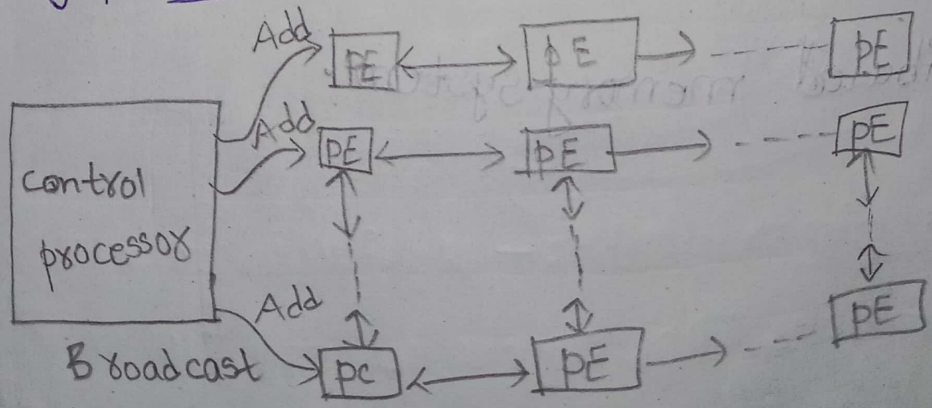
→ SISD:- our desktop computer falls under SISD.

→ SIMD:- Needs suphesticated software for implementation.

→ MISD:- It is unpratical and not at all implemented

→ MIMD:- Most multiprocessor and multicomputer systems falls under MIMD.

Array processors:- It uses SIMD form of parallel processing



It consists of grid of PE of same type. The CU broadcast same instruction to all the processors simultaneously. Each PE is connected to four neighboring PE. so, that they can exchange data easily every PE executes the same instruction sent by control processor and different data sets received from shared memory.

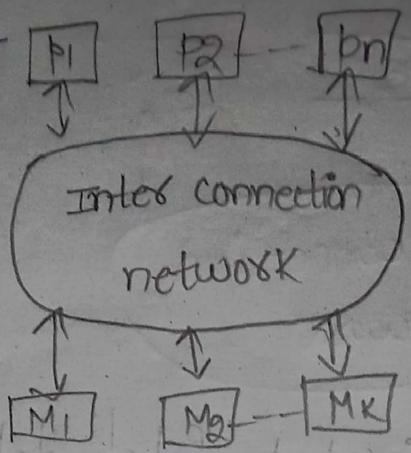
Structure of general purpose multiprocessor :-

MIMD form of parallel processing is very useful in the architecture of general purpose multiprocessors. It involves no. of processors capable of independently executing different routines in parallel.

There are three possible ways of implementing multiprocessor systems.

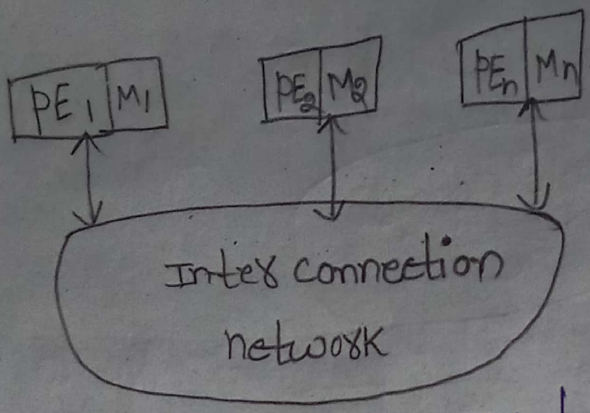
1. UMA multiprocessors (uniform memory access)
2. NUMA "
3. Distributed memory system.

