

LECTURE NOTES ON

Digital Logic Design and Computer Organization

Department of Information Technology

UNIT-1 PART-1

Basic Structure of Computers:

1.1.1 Computer Types

1.1.2 Functional units

1.1.3 Basic operational concepts

1.1.4 Bus structures

1.1.5 Software

1.1.6 Performance

1.1.7 multiprocessors and multi computers

1.1.8 Computer Generations.

Data Representation: Binary Numbers, Fixed Point Representation .Floating Point Representation. Number base conversions, Octal and Hexadecimal Numbers, components, Signed binary numbers, Binary codes.

1.1.1 Computer types

A computer can be defined as a fast electronic calculating machine that accepts the (data) digitized input information process it as per the list of internally stored instructions and produces the resulting information.

List of instructions are called programs & internal storage is called computer memory.

The different types of computers are

1. **Personal computers:** - This is the most common type found in homes, schools, Business offices etc., It is the most common type of desk top computers with processing and storage units along with various input and output devices.
2. **Note book computers:** - These are compact and portable versions of PC
3. **Work stations:** - These have high resolution input/output (I/O) graphics capability, but with same dimensions as that of desktop computer. These are used in engineering applications of interactive design work.
4. **Enterprise systems:** - These are used for business data processing in medium to large corporations that

require much more computing power and storage capacity than work stations. Internet associated with

servers have become a dominant worldwide source of all types of information.

5. **Super computers:** - These are used for large scale numerical calculations required in the applications like weather forecasting etc.,

1.1.2. Functional unit

A computer consists of five functionally independent main parts input, memory, arithmetic logic unit (ALU), output and control unit.

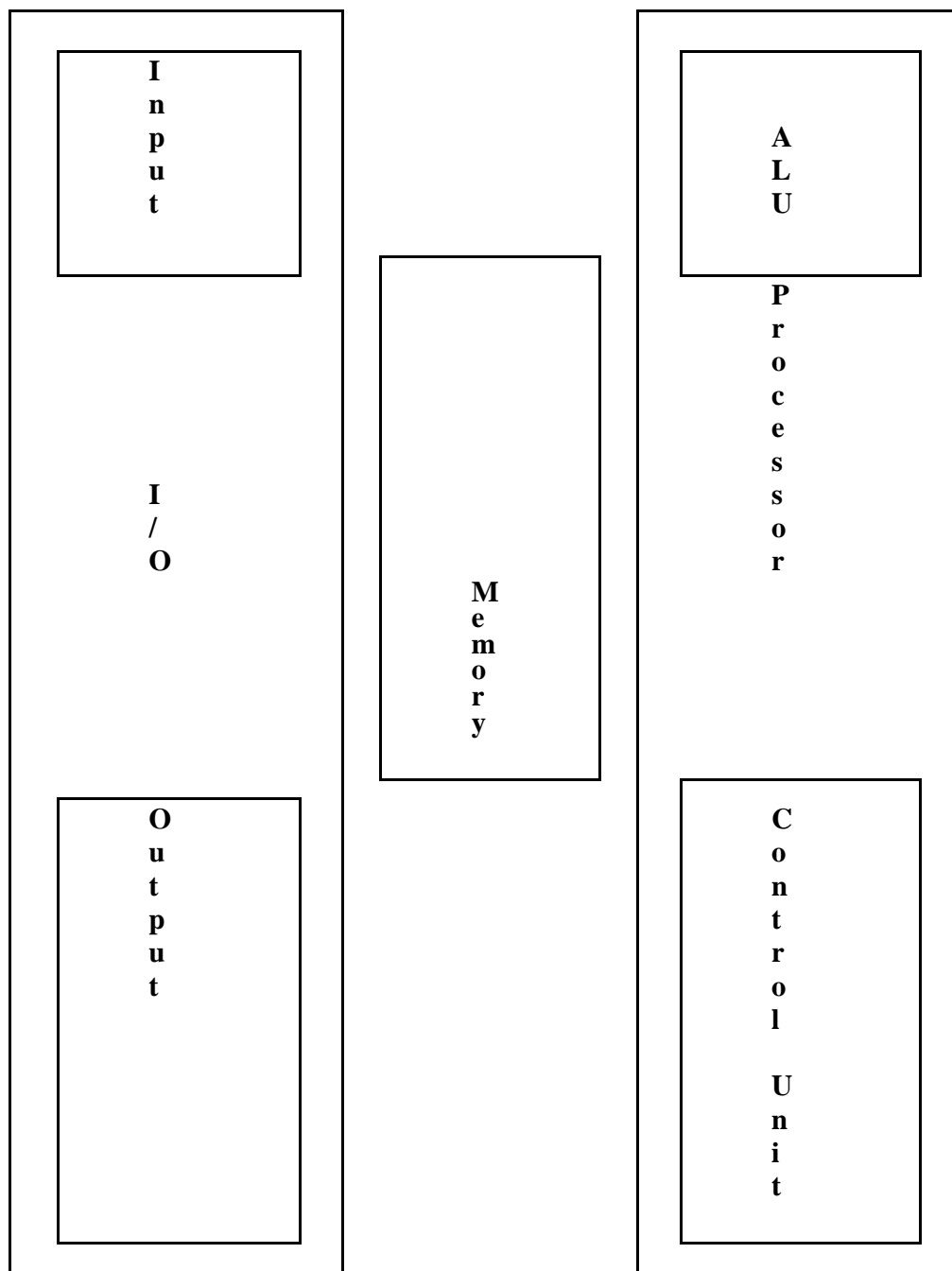


Fig a : Functional units of computer

Input device accepts the coded information as source program i.e. high level language. This is either stored in the memory or immediately used by the processor to perform the desired operations. The program stored in the memory determines the processing steps. Basically the computer converts one source program to an object program. i.e. into machine language.

Finally the results are sent to the outside world through output device. All of these actions are coordinated by the control unit.

Input unit: -

The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor.

Joysticks, trackballs, mouse, scanners etc are other input devices.

Memory unit: -

Its function into store programs and data. It is basically to two types

6. **Primary memory**
7. **Secondary memory**

Primary memory: - Is the one exclusively associated with the processor and operates at the electronics speeds

programs must be stored in this memory while they are being executed. The memory contains a large number of semiconductor storage cells. Each capable of storing one bit of information. These are processed in a group of fixed size called word.

To provide easy access to a word in memory, a distinct address is associated with each word location. **Addresses are** numbers that identify memory location.

Number of bits in each word is called word length of the computer. Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of processor.

Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called random-access memory (RAM).

The time required to access one word is called memory access time. Memory which is only readable by the user and contents of which can't be altered is called read only memory (ROM) it contains operating system.

Caches are the small fast RAM units, which are coupled with the processor and are often contained on the same IC chip to achieve high performance. Although primary storage is essential it tends to be expensive.

2 Secondary memory: - Is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently.

Examples: - Magnetic disks & tapes, optical disks (ie CD-ROM's), floppies etc.,

Arithmetic logic unit (ALU):-

Most of the computer operators are executed in ALU of the processor like addition, subtraction, division, multiplication, etc. the operands are brought into the ALU from memory and stored in high speed storage elements called register. Then according to the instructions the operation is performed in the required sequence.

The control and the ALU are many times faster than other devices connected to a computer system. This enables a single processor to control a number of external devices such as key boards, displays, magnetic and optical disks, sensors and other mechanical controllers.

Output unit:-

These actually are the counterparts of input unit. Its basic function is to send the processed results to the outside world.

Examples:- Printer, speakers, monitor etc.

Control unit:-

It effectively is the nerve center that sends signals to other units and senses their states. The actual timing signals that govern the transfer of data between input unit,

processor, memory and output unit are generated by the control unit.

1.1.3 Basic operational concepts

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 stored are also stored in the memory.

Examples: - Add LOCA, R0

This instruction adds the operand at memory location LOCA, to operand in register R0 & places the sum into register. This instruction requires the performance of several steps,

1.First the instruction is fetched from the memory into the processor. 2.The operand at LOCA is fetched and added to the contents of R0

3.Finally the resulting sum is stored in the register R0

The preceding add instruction combines a memory access operation with an ALU Operations. In some other type of computers, these two types of operations are performed by separate instructions for performance reasons.

Load LOCA, R1 Add R1, R0

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory.

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

The instruction register (IR):- Holds the instructions that is currently being executed. Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

The program counter PC:-

This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed.

The other two registers which facilitate communication with memory are: -

1.MAR – (Memory Address Register):- It holds the address of the location to be accessed.

2.MDR – (Memory Data Register):- It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
2. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
3. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
4. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
5. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
6. After one or two such repeated cycles, the ALU can perform the desired operation.
7. If the result of this operation is to be stored in the memory, the result is sent to MDR.
8. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
9. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal.

An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

The Diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue.

1.1.4 Bus structure

The simplest and most common way of interconnecting various parts of the

computer. To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time. A group of lines that serve as a connecting port for several devices is called a bus.

In addition to the lines that carry the data, the bus must have lines for address and control purpose. Simplest way to interconnect is to use the single bus as shown

Since the bus can be used for only actively use the bus at any given time. Bus requests for use of one bus, one transfer at a time, only two units can control lines are used to arbitrate multiple

Single bus structure is Low cost
Very flexible for attaching peripheral devices

Multiple bus structure certainly increases, the performance but also increases the cost significantly.

All the interconnected devices are not of same speed & time, leads to a bit of a problem. This is solved by using cache registers (ie buffer registers). These buffers are electronic registers of small capacity when compared to the main memory but of comparable speed.

The instructions from the processor at once are loaded into these buffers and then the complete transfer of data at a fast rate will take place.

1.1.5 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 program is affected by the design of its hardware. For best performance, it is necessary to design the compiles, the machine instruction set, and the hardware in a coordinated way.

The total time required to execute the program is elapsed time is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer. The time needed to execute a instruction is called the processor time.

Just as the elapsed time for the execution of a program depends on all units in a computer system, the processor time depends on the hardware involved in the execution of individual machine instructions. This hardware comprises the processor and the memory which are usually connected by the bus as shown in the fig c.

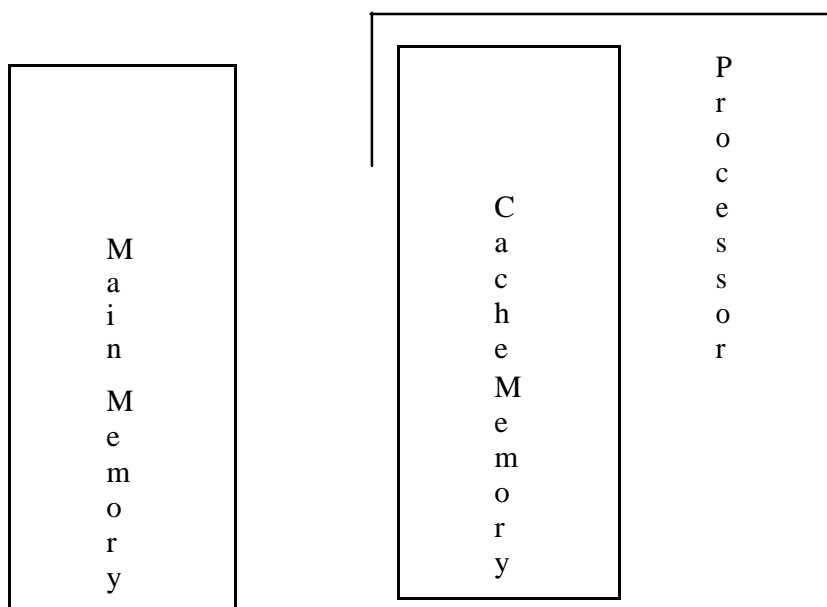


Fig d: The processor cache

Let us examine the flow of program instructions and data between the memory and the processor. At the start of execution, all program instructions and the required data are stored in the main memory. As the execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache later if the same instruction or data item is needed a second time, it is read directly from the cache.

The processor and relatively small cache memory can be fabricated on a single IC chip. The internal speed of performing the basic steps of instruction processing on chip is very high and is considerably faster than the speed at which the instruction and data can be fetched from the main memory. A program will be executed faster if the

movement of instructions and data between the main memory and the processor is minimized, which is achieved by using the cache.

For example:- Suppose a number of instructions are executed repeatedly over a short period of time 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. The same applies to the data that are used repeatedly.

Processor clock: -

Processor circuits are controlled by a timing signal called clock. The clock designer the regular time intervals called clock cycles. To execute a machine instruction the processor divides the action to be performed into a sequence of basic steps that each step can be completed in one clock cycle. The length P of one clock cycle is an important parameter that affects the processor performance.

Processor used in today's personal computer and work station have a clock rates that range from a few hundred million to over a billion cycles per second.

1.1.6 Basic performance equation

We now focus our attention on the processor time component of the total elapsed time. Let 'T' be the processor time required to execute a program that has been prepared in some high-level language. The compiler generates a machine language object program that corresponds to the source program. Assume that complete execution of the program requires the execution of N machine cycle language instructions. The number N is the actual number of instruction execution and is not necessarily equal to the number of machine cycle instructions in the object program. Some instruction may be executed more than once, which in the case for instructions inside a program loop others may not be executed all, depending on the input data used

Suppose that the average number of basic steps needed to execute one machine cycle instruction is S, where each basic step is completed in one clock cycle. If clock rate is 'R' cycles per second, the program execution time is given by

$$T = \frac{N \times S}{R}$$

this is often referred to as the basic performance equation.

We must emphasize that N, S & R are not independent parameters changing one may affect another. Introducing a new feature in the design of a processor will lead to improved performance only if the overall result is to reduce the value of T.

Pipelining and super scalar operation: -

We assume that instructions are executed one after the other. Hence the value of S is the total number of basic steps, or clock cycles, required to execute one instruction. A substantial improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called pipelining.

Consider Add R₁ R₂ R₃

This adds the contents of R_1 & R_2 and places the sum into R_3 .

The contents of R_1 & R_2 are first transferred to the inputs of ALU. After the addition operation is

performed, the sum is transferred to R_3 . The processor can read the next instruction from the memory, while the addition operation is being performed. Then of that instruction also uses, the ALU, its operand can be transferred to the ALU inputs at the same time that the add instructions is being transferred to R_3 .

In the ideal case if all instructions are overlapped to the maximum degree possible the execution proceeds at

the rate of one instruction completed in each clock cycle. Individual instructions still require several clock cycles to complete. But for the purpose of computing T , effective value of S is 1.

A higher degree of concurrency can be achieved if multiple instructions pipelines are implemented in the processor. This means that multiple functional units are used creating parallel paths through which different instructions can be executed in parallel with such an arrangement, it becomes possible to start the execution of several instructions in every clock cycle. This mode of operation is called superscalar execution. If it can be sustained for a long time during program execution the effective value of S can be reduced to less than one. But the parallel execution must preserve logical

correctness of programs, that is the results produced must be same as those produced by the serial execution of program instructions. Now a days many processors are designed in this manner.

1.1.7 Clock rate

These are two possibilities for increasing the clock rate ' R '.

1. Improving the IC technology makes logical circuit faster, which reduces the time of execution of basic steps. This allows the clock period P , to be reduced and the clock rate R to be increased.
2. Reducing the amount of processing done in one basic step also makes it possible to reduce the clock period P . However if the actions that have to be performed by an instructions remain the same, the number of basic steps needed may increase.

Increase in the value ' R ' that are entirely caused by improvements in IC technology affects all aspects of the processor's operation equally with the exception of the time it takes to access the main memory. In the presence of cache the percentage of accesses to the main memory is small. Hence much of the performance gain expected from the use of faster technology can be realized.

Instruction set CISC & RISC:-

Simple instructions require a small number of basic steps to execute. Complex instructions involve a large number of steps. For a processor that has only simple instructions a large number of instructions may be needed to perform a given programming task. This could lead to a large value of ' N ' and a small value of ' S '. On the other hand if individual instructions perform more complex operations, a fewer instructions will be needed, leading to a lower value of N and a larger value of S . It is not obvious if one choice is better than the other.

But complex instructions combined with pipelining (effective value of $S \approx 1$) would achieve one best performance. However, it is much easier to implement efficient pipelining in processors with simple instruction sets.

1.1.8 Performance measurements

It is very important to be able to access the performance of a computer, computer designers use performance estimates to evaluate the effectiveness of new features.

The previous argument suggests that the performance of a computer is given by the

execution time T , for the program of interest.

Inspite of the performance equation being so simple, the evaluation of 'T' is highly complex. Moreover the parameters like the clock speed and various architectural features are

not reliable indicators of the expected performance.

Hence measurement of computer performance using bench mark programs is done to make comparisons possible, standardized programs must be used.

The performance measure is the time taken by the computer to execute a given bench mark. Initially some attempts were made to create artificial programs that could be used as bench mark programs. But synthetic programs do not properly predict the performance obtained when real application programs are run.

A non profit organization called SPEC- system performance evaluation corporation selects and publishes bench marks.

The program selected range from game playing, compiler, and data base applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled under test, and the running time on a real computer is measured. The same program is also compiled and run on one computer selected as reference.

The 'SPEC' rating is computed as follows.

Running time on the reference computer

SPEC rating =

Running time on the computer under test If the SPEC rating =

50

Means that the computer under test is 50 times as fast as the ultra sparc 10. This is repeated for all the programs in the SPEC suit, and the geometric mean of the result is computed.

Let SPEC_i be the rating for program 'i' in the suite. The overall SPEC rating for the computer is given by

$$\text{SPEC rating} = \left(\prod_{i=1}^n \text{SPEC}_i \right)^{1/n}$$

Where 'n' = number of programs in suite.

Since actual execution time is measured the SPEC rating is a measure of the combined effect of all factors affecting performance, including the compiler, the OS, the processor, the memory of comp being tested.

1.1.9. Multiprocessor & microprocessors:-

Large computers that contain a number of processor units are called multiprocessor system.

These systems either execute a number of different application tasks in parallel or execute subtasks of a single large task in parallel.

All processors usually have access to all memory locations in such system & hence they are called shared memory multiprocessor systems.

The high performance of these systems comes with much

increased complexity and cost.

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

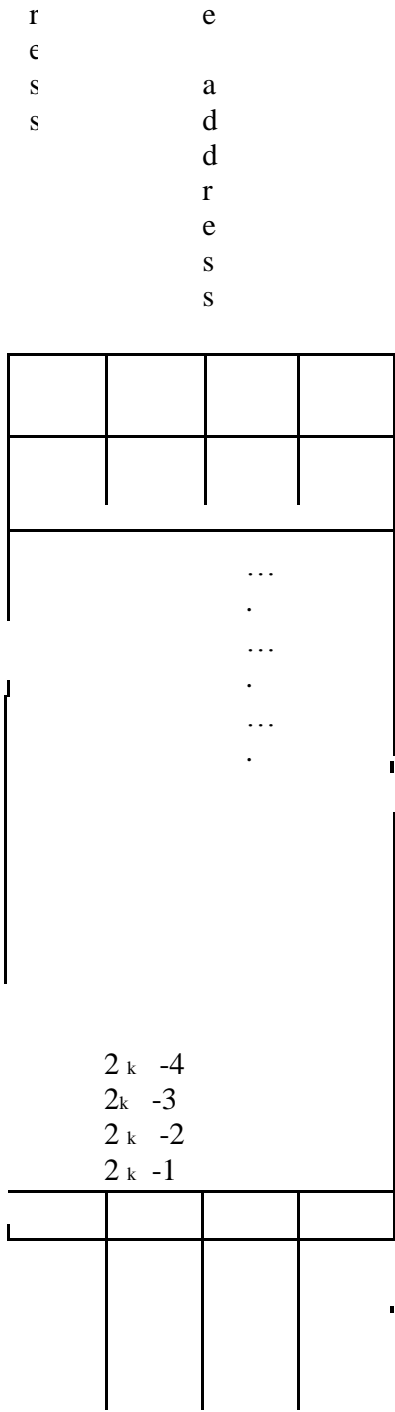
BIG-ENDIAN AND LITTLE-ENDIAN ASSIGNMENTS:-

There are two ways that byte addresses can be assigned across words, as shown in fig b. 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.

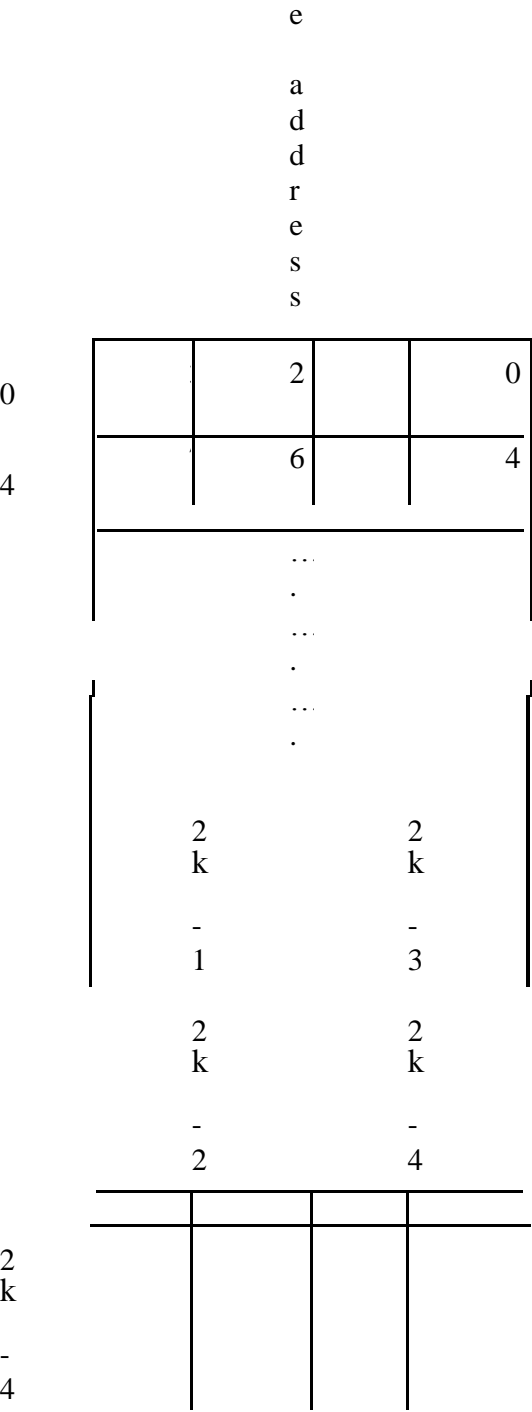
In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The same ordering is

also used for labeling bits within a byte, that is, b₇, b₆, ..., b₀, from left to right. Word

7	B	B
6	y	y
5	t	t



(a) Big-endian assignment



(b) Little-endian assignment

1.1.10. BUSES:

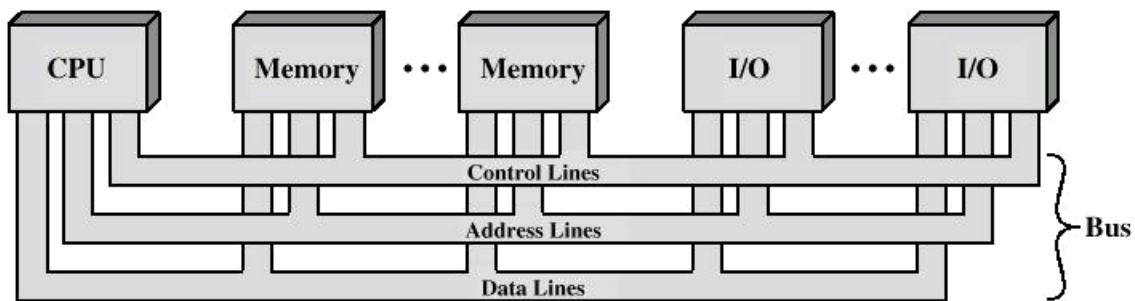
- There are a number of possible interconnection systems
- Single and multiple BUS structures are most common
- e.g. Control/Address/Data bus (PC)
- e.g. Unibus (DEC-PDP)

WHAT IS A BUS

- A communication pathway connecting two or more devices
- Usually broadcast (all components see signal)
- Often grouped
- A number of channels in one bus
- e.g. 32 bit data bus is 32 separate single bit channels
- Power lines may not be shown
-

BUS INTER CONNECTION

SCHEME



DATA BUS:

- Carries data
- Remember that there is no difference between “data” and “instruction” at this level
- Width is a key determinant of performance
- 8, 16, 32, 64 bit

ADDRESS BUS:

- Identify the source or destination of data
- e.g. CPU needs to read an instruction (data) from a given location in memory
- Bus width determines maximum memory capacity of system
- e.g. 8080 has 16 bit address bus giving 64k address space

CONTROL BUS:

- Control and timing information
- Memory read/write signal
- Interrupt request
- Clock signals

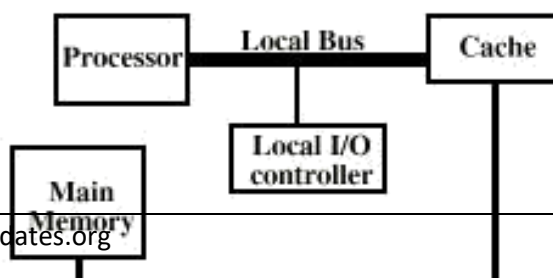
What do buses look like?

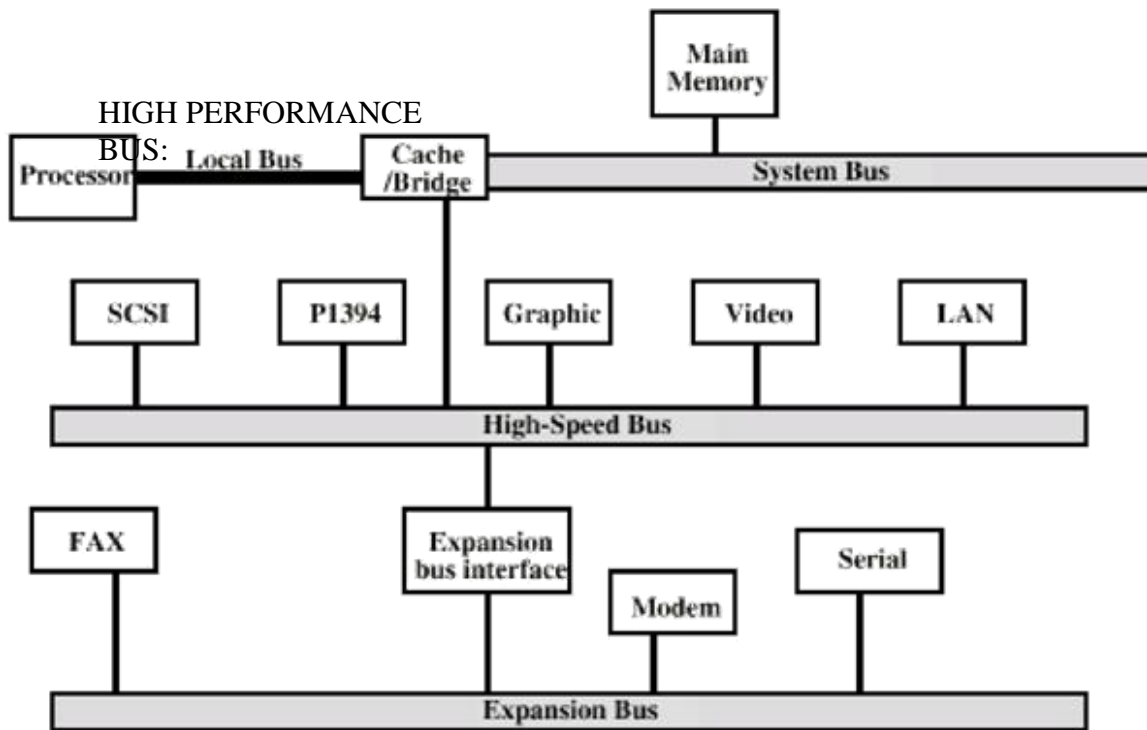
- Parallel lines on circuit boards
- Ribbon cables
- Strip connectors on mother boards
- e.g. PCI Sets of wires

SINGLE BUS PROBLEMS:

- Lots of devices on one bus leads to:
 - Propagation delays
 - Long data paths mean that co-ordination of bus use can adversely affect performance
 - If aggregate data transfer approaches bus capacity
- Most systems use multiple buses to overcome these problems

TRADITIONAL ISA WITH CACHE:





BUS TYPES:

- Dedicated
 - Separate data & address lines
- Multiplexed
 - Shared lines
 - Address valid or data valid control line
 - Advantage - fewer lines
 - Disadvantages
 - More complex control
 - Ultimate performance

BUS ARBITRATION:

- More than one module controlling the bus

— e.g. CPU and DMA controller

- Only one module may control bus at one time

- Arbitration may be centralised or distributed

CENTRALIZED

ARBITRATION:

- Single hardware device controlling bus access

— Bus Controller

- Arbiter

- May be part of CPU or separate

DISTRIBUTED ARBITRATION:

- Each module may claim the bus
 - Control logic on all modules PCI BUS:
- Peripheral Component Interconnection (PCI)
- Intel released to public domain
- 32 or 64 bit
- 50 lines

PCI BUS LINES(REQUIRED)

- Systems lines
 - Including clock and reset
- Address & Data
 - 32 time mux lines for address/data
 - Interrupt & validate lines
- Interface Control
- Arbitration
 - Not shared
 - Direct connection to PCI bus arbiter
- Error lines

PCI BUS LINES (OPTIONAL)

- Interrupt lines
 - Not shared
- Cache support
- 64-bit Bus Extension
 - Additional 32 lines
 - Time multiplexed
 - 2 lines to enable devices to agree to use 64-bit transfer
- JTAG/Boundary Scan

For testing procedures

PCI COMMANDS:

- Transaction between initiator (master) and target
- Master claims bus
- Determine type of transaction
 - e.g. I/O read/write
- Address phase
- One or more data phases

MULTIPROCESSOR AND MULTICOMPUTERS

Two categories of parallel computers are discussed below namely shared common memory or unshared distributed

memory.

