RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus

III Year - VI Semester: B.Tech. (Electrical Engineering)

6EE3-01: COMPUTER ARCHITECTURE

Credit: 2 **Max. Marks: 100(IA:20, ETE:80)**

2L+0T+0P **End Term Exam: 2 Hours**

SN	CONTENTS	HOURS
1	**Introduction:** Objective, scope and outcome of the course.	01
2	**Introduction to computer organization** Architecture and function of general computer system, CISC Vs RISC, Data types, Integer Arithmetic - Multiplication, Division, Fixed and Floating point representation and arithmetic, Control unit operation, Hardware implementation of CPU with Micro instruction, microprogramming, System buses, Multi-bus organisation	05
3	**Memory organization** System memory, Cache memory - types and organization, Virtual memory and its implementation, Memory management unit, Magnetic Hard disks, Optical Disks	04
4	**Input – output Organization** Accessing I/O devices, Direct Memory Access and DMA controller, Interrupts and Interrupt Controllers, Arbitration, Multilevel Bus Architecture, Interface circuits - Parallel and serial port. Features of PCI and PCI Express bus.	05
5	**16 and 32 microprocessors** 80x86 Architecture, IA – 32 and IA – 64, Programming model, Concurrent operation of EU and BIU, Real mode addressing, Segmentation, Addressing modes of 80x86, Instruction set of 80x86, I/O addressing in 80x86	05
6	**Pipelining** Introduction to pipelining, Instruction level pipelining (ILP), compiler techniques for ILP, Data hazards, Dynamic scheduling, Dependability, Branch cost, Branch Prediction, Influence on instruction set	04
7	**Different Architectures** VLIW Architecture, DSP Architecture, SoC architecture, MIPS Processor and programming	04
	TOTAL	**28**

Office of Dean Academic Affairs
Rajasthan Technical University, Kota

Introduction

Computer Architecture:

Computer Architecture deals with giving operational attributes of the computer or Processor to be specific. It deals with details like physical memory, ISA (Instruction Set Architecture) of the processor, the number of bits used to represent the data types, Input Output mechanism and technique for addressing memories.

Computer Organization:

Computer Organization is realization of what is specified by the computer architecture .It deals with how operational attributes are linked together to meet the requirements specified by computer architecture. Some organizational attributes are hardware details, control signals, peripherals.

Computer Organization	Computer Architecture
Often called microarchitecture (low level)	Computer architecture (a bit higher level)
Transparent from programmer (ex. a programmer doesnot worry much how addition is implemented in hardware)	Programmer view (i.e. Programmer has to be aware of which instructionset used)
Physical components (Circuit design, Adders, Signals, Peripherals)	Logic (Instruction set, Addressing modes, Data types, Cache optimization)
How to do ? (implementation of the architecture)	What to do ? (Instruction set)

BASIC TERMINOLOGY

•Input: Whatever is put into a computer system?

•Data: Refers to the symbols that represent facts, objects, or ideas.

•Information: The results of the computer storing data as bits and bytes; the words, umbers,sounds, and graphics.

•Output: Consists of the processing results produced by a computer.

•Processing: Manipulation of the data in many ways.

•Memory: Area of the computer that temporarily holds data waiting to be processed, stored,or output.

•Storage: Area of the computer that holds data on a permanent basis when it is notimmediately needed for processing.

•Assembly language program (ALP) –Programs are written using mnemonics

•Mnemonic –Instruction will be in the form of English like form

•Assembler –is a software which converts ALP to MLL (Machine Level Language)

•HLL (High Level Language) –Programs are written using English like statements

•Compiler -Convert HLL to MLL, does this job by reading source program at once

•Interpreter –Converts HLL to MLL, does this job statement by statement

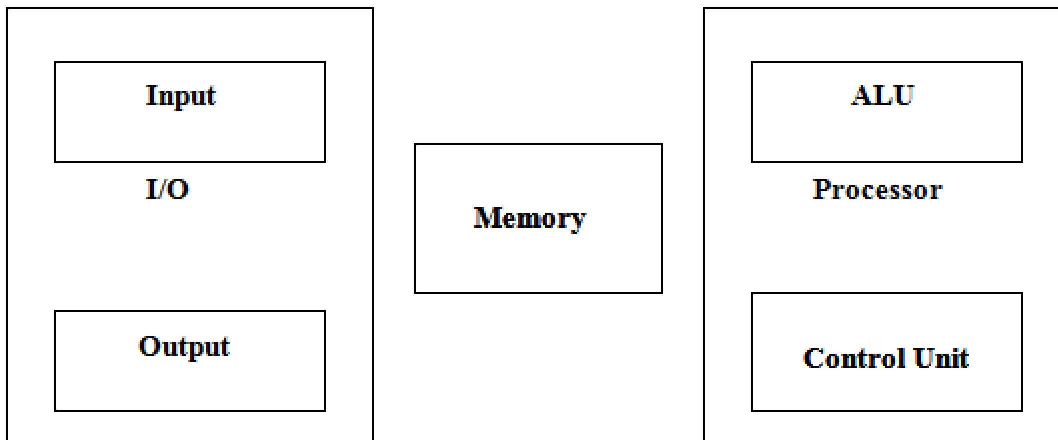
•System software –Program routines which aid the user in the execution of programs eg: Assemblers, Compilers

•Operating system –Collection of routines responsible for controlling and coordinating allthe activities in a computer system

Architecture and function of general computer system

Functional Unit

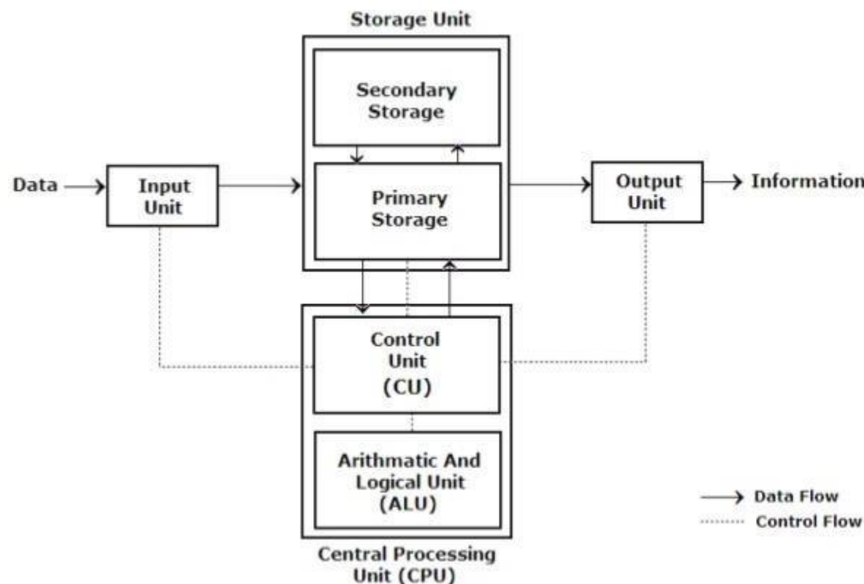
A computer consists of five functionally independent main parts input, memory, arithmetic logic unit (ALU), output unit and control unit.



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

1. **Primary memory**
2. **Secondary memory**

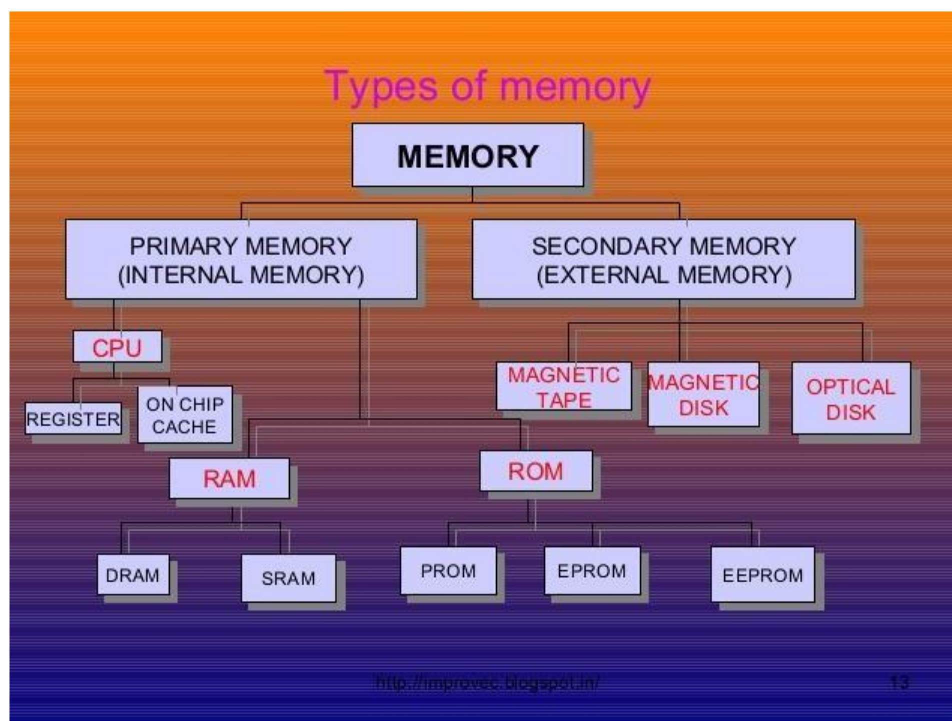
Word:

In computer architecture, a word is a unit of data of a defined bit length that can be addressed and moved between storage and the computer processor. Usually, the defined bit length of a word is equivalent to the width of the computer's data bus so that a word can be moved in a single operation from storage to a processor register. For any computer architecture with an eight-bit byte, the word

will be some multiple of eight bits. In IBM's evolutionary System/360 architecture, a word is 32 bits, or four contiguous eight-bit bytes. In Intel's PC processor architecture, a word is 16 bits, or two contiguous eight-bit bytes. A word cancontain a computer instruction, a storage address, or application data that is to be manipulated (for example, added to the data in another word space).

The number of bits in each word is known as word length. Word length refers to the number of bits processed by the CPU in one go. With modern general purpose computers, word size can be 16 **bits** to **64 bits**.

The time required to access one word is called the memory access time. The small, fast, RAM units are called caches. They are tightly coupled with the processor and are often contained on the same IC chip to achieve high performance.



1. **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 semiconductors storage cells. Each capable of storing one bit of information. These are processed in a group of fixed site 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 in 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 storageis 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 may 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.

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,

R1Add 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.

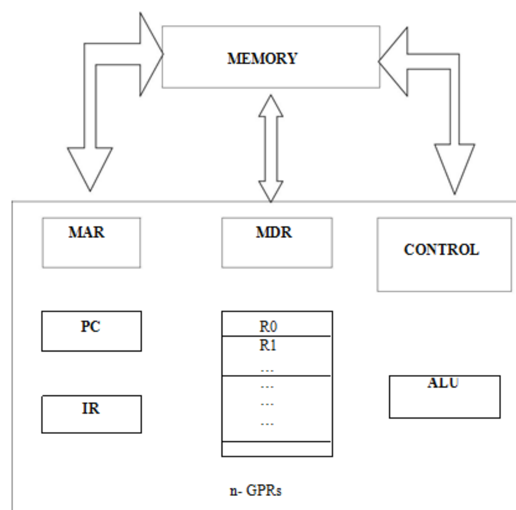


Fig b : Connections between the processor and the memory

The fig shows how memory & the processor can be connected. In addition to the ALU &

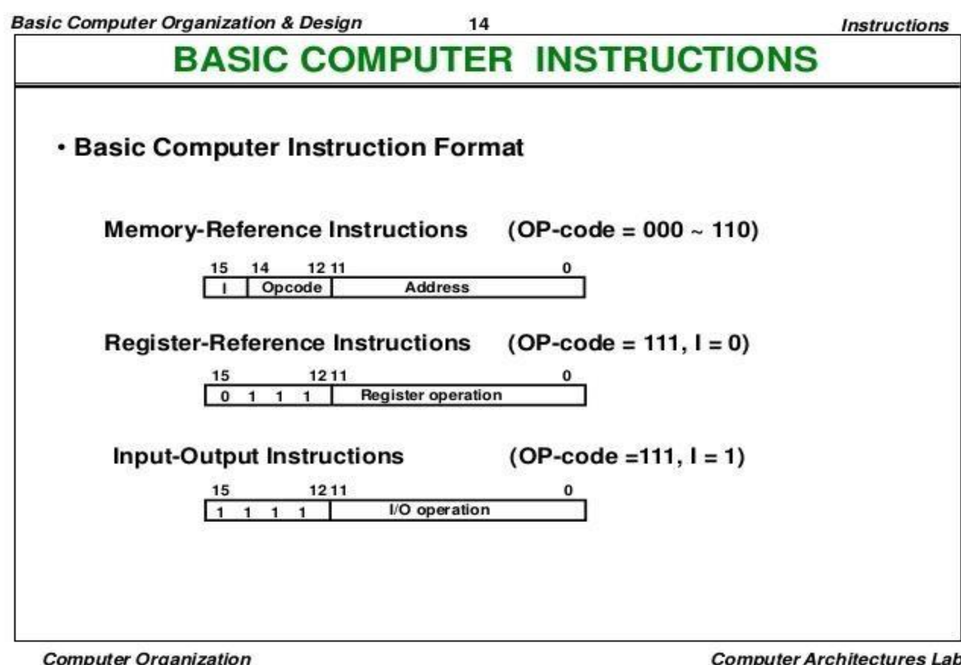
the control circuitry, the processor contains a number of registers used for several different purposes.

Register:

It is a special, high-speed storage area within the CPU. All data must be represented in a register before it can be processed. For example, if two numbers are to be multiplied, both numbers must be in registers, and the result is also placed in a register. (The register can contain the address of a memory location where data is stored rather than the actual data itself.)

The number of registers that a CPU has and the size of each (number of bits) help determine the power and speed of a CPU. For example a 32-bit CPU is one in which each register is 32 bits wide. Therefore, each CPU instruction can manipulate 32 bits of data. In high-level languages, the compiler is responsible for translating high-level operations into low-level operations that access registers.

Instruction Format:



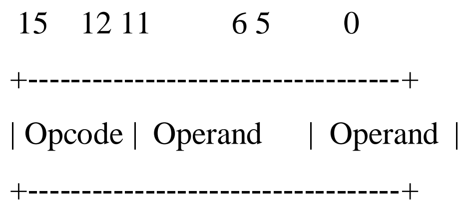
Computer instructions are the basic components of a machine language program. They are also known as *macro operations*, since each one is comprised of sequences of micro operations.

Each instruction initiates a sequence of micro operations that fetch operands from registers or memory, possibly perform arithmetic, logic, or shift operations, and store results in registers or memory.

Instructions are encoded as binary *instruction codes*. Each instruction code contains of a

operation code, or *opcode*, which designates the overall purpose of the instruction (e.g. add, subtract, move, input, etc.). The number of bits allocated for the opcode determined how many different instructions the architecture supports.

In addition to the opcode, many instructions also contain one or more *operands*, which indicate where in registers or memory the data required for the operation is located. For example, and add instruction requires two operands, and a not instruction requires one.

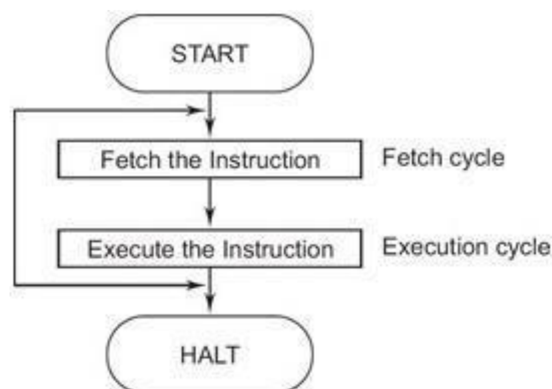


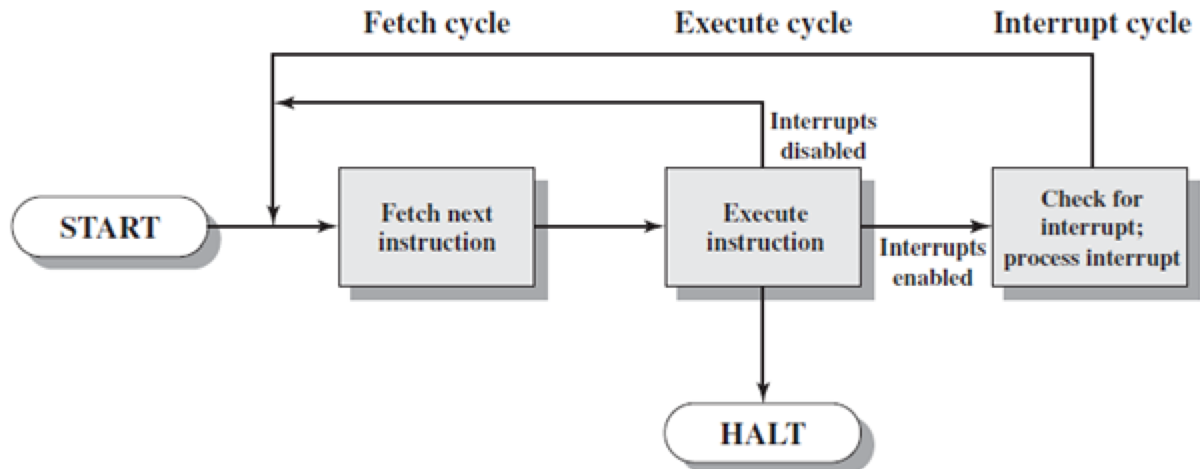
```
 15    12 11         6 5        0
+-----------------------------------+
| Opcode |  Operand     |  Operand  |
+-----------------------------------+
```

The opcode and operands are most often encoded as unsigned binary numbers in order to minimize the number of bits used to store them. For example, a 4-bit opcode encoded as a binary number could represent up to 16 different operations.

The *control unit* is responsible for decoding the opcode and operand bits in the instruction register, and then generating the control signals necessary to drive all other hardware in the CPU to perform the sequence of micro operations that comprise the instruction.

INSTRUCTION CYCLE:





Instruction Cycle with Interrupts

The instruction register (IR):- Holds the instructions that are 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.

Besides IR and PC, there are n-general purpose registers R0 through R_{n-1}.

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. After the time required to access the memory elapses, the address word is read out ofthe memory and loaded into the MDR.
4. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
5. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
6. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
7. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
8. After one or two such repeated cycles, the ALU can perform the desired operation.
9. If the result of this operation is to be stored in the memory, the result is sent to MDR.
10. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
11. 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

THE VON NEUMANN ARCHITECTURE

The task of entering and altering programs for the ENIAC was extremely tedious. The programming process can be easy if the program could be represented in a form suitable for storing in memory alongside the data. Then, a computer could get its instructions by reading them from memory, and a program could be set or altered by setting the values of a portion of memory. This

idea is known a the stored-program concept. The first publication of the idea

was in a 1945 proposal by von Neumann for a new computer, the EDVAC (Electronic Discrete Variable Computer).

In 1946, von Neumann and his colleagues began the design of a new stored-program computer, referred to as the IAS computer, at the Princeton Institute for Advanced Studies. The IAS computer, although not completed until 1952, is the prototype of all subsequent general-purpose computers.

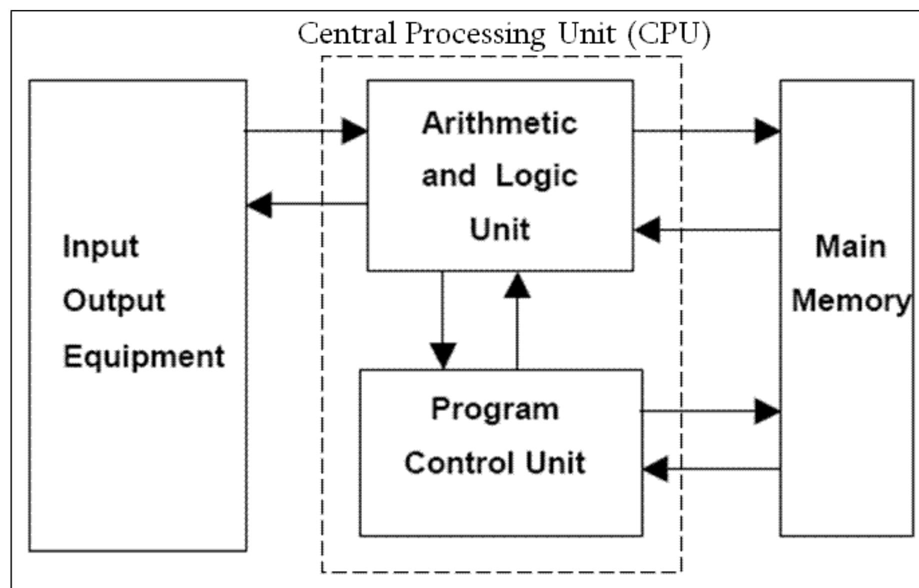


Figure : General structure of Von Neumann Architecture

It consists of

- A main memory, which stores both data and instruction
- An arithmetic and logic unit (ALU) capable of operating on binary data
- A control unit, which interprets the instructions in memory and causes them to be executed
- Input and output (I/O) equipment operated by the control unit

BUS STRUCTURES:

Bus structure and multiple bus structures are types of bus or computing. A bus is basically a subsystem which transfers data between the components of Computer components either within a computer or between two computers. It connects peripheral devices at the same time.

- A multiple Bus Structure has multiple inter connected service integration buses and for each bus the other buses are its foreign buses. A Single bus structure is very simple and consists of a single server.

- A bus cannot span multiple cells. And each cell can have more than one buses. - Published messages are printed on it. There is no messaging engine on Single bus structure

I) In single bus structure all units are connected in the same bus than connecting different buses as multiple bus structure.

II) Multiple bus structure's performance is better than single bus structure. Iii)single bus structure's cost is cheap than multiple bus structure.

Group of lines that serve as connecting path for several devices is called a bus (one bit perline). Individual parts must communicate over a communication line or path for exchanging data, address and control information as shown in the diagram below. Printer example – processor to printer. A common approach is to use the concept of buffer registers to hold the content during the transfer.

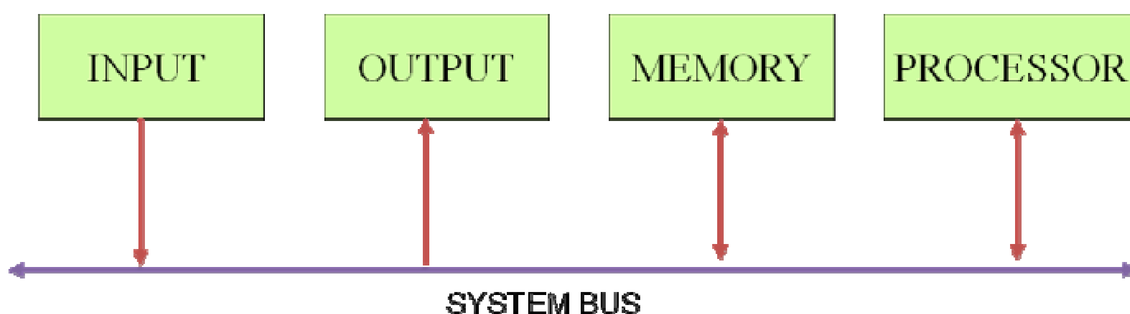


Figure 5: Single bus structure

Buffer registers hold the data during the data transfer temporarily. Ex: printing

Types of Buses:

1. Data Bus:

Data bus is the most common type of bus. It is used to transfer data between different components of computer. The number of lines in data bus affects the speed of data transfer between different components. The data bus consists of 8, 16, 32, or 64 lines. A 64-line data bus can transfer 64 bits of data at one time.

The data bus lines are bi-directional. It means that:

CPU can read data from memory using these lines CPU can write data to memory locations using these lines

2. Address Bus:

Many components are connected to one another through buses. Each component is assigned a unique ID. This ID is called the address of that component. It a component wants to communicate with another component, it uses address bus to specify the address of that component. The address bus is a unidirectional bus. It can carry information only in one direction. It carries address of memory location from microprocessor to the main memory.

3. Control Bus:

Control bus is used to transmit different commands or control signals from one component to another component. Suppose CPU wants to read data from main memory. It will use control is also used to transmit control signals like ASKS (Acknowledgement signals). A control signal contains the following:

1 Timing information: It specifies the time for which a device can use data and address bus.

2 Command Signal: It specifies the type of operation to be performed. Suppose that CPU gives a command to the main memory to write data. The memory sends acknowledgement signal to CPU after writing the data successfully. CPU receives the signal and then moves to perform some other action.

SOFTWARE

If a user wants to enter and run an application program, he/she needs a System Software. System Software is a collection of programs that are executed as needed to perform functions such as:

• Receiving and interpreting user commands

• Entering and editing application programs and storing then as files in secondary storage devices

• Running standard application programs such as word processors, spread sheets, gamesetc…

Operating system - is key system software component which helps the user to exploit thebelow underlying hardware with the programs.Types of software

A layer structure showing where Operating System is located on generally used software systems on desktops

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.

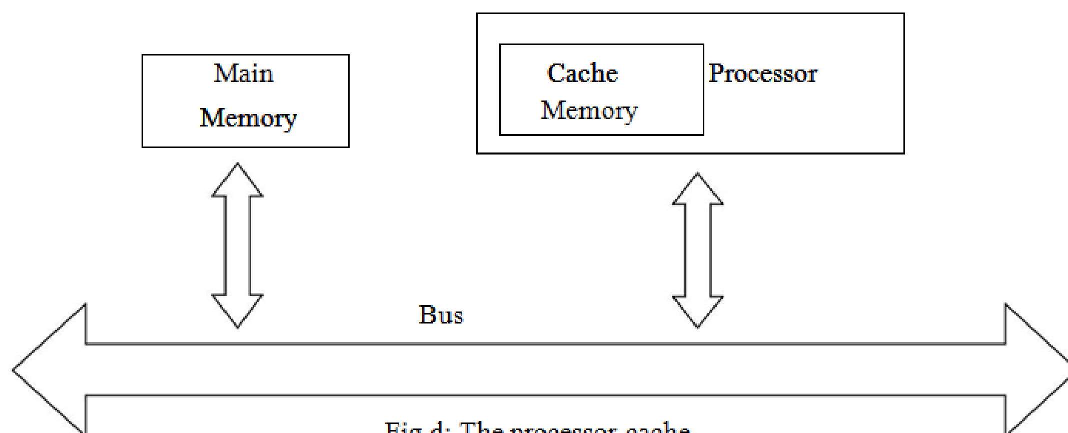


Fig d: The processor cache

The pertinent parts of the fig. c are repeated in fig. d which includes the cache memory as part of the processor unit.

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 thatrange from a few hundred million to over a billion cycles per second.

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 thatcorresponds 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 numberof 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=N*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 R1 & R2 and places the sum into R3.

The contents of R1 & R2 are first transferred to the inputs of ALU. After the addition operation is performed, the sum is transferred to R3. 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 R3.

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 days many processors are designed in this manner.

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 excepted from the use of faster technology can be realized.

CISC Vs RISC

Instruction set CISC & RISC:-

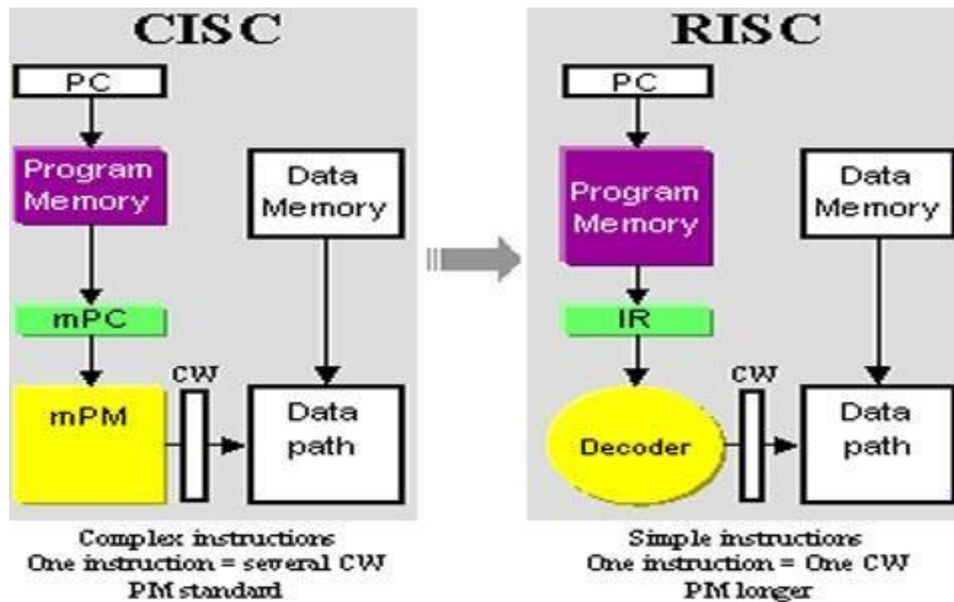
Simple instructions require a small number of basic steps to execute. Complex instructions involve a large number of steps. For a processor that has only simple instructiona large number of instructions may be needed to perform a given programming task. This could lead to a large value of 'N' and a small value of 'S' on the other hand if individual instructions perform more complex operations, a fewer instructions will be needed, leadingto 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 ¿ 1) would achieve one best performance. However, it is much easier to implement efficient pipelining in processors with simple instruction sets.RISC and CISC are computing systems developed for computers. Instruction set or instruction set architecture is the structure of the computer that provides commands to the computer to guide the computer for processing data manipulation. Instruction set consists of instructions, addressing modes, native data types, registers, interrupt, exception handling and memory architecture. Instruction set can be emulated in software by using an interpreter or built into hardware of the processor. Instruction Set Architecture can be considered as a boundary between the software and hardware. Classification of microcontrollers and microprocessors can be done based on the RISC and CISC instruction set architecture.

Comparison between RISC and CISC:

	RISC	**CISC**
Acronym	It stands for 'Reduced Instruction Set Computer'.	It stands for 'Complex Instruction Set Computer'.
Definition	The RISC processors have a smaller set of instructions with few addressing nodes.	The CISC processors have a larger set of instructions with many addressing nodes.
Memory unit	It has no memory unit and uses a separate hardware to implement instructions.	It has a memory unit to implement complex instructions.
Program	It has a hard-wired unit of programming.	It has a micro-programming unit.
Design	It is a complex complier design.	It is an easy complier design.

Calculations	The calculations are faster and precise.	The calculations are slow and precise.
Decoding	Decoding of instructions is simple.	Decoding of instructions is complex.
Time	Execution time is very less.	Execution time is very high.
External memory	It does not require external memory for calculations.	It requires external memory for calculations.
Pipelining	Pipelining does function correctly.	Pipelining does not function correctly.
Stalling	Stalling is mostly reduced in processors.	The processors often stall.
Code expansion	Code expansion can be a problem.	Code expansion is not a problem.
Disc space	The space is saved.	The space is wasted.
Applications	Used in high end applications such as video processing, telecommunications and image processing.	Used in low end applications such as security systems, home automations, etc.

CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy



1.8 Performance measurements

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.

SPEC rating = Running time on the reference computer/ Running time on the computer undertest

Data Types

Data Representation:
Registers are made up of flip-flops and flip-flops are two-state devices that can store only 1'sand 0's.

Numbering Systems		
System	**Base**	**Digits**
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

There are many methods or techniques which can be used to convert numbers from onebase to another. We'll demonstrate here the following –
Decimal to Other Base System
Other Base System to Decimal
Other Base System to Non-Decimal
Shortcut method – Binary to Octal
Shortcut method – Octal to Binary
Shortcut method – Binary to Hexadecimal
Shortcut method – Hexadecimal to BinaryDecimal to Other Base System
Steps
Step 1 – Divide the decimal number to be converted by the value of the new base.
Step 2 – Get the remainder from Step 1 as the rightmost digit (least significant digit)of new base number.
Step 3 – Divide the quotient of the previous divide by the new base.
Step 4 – Record the remainder from Step 3 as the next digit (to the left) of the new base number.
Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.
The last remainder thus obtained will be the Most Significant Digit (MSD) of the new basenumber.
Example –
Decimal Number: 29_{10}
Calculating Binary Equivalent –

Step	Operation	Result	Remainder
Step 1	29 / 2	14	1
Step 2	14 / 2	7	0
Step 3	7 / 2	3	1
Step 4	3 / 2	1	1
Step 5	1 / 2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that

the first remainder becomes the Least Significant Digit (LSD) and the last remainder becomes the Most Significant Digit (MSD).

Decimal Number − 29_{10} = Binary Number − 11101_2.Other Base System to Decimal System

Steps

Step 1 − Determine the column (positional) value of each digit (this depends on the position of the digit and the base of the number system).

Step 2 − Multiply the obtained column values (in Step 1) by the digits in the corresponding columns.

Step 3 − Sum the products calculated in Step 2. The total is the equivalent value in decimal.

Step	Binary Number	Decimal Number
Step 1	11101_2	$((1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	11101_2	$(16 + 8 + 4 + 0 + 1)_{10}$
Step 3	11101_2	29_{10}

Example

Binary Number − 11101_2 Calculating Decimal Equivalent −

Binary Number − 11101_2 = Decimal Number − 29_{10} Other Base System to Non-Decimal System

Steps

Step 1 − Convert the original number to a decimal number (base 10).

Step 2 − Convert the decimal number so obtained to the new base number.Example

Octal Number − 25_8

Calculating Binary Equivalent −Step 1 − Convert to Decimal

Step	Octal Number	Decimal Number
Step 1	25_8	$((2 \times 8^1) + (5 \times 8^0))_{10}$
Step 2	25_8	$(16 + 5)_{10}$
Step 3	25_8	21_{10}

Octal Number − 25_8 = Decimal Number − 21_{10}

Step 2 − Convert Decimal to Binary

Step	Operation	Result	Remainder
Step 1	21 / 2	10	1
Step 2	10 / 2	5	0
Step 3	5 / 2	2	1
Step 4	2 / 2	1	0
Step 5	1 / 2	0	1

Decimal Number − 21_{10} = Binary Number − 10101_2

Octal Number − 25_8 = Binary Number − 10101_2

Shortcut method - Binary to Octal

Steps

- **Step 1** − Divide the binary digits into groups of three (starting from the right).
- **Step 2** − Convert each group of three binary digits to one octal digit.