1. UMA Mp:-



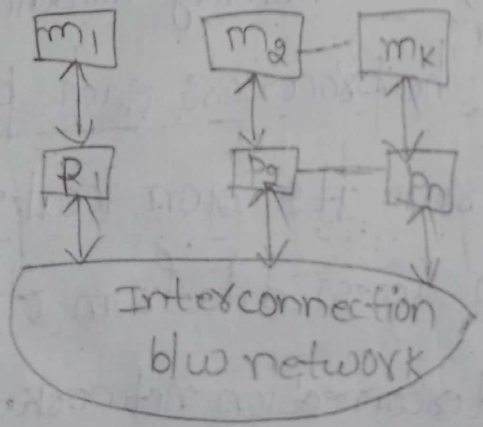
Here, n processors are connected to K memory modules through interconnection network. This interconnection network allows n processors to access data from K memory modules. Hence, a delay is introduced due to the presence of interconnection network. Each processor can fetch data from any of the K memory modules consuming equal no. of clock cycles.

2. NUMA Mp:- Here, the memory module are directly attach to the processors each processor can access data from its own local memory module and it can access data from other memory modules using interconnection network.



Less time is required to fetch data from it's own local memory module and more time is required to fetch data from other memory modules Because, interconnection n/w (network) introduces delay.

3. Distributed memory system :- Here this organisation provides global memory, where any processor can access any memory module without intermission of other processors.



Interconnection networks :- Multiprocessor systems consist of cpus, IDPs connected to I/o devices and

into separate modules. All this are interconnected through interconnection network. Some of the streams for Interconnections are

1) Timeshade common bus
2) Multiplext memory
3) crossbar switch
4) Multistage switching network
5) Hypercube system
6) Mesh
7) Tree
8) Ring
9) Mixed Topology
10) Symetric Multiprocessors