Shared memory multiprocessors

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all

memory as global address space.

- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
 - Shared memory machines can be divided into two main classes based upon memory access times: UMA , NUMA and COMA. Uniform Memory Access (UMA):
- Most commonly represented today by Symmetric Multiprocessor (SMP) machines
- Identical processors
- Equal access and access times to memory
 - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

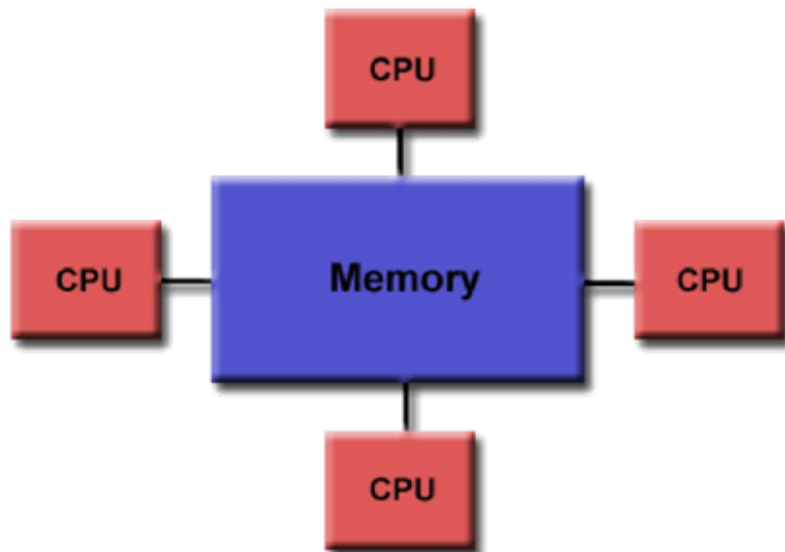


Figure 1.9 Shared Memory (UMA)

Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower

If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

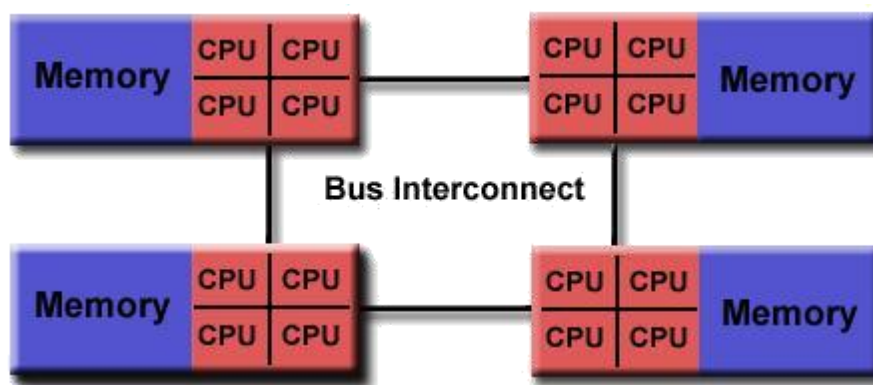


figure 1.10 Shared Memory (NUMA)

The COMA model : The COMA model is a special case of NUMA machine in which the distributed main memories are converted to caches. All caches form a global address space and there is no memory hierarchy at each processor node.

Advantages:

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:

- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increase traffic on the shared memory CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
 - Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

1.3.2 Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.

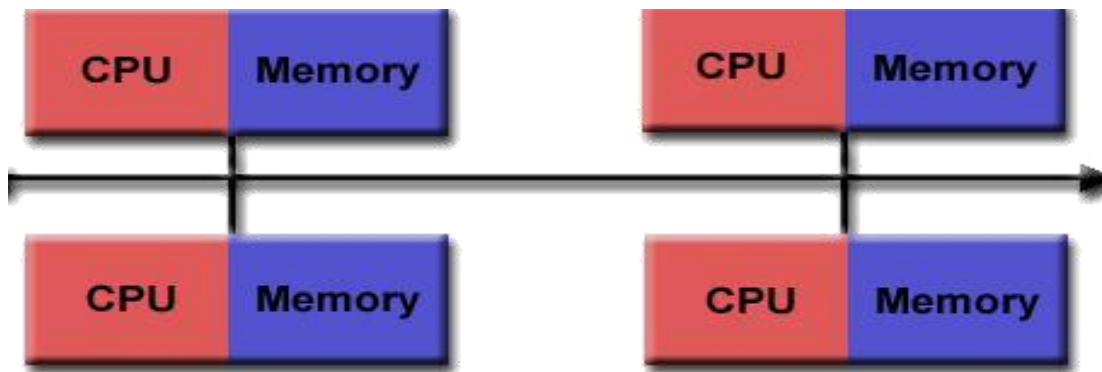


Figure distributed memory systems

- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- Modern multicomputer use hardware routers to pass message. Based on the interconnection and routers and channel used the multicomputers are divided into generation
 - o 1st generation : based on board technology using hypercube architecture and software controlled message switching.

- o 2nd Generation: implemented with mesh connected architecture, hardware message routing and software environment for medium distributed – grained computing.
- o 3rd Generation : fine grained multicomputer like MIT J-Machine.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

Advantages:

- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access (NUMA) times

1.2 The state of computing

Modern computers are equipped with powerful hardware technology at the same time loaded with sophisticated software packages. To access the art of computing we firstly review the history of computers then study the attributes used for analysis of performance of computers.

1.2.1 Evolution of computer system

Presently the technology involved in designing of its hardware components of computers and its overall architecture is changing very rapidly for example: processor clock rate increase about 20% a year, its logic capacity improve at about 30% in a year; memory speed at increase about 10% in a year and memory capacity at about 60% increase a year also the disk capacity increase at a 60% a year and so overall cost per bit improves about 25% a year.

But before we go further with design and organization issues of parallel computer architecture it is necessary to understand how computers had evolved. Initially, man used simple mechanical devices – abacus (about 500 BC) , knotted string, and the slide rule for computation. Early computing was entirely mechanical like : mechanical adder/subtractor (Pascal, 1642) difference engine design (Babbage, 1827) binary mechanical computer (Zuse, 1941) electromechanical decimal machine (Aiken, 1944). Some of these machines used the idea of a stored program a famous example of it is the Jacquard Loom and Babbage's Analytical Engine which is also often considered as the first real computer.

Mechanical and electromechanical machines have limited speed and reliability because of the many moving parts. Modern machines use electronics for most information transmission.

Computing is normally thought of as being divided into generations. Each successive generation is marked by sharp changes in hardware and software technologies. With

some exceptions, most of the advances introduced in one generation are carried through to later generations. We are currently in the fifth generation.

1st generation of computers (1945-54)

The first generation computers were based on vacuum tube technology. The first large electronic computer was **ENIAC** (Electronic Numerical Integrator and Calculator), which used high speed vacuum tube technology and were designed primarily to calculate the trajectories of missiles. They used separate memory block for program and data. Later in 1946 John Von Neumann introduced the concept of stored program, in which data and program were stored in same memory block. Based on this concept **EDVAC** (Electronic Discrete Variable

Automatic Computer) was built in 1951. On this concept IAS (Institute of advance studies, Princeton) computer was built whose main characteristic was CPU consist of two units (Program flow control and execution unit).

In general key features of this generation of computers where

- 1) The switching device used where vacuum tube having switching time between 0.1 to 1 milliseconds.
- 2) One of major concern for computer manufacturer of this era was that each of the computer designs had a unique design. As each computer has unique design one cannot upgrade or replace one component with other computer. Programs that were written for one machine could not execute on another machine, even though other computer was also designed from the same company. This created a major concern for designers as there were no upward-compatible machines or computer architectures with multiple, differing implementations. And designers always tried to manufacture a new machine that should be upward compatible with the older machines.
- 3) Concept of specialized registers where introduced for example index registers were introduced in the Ferranti Mark I, concept of register that save the return-address instruction was introduced in UNIVAC I, also concept of immediate operands in IBM 704 and the detection of invalid operations in IBM 650 were introduced.
- 4) Punch card or paper tape were the devices used at that time for storing the program. By the end of the 1950s IBM 650 became one of popular computers of that time and it used the drum memory on which programs were loaded from punch card or paper tape. Some high-end machines also introduced the concept of core memory which was able to provide higher speeds. Also hard disks started becoming popular.
- 5) In the early 1950s as said earlier were design specific hence most of them were designed for some particular numerical processing tasks. Even many of them used decimal numbers as their base number system for designing instruction set. In such machine there were actually ten vacuum tubes per digit in each register.
- 6) Software used was machine level language and assembly language.
- 7) Mostly designed for scientific calculation and later some systems were developed for simple business systems.
- 8) Architecture features Vacuum tubes and relay memories CPU driven by a program counter (PC) and accumulator Machines had only fixed-point arithmetic

9) Software and Applications

Machine and assembly language Single user at a time

No subroutine linkage mechanisms Programmed I/O required continuous use of CPU

10) examples: ENIAC, Princeton IAS, IBM 701

IInd generation of computers (1954 – 64)

The transistors were invented by Bardeen, Brattain and Shockely in 1947 at Bell Labs and by the 1950s these transistors made an electronic revolution as the transistor is smaller, cheaper and dissipate less heat as compared to vacuum tube. Now the transistors were used instead of a vacuum tube to construct computers. Another major invention was invention of magnetic cores for storage. These cores where used to large

random access memories. These generation computers has better processing speed, larger memory capacity, smaller size as compared to pervious generation computer.

The key features of this generation computers were

- 1) The IInd generation computer were designed using Germanium transistor, this technology was much more reliable than vacuum tube technology.
- 2) Use of transistor technology reduced the switching time 1 to 10 microseconds thus provide overall speed up.
- 2) Magnetic cores were used main memory with capacity of 100 KB. Tapes and disk peripheral memory were used as secondary memory.
- 3) Introduction to computer concept of instruction sets so that same program can be executed on different systems.
- 4) High level languages, FORTRAN, COBOL, Algol, BATCH operating system.
- 5) Computers were now used for extensive business applications, engineering design, optimation using Linear programming, Scientific research
- 6) Binary number system very used.
- 7) Technology and Architecture
 - Discrete transistors and core memories I/O processors, multiplexed memory access
 - Floating-point arithmetic available
 - Register Transfer Language (RTL) developed
- 8) Software and Applications
 - High-level languages (HLL): FORTRAN, COBOL, ALGOL with compilers and subroutine libraries
 - Batch operating system was used although mostly single user at a time
- 9) Example : CDC 1604, UNIVAC LARC, IBM 7090

IIIrd Generation computers(1965 to 1974)

In 1950 and 1960 the discrete components (transistors, registers capacitors) were manufactured packaged in a separate containers. To design a computer these discrete unit were soldered or wired together on a circuit boards. Another revolution in computer designing came when in the 1960s, the Apollo guidance computer and Minuteman missile were able to develop an integrated circuit (commonly called ICs). These ICs made the circuit designing more economical and practical. The IC based computers are called third generation computers. As integrated circuits, consists of transistors, resistors, capacitors on single chip eliminating wired interconnection, the space required for the computer was greatly reduced. By the mid-1970s, the use of ICs in computers became very common. Price of transistors reduced very greatly. Now it

was possible to put all components required for designing a CPU on a single printed circuit board. This advancement of technology resulted in development of minicomputers, usually with 16-bit words size these system have a memory of

range of 4k to 64K. This began a new era of microelectronics where it could be possible design small identical chips (a thin wafer of silicon's). Each chip has many gates plus number of input output pins.

Key features of IIIrd Generation computers:

- 1) The use of silicon based ICs, led to major improvement of computer system. Switching speed of transistor went by a factor of 10 and size was reduced by a factor of 10, reliability increased by a factor of 10, power dissipation reduced by a factor of 10. This cumulative effect of this was the emergence of extremely powerful CPUS with the capacity of carrying out 1 million instruction per second.
- 2) The size of main memory reached about 4MB by improving the design of magnetic core memories also in hard disk of 100 MB become feasible.
- 3) On line system become feasible. In particular dynamic production control systems, airline reservation systems, interactive query systems, and real time closed loop process control systems were implemented.
- 4) Concept of Integrated database management systems were emerged.
- 5) 32 bit instruction formats
- 6) Time shared concept of operating system.
- 7) Technology and Architecture features
 - Integrated circuits (SSI/MSI) Microprogramming
 - Pipelining, cache memories, lookahead processing
- 8) Software and Applications
 - Multiprogramming and time-sharing operating systems
 - Multi-user applications
- 9) Examples : IBM 360/370, CDC 6600, TI ASC, DEC PDP-82

IVth Generation computer ((1975 to 1990)

The microprocessor was invented as a single VLSI (Very large Scale Integrated circuit) chip CPU. Main Memory chips of 1MB plus memory addresses were introduced as single VLSI chip. The caches were invented and placed within the main memory and microprocessor. These VLSIs and VVSLIs greatly reduced the space required in a computer and increased significantly the computational speed.

- 1) Technology and Architecture feature semiconductor memory Multiprocessors, vector supercomputers, multicomputers
Shared or distributed memory Vector processors
- 2) Software and Applications Multiprocessor operating systems, languages, compilers,

parallel software tools

Examples : VAX 9000, Cray X-MP, IBM 3090, BBN TC2000

Fifth Generation computers(1990 onwards)

In the mid-to-late 1980s, in order to further improve the performance of the system the designers start using a technique known as “instruction pipelining”. The idea is to break the program into small instructions and the processor works on these instructions in different stages of completion. For example, the processor while calculating the result of the current instruction also retrieves the operands for the next instruction. Based on this concept later superscalar processor were designed, here to execute multiple instructions in parallel we have multiple execution unit i.e., separate arithmetic-logic units (ALUs).

Now instead executing single instruction at a time, the system divide program into several independent instructions and now CPU will look for several similar instructions that are not dependent on each other, and execute them in parallel. The example of this design are VLIW and EPIC.

1) Technology and Architecture features ULSI/VHSIC processors, memory, and switches High-density packaging

Scalable architecture Vector processors

2) Software and Applications

Massively parallel processing Grand challenge applications Heterogenous processing

3) Examples : Fujitsu VPP500, Cray MPP, TMC CM-5, Intel Paragon

UNIT-2

PART-2

Data Representation:

1.2.1 Binary Numbers,

1.2.2 Fixed Point Representation .

1.2.3 Floating Point Representation.

1.2.4 Number base conversions

1.2.5 Octal and Hexadecimal Numbers,

1.2.6 Signed binary numbers,

1.2.7 Binary codes.

DATA REPRESENTATION

1.2.1. Binary Numbers:

The ancient Indian writer Pingala developed advanced mathematical concepts for describing prosody, and in doing so presented the first known description of a binary numeral system. A full set of 8 trigrams and 64 hexagrams, analogous to the 3-bit and 6-bit binary numerals, were known to the ancient Chinese in the classic text *I Ching*. An arrangement of the hexagrams of the *I Ching*, ordered according to the values of the corresponding binary numbers (from 0 to 63), and a method for generating the same, was developed by the Chinese scholar and philosopher Shao Yong in the 11th century.

In 1854, British mathematician George Boole published a landmark paper detailing an algebraic system of logic that would become known as Boolean algebra. His logical calculus was to become instrumental in the design of digital electronic circuitry. In 1937, Claude Shannon produced his master's thesis at MIT that implemented Boolean algebra and binary arithmetic using electronic relays and switches for the first time in history. Entitled *A Symbolic Analysis of Relay and Switching Circuits*, Shannon's thesis essentially founded practical digital circuit design.

The radix (no of symbols in a number system is called radix) is 2. The positive binary numbers are stored in binary notation of sign magnitude representation and negative numbers are stored in sign magnitude or 1's complement form or 2's complement form.

In 1's complement and sign magnitude representation we have +0 and -0,

In 2's complement representation we have only one zero. Hence 2's complement representation is used in computers to store the data.

1.2.2. FIXED POINT REPRESENTATION:

It is used to represent integers either positive or negative. They are Sign Magnitude representation, 1's complement representation and 2's complement representation

1.2.3 FLOATING POINT REPRESENTATION:

It is used to represent real numbers. It consists of mantissa which is fraction i.e. the first digit is a non zero digit for normalization condition and an exponent which is an integer. If the number of bits allocated for mantissa increases the accuracy of the number is increased, if the number of bits allocated for exponent increases the range of the number is increased.

1.2.4. NUMBER BASE CONVERSIONS

Any number in one base system can be converted into another base system Types

1) Decimal to any base

2) A

n
yb
a
s
et
od
e
c
i
m
a
l

3) A

n
yb
a
s
et
oA
n
yb
a
s
e

Decimal number: $123.45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$.

Base b number: $N = a_{q-1}b^{q-1} + \dots + a_1b^1 + a_0b^0 + \dots + a_{-p}b^{-p}$

$b > 1, \quad 0 \leq a_i \leq b-1$

Integer part: $a_{q-1}a_{q-2} \dots a_0$

Fractional part: $a_{-1}a_{-2} \dots a_{-p}$.

Most significant digit: $a_{q-1} \dots$

Least significant digit: a_{-p}

Binary number ($b=2$): $1101.01 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2}$

Representing number N in base b : $(N)_b = \dots \dots \dots$

D
e
c
i
m
a
l

t
o

B
i
n
a
r
y

Example: Convert $(432.354)_{10}$ to binary

Q_i	r_i		
216	$0 = a_0$	0.354	$2 = 0.708$ hence $a_{-1} = 0$
108	$0 = a_1$	0.708	$2 = 1.416$ hence $a_{-2} = 1$
54	$0 = a_2$	0.416	$2 = 0.832$ hence $a_{-3} = 0$
27	$0 = a_3$	0.832	$2 = 1.664$ hence $a_{-4} = 1$
13	$1 = a_4$	0.664	$2 = 1.328$ hence $a_{-5} = 1$
6	$1 = a_5$	0.328	$2 = 0.656$ hence $a_{-6} = 0$
3	$0 = a_6$.	$a_{-7} = 1$
1	$1 = a_7$		etc.
	$1 = a_8$		

Thus, $(432.354)_{10} = (110110000.0101101...)_{2}$

O
c
t
a
l

T
o

**B
i
n
a
r
y**

Example: Convert $(123.4)_8$ to binary

$$(123.4)_8 = (001\ 010\ 011.100)_2$$

Example: Convert $(1010110.0101)_2$ to octal

$$(1010110.0101)_2 = (001\ 010\ 110.010\ 100)_2 = (126.24)_8$$

1.2.5. OCTAL AND HEXADECIMAL NUMBERS:

Octal is a 3-bit binary and Hexadecimal is a 4-bit binary.

1.2.7. SIGNED BINARY NUMBERS

Signed binary numbers are represented in three ways. They are

- (a) Sign Magnitude representation
- (b) 1's complement representation
- (c) 2's complement representation.

In first two representations zero is represented with two different binary numbers whereas in third i.e. 2's complement representation zero is represented with only one binary number. Hence this representation is used in computer for storing the signed binary numbers.

1.2.8. Binary codes

Binary codes are codes which are represented in binary system with modification from the original ones. Weighted Binary codes

Non Weighted Codes

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example.

Decimal	BCD 8421	Excess-3	84-2-1	2421	5211	Bi-Quinary 5043210			5	0	4	3	2	1	0
0	0000	0011	0000	0000	0000	0100001		0		X					X
1	0001	0100	0111	0001	0001	0100010		1		X				X	
2	0010	0101	0110	0010	0011	0100100		2		X			X		
3	0011	0110	0101	0011	0101	0101000		3		X		X			
4	0100	0111	0100	0100	0111	0110000		4		X	X				
5	0101	1000	1011	1011	1000	1000001		5	X						X
6	0110	1001	1010	1100	1010	1000010		6	X					X	
7	0111	1010	1001	1101	1100	1000100		7	X				X		
8	1000	1011	1000	1110	1110	1001000		8	X			X			
9	1001	1111	1111	1111	1111	1010000		9	X		X				

Reflective Code

A code is said to be reflective when code for 9 is complement for the code for 0, and so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code is not.

Sequential Codes

A code is said to be sequential when two subsequent codes, seen as numbers in binary

representation, differ by one. This greatly aids mathematical manipulation of data. The 8421 and Excess-3 codes are sequential, whereas the 2421 and 5211 codes are not.

Non weighted codes

Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value. Ex: Excess-3 code

Excess-3 Code

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).

Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit- distance code. In digital Gray code has got a special place.

Decimal Number	Binary Code	Gray Code	Decimal Number	Binary Code	Gray Code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Binary to Gray Conversion

- Gray Code MSB is binary code MSB.
- Gray Code MSB-1 is the XOR of binary code MSB and MSB-1.
- MSB-2 bit of gray code is XOR of MSB-1 and MSB-2 bit of binary code.
- MSB-N bit of gray code is XOR of MSB-N-1 and MSB-N bit of binary code

UNIT-2

PART-1

Digital Logic Circuits-I:

2.1.1 Basic Logic Functions,

2.1.2 Logic gates, 2.1.3 universal logic gates,
2.1.4 Minimization of Logic expressions. 2.1.5 Flip-flops,

2.1.1 **Boolean algebra:**

Boolean algebra, like any other deductive mathematical system, may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates. A *set* of elements is any collection of objects having a common property. If S is a set and x and y are certain objects, then $x \in S$ denotes that x is a member of the set S , and $y \notin S$ denotes that y is not an element of S . A set with a denumerable number of elements is specified by braces: $A = \{1, 2, 3, 4\}$, i.e. the elements of set A are the numbers 1, 2, 3, and 4. A *binary operator* defined on a set S of elements is a rule that assigns to each pair of elements from S a unique element from S . Example: In $a * b = c$, we say that $*$ is a binary operator if it specifies a rule for finding c from the pair (a, b) and also if $a, b, c \in S$.

CLOSURE: The Boolean system is *closed* with respect to a binary operator if for every pair of Boolean values, it produces a Boolean result. For example, logical AND is closed in the Boolean system because it accepts only Boolean operands and produces only Boolean results.

_ A set S is closed with respect to a binary operator if, for every pair of elements of S , the binary operator specifies a rule for obtaining a unique element of S .

_ For example, the set of natural numbers $N = \{1, 2, 3, 4, \dots, 9\}$ is closed with respect to the binary operator plus (+) by the rule of arithmetic addition, since for any $a, b \in N$ we obtain a unique $c \in N$ by the operation $a + b = c$.

ASSOCIATIVE LAW:

A binary operator $*$ on a set S is said to be associative whenever $(x * y) * z = x * (y * z)$ for all $x, y, z \in S$, for all Boolean values x, y and z .

COMMUTATIVE LAW:

A binary operator $*$ on a set S is said to be commutative whenever $x * y = y * x$ for all $x, y, z \in S$.

IDENTITY ELEMENT:

A set S is said to have an identity element with respect to a binary operation $*$ on S if there exists an element $e \in S$ with the property $e * x = x * e = x$ for every $x \in S$

BASIC IDENTITIES OF BOOLEAN ALGEBRA

- *Postulate 1 (Definition):* A Boolean algebra is a closed algebraic system containing a set K of two or more elements and the two operators \cdot and $+$ which refer to logical AND and logical OR
- $x + 0 = x$
- $x \cdot 0 = 0$
- $x + 1 = 1$
- $x \cdot 1 = x$
- $x + x = x$
- $x \cdot x = x$
- $x + x' = 1$
- $x \cdot x' = 0$
- $x + y = y + x$
- $xy = yx$
- $x + (y + z) = (x + y) + z$
- $x(yz) = (xy)z$
- $x(y + z) = xy + xz$
- $x + yz = (x + y)(x + z)$
- $(x + y)' = x'y'$
- $(xy)' = x' + y'$
- $(x')' = x$

DeMorgan's Theorem

$$(a + b)' = a'b' \quad (b) \quad (ab)' = a' + b'$$

Generalized DeMorgan's Theorem (a) $(a + b +$

$$\dots z)' = a'b' \dots z' \quad (b) \quad (a.b \dots z)' = a' + b' + \dots z'$$

**L
O
G
I
C

G
A
T
E
S**

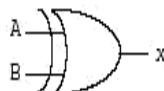
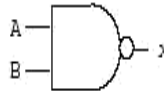
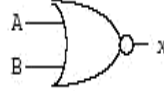
Formal logic: In formal logic, a statement (proposition) is a declarative sentence that is either true(1) or false (0). It is easier to communicate with computers using formal logic.

- **Boolean variable:** Takes only two values – either true (1) or false (0). They are used as basic units of formal logic.
- **Boolean algebra:** Deals with binary variables and logic operations operating on those variables.
- **Logic diagram:** Composed of graphic symbols for logic gates. A simple circuit sketch that represents inputs and outputs of Boolean functions.



2.1.1. UNIVERSAL GATES: NAND and NOR are universal gates.

- Other common gates include:

Name	Graphic symbol	Algebraic function	Truth table															
Exclusive-OR (XOR)		$x = A \oplus B$ $= A'B + AB'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	0	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
NAND		$x = (AB)'$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	1	0	1	1	1	0	1	1	1	0
A	B	x																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$x = A + B$	<table><tr><th>A</th><th>B</th><th>x</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	x	0	0	1	0	1	0	1	0	0	1	1	0
A	B	x																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

Parity check: True if only one is true.

Inversion of AND.

Inversion of OR.