Example

Binary Number − 101012

Calculating Octal Equivalent −

Step	Binary Number	Octal Number
Step 1	101012	010 101
Step 2	101012	28 58
Step 3	101012	258

Binary Number − 101012 = Octal Number − 258

Shortcut method - Octal to Binary

Steps

- **Step 1** − Convert each octal digit to a 3 digit binary number (the octal digits may be treated as decimal for this conversion).
- **Step 2** − Combine all the resulting binary groups (of 3 digits each) into a single binary number.

Example Octal Number − 258

Calculating Binary Equivalent −

Step	Octal Number	Binary Number
Step 1	258	210 510
Step 2	258	0102 1012
Step 3	258	0101012

Octal Number − 258 = Binary Number − 101012

Shortcut method - Binary to Hexadecimal

Steps

- **Step 1** − Divide the binary digits into groups of four (starting from the right).
- **Step 2** − Convert each group of four binary digits to one hexadecimal symbol.

Example Binary Number − 101012

Calculating hexadecimal Equivalent −

Step	Binary Number	Hexadecimal Number
Step 1	101012	0001 0101
Step 2	101012	110 510
Step 3	101012	1516

Binary Number − 101012 = Hexadecimal Number − 1516

Shortcut method - Hexadecimal to Binary

Steps

- **Step 1** − Convert each hexadecimal digit to a 4 digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
- **Step 2** − Combine all the resulting binary groups (of 4 digits each) into a single binary number.

Example

Hexadecimal Number − 1516

Calculating Binary Equivalent −

Step	Hexadecimal Number	Binary Number
Step 1	1516	110 510
Step 2	1516	00012 01012
Step 3	1516	000101012

Hexadecimal Number − 1516 = Binary Number − 101012

Binary Coded Decimal (BCD) code

In this code each decimal digit is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only first ten of these are used (0000 to 1001). The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.

Decimal	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Advantages of BCD Codes

- It is very similar to decimal system.
- We need to remember binary equivalent of decimal numbers 0 to 9 only.

Disadvantages of BCD Codes

- The addition and subtraction of BCD have different rules.
- The BCD arithmetic is little more complicated.
- BCD needs more number of bits than binary to represent the decimal number. So BCD is

less efficient than binary.

Alphanumeric codes

A binary digit or bit can represent only two symbols as it has only two states '0' or '1'. But this is not enough for communication between two computers because there we need many more symbols for communication. These symbols are required to represent 26 alphabets with capital and small letters, numbers from 0 to 9, punctuation marks and other symbols.

The alphanumeric codes are the codes that represent numbers and alphabetic characters. Mostly such codes also represent other characters such as symbol and various instructions necessary for conveying information. An alphanumeric code should at least represent 10 digits and 26 letters of alphabet i.e. total 36 items. The following three alphanumeric codes are very commonly used for the data representation.

- American Standard Code for Information Interchange (ASCII).
- Extended Binary Coded Decimal Interchange Code (EBCDIC).
- Five bit Baudot Code.

ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers.

Computer Arithmetic-Multiplication, Division

Complements are used in the digital computers in order to simplify the subtraction operation and for the logical manipulations. For each radix-r system (radix r represents base of number system) there are two types of complements.

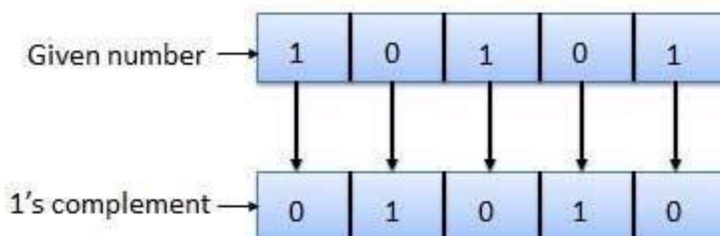
S.N.	Complement	Description
1	Radix Complement	The radix complement is referred to as the r'scomplement
2	Diminished Radix Complement	The diminished radix complement is referred to as the (r-1)'s complement

Binary system complements

As the binary system has base r = 2. So the two types of complements for the binary system are 2's complement and 1's complement.

1's complement

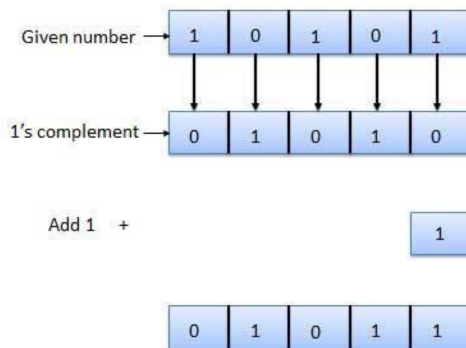
The 1's complement of a number is found by changing all 1's to 0's and all 0's to 1's. This is called as taking complement or 1's complement. Example of 1's Complement is as follows.



2's complement

The 2's complement of binary number is obtained by adding 1 to the Least Significant Bit (LSB) of 1's complement of the number.

2's complement = 1's complement + 1 Example of 2's Complement is as follows.



Binary Arithmetic

Binary arithmetic is essential part of all the digital computers and many other digital system.

Binary Addition

It is a key for binary subtraction, multiplication, division. There are four rules of binary addition.

Case	A + B	Sum	Carry
1	0 + 0	0	0
2	0 + 1	1	0
3	1 + 0	1	0
4	1 + 1	0	1

In fourth case, a binary addition is creating a sum of (1 + 1 = 10) i.e. 0 is written in the given column and a carry of 1 over to the next column.

Example – Addition

0011010 + 001100 = 00100110

```
  1 1        carry
 0 0 1 1 0 1 0   = 26₁₀
+0 0 0 1 1 0 0   = 12₁₀
______________
 0 1 0 0 1 1 0   = 38₁₀
```

Binary Subtraction

Subtraction and Borrow, these two words will be used very frequently for the binary subtraction. There are four rules of binary subtraction.

Case	A - B	Subtract	Borrow
1	0 - 0	0	0
2	1 - 0	1	0
3	1 - 1	0	0
4	0 - 1	0	1

Example – Subtraction

0011010 - 001100 = 00001110

```
      1 1      borrow
 0 0 1 1 0 1 0   = 26₁₀
-0 0 0 1 1 0 0   = 12₁₀
______________
 0 0 0 1 1 1 0   = 14₁₀
```

Binary Multiplication

Binary multiplication is similar to decimal multiplication. It is simpler than decimal multiplication because only 0s and 1s are involved. There are four rules of binary multiplication.

Case	A x B	Multiplication
1	0 x 0	0
2	0 x 1	0
3	1 x 0	0
4	1 x 1	1

Example − Multiplication

Example:

0011010 x 001100 = 100111000

```
      0011010   = 26₁₀
     x0001100   = 12₁₀
   -----------
      0000000
     0000000
    0011010
   0011010
   -----------
   0100111000   = 312₁₀
```

Binary Division

Binary division is similar to decimal division. It is called as the long division procedure.

Example − Division

101010 / 000110 = 000111

```
              111     = 7₁₀
000110 ) 101010      = 42₁₀
         -110         = 6₁₀
          ----
          1001
          -110
          ----
            110
           -110
           ----
              0
```

Subtraction by 1's Complement

In subtraction by 1's complement we subtract two binary numbers using carried by 1's complement.

The steps to be followed in subtraction by 1's complement are:

i) To write down 1's complement of the subtrahend.

ii) To add this with the minuend.

iii) If the result of addition has a carry over then it is dropped and an 1 is added in the last bit.

iv) If there is no carry over, then 1's complement of the result of addition is obtained to get the final result and it is negative.

Evaluate:

(i) 110101 – 100101

Solution:

1's complement of 10011 is 011010. Hence

Minued -		1 1 0 1 0 1
1's complement of subtrahend -		0 1 1 0 1 0
Carry over -	1	0 0 1 1 1 1
		1
		0 1 0 0 0 0

The required difference is 10000

(ii) 101011 – 111001

Solution:

1's complement of 111001 is 000110. Hence

Minued -	1 0 1 0 1 1
1's complement -	0 0 0 1 1 0
	1 1 0 0 0 1

Hence the difference is – 1 1 1 0

(iii) 1011.001 – 110.10

Solution:

1's complement of 0110.100 is 1001.011 Hence

Minued - 1 0 1 1 . 0 0 1

1's complement of subtrahend - 1 0 0 1 . 0 1 1

Carry over - 1 0 1 0 0 . 1 0 0

1

0 1 0 0 . 1 0 1

Hence the required difference is 100.101

(iv) 10110.01 – 11010.10

Solution:

1's complement of 11010.10 is 00101.01

1 0 1 1 0 . 0 1

0 0 1 0 1 . 0 1

1 1 0 1 1 . 1 0

Hence the required difference is – 00100.01 i.e. – 100.01

Subtraction by 2's Complement

With the help of subtraction by 2's complement method we can easily subtract two binary numbers.

The operation is carried out by means of the following steps:

(i) At first, 2's complement of the subtrahend is found.

(ii) Then it is added to the minuend.

(iii) If the final carry over of the sum is 1, it is dropped and the result is positive.

(iv) If there is no carry over, the two's complement of the sum will be the result and it is

negative.

The following examples on subtraction by 2's complement will make theprocedure clear:

Evaluate:

(i) 110110 - 10110

Solution:

The numbers of bits in the subtrahend is 5 while that of minuend is 6. We make the number ofbits in the subtrahend equal to that of minuend by taking a `0' in the sixth place of the subtrahend.

Now, 2's complement of 010110 is (101101 + 1) i.e.101010. Adding this with the minuend.

	1	1 0 1 1 0	Minuend
	1	0 1 0 1 0	2's complement of subtrahend
Carry over 1	1	0 0 0 0 0	Result of addition

After dropping the carry over we get the result of subtraction to be 100000.

(ii) 10110 – 11010

Solution:

2's complement of 11010 is (00101 + 1) i.e. 00110. Hence

Minued -	1 0 1 1 0
2's complement of subtrahend -	0 0 1 1 0
Result of addition -	1 1 1 0 0

As there is no carry over, the result of subtraction is negative and is obtained by writing the 2's complement of 11100 i.e.(00011 + 1) or 00100.

Hence the difference is – 100.

(iii) 1010.11 – 1001.01

Solution:

2's complement of 1001.01 is 0110.11. Hence

Minued - 1 0 1 0 . 1 1

2's complement of subtrahend - 0 1 1 0 . 1 1

Carry over 1 0 0 0 1 . 1 0

After dropping the carry over we get the result of subtraction as 1.10.

(iv) 10100.01 – 11011.10

Solution:

2's complement of 11011.10 is 00100.10. Hence

Minued - 1 0 1 0 0 . 0 1

2's complement of subtrahend - 0 1 1 0 0 . 1 0

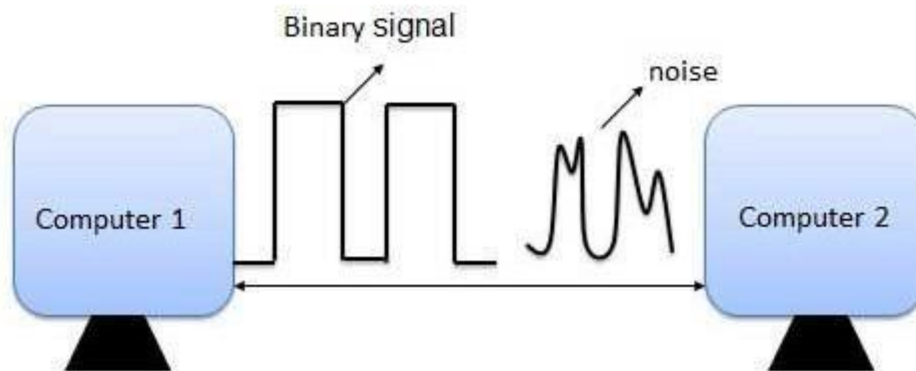
Result of addition - 1 1 0 0 0 . 1 1

As there is no carry over the result of subtraction is negative and is obtained by writing the 2's complement of 11000.11.

Hence the required result is – 00111.01.

Error Detection & Correction

Error is a condition when the output information does not match with the input information. During transmission, digital signals suffer from noise that can introduce errors in the binary bits travelling from one system to other. That means a 0 bit may change to 1 or a 1 bit may change to 0.



Error-Detectingcodes

Whenever a message is transmitted, it may get scrambled by noise or data may get corrupted. To avoid this, we use error-detecting codes which are additional data added to a given digital message to help us detect if an error occurred during transmission of the message. A simple example of error-detecting code is **parity check**.

Error-Correctingcodes

Along with error-detecting code, we can also pass some data to figure out the original message from the corrupt message that we received. This type of code is called an error- correcting code. Error-correcting codes also deploy the same strategy as error-detecting codes but additionally, such codes also detect the exact location of the corrupt bit.

In error-correcting codes, parity check has a simple way to detect errors along with a sophisticated mechanism to determine the corrupt bit location. Once the corrupt bit is located, its value is reverted (from 0 to 1 or 1 to 0) to get the original message.

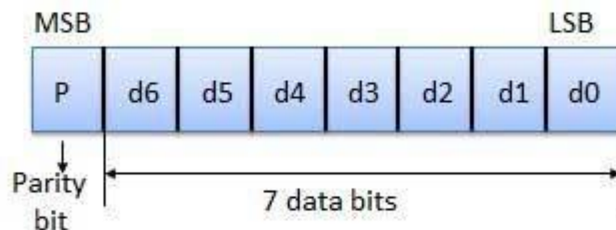
HowtoDetectandCorrectErrors?

To detect and correct the errors, additional bits are added to the data bits at the time of transmission.

- The additional bits are called **parity bits**. They allow detection or correction of theerrors.
- The data bits along with the parity bits form a **code word**.

Parity Checking of Error Detection

It is the simplest technique for detecting and correcting errors. The MSB of an 8-bits word is used as the parity bit and the remaining 7 bits are used as data or message bits. The parity of 8-bits transmitted word can be either even parity or odd parity.



Even parity -- Even parity means the number of 1's in the given word including the parity bit should be even (2,4,6,).

Odd parity -- Odd parity means the number of 1's in the given word including the parity bit should be odd (1,3,5,).

Use of Parity Bit

The parity bit can be set to 0 and 1 depending on the type of the parity required.

- For even parity, this bit is set to 1 or 0 such that the no. of "1 bits" in the entire word is even. Shown in fig. (a).
- For odd parity, this bit is set to 1 or 0 such that the no. of "1 bits" in the entire word isodd. Shown in fig. (b).

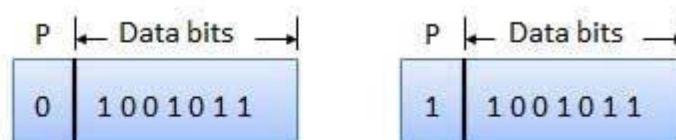


Fig. (a)

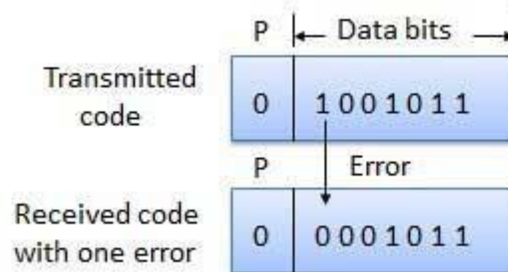
Fig. (b)

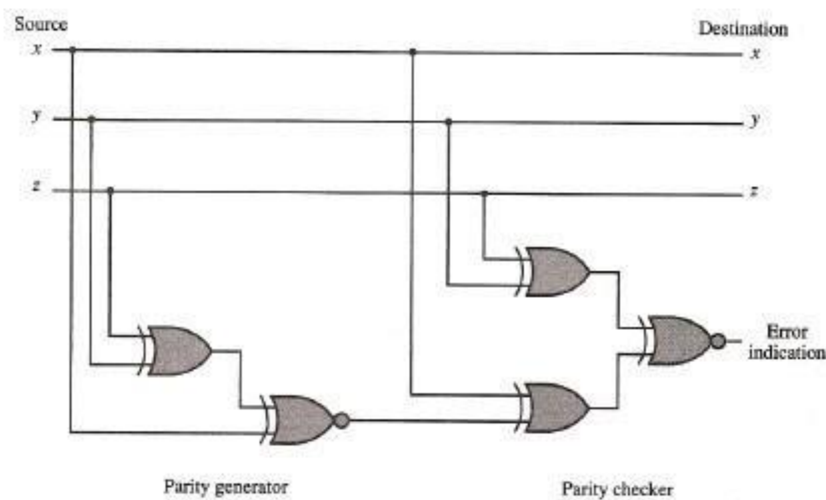
How Does Error Detection Take Place?

Parity checking at the receiver can detect the presence of an error if the parity of the receiversignal is different from the expected parity. That means, if it is known that the parity of the transmitted

signal is always going to be "even" and if the received signal has an odd parity, then the receiver can conclude that the received signal is not correct. If an error is detected,

then the receiver will ignore the received byte and request for retransmission of the same byte to the transmitter.





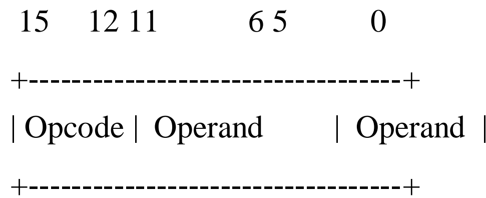
Instruction Codes

Computer instructions are the basic components of a machine language program. They are also known as *macro operations*, since each one is comprised of sequences of micro operations. Each instruction initiates a sequence of micro operations that fetch operands from registers or memory, possibly perform arithmetic, logic, or shift operations, and store results in registers or memory.

Instructions are encoded as binary *instruction codes*. Each instruction code contains of a *operation code*, or *opcode*, which designates the overall purpose of the instruction (e.g. add, subtract, move, input, etc.). The number of bits allocated for the opcode determined how many different instructions the architecture supports.

In addition to the opcode, many instructions also contain one or more *operands*, which

indicate where in registers or memory the data required for the operation is located. For example, and add instruction requires two operands, and a not instruction requires one.



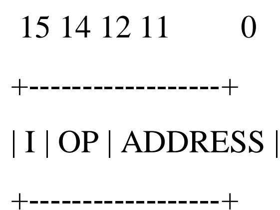
The opcode and operands are most often encoded as unsigned binary numbers in order to minimize the number of bits used to store them. For example, a 4-bit opcode encoded as a binary number could represent up to 16 different operations.

The *control unit* is responsible for decoding the opcode and operand bits in the instruction register, and then generating the control signals necessary to drive all other hardware in the CPU to perform the sequence of microoperations that comprise the instruction.

Basic Computer Instruction Format:

The Basic Computer has a 16-bit instruction code similar to the examples described above. It supports direct and indirect addressing modes.

How many bits are required to specify the addressing mode?



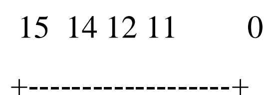
I = 0: direct

I = 1: indirect

Computer Instructions

All Basic Computer instruction codes are 16 bits wide. There are 3 instruction code formats:

Memory-reference instructions take a single memory address as an operand, and have the format:



```
| I | OP  | Address |
+-------------------+
```

If I = 0, the instruction uses direct addressing. If I = 1, addressing in indirect. How many memory-reference instructions can exist?

Register-reference instructions operate solely on the AC register, and have the following format:

```
 15  14 12 11       0
+------------------+
| 0 | 111 | OP       |
+------------------+
```

How many register-reference instructions can exist? How many memory- reference instructions can coexist with register-reference instructions?

Input/output instructions have the following format:15 14 12

```
 11                 0
+------------------+
| 1 | 111 | OP       |
+------------------+
```

How many I/O instructions can exist? How many memory-reference instructions can coexist with register-reference and I/O instructions?

Timing and Control

All sequential circuits in the Basic Computer CPU are driven by a master clock, withthe exception of the INPR register. At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU.

Control unit design and implementation can be done by two general methods:

- A *hardwired* control unit is designed from scratch using traditional digital logic design techniques to produce a minimal, optimized circuit. In other words, the control unit is like

an ASIC (application-specific integrated circuit).

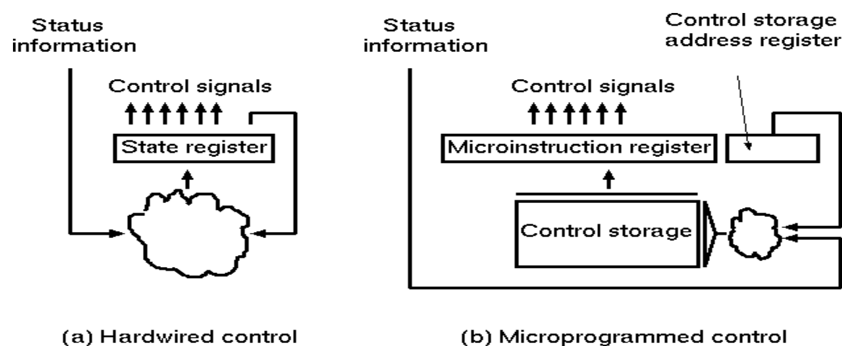
- A *micro-programmed* control unit is built from some sort of ROM. The desired control signals are simply stored in the ROM, and retrieved in sequence to drive the micro operations needed by a particular instruction.

Micro programmed control:

Micro programmed control is a control mechanism to generate control signals byusing a memory called control storage (CS), which contains the control signals. Although micro programmed control seems to be advantageous to CISC machines, since CISC requires systematic development of sophisticated control signals, there is no intrinsic difference between these 2 control mechanisms.

Hard-wired control:

Hardwired control is a control mechanism to generate control signals by using appropriate finite state machine (FSM). The pair of "microinstruction-register" and "control storage address register" can be regarded as a "state register" for the hardwired control. Note that the control storage can be regarded as a kind of combinational logic circuit. We can assign any 0, 1 values to each output corresponding to each address, which can be regarded as the input for a combinational logic circuit. This is a truth table.



(a) Hardwired control (b) Microprogrammed control

Microprogrammed control	Hardwired control
It is the microprogram in control store that generates control signals.	It is the sequential circuit that generates control signals.
Speed of operation is low, because it involves memory access.	Speed of operation is high.
Changes in control behavior can be implemented easily by modifying the microinstruction in the control store.	Changes in control unit behavior can be implemented only by redesigning the entire unit.

Instruction Cycle

In this chapter, we examine the sequences of micro operations that the Basic Computer goes through for each instruction. Here, you should begin to understand how the required control signals for each state of the CPU are determined, and how they are generated by the control unit.

The CPU performs a sequence of micro operations for each instruction. The sequence for each instruction of the Basic Computer can be refined into 4 abstract phases:

1. Fetch instruction
2. Decode
3. Fetch operand
4. Execute

Program execution can be represented as a top-down design:

1. Program execution
    a. Instruction 1
        i. Fetch instruction
        ii. Decode
        iii. Fetch operand
        iv. Execute
    b. Instruction 2
        i. Fetch instruction
        ii. Decode
        iii. Fetch operand

iv. Execute

c. Instruction 3 ...

Program execution begins with:

PC ← address of first instruction, SC ← 0

After this, the SC is incremented at each clock cycle until an instruction is completed, and then it is cleared to begin the next instruction. This process repeats until a HLT instruction is executed, or until the power is shut off.

Instruction Fetch and Decode

The instruction fetch and decode phases are the same for all instructions, so the control functions and micro operations will be independent of the instruction code. Everything that happens in this phase is driven entirely by timing variables T0, T1 and T2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

T2: D0-7 ← decoded IR(12-14), AR ← IR(0-11), I ← IR(15)

For every timing cycle, we assume SC ← SC + 1 unless it is stated that SC ← 0.

The operation D0-7 ← decoded IR(12-14) is not a register transfer like most of our micro operations, but is actually an inevitable consequence of loading a value into the IR register. Since the IR outputs 12-14 are directly connected to a decoder, the outputs of that decoder will change as soon as the new values of IR(12-14) propagate through the decoder.

Note that incrementing the PC at time T1 assumes that the next instruction is at the next address. This may not be the case if the current instruction is a branch instruction. However, performing the increment here will save time if the next instruction immediately follows, and will do no harm if it doesn't. The incremented PC value is simply overwritten by branch instructions.

In hardware development, unlike serial software development, it is often advantageous to perform work that may not be necessary. Since we can perform multiple micro operations at the same time, we might was well do everything that *might* be useful at the earliest possible time. Likewise, loading AR with the

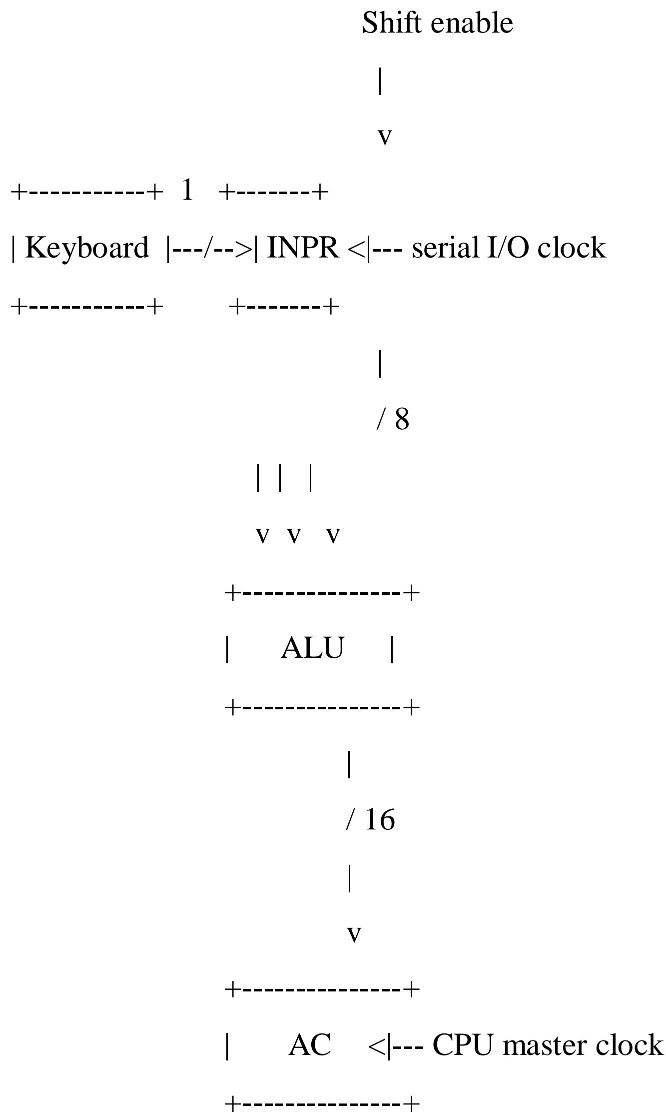
address field from IR at T2 is only useful if the instruction is a memory-reference instruction. We won't know this until T3, but there is no reason to wait since there is no harm in loading AR immediately.

Input-Output and Interrupt

Hardware Summary

The Basic Computer I/O consists of a simple terminal with a keyboard and aprinter/monitor.

The keyboard is connected serially (1 data wire) to the INPR register. INPR is a shift register capable of shifting in external data from the keyboard one bit at a time. INPR outputs are connected in parallel to the ALU.



How many CPU clock cycles are needed to transfer a character from the keyboard to the INPR register? (tricky)

Are the clock pulses provided by the CPU master clock?

RS232, USB, Firewire are serial interfaces with their own clock independent of the CPU. (USB speed is independent of processor speed.)

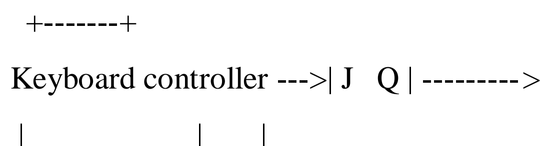
- RS232: 115,200 kbps (some faster)
- USB: 11 mbps
- USB2: 480 mbps
- FW400: 400 mbps
- FW800: 800 mbps
- USB3: 4.8 gbps

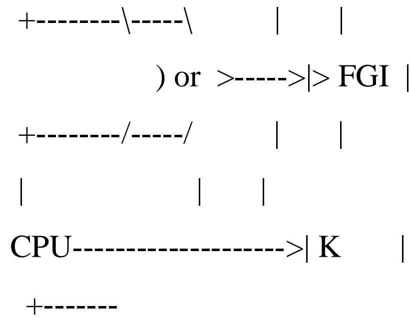
OUTR inputs are connected to the bus in parallel, and the output is connected serially to the terminal. OUTR is another shift register, and the printer/monitor receives an end-bit during each clock pulse.

I/O Operations

Since input and output devices are not under the full control of the CPU (I/O events are asynchronous), the CPU must somehow be told when an input device has new input ready to send, and an output device is ready to receive more output. The FGI flip- flop is set to 1 after a new character is shifted into INPR. This is done by the I/O interface, not by the control unit. This is an example of an asynchronous input event (not synchronized with or controlled by the CPU).

The FGI flip-flop must be cleared after transferring the INPR to AC. This must be done as a micro operation controlled by the CU, so we must include it in the CU design. The FGO flip-flop is set to 1 by the I/O interface after the terminal has finished displaying the last character sent. It must be cleared by the CPU after transferring a character into OUTR. Since the keyboard controller only sets FGI and the CPU only clears it, a JK flip-flop is convenient:





How do we control the CK input on the FGI flip-flop? (Assume leading-edge triggering.)

There are two common methods for detecting when I/O devices are ready, namely *software polling* and *interrupts*. These two methods are discussed in the following sections.

Stack Organization

Stack is the storage method of the items in which the last item included is the first one to be removed/taken from the stack. Generally a stack in the computer is a **memory** unit with an address **register** and the **register** holding the address of the stack is known as the Stack Pointer (SP). A stack performs Insertion and Deletion operation, were the operation of inserting an item is known as Push and operation of deleting an item is known as Pop. Both Push and Pop operation results inincrementing and decrementing the stack pointer respectively.

Register Stack

Register or **memory** words can be organized to form a stack. The stack pointer is a **register** that holds the **memory** address of the top of the stack. When an item need to be deleted from the stack, item on the top of the stack is deleted and the stack pointer is decremented. Similarly, when an item needs to be added, the stack pointer is incremented and writing the word at the position indicated by the stack pointer. There are two 1 bit register; FULL and EMTY that are used for describing the stack **overflow** and **underflow** conditions. Following micro-operations are performedduring inserting and deleting an item in/from the stack.

Insert:

SP <- SP + 1 // Increment the stack pointer to point the next higher address//

M[SP] <- DR // Write the item on the top of the stack//

If (SP = 0) then (Full <- 1) // Check overflow condition //EMTY

<- 0 // Mark that the stack is not empty //

Delete:

DR <- M[SP] //Read an item from the top of the stack// SP <-

SP 1 //Decrement the stack pointer //

If (SP = 0) then (EMTY <- 1) //Check underflow condition //FULL <-

0 //Mark that the stack is not full //

Get all the resource regarding the **homework help** and **assignment help** at Transtutors.com. With our team of experts, we are capable of providing **homework help** and **assignment help** for all levels. With us you can be rest assured the all the content provided for **homework help** and **assignment help** will be original and plagiarism free.

Register Stack:-

A stack can be placed in a portion of a large memory as it can be organized as a collection of a finite number of memory words as register.

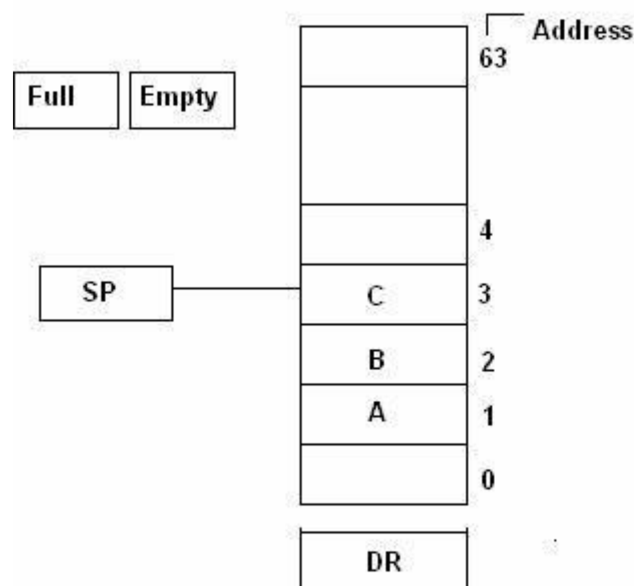


Figure :3 Block Diagram of a 64-word stack

In a 64- word stack, the stack pointer contains 6 bits because 26 = 64.

The one bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty. DR is the data register that holes the binary data to be written into on read out of the stack. Initially, SP is decide to O, EMTY is set to 1, FULL = 0, so that SP points to the word at address O and the stack is masked empty and not full.

PUSH SP ® SP + 1 increment stack pointer M

[SP] ® DR unit item on top of the Stack

It (SP = 0) then (FULL ® 1) check it stack is full

EMTY ® 0 mask the stack not empty.

POP DR ® [SP] read item trans the top of stack SP

® SP –1 decrement SP

It (SP = 0) then (EMTY ® 1) check it stack is empty

FULL ® 0 mark the stack not full.

A stack can be placed in a portion of a large memory or it can be organized asa collection of a finite number of memory words or registers. Figure X shows the organization of a 64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently ontop of the stack.

Three items are placed in the stack: A, B, and C in the order. item C is on the top of the stack so that the content of sp is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next higher location in the stack. Note that item C has read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because 26=64. since SP has only six bits, it cannot exceed a number greater than 63(111111 in binary). When

63 is incremented by 1, the result is 0 since 111111 + 1 =1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one bit register Full is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty ofitems. DR is the data register that holds the binary data to be written in to or read out of the stack.

Initially, SP is cleared to 0, Emty is set to 1, and Full is cleared to 0, so that SP points to the word at address o and the stack is marked empty and not full. if the stack is not full , a new item is inserted with a push operation. the push operation is implemented with the following sequence of micro-operation.

SP ←SP + 1 (Increment stack pointer) M(SP) ←
DR (Write item on top of the stack)
if (sp=0) then (Full ← 1) (Check if stack is full) Emty
← 0 (Marked the stack not empty)

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack. Note that SP holds the address of the top of the stack and that M(SP) denotes the memory word specified by the address presently available in SP, the first item stored in the stack is at address 1. The last item is stored at address 0, if SP reaches 0, the stack is full of item, so FULLL is set to 1.