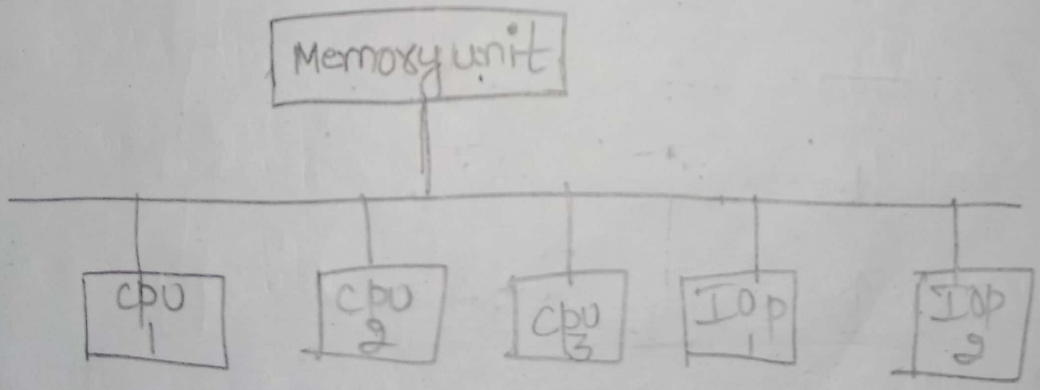


fig (a) Time shared common bus organisation

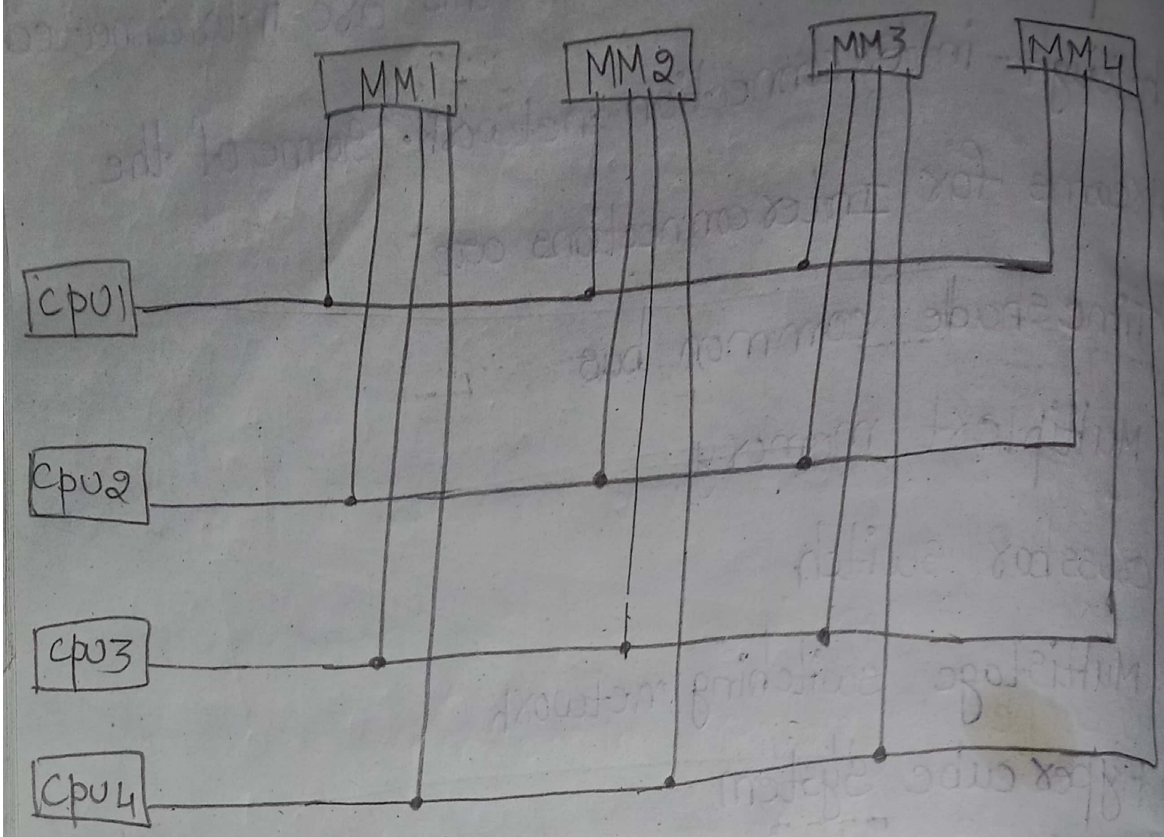