2.1.2. MINIMIZATION OF BOOLEAN FUNCTIONS

Minimization of switching functions is to obtain logic circuits with least circuit complexity. This goal is very difficult since how a minimal function relates to the implementation technology is important. For

example, If we are building a logic circuit that uses discrete logic made of small scale Integration ICs(SSIs) like 7400 series, in which basic building block are constructed and are available for use. The goal of minimization would be to reduce the number of ICs and not the logic gates. For example, If we require two 6 and gates and 5 Or gates, we would require 2 AND ICs(each has 4 AND gates) and one OR IC. (4 gates). On the other hand if the same logic could be implemented with only 10 nand gates, we require only 3 ICs. Similarly when we design logic on Programmable device, we may implement the design with certain number of gates and remaining gates may not be used. Whatever may be the criteria of minimization we would be guided by the following:

■ Boolean algebra helps us simplify expressions and circuits

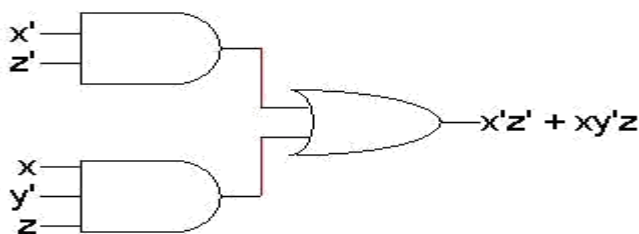
■ Karnaugh Map: A graphical technique for simplifying a Boolean expression into either form:

- o minimal sum of products (MSP)

- o minimal product of sums (MPS)

■ Goal of the simplification: There are a minimal number of product/sum terms and Each term has a minimal number of literals

■ Circuit-wise, this leads to a *minimal* two-level implementation



- A two-variable function has four possible minterms. We can re-arrange these minterms into a

Karnaugh map

	m i n t e r m
	x , y ,
	x , y
	x y ,
	x y

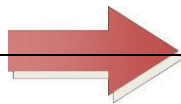
Y

0
1

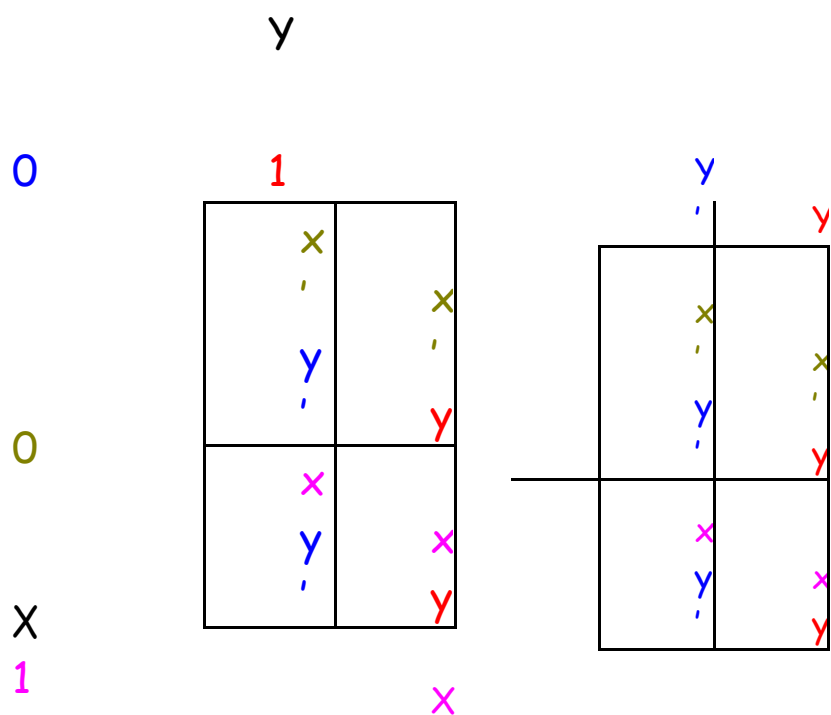
X

	x , y ,		x , y
	x y ,		x y

1



- Now we can easily see which minterms contain common literals
 - Minterms on the left and right sides contain y' and y respectively
 - Minterms in the top and bottom rows contain x' and x respectively



k-map Simplification

- Imagine a two-variable sum of minterms

$$x'y' + x'y$$

- Both of these minterms appear in the top row of a Karnaugh map, which means that they both contain the literal x'

			y
		x'	y'
		$x'y'$	$x'y$
x		x	y
		$x'y$	$x'y'$

- What happens if you simplify this expression using Boolean algebra?

$$\begin{aligned}
 &x'y' + x'y \\
 &= x'(y' + y) \\
 &= x'(1) \\
 &= x'
 \end{aligned}$$

[Distributive]

- With this ordering, any group of 2, 4 or 8 adjacent squares on the map contains common literals that can be factored out

		Y		
	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X	$xy'z'$	$xy'z$	xyz	xyz'
		Z		

$$\begin{aligned}
 &= x'y'z + x'yz \\
 &= x'z(y' + y) \\
 &= x'z \cdot 1 \\
 &= x'z
 \end{aligned}$$

- "Adjacency" includes wrapping around the left and right sides:

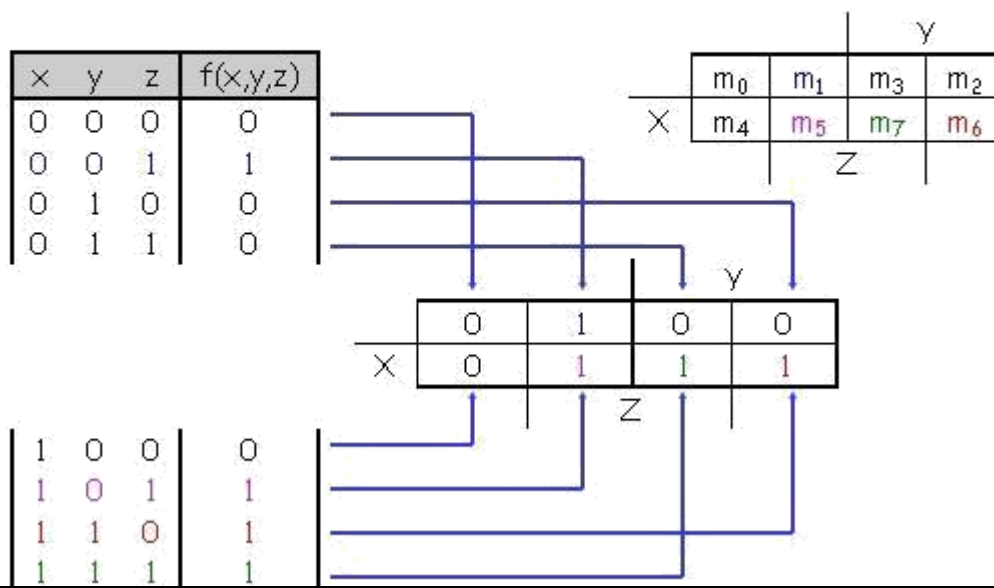
		Y		
	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X	$xy'z'$	$xy'z$	xyz	xyz'
		Z		

$$\begin{aligned}
 &= x'y'z' + xy'z' + x'yz' + xyz' \\
 &= z'(x'y' + xy' + x'y + xy) \\
 &= z'(y'(x' + x) + y(x' + x)) \\
 &= z'(y' + y) \\
 &= z'
 \end{aligned}$$

- We'll use this property of adjacent squares to do our simplifications.

K-maps From Truth Tables

- We can fill in the K-map directly from a truth table
 - The output in row i of the table goes into square m_i of the K-map
 - Remember that the rightmost columns of the K-map are "switched"



- You can find the minimal SoP expression
 - Each rectangle corresponds to one product term
 - The product is determined by finding the common literals in that rectangle

The image shows two Karnaugh maps for the function $F(x, y, z) = x'y'z + x'yz' + xy$. The top map is a 2x4 grid with columns labeled 0, 1, 0, 0 and rows labeled 0, 1. The bottom map is a 2x4 grid with columns labeled $x'y'z'$, $x'y'z$, $x'yz$, $x'yz'$ and rows labeled $xy'z'$, xyz' . Both maps show prime implicants highlighted in pink and blue.

$$F(x,y,z)=y'z+xy$$

Grouping the Minterms Together

- The most difficult step is grouping together all the 1s in the K-map
 - Make **rectangles** around groups of one, two, four or eight 1s
 - All of the 1s in the map should be included in at least one rectangle
 - Do *not* include any of the 0s
 - Each group corresponds to one product term

			y	
	0	1	0	0
x	0	1	1	1
		z		

K-map Simplification of SoP Expressions

- Let's consider simplifying $f(x,y,z) = xy + y'z + xz$
- You should convert the expression into a sum of minterms form,
 - The easiest way to do this is to make a truth table for the function, and then read off the minterms
 - You can either write out the literals or use the minterm shorthand
- Here is the truth table and sum of minterms for our example:

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$\begin{aligned}
 f(x,y,z) &= x'y'z + xy'z + xyz' + xyz \\
 &= m_1 + m_5 + m_6 + m_7
 \end{aligned}$$

Unsimplifying Expressions

- You can also convert the expression to a sum of minterms with Boolean algebra
 - Apply the distributive law in reverse to add in missing variables.
 - Very few people actually do this, but it's occasionally useful.

$$\begin{aligned}
 xy + y'z + xz &= (xy \cdot 1) + (y'z \cdot 1) + (xz \cdot 1) \\
 &= (xy \cdot (z + z')) + (y'z \cdot (x' + x)) + (xz \cdot (y' + y)) \\
 &= (xyz' + xyz) + (x'y'z + xy'z) + (xy'z + xyz) \\
 &= xyz' + xyz + x'y'z + xy'z \\
 &= m_1 + m_5 + m_6 + m_7
 \end{aligned}$$

- In both cases, we're actually "unsimplifying" our example expression
 - The resulting expression is larger than the original one!
 - But having all the individual minterms makes it easy to combine them

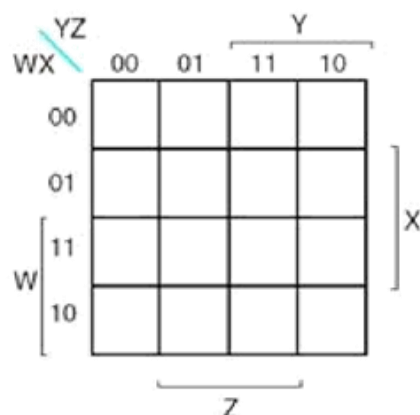
together with the K-map

Four-variable

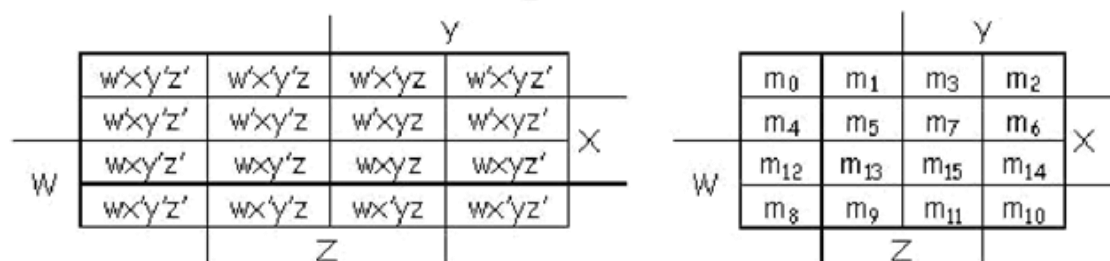
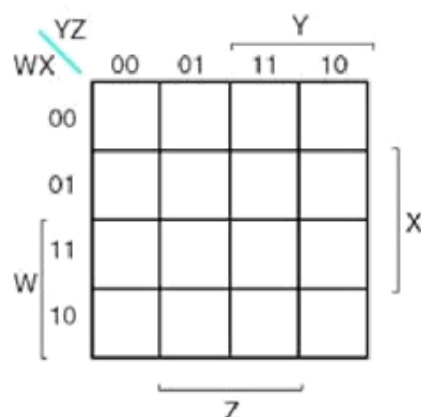
K-maps –

$f(W,X,Y,Z)$

- We can do four-variable expressions too!
 - The minterms in the third and fourth columns, *and* in the third and fourth rows, are switched around.
 - Again, this ensures that adjacent squares have common literals



- Grouping minterms is similar to the three-variable case, but:
 - You can have rectangular groups of 1, 2, 4, 8 or 16 minterms
 - You can wrap around *all four* sides



Simplify

$$m_0 + m_2 + m_5 + m_8 +$$

$$m_{10} + m_{13}$$

- The expression is already a sum of minterms, so here's the K-map:

		Y		
	1	0	0	1
	0	1	0	0
W	0	1	0	0
	1	0	0	1
	Z			X

		Y		
	m_0	m_1	m_3	m_2
	m_4	m_5	m_7	m_6
W	m_{12}	m_{13}	m_{15}	m_{14}
	m_8	m_9	m_{11}	m_{10}
	Z			X

- We can make the following groups, resulting in the MSP $x'z' + xy'z$

		Y		
	1	0	0	1
	0	1	0	0
W	0	1	0	0
	1	0	0	1
	Z			X

		Y		
	$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
W	$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$
	Z			X

Five-variable K-maps – $f(V, W, X, Y, Z)$

		Y		
	00	01	11	10
WX	00			
	01			
W	11			
	10			
	Z			X
	V = 0			

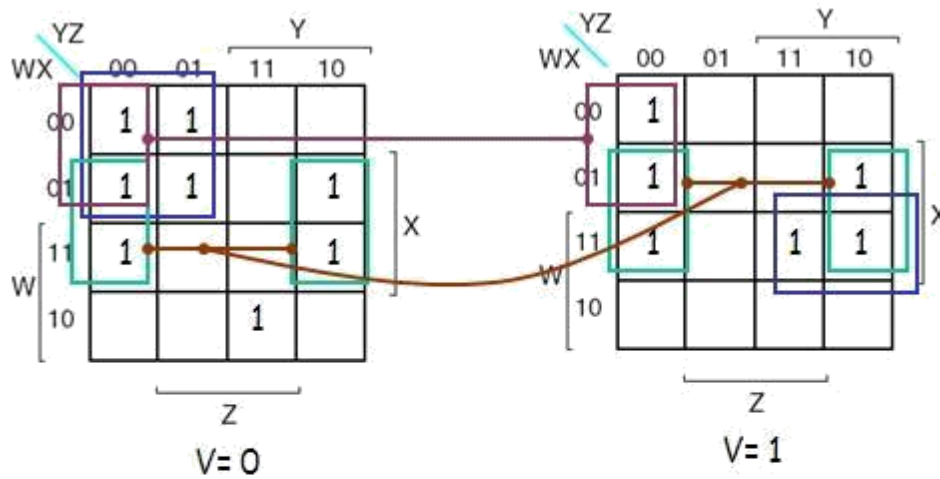
		Y		
	00	01	11	10
WX	00			
	01			
W	11			
	10			
	Z			X
	V = 1			

		Y		
	m_0	m_1	m_3	m_2
	m_4	m_5	m_7	m_6
W	m_{12}	m_{13}	m_{15}	m_{14}
	Z			X

		Y		
	m_{16}	m_{17}	m_{19}	m_{18}
	m_{20}	m_{21}	m_{23}	m_{22}
W	m_{28}	m_{29}	m_{31}	m_{30}
	Z			X

Simplify

$f(V,W,X,Y,Z) = \Sigma m(0,1,4,5,6,11,12,14,16,20,22,28,30,31)$



$$\begin{aligned}
 f &= XZ' & \Sigma m(4,6,12,14,20,22,28,30) \\
 &+ V'W'Y' & \Sigma m(0,1,4,5) \\
 &+ W'Y'Z' & \Sigma m(0,4,16,20) \\
 &+ VWXY & \Sigma m(30,31) \\
 &+ V'WX'YZ & m11
 \end{aligned}$$

PoS Optimization

- Maxterms are grouped to find minimal PoS expression

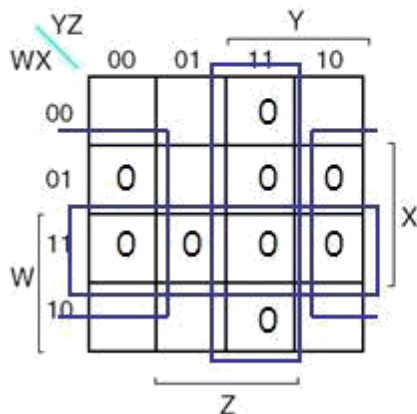
		yz			
		00	01	11	10
x	0	$x+y+z$	$x+y+z'$	$x+y'+z'$	$x+y'+z$
	1	$x'+y+z$	$x'+y+z'$	$x'+y'+z'$	$x'+y'+z$

Karnaugh map for $F(x, y, z) = x + yz$:

	00	01	11	10
0	$x + y + z$	$x + y + z'$	$x + y' + z'$	$x + y' + z$
1	$x' + y + z$	$x' + y + z'$	$x' + y' + z'$	$x' + y' + z$

Karnaugh map for $f(x, y, z) = x + yz$:

	00	01	11	10
0	0	0	1	0
1	0	0	1	1

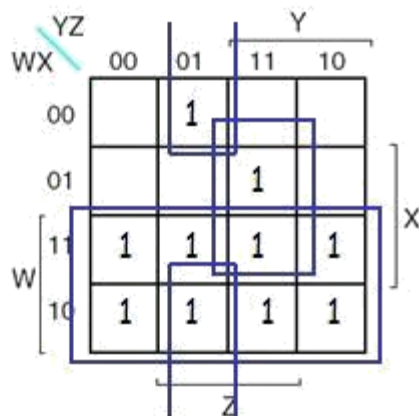
$$F(W,X,Y,Z) = \sum m(0,1,2,5,8,9,10) \\ = \prod M(3,4,6,7,11,12,13,14,15)$$


Which one is the minimal one?

SoP Optimization from PoS

$$F(W,X,Y,Z) = \prod M(0,2,3,4,5,6)$$

$$= \sum m(1,7,8,9,10,11,12,13,14,15)$$



$$F(W,X,Y,Z) = W + XYZ + X'Y'Z$$

Don't care

- You don't always need all 2^n input combinations in an n-variable function
 - If you can guarantee that certain input combinations never occur
 - If some outputs aren't used in the rest of the circuit
- We mark don't-care outputs in truth tables and K-maps with Xs.

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	X
1	1	1	1

- Find a MSP for

$$f(w,x,y,z) = \sum m(0,2,4,5,8,14,15), d(w,x,y,z) = \sum m(7,10,13)$$

This notation means that input combinations $wxyz = 0111, 1010$ and 1101 (corresponding to minterms m_7, m_{10} and m_{13}) are unused.

				y
	1	0	0	1
	1	1	x	0
	0	x	1	1
w	1	0	0	x
				z

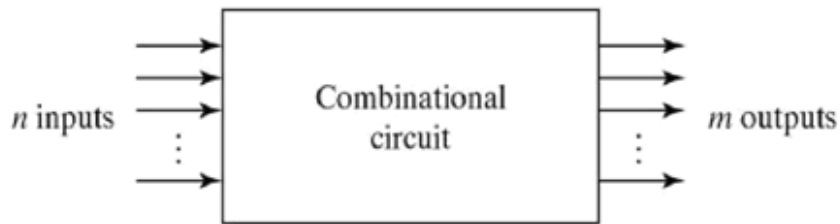
K-map Summary

- K-maps are an alternative to Boolean algebra for simplifying expressions
 - The result is a MSP/MPS, which leads to a minimal two-level circuit
 - It's easy to handle don't-care conditions
 - K-maps are really only good for manual simplification of small expressions...
 - Things to keep in mind:
 - Remember the correct order of minterms/maxterms on the K-map
 - When grouping, you can wrap around all sides of the K-map, and your groups can overlap
 - Make as few rectangles as possible, but make each of them as large as possible. This leads to fewer, but simpler, product terms
 - There may be more than one valid solution

2.1.3. Combinational Logic

- Logic circuits for digital systems may be combinational or sequential.

- A combinational circuit consists of input variables, logic gates, and output variables.



Analysis procedure

To obtain the output Boolean functions from a logic diagram, proceed as follows:

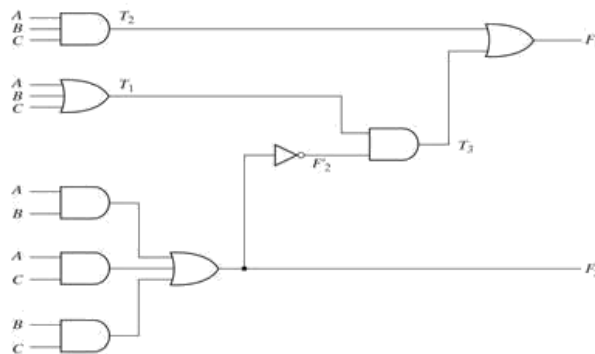
1. Label all gate outputs that are a function of input variables with arbitrary symbols. Determine the Boolean functions for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Example

$$F_2 = AB + AC + BC; \quad T_1 = A + B + C; \quad T_2 = ABC; \quad T_3 = F_2' T_1;$$

$$F_1 = T_3 + T_2$$

$$F_1 = T_3 + T_2 = F_2' T_1 + ABC = A'BC' + A'B'C + AB'C' + ABC$$



Derive truth table from logic diagram

We can derive the truth table by using the above circuit

A	B	C	F ₂	F ₂	T ₁	T ₂	T ₃	F ₁
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

2.1.4. FLIP FLOPS

There are a variety of flip-flops, all of which share two properties:

1. The flip-flop is a bistable device either 0 or 1. It exists in one of two states and, in the absence of input, remains in that state. Thus, the flip-flop can function as a 1-bit memory.
2. The flip-flop has two outputs, which are always the complements of each other. These are generally labeled

Q and Q.

Table 1 shows symbolic graphic and feature table for three types of flip-flop that are S-R, J-K and D flip-flops. Flip-flop is a form of memory element used to construct sequential circuits that are more complex, such as registers etc. Sequential circuits can be divided into

1. Synchronous
2. Asynchronous

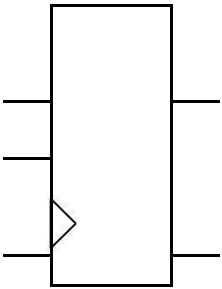
In synchronous sequential circuit, all flip-flops are moved by the same clock pulse so that all flip-flops involved change simultaneously.

In asynchronous circuit, the change of flip-flop condition depends on the change that

occurs on the input and the late time that is in the circuit.

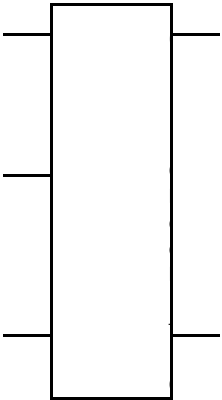
Name	Graphical Symbol	Feature Table
------	------------------	---------------

S
-
R



S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	-

J
-
K



J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	Change condition

D

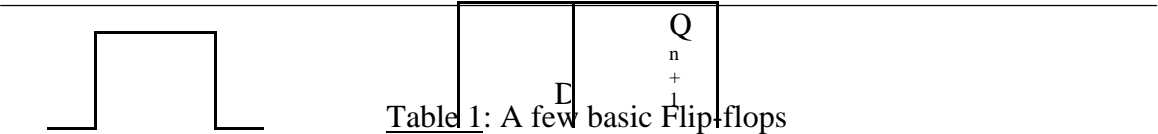
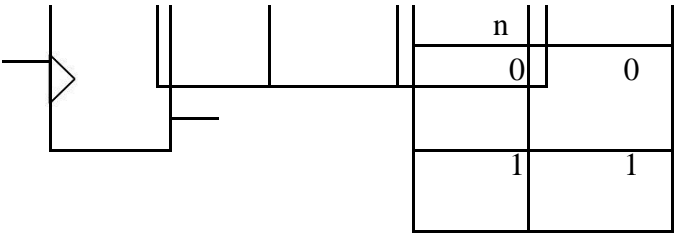


Table 1: A few basic Flip-flops

S
-
R

F
l
i
p
-
f
l
o
p

S-R flip-flop has 2 inputs, S (set) and R (reset) like Diagram 3 below. In the diagram below, (also for JK and D flip-flops), there use another input called clock. It is to control the movement of input that is input will only occur when given a clock pulse (synchronous circuit)

The features of S-R flip-flop can be depicted in Table 2 below. It can be summarized that:

1. If the value of both S and R are 0, the flip-flop will remain in its present condition (either 0 or 1).
2. If S = 0 and R = 1 (reset), then the flip-flop condition will change to 0 (its output, Q = 0).
3. If S = 1 (set) and R = 0, then the flip-flop condition will change to 1 (output, Q = 1).
4. This circuit does not allow combinational input of input S = 1 and R = 1.

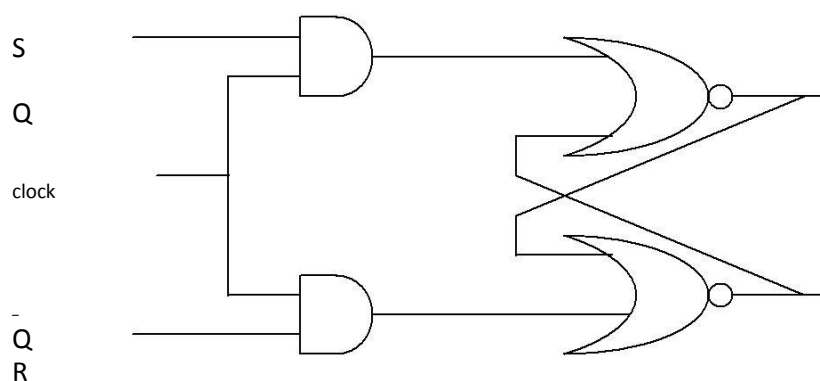


Diagram 3 : S-R Flip-flop

S	R	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-

1	1	1	-

Table 2 : Feature table of S-R Flip-flop

J-K Flip-flop

J-K flip-flop also has 2 inputs, J and K. The function of clock is same as S-R flip-flop. Unlike S-R flip-flop, J-K flip-flop allows all combination of inputs. The logic circuit for J-K flip-flop is shown in Diagram 2 below. Table 3 shows the features of J-K flip-flop. From the table, it can be summarized that:

1. If $J = 0$ and $K = 0$, it will maintain the flip-flop condition like before
2. If $J = 0$ and $K = 1$, it will cause flip-flop to change to condition 0 (reset).
3. If $J = 1$ and $K = 0$, it will cause flip-flop to change to condition 1 (set).
4. If $J = 1$ and $K = 1$, it will change the flip-flop condition, that is it will become complementary to the initial or prior condition

It can be observed that J-K flip-flop is built to address the input problem of $S = R = 1$ in S-R flip-flop. Features 1 till 3 are same as S-R flip-flop.