This condition is reached if the top item prior to the last push was in location 63 and after increment SP, the last item stored in location 0. Once an item is stored in location 0, there are no more empty register in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.

DR← M[SP] Read item from the top of stack SP
← SP-1 Decrement stack Pointer
if(SP=0) then (Emty ← 1) Check if stack is empty
FULL ← 0 Mark the stack not full

The top item is read from the stack into DR. The stack pointer is then decremented. if its value reaches zero, the stack is empty, so Empty is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP. Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equal to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when FULL=1 or popped when EMTY =1.

Memory Stack :

A stack can exist as a stand-alone unit as in figure 4 or can be implemented in a random access memory attached to CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure shows a portion of computer memory partitioned into three segment program, data and stack. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data. The stack pointer SP points at the top of the stack. The three register are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or POP items into or fromthe stack.

As show in figure :4 the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address hat can be used for the stack is 3000. No previous are available for stack limit checks. We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows.

$$SP \leftarrow SP-1$$
$$M[SP] \leftarrow DR$$

he stack pointer is decremented so that it points at the address of the next word. A Memory write operation insertion the word from DR into the top of the stack. A new item is deleted with a pop operation as follows.

$$DR \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

The top item is read from the stack in to DR. The stack pointer is then incremented to point at the next item in the stack. Most computers do not provide hardware to check for stack overflow (FULL) or underflow (Empty). The stack limit can be checked by using two processor register: one to hold upper limit and other hold the lower limit. After the pop or push operation SP is compared with lower or upper limit register.

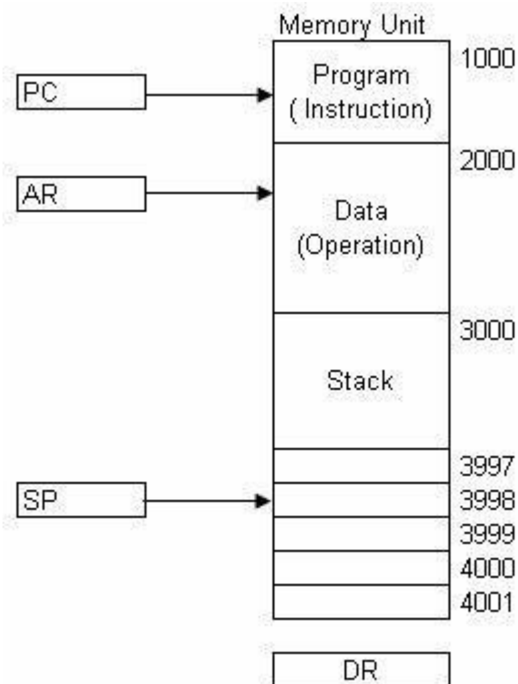


Figure : 4 Computer memory with program, data and stack segments

REVERSE POLISH NOTATION

For example: A x B + C x D is an arithmetical expression written in infix notation, herex (denotes multiplication). In this expression A and B are two operands and x is an operator, similarly C and D are two operands and x is an operator. In this expression +

is another operator which is written between (A x B) and (C x D). Because of theprecedence of the operator multiplication is done first. The order of precedence is as:

1. Exponentiation have precedence one.
2. Multiplication and Division has precedence two.

3. Addition and subtraction has precedence three.

Reverse polish notation is also known as postfix notation is defined as: In postfix notation operator is written after the operands. Examples of postfix notation are AB+ and CD-. Here A and B are two operands and the operator is written after these two operands. The conversion from infix expression into postfix expression is shown below.

- Convert the infix notation A x B + C x D + E x F into postfix notation?

SOLUTION

A x B + C x D + E x F

= [ABx] + [CDx] + [EFx]

= [ABxCDx] + [EFx]

= [ABxCDxEFx]

= ABxCDxEFx

So the postfix notation is ABxCDxEFx.

- Convert the infix notation {A – B + C x (D x E – F)} / G + H x K into postfix notation?

{A – B + C x (D x E – F)} / G + H x K

= {A – B + C x ([DEx] – F)} / G + [HKx]

= {A – B + C x [DExF-]} / [GHKx+]

= {A – B + [CDExF-x]} / [GHKx+]

= {[AB-] + [CDExF-x]} / [GHKx+]

= [AB-CDExF-x+] / [GHKx+]

= [AB-CDExF-x+GHKx+/]

= AB-CDExF-x+GHKx+/

So the postfix notation is AB-CDExF-x+GHKx+/.

Now let’s how to evaluate a postfix expression, the algorithm for the evaluation ofpostfix notation is shown below:

ALGORITHM:

(Evaluation of Postfix notation) This algorithm finds the result of a postfix expression. Step1: Insert a symbol (say #) at the right end of the postfix expression.

Step2: Scan the expression from left to right and follow the Step3 and Step4 for each ofthe symbol encountered.

Step3: if an element is encountered insert into stack.

Step4: if an operator (say &) is encountered pop the top element A (say) and next to top element B (say) perform the following operation x = B&A. Push x into the top of the stack.

Step5: if the symbol # is encountered then stop scanning.

- Evaluate the post fix expression 50 4 3 x 2 – + 7 8 x 4 / -?

SOLUTION

Put symbol # at the right end of the expression: 50 4 3 x 2 – + 7 8 x 4 / – #.

Postfix expression	Symbol scanned	Stack
50 4 3 x 2 – + 7 8 x 4 / – #	–	–
4 3 x 2 – + 7 8 x 4 / – #	50	50
3 x 2 – + 7 8 x 4 / – #	4	50, 4
x 2 – + 7 8 x 4 / – #	3	50, 4, 3
2 – + 7 8 x 4 / – #	x	50, 12
– + 7 8 x 4 / – #	2	50, 12, 2
+ 7 8 x 4 / – #	–	50, 10
7 8 x 4 / – #	+	60
8 x 4 / – #	7	60, 7
x 4 / – #	8	60, 7, 8
4 / – #	x	60, 56
/ – #	4	60, 56, 4

– #	/	60, 14

#	–	46
_	#	Result = 46

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 ® AC + M [×]

(2) General Register Organization ADD R1, R2, R3 R ® 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 are memory operand.

ADD R1, A, B A1 ® M [A] + M [B]

ADD R2, C, D R2 ® 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 on a memory word.

MOV R1, A R1 ® M [A]

ADD R1, B R1 ® R1 + M [B]

MOV R2, C R2 ® M [C] X = (A + B) * (C + D)

ADD R2, D R2 ® R2 + M [D]

MUL R1, R2 R1 ® R1 * R2

MOV X1 R1 M [X] ® R1

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.

LOAD A AC ® M [A]

ADD B AC ® AC + M [B]

STORE T M [T] ® AC X = (A +B) × (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.

LOAD C AC ® M (C)

ADD D AC ® AC + M (D)

ML T AC ® AC + M (T)

STORE X M [×]® AC

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)

PUSH A TOS ® A

PUSH B TOS ® B

ADD TOS ® (A + B)

PUSH C TOS ® C

PUSH D TOS ® D

ADD TOS ® (C + D)

MUL TOS ® (C + D) * (A + B)

POP X M [X] TOS

Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer register as memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction between the operand is activity referenced. Computer use addressing mode technique for the purpose of accommodating one or both of the following provisions.

(1) To give programming versatility to the uses by providing such facilities as pointer to memory, counters for top control, indexing of data, and program relocation.

(2) To reduce the number of bits in the addressing fields of the instruction.

Addressing Modes: The most common addressing techniques are

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement
- Stack

All computer architectures provide more than one of these addressing modes. The question arises as to how the control unit can determine which addressing mode is being used in a particular instruction. Several approaches are used. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction

format can be used as a mode field. The value of the mode field determines which addressing mode is to be used.

What is the interpretation of effective address. In a system without virtual memory, the effective address will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer.

Opcode	Mode	Address

Immediate Addressing:

The simplest form of addressing is immediate addressing, in which theoperand is actually present in the instruction:

OPERAND = A

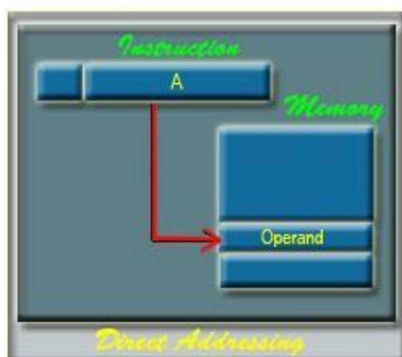
This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the world length.

Direct Addressing:

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

EA = A

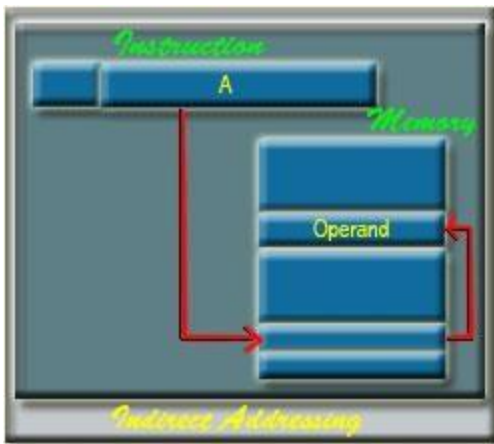
It requires only one memory reference and no special calculation.



Indirect Addressing:

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as indirect addressing:

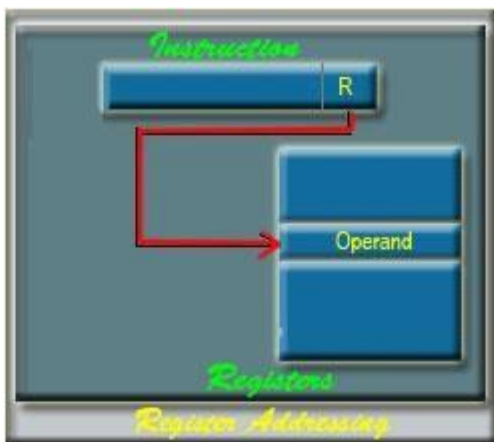
EA = (A)



Register Addressing:

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address: EA = R

The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.



The exact register location of the operand in case of Register Addressing Mode is shown

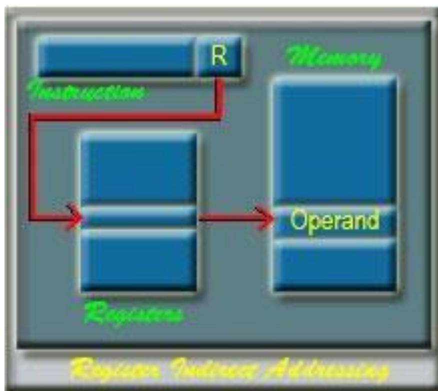
in the Figure 34.4. Here, 'R' indicates a register where the operand is present.

Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location. It requires only one memory reference and no special calculation.

EA = (R)

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.



Displacement Addressing:

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing:

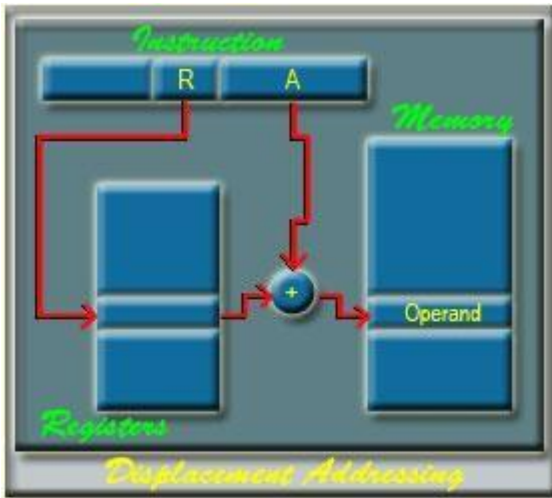
EA = A + (R)

Displacement addressing requires that the instruction have two address fields,at least one of which is explicit. The value contained in one address field (value = A)is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

The general format of Displacement Addressing is shown in the Figure 4.6. Three of the most common use of displacement addressing are:

- Relative addressing
- Base-register addressing

• Indexing



Relative Addressing:

For relative addressing, the implicitly referenced register is the programcounter (PC). That is, the current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the addressof the instruction.

Base-Register Addressing:

The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be explicit or implicit. In some implementation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly.

Indexing:

The address field references a main memory address, and the reference register contains a positive displacement from that address. In this case also the register reference is sometimes explicit and sometimes implicit. Generally index register are used for iterative tasks, it is typical that there is a need to increment or decrement the index register after each reference to it. Because this is such a common operation, some system will automatically do this as part of the same instruction cycle.

This is known as auto-indexing. We may get two types of auto-indexing: -one is auto-incrementing and the other one is -auto-decrementing. If certain registers are devoted exclusively to indexing,

then auto-indexing can be invoked implicitly andautomatically. If general purpose register are used, the auto index operation may need to be signaled by a bit in the instruction.

Auto-indexing using increment can be depicted as follows:

EA = A + (R)

R = (R) + 1

Auto-indexing using decrement can be depicted as follows:

EA = A + (R)

R = (R) - 1

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. There are two possibilities: The indexing is performed either before or after the indirection. If indexing is performed after the indirection, it is termed post indexing

EA = (A) + (R)

First, the contents of the address field are used to access a memory location containing an address. This address is then indexed by the register value. With pre indexing, the indexing is performed before the indirection: EA = (A + (R)

An address is calculated, the calculated address contains not the operand, but the address of the operand.

Stack Addressing:

A stack is a linear array or list of locations. It is sometimes referred to as a pushdown list or last-in-first-out queue. A stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses. The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

Data Transfer & Manipulation

Computer provides an extensive set of instructions to give the user the flexibility to carryout various computational task. Most computer instruction can be classified into three categories.

(1) Data transfer instruction

(2) Data manipulation instruction

(3) Program control instruction

Data transfer instruction cause transferred data from one location to another without changing the binary instruction content. Data manipulation instructions are those that perform arithmetic logic, and shift operations. Program control instructions provide

decision-making capabilities and change the path taken by the program when executed in the computer.

(1) Data Transfer Instruction

Data transfer instruction move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processes registers, between processes register & input or output, and between processes register themselves

(Typical data transfer instruction)

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

(2) Data Manipulation Instruction

It performs operations on data and provides the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types.

(a) Arithmetic Instruction

(b) Logical bit manipulation Instruction

(c) Shift Instruction.

(a) Arithmetic Instruction

Name	**Mnemonic**
Increment	INC
Decrement	DEC
Add	Add
Subtract	Sub
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with Basses	SUBB
Negate (2's Complement)	NEG

(b) Logical & Bit Manipulation Instruction

Name	**Mnemonic**
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-Or	XOR
Clear Carry	CLRC
Set Carry	SETC
Complement Carry	COMC
Enable Interrupt	ET
Disable Interrupt	OI

(c) Shift Instruction

Instructions to shift the content of an operand are quite useful and one often provided in several variations. Shifts are operation in which the bits of a word are moved to the left or right. The bit-shifted in at the and of the word determines the type of shift used. Shift instruction may specify

either logical shift, arithmetic shifts, or rotate type shifts.

Name	**Mnemonic**
Logical Shift right	SHR
Logical Shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Introduction about Program Control:-

A program that enhances an operating system by creating an environment in which you can run other programs. Control programs generally provide a graphical interface and enable you to run several programs at once in different windows.

Control programs are also called *operating environments*.

The program control functions are used when a series of conditional or unconditional jump and return instruction are required. These instructions allow the program to execute only certain sections of the control logic if a fixed set of logic conditions are met. The most common instructions for the program control availablein most controllers are described in this section.

Introduction About status bit register:-

A **status register**, **flag register**, or **condition code register** is a collection of status flag bits for a processor. An example is the FLAGS register of the computer architecture. The flags might be part of a larger register, such as a program status word (PSW) register.

The status register is a hardware register which contains information about the state of the processor. Individual bits are implicitly or explicitly read and/or written by the machine code instructions executing on the processor. The status register in a traditional processor design includes at least three central flags: Zero, Carry, and Overflow, which are set or cleared

automatically as effects of arithmetic and bit manipulation operations. One or more of the flags may then be read by a subsequent conditional jump instruction (including conditional calls, returns, etc. in some machines) or by some arithmetic, shift/rotate or bitwise operation, typically using the carry flag as input in addition to any explicitly given operands. There are also processors where other classes of instructions may read or write the fundamental

zero, carry or overflow flags, such as block-, string- or dedicated input/output instructions, for instance.

Some CPU architectures, such as the MIPS and Alpha, do not use a dedicated flag register. Others do not implicitly set and/or read flags. Such machines either do not pass *implicit* status information between instructions at all, or do they pass it in a explicitly selected general purpose register.

A status register may often have other fields as well, such as more specialized flags, interrupt enable bits, and similar types of information. During an interrupt, the status of the thread currently executing can be preserved (and later recalled) bystoring the current value of the status register along with the program counter andother active registers into the machine stack or some other reserved area of memory. ***Common flags:-***

This is a list of the most common CPU status register flags, implemented in almost all modern processors.

Flag	Name	Description
Z	Zero flag	Indicates that the result of arithmetic or logical operation (or, sometimes, a load) was zero.
C	Carry flag	Enables numbers larger than a single word to beadded/subtracted by carrying a binary digit from a less significant word to the least significant bit of a more significant word as needed. It is also used to extend bit shifts and rotates in a similar manner on many processors (sometimes done via a dedicated **X** flag).

S / N	Sign flag Negative flag	Indicates that the result of a mathematical operation is negative. In some processors, the N and S flags are distinct with different meanings and usage: One indicates whether the last result was negative whereas the other indicates whether a subtraction or addition has taken place.
V / O / W	Overflow flag	Indicates that the signed result of an operation is too large to fit in the register width using twos complement representation.

Introduction About Conditional branch instruction:-

Conditional branch instruction:-

Conditional branch instruction is the branch instruction bit and BR instruction is theProgram control instruction.

The conditional Branch Instructions are listed as Bellow:-

Mnemonics	Branch Instruction	Tested control
BZ	Branch if Zero	Z=1
BNZ	Branch if not Zero	Z=0
BC	Branch if Carry	C=1
BNC	Branch if not Carry	C=0
BP	Branch if Plus	S=0
BM	Branch if Minus	S=1
BV	Branch if Overflow	V=1
BNV	Branch if not Overflow	V=0

Unsigned Compare(A-B):-

Mnemonics	Branch Instruction	Tested control
BHI	Branch if Higher	A > B
BHE	Branch if Higher or Equal	A >= B
BLO	Branch if Lower	A < B
BLE	Branch if Lower or Equal	A <= B
BE	Branch if Equal	A=B
BNE	Branch if not Equal	A not = B

Signed Compare(A-B):

Mnemonics	Branch Instruction	Tested control
BGT	Branch if Greater Than	A > B
BGE	Branch if Greater Than or Equal	A >= B
BLT	Branch if Less Than	A < B
BLE	Branch if Less Than or Equal	A <= B
BE	Branch if Equal	A=B
BNE	Branch if not Equal	A not = B

Introduction About program interrupt:-

When a Process is executed by the CPU and when a user Request for another Process then this will create disturbance for the Running Process. This is also called as the Interrupt.

Interrupts can be generated by User, Some Error Conditions and also by Software's and the hardware's. But CPU will handle all the Interrupts very carefully because when Interrupts are generated then the CPU must handle all the Interrupts Very carefully means the CPU will also Provide Response to the Various Interrupts those are generated. So that When an interrupt has Occurred then the CPU will handle by using the Fetch, decode and Execute Operations.

Interrupts allow the operating system to take notice of an external event, such as a mouse click. Software interrupts, better known as exceptions, allow the OS to handle unusual events like divide-by-zero errors coming from code execution.

The sequence of events is usually like this:

Hardware signals an interrupt to the processor

The processor notices the interrupt and suspends the currently running softwareThe processor jumps to the matching interrupt handler function in the OS

The interrupt handler runs its course and returns from the interrupt

The processor resumes where it left off in the previously running software

The most important interrupt for the operating system is the timer tick interrupt. The timer tic interrupt allows the OS to periodically regain control from the currently running user process. The OS can then decide to schedule another process, return back

to the same process, do housekeeping, etc. The timer tick interrupt provides the foundation for the concept of preemptive multitasking.

TYPES OF INTERRUPTS

Generally there are three types of Interrupts those are Occurred For Example

1) Internal Interrupt
2) External Interrupt.
3) Software Interrupt.

1.Internal Interrupt:

• When the hardware detects that the program is doing something wrong, it will usually generate an interrupt usually generate an interrupt.

– Arithmetic error - Invalid Instruction

– Addressing error - Hardware malfunction

– Page fault – Debugging

• A Page Fault interrupt is not the result of a program error, but it does require the operating system to get control.

The Internal Interrupts are those which are occurred due to Some Problem in the Execution For Example When a user performing any Operation which contains any Error and which contains any type of Error. So that Internal Interrupts are those which are occurred by the Some Operations or by Some Instructions and the Operations those are not Possible but a user is trying for that Operation. And The Software Interrupts are those which are made some call to the System for Example while we are Processing Some Instructions and when we wants to Execute one more Application Programs.

2.External Interrupt:

• I/O devices tell the CPU that an I/O request has completed by sending an interrupt signal to the processor.

• I/O errors may also generate an interrupt.

• Most computers have a timer which interrupts the CPU every so many interrupts the CPU every so many milliseconds.

The External Interrupt occurs when any Input and Output Device request for any Operation and the CPU will Execute that instructions first For Example When a Program is executed and when we move the Mouse on the Screen then the CPU will handle this External interrupt first and after that he will resume with his Operation.

3. Software interrupts:

These types if interrupts can occur only during the execution of an instruction. They can be used by a programmer to cause interrupts if need be. The primary purpose of such interrupts is to switch from user mode to supervisor mode.

A software interrupt occurs when the processor executes an INT instruction. Written in the program, typically used to invoke a system service. A processor interrupt is caused by an electrical signal on a processor pin. Typically used by devices to tell a driver that they require attention. The clock tick interrupt is very common; it wakes up the processor from a halt state and allows the scheduler to pick other work to perform.

A processor fault like access violation is triggered by the processor itself when it encounters a condition that prevents it from executing code. Typically when it tries to read or write from unmapped memory or encounters an invalid instruction.

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.

- A large number of instructions - typically from 100 to 250 instructions.
- Some instructions that perform specialized tasks and are used infrequently.
- A large variety of addressing modes - typically 5 to 20 different modes.
- Variable-length instruction formats

- 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,

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

CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

Example of RISC & CISC

Examples of CISC instruction set architectures are PDP-11, VAX, Motorola 68k, and your desktop PCs on intel's x86 architecture based too .

Examples of RISC families include DEC Alpha, AMD 29k, ARC, Atmel AVR, Blackfin, Intel i860 and i960, MIPS, Motorola 88000, PA-RISC, Power (including PowerPC), SuperH, SPARC and ARM too.

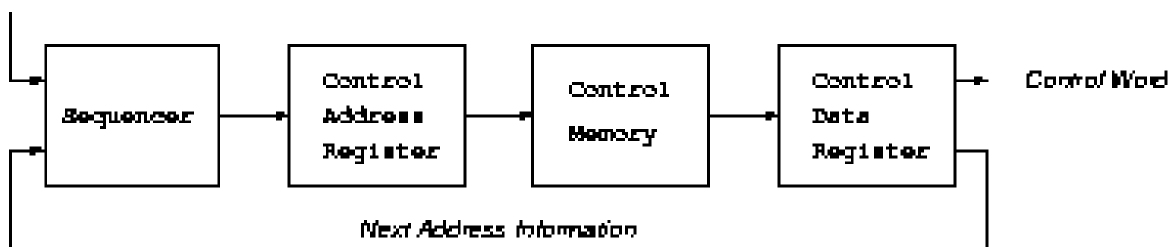
Which one is better ?

We cannot differentiate RISC and CISC technology because both are suitable at its specific application. What counts is how fast a chip can execute the instructions it is given and how well it runs existing software. Today, both RISC and CISC manufacturers are doing everything to get an edge on the competition.

<u>Control Memory:</u>

Control memory is a random access memory(RAM) consisting of addressable storage registers. It is primarily used in mini and mainframe computers. It is used as a temporary storage for data. Access to control memory data requires less time than to main memory; this speeds up CPU operation by reducing the number of memory references for data storage and retrieval. Access is performed as part of a control section sequence while the master clock oscillator is running. The control memory addresses are divided into two groups: a task mode and an executive (interrupt) mode.

Addressing words stored in control memory is via the address select logic for each of the register groups. There can be up to five register groups in control memory. These groups select a register for fetching data for programmed CPU operation or for maintenance console or equivalent display or storage of data via maintenance console or equivalent. During programmed CPU operations, these registers are accessed directly by the CPU logic. Data routing circuits are used by control memory to interconnect the registers used in control memory. Some of the registers contained in a control memory that operate in the task and the executive modes include the following: Accumulators Indexes Monitor clock status indicating registers Interrupt data registers



• The control unit in a digital computer initiates sequences of micro operations

• The complexity of the digital system is derived form the number of sequences that are performed

• When the control signals are generated by hardware, it is hardwired

• In a bus-oriented system, the control signals that specify micro operations are groups of bits that select the paths in multiplexers, decoders, and ALUs.

• The control unit initiates a series of sequential steps of micro operations

• The control variables can be represented by a string of 1's and 0's called a control word

• A micro programmed control unit is a control unit whose binary control variables are stored in memory

• A sequence of microinstructions constitutes a micro program

• The control memory can be a read-only memory

• Dynamic microprogramming permits a micro program to be loaded and uses a writable control memory

• A computer with a micro programmed control unit will have two separate memories: a main memory and a control memory

• The micro program consists of microinstructions that specify various internal control signals for execution of register micro operations

• These microinstructions generate the micro operations to:

- ➢ fetch the instruction from main memory
- ➢ evaluate the effective address
- ➢ execute the operation
- ➢ return control to the fetch phase for the next instruction

• The control memory address register specifies the address of the microinstruction

• The control data register holds the microinstruction read from memory

• The microinstruction contains a control word that specifies one or more micro operations for the data processor

• The location for the next micro instruction may, or may not be the next in sequence

• Some bits of the present micro instruction control the generation of the address of the next micro instruction

• The next address may also be a function of external input conditions

• While the micro operations are being executed, the next address is computed in the next address generator circuit (sequencer) and then transferred into the CAR to read the next micro instructions

• Typical functions of a sequencer are: o incrementing the CAR by one

- ➢ loading into the CAR and address from control memory

- transferring an external address
- loading an initial address to start the control operations

• A clock is applied to the CAR and the control word and next-address information are taken directly from the control memory

• The address value is the input for the ROM and the control work is the output

• No read signal is required for the ROM as in a RAM

• The main advantage of the micro programmed control is that once the hardware configuration is established, there should be no need for h/w or wiring changes

• To establish a different control sequence, specify a different set of microinstructions for control memory

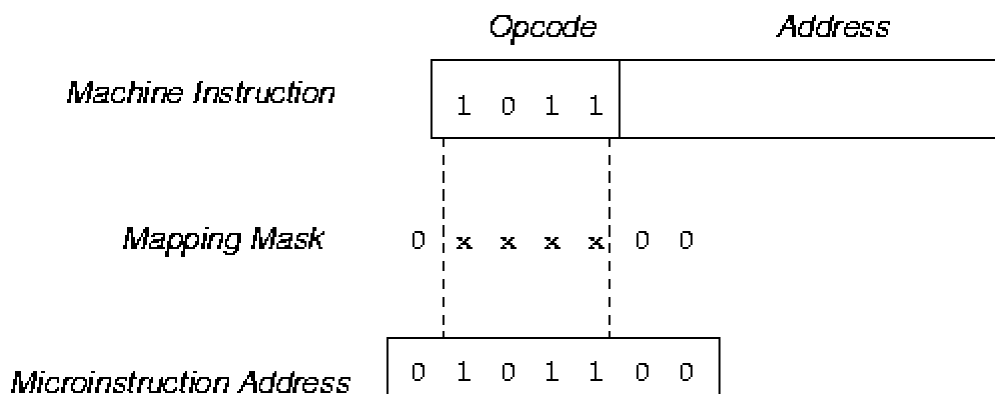
Addressing Sequencing:

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.



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

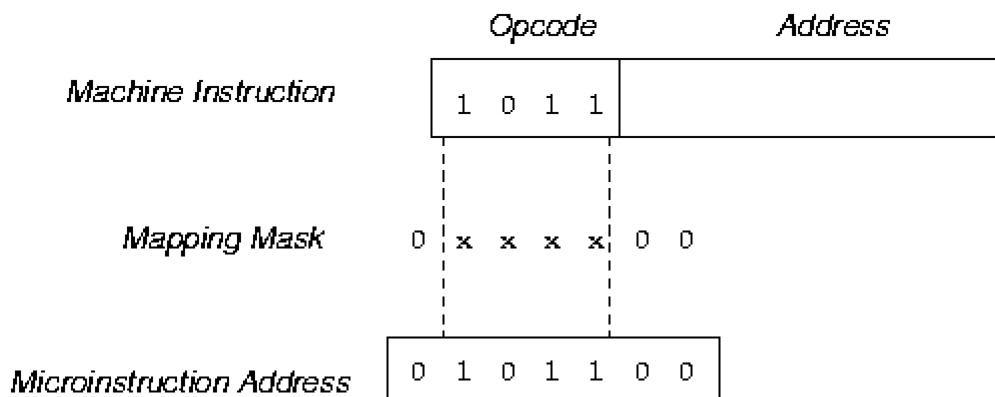
used to hold return address for subroutine-call branch operations

Conditional branches are necessary in the micro program. 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 micro program 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 determine 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:



Alternately, the *n*-bit opcode value can be used as the ``address" input of a ***2n x 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.

Eight Conditions for Signed-Magnitude Addition/Subtraction

	Operation	ADD Magnitudes	SUBTRACT Magnitudes		
			A > B	A < B	A = B
1	(+A) + (+B)	+ (A + B)			
2	(+A) + (-B)		+ (A – B)	- (B – A)	+ (A – B)
3	(-A) + (+B)		- (A – B)	+ (B – A)	+ (A – B)
4	(-A) + (-B)	- (A + B)			
5	(+A) - (+B)		+ (A – B)	- (B – A)	+ (A – B)
6	(+A) - (-B)	+ (A + B)			
7	(-A) - (+B)	- (A + B)			
8	(-A) - (-B)		- (A – B)	+ (B – A)	+ (A – B)

Addition and Subtraction

Four basic computer arithmetic operations are addition, subtraction, division and multiplication. The arithmetic operation in the digital computer manipulate data to produce results. It is necessary to design arithmetic procedures and circuits to program arithmetic operations using algorithm. The algorithm is a solution to any problem and it is stated by a finite number of well-defined procedural steps. The algorithms can be developed for the following types of data.

1. Fixed point binary data in signed magnitude representation
2. Fixed point binary data in signed 2's complement representation.
3. Floating point representation
4. Binary Coded Decimal (BCD) data

Addition and Subtraction with signed magnitude

Consider two numbers having magnitude A and B. When the signed numbers are added or subtracted, there can be 8 different conditions depending on the sign and the operation performed as shown in the table below:

Operation	Add magnitude	When A > B	When A < B	When A = B
(+A) + (+B)	+(A + B)	--	--	--
(+A) + (-B)	--	+(A - B)	-(B - A)	+(A - B)
(-A) + (+B)	--	-(A - B)	+(B - A)	+(A - B)
(-A) + (-B)	-(A + B)	--	--	--
(+A) - (+B)	--	+(A - B)	-(B - A)	+(A - B)
(+A) - (-B)	+(A + B)	--	--	--
(-A) - (+B)	-(A + B)	--	--	--
(-A) - (-B)	--	-(A - B)	+(B - A)	+(A - B)

From the table, we can derive an algorithm for addition and subtraction as follows:

Addition (Subtraction) Algorithm:

- When the signs of A & B are identical, add the two magnitudes and attach the sign of A to the result.
- When the sign of A & B are different, compare the magnitude and subtract the smaller number from the large number. Choose the sign of the result to be same as A if A > B, or the complement of the sign of A if A < B. If the two numbers are equal, subtract B from A and make the sign of the result positive.

Hardware Implementation

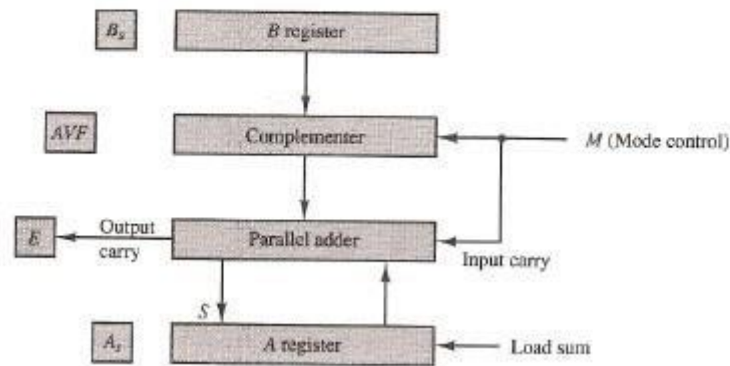


fig: Hardware for signed magnitude addition and subtraction

The hardware consists of two registers A and B to store the magnitudes, and two flip- flops As and Bs to store the corresponding signs. The results can be stored in the register A and As which acts as an accumulator. The subtraction is performed by adding A to the 2’s complement of B. The output carry is transferred to the flip-flop E. The overflow may occur during the add operation which is stored in the flip-flop A Ë… F. When m = 0, the output of E is transferred to the adder without any change along with the input carry of ‘0".

The output of the parallel adder is equal to A + B which is an add operation. When m = 1, the content of register B is complemented and transferred to parallel adder along with the input carry of 1. Therefore, the output of parallel is equal to A + B’ + 1 = A – B which is a subtract operation.