fig (b) Multiport Memory organisation

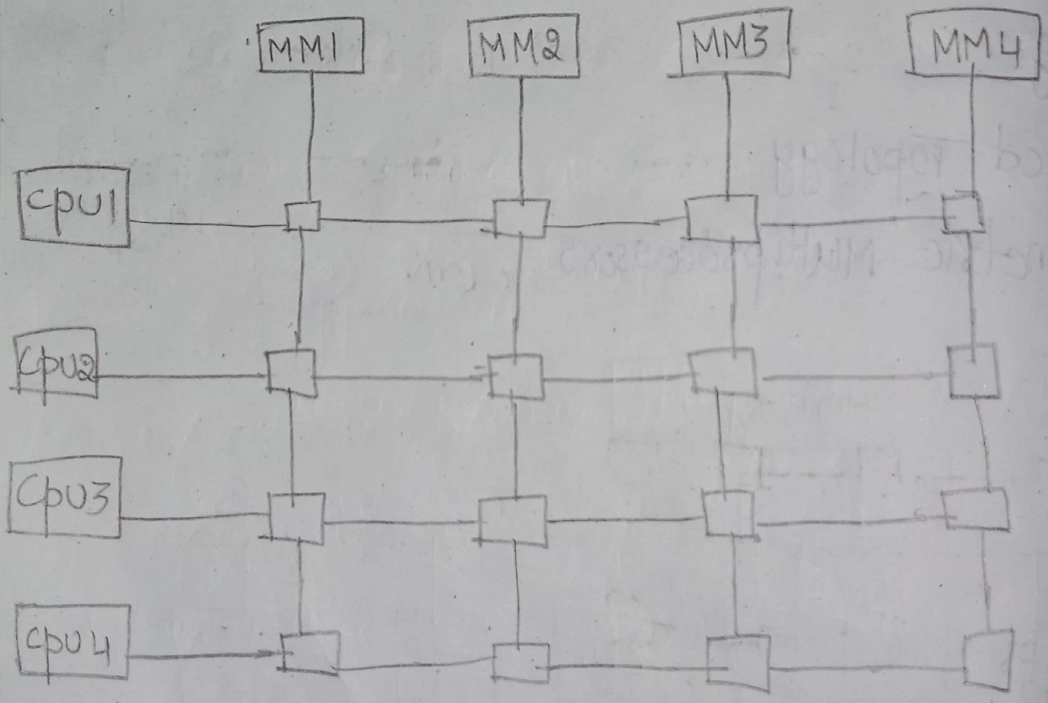
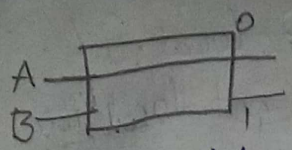
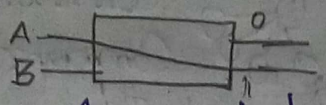