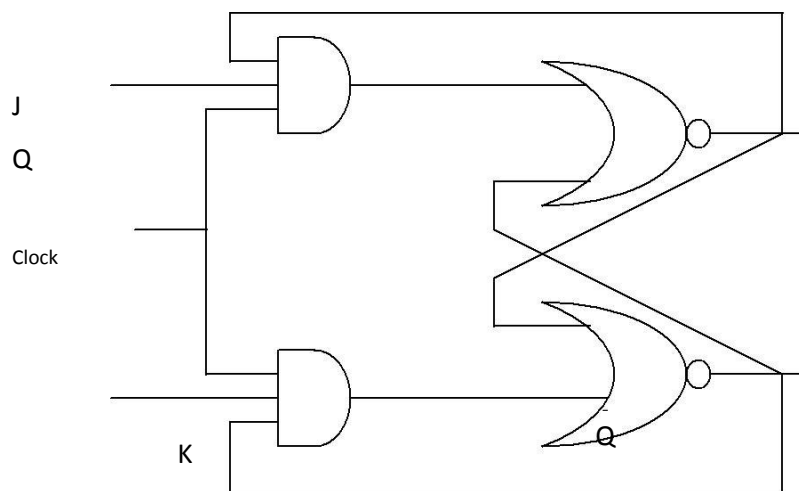


Diagram 2: J-K Flip-flop

J	K	Q_n	Q_{n+1}
0	0	0	0

0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 3: Features table of J-K flip-flop

D Flip-flop

Logic circuit for D flip-flop is shown in Diagram 5 below. This flip-flop only has one input that is D. The clock function is same as S-R and J-K flip-flops. The features of D flip-flop can be illustrated by Table 2. From the table, it can be seen that this flip-flop produces the same output as its input regardless of the condition of the stated flip-flop. This feature is very suitable to be used as memory element and this flip-flop is mostly used to make registers and computer memory (RAM)

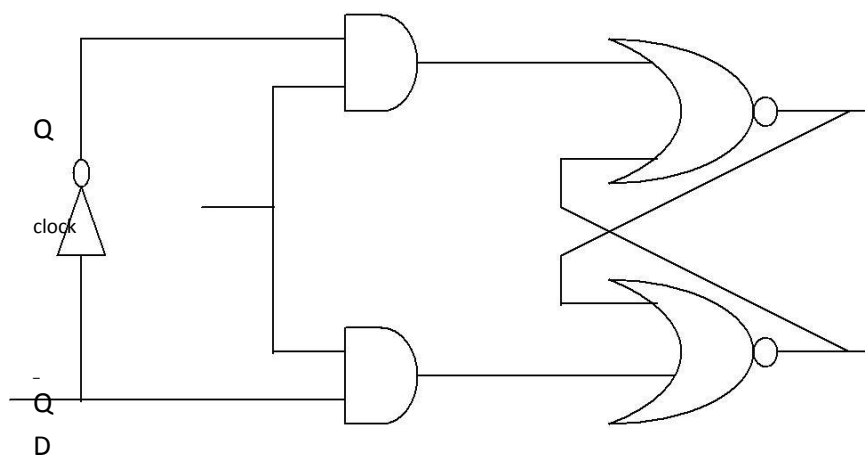


Diagram 5 : D Flip-flop

D	Q_n	Q_{n+1}
0	0	0

0	1	0
1	0	1
1	1	1

Table 2 : Feature table of D Flip-flop

PART-2

Combinational and Sequential Circuit

Registers,
Shift Registers,
Binary counters
Decoders,
Multiplexers,
Programmable Logic Devices.

Combinational Circuit and Sequential Circuit Criterions

Logic
Circuits can
be divided
into :

1. Combinational Logic Circuit
2. Sequential Logic Circuit

Combinational Logic Circuit

Combinational Logic circuit contains logic gates where its output is determined by the combination of the current input, regardless of the output or the prior combination of input.

Basically, combinational circuit can be depicted by Diagram 1 below:

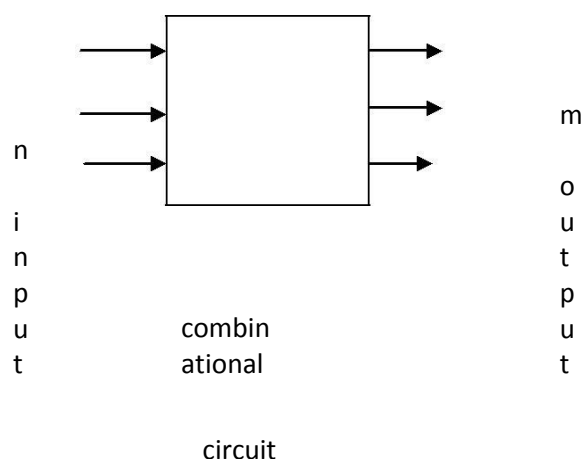


Diagram 1

Examples of Combinational circuits in the computer system are decoder, parallel adder, and multiplexer

(Note: Students are encouraged to obtain examples of combinational circuits stated above)

Sequential Logic Circuit

Sequential Logic Circuit contains logic gates arranged in parallel and its output is not only determined by the combination of the current input, but also the prior output. The circuit also contains memory elements that enable it to store the information of the prior output.

Examples of sequential circuits in the computer system are like registers, counters and serial adders

Some Examples of Combinational Circuit: Parallel Adder, Decoder, etc

The circuits learnt in chapter 3 are combinational circuits. The steps to design combinational circuits are as the following:

1. Understand the problem
2. Determine the number of input and output variables that are needed
3. Give symbols for the stated input and output
4. Construct a truth table that defines the relationship between the input and output
5. Obtain the Boolean function or the logical expression from the truth table in (2) using Karnaugh Map or other known methods.
6. Draw a logic circuit based on the expression obtained from (5) above.

Below are examples of designing combinational circuits that are in the computer system that is the adder. Because computers use binary system for its data, its adder is based on the addition of the binary system. There are 2 kinds of addition, which are identified to be half addition and full addition.

Half addition is the addition of 2 bits data (doesn't involve carry) that produces 2 bits output, that is the result and the carrier. Full addition is the addition of 3 bits data (2 bits data and 1 bit carry) that produces 2 bits output (sum and carry). Logic circuit for half addition is known as Half Adder while the logic circuit for full addition is known as Full Adder

Designing a Circuit for Half Adder

The steps are as below:

Problem: to build a logic circuit for the addition of 2 bits data

1. Number of input : 2 Number of output : 2
2. Variables for input: x and y Variables for output : s (sum) and c (carry)
3. The Truth Table for the problem :

I N P U T		1. OUTPUT	
	y	s	c
	0	0	0
	1	1	0
	0	1	0
	1	0	1

4. The expressions for r and c using Karnaugh Map

F

o

r

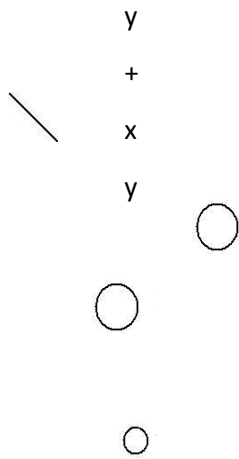
s

x

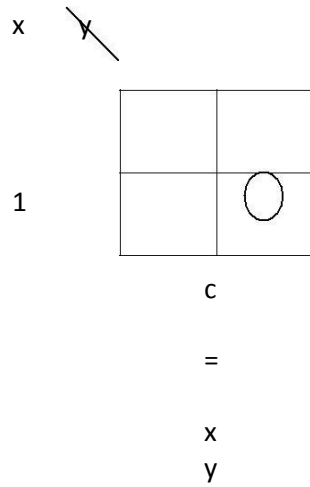
— s —

=

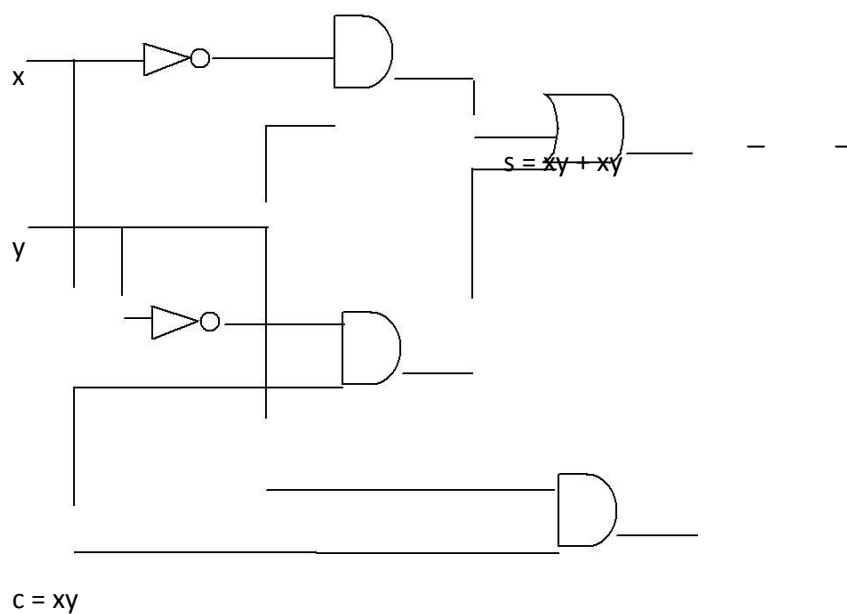
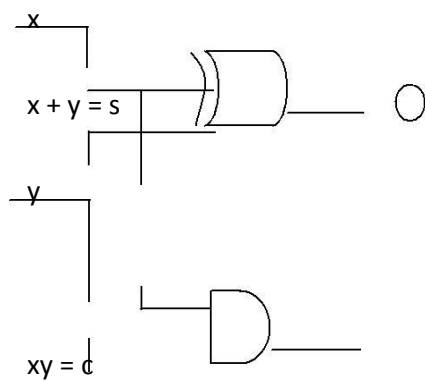
x



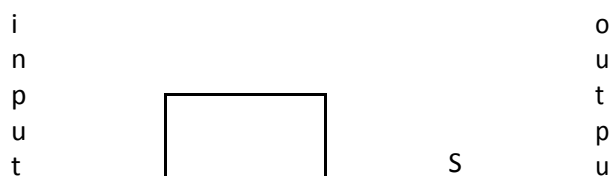
For c

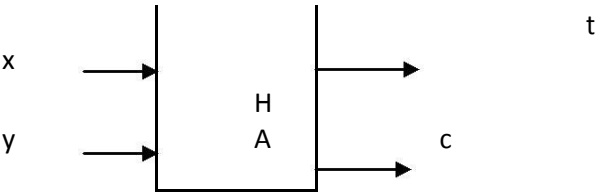


5. A logic circuits for Half Adder (HA)

**OR**

A Block
Diagram for
HA is as
below:





Designing a Circuit for Full Adder (FA)

The same method used to design HA.

1. Problem: Build logic circuit for the addition of 3 bit data
2. Number of input : 3 Number of output : 2
3. Variables for input: x , y and c_i
Variables for output : s (sum) and c_o (carry)
4. The truth table for the problem :

I N P U T			O U T P U T	
x	y	c_i	s	
0	0	0	0	
0	0	1	1	
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	
1	1	0	0	
1	1	1	1	

5. Obtain the expression for r and c_o using Karnaugh Map (**Students are required to try this out themselves**):

w
i
l
l

o
b

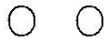
$s =$
 $x y$
 $p_i +$
 $x y$
 $c_i +$

■ ■ - - - -

t
a
i
n

x y

$$= x + y + c_i$$

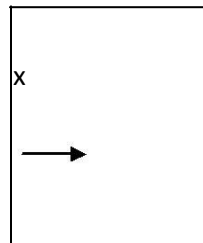
a
n
d

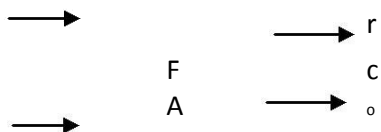
$$c_o = x y + y$$

$$c_i + x c_i$$

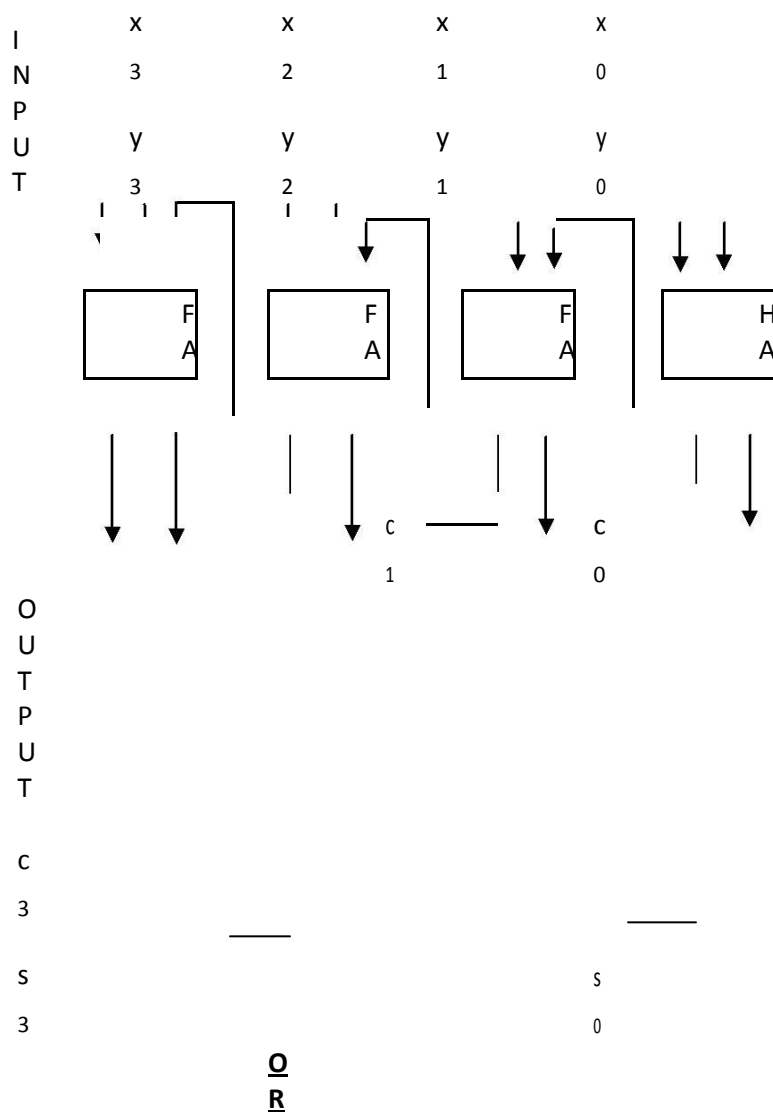
6. Draw the circuit for FA (**Students are required to try this out themselves**):

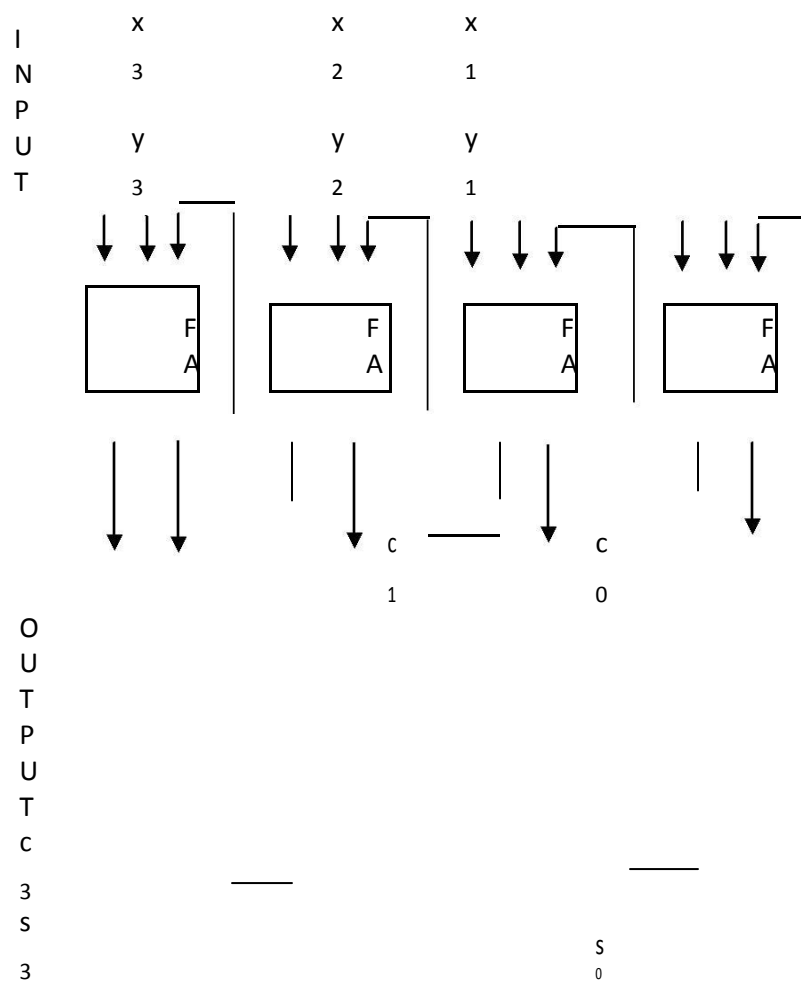
Generally, the block diagram for FA is shown as below :





To construct a 2-bit parallel adder, 3 FA and 1 HA are required like the diagram below with the input as $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$ (X and Y are binary numbers 2-bit) and the output (addition result) is $r_3r_2r_1r_0$.





Some Examples of Sequential Circuits: Flip-flop, Register, Serial Adder, etc.

Sequential circuits are a kind of logic circuit where the current output not only depends on the current input but also on the past history of inputs. Another and generally more useful way to view it is that the current output of a sequential circuit depends on the current input and the current state of that circuit. The simplest form of sequential circuit is the flip-flop. Flip-flop is a kind of logic circuit that is capable of exhibiting 2 stable conditions. It is also known as 1-bit memory element and is mostly used to make important computer components such as registers, counters, memory etc.

A few examples of Flip-flop (Sequential Circuit) usage

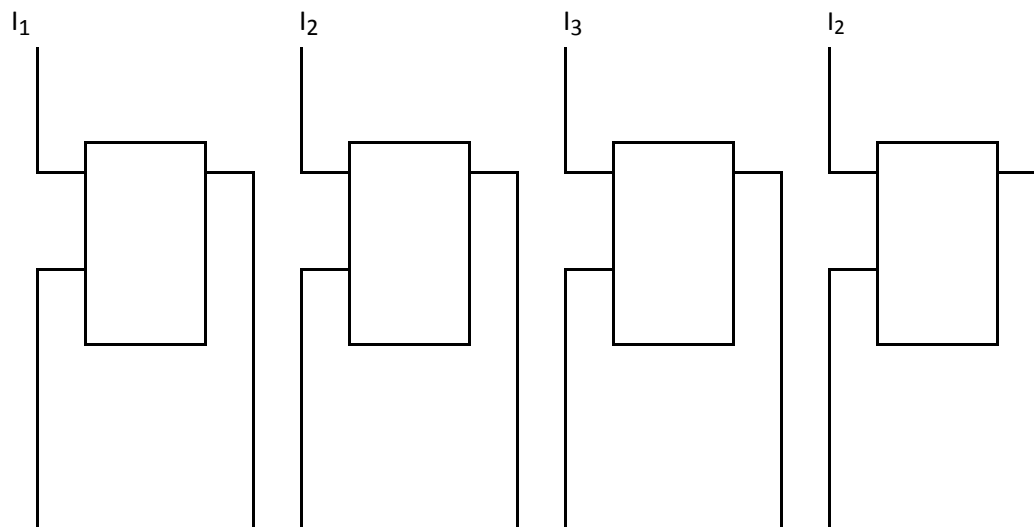
As priorly stated, flip-flop is an example of the simplest form of sequential circuit. It is also a form of memory element where a flip-flop can store 1 bit of data. In this section, examples of sequential circuits that use flip-flop will be given:

Register

Register is an important component in the computer. Generally, it can be categorized into:

1. Storage Register (or Parallel Register)
2. Shift Register (or Serial Register)

Parallel register is made up of a set of 1-bit (flip-flop) that can be written on and read simultaneously. This register is used to store data (output=input). The amount of flip-flop used depends on the size of the register that is to be built. If a parallel register that can store 8 bits of data is to be built, then 8 flip-flops are needed. Diagram 6 below is a 2 bit parallel register that uses flip-flop D. (Note: all kinds of flip-flop can be used to build storage register, but its circuit will differ because ever flip-flop has its own features)



| | | |

Diagram 6: A 2-bit parallel register
that uses D Flip-flop

Q

2

In the above diagram, 2 bits of input is admitted simultaneously, that is I_1 , I_2 , I_3 and I_2 , whereas its output is also is simultaneous or parallel, that is Q_1 , Q_2 , Q_3 and Q_2 .

In shift register, only one output is produced at a time. There are 2 types of shift register that is shift to right and shift to left. Shift to right register means the rightmost bit of the stated will be taken out first followed by the

following bits after a given clock beat. It's vice versa for move to shift to left register.

Diagram 7 below is an example of 2-bit shift to right register that utilizes J-K flip-flop.

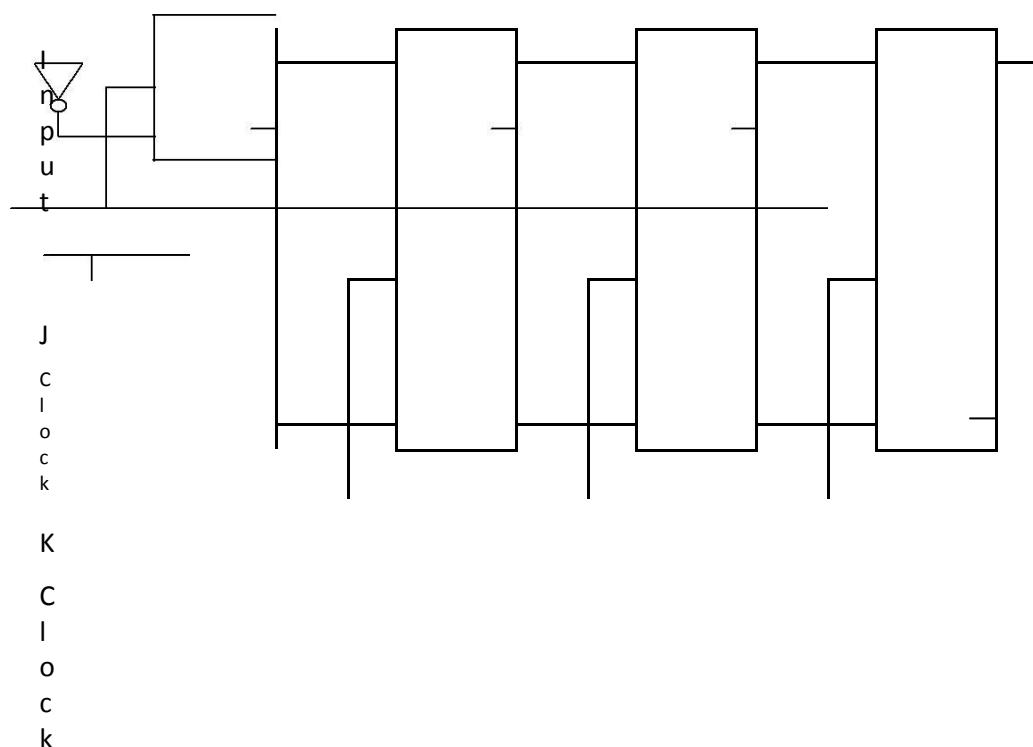


Diagram 7: Shift to Right Register Using J-K Flip-flop

Parallel Adder

In the computer environment, there are 2 types of adders:

1. Parallel Adder
2. Serial Adder

Parallel adder is an adder that performs addition concurrently for each bit involved. Adder in [section 2.2](#) is called a serial adder. Serial Adder performs addition bit by bit starting with the rightmost bit, followed by the following bits. [Diagram 8](#) below is an example of a serial 2-bit adder. This adder uses two Shift to Right Registers, X and Y to hold operand 1 ($A = A_3A_2A_1A_0$) and operand 2 ($B = B_3B_2B_1B_0$), a full adder (see [section 2.2](#)) and a flip-flop (usually D flip-flop) to hold the carrier value.

The addition process in the adder are as below :

$$X = X + Y$$

that is the X and Y registers will hold operand 1 and operand 2 and the addition result will be kept in the X register. Hence, in the addition, the value in the Y (Operand 2) register cannot change while the X register holds the addition result (the value of operand 1 will be lost)

Note: observe and understand the data movement in the stated circuit after every clock pulse is given.

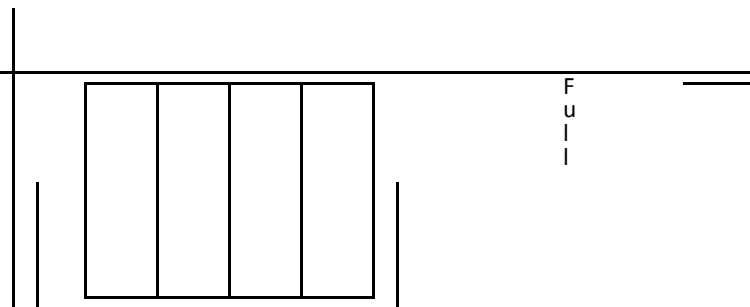
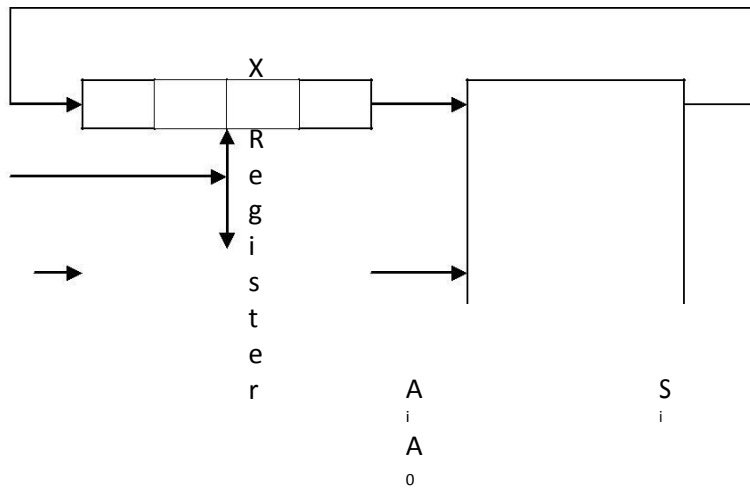


Diagram 8 : 2-bit Serial Adder

UNIT-3 PART-1

3.1.1. Algorithms for fixed point and floating point addition

3.1.1.1. Algorithms for fixed point addition

3.1.1.2. Algorithms for floating point addition

- 3.1.2. Subtraction, multiplication and division operations.
- 3.1.3. Hardware Implementation of arithmetic and logic operations,
- 3.1.4. High Performance arithmetic

UNIT-3 PART-2

Instruction set & Addressing

3.2.1. Memory Locations and Addresses

Mainmemory is the second major subsystem in a computer. It consists of a collection of storage locations, each with a unique identifier, called an address.

Data is transferred to and from memory in groups of bits called words. A word can be a group of 8bits, 16bits, 32bits or 64bits (and growing).

- If the word is 8 bits, it is referred to as a byte. The term “byte” is so common in computer science that sometimes a 16-bit word is referred to as a 2-byte word, or a 32-bit word is referred to as a 4 byteword.

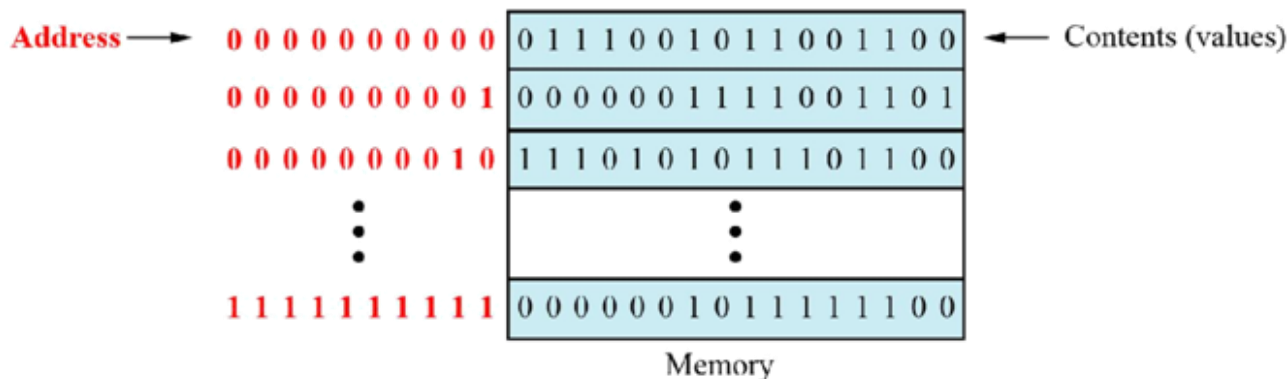


Figure 5.3 Main memory

Address space

- To access a word in memory requires an identifier. Although programmers use a name to identify a word (or a collection of words), at the hardware level each word is identified by an address.
- The total number of uniquely identifiable locations in memory is called the **addressspace**.
- For example, a memory with 64kilobytes (16address line required) and a wordsize of 1byte has an address space that ranges from 0 to 65,535

Table 5.1 Memory units

<i>Unit</i>	<i>Exact Number of Bytes</i>	<i>Approximation</i>
kilobyte	2^{10} (1024) bytes	10^3 bytes
megabyte	2^{20} (1,048,576) bytes	10^6 bytes
gigabyte	2^{30} (1,073,741,824) bytes	10^9 bytes
terabyte	2^{40} bytes	10^{12} bytes

Assignment of byte addresses

Little Endian (e.g., in DEC, Intel)

a) low order byte stored at lowest address

b) byte0 byte1 byte2 byte3

Big Endian

Big Endian (e.g., in IBM, Motorola, Sun, HP)

- a) high order byte stored at lowest address
- b) byte3 byte2 byte1 byte0

Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines

In case of 16 bit data, aligned words begin at byte addresses of 0,2,4,.....