Hardware Algorithm

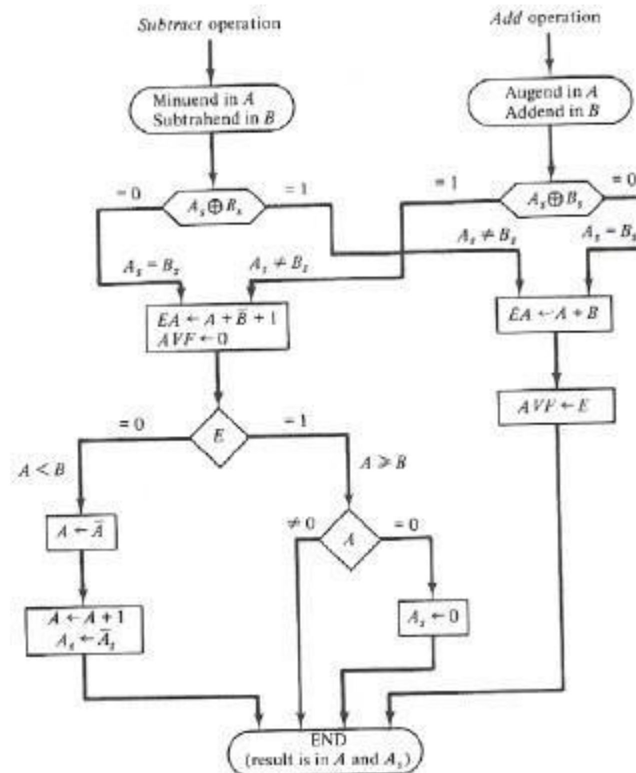


fig: flowchart for add and subtract operations

As and Bs are compared by an exclusive-OR gate. If output=0, signs are identical, if 1 signs are different.

- For Add operation, identical signs dictate addition of magnitudes and for operation identical signs dictate addition of magnitudes and for *subtraction*, different magnitudes dictate magnitudes be added. Magnitudes are added with a micro operation EA
- Two magnitudes are subtracted if signs are different for add operation and identical for subtract operation. Magnitudes are subtracted with a micro operation EA = B and number (this number is checked again for 0 to make positive 0 [As=0]) in A is correct result. E = 0 indicates A < B, so we take 2's complement of A.

Multiplication

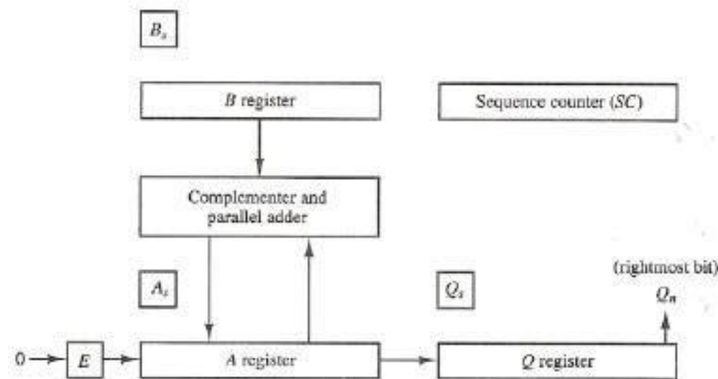
Hardware Implementation and Algorithm

Generally, the multiplication of two final point binary number in signed magnitude representation is performed by a process of successive shift and ADD operation. The process consists of looking at the successive bits of the multiplier (least significant bit first). If the multiplier is 1, then the

multiplicand is copied down otherwise, 0's are copied. The numbers

copied down in successive lines are shifted one position to the left and finally, all the numbers are added to get the product.

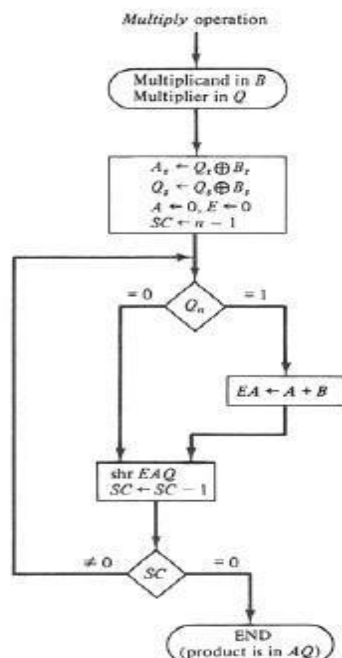
But, in digital computers, an adder for the summation (∑) of only two binary numbers are used and the partial product is accumulated in register. Similarly, instead of shifting the multiplicand to the left, the partial product is shifted to the right. The hardware for the multiplication of signed magnitude data is shown in the figure below.



Hardware for multiply operation

Initially, the multiplier is stored q register and the multiplicand in the B register. A register is used to store the partial product and the sequence counter (SC) is set to a number equal to the number of bits in the multiplier. The sum of A and B form the partial product and both shifted to the right using a statement "Shr EAQ" as shown in the hardware algorithm. The flip flops As, Bs & Qs store the sign of A, B & Q respectively. A binary '0" inserted into the flip-flop E during the shift right.

Hardware Algorithm



flowchart for multiply algorithm

Example: Multiply 23 by 19 using multiply algorithm.

multiplicand	E	A	Q	SC
Initially,	0	00000	10011	101(5)
Iteration1(Qn=1), add B first partial product	0	00000 +10111 10111		
shrEAQ,	0	01011	11001	100(4)
Iteration2(Qn=1) Add B Second partial product	1	01011 +10111 00010	11001	
shrEAQ,	0	10001	01100	011(3)
Iteration3(Qn=0) shrEAQ,	0	01000	10110	010(2)
Iteration4(Qn=0) shrEAQ,	0	00100	01011	001(1)
Iteration5(Qn=1 Add B Fifth partial product	0	00100 +10111 11011	01011	
shrEAQ,	0	01101	10101	000
FinalProductinAQ	0110110101			

The final product is in register A & Q. therefore, the product is 0110110101.

Booth Algorithm

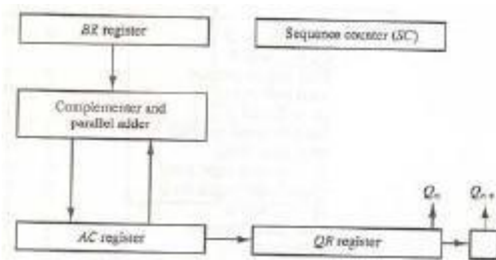
The algorithm that is used to multiply binary integers in signed 2's complement form is called booth multiplication algorithm. It works on the principle that the string 0's in the multiplier doesn't need addition but just the shifting and the sting of 1's from bit weight 2^k to 2^m can be treated as $2^{k+1} – 2^m$ (Example, +14 = 001110 = $2^{3=1} – 2^1$ = 14). The product can be obtained by shifting the binary multiplication to the left and subtraction the multiplier shifted left once.

According to booth algorithm, the rule for multiplication of binary integers in signed 2's

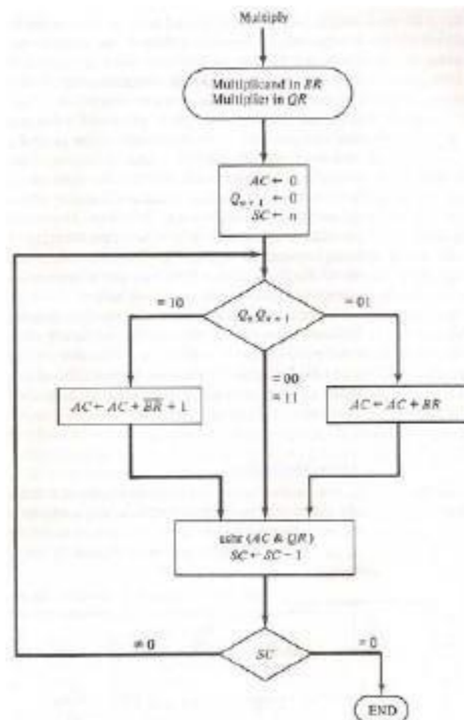
complement form are:

- The multiplicand is subtracted from the partial product of the first least significant bit is 1 in a string of 1's in the multiplicand.
- The multiplicand is added to the partial product if the first least significant bit is 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
- The partial product doesn't change when the multiplier bit is identical to the previous multiplier bit.

This algorithm is used for both the positive and negative numbers in signed 2's complement form. The hardware implementation of this algorithm is in figure below:



The flowchart for booth multiplication algorithm is given below:



flowchart for booth multiplication algorithm

Numerical Example: Booth algorithm

BR=10111(**Multiplicand**)

QR=10011(**Multiplier**)

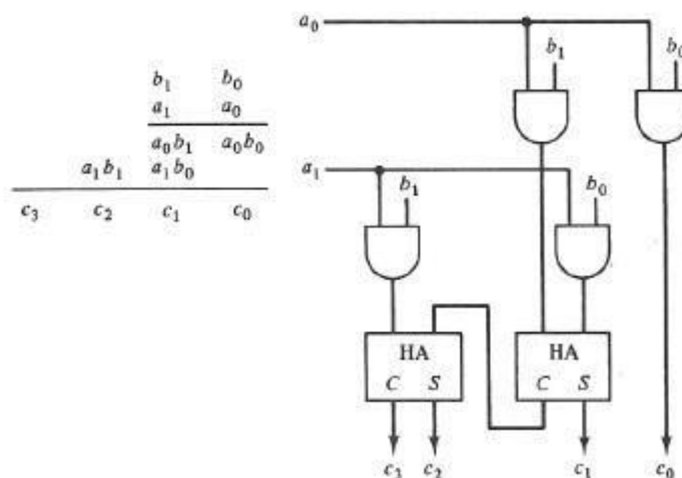
Array Multiplier

The multiplication algorithm first check the bits of the multiplier one at time and form partial product. This is a sequential process that requires a sequence of add and shift micro operation. This method is complicated and time consuming. The multiplication of 2 binary

numbers can also be done with one micro operation by using combinational circuit that provides the product all at once.

Example.

Consider that the multiplicand bits are b1 and b0 and the multiplier bits are a1 and a0. The partial product is c3c2c1c0. The multiplication two bits a0 and a1 produces a binary 1 if both the bits are 1, otherwise it produces a binary 0. This is identical to the AND operation and can be implemented with the AND gates as shown in figure.



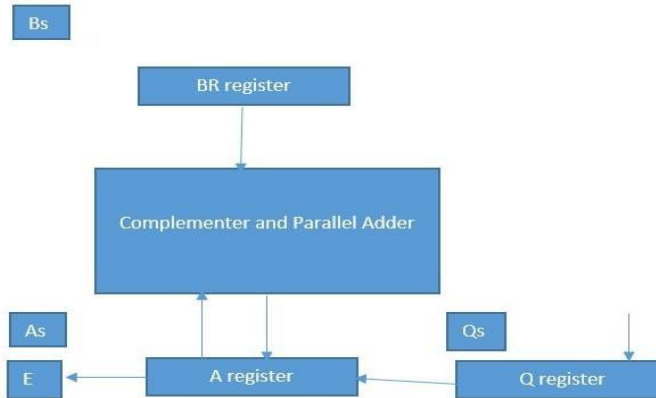
2-bit by 2-bit array multiplier

Division Algorithm

The division of two fixed point signed numbers can be done by a process of successive compare shift and subtraction. When it is implemented in digital computers, instead of shifting the divisor to the right, the dividend or the partial remainder is shifted to the left. The subtraction can be obtained by adding the number A to the 2's complement of number B. The information about the relative magnitudes of the information about the relative magnitudes of numbers can be obtained from the end carry,

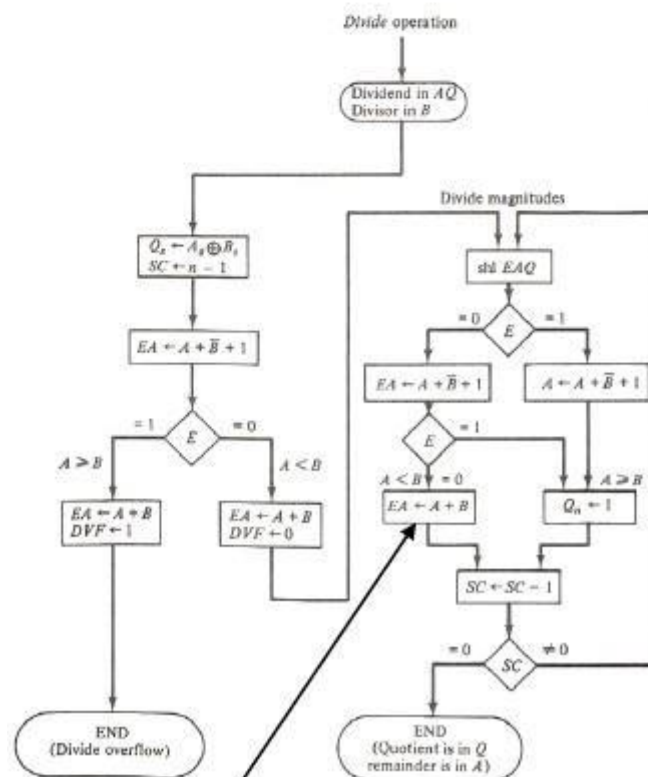
Hardware Implementation

The hardware implementation for the division signed numbers is shown id the figure.



Division Algorithm

The divisor is stored in register B and a double length dividend is stored in register A and Q. the dividend is shifted to the left and the divider is subtracted by adding twice complement of the value. If E = 1, then A >= B. In this case, a quotient bit 1 is inserted into Qn and the partial remainder is shifted to the left to repeat the process. If E = 0, then A > B. In this case, the quotient bit Qn remains zero and the value of B is added to restore the partial remainder in A to the previous value. The partial remainder is shifted to the left and approaches continues until the sequence counter reaches to 0. The registers E, A & Q are shifted to the left with 0 inserted into Qn and the previous value of E is lost as shown in the flow chart for division algorithm.



flowchart for division algorithm

This algorithm can be explained with the help of an example.

Consider that the divisor is 10001 and the dividend is 01110.

Divisor $B = 10001$, $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

binary division with digital hardware

Restoring method

Method described above is restoring **method** in which partial remainder is restored by adding the divisor to the negative result. Other methods:

Comparison method: A and B are compared prior to subtraction. Then if A >= B, B is subtracted from A. if A < B nothing is done. The partial remainder is then shifted left and numbers are compared again. Comparison inspects end carry out of the parallel adder before transferring to E.

Non-restoring method: In contrast to restoring method, when A -B is negative, B is not added to restore A but instead, negative difference is shifted left and then B is added. How is it possible? Let's argue:

- In flowchart for restoring method, when A < B, we restore A by operation A - B + B. Next time in a loop,

  this number is shifted left (multiplied by 2) and B subtracted again, which gives: 2 (A - B + B) – B = 2 A - B.

- In Non-restoring method, we leave A - B as it is. Next time around the loop, the number is

shifted left and B is added: 2 (A - B) + B = 2 A - B (same as above).

Divide Overflow

The division algorithm may produce a quotient overflow called dividend overflow. The overflow can occur of the number of bits in the quotient are more than the storage capacity of the register. The overflow flip-flop DVF is set to 1 if the overflow occurs.

The division overflow can occur if the value of the half most significant bits of the dividend is equal to or greater than the value of the divisor. Similarly, the overflow can occue=r if the dividend is divided by a 0. The overflow may cause an error in the result or sometimes it may stop the operation. When the overflow stops the operation of the system, then it is called divide stop.

Arithmetic Operations on Floating-Point Numbers

The rules apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations. Intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively & overflow or an underflow may occur. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. If their exponents differ, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider a

decimal example in which we wish to add 2.9400 x 10^2 to 4.3100 x 10^4. We rewrite 2.9400 x 10^2 as 0.0294 x 10^4 and then perform addition of the mantissas to get 4.3394 x 10^4. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

The steps in addition (FA) or subtraction (FS) of floating-point numbers (s1, eˆ , f1) fad{s2, eˆ 2, f2) are as follows.

1. Unpack sign, exponent, and fraction fields. Handle special operands such as zero, infinity, or NaN(not a number).

2. Shift the significand of the number with the smaller exponent right by $|e1 - e2|$ bits.

3. Set the result exponent er to max(e1,e2).

4. If the instruction is FA and s1= s2 or if the instruction is FS and s1 ≠ s2 then add the

significands; otherwise subtract them.

5. Count the number z of leading zeros. A carry can make z = -1. Shift the result significand left z bits or right 1 bit if z = -1.

6. Round the result significand, and shift right and adjust z if there is rounding overflow, which is a carry-out of the leftmost digit upon rounding.

7. Adjust the result exponent by er = er - z, check for overflow or underflow, and pack the result sign, biased exponent, and fraction bits into the result word.

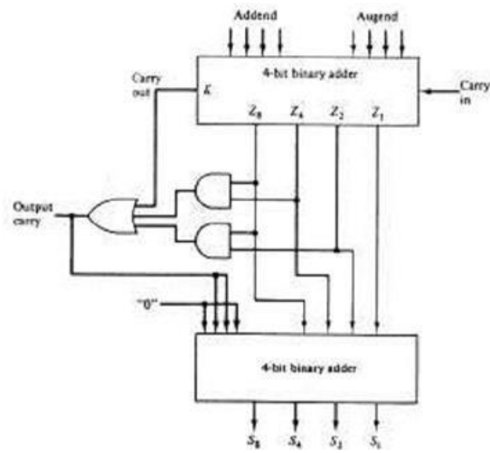
Operands	Alignment	Normalize and round
6.144×10^2	0.06144×10^4	1.003644×10^5
$+9.975 \times 10^4$	$+9.975 \times 10^4$	$+.0005 \times 10^5$
	10.03644×10^4	1.004×10^5

Operands	Alignment	Normalize and round
1.076×10^{-7}	1.076×100^{-7}	7.7300×100^{-9}
-9.987×100^{-8}	-0.9987×100^{-7}	$+.0005 \times 100^{-9}$
	0.0773×100^{-7}	7.730×100^{-9}

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

BCD Adder:

BCD adder A 4-bit binary adder that is capable of adding two 4-bit words having a BCD (binary-coded decimal) format. The result of the addition is a BCD-format 4-bit output word, representing the decimal sum of the addend and augend, and a carry that is generated if this sum exceeds a decimal value of 9. Decimal addition is thus possible using these devices.


Addend
Augend
4-bit binary adder
Carry out
K
Carry in
Z8
Z4
Z2
Z1
Output carry
"0"
4-bit binary adder
S8
S4
S2
S1

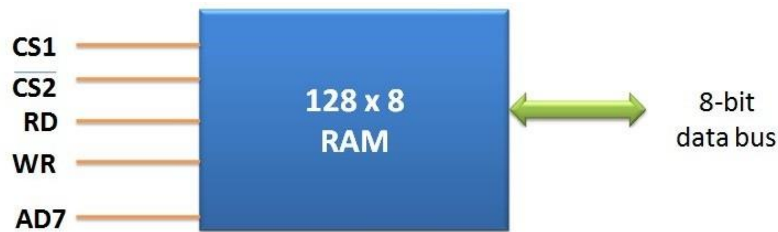
Semiconductor Memory Technologies:

Semiconductor random-access memories (RAMs) are available in a wide range of speeds.

Their cycle times range from 100 ns to less than 10 ns. Semiconductor memory is used in any electronics assembly that uses computer processing technology. The use of semiconductor memory has grown, and the size of these memory cards has increased as the need for larger and larger amounts of storage is needed.

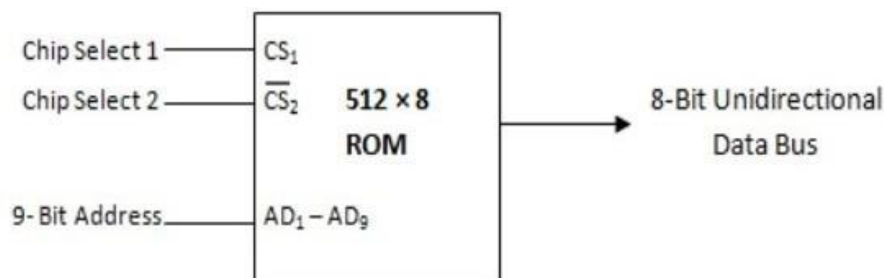
There are two main types or categories that can be used for semiconductor technology.

RAM - Random Access Memory: As the names suggest, the RAM or random access memory is a form of semiconductor memory technology that is used for reading and writing data in any order - in other words as it is required by the processor. It is used for such applications as the computer or processor memory where variables and other stored and are required on a random basis. Data is stored and read many times to and from this type of memory.



Block Diagram Representing 128 x 8 RAM
(Random Access Memory)

ROM - Read Only Memory: A ROM is a form of semiconductor memory technology used where the data is written once and then not changed. In view of this it is used where data needs to be stored permanently, even when the power is removed - many memory technologies lose the data once the power is removed. As a result, this type of semiconductor memory technology is widely used for storing programs and data that must survive when a computer or processor is powered down. For example the BIOS of a computer will be stored in ROM. As the name implies, data cannot be easily written to ROM. Depending on the technology used in the ROM, writing the data into the ROM initially may require special hardware. Although it is often possible to change the data, this gain requires special hardware to erase the data ready for new data to be written in.



The different memory types or memory technologies are detailed below:
DRAM: Dynamic RAM is a form of random access memory. DRAM uses a capacitor to store each bit of data, and the level of charge on each capacitor determines whether that bit is a logical 1 or 0. However these capacitors do not hold their charge indefinitely, and therefore the data needs to be refreshed periodically. As a result of this dynamic refreshing it gains its name of being a dynamic RAM. DRAM is the form of semiconductor memory that is often used in equipment including personal computers and workstations where it forms the main RAM for the computer.

EEPROM: This is an Electrically Erasable Programmable Read Only Memory. Data can be written to it and it can be erased using an electrical voltage. This is typically applied to an erase pin on the chip. Like other types of PROM, EEPROM retains the contents of the memory even when the power is turned off. Also like other types of ROM, EEPROM is not as fast as RAM.

EPROM: This is an Erasable Programmable Read Only Memory. This form of semiconductor memory can be programmed and then erased at a later time. This is normally achieved by exposing the silicon to ultraviolet light. To enable this to happen there is a circular window in the package of the EPROM to enable the light to reach the silicon of the chip. When the PROM is in use, this window is normally covered by a label, especially when the data may need to be preserved for an extended period. The PROM stores its data as a charge on a capacitor. There is a charge storage capacitor for each cell and this can be read repeatedly as required. However it is found that after many years the charge may leak away and the data may be lost. Nevertheless, this type of semiconductor memory used to be widely used in applications where a form of ROM was required, but where the data needed to be changed periodically, as in a development environment, or where quantities were low.

FLASH MEMORY: Flash memory may be considered as a development of EEPROM technology. Data can be written to it and it can be erased, although only in blocks, but data can be read on an individual cell basis. To erase and re-programme areas of the chip, programming voltages at levels that are available within electronic equipment are used. It is also non-volatile, and this makes it particularly useful. As a result Flash memory is widely used in many applications including memory cards for digital cameras, mobile phones, computer memory sticks and many other applications.

F-RAM: Ferroelectric RAM is a random-access memory technology that has many similarities to the standard DRAM technology. The major difference is that it incorporates a ferroelectric layer instead of the more usual dielectric layer and this provides its non-volatile capability. As it offers a non-volatile capability, F-RAM is a direct competitor to Flash.

MRAM: This is Magneto-resistive RAM, or Magnetic RAM. It is a non-volatile RAM memory technology that uses magnetic charges to store data instead of electric charges. Unlike technologies including DRAM, which require a constant flow of electricity to maintain the integrity of the data, MRAM retains data even when the power is removed. An additional advantage is that it only requires low power for active operation. As a result this technology could become a major player in the electronics industry now that production processes have been developed to enable it to be produced.

P-RAM / PCM: This type of semiconductor memory is known as Phase change Random Access Memory, P-RAM or just Phase Change memory, PCM. It is based around a phenomenon

where a form of chalcogenide glass changes is state or phase between an amorphous state (high resistance) and a polycrystalline state (low resistance). It is possible to detect the state of an individual cell and hence use this for data storage. Currently this type of memory has not been widely commercialized, but it is expected to be a competitor for flash memory.

PROM: This stands for Programmable Read Only Memory. It is a semiconductor memory which can only have data written to it once - the data written to it is permanent. These memories are bought in a blank format and they are programmed using a special PROM programmer. Typically a PROM will consist of an array of fuseable links some of which are "blown" during the programming process to provide the required data pattern.

SDRAM: Synchronous DRAM. This form of semiconductor memory can run at faster speeds than conventional DRAM. It is synchronised to the clock of the processor and is capable of keeping two sets of memory addresses open simultaneously. By transferring data alternately from one set of addresses, and then the other, SDRAM cuts down on the delays associated with non-synchronous RAM, which must close one address bank before opening the next.

SRAM: Static Random Access Memory. This form of semiconductor memory gains its name from the fact that, unlike DRAM, the data does not need to be refreshed dynamically. It is able to support faster read and write times than DRAM (typically 10 ns against 60 ns for DRAM), and in addition its cycle time is much shorter because it does not need to pause between accesses. However it consumes more power, is less dense and more expensive than DRAM. As a result of this it is normally used for caches,while DRAM is used as the main semiconductor memory technology.

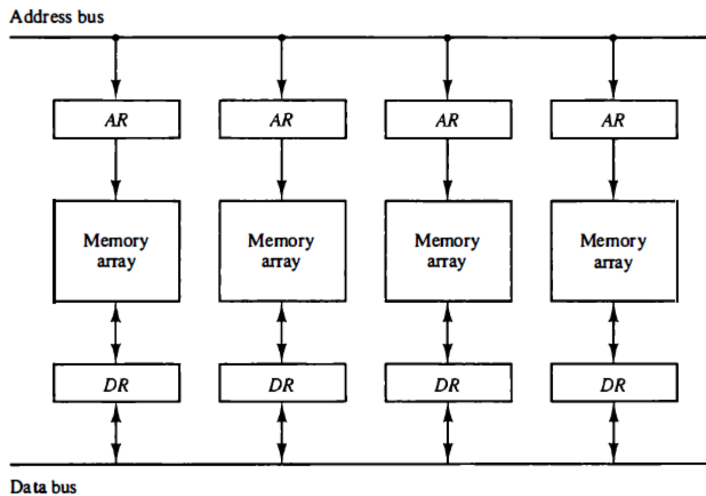
MEMORY ORGANIZATION

Memory Interleaving:

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.

Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure 9-13 shows a memory unit with four modules. Each memory array has its own address register AR and data register DR.

Figure 9-13 Multiple module memory organization.



The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.

The advantage of a modular memory is that it allows the use of a technique called interleaving.In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other.

Concept of Hierarchical Memory Organization

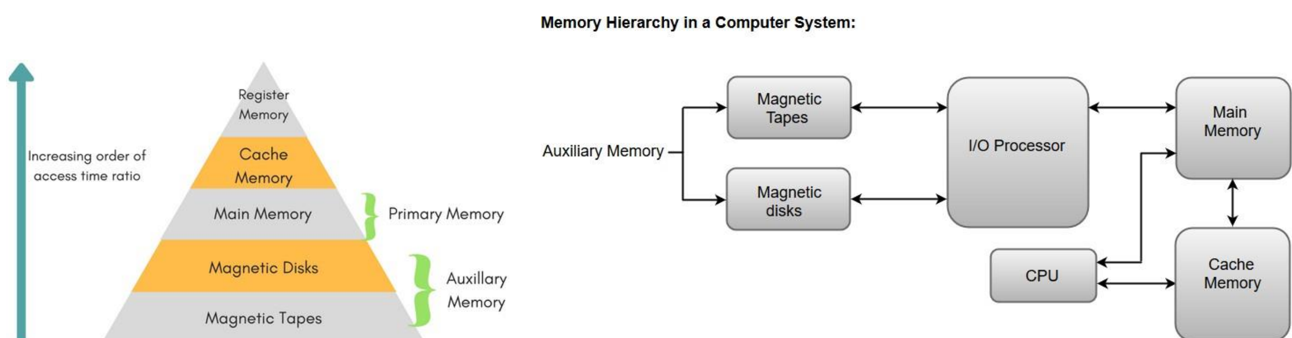
This Memory Hierarchy Design is divided into 2 main types:

External Memory or Secondary Memory

Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices whichare accessible by the processor via I/O Module.

Internal Memory or Primary Memory

Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by theprocessor.



Characteristics of Memory Hierarchy

Capacity:

It is the global volume of information the memory can store. As we move from top to bottom inthe Hierarchy, the capacity increases.

Access Time:

It is the time interval between the read/write request and the availability of the data. As we movefrom top to bottom in the Hierarchy, the access time increases.

Performance:

Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.

Cost per bit:

As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memoryis costlier than External Memory.

Cache Memories:

The cache is a small and very fast memory, interposed between the processor and the mainmemory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference.

Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other.

The cache memory can store a reasonable number of blocks at any given time, but this number issmall compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space forthe new block that contains the referenced word. The collection of rules for making this decision constitutes the cache‟s *replacement algorithm*.

Cache Hits

The processor does not need to know explicitly about the existence of the cache. It simply issuesRead andWrite requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache.

If it does, the Read orWrite operation is performed on the appropriate cache location. In this case, a *read*
or *write hit* is said to have occurred.

Cache Misses

A Read operation for a word that is not in the cache constitutes a *Read miss*. It causes the blockof words containing the requested word to be copied from the main memory into the cache.

Cache Mapping:

There are three different types of mapping used for the purpose of cache memory which are as follows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained as following below.

Direct mapping

The simplest way to determine cache locations in which to store memory blocks is the *direct- mapping* technique. In this technique, block *j* of the main memory maps onto block *j* modulo 128 of thecache, as depicted in Figure 8.16. Thus, whenever one of the main memory blocks 0*,* 128*,* 256*, . . .* is loaded into the cache, it is stored in cache block 0. Blocks 1*,* 129*,* 257*, . . .* are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full.

For example, instructions of a program may start in block 1 and continue in block 129, possiblyafter a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache isdetermined by its memory address. The memory address can be divided into three fields, as shown inFigure 8.16. The low-order 4 bits select one of 16 words in a block.

When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memoryand loaded into the cache.

The direct-mapping technique is easy to implement, but it is not very flexible.

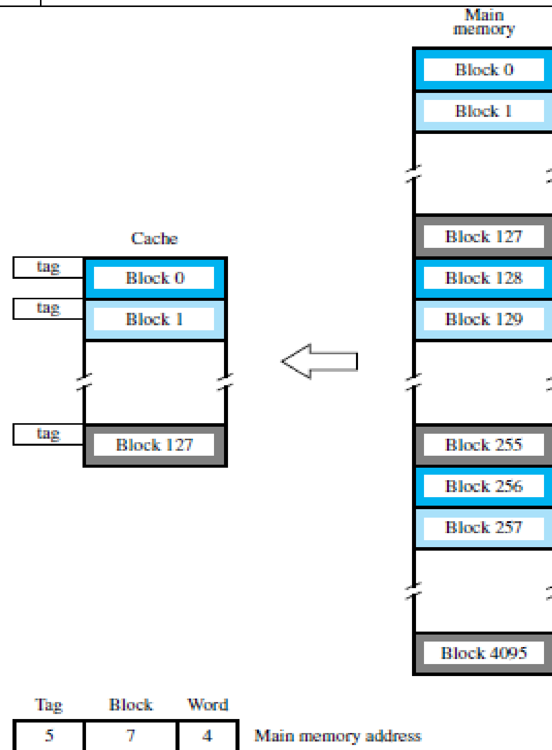


Figure 8.16 Direct-mapped cache.

Associative Mapping

In Associative mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in thecache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping* technique.

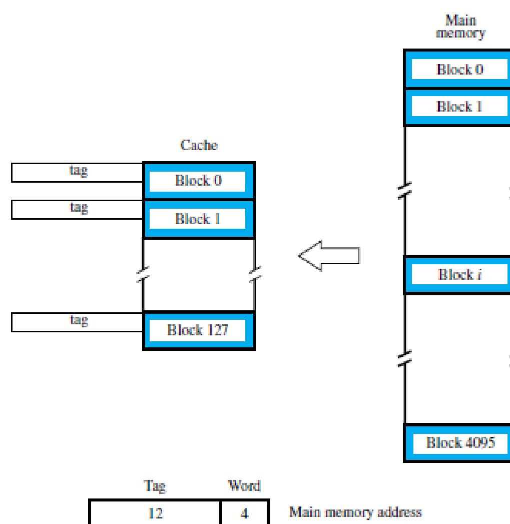


Figure 8.17 Associative-mapped cache.

It gives complete freedom in choosing the cache location in which to place the

memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, itreplaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced.

To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an
associative search.

Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory toreside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement.

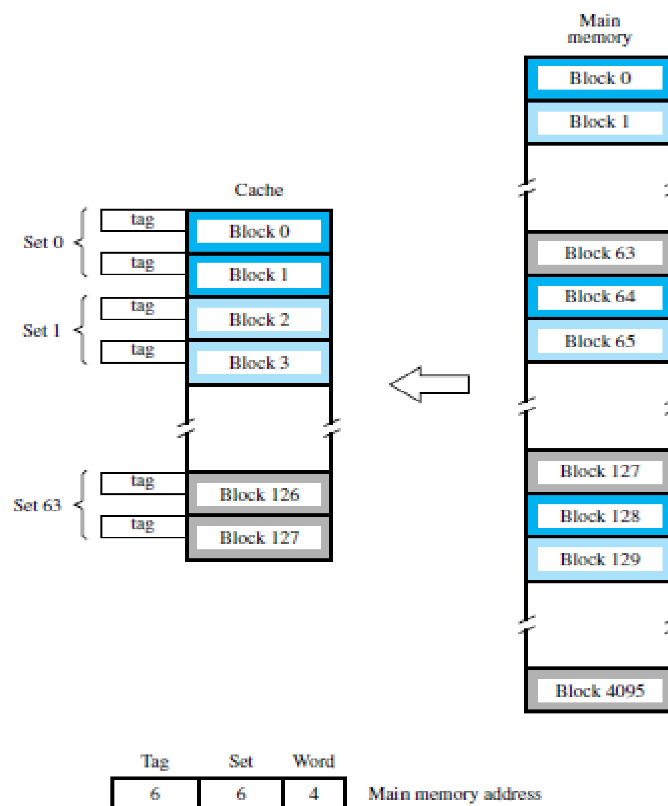


Figure 8.18 Set-associative-mapped cache with two blocks per set.