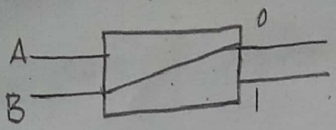
fig (c) cross bus switch



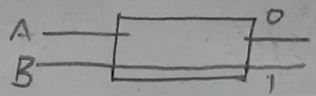
A connected to 0



A connected to 1



B connected to 0



B connected to 1

fig (4) Multistage switch

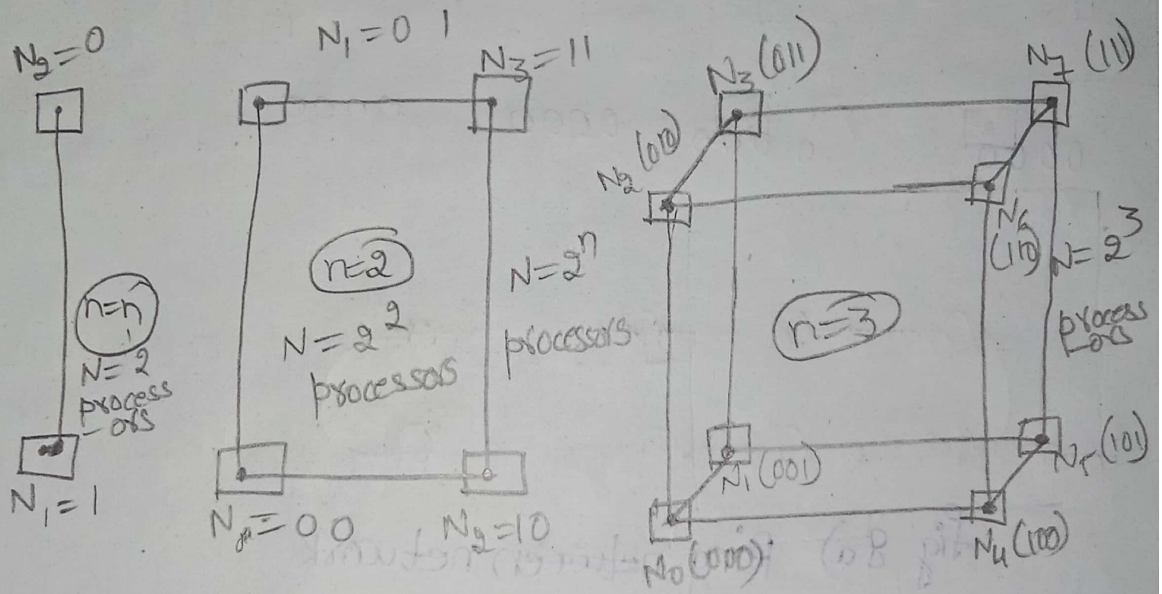


fig (5) Hype cube inter connection

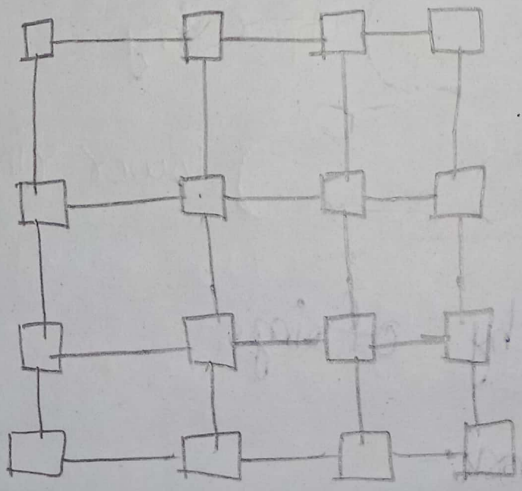
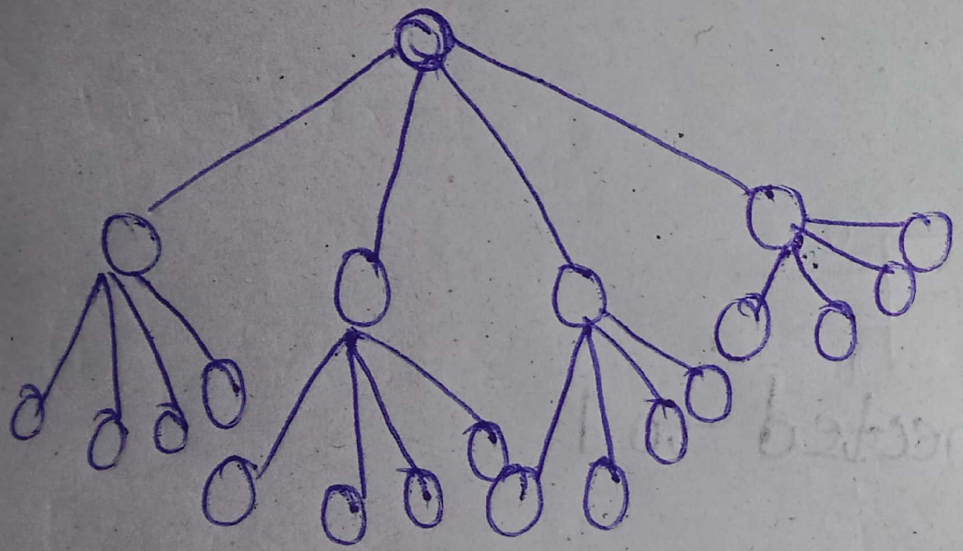
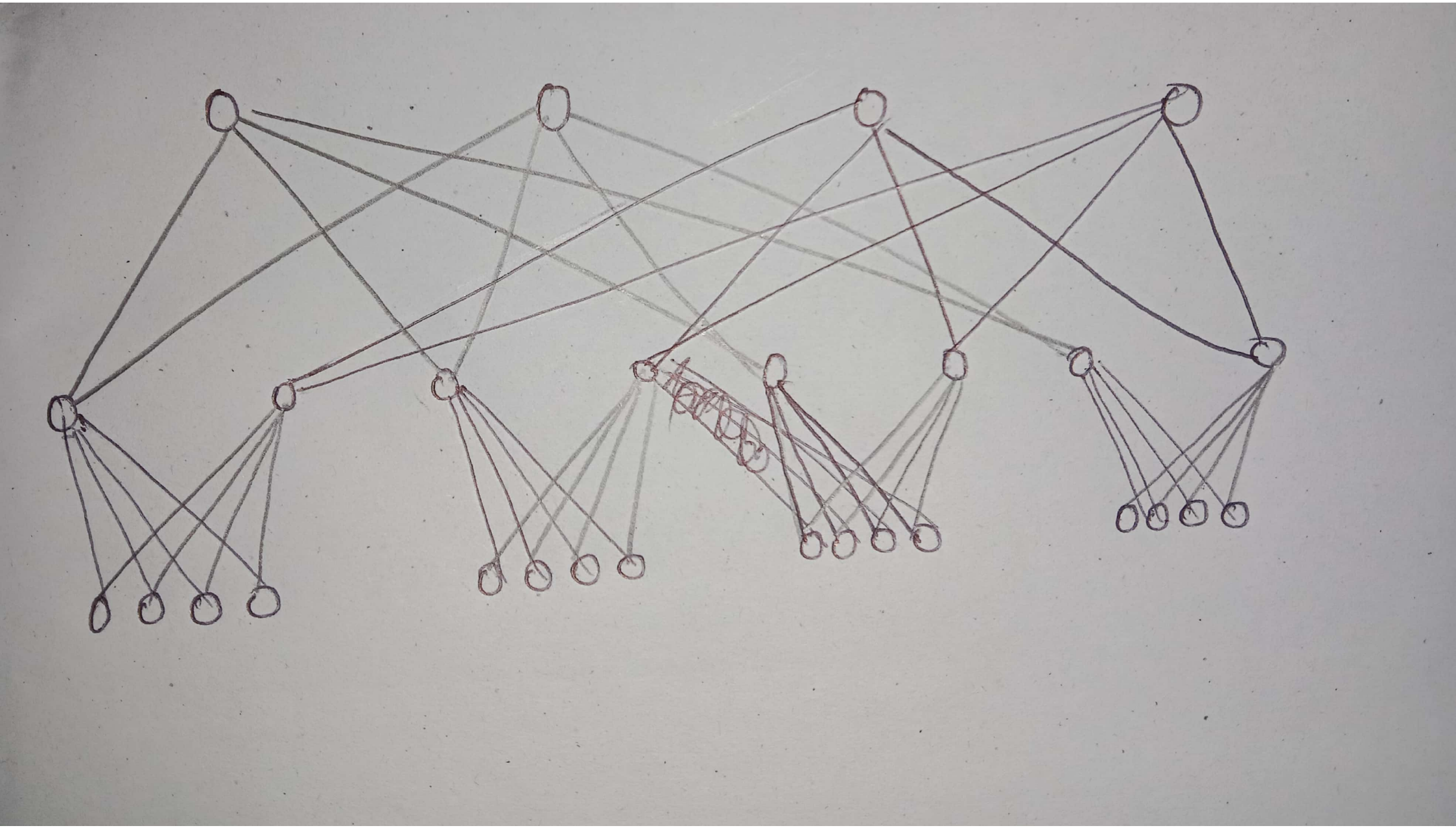