- a) In case of 32 bit data, aligned words begin at byte address of 0,4,8,.....
- b) In case of 64 bit data, aligned words begin at byte addresses of 0,8,16,.....
- c) In some cases words can start at an arbitrary byte address also then, we say that word locations are unaligned

MEMORY OPERATIONS

Today, **general-purpose computers** use a set of instructions called a **program** to process data.

- a) A computer executes the program to create output data from input data
- b) Both program instructions and data operands are stored in memory

Two basic operations requires in memory access
 Load operation (Read or Fetch)-Contents of specified memory location are read by processor
 Store operation (Write)-Data from the processor is stored in specified memory location

INSTRUCTION SET ARCHITECTURE:-Complete instruction set of the processor

BASIC 4 TYPES OF OPERATION:-

- i) Data transfer between memory and processor register
- ii) Arithmetic and logic operation
- iii) Program sequencing and control
- iv) I/O transfer

Register transfer notation (RTN)

Transfer between processor registers & memory, between processor register & I/O devices
 Memory locations, registers and I/O register names are identified by a symbolic name in uppercase alphabets

LOC,PLACE,MEM are the address of memory location R1 , R2,... are processor registers
DATA_IN, DATA_OUT are I/O registers

Contents of location is indicated by using square brackets [] RHS of RTN always denotes a values, and is called Source

LHS of RTN always denotes a symbolic name where value is to be stored and is called destination not modified

ASSEMBLY LANGUAGE NOTATION (ALN)

RTN is easy to understand and but cannot be used to represent machine instructions

Mnemonics can be converted to machine language, which processor understands using assembler

Eg:

1. MOVE LOCN, R2
2. ADD R3, R2, R4

3.2.2. Machine addresses and sequencing

Each machine instruction is executed through the application of a sequence of microinstructions. Clearly, we must be able to sequence these; the collection of microinstructions which implements a particular machine instruction is called a routine.

The MCU typically determines the address of the first microinstruction which implements a machine instruction based on that instruction's opcode. Upon machine power-up, the CAR should contain the address of the first microinstruction to be executed.

The MCU must be able to execute microinstructions sequentially (e.g., within routines), but must also be able to "branch" to other microinstructions as required; hence, the need for a sequencer.

The microinstructions executed in sequence can be found sequentially in the CM, or can be found by branching to another location within the CM. Sequential retrieval of microinstructions can be done by simply incrementing the current CAR contents; branching requires determining the desired CW address, and loading that into the CAR.

Addressing Sequencing

CAR

Control Address Register

control ROM

control memory (CM); holds CWs

opcode

opcode field from machine instruction

mapping logic

hardware which maps opcode into microinstruction address

branch logic

determines how the next CAR value will be determined from all the various possibilities

multiplexors

implements choice of branch logic for next CAR value

incrementer

generates $CAR + 1$ as a possible next CAR value

SBR

used to hold return address for subroutine-call branch operations

Conditional branches are necessary in the microprogram. We must be able to perform some sequences of micro-ops only when certain situations or conditions exist (e.g., for conditional branching at the machine instruction level); to implement these, we need to be able to conditional execute or avoid certain microinstructions within routines.

Subroutine branches are helpful to have at the microprogram level. Many routines contain identical sequences of microinstructions; putting them into subroutines allows those routines to be shorter, thus saving memory.

Mapping of opcodes to microinstruction addresses can be done very simply. When the CM is designed, a "required" length is determined for the machine instruction routines (i.e., the length of the longest one). This is rounded up to the next power of 2, yielding a value k such that 2^k microinstructions will be sufficient to implement any routine.

The first instruction of each routine will be located in the CM at multiples of this "required" length. Say this is N . The first routine is at 0; the next, at N ; the next, at $2*N$; etc. This can be accomplished very easily. For instance, with a four-bit opcode and routine length of four microinstructions, k is two; generate the microinstruction address by appending two zero bits to the opcode:

addressing

Alternately, the n -bit opcode value can be used as the "address" input of a $2^n \times M$ ROM; the contents of the selected "word" in the ROM will be the desired M -bit CAR address for the beginning of the routine implementing that instruction. (This technique allows for variable-length routines in the CM.) >pp We choose between all the possible ways of generating CAR values by feeding them all into a multiplexor bank, and implementing special branch logic which will determine how the muxes will pass on the next address

to the CAR. As there are four possible ways of determining the next address, the multiplexor bank is made up of N 4x1 muxes, where N is the number of bits in the address of a CW. The branch logic is used to determine which of the four possible "next address" values is to be passed on to the CAR; its two output lines are the select inputs for the muxes

3.2.3. Addressing Modes

The term **addressing modes** refers to the way in which the operand of an instruction is specified. Information contained in the instruction code is the value of the operand or the address of the result/operand. Following are the main addressing modes that are used on various platforms and architectures.

1) Immediate Mode

The operand is an immediate value is stored explicitly in the instruction:

Example: SPIM (opcode dest, source)

li \$11, 3 // loads the immediate value of 3 into register \$11

li \$9, 8 // loads the immediate value of 8 into register \$9

Example : (textbook uses instructions type like, opcode source, dest)

move #200, R0; // move immediate value 200 in register R0

2) Index Mode

The address of the operand is obtained by adding to the contents of the general register (called index register) a constant value. The number of the index register and the constant value are included in the instruction code. Index Mode is used to access an array whose elements are in successive memory locations. The content of the instruction code, represents the starting address of the array and the value of the index register, and the index value of the current element. By incrementing or decrementing index register different element of the array can be accessed.

Example: SPIM/SAL - Accessing Arrays

```
.data
array1: .byte 1,2,3,4,5,6
.text
__start:
move $3, $0      # $3 initialize index register with 0
add $3, $3,4     # compute the index value of the fifth element
sb $0, array1($3) # array1[4]=0
# store byte 0 in the fifth element of the array
# index addressing mode
done
```

3) Indirect Mode

The effective address of the operand is the contents of a register or main memory location, location whose address appears in the instruction. Indirection is noted by placing the name of the register or the memory address given in the instruction in parentheses. The register or memory location that contains the address of the operand is a pointer. When an execution takes place in such mode, instruction may be told to go to a

specific address. Once it's there, instead of finding an operand, it finds an address where the operand is located.

NOTE:

Two memory accesses are required in order to obtain the value of the operand (fetch operand address and fetch operand value).

Example: (textbook) ADD (A), R0

(address A is embedded in the instruction code and (A) is the operand address = pointer variable)

Example: SPIM - simulating pointers and indirect register addressing

The following "C" code:

```
int *alpha=0x00002004, q=5; *alpha = q;
```

could be translated into the following assembly code:

```
alpha: .word 0x00002004 # alpha is an address variable # address value is
0x00002004 q: .word 5
```

```
....
```

```
lw $10,q # load word value from address q into $10
# $10 is 5
```

```
lw $11,alpha # $11 gets the value 0x0002004
```

```
# this is similar with a load immediate address value
```

```
sw $10,($11) # store value from register $10 at memory location
```

```
# whose address is given by the contents of register $11
```

```
# (store 5 at address 0x00002004)
```

Example: SPIM/SAL - - array pointers and indirect register addressing

```
.data
```

```
array1: .byte 1,2,3,4,5,6
```

```
.text
```

```
__start:
```

```
la $3, array1 # array1 is direct addressing mode
```

```
add $3, $3,4 # compute the address of the fifth element
```

```
sb $0, ($3) # array1[4]=0 , byte accessing
```

```
# indirect addressing mode
```

```
done
```


4) Absolute (Direct) Mode

The address of the operand is embedded in the instruction code.

Example: (SPIM)

```
beta: .word 2000
```

```
lw $11, beta    # load word (32-bit quantity) at address beta into register $11
# address of the word is embedded in the instruction code
# (register $11 will receive value 2000)
```

5) Register Mode

The name (the number) of the CPU register is embedded in the instruction. The register contains the value of the operand. The number of bits used to specify the register depends on the total number of registers from the processor set.

Example (SPIM)

```
add $14,$14,$13 # add contents of register $13 plus contents of
# register $14 and save the result in register $14
```

No memory access is required for the operand specified in register mode.

6) Displacement Mode

Similar to index mode, except instead of a index register a base register will be used. Base register contains a pointer to a memory location. An integer (constant) is also referred to as a displacement. The address of the operand is obtained by adding the contents of the base register plus the constant. The difference between index mode and displacement mode is in the number of bits used to represent the constant. When the constant is represented a number of bits to access the memory, then we have index mode. Index mode is more appropriate for array accessing; displacement mode is more appropriate for structure (records) accessing.

Example: SPIM/SAL - Accessing fields in structures

```
.data
student: .word 10000 #field code
.ascii "Smith" #field name
.byte # field test
.byte 80,80,90,100 # fields hw1,hw2,hw3,hw4
.text __start:
la $3, student # load address of the structure in $3
# $3 base register
add $17,$0,90 # value 90 in register $17
# displacement of field "test" is 9 bytes
#
sb $17, 9($3) # store contents of register $17 in field "test"
```

displacement addressing mode
done

7) Autoincrement /Autodecrement Mode

A special case of indirect register mode. The register whose number is included in the instruction code, contains the address of the operand. Autoincrement Mode = after operand addressing, the contents of the register is incremented. Decrement Mode = before operand addressing, the contents of the register is decrement.

Example: SPIM/SAL - - simulating autoincrement/autodecrement addressing mode

(MIPS has no autoincrement/autodecrement mode)

```
lw $3, array1($17) #load in reg. $3 word at address array1($17)
addi $17,$17,4      #increment address (32-bit words) after accessing
#operand this can be re-written in a "autoincrement like mode":
lw+ $3,array1($17)  # lw+ is not a real MIPS instruction
subi $17,$17,4      # decrement address before accessing the operand
lw $3,array1($17)
```

NOTE: the above sequence can be re-written proposing an "autodecrement instruction", **not real** in MIPS architecture.

```
-lw $3, array1($17)
```

3.2.4. Instruction Formats

The most common fields found in instruction format are:-

- (1) An operation code field that specified the operation to be performed
- (2) An address field that designates a memory address or a processor registers.
- (3) A mode field that specifies the way the operand or the effective address is determined.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address field in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organization.

- (1) Single Accumulator organization $ADD\ X\ AC \leftarrow AC + M[X]$
- (2) General Register Organization $ADD\ R1,\ R2,\ R3\ R \leftarrow R2 + R3$
- (3) Stack Organization $PUSH\ X$

Three address Instruction

Computer with three addresses instruction format can use each address field to specify either processor register or memory operand.

ADD R1, A, B $A1 \oplus M[A] + M[B]$

ADD R2, C, D $R2 \oplus M[C] + M[B]$ $X = (A + B) * (C + A)$

MUL X, R1, R2 $M[X] R1 * R2$

The advantage of the three address formats is that it results in short program when evaluating arithmetic expression. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instruction

Most common in commercial computers. Each address field can specify either a processes register or a memory word.

	R		
	1		
	\oplus		
R	M		
1			
,	[
	A		
A]		
	R		
	1		
	\oplus		
	R		
	1		
	+		
R	M		
1			
,	[
	B		
B]		
R	R	X	
2	2		
,		=	
	\oplus		
C		(

M
[
C
]

A
+
B
)
*
(
C
+
D
)

R
2
®

R
2
+

R
2
,
D
M
[
D
]
R
1

®

R
1
,
R
2
X
R
1
*

R
2
M

$$\begin{array}{c} 1 \\ R \\ 1 \end{array} \quad \begin{array}{c} [\\ X \\] \end{array}$$

$$\otimes$$

$$\begin{array}{c} R \\ 1 \end{array}$$

One Address instruction

It used an implied accumulator (AC) register for all data manipulation. For multiplication/division, there is a need for a second register.

$$\begin{array}{c} A \\ C \\ \otimes \\ M \\ [\\ A \\] \\ A \\ C \\ \otimes \\ A \\ C \\ + \\ M \\ [\\ B \\] \\ S \\ T \end{array} \quad \begin{array}{c} M \\ X \end{array}$$

O	[=
R	T	
E]	(
		A
T	®	+
	A	B
	C)
		×
		(
		C
		+
		A
)

All operations are done between the AC register and a memory operand. It's the address of a temporary memory location required for storing the intermediate result.

A
C
®
M
(
C
)
A
C
®
A
C
+
M

(
D
)
A
C

®

A
C

+

M

(
T
)
M

S
T
O
R
E

X
[
×
]
®

A
C

Zero – Address Instruction

A stack organized computer does not use an address field for the instruction ADD and MUL. The PUSH & POP instruction, however, need an address field to specify the operand that communicates with the stack (TOS[®] top of the stack)

	T
	O
	S
	®
	A
	T
	O
	S
	®
	B
	T
	O
	S
	®
/	(A
	+
	B)
	T
	O
	S
	®
	C
	T
	O
	S
	®
	D
	T
	O
	S
	®
/	(C
	+
	D)
	T
	O
	S
	®
	(C
	+

	D)
	*
	(A
	+
	B)
	M
	[X
]
I	T
(O
I	S

CISC Characteristics

A computer with large number of instructions is called complex instruction set computer or CISC. Complex instruction set computer is mostly used in scientific computing applications requiring lots of floating point arithmetic.

1. A large number of instructions - typically from 100 to 250 instructions.
2. Some instructions that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes - typically 5 to 20 different modes.
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory.

RISC Characteristics

A computer with few instructions and simple construction is called reduced instruction set computer or RISC. RISC architecture is simple and efficient. The major characteristics of RISC architecture are,

1. *Relatively few instructions*
2. *Relatively few addressing modes*
3. *Memory access limited to load and store instructions*
4. *All operations are done within the registers of the CPU*
5. *Fixed-length and easily-decoded instruction format.*
6. *Single cycle instruction execution*
7. *Hardwired and micro programmed control*

3.2.5. Basic Machine Instructions IA-32 Pentium example.

Pentium Addressing Modes

Virtual or effective address is offset into segment

— Starting address plus offset gives linear address

— This goes through page translation if paging is enabled

9 addressing modes available

— Immediate

— Register operand

— Displacement : offset in a segment

— Base : same as register indirect addressing

— Base with displacement

— Scaled index with displacement : scale factor is used

— scale factor of 2 can be used to index an

array of 16-bit integers

- Base with index and displacement
- Base scaled index with displacement
- Relative
- used in transfer-of-control instructions

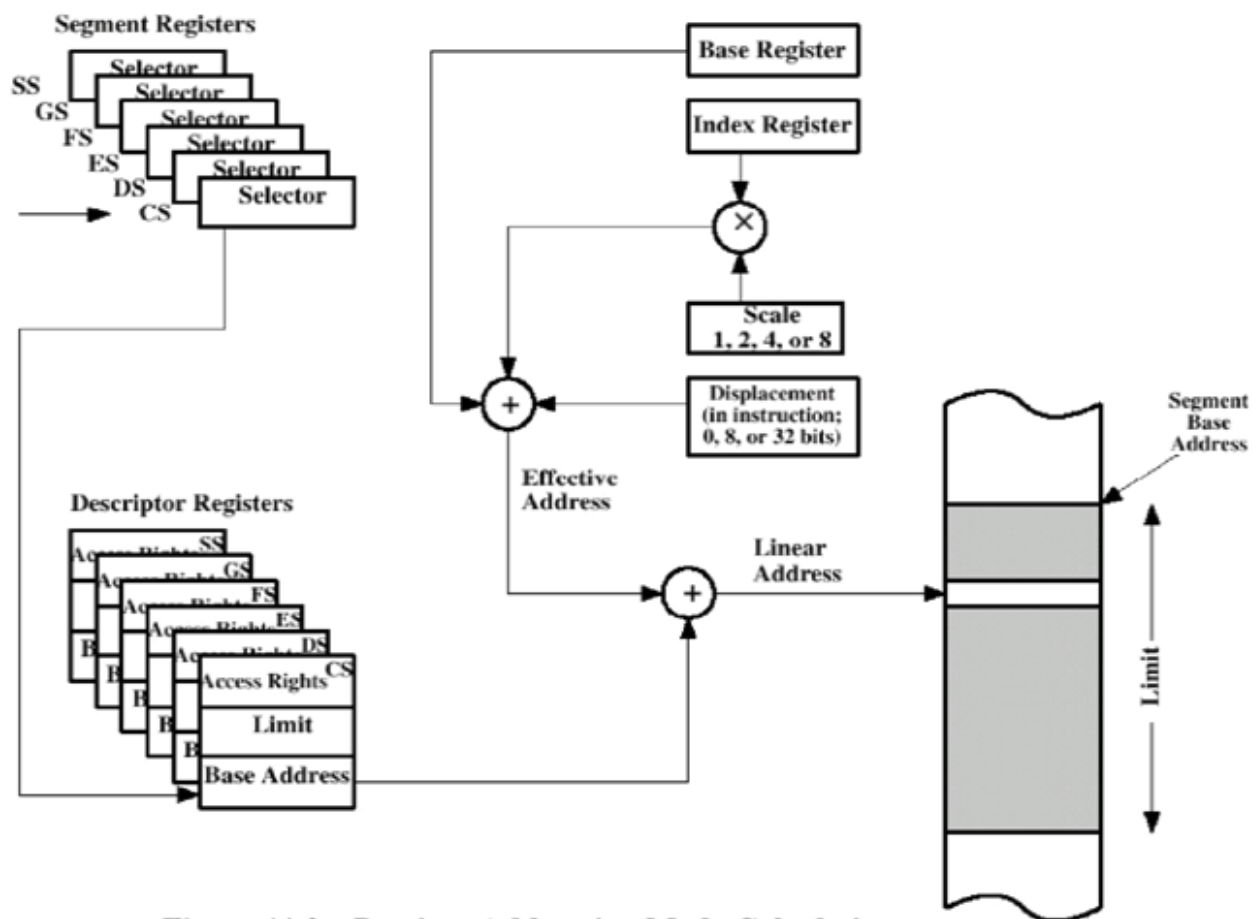


Figure 11.2 Pentium Addressing Mode Calculation

Mode	Algorithm
Immediate	$\text{Operand} = A$
Register Operand	$LA = R$
Displacement	$LA = (SR) + A$
Base	$LA = (SR) + (B)$
Base with Displacement	$LA = (SR) + (B) + A$
Scaled Index with Displacement	$LA = (SR) + (I) \times S + A$
Base with Index and Displacement	$LA = (SR) + (B) + (I) + A$
Base with Scaled Index and Displacement	$LA = (SR) + (I) \times S + (B) + A$
Relative	$LA = (PC) + A$

LA	=	linear address
(X)	=	contents of X
SR	=	segment register
PC	=	program counter
A	=	contents of an address field in the instruction
R	=	register
B	=	base register
I	=	index register
S	=	scaling factor

Pentium Instruction Formats

- Instruction consists of

0-4 optional prefixes

1-2 byte opcode

Optional address specifier

Consists of ModR/m byte

and Scale Index byte

Optional displacement

Optional immediate field

Pentium Instruction

Formats

Prefix bytes Instruction prefixes

LOCK or one of repeat prefixes

LOCK is used to ensure exclusive use of shared memory in multiprocessor environments

REP, REPE, REPZ, REPNE, and REPNZ Specify repeated operation on strings

Repeat until counter in CX goes zero or until the condition is met Segment override

Address size

Switches between 32-bit and 16-bit address generation Operand size

Switches between 32-bit and 16-bit operands

Pentium Instruction Formats

Instruction

1. Opcode

2. ModR/m

Specify whether an operand is in a register or in memory

3. SIB

Specify fully the addressing mode

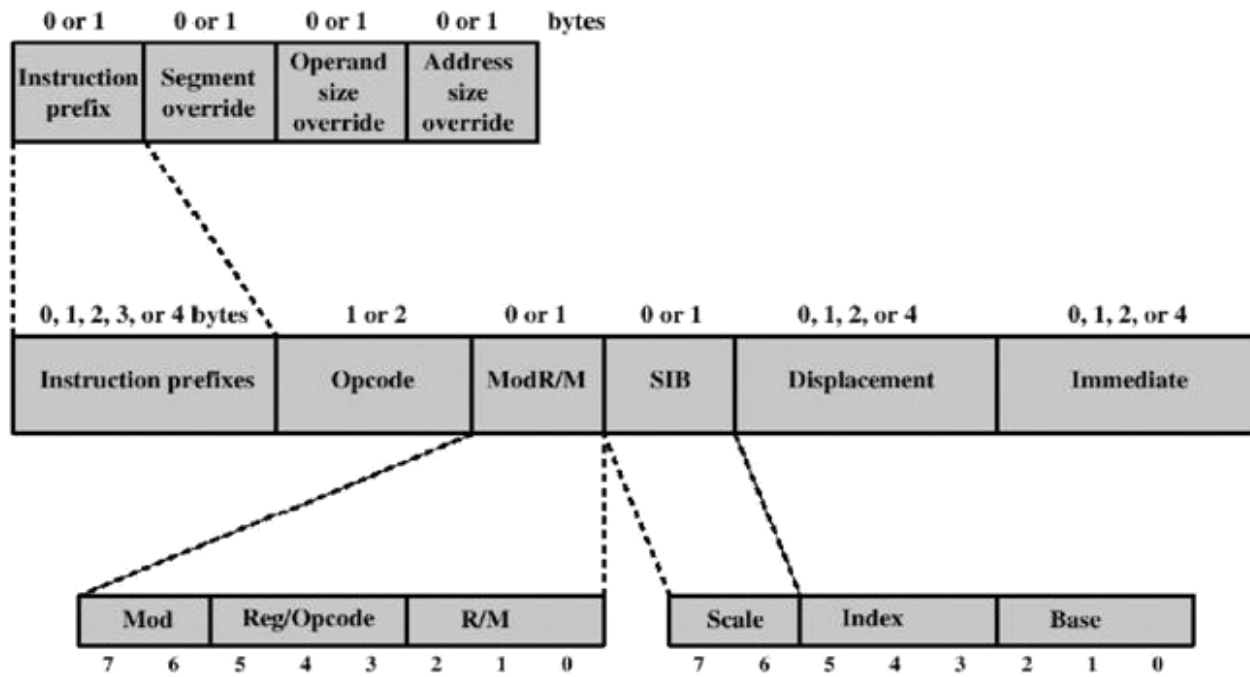
4. Displacement

– When used, 8-, 16-, or 32-bit displacement field is added

5. Immediate

When used, 8-, 16-, or 32-bit operand is provided

**Pentiu
m
Instruc
tion
Forma
ts**



UNIT-4

Memory organization

4.1 Concept Of Memory,

4.2 RAM, ROM Memories

4.3 Memory Hierarchy 4.4 Cache

4.5 Memories 4.6 Virtual Memory,

4.7 Secondary Storage,

Memory Management Requirements.

4.1 MEMORY HIERARCHY

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in a typical computer. Moreover, most computer users accumulate and continue to accumulate large amounts of data-processing software. Not all accumulated information is needed by the processor at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU. The memory unit that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.

The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic. **Figure** illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

A special very-high speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main

memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations by

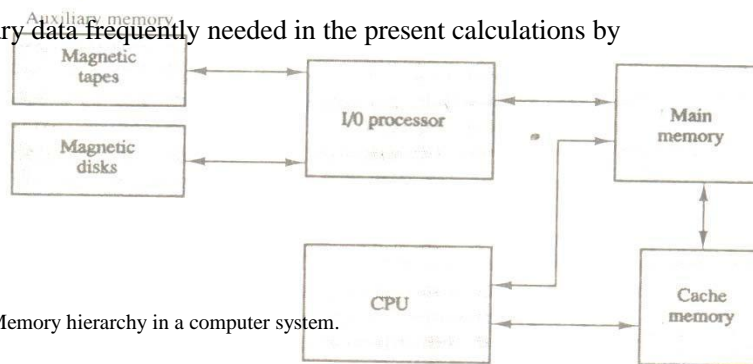


Figure - Memory hierarchy in a computer system.

Making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100ns, while main memory access time may be 700ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from 256 to 2048 words, while cache block size is typically from 1 to 16 words.

Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called multiprogramming, refers to the existence of two or more programs in different parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose that a program is being executed in the CPU and an I/O transfer is required. The CPU initiates the I/O processor to start executing the transfer. This leaves the CPU free to execute another program. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

MAIN MEMORY

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, static and dynamic. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charges on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorted read and write cycles.

Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips. Originally, RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in value one the production of the computer is completed.

Among other things, the ROM portion of main memory is needed for storing an initial program called a bootstrap loader. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer from general use.

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the

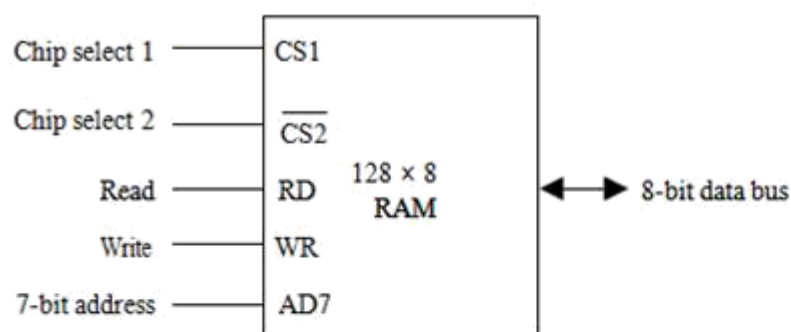
chip interconnection, we will show an example of a 1024×8 memory constructed with 128×8 RAM chips and 512×8 ROM chips.

4.2 RAM AND ROM CHIPS

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a

write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance. The block diagram of a RAM chip is shown in Fig. The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit.

Figure- Typical RAM Chip.



(a) Block diagram

CS1	$\overline{\text{CS2}}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedance
0	1	x	x	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data from RAM
1	1	x	x	Inhibit	High-impedance

(b) Function table

Address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.

The function table listed in Fig. (b) Specifies the operation of the RAM chip. The unit is in operation only when $\text{CS1} = 1$ and $\text{CS2} = 0$. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When $\text{CS1} = 1$ and $\text{CS2} = 0$, the memory can be placed in a

write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in below Fig. For the same-size chip, it is possible to have more bits of ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and CS2 = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

4.2.1.1. MEMORY ADDRESS MAP

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a memory address map, is a pictorial representation of assigned address space for each chip in the system.

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips

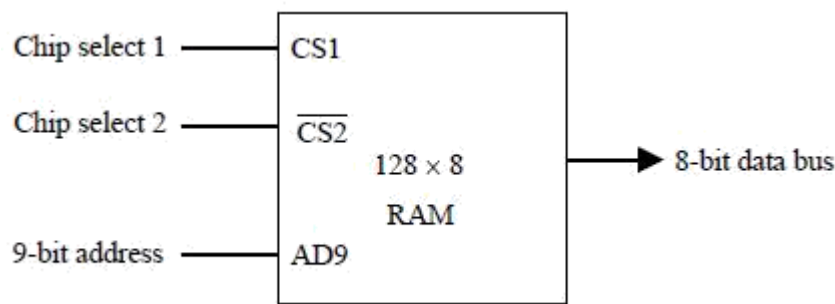


Figure-Typical ROM chip.

To be used are specified in Fig Typical RAM chip and Typical ROM chip. The memory address map for this configuration is shown in Table 7-1. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent

addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero.