At the same time, the hardware cost is reduced by decreasing the size of the associative search.
An example of this *set-associative-mapping* technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they canoccupy either of the two block positions within this set.

Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to thetags of the two blocks of the set to check if the

desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure 8.18, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping.

Replacement Algorithms

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility.

When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite.

This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced.

The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore,when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the *least recently used* (LRU) block, and the technique iscalled the *LRU replacement algorithm*.

The LRU algorithm has been used extensively. Although it performs well for many accesspatterns, it can lead to poor performance in some cases.

Write Policies

The write operation is proceeding in 2 ways.

- Write-through protocol
- Write-back protocol

Write-through protocol:

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

Write-back protocol:

- This technique is to update only the cache location and to mark it as with associated flag bit called dirty/modified bit.
- The word in the main memory will be updated later, when the block containing thismarked word is to be removed from the cache to make room for a new block.

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

BASIC CONCEPTS OF MEMORY SYSTEM

The maximum size of the Main Memory (MM) that can be used in any computer is determined by its addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing upto 216 =64K memory locations. If a machine generates 32-bit addresses, it can access upto 2^{32} = 4G memory locations. This number represents the size of address space of the computer.

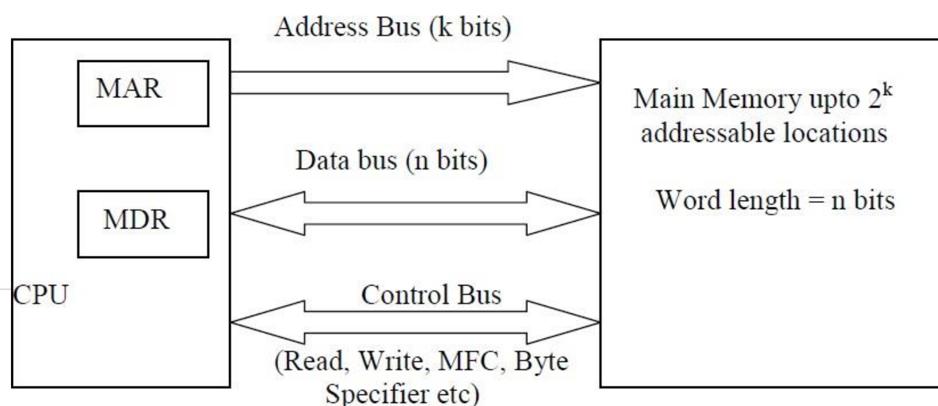
If the smallest addressable unit of information is a memory word, the machine is called word-addressable. If individual memory bytes are assigned distinct addresses, the computer is called byte-addressable. Most of the commercial machines are byte addressable. For example in a byte-addressable 32-bit computer, each memory word contains 4 bytes. A possible word-address assignment would be:
Word Address Byte Address

0	0 1 2 3
4	4 5 6 7
8	8 9 10 11
.	….. . ….. . …..

With the above structure a READ or WRITE may involve an entire memory word or it may involve only a byte. In the case of byte read, other bytes can also be read but ignored by the CPU. However, during a write cycle, the control circuitry of the MM must ensure that only the specified byte is altered. In this case, the higher-order 30 bits can specify the word and the lower-order 2 bits can specify the byte within the word.

CPU-Main Memory Connection – A block schematic: -
From the system standpoint, the Main Memory (MM) unit can be viewed as a “block box”. Data transfer between CPU and MM takes place through the use of two CPU registers, usually called MAR (Memory Address Register) and MDR (Memory Data Register). If MAR is K bits long and MDR is ‘n’ bits long, then the MM unit may contain upto 2^k addressable locations and
each location will be ‘n’ bits wide, while the word length is equal to ‘n’ bits. During a “memory cycle”, n bits of data may be transferred between the MM and CPU.

This transfer takes place over the processor bus, which has k address lines (address bus), n data lines (data bus) and control lines like Read, Write, Memory Function completed (MFC), Bytes specifiers etc (control bus). For a read operation, the CPU loads the address into MAR, set READ to 1 and sets other control signals if required. The data from the MM is loaded into MDR and MFC is set to 1. For a write operation, MAR, MDR are suitably loaded by the CPU, write is set to 1 and other control signals are set suitably. The MM control circuitry loads the data into appropriate locations and sets MFC to 1. This organization is shown in the following block schematic.



Address Bus (k bits) Main Memory upto 2k addressable locations Word length = n bits Data bus (n bits) Control Bus (Read, Write, MFC, Byte Specifier etc) MAR MDR CPU

Some Basic Concepts

Memory Access Times: - It is a useful measure of the speed of the memory unit. It is the time that elapses between the initiation of an operation and the completion of that operation (for example, the time between READ and MFC).

Memory Cycle Time :- It is an important measure of the memory system. It is the minimum time delay required between the initiations of two successive memory operations (for example, the time between two successive READ operations). The cycle time is usually slightly longer than the access time.

Random Access Memory (RAM):

A memory unit is called a Random Access Memory if any location can be accessed for a READ or WRITE operation in some fixed amount of time that is independent of the location's address. Main memory units are of this type. This distinguishes them from serial or partly serial access storage devices such as magnetic tapes and disks which are used as the secondary storage device.

Cache Memory:-

The CPU of a computer can usually process instructions and data faster than they can be fetched from compatibly priced main memory unit. Thus the memory cycle time become the bottleneck in the system. One way to reduce the memory access time is to use cache memory. This is a small and fast memory that is inserted between the larger, slower main memory and the CPU. This holds the currently active segments of a program and its data. Because of the locality of address references, the CPU can, most of the time, find the relevant information in the cache memory itself (cache hit) and infrequently needs access to the main memory (cache miss) with suitable size of the cache memory, cache hit rates of over 90% are possible leading to a cost-effective increase in the performance of the system.

Memory Interleaving: -

This technique divides the memory system into a number of memory modules and arranges addressing so that successive words in the address space are placed in different modules. When requests for memory access involve consecutive addresses, the access will be to different modules. Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.

Virtual Memory: -

In a virtual memory System, the address generated by the CPU is referred to as a virtual or logical address. The corresponding physical address can be different and the required mapping is implemented by a special memory control unit, often called the memory management unit. The mapping function itself may be changed during program execution according to system requirements.

Because of the distinction made between the logical (virtual) address space and the physical address space; while the former can be as large as the addressing capability of the CPU, the actual physical memory can be much smaller. Only the active portion of the virtual address space is mapped onto the physical memory and the rest of the virtual address space
is mapped onto the bulk storage device used. If the addressed information is in the Main Memory (MM), it is accessed and execution proceeds.

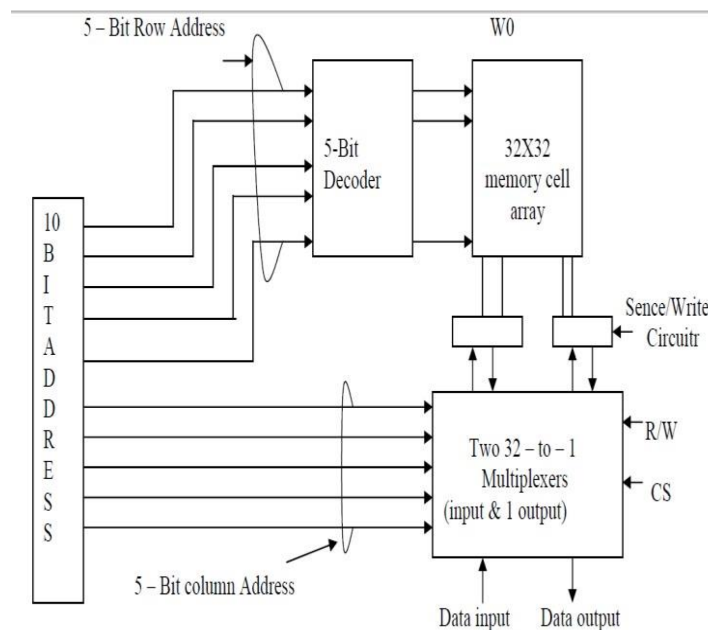
Otherwise, an exception is generated, in response to which the memory management unit transfers a contiguous block of words containing the desired word from the bulk storage unit to the MM, displacing some block that is currently inactive. If the memory is managed in such a way that, such transfers are required relatively infrequency (ie the CPU will generally find the required information in the MM), the virtual memory system can provide a reasonably good performance and succeed in creating an illusion of a large memory with a small, in expensive MM.

Internal Organization of Semiconductor Memory Chips:-

Memory chips are usually organized in the form of an array of cells, in which each cell is capable of storing one bit of information. A row of cells constitutes a memory word, and the cells of a row are connected to a common line referred to as the word line, and this line is driven by the address decoder on the chip. The cells in each column are connected to a sense/write circuit by two lines known as bit lines. The sense/write circuits are connected to the data input/output lines of the chip. During a READ operation, the Sense/Write circuits sense, or read, the information stored in the cells selected by a word line and transmit this information to the output lines. During a write operation, they receive input information and store it in the cells of the selected word.

The following figure shows such an organization of a memory chip consisting of 16 words of 8 bits each, which is usually referred to as a 16 x 8 organization.

The data input and the data output of each Sense/Write circuit are connected to a single bi-directional data line in order to reduce the number of pins required. One control line, the R/W (Read/Write) input is used a specify the required operation and another control line, the CS (Chip Select) input is used to select a given chip in a multichip memory system. This circuit requires 14 external connections, and allowing 2 pins for power supply and ground connections, can be manufactured in the form of a 16-pin chip. It can store 16 x 8 = 128 bits. Another type of organization for 1k x 1 format is shown below:



The 10-bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. One of these, selected by the column address, is connected to the external data lines by the input and

output multiplexers. This structure can store 1024 bits, can be implemented in a 16-pin chip.

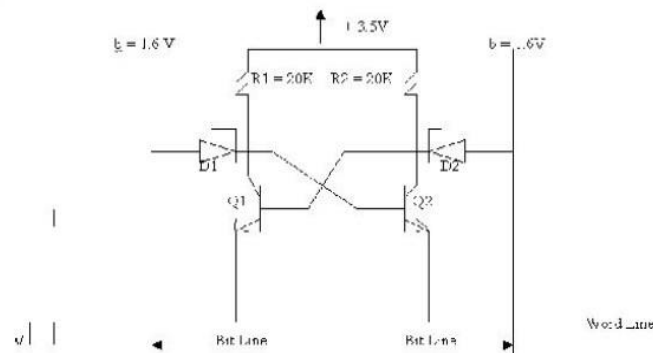
A Typical Memory Cell

Semiconductor memories may be divided into bipolar and MOS types. They may be compared as follows:

Characteristic	Bipolar	MOS
Power Dissipation	More	Less
Bit Density	Less	More
Impedance	Lower	Higher
Speed	More	Less

Bipolar Memory Cell

A typical bipolar storage cell is shown below:



Two transistor inverters connected to implement a basic flip-flop. The cell is connected to one word line and two bits lines as shown. Normally, the bit lines are kept at about 1.6V, and the word line is kept at a slightly higher voltage of about 2.5V. Under these conditions, the two diodes D1 and D2 are reverse biased. Thus, because no current flows through the diodes, the cell is isolated from the bit lines.

Read Operation:

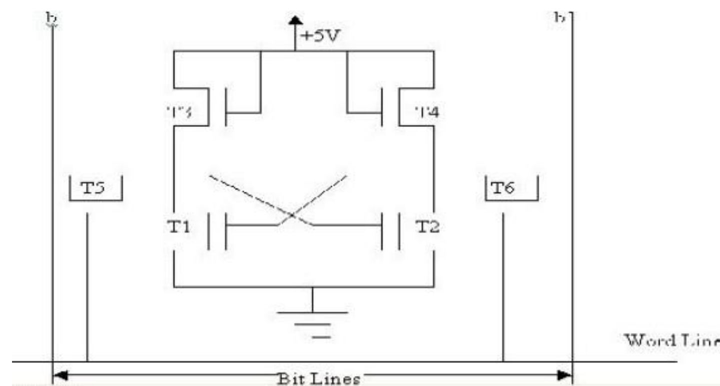
Let us assume the Q1 on and Q2 off represents a 1 to read the contents of a given cell, the voltage on the corresponding word line is reduced from 2.5 V to approximately 0.3 V. This causes one of the diodes D1 or D2 to become forward-biased, depending on whether the transistor Q1 or Q2 is conducting. As a result, current flows from bit line b when the cell is in the 1 state and from bit line b when the cell is in the 0 state. The Sense/Write circuit at the end of each pair of bit lines monitors the current on lines b and b' and sets the output bit line accordingly.

Write Operation:

While a given row of bits is selected, that is, while the voltage on the corresponding word line is 0.3V, the cells can be individually forced to either the 1 state by applying a positive voltage of about 3V to line b' or to the 0 state by driving line b. This function is performed by the Sense/Write circuit.

MOS Memory Cell:

MOS technology is used extensively in Main Memory Units. As in the case of bipolar memories, many MOS cell configurations are possible. The simplest of these is a flip-flop circuit. Two transistors T1 and T2 are connected to implement a flip-flop. Active pull-up to VCC is provided through T3 and T4. Transistors T5 and T6 act as switches that can be opened or closed under control of the word line. For a read operation, when the cell is selected, T5 or T6 is closed and the corresponding flow of current through b or b' is sensed by the sense/write circuits to set the output bit line accordingly. For a write operation, the bit is selected and a positive voltage is applied on the appropriate bit line, to store a 0 or 1. This configuration is shown below:



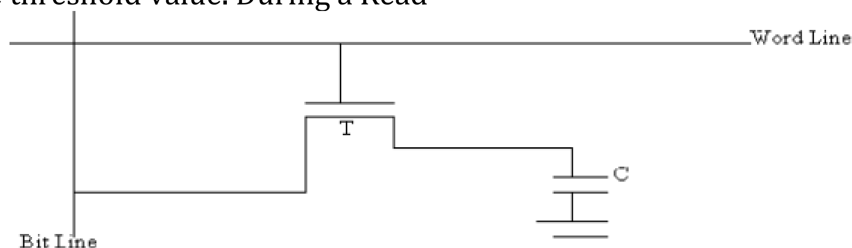
Static Memories Vs Dynamic Memories:-

Bipolar as well as MOS memory cells using a flip-flop like structure to store information can maintain the information as long as current flow to the cell is maintained. Such memories are called static memories. In contracts, Dynamic memories require not only the maintaining of a power supply, but also a periodic "refresh" to maintain the information stored in them. Dynamic memories can have very high bit densities and very lower power consumption relative to static memories and are thus generally used to realize the main memory unit.

Dynamic Memories:-

The basic idea of dynamic memory is that information is stored in the form of a charge on the capacitor. An example of a dynamic memory cell is shown below:

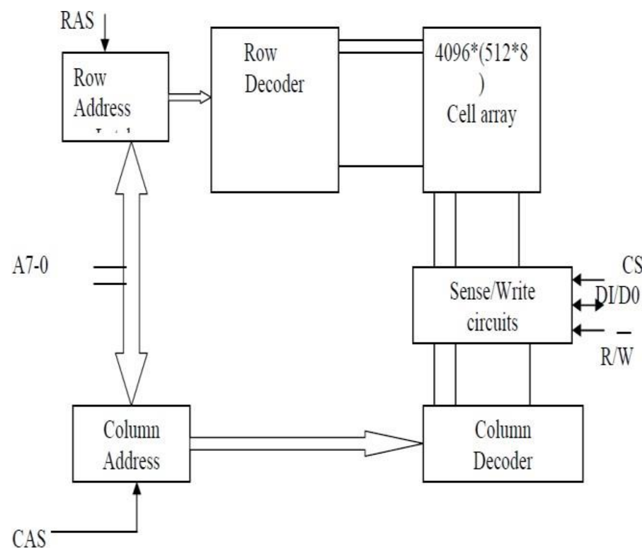
When the transistor T is turned on and an appropriate voltage is applied to the bit line, information is stored in the cell, in the form of a known amount of charge stored on the capacitor. After the transistor is turned off, the capacitor begins to discharge. This is caused by the capacitor's own leakage resistance and the very small amount of current that still flows through the transistor. Hence the data is read correctly only if is read before the charge on the capacitor drops below some threshold value. During a Read



operation, the bit line is placed in a high-impedance state, the transistor is turned on and a sense circuit connected to the bit line is used to determine whether the charge on the capacitor is above or below the threshold value. During such a Read, the charge on the capacitor is restored to its

original value and thus the cell is refreshed with every read operation.

Typical Organization of a Dynamic Memory Chip:-



A typical organization of a 64k x 1 dynamic memory chip is shown below:

The cells are organized in the form of a square array such that the high-and lower-order 8 bits of the 16-bit address constitute the row and column addresses of a cell, respectively. In order to reduce the number of pins needed for external connections, the row and column address are multiplexed on 8 pins.

To access a cell, the row address is applied first. It is loaded into the row address latch in response to a single pulse on the Row Address Strobe (RAS) input. This selects a row of cells. Now, the column address is applied to the address pins and is loaded into the column address latch under the control of the Column Address Strobe (CAS) input and this address selects the appropriate sense/write circuit. If the R/W signal indicates a Read operation, the output of the selected circuit is transferred to the data output. Do. For a write operation, the data on the DI line is used to overwrite the cell selected.

It is important to note that the application of a row address causes all the cells on the corresponding row to be read and refreshed during both Read and Write operations. To ensure that the contents of a dynamic memory are maintained, each row of cells must be addressed periodically, typically once every two milliseconds. A Refresh circuit performs this function. Some dynamic memory chips in-corporate a refresh facility the chips themselves and hence they appear as static memories to the user! such chips are often referred to as Pseudostatic.

Another feature available on many dynamic memory chips is that once the row address is loaded, successive locations can be accessed by loading only column addresses.

Such block transfers can be carried out typically at a rate that is double that for transfers involving random addresses. Such a feature is useful when memory access follow a regular pattern, for example, in a graphics terminal Because of their high density and low cost, dynamic memories are widely used in the main memory units of computers. Commercially available chips range in size from 1k to 4M bits or more, and are available in various organizations like 64k x 1, 16k x 4, 1MB x 1 etc.

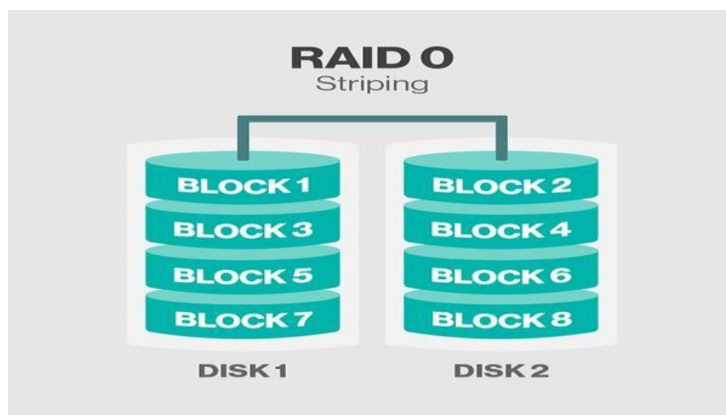
RAID (Redundant Array of Independent Disks)

RAID (*redundant array of independent disks*; originally *redundant array of inexpensive disks*) provides a way of storing the same data in different places (thus, redundantly) on multiple hard disks (though not all RAID levels provide redundancy). By placing data on multiple disks, input/output (I/O) operations can overlap in a balanced way, improving performance. Since multiple disks increase the mean time between failures (MTBF), storing data redundantly also increases fault tolerance.

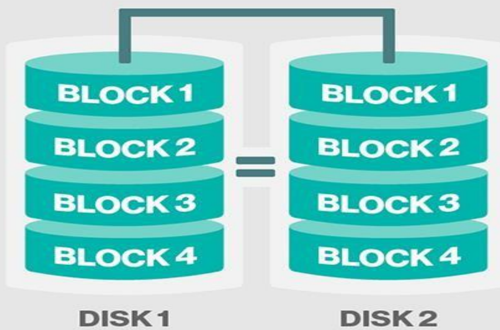
RAID arrays appear to the operating system (OS) as a single logical hard disk. RAID employs the technique of disk mirroring or disk striping, which involves partitioning each drive's storage space into units ranging from a sector (512 bytes) up to several megabytes. The stripes of all the disks are interleaved and addressed in order.

In a single-user system where large records, such as medical or other scientific images,are stored, the stripes are typically set up to be small (perhaps 512 bytes) so that a single record spans all disks and can be accessed quickly by reading all disks at the same time.
In a multi-user system, better performance requires establishing a stripe wide enough to hold the typical or maximum size record. This allows overlapped disk I/O across drives.

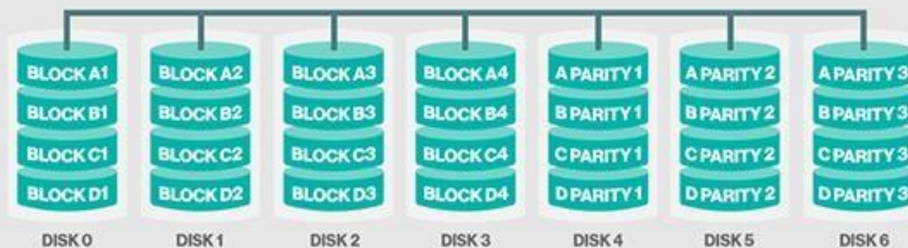
Standard RAID levels
RAID 0: This configuration has striping but no redundancy of data. It offers the best performance but no fault-tolerance.



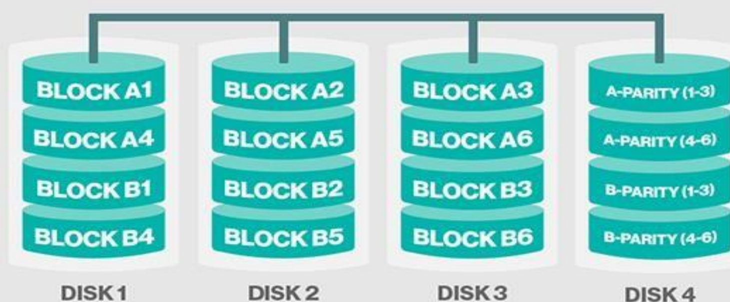
RAID 1: Also known as *disk mirroring*, this configuration consists of at least two drives that duplicate the storage of data. There is no striping. Read performance is improved since either disk can be read at the same time. Write performance is the same as for single disk storage.



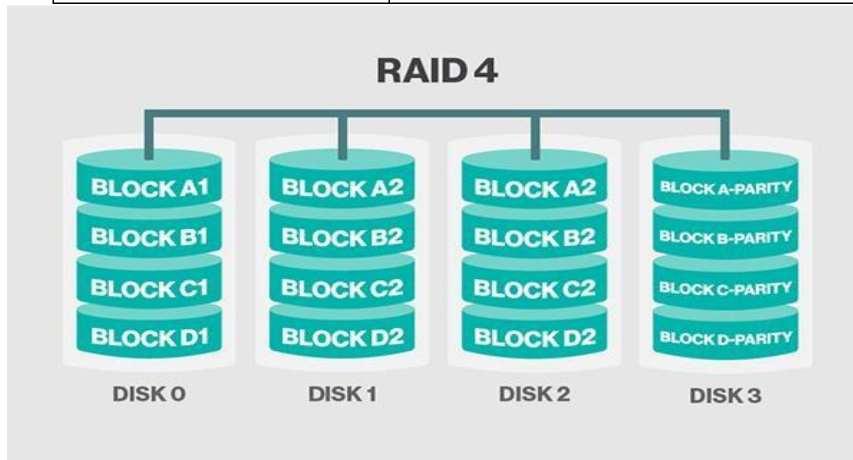
RAID 2: This configuration uses striping across disks with some disks storing error checking and correcting (ECC) information. It has no advantage over RAID 3 and is no longer used.



RAID 3: This technique uses striping and dedicates one drive to storing parity information. The embedded ECC information is used to detect errors. Data recovery is accomplished by calculating the exclusive OR (XOR) of the information recorded on the other drives. Since an I/O operation addresses all drives at the same time, RAID 3 cannot overlap I/O. For this reason, RAID 3 is best for single-user systems with long record applications.

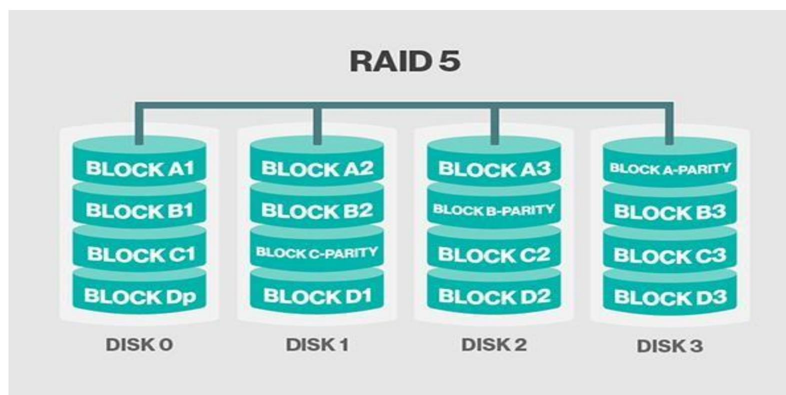


RAID 4: This level uses large stripes, which means you can read records from any single drive. This allows you to use overlapped I/O for read operations. Since all write operations have to update the parity drive, no I/O overlapping is possible. RAID 4 offers no advantage over RAID 5.

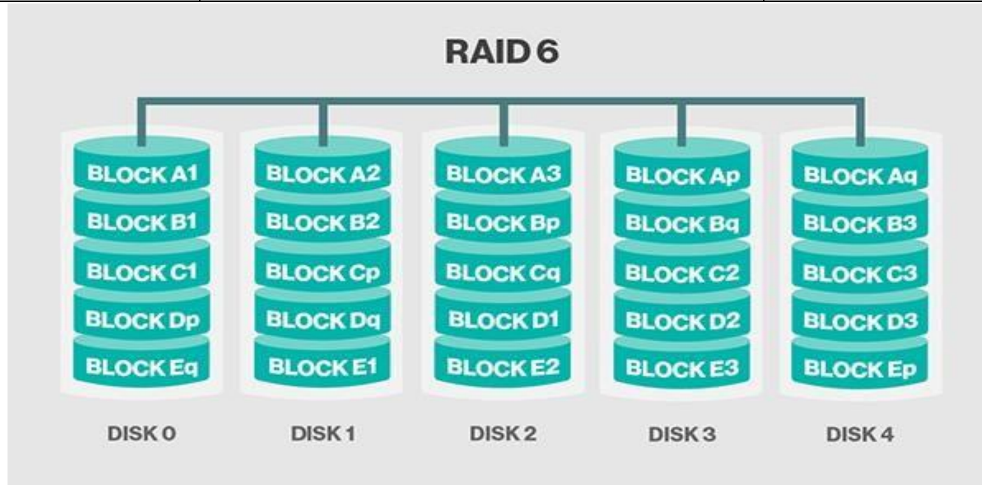


RAID 5: This level is based on block-level striping with parity. The parity information is striped across each drive, allowing the array to function even if one drive were to fail. The array's architecture allows read and write operations to span multiple drives. This results in performance that is usually better than that of a single drive, but not as high as that of a RAID 0 array. RAID 5 requires at least three disks, but it is often recommended to use at least five disks for performance reasons.

RAID 5 arrays are generally considered to be a poor choice for use on write-intensive systems because of the performance impact associated with writing parity information. When a disk does fail, it can take a long time to rebuild a RAID 5 array. Performance is usually degraded during the rebuild time and the array is vulnerable to an additional disk failure until the rebuild is complete.



RAID 6: This technique is similar to RAID 5 but includes a second parity scheme that is distributed across the drives in the array. The use of additional parity allows the array to continue to function even if two disks fail simultaneously. However, this extra protection comes at a cost. RAID 6 arrays have a higher cost per gigabyte (GB) and often have slower write performance than RAID 5 arrays.



Direct Memory Access (DMA)

DMA stands for "**Direct Memory Access**" and is a method of transferring data from the computer's RAM to another part of the computer without processing it using the CPU. While most data that is input or output from your computer is processed by the CPU, some data does not require processing, or can be processed by another device.

In these situations, DMA can save processing time and is a more efficient way to move data from the computer's memory to other devices. In order for devices to use direct memory access, they must be assigned to a DMA channel. Each type of port on a computer has a set of DMA channels that can be assigned to each connected device. For example, a PCI controller and a hard drive controller each have their own set of DMA channels.

For example, a sound card may need to access data stored in the computer's RAM, but since it can process the data itself, it may use DMA to bypass the CPU. Video cards that support DMA can also access the system memory and process graphics without needing the CPU. Ultra DMA hard drives use DMA to transfer data faster than previous hard drives that required the data to first be run through the CPU.

An alternative to DMA is the Programmed Input/Output (PIO) interface in which all data transmitted between devices goes through the processor. A newer protocol for the ATAIIDE interface is Ultra DMA, which provides a burst data transfer rate up to 33 mbps. Hard drives that come with Ultra DMAl33 also support PIO modes 1, 3, and 4, and multiword DMA mode 2 at 16.6 mbps.

DMA Transfer Types

Memory To Memory Transfer

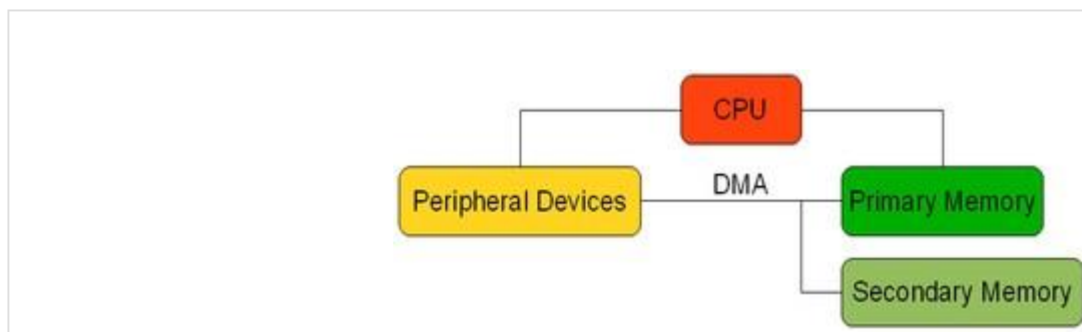
In this mode block of data from one memory address is moved to another memory address. In this mode current address register of channel 0 is used to point the source address and the current address register of channel is used to point the destination address in the first transfer cycle, data byte from the source address is loaded in the temporary register of the DMA controller and in the next transfer cycle the data from the temporary register is stored in the memory pointed by destination address.

After each data transfer current address registers are decremented or incremented according to current settings. The channel 1 current word count register is also decremented by 1 after each data transfer. When the word count of channel 1 goes to FFFFH, a TC is generated which activates EOP output terminating the DMA service.

Auto initialize
In this mode, during the initialization the base address and word count registers are loaded simultaneously with the current address and word count registers by the microprocessor.The address and the count in the base registers remain unchanged throughout the DMA service.
After the first block transfer i.e. after the activation of the EOP signal, the original values of the current address and current word count registers are automatically restored from the base address and base word count register of that channel. After auto initialization the channel is ready to perform another DMA service, without CPU intervention.

DMA Controller
*The co*ntroller is integrated into the processor board and manages all DMA data transfers. Transferring data between system memory and a 110 device requires two steps. Data goes from the sending device to the DMA controller and then to the receiving device. The microprocessor gives the DMA controller the location, destination, and amount of data that is to be transferred. Then the DMA controller transfers the data, allowing the microprocessor to continue with other processing tasks.

When a device needs to use the Micro Channel bus to send or receive data, it competes with all the other devices that are trying to gain control of the bus. This process is known as arbitration. The DMA controller does not arbitrate for control of the BUS instead; the I/O device that is sending or receiving data (the DMA slave) participates in arbitration. It is the DMA controller, however, that takes control of the bus when the central arbitration control point grants the DMA slave's request.
DMA vs. interrupts vs. polling



A diagram showing the position of the DMA in relation to peripheral devices, the CPU and internal memory

Works in the background without CPU intervention
This speed up data transfer and CPU speed
The DMA is used for moving large files since it would take too much of CPU capacity

Interrupt Systems
Interrupts take up time of the CPU
they work by asking for the use of the CPU by sending the interrupt to which the CPU responds
o*Note: In order to save time the CPU does not check if it has to respond*
Interrupts are used when a task has to be performed immediately

Polling
Polling requires the CPU to actively monitor the process
The major advantage is that the polling can be adjusted to the needs of the device
polling is a low level process since the peripheral device is not in need of a quick response

Characteristics of Multiprocessors

A multiprocessor system is an interconnection of two or more CPU, with memory and input-output equipment. As defined earlier, multiprocessors can be put under MIMD category. The term multiprocessor is sometimes confused with the term multi computers. Though both support concurrent operations, there is an important difference between a system with multiple computers and a system with multiple processors.

In a multi computers system, there are multiple computers, with their own operating systems, which communicate with each other, if needed, through communication links. A multiprocessor system, on the other hand, is controlled by a single operating system, which coordinate the activities of the various processors, either through shared memory or inter processor messages.

The advantages of multiprocessor systems are:
Increased reliability because of redundancy in processors
Increased throughput because of execution of multiple jobs in parallel portions of the same job in parallel

A single job can be divided into independent tasks, either manually by the programmer, or by the compiler, which finds the portions of the program that are data independent, and can be executed in parallel. The multiprocessors are further classified into two groups depending on the way their memory is organized. The processors with shared memory are called tightly coupled or shared memory processors.

The information in these processors is shared through the common memory. Each of the processors can also have their local memory too. The other class of multiprocessors is loosely coupled or distributed memory multi-processors. In this, each processor has their own private memory, and they share information with each other through interconnection switching scheme or message passing.

The principal characteristic of a multiprocessor is its ability to share a set of main memory and some I/O devices. This sharing is possible through some physical connections between them called the interconnection structures.

Inter processor Arbitration

Computer system needs buses to facilitate the transfer of information between its various components. For example, even in a uniprocessor system, if the CPU has to access a memory location, it sends the address of the memory location on the address bus. This address activates a memory chip. The CPU then sends a red signal through the control bus, in the response of which the memory puts the data on the address

bus.