fig (6) Dimensional Mesh inter connection



Performance is good when the Communication is local. If Communication is not local, the buttle neck Problem arises.

fig 7) :- 4 Way tree.



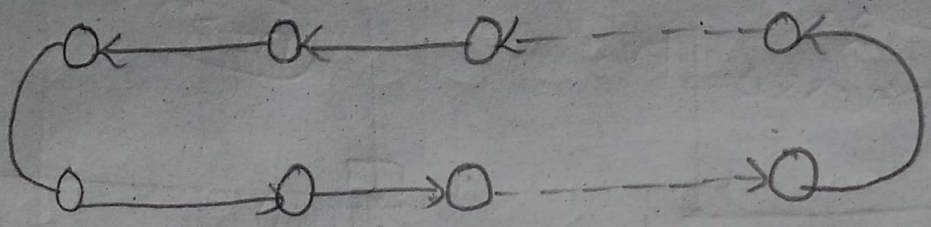


fig 8a) Ring network

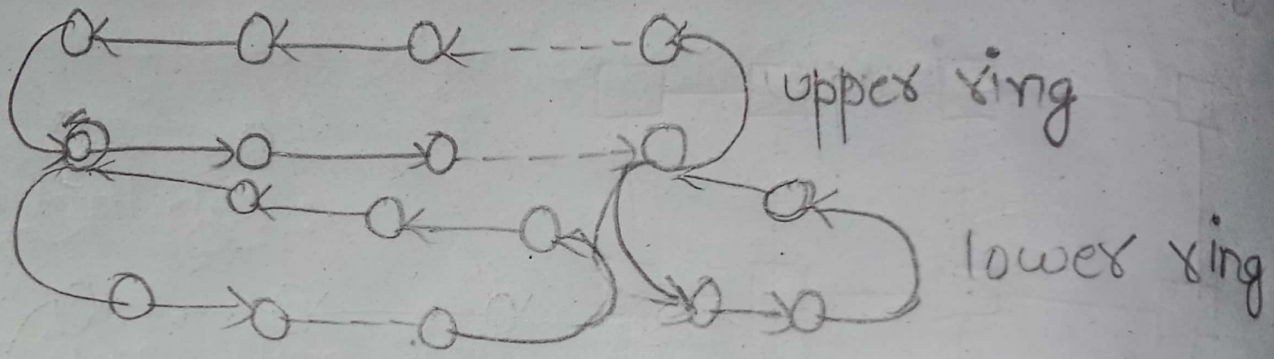


fig 8b) Hierarchy of rings

Mixed Topology network:-

Multiple buses with separate address lines Data lines control lines and specific switches like crossbar

switch, multistage switch are used with any of the topology discussed before.

Theory for each diagram:-

① Time shared common bus organisation:-

Here all the processors & memory unit is connected to a common bus. The processors which wants to use the bus for data transfer with memory unit places a request to grant the bus. Among all those processor the the processor which is having highest priority is allowed to use the bus and it performs data transfer (R/W) operation with memory unit.

When any of the processor is using the bus, the bus becomes busy and other processors are not allowed to use the bus.

Hence when the bus is busy then all other processors which want to perform data transfer should wait till the bus becomes free.

Advantages

① This type of interconnection network is simple in orginisation i.e. simple to construct.

Disadvantage:-

① processor needs to wait if bus is busy, hence processor time gets waisted.

② Needs to allocate priorities to each processor.

② Multiport memory

Here the memory modules contains multiple number of ports equal to the number of processors. All the processor are connected to the every memory module.

Hence all the processors can access data from different memory modules simultaneously.

Advantages:-

① This ~~organisiat~~ organisation saves processor time

Disadvantages:-

This organisation is complex because every memory module should have multiple number of ports and multiple number of buses are required

③ Crossbar switch:-

To reduce the number of ports in each memory module and multiple number of buses in the previous organisation crossbar switch is used as shown in fig. The number of switches required is 16 for 4 processor & 4 memory modules. Each small square in the figure represent a crossbar switch.

To connect a processor to a specific memory module corresponding switchs are in closed, for example to connect cpu2 to memory module 3, switches 5, 6, 7, 3 are closed.

Advantages:-

① Every memory module only one part.

② Less number of busses are required.

Disadvantages:-

① Require more hardware i.e. many number of switches are to be included in the organisation.

④ Multistage Switching Network:-

Instead of using crossbar switch, hear multistage switch, are used. Each switch has two

input ports & two output port, Hence to each port one processor (or) memory module is connected. The fig shows the 2x2 switch where input A can be connected to output port 0 (or) 1 as required.

Advantages:-

① Less number of multistage switches are required in comparision with crossbar switches network.

Disadvantage:-

Multistage switch is bit complex hardware.

⑤ Hyper cube Interconnection:-

It is also called as n-cube multiprocessor. It is a loosely coupled system i.e. all the nodes represents processing elements, with every processor there will be a memory module attached.

$$N=2^n$$

'N' is number of processors

'n' i) is number of bits required to provide address to each node (processor) binary.

2) also n-dimensional cube.

3) also n is the number of processors directly connected to each processor.

4) also at most n Links and needed

for n=1;	N=2	to reach
n=2;	N=4	any other
n=3;	N=8	node

Advantages:-

1) Less number of busses are required Hence used in many machines like eg:- Intel

IPSC

It uses 7 dimensional cube.

1) concept of pipeling
2) Hazards, Types.
3) Handling of pipeline hazards.
4) Influences of Instruction set.

Note:- Important concept

Handling of hazards (i.e. how to avoid hazards) (Techniques used to avoid hazards) are

① To handle data hazards

(1) operand forwarding using hardware.

(2) Introducing Nop (No operation instruction) using software.

To handle instruction hazards

To avoid hazard due to unconditional branch instruction.

① A Instruction queue between fetch unit & Decode unit.

③ To avoid hazards due conditional branch

① Delay the branch.

② Rearranging the instructions.