The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for

this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space from RAM equal to $2^9 = 512$ bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

The equivalent hexadecimal address for each chip is obtained from the information under the address bus assignment. The address bus lines are subdivided into groups of four bits each so

TABLE-Memory Address Map for Microprocessor

Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000—007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080—00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100—017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180—01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200—03FF	1	x	x	x	x	x	x	x	x	x

That each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. This x's represent a binary number that can range from an all-0's to an all-1's value.

4.2 MEMORY CONNECTION TO CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. The connection of memory chips to the CPU is shown in below Fig. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of Table 7-1. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2×4 decoder whose outputs go to the \overline{CS} input in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The \overline{RD} and \overline{WR} outputs from the microprocessor are applied to the inputs of each RAM

chip.

The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

The example just shown gives an indication of the interconnection complexity that can exist between memory chips and the CPU. The more chips that are connected, the more external decoders are required for selection among the chips. The designer must establish a memory map that assigns addresses to the various chips from which the required connections are determined.

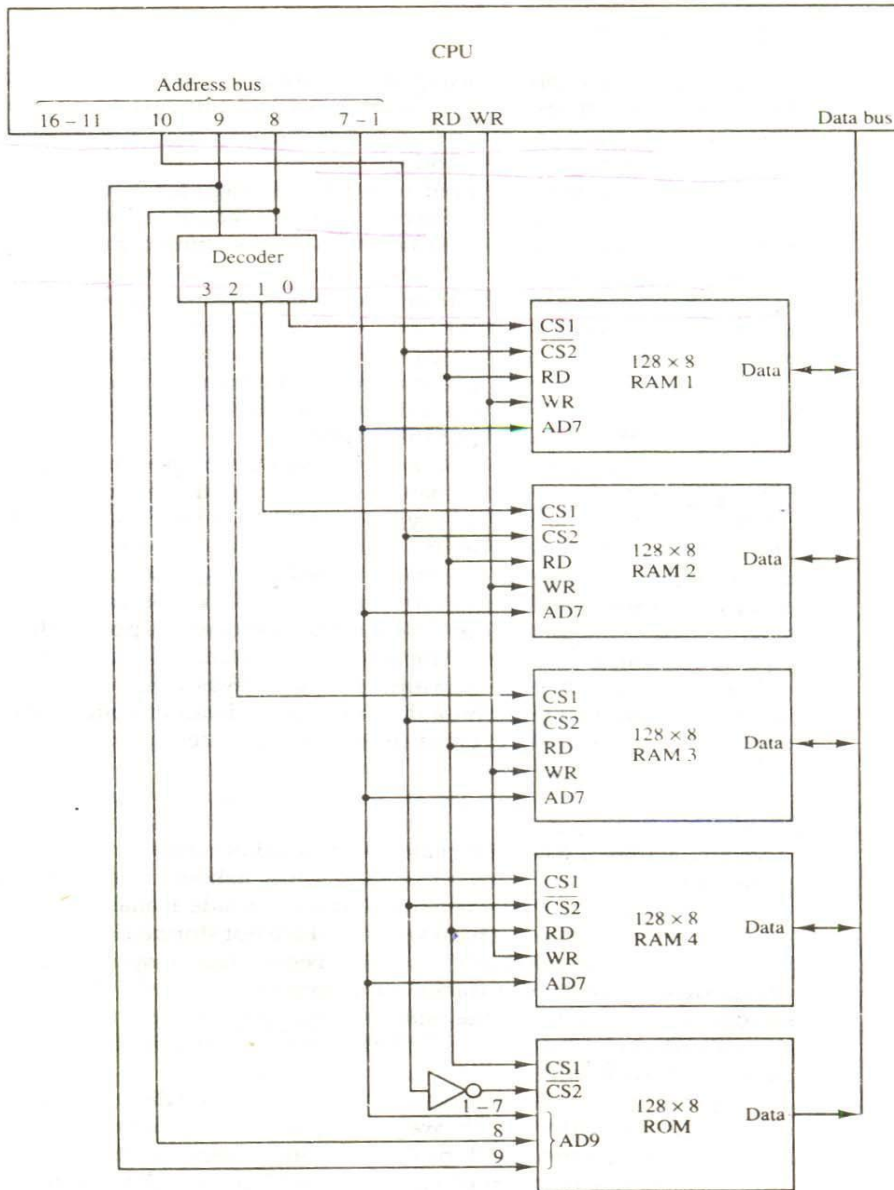


Figure -Memory connection to the CPU.

ASSOCIATIVE MEMORY

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. An account number may be searched in a file to determine the holder's name and account status. The established way to search a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a

sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an associative memory or content addressable memory (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading.

Because of its organization, the associative memory is uniquely suited to do parallel searches by data association. Moreover, searches can be done on an entire word or on a specific field within a word. An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

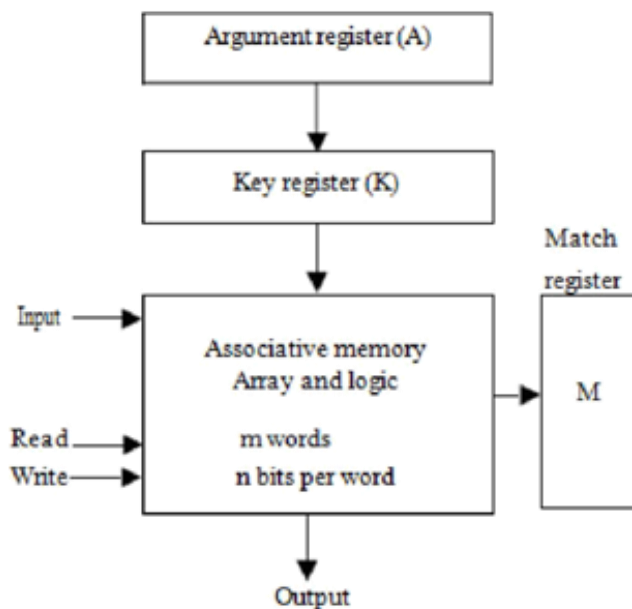
HARDWARE ORGANIZATION

The block diagram of an associative memory is shown in below Fig. It consists of a memory array and logic from words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

The key register provides a mask for choosing a particular field or key in the argument word.

The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared. Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.

Figure- Block diagram of associative memory



To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

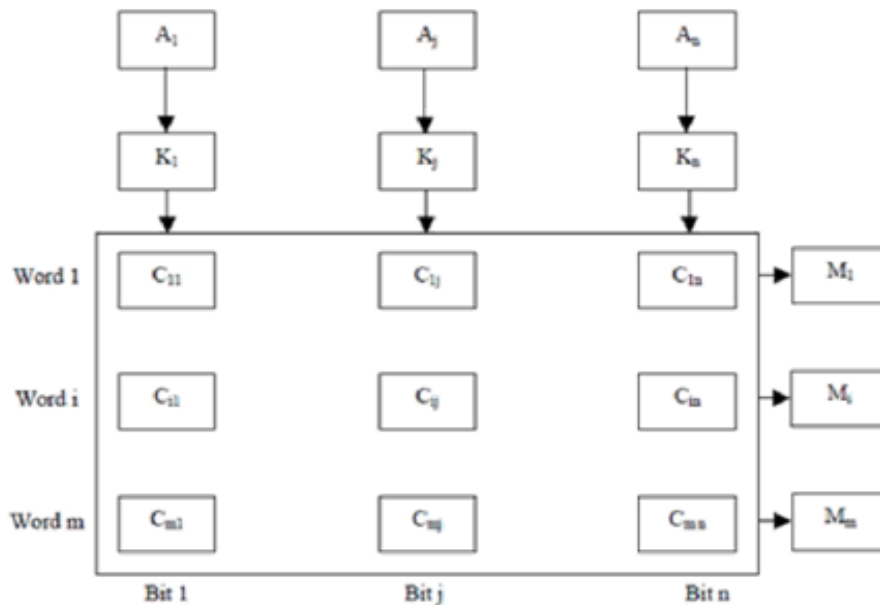
		1	
		1	
		1	
	1	1	
A	(0	
	1	0	
		0	
		0	
		0	
	1	0	
	1	0	
K	1	0	
			n
			o
V		1	
o		1	m
r		1	a
d	1	1	t
	(0	c
1	(0	h
V		0	
o		0	m
r		0	a
d	1	0	t
	(0	c
2	1	1	h

Word 2 matches the unmasked argument field because the three leftmost bits of

the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in below Fig. The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that $K_j = 1$. This is done for all columns $j = 1, 2, \dots, n$. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

Figure -Associative memory of m word, n cells per word



The internal organization of a typical cell C_{ij} is shown in Fig.. It consists of a flip-

Flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i .

MATCH LOGIC

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we neglect the key bits and compare the argument in A with the bits stored in the cells of the words. Word i is equal to the argument in A if $A_j = F_{ij}$ for $j = 1, 2, \dots, n$. Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function $x_j = A_j F_{ij} + A_j \bar{F}_{ij}$

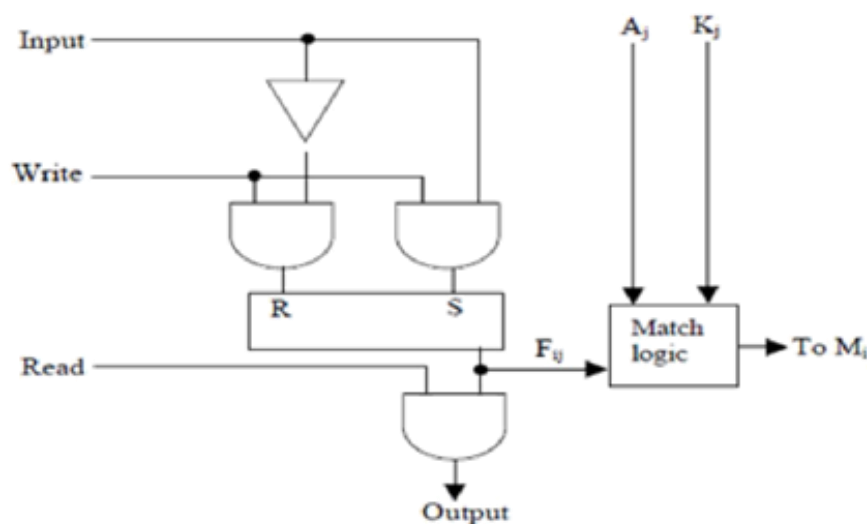
Where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j = 0$.

For a word i to be equal to the argument in a we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is

$$M_i = x_1 x_2 x_3 \dots x_n$$

And constitutes the AND operation of all pairs of matched bits in a word. **Figure**One cell of associative memory.

Figure- One cell of associative memory.



We now include the key bit K_j in the comparison logic. The requirement is that if $K_j = 0$, the corresponding bits of A_j and F_{ij} need no comparison. Only when $K_j = 1$ must they be compared. This requirement is achieved by ORing each term with K_j' , thus:

$$M_i = \bigwedge_{j=1}^n (x_j + K_j')$$

When $K_j = 1$, we have $K_j' = 0$ and $x_j + 0 = x_j$. When $K_j = 0$, then $K_j' = 1$ and $x_j + 1 = 1$. A term $(x_j + K_j')$ will be in the 1 state if its pair of bits is not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when $K_j = 1$.

The match logic for word i in an associative memory can now be expressed by the following Boolean function:

$$M_i = (x_1 + K_1') (x_2 + K_2') (x_3 + K_3') \dots (x_n + K_n')$$

Each term in the expression will be equal to 1 if its corresponding $K_j = 0$.
 if $K_j = 1$, the term will be either 0 or 1 depending on the value of x_j . A match
 will occur and M_i will be equal to 1 if all terms are equal to 1.

If we substitute the original definition of x_j . The Boolean function above can
 be expressed as follows:

$$M_i = \prod_{j=1}^n (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

Where \prod is a product symbol designating the AND operation of all n terms.
 We need m such functions, one for each word $i = 1, 2, 3, \dots, m$.

The circuit for catching one word is shown in below Fig. Each cell
 requires two AND gates and one OR gate. The inverters for A_j and K_j are
 needed once for each column and are used for all bits in the column. The
 output of all OR gates in the cells of the same word go to the input of a
 common AND gate to generate the match signal for M_i . M_i will be logic 1 if a
 catch occurs and 0 if no match occurs.

Note that if the key register contains all 0's, output M_i will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

READ OPERATION

If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the catch register. It is then necessary to scan the bits of the match register one at a time. The matched words are read in sequence by applying a read signal to each word line whose corresponding M_i bit is a 1.

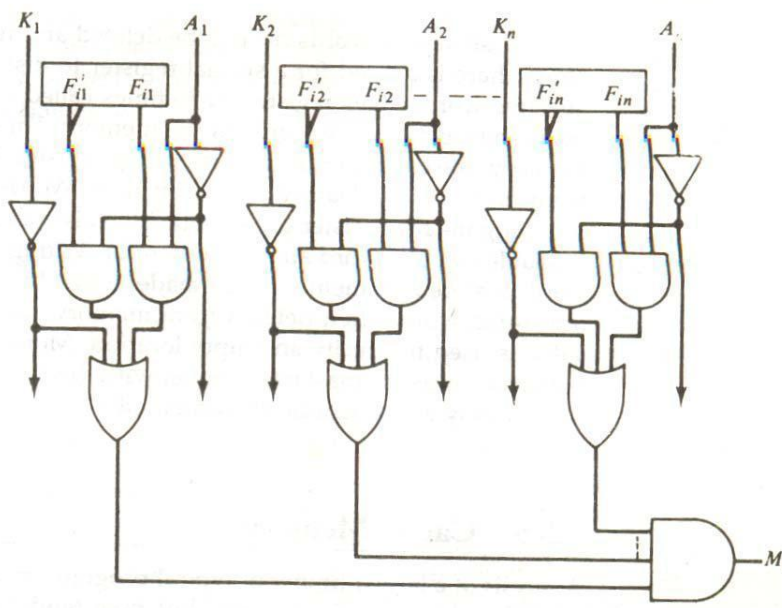


Figure - Match logic for one word of associative memory

In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output M_i directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude words having zero content, an all-zero output will indicate that no match occurred and that the searched item is not available in memory.

WRITE OPERATION

An associative memory must have a write capability for storing the information to be searched. Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make

the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of address lines can be reduced by the decoder to d lines, where $m = 2^d$.

If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special

register to distinguish between active and inactive words. This register, sometimes called a tag register, would have as many bits as there are words in the memory. For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this value is used to specify an empty location. Moreover, the words that have a tag bit of 0 must be masked (together with the K_j bits) with the argument word so that only active words are compared.

4.5 CACHE MEMORY

Analysis of a large number of typical programs has shown that the references, to memory at any given interval of time tend to be confined within a few localized areas in memory. The phenomenon is known as the property of locality of reference. The reason for this property may be understood considering that a typical computer program flows in a straight-line fashion with program loops and subroutine calls encountered frequently. When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, its set of instructions is fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table-lookup procedures repeatedly refer to that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory. The result of all these observations is the locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively frequently.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between the CPU and main memory as illustrated in below Fig. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block

of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

The performance of cache memory is frequently measured in terms of a quantity called hit ratio. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If the word is not found in cache, it is in main memory and it counts as a miss. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The hit ratio is best measured experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

The average memory access time of a computer system can be improved considerably by use of a cache. If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory, the average access time is closer to the access time of the fast cache memory. For example, a computer with cache access time of 100 ns, a main memory access time of 1000 ns, and a hit ratio of 0.9 produces an average access time of 200 ns. This is a considerable improvement over a similar computer without a cache memory, whose access time is 1000 ns.

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

1. Associative mapping
2. Direct mapping
3. Set-associative mapping

To helping the discussion of these three mapping procedures we will use a specific example of a memory organization as shown in below Fig. The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory.

The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

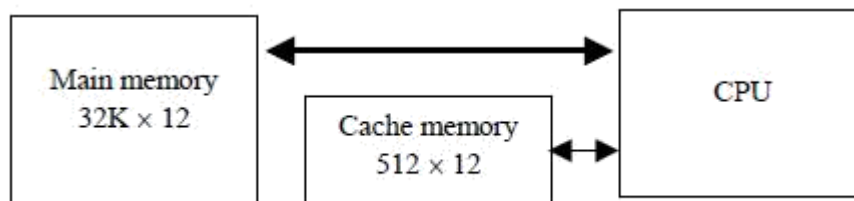
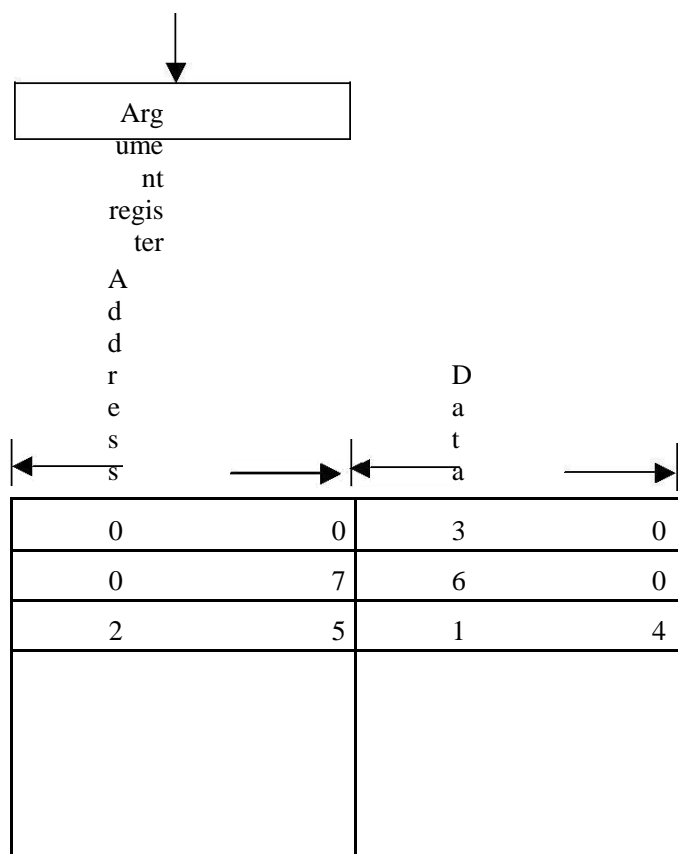


Figure - Example of cache memory**ASSOCIATIVE MAPPING**

The fastest and most flexible cache organization uses an associative memory. This organization is illustrated in below Fig. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read

Figure-Associative mapping cache (all numbers in octal) CPU address (15 bits)

And sent to the CPU. If no match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory. If the cache is full, an address-data pair must be displaced to make room for a pair that is needed and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

DIRECT MAPPING

Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Fig. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and the remaining six bits from the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n -bit memory address is divided into two fields: k bits for the index field and $n - k$ bits for the tag field. The direct mapping cache organization uses the n -bit address to access the main memory and the k -bit

index to access the cache. The internal organization of the words in the cache memory is as shown in Fig. (b). Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.

The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value. The disadvantage of direct mapping is that the hit ratio can droop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such words are relatively far apart in the address range (multiples of 512 locations in this example).

To see how the direct-mapping organization operates, consider the numerical example shown in Fig. The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

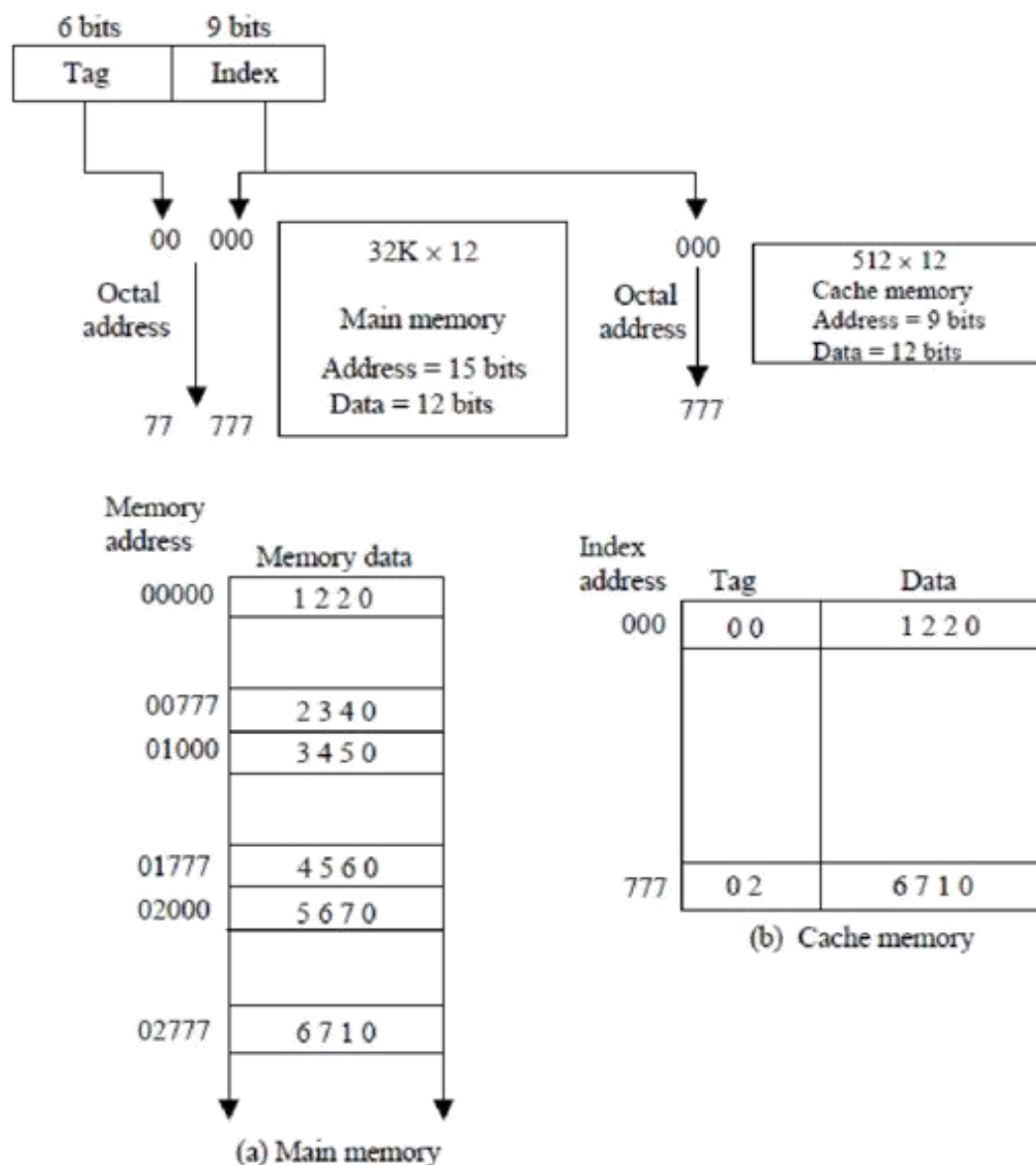
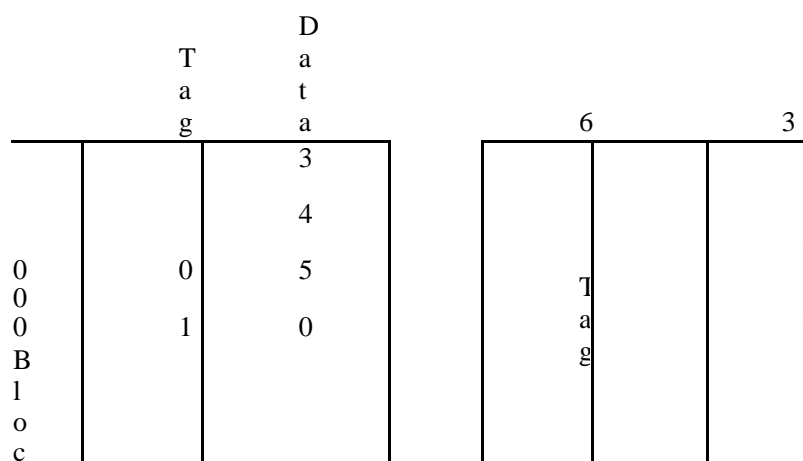


Fig-Direct mapping cache organization

The direct-mapping example just described uses a block size of one word. The same

organization but using a block size of 8 words is shown in below Fig. The index





Specworld.in

and the word within the block is specified with a 3-bit field. The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.

SET-ASSOCIATIVE MAPPING

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time. A third type of cache organization, called set-associative mapping, is an improvement over the direct-mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set. An example of a set-associative cache organization for a set size of two is shown in Fig. Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is $2(6 + 12) = 36$ bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512×36 . It can accommodate 1024 words of main memory since each word of cache contains two data words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

T a g		D a t a		T a g		D a t a	
		3				5	
		4				6	
0		5		0		7	
1		0		2		0	
		6		0		2	

	7		0	3
	1			4
	0			0

Figure- Two-way set-associative mapping cache.

The octal numbers listed in above Fig. are with reference to the main memory content illustrated in Fig.(a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a catch occurs. The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name “set-associative”. The hit ratio will improve as the set size increases because more words with the same index but different tag can reside in cache. However, an increase in the set size increases the number of bits in words of cache and requires more complex comparison logic.

When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: random replacement, first-in, first out (FIFO), and least recently used (LRU). With the random replacement policy the control chooses one tag-data item for replacement at random. The FIFO procedure selects for replacement the item that has been in the set the longest. The LRU algorithm selects for replacement the item that has been least recently used by the CPU. Both FIFO and LRU can be implemented by adding a few extra bits in each word of cache.

WRITING INTO CACHE

An important aspect of cache organization is concerned with memory write requests. When the CPU finds a word in cache during read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the write-through method. This method has the advantage that main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

The second procedure is called the write-back method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the words are removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests for the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy needs to be rewritten into main memory. Analytical results indicate that the number of memory writes in a typical program ranges between 10 and 30 percent of the total references to memory.

CACHE INITIALIZATION

One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, but in effect it contains some non-valid data. It is customary to include with each word in cache a valid bit to indicate

whether or not the word contains valid data.

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The introduction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data. Thus the initialization condition has the effect of forcing misses from the cache until it fills with valid data.

VIRTUAL MEMORY

In a memory hierarchy system, programs and data are brought into main memory as they are needed by the CPU. Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

ADDRESS SPACE AND MEMORY SPACE

An address used by a programmer will be called a virtual address, and the set of such addresses the address space. An address in main memory is called a location or physical address. The set of such locations is called the memory space. Thus the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing. In most computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space in computers with virtual memory.

As an illustration, consider a computer with a main -memory capacity of 32K words ($K = 1024$). Fifteen bits are needed to specify a physical address in memory since $32K = 2^{15}$. Suppose that the computer has available auxiliary memory for storing $2^{20} = 1024K$ words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M , we then have for this example $N = 1024K$ and $M = 32K$.