This address activates a memory chip. The CPU then sends a read signal through the control bus, in the response of which the memory puts the data on the data bus. Similarly, in amultiprocessor system, if any processor has to read a memory location from the shared areas,it follows the similar routine.

There are buses that transfer data between the CPUs and memory. These are called memory buses. An I/O bus is used to transfer data to and from input and output devices. A bus that connects major components in a multiprocessor system, such as CPUs, I/Os, and memory is called system bus. A processor, in a multiprocessor system, requests the access ofa component through the system bus.

In case there is no processor accessing the bus at that time, it is given then control of the bus immediately. If there is a second processor utilizing the bus, then this processor hasto wait for the bus to be freed. If at any time, there is request for the services of the bus by more than one processor, then the arbitration is performed to resolve the conflict. A bus controller is placed between the local bus and the system bus to handle this.

Inter processor Communication and Synchronization
In a multiprocessor system, it becomes very necessary, that there be proper communication protocol between the various processors. In a shared memory multiprocessor system, a common area in the memory is provided, in which all the messages that need to be communicated to other processors are written.
A proper synchronization is also needed whenever there is a race of two or more processors for shared resources like I/O resources. The operating system in this case is given the task of allocating the resources to the various processors in a way, that at any time not more than one processor use the resource.

A very common problem that can occur when two or more resources are trying to access a resource which can be modified. For example processor 1 and 2 are simultaneously trying to access memory location 100. Say the processor 1 is writing on to the location while processor 2 is reading it. The chances are that processor 2 will end up reading erroneous data. Such kind of resources which need to be protected from simultaneous access of more than one processors are called critical sections. The following assumptions are made regarding the critical sections:
Mutual exclusion: At most one processor can be in a critical section at a time
-Termination : The critical section is executed in a finite time
Fair scheduling: A process attempting to enter the critical section will eventually do so in afinite time.
A binary value called a semaphore is usually used to indicate whether a processor is currentlyExecuting the critical section.

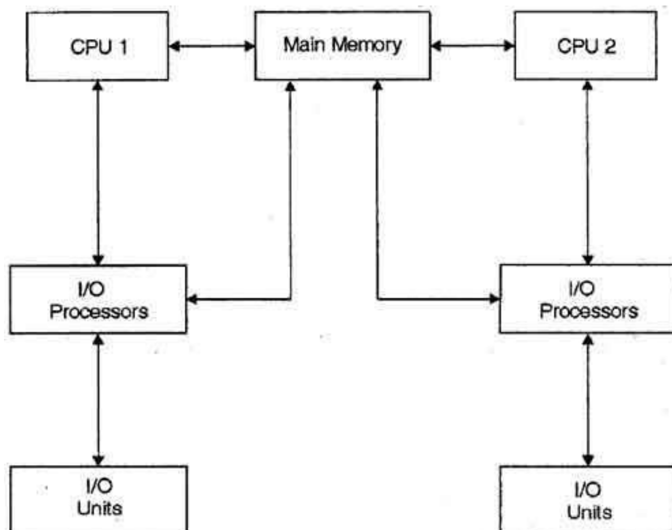
Cache Coherence

As discussed in unit 2, cache memories are high speed buffers which are inserted between the processor and the main memory to capture those portions of the contents of main memory which are currently in use. These memories are five to ten times faster than main memories, and therefore, reduce the overall access time. In a multiprocessor system, with shared memory, each processor has its own set of private cache.

Multiple copies of the cache are provided with each processor to reduce the access time. Each processor, whenever accesses the shared memory, also updates its private cache. This introduced the problem of cache coherence, which may result in data inconsistency. That is, several copies of the same data may exist in different caches at any given time.

For example, let us assume there are two processors x and y. Both have the same copy of the cache. Processor x, produces data 'a' which is to be consumed by processor y. Processor update the value of 'a' in its own private copy of the cache. As it does not have any access tothe private copy of cache of processor y, the processor y continues to use the variable 'a' with old value, unless it is informed of the change.

Thus, in such kind of situations if the system is to perform correctly, every updation in the cache should be informed to all the processors, so that they can make necessary changes in their private copies of the cache.



Introduction of Cache Memory

Basic Cache Structure

Processors are generally able to perform operations on operands faster than the access time of large capacity main memory. Though semiconductor memory which can operate at speeds comparable with the operation of the processor exists, it is not economical to provide all the main memory with very high speed semiconductor memory. The problem can be alleviated by introducing a small block of high speed memory called a cache between the main memory and the processor.

The idea of cache memories is similar to virtual memory in that some active portion of a low-speed memory is stored in duplicate in a higher- speed cache memory. When a memory request is generated, the request is first presented to the cache memory, and if the cache cannot respond, the request is then presented to main memory.

The difference between cache and virtual memory is a matter of implementation; the two notions are conceptually the same because they both rely on the correlation properties observed in sequences of address references. Cache implementations are totally different from virtual memory implementation because of the speed requirements of cache.

We define a cache miss to be a reference to a item that is not resident in cache, but is resident in main memory. The corresponding concept for cache memories is page fault, which is defined to be a reference to a page in virtual memory that is not resident in main memory. For cache misses, the fast memory is cache and the slow memory is main memory. For page faults the fast memory is main memory, and the slow memory is auxiliary memory.

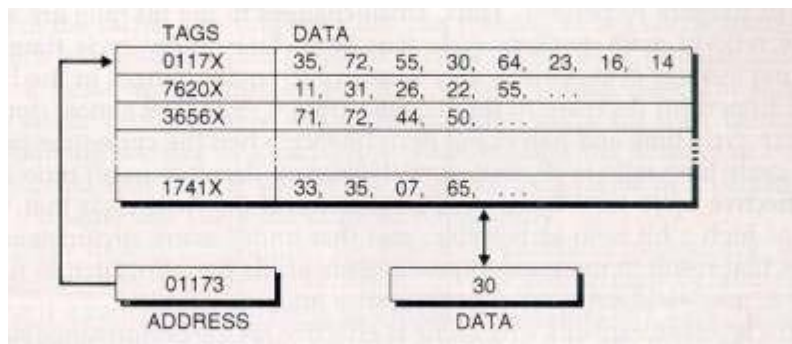


Fig 1. A cache-memory reference. The tag 0117X matches address 01173, so the cache returns the item in the position X=3 of the matchedblock

Figure 1 shows the structure of a typical cache memory. Each reference to a cell in memory is presented to the cache. The cache searches its directory of address tags shown in the figure to see if the item is in the cache. If the item is not in the cache, a miss occurs.

For READ operations that cause a cache miss, the item is retrieved from main memory and copied into the cache. During the short period availablebefore the main-memory operation is complete, some other item in cache is removed form the cache to make rood for the new item.

The cache-replacement decision is critical; a good replacement algorithm can yield somewhat higher performance than can a bad replacement algorithm. The effective cycle-time of a cache memory (t_{eff}) is the averageof cache-memory cycle time (t_{cache}) and main-memory cycle time (t_{main}), where the probabilities in the averaging process are the probabilities of hits and misses.

If we consider only READ operations, then a formula for the averagecycle-time is:

$$t_{eff} = t_{cache} + (1 - h) t_{main}$$

where h is the probability of a cache hit (sometimes called the hit rate), thequantity (1 - h), which is the probability of a miss, is know as the *miss rate*.

In Fig.1 we show an item in the cache surrounded by nearby items, all of which are moved into and out of the cache together. We call such a groupof data a *block* of the cache.

Cache Memory Organizations

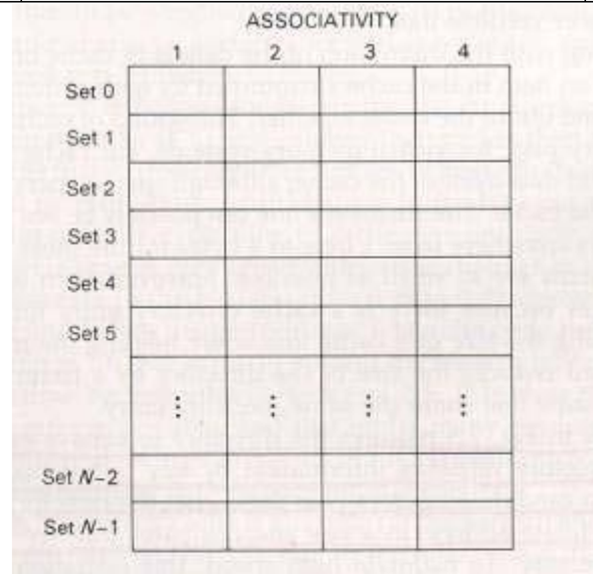


fig.2 The logical organization of a four-way set-associate cache

Fig.2 shows a conceptual implementation of a cache memory. This system is called set associative because the cache is partitioned into distinct sets of blocks, ad each set contains a small fixed number of blocks. The sets are represented by the rows in the figure. In this case, the cache has N sets, and each set contains four blocks. When an access occurs to this cache, the cache controller does not search the entire cache looking for a match.
Instead, the controller maps the address to a particular set of the cache and searches only the set for a match.

If the block is in the cache, it is guaranteed to be in the set that is searched. Hence, if the block is not in that set, the block is not present in the cache, and the cache controller searches no further. Because the search is conducted over four blocks, the cache is said to be four-way set associative or, equivalently, to have an associativity of four.

Fig.2 is only one example, there are various ways that a cache can be arranged internally to store the cached data. In all cases, the processor reference the cache with the main memory address of the data it wants. Hence each cache organization must use this address to find the data in the cache if it is stored there, or to indicate to the processor when a miss has occurred. The problem of mapping the information held in the main memory into the cache must be totally implemented in hardware to achieve improvements in the system operation. Various strategies are possible.

- **Fully associative mapping**

Perhaps the most obvious way of relating cached data to the main memory address is to store both memory address and data together in the cache. This the fully associative mapping approach. A fully associative cache requires the cache to be composed of associative memory holding both the memory address and the data for each cached line. The incoming memory address is simultaneously compared with all stored addresses using the internal logic of the associative memory, as shown in Fig.3. If a match is fund, the corresponding data is read out. Single words form anywhere within the main memory could be held in the cache, if the associative part of the cache is capable of holding a full address

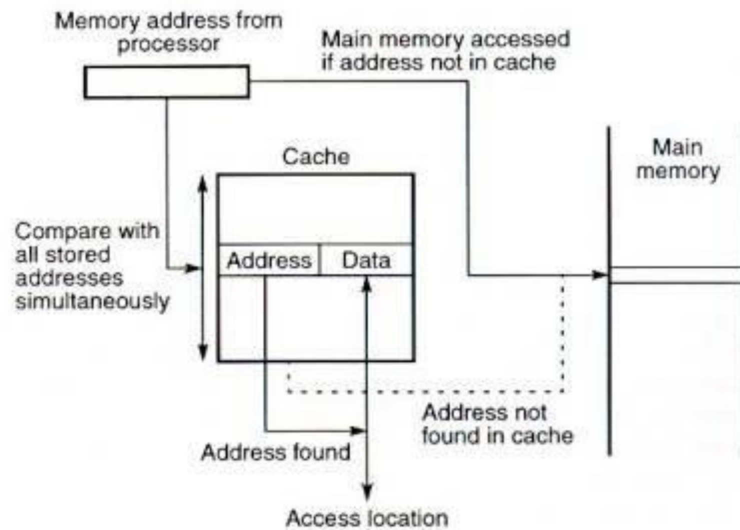


Fig.3 Cache with fully associative mapping

In all organizations, the data can be more than one word, i.e., a block of consecutive locations to take advantage of spatial locality. In Fig.4 aline constitutes four words, each word being 4 bytes. The least significant part of the address selects the particular byte, the next part selects the word, and the remaining bits form the address compared to the address in the cache. The whole line can be transferred to and from the cache in one transaction if there are sufficient data paths between the main memory and the cache. With only one data word path, the words of the line have to be transferred in separate transactions.

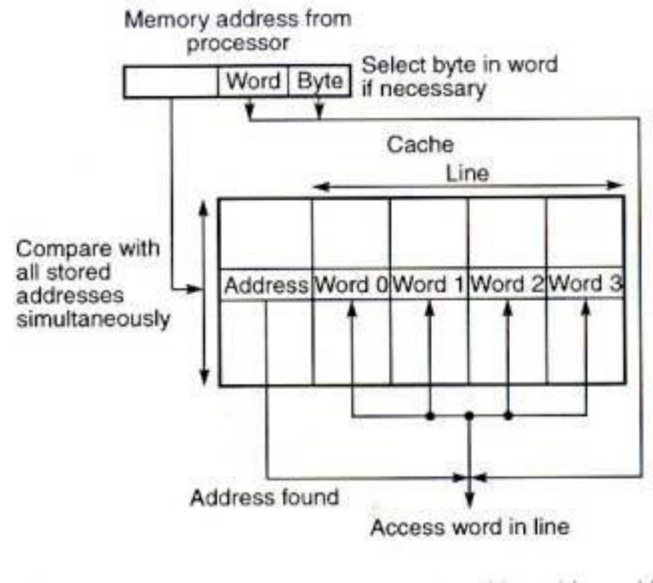


Fig.5 Fully associative mapped cache with multi-word lines

The fully associate mapping cache gives the greatest flexibility of holding combinations of blocks in the cache and minimum conflict for a given sized cache, but is also the most expensive, due to the cost of the associative memory. It requires a replacement algorithm to select a block to remove upon a miss and the algorithm must be implemented in hardware to maintain a high speed of operation. The fully associative cache can only be formed economically with a moderate size capacity.
Microprocessors with small internal caches often employ the fully associative mechanism.

Direct mapping

The fully associative cache is expensive to implement because of requiring a comparator with each cache location, effectively a special type of memory. In direct mapping, the cache consists of normal high speed random access memory, and each location in the cache holds the data, at an address in the cache given by the lower significant bits of the main memory address. This enables the block to be selected directly from the lower significant bits of the memory address. The remaining higher significant bits of the address are stored in the cache with the data to complete the identification of the cached data.

Consider the example shown in Fig.5. The address from the processor is divided into tow fields, a tag and an index. The tag consists of the higher significant bits of the address, which are stored with the data. The index is the lower significant bits of the address used to address the cache.

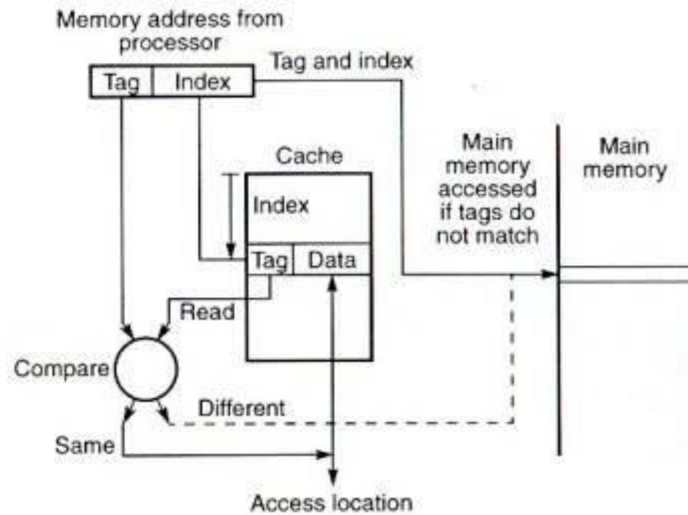


Fig.5 Cache with direct mapping

When the memory is referenced, the index is first used to access a word in the cache. Then the tag stored in the accessed word is read and compared with the tag in the address. If the two tags are the same, indicating that the word is the one required, access is made to the addressed cache word. However, if the tags are not the same, indicating that the required word is not in the cache, reference is made to the main memory to find it. For a memory read operation, the word is then transferred into the cache where it is accessed. It is possible to pass the information to the cache and the processor simultaneously, i.e., to read-through the cache, on a miss. The cache location is altered for a write operation. The main memory may be altered at the same time (write-through) or later.

Fig.6. shows the direct mapped cache with a line consisting of more than one word. The main memory address is composed of a tag, an index, and a word within a line. All the words within a line in the cache have the same stored tag. The index part to the address is used to access the cache and the stored tag is compared with required tag address. For a read operation, if the tags are the same the word within the block is selected for transfer to the

processor. If the tags are not the same, the block containing the required word is first transferred to the cache.

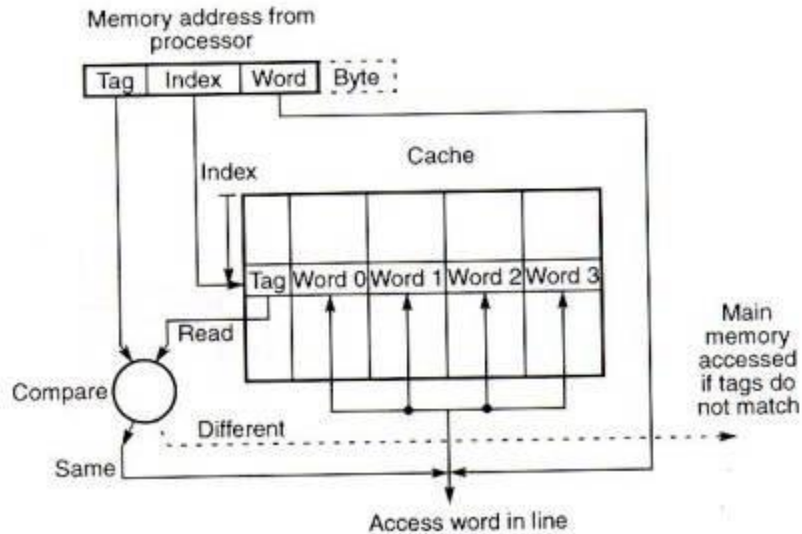


Fig.6 Direct mapped cache with a multi-word block

In direct mapping, the corresponding blocks with the same index in the main memory will map into the same block in the cache, and hence only blocks with different indices canbe in the cache at the same time. A replacement algorithm is unnecessary, since there is only one allowable location for each incoming block. Efficient replacement relies on the low probability of lines with the same index being required. However there are such occurrences, for example,when two data vectors are stored starting at the same indexand pairs of elements need to processed together. To gain the greatest performance, data arrays and vectors need to bestored in a manner which minimizes the conflicts in processing pairs of elements. Fig.6 shows the lower bits of the processor address used to address the cache location directly. It is possible to introduce a mapping function between the address index and the cache index so that they are not the same.

Set-associative mapping

In the direct scheme, all words stored in the cache must have different indices. The tags may be the same or different. In the fully associative scheme, blocks can displace any other block and can be placed anywhere, but

the cost of the fully associative memories operate relatively slowly.

Set-associative mapping allows a limited number of blocks, with the same index and different tags, in the cache and can therefore be considered as a compromise between a fully associative cache and a direct mapped cache. The organization is shown in Fig.7. The cache is divided into "sets" of blocks. A four-way set associative cache would have four blocks in each set. The number of blocks in a set is know as the associativity or set size. Each block in each set has a stored tag which, together with the index, completes the identification of the block. First, the index of the address from the processor is used to access the set.
Then, comparators are used to compare all tags of the selected set with the incoming tag. If a match is found, the corresponding location is accessed, other wise, as before, an access to the main memory is made.

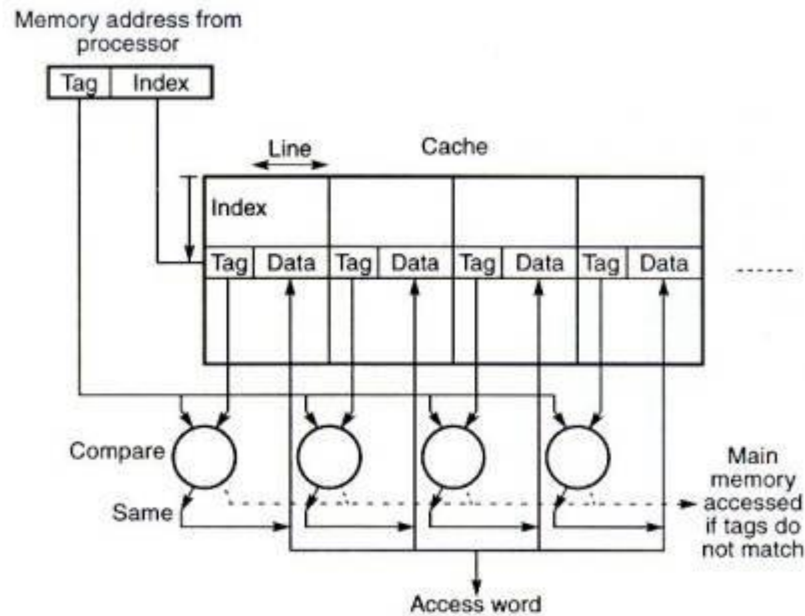


Fig.7 Cache with set-associative mapping

The tag address bits are always chosen to be the most significant bits of the full address, the block address bits are the next significant bits and the word/byte address bits form the least significant bits as this spreads out consecutive man memory blocks throughout consecutive sets in the cache.
This addressing format is known as bit selection and is used by all known systems. In a set-associative cache it would

be possible to have the set address bits as the most significant bits of the address and the block address bits as the next significant, with the word within the block as the least significant bits, or with the block address bits as the least significant bits and the word within the block as the middle bits.

Notice that the association between the stored tags and the incoming tag is done using comparators and can be shared for each associative search, and all the information, tags and data, can be stored in ordinary random access memory. The number of comparators required in the set-associative cache is given by the number of blocks in a set, not the number of blocks in all, as in a fully associative memory. The set can be selected quickly and all the blocks of the set can be read out simultaneously with the tags before waiting for the tag comparisons to be made. After a tag has been identified, the corresponding block can be selected.

The replacement algorithm for set-associative mapping need only consider the lines in one set, as the choice of set is predetermined by the index in the address. Hence, with two blocks in each set, for example, only one additional bitis necessary in each set to identify the block to replace.

Sector mapping

In sector mapping, the main memory and the cache are bothdivided into sectors; each sector is composed of a number of blocks. Any sector in the main memory can map into anysector in the cache and a tag is stored with each sector in the cache to identify the main memory sector address.
However, a complete sector is not transferred to the cache or back to the main memory as one unit. Instead, individual blocks are transferred as required. On cache sector miss, the required block of the sector is transferred into a specific location within one sector. The sector location in the cacheis selected and all the other existing blocks in the sector in the cache are from a previous sector.

Sector mapping might be regarded as a fully associative mapping scheme with valid bits, as in some microprocessor caches. Each block in the fully associative mapped cache corresponds to a sector, and each byte corresponds to a "sector block".

Cache Performance

The performance of a cache can be quantified in terms of the hit and miss rates, the cost of a hit, and the miss penalty, where a cache hit is a memoryaccess that finds data in the cache and a cache miss is one that does not.

When reading, the cost of a cache hit is roughly the time to access an entryin the cache. The miss penalty is the additional cost of replacing a cache line with one containing the desired data.

(Access time) = (hit cost) + (miss rate)*(miss penalty)

=(Fast memory access time) + (miss rate)*(slow memory access time)

Note that the approximation is an underestimate - control costs have beenleft out. Also note that only one word is being loaded from the faster memory while a whole cache block's worth of data is being loaded from the slower memory.

Since the speeds of the actual memory used will be improving ``independently'', most effort in cache design is spent on fast control and decreasing the miss rates. We can classify misses into three categories, compulsory misses, capacity misses and conflict misses. Compulsory misses are when data is loaded into the cache for the first time (e.g. program startup) and are unavoidable. Capacity misses are when data is reloaded because the cache is not large enough to hold all the data no matter how we organize the data (i.e. even if we changed the hash functionand made it omniscient). All other misses are conflict misses - there is theoretically enough space in the cache to avoid the miss but our fast hash function caused a miss anyway.

Fetch and write mechanism

Fetch policy

We can identify three strategies for fetching bytes or blocksfrom the main memory to the cache, namely:

Demand fetch

Which is the fetching a block when it is needed and is not already in the cache, i.e. to fetch the required block on a miss. This strategy is the simplest and requires no additional hardware or tags in the cache recording the references, except to identify the block in the cache to be replaced.

Prefetch

Which is fetching blocks before they are requested. A simple prefetch strategy is to prefetch the (i+1)th block when the ith blockis initially referenced on the expectation that it is likely to be needed if the ith block is needed. On the simple prefetch strategy, not all first references will induce a miss, as some will be to prefetched blocks.

Selective fetch

Which is the policy of not always fetching blocks, dependent upon some defined criterion, and in these cases using the main memory rather than the cache to hold the information. For example, shared writable data might be easier to maintain if it is always kept in the main memory and not passed to a cache for access, especially in multi-processor systems. Cache systems need to be designed so that the processor can access the main memory directly and bypass the cache. Individual locations could be tagged as non-cacheable.

Instruction and data caches

The basic stored program computer provides for one main memory for holding both program instructions and program data. The cache can be organized in the same fashion, with the cache holding both program instructions and data. This is called a unified cache. We also can separate the cache into two parts: data cache and instruction (code) cache. Thegeneral arrangement of separate caches is shown in fig.8. Often the cache will be integrated inside the processor chip.

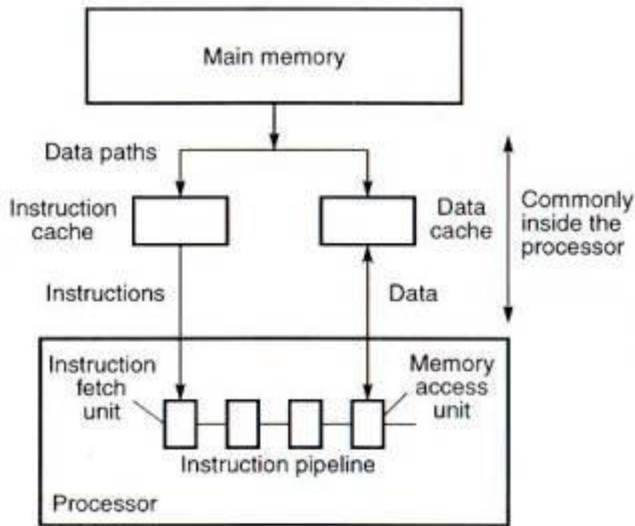


Fig.8 Separate instruction and data caches

Write operations

As reading the required word in the cache does not affect the cache contents, there can be no discrepancy between thecache word and the copy held in the main memory after a memory read instruction. However, in general, writing can occur to cache words and it is possible that the cache word and copy held in the main memory may be different. It is necessary to keep the cache and the main memory copy identical if input/output transfers operate on the main memory contents, or if multiple processors operate on the main memory, as in a shared memory multiple processor system.

If we ignore the overhead of maintaining consistency and the time for writing data back to the main memory, then the average access time is given by the previous equation, i.e. $t_{eff} = t_{cache} + (1 - h) t_{main}$, assuming that all accesses are first made to the cache. The average access time including write operations will add additional time to this equation that will depend upon the mechanism used to maintain dataconsistency.

There are two principal alternative mechanisms to update the main memory, namely the *write-through* mechanism and the *write-back* mechanism.

Write-through mechanism

In the write-though mechanism, every write operation to the cache is repeated to the main memory, normally at the same time. The additional write operation to the main memory will, of course, take much longer than to the cache and will dominate the access time for write operations. The average access time of write-through with transfers from main memory to the cache on all misses (read and write) isgiven by:

$$t_a = t_{cache} + (1 - h)\, t_{trans} + w(t_{main} - t_{cache})$$
$$= (1 - w)\, t_{cache} + (1 - h)\, t_{trans} + w t_{main}$$

Where t_{trans} = time to transfer block to cache, assuming the whole block must be transferred together

W = fraction of write references.

The term (t_{main} - t_{cache}) is the additional time to write the word to main memory whether a hit or a miss has occurred, given that both cache and main memory write operation occur simultaneously but the main memory write operation must complete before any subsequent cache read/write operation can be proceed. If the size of the block matches the external data path size, a whole block can be transferred in one transaction and $t_{trans} = t_{main}$.

On a cache miss, a block could be transferred from the main memory to the cache whether the miss was caused by a write or by a read operation. The term *allocate on write* isused to describe a policy of bringing a word/block from themain memory into the cache for a write operation. In write-through, fetch on write transfers are often not done on a miss, i.e., a *Non- allocate on write* policy. The information will be written back to the main memory but not kept in thecache.

The write-through scheme can be enhanced by incorporating buffers, as shown in Fig.9, to hold information to be written back to the main memory, freeingthe cache for subsequent accesses.

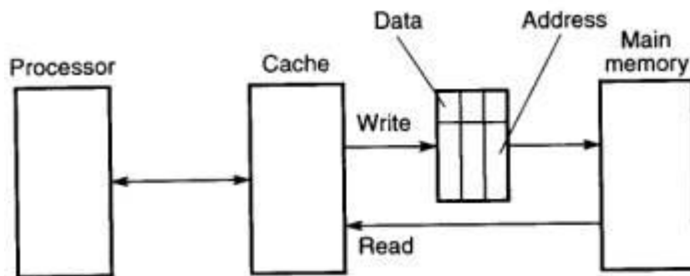


Fig.9 Cache with write buffer

For write-through, each item to be written back to the mainmemory is held in a buffer together with the correspondingmain memory address if the transfer cannot be made immediately.

Immediate writing to main memory when new values are generated ensures that the most recent values are held in the main memory and hence that any device or processor accessing the main memory should obtain the most recent values immediately, thus avoiding the need for complicated consistency mechanisms. There will be latency before the main memory has been updated, and the cache and main memory values are not consistent during this period.

Write-back mechanism

In the write-back mechanism, the write operation to the main memory is only done at block replacement time. At this time, the block displaced by the incoming block might be written back to the main memory irrespective of whether the block has been altered. The policy is known as simple write-back, and leads to an average access time of:

$t_a = t_{cache} + (1 - h) t_{trans} + (1 - h) t_{trans}$

Where one $(1 - h) t_{trans}$ term is due to fetching a block frommemory and the other $(1 - h) t_{trans}$ term is due to writing back a block. Write-back normally handles write misses asallocate on write, as opposed to write-through, which oftenhandles write misses as Non-allocate on write.

The write-back mechanism usually only writes back lines that have been altered. To implement this policy, a 1-bit tagis associated with each cache line and is set whenever the

block is altered. At replacement time, the tags are examined to determine whether it is necessary to write the block back to the main memory. The average access time now becomes:

$t_a = t_{cache} + (1 - h)\, t_{trans} + w_b(1 - h)\, t_{trans}$ where w_b is the probability that a block has been altered

(fraction of blocks altered). The probability that a block has been altered could be as high as the probability of write references, w, but is likely to be much less, as more than one write reference to the same block is likely and some references to the same byte/word within the block are likely. However, under this policy the complete block is written back, even if only one word in the block has been altered, and thus the policy results in more traffic than is necessary, especially for memory data paths narrower than a line, but still there is usually less memory traffic than write-through, which causes every alteration to be recorded in the main memory. The write-back scheme can also be enhanced by incorporating buffers to hold information to be written back to the main memory, just as is possible and normally done with write-through.

Replacement policy

When the required word of a block is not held in the cache, we have seen that it is necessary to transfer the block from the main memory into the cache, displacing an existing block if the cache is full. Except for direct mapping, which does not allow a replacement algorithm, the existing block in the cache is chosen by a *replacement algorithm*. The replacement mechanism must be implemented totally in hardware, preferably such that the selection can be made completely during the main memory cycle for fetching the new block. Ideally, the block replaced will not be needed again in the future. However, such future events cannot be known and a decision has to be made based upon facts that are known at the time.

Random replacement algorithm

Perhaps the easiest replacement algorithm to implement is a pseudo-random replacement algorithm. A true random replacement algorithm would select a block to replace in a totally random order, with no regard to memory references or previous selections; practical random replacement

algorithms can approximate this algorithm in one of several ways. For example, one counter for the whole cache could be incremented at intervals (for example after each clock cycle, or after each reference, irrespective of whether it is a hit or a miss). The value held in the counter identifies the block in the cache (if fully associative) or the block in the set if it is a set-associative cache. The counter should have sufficient bits to identify any block. For a fully associative cache, an n-bit counter is necessary if there are 2^n words in the cache. For a four-way set-associative cache, one 2-bit counter would be sufficient, together with logic to increment the counter.

First-in first-out replacement algorithm

The first-in first-out replacement algorithm removes the block that has been in the cache for the longest time. The first-in first-out algorithm would naturally be implemented with a first-in first-out queue of block address, but can be more easily implemented with counters, only one counter for a fully associative cache or one counter for each set in a set-associative cache, each with a sufficient number of bits to identify the block.

Least recently used algorithm for a cache

In the *least recently* used (LRU) algorithm, the block which has not been referenced for the longest time is removed from the cache. Only those blocks in the cache are considered. The word "recently" comes about because the block is not the least used, as this is likely to be back in memory. It is the least used of those blocks in the cache, and all of those are likely to have been recently used otherwise they would not be in the cache. The least recently used (LRU) algorithm is popular for cache systems and can be implemented fully when the number of blocks involved is small. There are several ways the algorithm can be implemented in hardware for a cache, these include:

Counters

In the counter implementation, a counter is associated with each block. A simple implementation would be to increment each counter at regular intervals and to reset a counter when the associated line had been referenced.
Hence the value in each counter would indicate the age of a

block since last referenced. The block with the largest age would be replaced at replacement time.

Register stack

In the register stack implementation, a set of n-bit registers is formed, one for each block in the set to be considered.
The most recently used block is recorded at the "top" of thestack and the least recently used block at the bottom.
Actually, the set of registers does not form a conventional stack, as both ends and internal values are accessible. The value held in one register is passed to the next register under certain conditions. When a block is referenced, starting at the top of the stack, starting at the top of the stack, the values held in the registers are shifted one place towards the bottom of the stack until a register is found to hold the same value as the incoming block identification. Subsequent registers are not shifted. The top register is loaded with the incoming block identification. This has the effect of moving the contents of the register holding the incoming block number to the top of the stack. This logic isfairly substantial and slow, and not really a practical solution.

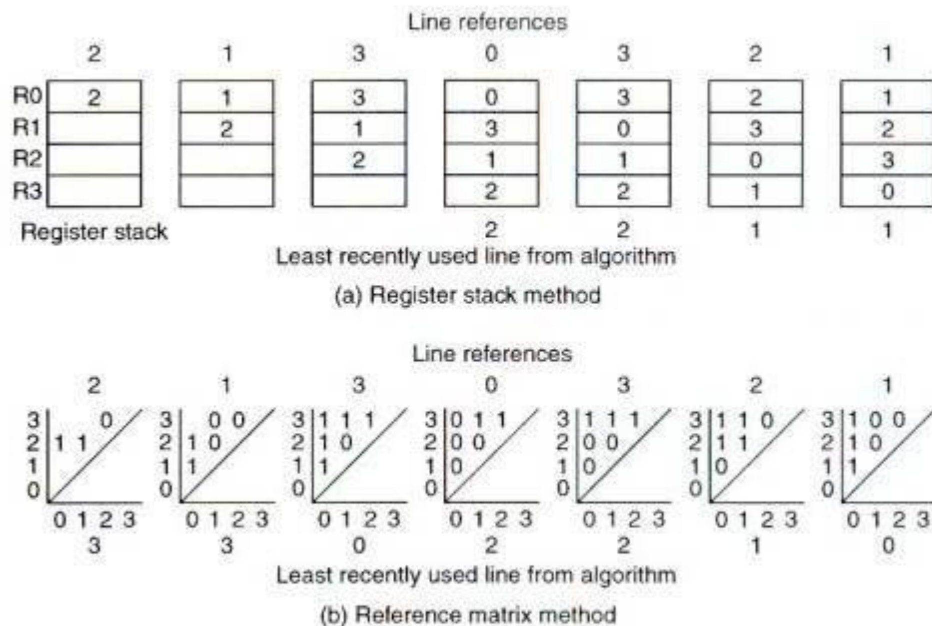


Fig.10 Least recently used replacement algorithm implementation

Reference matrix

The reference matrix method centers around a matrix of status bits. There is more than one version of the method. In one version (Smith, 1982), the upper triangular matrix of a B X B matrix is formed without the diagonal, if there are B blocks to consider. The triangular matrix has (B * (B - 1))/2 bits. When the ith block is referenced, all the bits in the ith row of the matrix are set to 1 and then all the bits in the ith column are set to 0. The least recently used block is one which has all 0's in its row and all 1's in its column, which can be detected easily by logic. The method is demonstrated in Fig.10 for B = 4 and the reference sequence 2, 1, 3, 0, 3, 2, 1, ..., together with the values thatwould be obtained using a register stack.

Approximate methods.

When the number of blocks to consider increases above about four to eight, approximate methods are necessary forthe LRU algorithm. Fig.11 shows a two-stage approximation method with eight blocks, which is applicable to any replacement algorithm. The eight blocksin Fig.11 are divided into four pairs, and each pair has one status bit to indicate the most/least recently used block in the pair (simply set or reset by reference to each block).
The least recently used replacement algorithm now only considers the four pairs. Six status bits are necessary (using the reference matrix) to identify the least recently used pair which, together with the status bit of the pair, identifies the least recently used block of a pair.

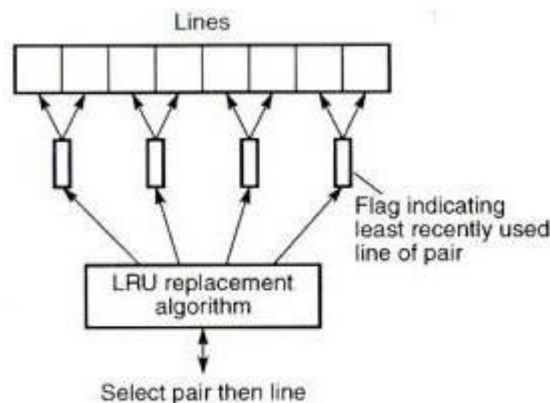


Fig.11 Two-stage replacement algorithm

The method can be extended to further levels. For example, sixteen blocks can be divided into four groups, each group having two pairs. One status bit can be associated with each pair, identifying the block in the pair, and another with each group, identifying the group in a pair of groups. A true least recently used algorithm is applied to the groups. In fact, the scheme could be taken to its logical conclusion of extending to a full binary tree. Fig.12 gives an example.

Here, there are four blocks in a set. One status bit, B0, specifies which half o the blocks are most/least recently used. Two more bits, B1 and B2, specify which block of pairs is most/least recently used. Every time a cache block is referenced (or loaded on a miss), the status bits are updated. For example, if block L2 is referenced, B2 is set to a 0 to indicate that L2 is the most recently used of the pair L2 and L3. B0 is set to a 1 to indicate that L2/L3 is the most recently used of the four blocks, L0, L1, L2 and L3. To identify the line to replace on a miss, the status bits are examined. If B0 = 0, then the block is either L0 or L1. If then B1 = 0, it is L0.

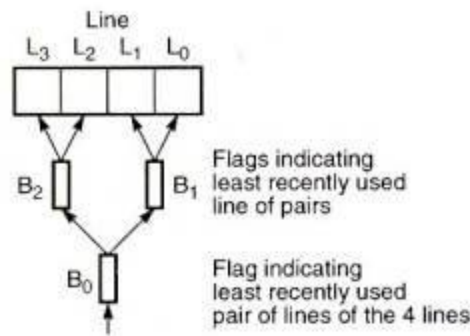


Fig.12 Replacement algorithm using a tree selection

Second-level caches

When the cache is integrated into the processor, it will be impossible to increase its size should the performance not be sufficient. In any case, increasing the size of the cache may create a slower cache. As an alternative, which has become very popular, a second larger cache can be introduced between the first cache and the main memory as shown in Fig.13. This "second-level" cache is sometimes called a secondary cache.

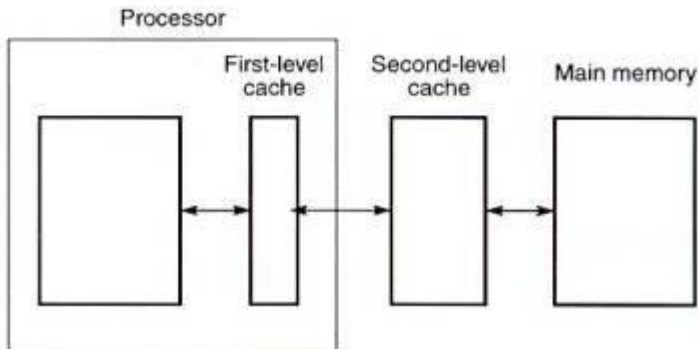


Fig.13 Two-level caches

On a memory reference, the processor will access the first-level cache. If the information is not found there (a first-level cache miss occurs), the second-level cache will be accessed. If it is not in the second cache (a second-level cache miss occurs), then the main memory must be accessed. Memory locations will be transferred to the second-level cache and then to the first-level cache, so that two copies of a memory location will exist in the cache system at least initially, i.e., locations cached in the second-level cache also exist in the first-level cache. This is known as the Principle of Inclusion. (Of course the copies of locations in the second-level cache will never be needed as they will be found in the first-level cache.) Whether this continues will depend upon the replacement and write policies. The replacement policy practiced in both caches would normally be the least recently used algorithm. Normally write-through will be practiced between the caches, which will maintain duplicate copies. The block size of the second-level cache will be at least the same if not larger than the block size of the first-level cache, because otherwise on a first-level cache miss, more than one second-level cache line would need to be transferred into the first-level cache block.

Optimizing the data cache performance
------- Taking advantage of locality in matrix multiplication

When we dealing with multiple arrays, with some arrays accessed by rows and some by columns. Storing the arrays row-by-row or column-by- column does not solve the problem because both rows and columns are used in every iteration of the loop. We must bring the same data into the cache again and again if the cache is not large enough to hold all the data, which is a waste. We will use a matrix multiplication (C = A.B, where A, B, and C are respectively m x p, p x n, and m x n matrices) as an example to show how to utilize the locality to improve cache performance.

Principle of Locality

Since code is generally executed sequentially, virtually all programs repeat sections of code and repeatedly access the same or nearby data. This characteristic is embodied in the Principle of Locality, which has been found empirically to be obeyed by most programs. It applies to both instruction references and data references, though it is more likely in instruction references. It has two main aspects:

Temporal locality (locality in time) -- individual locations, once referenced, are likely to be referenced again in the near future.
Spatial locality (locality in space) - references, including the next location, are likely to be near the last reference.

Temporal locality is found in instruction loops, data stacks and variable accesses. Spatial locality describes the characteristic that programs access a number of distinct regions. Sequential locality describes sequential locations being referenced and is a main attribute of program construction. It can also be seen in data accesses, as data item are often stored in sequential locations.

Taking advantage of temporal locality

When instructions are formed into loops which are executed many times, the length of a loop is usually quite small. Therefore once a cache is loaded with loops of instructions from the main memory, the instructions are used more than once before new instructions are required from the main memory. The same situation applies to data; data is repeatedly accessed. Suppose the reference is

repeated n times in all during a program loop and after thefirst reference, the location is always found in the cache, then the average access time would be:

$t_a = (n*t_{cache} + t_{main})/n = t_{cache} + t_{main}/n$

where n = number of references. As n increases, the average access time decreases. The increase in speed will, of course, depend upon the program. Some programs might have a large amount of temporal locality, while others have less. We can do some optimization about this.

Taking advantage of spatial locality

To take advantage of spatial locality, we will transfer not just one byte or word from the main memory to the cache (and vice versa) but a series of sequential locations called a block. We have assumed that it is necessary to reference the cache before a reference is make to the main memory tofetch a word, and it is usual to look into the cache first to see if the information is held there.

Data Blocking

For the matrix multiplication C = A.B, if we made code as below:

```
For (I = 0; I < m; I++)
For (J = 0; J < n; J = J++) {R = 0;
For (K = 0; K < p; K++)
R = R + A[I][K] * B[K][J];C[I][J] = R; }
```

The two inner loops read all p by n elements of B and access the same p elements in a row of A repeatedly, and write one row of n elements of C. The number of capacity misses clearly depends on the dimension parameters: m, n, p and the size of the cache. If the cache can hold all three metrics, then all is well, provided there are no cache conflicts. In theworst case, there would be (2*m*n*p + m*n) words read form memory for m*n*p operations.

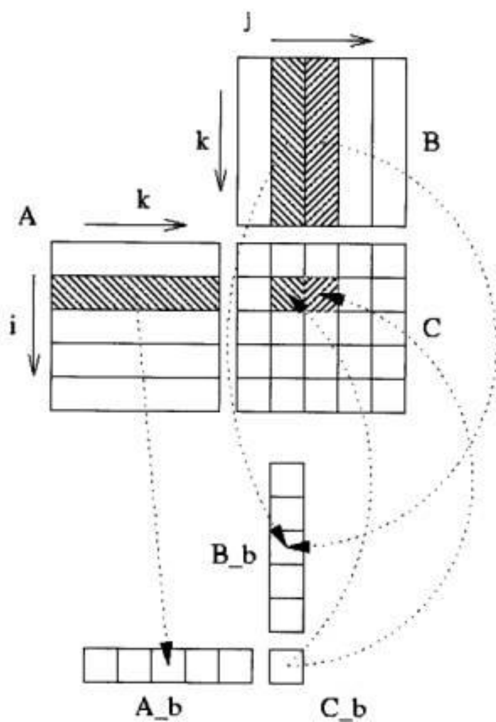
To enhance the cache performance if it is not big enough, we use an optimization technique: *blocking*. The block method for this matrixproduct consist of:

Split result matrix C into blocks $C_{I,J}$ of size N_b x N_b, each blocks isconstructed into a continuous array C_b which is then copied back into the right $C_{I,J}$.
Matrices A and B are spit into panels A_I and B_J of size (N_b x p) and(p x N_b) each panel is copied into continuous arrays A_b and Bb. The choice of N_b must ensure that C_b, A_b and B_b fit into one level of cache, usually L_2 cache.

Then we rewrite the code as:

For (I = 0; I < m/N_b; I++){$A_b = A_I$;
For (J = 0; J < n/N_b; J++) {$B_b = B_J$; C_b = 0;
For (K = 0; K < p/N_b; K++)
$C_b = C_b + A_{bK}*B_{Kb}$;
$C_{I,J} = C_b$; }}   here "=" means assignment for matrix

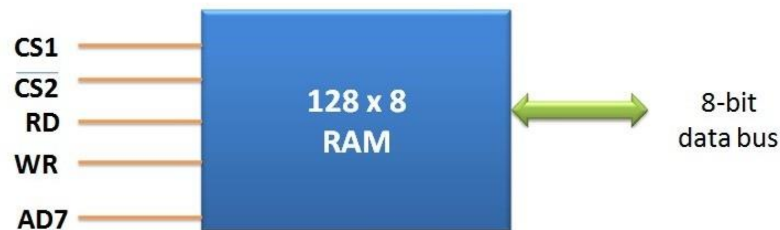
We suppose for simplicity that N_b divides m, n and p. The figure below may help you in understanding operations performed on blocks. In the case of previous algorithm matrix A is loaded only one time into cache compared to the n times access of the original one, while matrix B is stillaccessed m times. This simple block method greatly reduce memory access and real codes may choose by looking at matrix size which loop structure (ijk vs. jik) is best appropriate and if some matrix operand fits totally into cache.



In the previous we do not talk about L_1 cache use. In fact L_1 will be generally too small to handle a $C_{I,J}$ block and one panel of A and B, but remember that operation performed at $C_b = C_b + A_{bK}*B_{Kb}$ is a matrix- matrix product so each operand A_{bK} and B_{Kb} is aceessed N_b times: this partcould also use a block method. Since N_b is relatively small, the implementation may load only one of C_b, A_{bK}, B_{Kb} into L_1 cache and works with others from L_2.

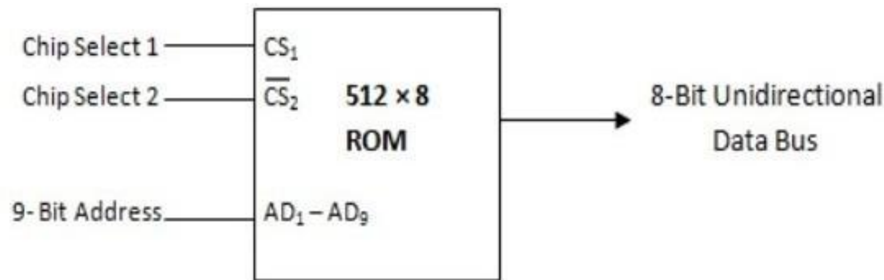
Memory management unit
A computer's memory management unit (MMU) is the physical hardware that handles its virtual memory and caching operations. The MMU is usually located within the computer's central processing unit (CPU), but sometimes operates in a separate integrated chip (IC). All data request inputs are sent to the MMU, which in turn determines whether the data needs to be retrieved from RAM or ROM storage.
A memory management unit is also known as a paged memory management unit.
The memory management unit performs three major functions:
Hardware memory management
Operating system (OS) memory management
Application memory management
Hardware memory management deals with a system's RAM and cache memory, OS memory management regulates resources among objects and data structures, and application memory management allocates and optimizes memory among programs.
The MMU also includes a section of memory that holds a table that matches virtual addresses to physical addresses, called the translation lookaside buffer (TLB).

Semiconductor Memory Technologies:
Semiconductor random-access memories (RAMs) are available in a wide range of speeds.
Their cycle times range from 100 ns to less than 10 ns. Semiconductor memory is used in any electronics assembly that uses computer processing technology. The use of semiconductor memory has grown, and the size of these memory cards has increased as the need for larger and larger amounts of storage is needed.
There are two main types or categories that can be used for semiconductor technology.
RAM - Random Access Memory: As the names suggest, the RAM or random access memory is a form of semiconductor memory technology that is used for reading and writing data in any order - in other words as it is required by the processor. It is used for such applications as the computer or processor memory where variables and other stored and are required on a random basis. Data is stored and read many times to and from this type of memory.



Block Diagram Representing 128 x 8 RAM
(Random Access Memory)

ROM - Read Only Memory: A ROM is a form of semiconductor memory technology used where the data is written once and then not changed. In view of this it is used where data needs to be stored permanently, even when the power is removed - many memory technologies lose the data once the power is removed. As a result, this type of semiconductor memory technology is widely used for storingprograms and data that must survive when a computer or processor is powered down. For example the BIOS of a computer will be stored in ROM. As the name implies, data cannot be easily written to ROM. Depending on the technology used in the ROM, writing the data into the ROM initially may require special hardware. Although it is often possible to change the data, this gain requires special hardware to erase the data ready for new data to be written in.



he different memory types or memory technologies are detailed below:

DRAM: Dynamic RAM is a form of random access memory. DRAM uses a capacitor to store each bit of data, and the level of charge on each capacitor determines whether that bit is a logical 1 or 0.
However these capacitors do not hold their charge indefinitely, and therefore the data needs to be refreshed periodically. As a result of this dynamic refreshing it gains its name of being a dynamic RAM. DRAM is the form of semiconductor memory that is often used in equipment including personal computers and workstations where it forms the main RAM for the computer.

EEPROM: This is an Electrically Erasable Programmable Read Only Memory. Data can be written to it and it can be erased using an electrical voltage. This is typically applied to an erase pin on the chip. Like other types of PROM, EEPROM retains the contents of the memory even when the power is turned off. Also like other types of ROM, EEPROM is not as fast as RAM.

EPROM: This is an Erasable Programmable Read Only Memory. This form of semiconductor memory can be programmed and then erased at a later time. This is normally achieved by exposing the silicon to ultraviolet light. To enable this to happen there is a circular window in the package of the EPROM to enable the light to reach the silicon of the chip. When the PROM is in use, this window is normally covered by a label, especially when the data may need to be preserved for an extended period. The PROM stores its data as a charge on a capacitor. There is a charge storage capacitor for each cell and this can be read repeatedly as required. However it is found that after many years the charge may leak away and the data may be lost. Nevertheless, this type of semiconductor memory used to be widely used in applications where a form of ROM was required, but where the data needed to be changed periodically, as in a development environment, or where quantities were low.

FLASH MEMORY: Flash memory may be considered as a development of EEPROM technology. Data can be written to it and it can be erased, although only in blocks, but data can be read on an individual cell basis. To erase and re-programme areas of the chip, programming voltages at levels that are available within electronic equipment are used. It is also non-volatile, and this makes it particularly useful. As a result Flash memory is widely used in many applications including memory cards for digital cameras, mobile phones, computer memory sticks and many other applications.

F-RAM: Ferroelectric RAM is a random-access memory technology that has many similarities to the standard DRAM technology. The major difference is that it incorporates a ferroelectric layer instead of the more usual dielectric layer and this provides its non-volatile capability. As it offers a non-volatile capability, F-RAM is a direct competitor to Flash.

MRAM: This is Magneto-resistive RAM, or Magnetic RAM. It is a non-volatile RAM memory technology that uses magnetic charges to store data instead of electric charges. Unlike technologies including DRAM,

which require a constant flow of electricity to maintain the integrity of the data, MRAM retains data even when the power is removed. An additional advantage is that it only requires low power for active operation. As a result this technology could become a major player in the electronics industry now that production processes have been developed to enable it to be produced.

P-RAM / PCM: This type of semiconductor memory is known as Phase change Random Access Memory, P-RAM or just Phase Change memory, PCM. It is based around a phenomenon where a form of chalcogenide glass changes is state or phase between an amorphous state (high resistance) and a polycrystalline state (low resistance). It is possible to detect the state of an individual cell and hence use this for data storage. Currently this type of memory has not been widely commercialized, but it is expected to be a competitor for flash memory.

PROM: This stands for Programmable Read Only Memory. It is a semiconductor memory which canonly have data written to it once - the data written to it is permanent. These memories are bought in a blank format and they are programmed using a special PROM programmer. Typically a PROM will consist of an array of fuseable links some of which are "blown" during the programming process to provide the required data pattern.

SDRAM: Synchronous DRAM. This form of semiconductor memory can run at faster speeds than conventional DRAM. It is synchronised to the clock of the processor and is capable of keeping two sets of memory addresses open simultaneously. By transferring data alternately from one set of addresses, and then the other, SDRAM cuts down on the delays associated with non-synchronous RAM, which must close one address bank before opening the next.

SRAM: Static Random Access Memory. This form of semiconductor memory gains its name from thefact that, unlike DRAM, the data does not need to be refreshed dynamically. It is able to support faster read and write times than DRAM (typically 10 ns against 60 ns for DRAM), and in addition its cycle time is much shorter because it does not need to pause between accesses. However it consumes more power, is less dense and more expensive than DRAM. As a result of this it is normally used for caches,while DRAM is used as the main semiconductor memory technology.

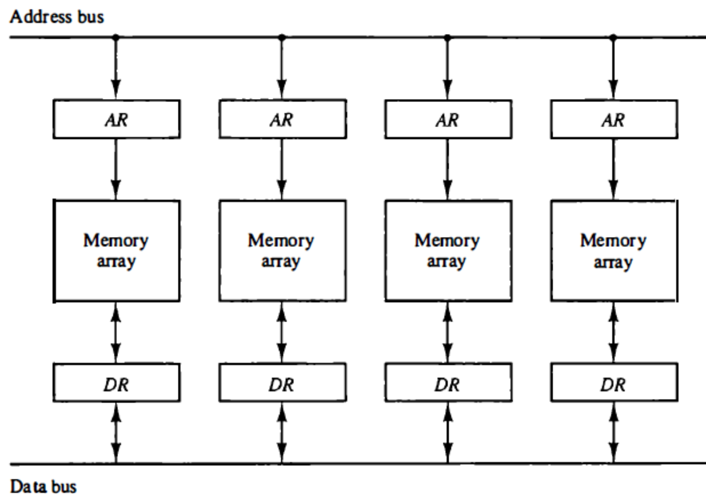
MEMORY ORGANIZATION

Memory Interleaving:

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and an operand at the same time from two different segments.

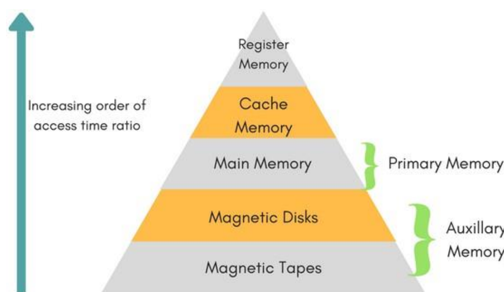
Similarly, an arithmetic pipeline usually requires two or more operands to enter the pipeline atthe same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own address and data registers. Figure 9-13 shows amemory unit with four modules. Each memory array has its own address register AR and data register DR.

Figure 9-13 Multiple module memory organization.

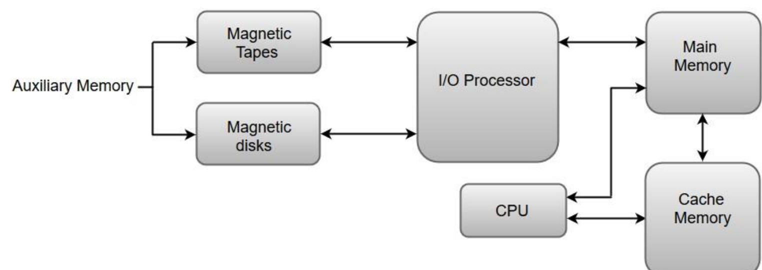


The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus. The two least significant bits of the address can be used to distinguish between the four modules. The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.
The advantage of a modular memory is that it allows the use of a technique called interleaving. In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other.

Concept of Hierarchical Memory Organization
This Memory Hierarchy Design is divided into 2 main types:
External Memory or Secondary Memory
Comprising of Magnetic Disk, Optical Disk, Magnetic Tape i.e. peripheral storage devices whichare accessible by the processor via I/O Module.
Internal Memory or Primary Memory
Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by theprocessor.



Memory Hierarchy in a Computer System:



Characteristics of Memory Hierarchy

Capacity:
It is the global volume of information the memory can store. As we move from top to bottom inthe Hierarchy, the capacity increases.
Access Time:
It is the time interval between the read/write request and the availability of the data. As we movefrom top to bottom in the Hierarchy, the access time increases.
Performance:
Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far downthe memory hierarchy one has to go to manipulate data.
Cost per bit:
As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Cache Memories:
The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference.
Analysis of programs shows that most of their execution time is spent in routines in which manyinstructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other.
The cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.
When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache‟s *replacement algorithm*.

Cache Hits
The processor does not need to know explicitly about the existence of the cache. It simply issues Read andWrite requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache.
If it does, the Read orWrite operation is performed on the appropriate cache location. In this case, a *read* or *write hit* is said to have occurred.

Cache Misses
A Read operation for a word that is not in the cache constitutes a *Read miss*. It causes the block of words containing the requested word to be copied from the main memory into the cache.

Cache Mapping:
There are three different types of mapping used for the purpose of cache memory which are asfollows: Direct mapping, Associative mapping, and Set-Associative mapping. These are explained as following below.
Direct mapping
The simplest way to determine cache locations in which to store memory blocks is the *direct- mapping*

technique. In this technique, block *j* of the main memory maps onto block *j* modulo 128 of the cache, as depicted in Figure 8.16. Thus, whenever one of the main memory blocks 0, 128, 256, . . . is loaded into the cache, it is stored in cache block 0. Blocks 1, 129, 257, . . . are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full.

For example, instructions of a program may start in block 1 and continue in block 129, possiblyafter a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure 8.16. The low-order 4 bits select one of 16 words in a block.

When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache.

The direct-mapping technique is easy to implement, but it is not very flexible.

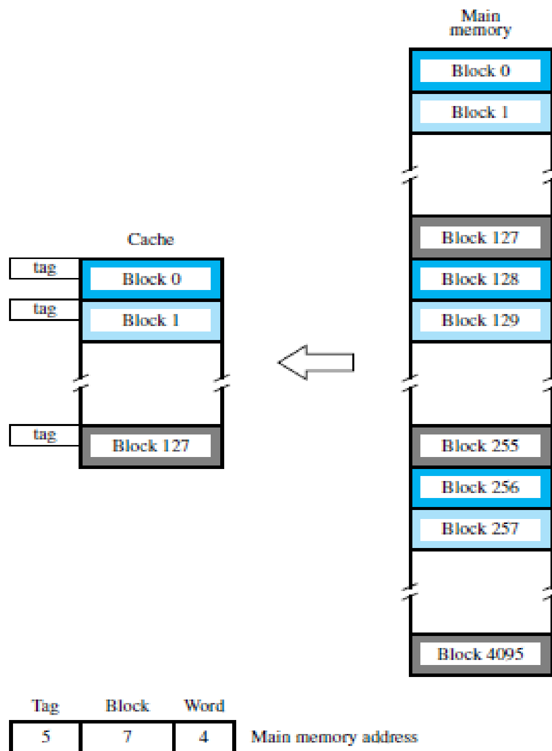


Figure 8.16 Direct-mapped cache.

Associative Mapping

In Associative mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in thecache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping* technique.

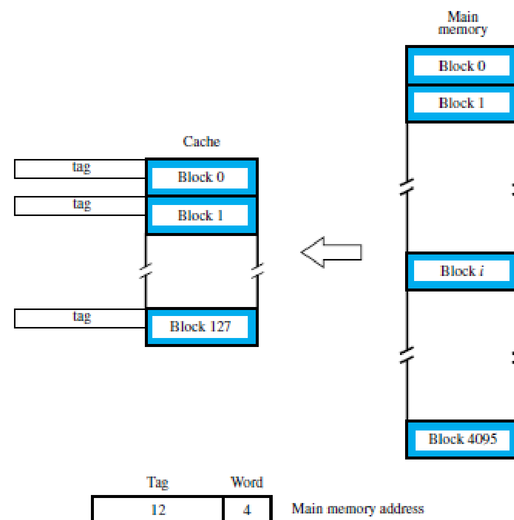


Figure 8.17 Associative-mapped cache.

It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, itreplaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced.

To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an *associative search*.

Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement.

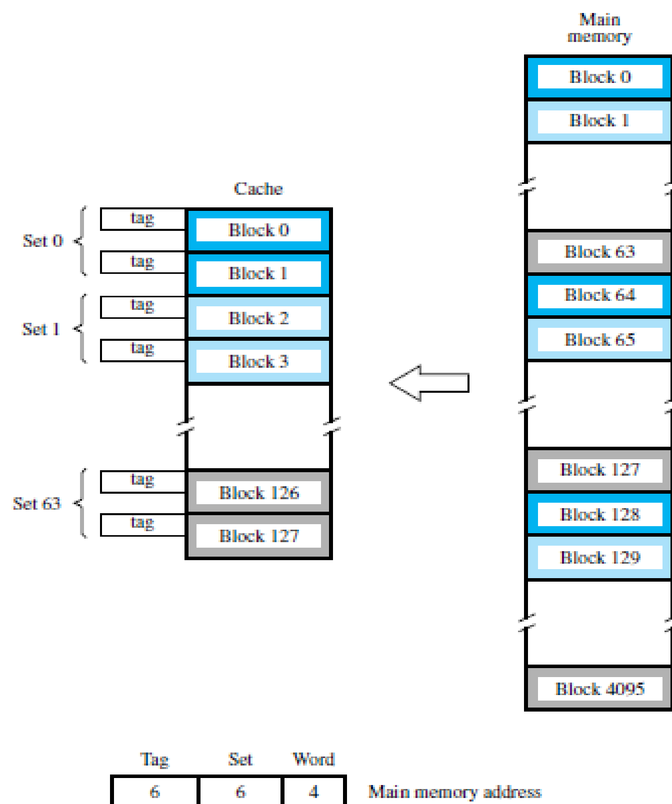


Figure 8.18 Set-associative-mapped cache with two blocks per set.

At the same time, the hardware cost is reduced by decreasing the size of the associative search.
An example of this *set-associative-mapping* technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0*,* 64*,* 128*, . . . ,* 4032 map into cache set 0, and they can occupy either of the two block positions within this set.
Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.
The number of blocks per set is a parameter that can be selected to suit the requirements
of a particular computer. For the main memory and cache sizes in Figure 8.18, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme

condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping.

Replacement Algorithms

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility.

When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite.

This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced.

The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the *least recently used* (LRU) block, and the technique is called the *LRU replacement algorithm*.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases.

Write Policies

The write operation is proceeding in 2 ways.

Write-through protocol

Write-back protocol

Write-through protocol:

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

Write-back protocol:

This technique is to update only the cache location and to mark it as with associated flag bit called dirty/modified bit.

The word in the main memory will be updated later, when the block containing this marked word is to be removed from the cache to make room for a new block.

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

I/O organization

Peripherals:

Input-output device attached to the computer are also called peripherals.

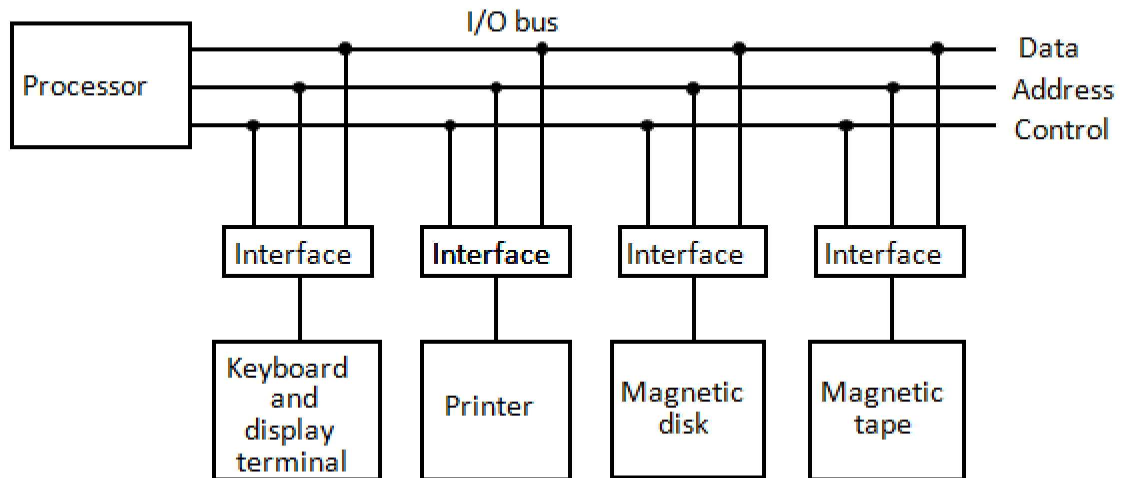


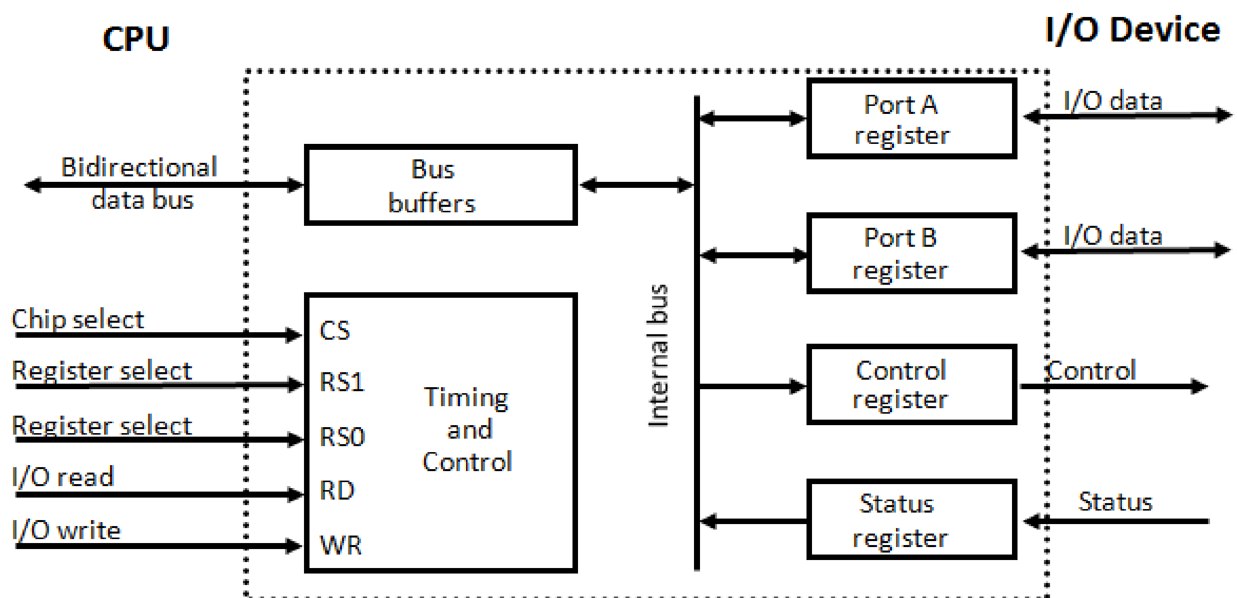
Figure 1.1: Connection of I/O bus to input-output device.