In a multiprogramming computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data are moved from auxiliary memory into main memory as shown in Fig. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

In a virtual memory system, programmers are told that they have the

total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long. (Remember

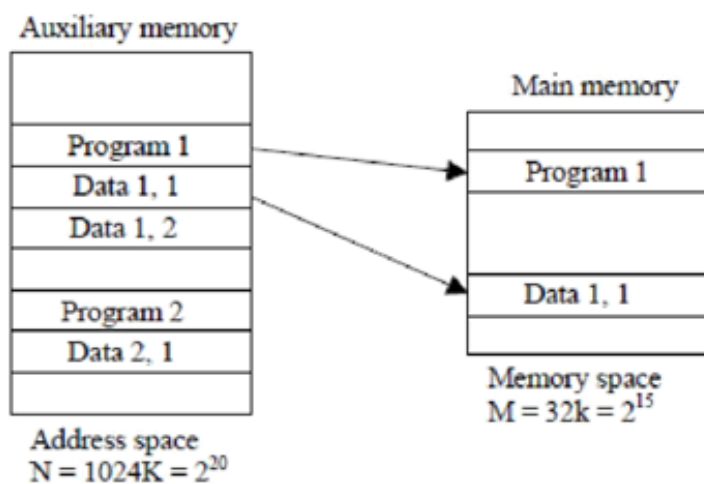
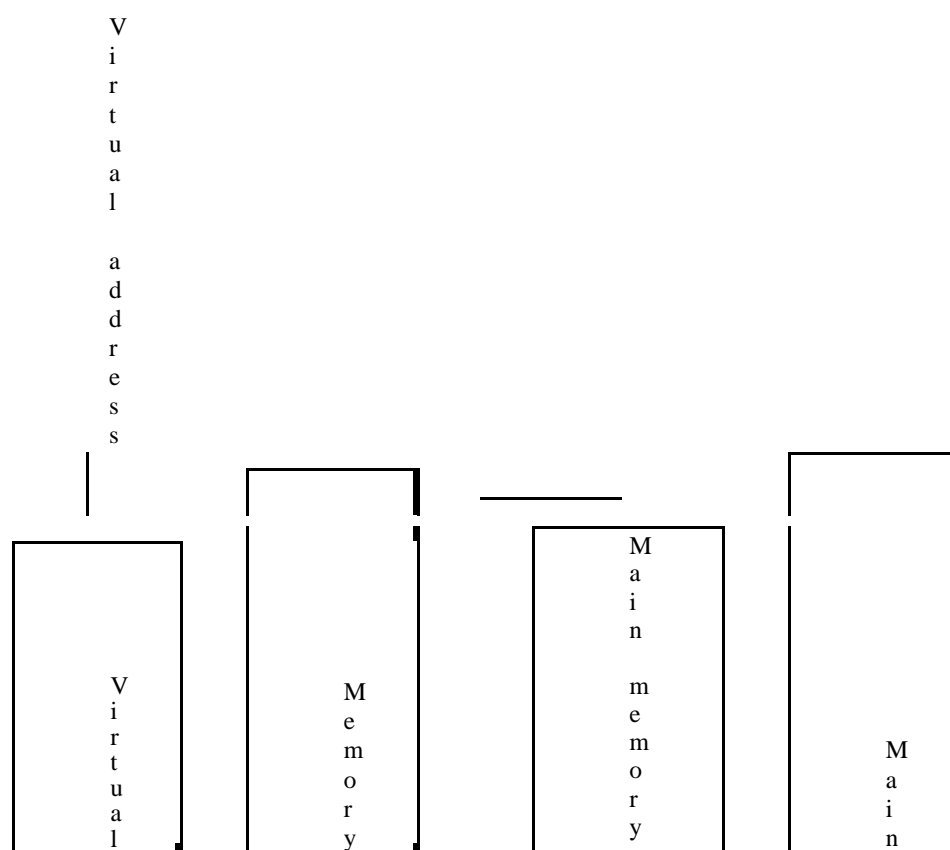


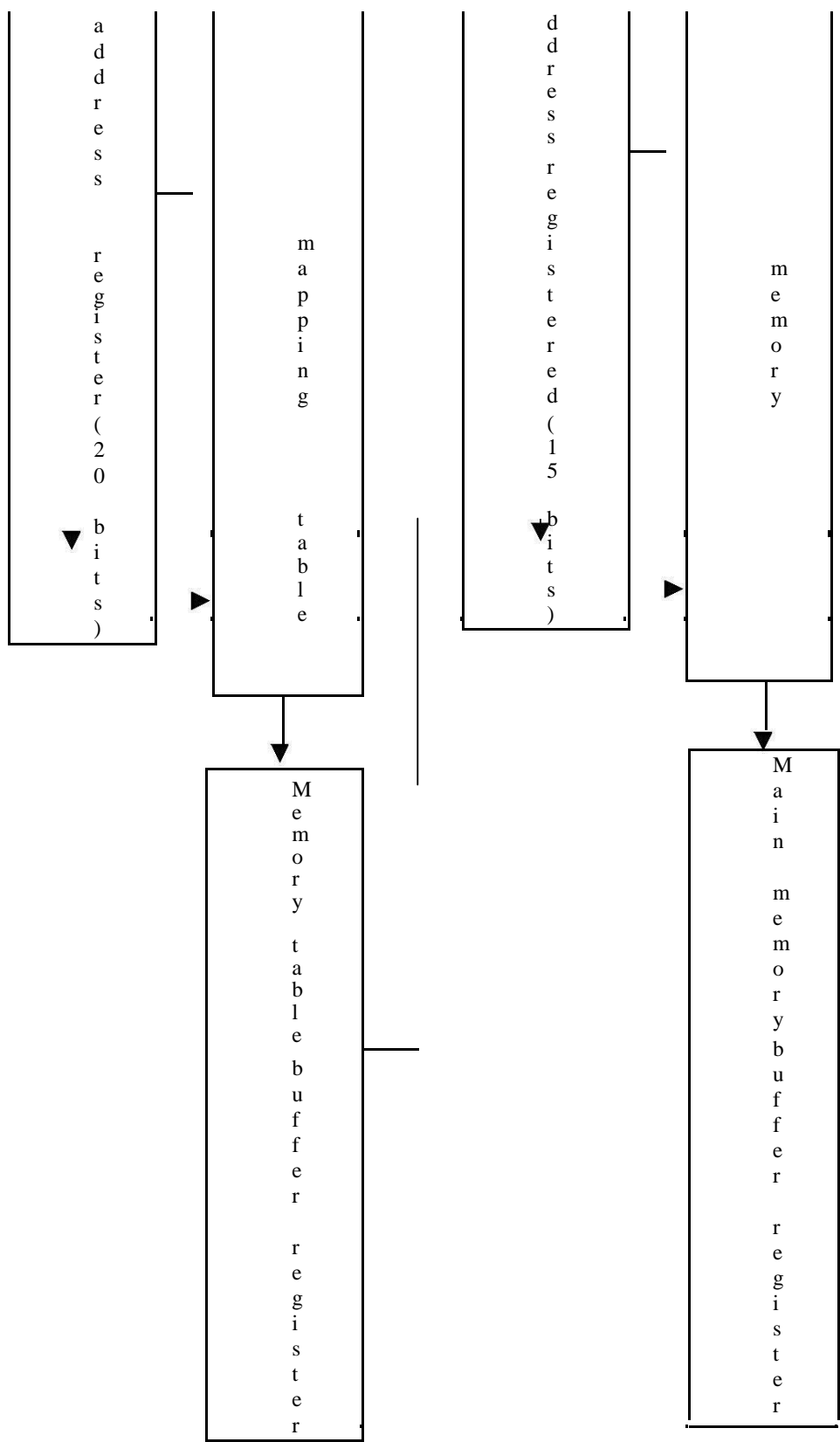
Fig-Relation between address and memory space in a virtual memory system

That for efficient transfers, auxiliary storage moves an entire record to the main memory). A table is then needed, as shown in Fig, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

The mapping table may be stored in a separate memory as shown in Fig. or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table

Figure - Memory table for mapping a virtual address.





Takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory as explained below.

ADDRESS MAPPING USING PAGES

The table implementation of the address mapping is simplified if the

information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size. For example, if a page or block consists of 1K words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks. Although both a page and a block are split into groups of 1K words, a page

refers to the organization of address space, while a block refers to the organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term “page frame” is sometimes used to denote a block.

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example of Fig, a virtual address has 13 bits. Since each page consists of $2^{10} = 1024$ words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

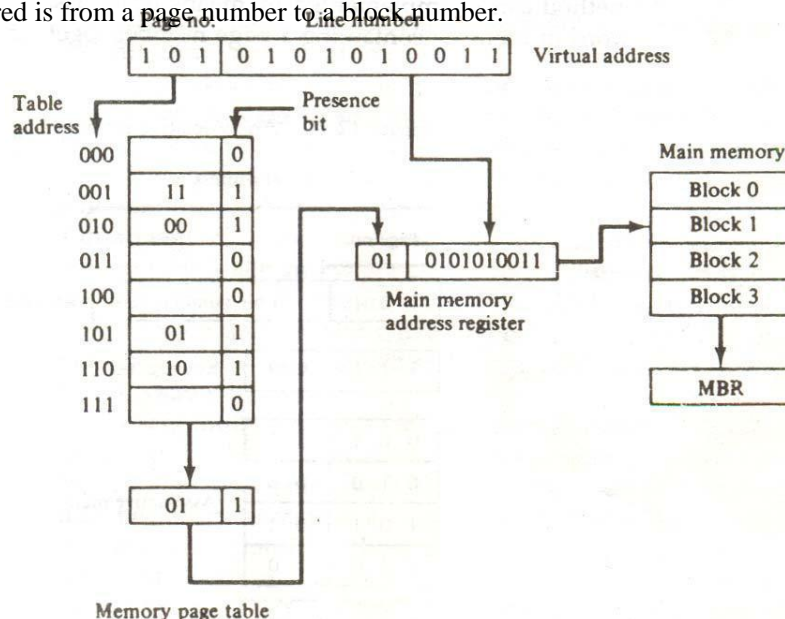


Figure-Memory table in a paged system.

The word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before

resuming computation.

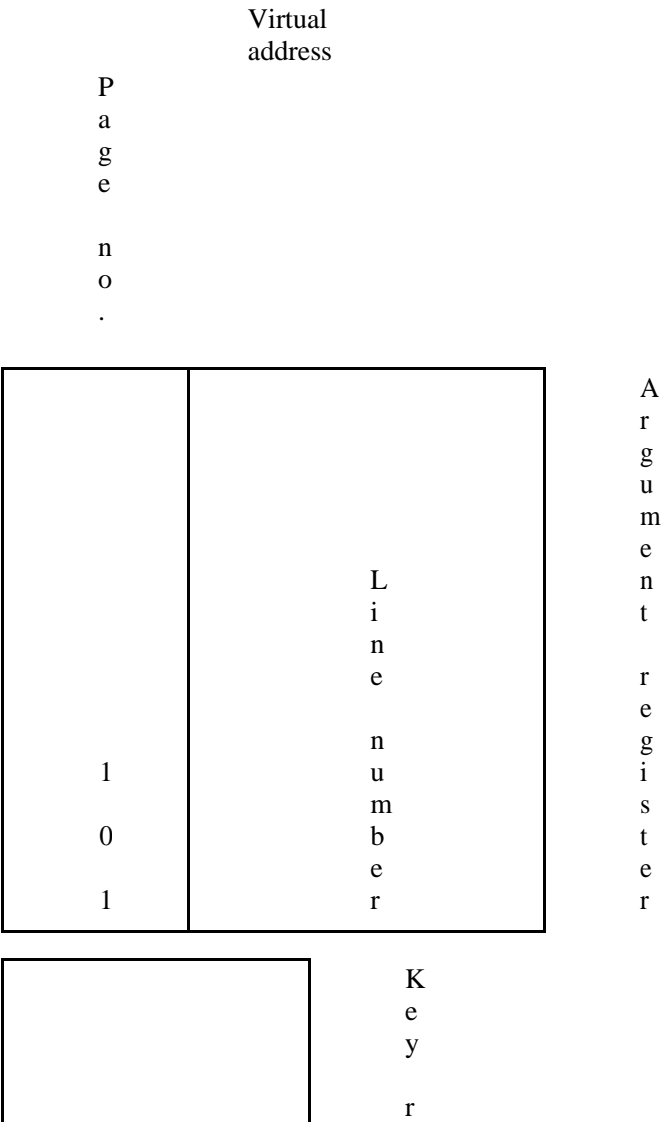
ASSOCIATIVE MEMORY PAGE TABLE

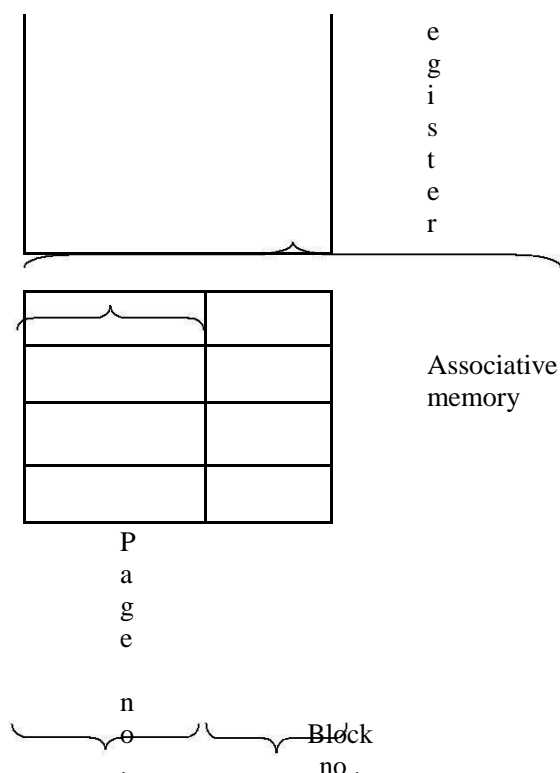
A random-access memory page table is inefficient with respect to storage utilization. In the example of below Fig. we observe that eight words of memory are needed, one for each page, but at least four words will always be marked empty because main memory cannot accommodate more than

four blocks. In general, system with n pages and m blocks would require a memory-page table of n locations of which up to m blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding block number. The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Figure -An associative memory page table.





Consider again the case of eight pages and four blocks as in the example of Fig. We replace the random access memory-page table with an associative memory of four words as shown in Fig. Each entry in the associative memory array consists of two fields. The first three bits specify a field from storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

4.5.1.1 PAGE REPLACEMENT

A virtual memory system is a combination of hardware and software techniques. The memory

management software system handles all the software operations for the efficient utilization of memory space. It must decide (1) which page in main memory ought to be removed to make room for a new page, (2) when a new page is to be transferred from auxiliary memory to main memory, and (3) where

the page is to be placed in main memory. The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual memory.

When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called page fault. When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In the meantime, controls transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.

When a page fault occurs in a virtual memory system, it signifies that the page referenced by the CPU is not in main memory. A new page is then transferred from auxiliary memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make room for the new page. The policy for choosing pages to remove is determined from the replacement algorithm that is used. The goal of a replacement policy is to try to remove the page least likely to be referenced in the immediate future.

Two of the most common replacement algorithms used are the first-in first-out (FIFO) and the least recently used (LRU). The FIFO algorithm selects for replacement the page that has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks. When a new page must be loaded, the page least recently brought in is removed. The page to be removed is easily determined because its identification number is at the top of the FIFO stack. The FIFO replacement policy has the advantage of being easy to implement. It has the disadvantages that under certain circumstances pages are removed and loaded from memory too frequently.

The LRU policy is more difficult to implement but has been more attractive on the assumption that the least recently used page is a better candidate for removal than the least recently loaded pages in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the

page with the highest count. The counters are often called aging registers, as their count indicates their age, that is, how long ago their associated pages have been reference.

UNIT-5

Input / Output Organization:

- 5.1 Introduction To I/O
- 5.2 Interrupts- Hardware
- 5.3 Enabling And Disabling Interrupts
- 5.4 Device Control
- 5.5 Direct Memory Access
- 5.6 Buses
- 5.7 Interface Circuits

5.1 INTRODUCTION TO I/O DEVICES

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement. The bus enables all the devices connected to it to exchange information. Typically, it consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address line, the device that recognizes this address responds to the commands issued on the control lines. The processor requests 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.

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 the input buffer associated with the keyboard, the instruction

Move DATAIN, R0

from DATAIN and stores them into processor register R0. Similarly, the instruction Move R0, DATAOUT

Sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer.

Most computer systems use memory-mapped I/O. some processors have special In and Out instructions to perform I/O transfers. When building a computer system based on these processors, the designer had the option of connecting I/O devices to use the special I/O address space or simply incorporating them as part of the memory address space. The I/O devices examine the low-order bits of the address bus to determine whether they should respond.

The hardware required to connect an I/O device to the bus. The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus

and assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. Also, we must make sure that an input character is read only once.

This example illustrates program-controlled I/O, in which the processor repeatedly checks a status flag to achieve the required synchronization between the processor and an input or output device. We say that the processor polls the device. There are two other commonly used mechanisms for implementing I/O operations: interrupts and direct memory access. In the case of interrupts, synchronization is achieved by having the I/O device send a special signal over the bus whenever it is ready for a data transfer operation. Direct memory access is a technique used for high-speed I/O devices. It involves having the device interface transfer data directly to or from the memory, without continuous involvement by the processor.

The routine executed in response to an interrupt request is called the interrupt-service routine, which is the PRINT routine in our example. Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction i in figure 1

Figure 1. Transfer of control through the use of interrupts

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 has to come back 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 the temporary storage location, causing execution to resume at instruction $i + 1$. In many processors, the return address is saved 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 may be accomplished by means of a special control signal on the bus. An interrupt-acknowledge signal. The execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface implicitly informs that 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. However, the interrupt-service routine may not have anything in

common with the program being executed at the time the interrupt request is received. In fact, the two programs often belong to different users. Therefore, before starting execution of the interrupt-service routine, any information that may be altered during the execution of that routine must be saved. This information must be restored before execution of the interrupt 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 information that needs to be saved and restored typically includes the condition code flags and the contents of any registers used by both the interrupted program and the interrupt-service routine.

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 registers involves memory transfers that increase the total execution time, and hence represent execution overhead. Saving registers also increase 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.

5.2 INTERRUPT HARDWARE:

We pointed out that an I/O device requests an interrupt by activating a bus line called interrupt-request. Most computers are likely to have several I/O devices that can request an interrupt. A single interrupt-request line may be used to serve n devices as depicted. All devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all interrupt-request signals $INTR_1$ to $INTR_n$ are inactive, that is, if all switches are open, the voltage on the interrupt-request line will be equal to V_{DD} . This is the inactive state of the line. Since the closing of one or more switches will cause the line voltage to drop to 0, the value of $INTR$ is the logical OR of the requests from individual devices, that is,

$$INTR = INTR_1 + \dots + INTR_n$$

It is customary to use the complemented form, ***INTR***, to name the interrupt-request signal on the common line, because this signal is active when in the low-voltage state.

5.3 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 the envisaged by the programmer. Hence, the interruption of program execution must be carefully controlled.

Let us consider in detail 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.

The first possibility is to have the processor hardware ignore the interrupt-request line until the execution of the first instruction of the interrupt-service routine has been completed. Then, by using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interrupt-enable instruction is executed. Typically, the Interrupt-enable instruction will be the last instruction in the interrupt-service routine before the Return-from-interrupt instruction. The processor must guarantee that execution of the Return-from-interrupt instruction is completed before further interruption can occur.

The second option, which is suitable for a simple processor with only one interrupt-request line, is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. After saving the contents of the PC

and the processor status register (PS) on the stack, the processor performs the equivalent of executing an Interrupt-disable instruction. It is often the case that one bit in the PS register, called Interrupt-enable, indicates whether interrupts are enabled.

In the third option, the processor has a special interrupt-request line for which the interrupt-handling circuit responds only to the leading edge of the signal. Such a line is said to be edge-triggered.

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, the following is a typical scenario.

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.
3. Interrupts are disabled by changing the control bits in the PS (except in the case of edge-triggered interrupts).
4. The device is informed that its request has been recognized, and in response, it deactivates the

interrupt-request signal.

5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

5.4. 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 in 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

How can the processor recognize the device requesting an interrupts?

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?

Should a device be allowed to interrupt the processor while another interrupt is being serviced? How should two or more simultaneous interrupt requests be handled?

The means by which these problems are resolved 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 a request is received over the common interrupt-request line, additional information is needed to identify the particular device that activated the line.

The information needed to determine whether a device is requesting an interrupt is available in its status register. When a device raises an interrupt request, it sets to 1 one of the bits in its status register, which we will call the IRQ bit. For example, bits KIRQ and DIRQ are the interrupt request bits for the keyboard and the display, respectively. The simplest way to identify the interrupting device is to have the interrupt-service routine poll all the I/O devices connected to the bus. The first device encountered with its IRQ bit set is the device that should be serviced. An appropriate subroutine is 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 all the 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 all interrupt-handling schemes based on this approach.

A device requesting an interrupt can identify itself by sending a special code to the processor over the bus. This enables the processor to identify individual devices even if they share a single interrupt-request line. The code supplied by the device may represent the starting address of the interrupt-service routine for that device. The code length is typically in the range of 4 to 8 bits. The remainder of the address is supplied by the processor based on the area in its memory where the addresses for interrupt-service routines are located.

This arrangement implies that the interrupt-service routine for a given device must always start at the same location. The programmer can gain some flexibility by storing in this location an instruction that causes a branch to the appropriate routine.

Interrupt Nesting: -

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.

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 another 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.

The processor's priority is usually encoded in a few bits of the processor status word. It can be changed by program instructions that write into the PS. These are privileged instructions, which can be executed only while the processor is running in the supervisor mode. The processor is in the supervisor mode only when executing operating system routines. It switches to the user mode before beginning to execute application programs. Thus, a user program cannot accidentally, or intentionally, change the priority of the processor and disrupt the system's operation. An attempt to execute a privileged instruction while in the user mode leads to a special type of interrupt called a privileged instruction.

A multiple-priority scheme can be implemented easily by using separate interrupt-request and interrupt-acknowledge lines for each device, as shown in figure. Each of the interrupt-request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

Priority arbitration Circuit

Figure2: Implementation of interrupt priority using individual interrupt-request

and acknowledge lines.

Simultaneous Requests:-

Let us now consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which requests to service first. Using a priority scheme such as that of figure, the solution is straightforward. The processor simply accepts the

requests having the highest priority.

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. A widely used scheme

is to connect the devices to form a daisy chain, as shown in figure 3a. The interrupt-

INTR
request line is

common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices.

5.5. DIRECT MEMORY ACCESS:

The discussion in the previous sections concentrates on data transfer between the processor and I/O devices. Data are transferred by executing instructions such as

Move DATAIN, R0

An instruction to transfer input or output data is executed only after the processor determines that the I/O device is ready. To do this, the processor either polls a status flag in the device interface or waits for the device to send an interrupt request. In either case, considerable overhead is incurred, because several program instructions must be executed for each data word transferred. In addition to polling the status register of the device, instructions are needed for incrementing the memory address and keeping track of the word count. When interrupts are used, there is the additional overhead associated with saving and restoring the program counter and other state information.

To transfer large blocks of data at high speed, an alternative approach is used. A

special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called direct memory access, or DMA.

DMA transfers are performed by a control circuit that is part of the I/O device interface. We refer to this circuit as a DMA controller. 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 all the bus signals that control data transfer. Since it has to transfer blocks of data, the DMA controller must increment the memory address for successive words and keep track of the number of transfers.

Although a DMA controller can transfer data without intervention by the processor, its operation must be under the control of a program executed by the processor. To initiate the transfer of a block of words, the processor sends the starting address, the number of words in the block, and the direction of the transfer. On receiving this information, the DMA controller proceeds to perform the requested operation. When the entire block has been transferred, the controller informs the processor by raising an interrupt signal.

While a DMA transfer is taking place, the program that requested the transfer cannot continue, and the processor can be used to execute another program. After the DMA transfer is completed, the processor can return to the program that requested the transfer.

I/O operations are always performed by the operating system of the computer in response to a request from an application program. The OS is also responsible for suspending the execution of one program and starting another. Thus, for an I/O operation involving DMA, the OS puts the program that requested the transfer in the Blocked state, initiates the DMA operation, and starts the execution of another program. When the transfer is completed, the DMA controller informs the processor by sending an interrupt request. In response, the OS puts the suspended program in the Runnable state so that it can be selected by the scheduler to continue execution.

Figure 4 shows an example of the DMA controller registers that are

accessed by the processor to initiate transfer operations. Two registers are used for storing the

S
t
a
t
u
s

a
n
d

C
o
n
t
r
o
l

	3	3	1	
		0		

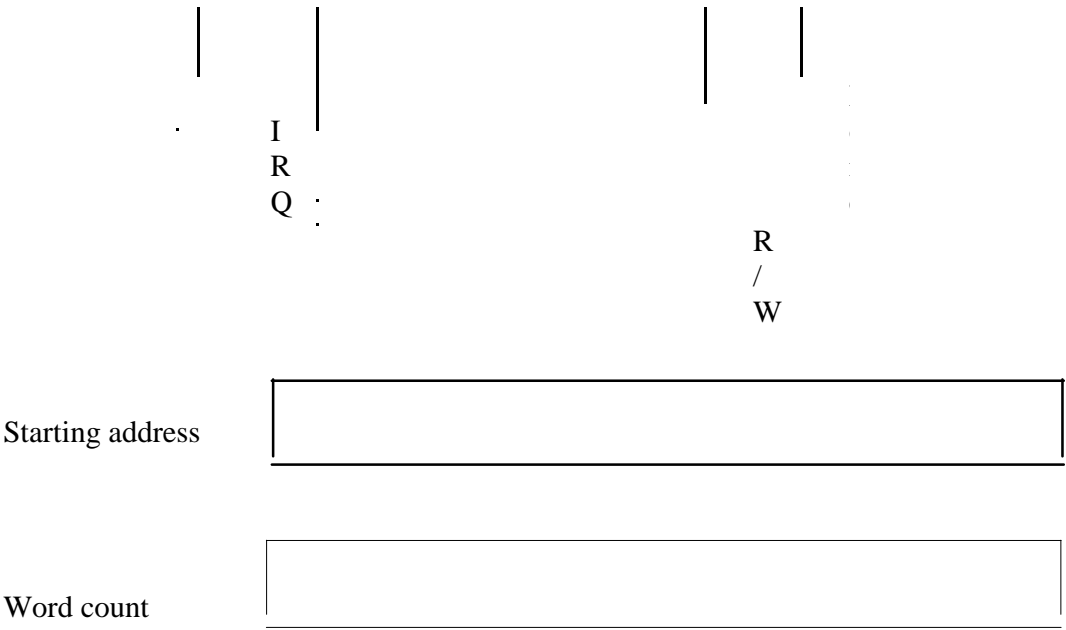
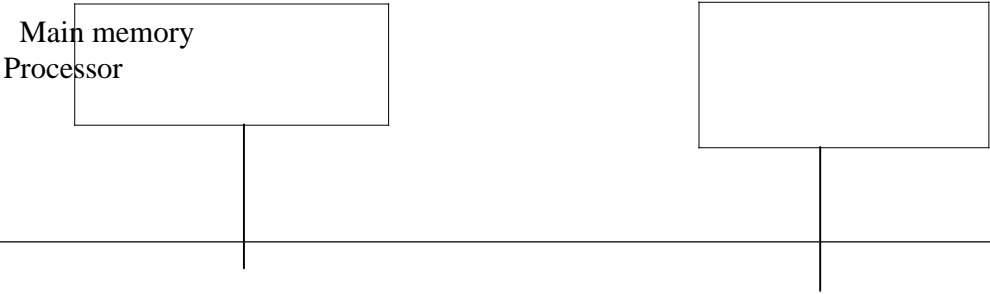


Figure 4 Registers in DMA interface



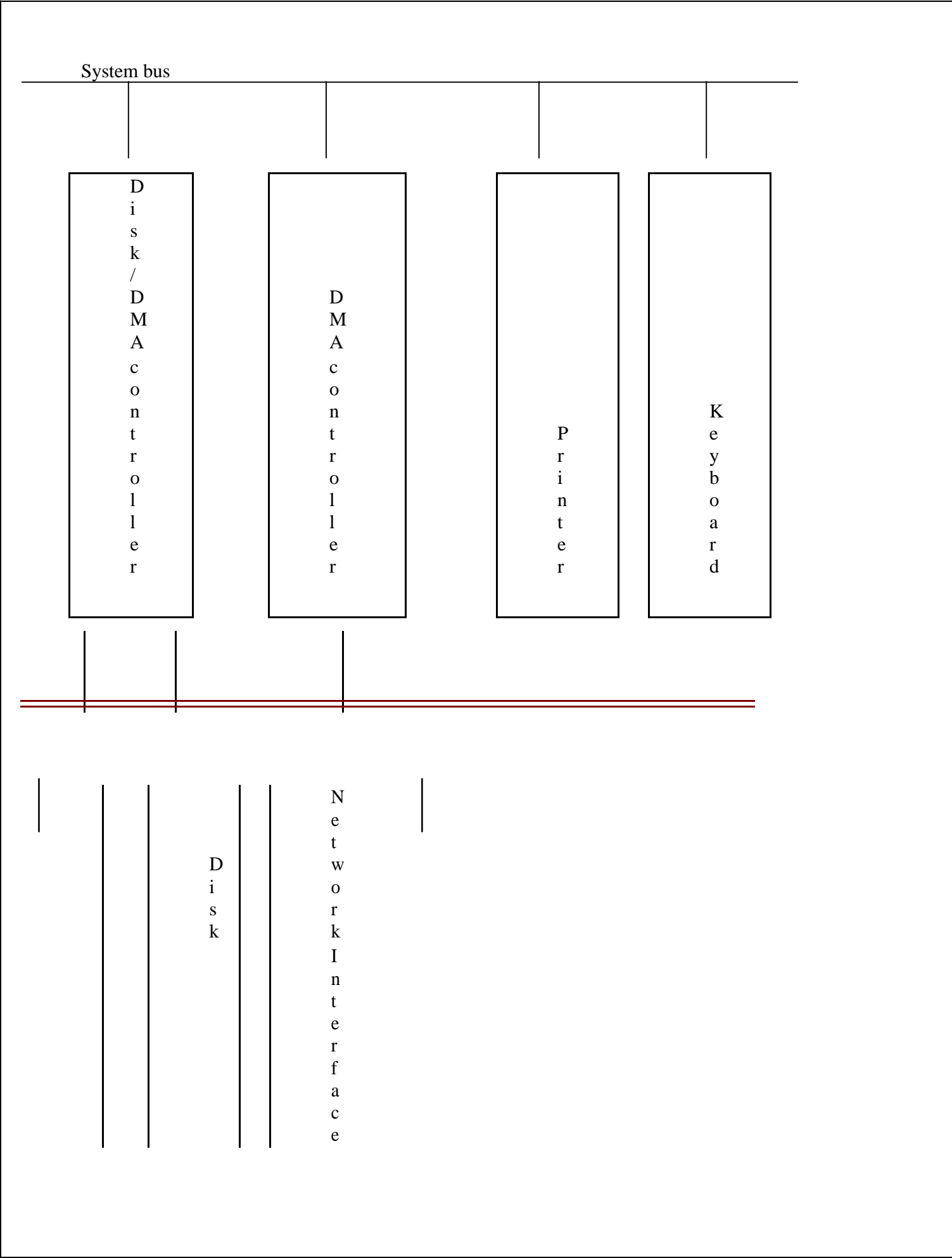


Figure 5 Use of DMA controllers in a computer system

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. 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.

An example of a computer system is given in above figure, showing how DMA controllers may be used. A DMA controller connects a high-speed network to the computer bus. 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 device.

To start a DMA transfer of a block of data from the main memory to one of the disks, a program writes the address and word count information into the registers of the corresponding channel of the disk controller. It also provides the disk controller with information to identify the data for future retrieval. The DMA controller proceeds independently to implement the specified operation when the DMA 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 can also be used to record other information, such as whether the transfer took place correctly or errors occurred.

Memory accesses by the processor and the DMA controller are interwoven. Requests by DMA devices for using the bus are always given higher priority than processor requests. Among different DMA devices, top priority is given to high-speed peripherals such as a disk, a high-speed network interface, or a graphics display device. Since the processor originates most memory access cycles, the DMA controller can be said to “steal” memory cycles from the processor. Hence, the interweaving technique is usually called cycle stealing. Alternatively, the DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as block or burst mode.

Most DMA controllers incorporate a data storage buffer. In the case of the network interface in figure 5 for example, the DMA controller reads a block of data from the main memory and stores it into its input buffer. This transfer takes place using burst mode at a speed appropriate to the memory and the computer bus. Then, the data in the buffer are transmitted over the network at the speed of the network.

A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus

at the same time to access the main memory. To resolve these conflicts, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers.

Bus Arbitration:-

The device that is allowed to initiate data transfers on the bus at any given time is called the bus master. When the current master relinquishes control of the bus, another device can acquire this status. Bus arbitration is the process by which the next device to become the bus master is selected and bus mastership is transferred to it. The selection of the bus master must take into account the needs of various devices by establishing a priority system for gaining access to the bus.

There are two approaches to bus arbitration: centralized and distributed. In centralized arbitration, a single bus arbiter performs the required arbitration. In distributed arbitration, all devices participate in the selection of the next bus master.

Centralized Arbitration:-

The bus arbiter may be the processor or a separate unit connected to the bus. A basic arrangement in which the processor contains the bus arbitration circuitry. In this case, the processor is normally the bus master unless it grants bus mastership to one of the DMA controllers. A DMA controller indicates that it needs to become the bus

master by activating the Bus-Request line, *BR*. The signal on the Bus-Request line is the logical OR of the bus

requests from all the devices connected to it. When Bus-Request is activated, the processor activates the Bus-Grant signal, BG1, indicating to the DMA controllers that they may use the bus when it becomes free. This signal is connected to all DMA controllers using a daisy-chain arrangement. Thus, if DMA controller 1 is requesting the bus, it blocks the propagation of the grant signal to other devices. Otherwise, it passes the grant downstream by asserting BG2. The current bus master indicates to all device that it is using the bus by activating another open-controller line

called Bus-Busy, *BBSY*. Hence, after receiving the Bus-Grant signal, a DMA controller waits for Bus-Busy to become inactive, then assumes mastership of the bus. At this time, it activates Bus-Busy to prevent other devices from using the bus at the same time.

5.6.BUSES

Peripheral Component Interconnect (PCI) Bus:-

The PCI bus is a good example of a system bus that grew out of the need for standardization. It supports the functions found on a processor bus but in a standardized format that is independent of any particular processor. Devices connected to the PCI bus appear to the processor as if they were connected directly to the processor bus. They are assigned addresses in the memory address space of the processor.

The PCI follows a sequence of bus standards that were used primarily in IBM PCs. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 80x86 processors. Later, the 16-bit bus used on the PC At computers became known as the ISA bus. Its extended 32-bit version is known as the EISA bus. Other buses developed in the eighties with similar capabilities are the Microchannel used in IBM PCs and the NuBus used in Macintosh computers.

The PCI was developed as a low-cost bus that is truly processor independent. Its design anticipated a rapidly growing demand for bus bandwidth to support high-speed disks and graphic and video devices, as well as the specialized needs of multiprocessor systems. As a result, the PCI is still popular as an industry standard almost a decade after it was first introduced in 1992.

An important feature that the PCI pioneered is a plug-and-play capability for connecting I/O devices. To connect a new device, the user simply connects the device interface board to the bus. The software takes care of the rest.

Data Transfer:-

In today's computers, most memory transfers involve a burst of data rather than just one word. The reason is that modern processors include a cache memory. Data are transferred between the cache and the main memory in burst of several words each. The words involved in such a transfer are stored at successive memory locations. When the processor (actually the cache controller) specifies an address and requests a read operation from the main memory, the memory responds by sending a sequence of data words starting at that address. Similarly, during a write operation, the processor sends a memory address followed by a sequence of data words, to be written in successive memory locations starting at the address. The PCI is designed primarily to support this mode of operation. A read or write operation involving a single word is simply treated as a burst of length one.

The bus supports three independent address spaces: memory, I/O, and configuration. The first two are self explanatory. The I/O address space is intended for use with processors, such as Pentium, that have a separate I/O address space. However, as noted, the system designer may choose to use memory-mapped I/O even when a separate I/O address space is available. In fact, this is the approach recommended by the PCI its plug-and-play capability. A 4-bit command that accompanies the address identifies which of the three spaces is being used in a given data transfer operation.

The signaling convention on the PCI bus is similar to the one used, we assumed that the master maintains the address information on the bus until data transfer is completed. But,

this is not necessary. The address is needed only long enough for the slave to be selected. The slave can store the address in its internal buffer. Thus, the address is needed on the bus for one clock cycle only, freeing the address lines to be used for sending data in subsequent clock cycles. The result is a significant cost reduction because the number of wires on a bus is an important cost factor. This approach is used in the PCI bus.

At any given time, one device is the bus master. It has the right to initiate data transfers by issuing read and

write commands. A master is called an initiator in PCI terminology. This is either a processor or a DMA controller. The addressed device that responds to read and write commands is called a target.

Device Configuration:-

When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it.

The PCI simplifies this process by incorporating in each I/O device interface a small configuration ROM memory that stores information about that device. The configuration ROMs of all devices is accessible in the configuration address space. The PCI initialization software reads these ROMs whenever the system is powered up or reset. In each case, it determines whether the device is a printer, a keyboard, an Ethernet interface, or a disk controller. It can further learn about various device options and characteristics.

Devices are assigned addresses during the initialization process. This means that during the bus configuration operation, devices cannot be accessed based on their address, as they have not yet been assigned one. Hence, the configuration address space uses a different mechanism. Each device has an input signal called Initialization Device Select, IDSEL#.

The PCI bus has gained great popularity in the PC world. It is also used in many other computers, such as SUNs, to benefit from the wide range of I/O devices for which a PCI interface is available. In the case of some processors, such as the Compaq Alpha, the PCI-processor bridge circuit is built on the processor chip itself, further simplifying system design and packaging.

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) under the designation X3.131. In the original specifications of the standard, devices such as disks 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 up to 5 megabytes/s.

The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options. A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Alternatively, a wide SCSI bus has 16 data lines and transfers data 16 bits at a time. There are also several

options for the electrical signaling scheme used.

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. The SCSI bus is connected to the processor bus through a SCSI controller. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa. A packet may contain a block of data, commands from the processor to the device, or status information about the device.

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.

A controller connected to a SCSI bus is one of two types – an initiator or a target. An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. Clearly, the controller on the processor side, such as the SCSI controller, must be able to operate as an initiator. The disk controller operates as a target. It carries out the commands it receives from the initiator. The initiator establishes a logical connection with the intended target. Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data. While a particular connection is suspended, other device can use the bus to transfer information. This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator.

The processor sends a command to the SCSI controller, which causes the following sequence of event to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
1. The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
2. The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
3. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.

4. The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
5. The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator as before. At the end of this transfers, the logical connection between the two controllers is terminated.
6. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.

7. The SCSI controller sends an interrupt to the processor to inform it that the requested operation has been completed

This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. In this context, a “higher level” means that the messages refer to 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 operation of the particular device involved in a data transfer. In the preceding example, the processor need not be involved in the disk seek operation.

UNIVERSAL SERIAL BUS (USB):-

The synergy between computers and communication is at the heart of today's information technology revolution. A modern computer system is likely to involve a wide variety of devices such as keyboards, microphones, cameras, speakers, and display devices. Most computers also have a wired or wireless connection to the Internet. A key requirement in such an environment is the availability of a simple, low-cost mechanism to connect these devices to the computer, and an important recent development in this regard is the introduction of the Universal Serial Bus (USB). This is an industry standard developed through a collaborative effort of several computer and communication companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, and Philips.

The USB supports two speeds of operation, called low-speed (1.5 megabits/s) and full-speed (12 megabits/s). The most recent revision of the bus specification (USB 2.0) introduced a third speed of operation, called high-speed (480 megabits/s). The USB is quickly gaining acceptance in the market place, and with the addition of the high-speed capability it may well become the interconnection method of choice for most computer devices.

The USB has been designed to meet several key objectives:

1. Provides a simple, low-cost and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer.
2. Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections.
3. Enhance user convenience through a “plug-and-play” mode of operation

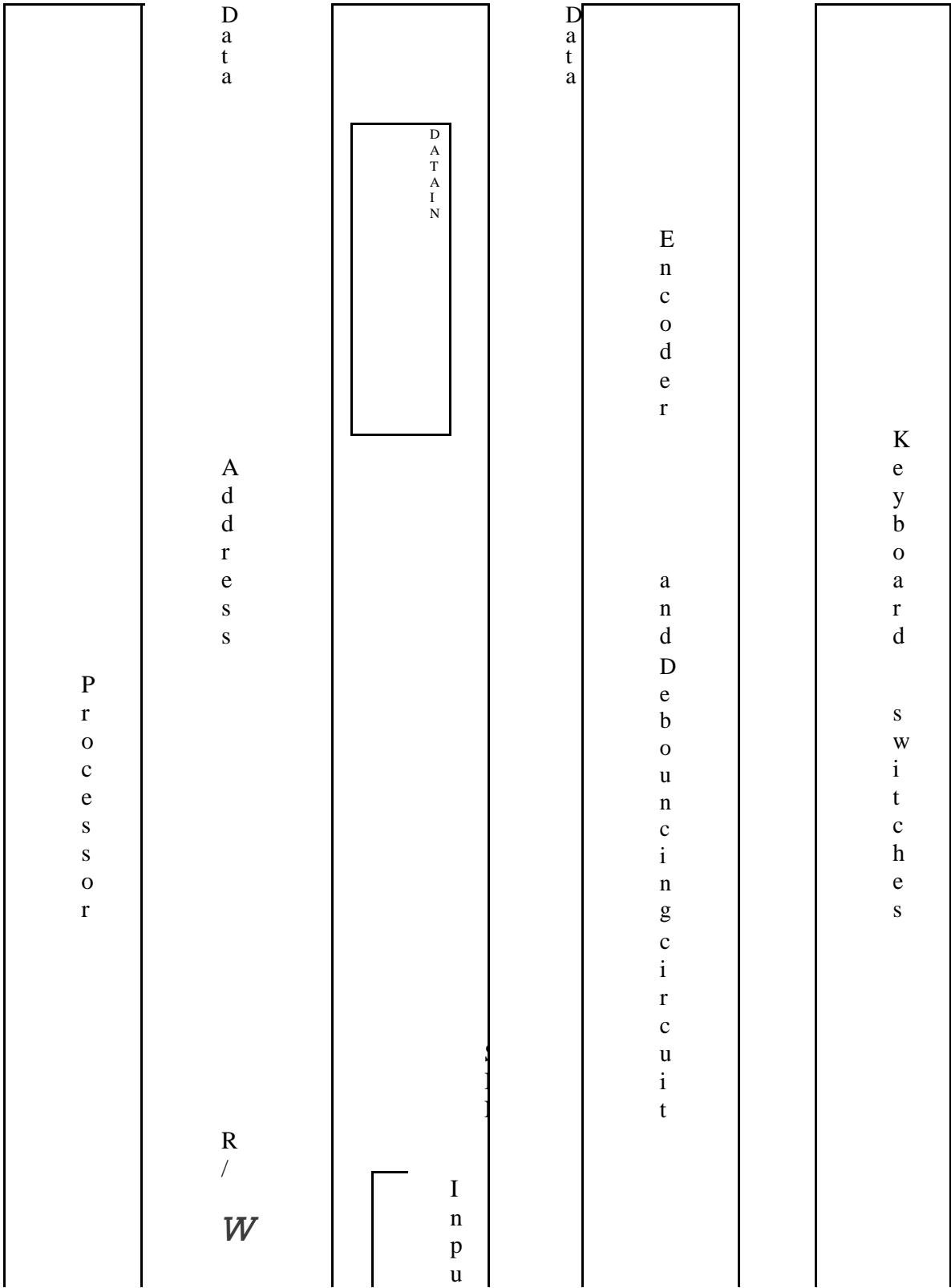
5.7.INTERFACE CIRCUITS

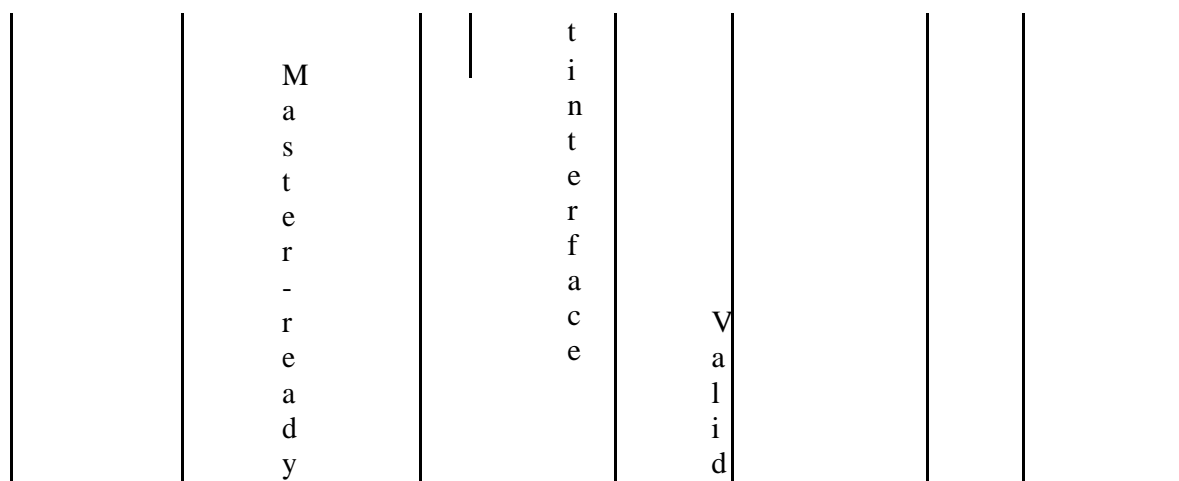
Parallel port

The hardware components needed for connecting a keyboard to a processor. 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.

Figure 11 Keyboard to processor connection.





Slave-ready

The output of the encoder consists of the bits that represent the encoded character and one control signal called Valid, which indicates that a key is being pressed. This information is sent to the interface circuit, which contains a data register, DATAIN, and a status flag, SIN. When a key is pressed, the Valid signal changes from 0 to 1, causing the ASCII code to be loaded into DATAIN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register. The interface circuit is connected to an asynchronous bus on which transfers are controlled using the handshake signals Master-ready and Slave-ready, as indicated in

figure 11. The third control line, R/*W* distinguishes read and write transfers.

Figure 12 shows a suitable circuit for an input interface. The output lines of the DATAIN register are connected to the data lines of the bus by means of three-state drivers, which are turned on when the processor issues a read instruction with the address that selects this register. The SIN signal is generated by a status flag circuit. This signal is also sent to the bus through a three-state driver. It is connected to bit D0, which means it will appear as bit 0 of the status register. Other bits of this register do not contain valid information. An address decoder is used to select the input interface when the high-order 31 bits of an address correspond to any of the addresses assigned to this interface. Address bit A0 determines whether the status or the data registers is to be read when the Master-ready signal is active. The control handshake is accomplished by activating the Slave-ready signal when either Read-status or Read-data is equal to 1.

Fig Printer to processor connection

Let us now consider an output interface that can be used to connect an output device, such as a printer, to a processor, as shown in figure 13. The printer operates under control of the handshake signals Valid and Idle in a manner similar to the handshake used on the bus with the Master-ready and Slave-ready signals. When it is ready to accept a character, the printer asserts its Idle signal. The interface circuit can then place a new character on the data lines and activate the Valid signal. In response, the printer starts printing the new character and negates the Idle signal, which in turn causes the interface to deactivate the Valid signal.

The circuit in figure 16 has separate input and output data lines for connection to an I/O device. A more flexible parallel port is created if the data lines to I/O devices are bidirectional. Figure 17 shows a general-purpose parallel interface circuit that can be configured in a variety of ways. Data lines P7 through P0 can be used for either input or output purposes. For increased flexibility, the circuit makes it possible for some lines to serve as inputs and some lines to serve as outputs, under program control. The DATAOUT register is connected to these lines via three-state drivers that are controlled by a data direction register, DDR. The processor can write any 8-bit pattern into DDR. For a given bit, if the DDR value is 1, the corresponding data line acts as an output line; otherwise, it acts as an input line.

5.8. STANDARD I/O INTERFACES

The processor bus is the bus defined by the signals on the processor chip itself. Devices that require a very high-speed connection to the processor, such as the main memory, may be connected directly to this bus. For electrical reasons, only a few devices can

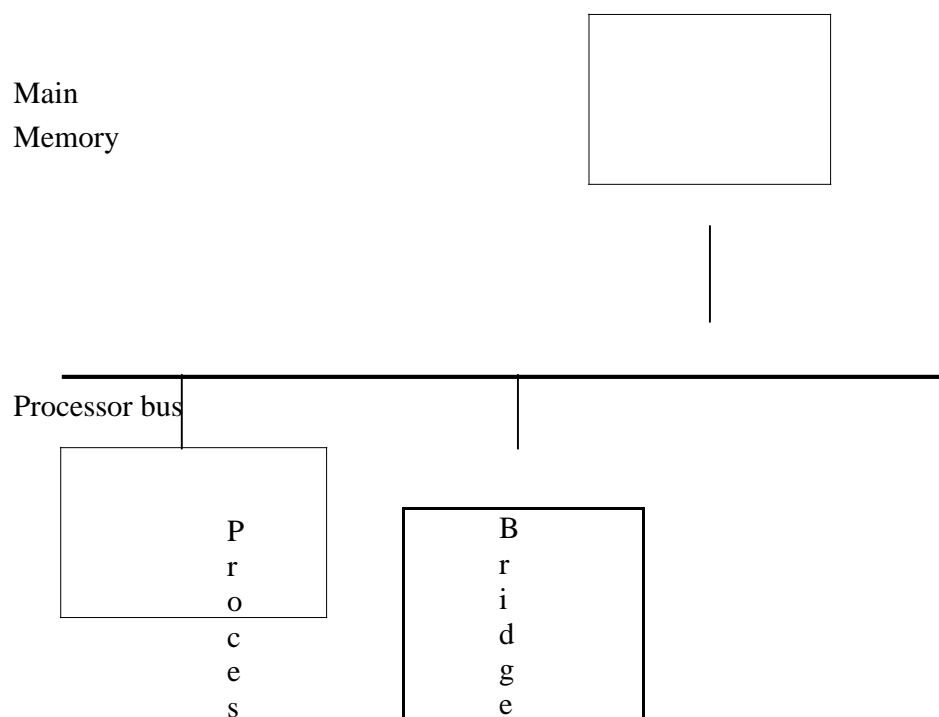
be connected in this manner. The motherboard usually provides another bus that can support more devices. The two buses are interconnected by a circuit, which we will call a bridge, that translates the signals and protocols of one bus into those of the other. Devices connected to the expansion bus appear to the processor as if they were connected directly to the processor's own bus. The only difference is that the bridge circuit introduces a small delay in data transfers between the processor and those devices.

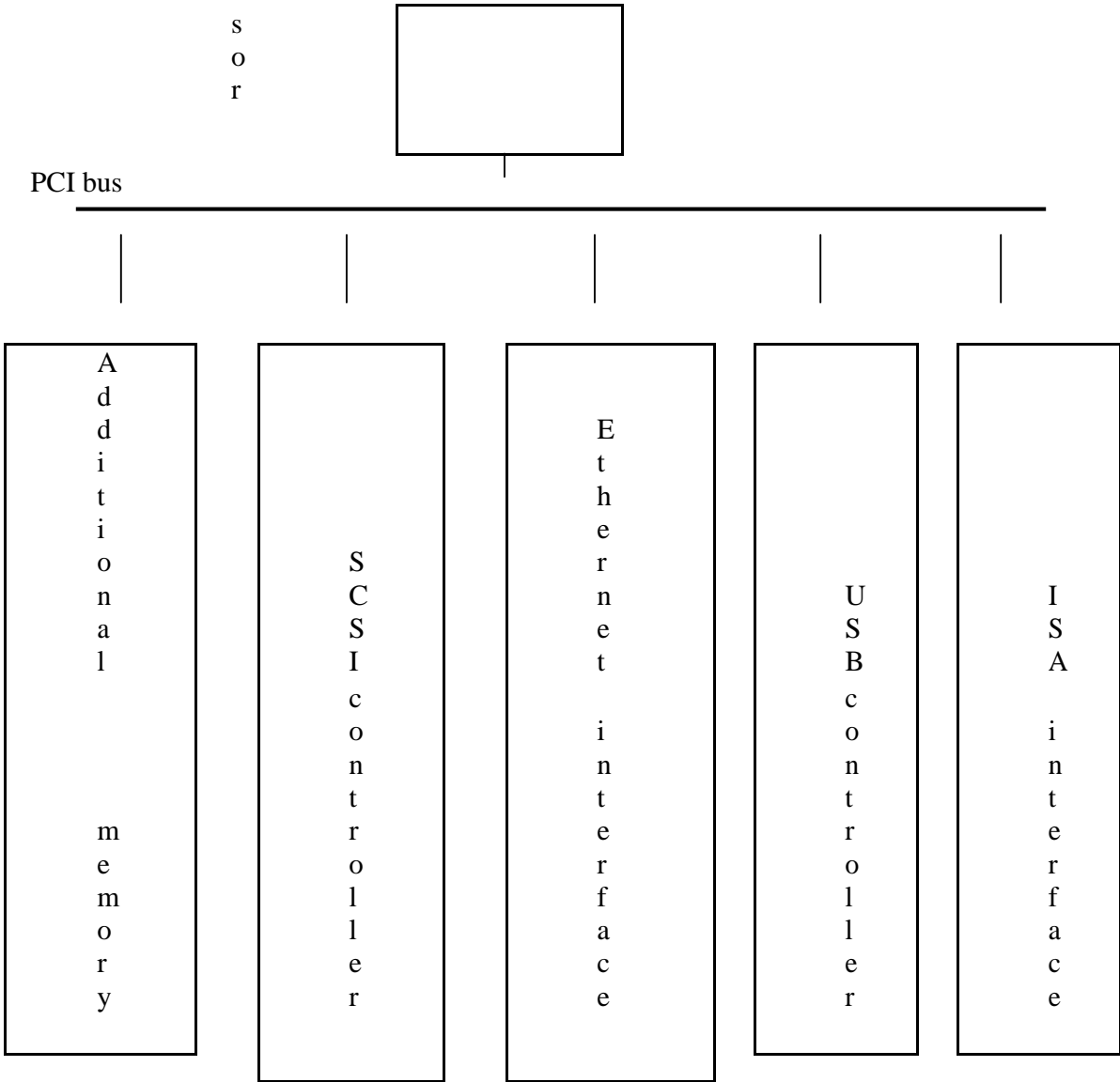
It is not possible to define a uniform standard for the processor bus. The structure of this bus is closely tied to the architecture of the processor. It is also dependent on the electrical characteristics of the processor chip, such as its clock speed. The expansion bus is not subject to these limitations, and therefore it can use a standardized signaling scheme. A number of standards have been developed. Some have evolved by default, when a particular design became commercially successful. For example, IBM developed a bus they called ISA (Industry Standard Architecture) for their personal computer known at the time as PC AT.

Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), or international bodies such as ISO (International Standards Organization) have blessed these standards and given them an official status.

A given computer may use more than one bus standards. A typical Pentium computer has both a PCI bus and an ISA bus, thus providing the user with a wide range of devices to choose from.

Figure 21 An example of a computer system using different interface standards





Port Limitation:-

The parallel and serial ports described in previous section provide a general-purpose point of connection through which a variety of low-to medium-speed devices can be connected to a computer. For practical reasons, only a few such ports are provided in a typical computer.

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 such devices vary significantly.

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 the controls and manipulators used with video games are good examples.

Plug-and-Play:-

As computers become part of everyday life, their existence should become increasingly transparent. For example, when operating a home theater system, which includes at least one computer, the user should not find it necessary to turn the computer off or to restart the system to connect or disconnect a device.

The plug-and-play feature means that a new device, such as an additional speaker, can be connected at any time while the system is operating. The system should detect the existence of this new device automatically, identify the appropriate device-driver software and any other facilities needed to service that device, and establish the appropriate addresses and logical connections to enable them to communicate. The plug-and-play requirement has many implications at all levels in the system, from the hardware to the operating system and the applications software. One of the primary objectives of the design of the USB has been to provide a plug-and-play capability.