A typical communication link between the processor and several peripherals is shown in figure 1.1.
The I/O bus consists of data lines, address lines, and control lines.
The magnetic disk, printer, and terminal are employed in practically any general purpose computer.
Each peripheral device has associated with it an interface unit.
Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller.
It also synchronizes the data flow and supervises the transfer between peripheral and processor.
Each peripheral has its own controller that operates the particular electromechanical device.
For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters.
The I/O bus from the processor is attached to all peripheral interfaces.
To communicate with a particular device, the processor places a device address on the address lines.
Each interface attached to the I/O bus contains an address decoder that monitors the address lines.
When the interface detects its own address, it activates the path between the bus lines and the device that it controls.
All peripherals whose address does not correspond to the address in the bus are disabled by their interface selected responds to the function code and proceeds to execute it.
The function code is referred to as an I/O command.
There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.
A control command is issued to activate the peripheral and to inform it what to do.
For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction.
A status command is used to test various status conditions in the interface and the peripheral.
For example, the computer may wish to check the status of the peripheral before a transfer is initiated.
During the transfer, one or more errors may occur which are detected by the interface.
These errors are designated by setting bits in a status register that the processor can read at certain intervals.

A data output command causes the interface to respond by transferring data from the bus into one of its registers.
The computer starts the tape moving by issuing a control command.
The processor then monitors the status of the tape by means of a status command.
When the tape is in the correct position, the processor issues a data output command.
The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register.
The interface that communicates with the tape controller and sends the data to be stored on tape.
The data input command is the opposite of the data output.
In this case the interface receives an item of data from the peripheral and places it in its buffer register.
The processor checks if data are available by means of a status command and then issues a data input command.
The interface places the data on the data lines, where they are accepted by the processor.

I/O interface
An example of an I/O interface units is shown in block diagram from in figure 1.2.
It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuit.
The interface communicates with the CPU through the data bus.
The chip select and register select inputs determine the address assigned to the interface.
The I/O read and writes are two control lines that specify an input or output, respectively.
The four registers communicate directly with the I/O device attached to the interface.
The I/O data to and from the device can be transferred into either port A or port B.

If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data.
A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines.
A command is passed to the I/O device by sending a word to the appropriate interface register.
The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes.



CS	RS1	RS0	Register Selected
0	X	X	None: data bus in high impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Figure 1.2: Example of I/O interface unit

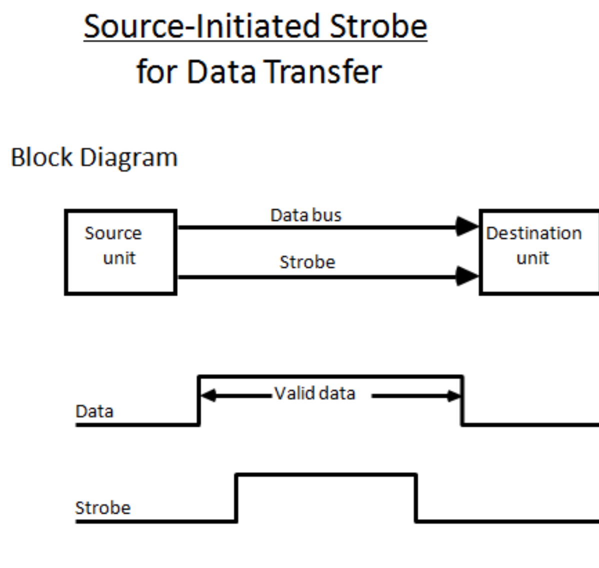
For example, port A may be defined as an input port and port B as an output port.
A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in the forward direction.
The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer.
For example, a status bit may indicate that port A has received a new data item from the I/O device.
Another bit in the status register may indicate that a parity error has occurred during the transfer.

The interface registers communicate with the CPU through the bidirectional data bus.
The address bus selects the interface unit through the chip select and the two register select inputs.
A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers.
This circuit enables the chip select (CS) input when the interface is selected by the address bus.
The two register select inputs RS1 and RS0 are usually connected to the two least significant lines of the address bus.
These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram.
The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enables.
The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.
Asynchronous data transfer and Strobe control
Asynchronous data transfer
Data transfer between two independent units, where internal timing in each unit is independent from the other. Such two units are said to be asynchronous to each other.

Strobe Control
The Strobe control method of asynchronous data transfer employs a single control line to time each transfer.

Source-initiated strobe for data transfer
The strobe may be activated by either the source or the destination unit. Figure 8.3 shows a source-initiated transfer.
The data bus carries the binary information from source unit to the destination unit.
The strobe is a single line that informs the destination unit when a valid data word is available in the bus.
The source unit first places the data on the data bus.

After a delay to ensure that the data settle to a steady value, the source activates the strobe pulse.
The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data.
The source removes the data from the bus a brief period after it disables its strobe pulse.



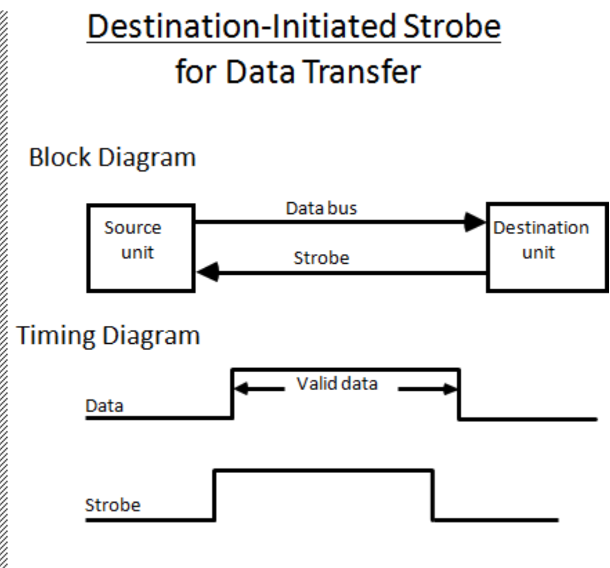


Figure 8.3: Source-initiated strobe for data transfer **Figure 8.4: Destination-initiated strobe for data transfer**

Destination-initiated strobe for data transfer
Figure 8.4 shows a data transfer initiated by the destination unit. In this case the destination unit activates the strobe pulse, informing the source to provide the data.
The source unit responds by placing the requested binary information on the data bus.
The data must be valid and remain in the bus long enough for the destination unit to accept it.
The falling edge of the strobe pulse can be used again to trigger a destination register.
The destination unit then disables the strobe. The source removes the data from the bus after a predetermined time interval.
The transfer of data between the CPU and an interface unit is similar to the strobe transfer just described.
Disadvantage of Strobe method:
The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus
Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus.

Asynchronous data transfer with Handshaking method.
The handshake method solves the problem of Strobe method by introducing a second control signal that provides a reply to the unit that initiates the transfer.

Source-initiated transfer using handshaking
One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valid data in the bus.

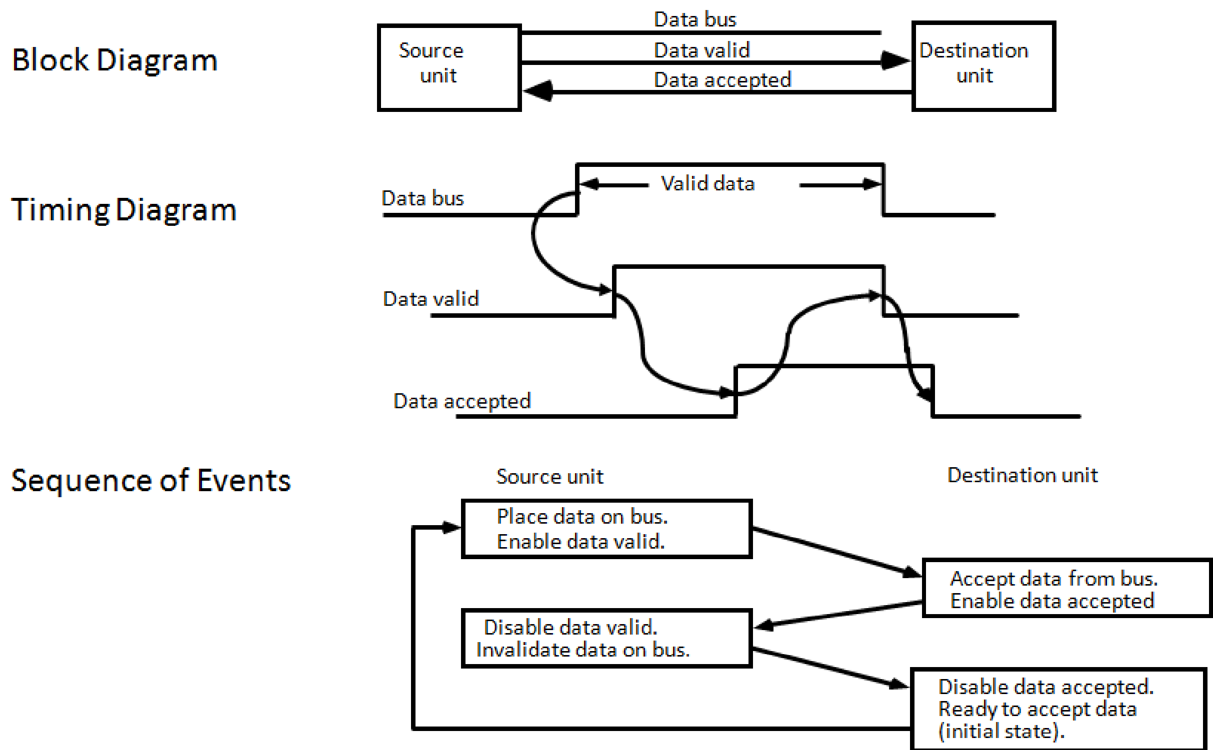


Figure 1.5: Source-initiated transfer using handshaking

The other control line is in the other direction from the destination to the source.
It is used by the destination unit to inform the source whether it can accept data.
The sequence of control during the transfer depends on the unit that initiates the transfer.
Figure 1.5 shows the data transfer procedure initiated by the source.
The two handshaking lines the data valid, which is generated by the source unit, and data accepted, generated by the destination unit, the timing diagram shows the exchange of signals between the two units.
The sequence of events listed in figure 1.5 shows the four possible states that the system can be at any given time.
The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal.

The data accepted signal is activated by the destination unit after it accepts the data from the bus.
The source unit then disables its data valid signal, which invalidates the data on the bus.
The destination unit then disables its data accepted signal and the system goes into its initial state.
The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal.
This scheme allows arbitrary delays from one state to the next and permits each unit to respond at its own data transfer rate.
Destination-initiated transfer using handshaking
The destination-initiated transfer using handshaking lines is shown in figure 1.6.
Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning.
The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit.
From there on, the handshaking procedure follows the same pattern as in the source- initiated case.
Note that the sequence of events in both cases would be identical if we consider the ready for data signal as the complement of data accepted.

In fact, the only difference between the source-initiated and the destination-initiated transfer is in their choice of initial state.

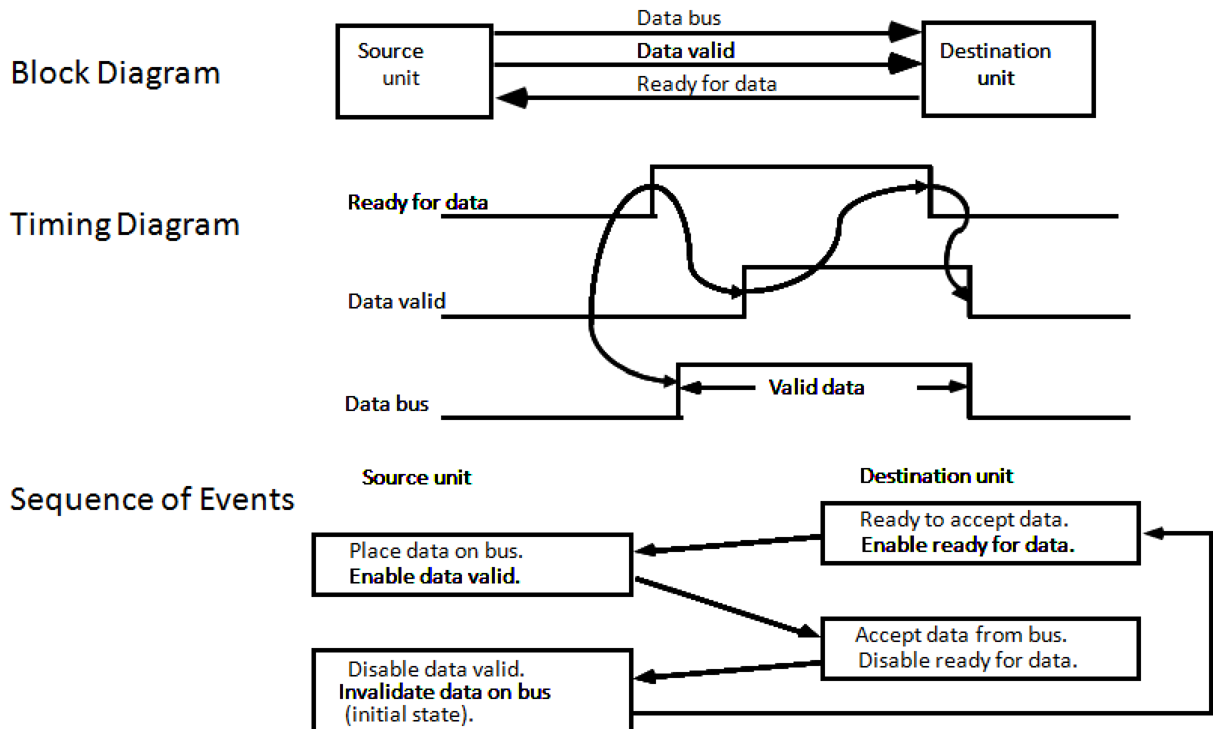


Figure 1.6: Destination-initiated transfer using handshaking

Programmed I/O

Programmed I/O:

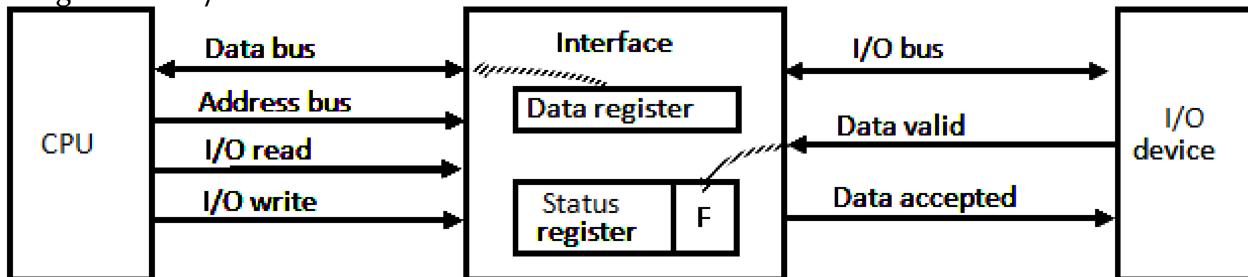


Figure 1.7: Data transfer from I/O device to CPU

In the programmed I/O method, the I/O device does not have direct access to memory.
An example of data transfer from an I/O device through an interface into the CPU is shown in figure 1.7.
When a byte of data is available, the device places it in the I/O bus and enables its data valid line.
The interface accepts the byte into its data register and enables the data accepted line.
The interface sets a bit in the status register that we will refer to as an F or "flag" bit.
The device can now disables the data valid line, but it will not transfer another byte until the data accepted line is disables by the interface.
A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device.
This is done by reading the status register into a CPU register and checking the value of the flag bit.
Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.
Example of Programmed I/O:
A flowchart of the program that must be written for the CPU is shown in figure 8.8.
It is assumed that the device is sending a sequence of bytes that must be stored in memory.
The transfer of each byte requires three instructions :
Read the status register.
Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
Read the data register.
Each byte is read into a CPU register and then transferred to memory with a store instruction.
A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer.

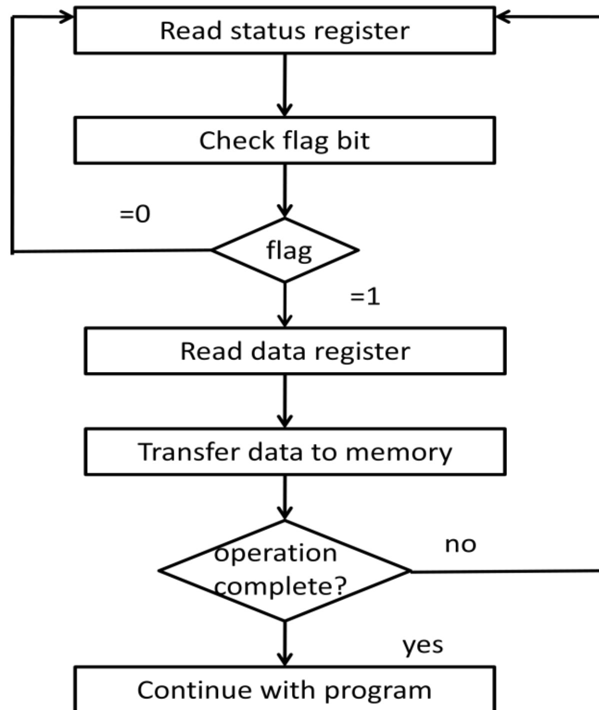


Figure 1.8: Flowchart for CPU program to input data

Interrupt Initiated I/O

In programmed initiated, CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer.

This is a time consuming process since it keeps the processor busy needlessly.

It can be avoided by using an interrupt facility and a special command to inform the interface to issue an interrupt request signal when data are available from the device.

In the meantime CPU can proceed to execute another program.

The interface meanwhile keeps monitoring the device.

When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer.

While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set.

The CPU deviates from what it is doing to take care of the input or output transfer.

After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer.

The way that the processor chooses the branch address of the service routine varies from one unit to another.
In **non-vectored interrupt**, branch address is assigned to a fixed location in memory.
In a **vectored interrupt**, the source that interrupts supplies the branch information to the computer. The information is called vector interrupt.
In some computers the interrupt vector is the first address of the I/O service routine.
In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

Priority interrupt and Daisy Chaining.
Determines which interrupt is to be served first when two or more requests are made simultaneously
Also determines which interrupts are permitted to interrupt the computer while another is being serviced
Higher priority interrupts can make requests while servicing a lower priority interrupt.

Daisy Chaining Priority

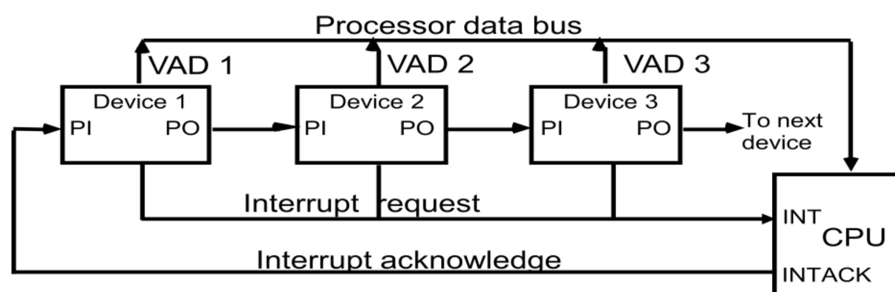


Figure 1.9: Daisy-chain priority interrupt

The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt.
The device with the highest priority is placed in the first position, followed by lower- priority devices up to the device with the lowest priority, which is placed last in the chain.
This method of connection between three devices and the CPU is shown in figure 1.9.
If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU.
When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU.
The CPU responds to an interrupt request by enabling the interrupt acknowledge line.
This signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt.
If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output.

It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.
A device with a 0 in its Pl input generates a 0 in its PO output to inform the next-lower- priority device that the acknowledge signal has been blocked.
A device that is requesting an interrupt and has a 1 in its Pl input will intercept the acknowledge signal by placing a 0 in its PO output.
If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output.
Thus the device with Pl = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus.
The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU.
The farther the device is from the first position; the lower is its priority.

Direct Memory Access (DMA).
Direct Memory Access
Transfer of data under programmed I/O is between CPU and peripheral.
In direct memory access (DMA), Interface transfers data into and out of memory through the memory bus.
The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks.
When the transfer is made, the DMA requests memory cycles through the memory bus.
When the request is granted by the memory controller, DMA transfers the data directly into memory.

DMA controller
DMA controller - Interface which allows I/O transfer directly between Memory and Device, freeing CPU for other tasks
CPU initializes DMA Controller by sending memory address and the block size (number of words).

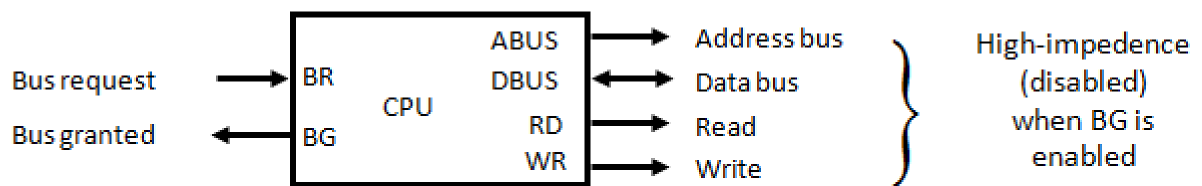


Figure 1.10: CPU bus signals for DMA transfer

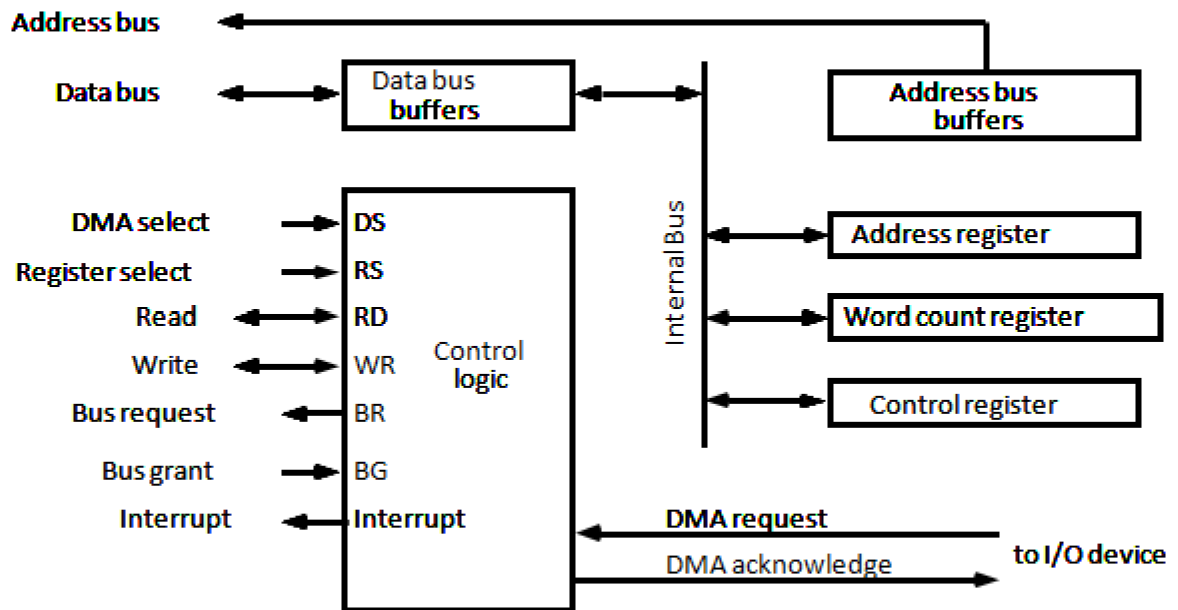


Figure 1.11: Block diagram of DMA controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device.
In addition, it needs an address register, a word count register, and a set of address lines.
The address register and address lines are used for direct communication with the memory.
The word count register specifies the number of words that must be transferred.
The data transfer may be done directly between the device and memory under control of the DMA.
Figure 8.11 shows the block diagram of a typical DMA controller.
The unit communicates with the CPU via the data bus and control lines.
The register in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs.
The RD (read) and WR (write) inputs are bidirectional.
When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers.
When BG= 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address but and activating the RD or WR control.
The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.
The DMA controller has three registers: an address register, a word count register, and a control register.
The address register contains an address to specify the desired location in memory.
The word count register holds the number of words to be transferred.

This register is decremented by one after each word transfer and internally tested for zero.
The control register specifies the mode of transfer.
All registers in the DMA appear to the CPU as I/O interface registers.
Thus the CPU can read from or write into the DMA register under program control via the data bus.
The DMA is first initialized by the CPU.
After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred.
The CPU initializes the DMA by sending the following information through the data bus
The staring address of the memory block where data are available (for read) or where data are to be stored (for write)
The word count, which is the number of words in the memory block.
Control to specify the mode of transfer such as read or write.
The starting address is stored in the address register.

Input- Output Processor (IOP)

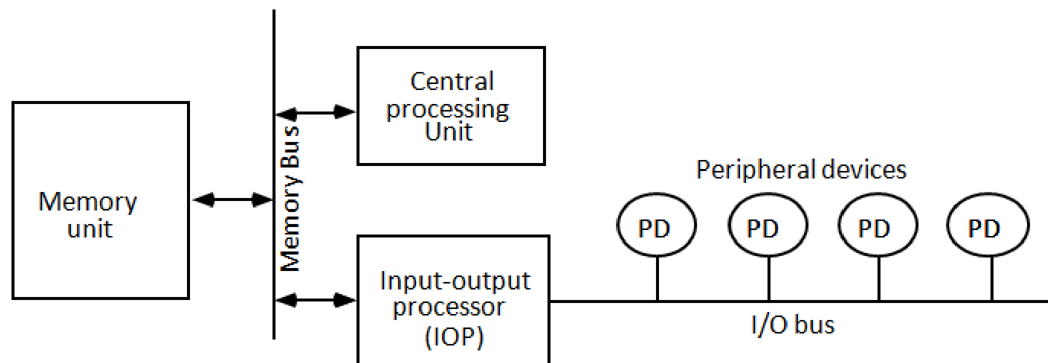


Figure 1.12: Block diagram of a computer with I/O processor

IOP is similar to a CPU except that it is designed to handle the details of I/O processing.
Unlike the DMA controller that must be setup entirely by the CPU, the IOP can fetch and execute its own instruction.
IOP instructions are specifically designed to facilitate I/O transfers.
In addition, IOP can perform other processing tasks, such as arithmetic, logic branching, and code translation.
The block diagram of a computer with two processors is shown in figure 1.12.
The memory unit occupies central position and can communicate with each processor by means of direct memory access.
The CPU is responsible for processing data needed in the solution of computational tasks.
The IOP provides a path of for transfer of data between various peripheral devices and memory unit.

The CPU is usually assigned the task of initiating the I/O program.
From then, IOP operates independent of the CPU and continues to transfer data from external devices and memory.
The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources.
For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory.
Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program.
After the input data are assembled into a memory word, they are transferred from IOP directly into memory by "**stealing**" one memory cycle from the CPU.
Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output word transferred from memory to the IOP.
In most computer systems, the CPU is the master while the IOP is a slave processor.
The CPU is assigned the task of initiating all operations, but I/O instructions are executed in the IOP.
CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities.
The IOP, in turn, typically asks for CPU attention by means of an interrupt.
Instructions that are read from memory by an IOP are sometimes called commands, to distinguish them from instructions that are read by the CPU.

CPU-IOP Communication.
The communication between CPU and IOP may take different forms, depending on the particular computer considered.
In most cases the memory unit acts.
The sequence of operations may be carried out as shown in the flowchart of figure 1.13.
The CPU sends an instruction to test the IOP path.
The IOP responds by inserting a status word in memory for the CPU to check.
The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer.
The CPU refers to the status word in memory to device what do next.
If all is in order, the CPU sends the instruction to start I/O transfer.
The memory address received with this instruction tells the IOP where to find its program.
The CPU can now continue with another program while the IOP is busy with the I/O program.
Both programs refer to memory by means of DMA transfer.
When the IOP terminates the execution of its program, it sends an interrupt request to the CPU.
The CPU responds to the interrupt by issuing an instruction to read the status from the IOP.

The IOP responds by placing the contents of its status report into a specified memory location.
The status word indicates whether the transfer has been completed or if any errors occurred during the transfer.
From inspection of the bits in the status word, the CPU determines if the I/O operation was completed satisfactorily without errors.
The IOP takes care of all data transfers between several I/O units and the memory while the CPU is processing another program.
The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory.

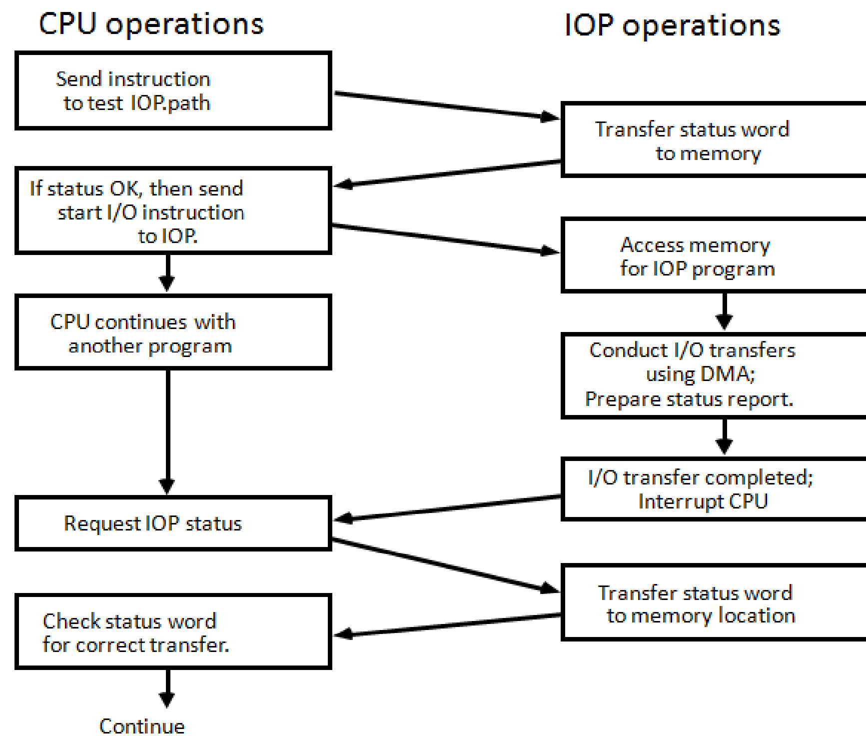


Figure 1.13: CPU-IOP communication

The processor provides 16 registers for use in general system and application programming. These registers can be grouped as follows:
General-purpose data registers. These eight registers are available for storing operands and pointers.
Segment registers. These registers hold up to six segment selectors.
Status and control registers. These registers report and allow modification of the state of the processor and of the program being executed.

General-Purpose Data Registers
The 32-bit general-purpose data registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:
Operands for logical and arithmetic operations
Operands for address calculations
Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for any other purpose.

Segment Registers
The 6 Segment Registers are:

Stack Segment (SS). Pointer to the stack.
Code Segment (CS). Pointer to the code.
Data Segment (DS). Pointer to the data.

- **Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').**
- **F Segment (FS). Pointer to more extra data ('F' comes after 'E').**
- **G Segment (GS). Pointer to still more extra data ('G' comes after 'F').**

Most applications on most modern operating systems (FreeBSD, Linux or Microsoft Windows) use a memory model that points nearly all segment registers to the same place and uses paging instead, effectively disabling their use. Typically the use of FS or GS is an exception to this rule, instead being used to point at thread-specific data.

x86 Processor Registers and Fetch-Execute Cycle

There are 8 registers that can be specified in assembly-language instructions: eax, ebx, ecx, edx, esi, edi, ebp, and esp. Register esp points to the "top" word currently in use on the stack (which grows down).

Register ebp is typically used as a pointer to a location in the stack frame of the currently executing function.

Register ecx can be used in binary arithmetic operations to hold the second operand.

There are two registers that are used implicitly in x86 programs and cannot be referenced by name in an assembly language program.

These are eip, the "instruction pointer" or "program counter"; and eflags, which contains bits indicating the result of arithmetic and compare instructions.

The basic operation of the processor is to repeatedly fetch and execute instructions.

```
while (running) {
    fetch instruction beginning at address in
    eip; eip <- eip + length of instruction;
    execute fetched instruction;
}
```

Execution continues sequentially unless execution of an instruction causes a jump, which is done by storing the target address in eip (this is how conditional and unconditional jumps, and function call and return are implemented).

Addressing modes

The addressing mode indicates how the operand is presented.

Register Addressing

Operand address R is in the address field.

```
mov ax, bx ; moves contents of register bx into ax
```

Immediate

Aactual value is in the field.

```
mov ax, 1 ; moves value of 1 into register ax
```

Or:

```
mov ax, 010Ch ; moves value of 0x010C into register ax
```

Direct memory addressing

Operand address is in the address field.

```
.data
my_var dw 0abcdh ; my_var = 0xabcd
.code
mov ax, [my_var] ; copy my_var content in ax (ax=0xabcd)
```

Direct offset addressing

Uses arithmetics to modify address.

```
byte_tbl db 12,15,16,22, ; Table of bytes
mov al,[byte_tbl+2]
mov al,byte_tbl[2] ; same as the former
```

Register Indirect

Field points to a register that contains the operand address.

```
mov ax,[di]
```

The registers used for indirect addressing are BX, BP, SI, DI

Base-index

```
mov ax,[bx + di]
```

For example, if we are talking about an array, BX contains the address of the beginning of the array, andDI contains the index into the array.

Base-index with displacement

```
mov ax,[bx + di + 10]
```

CPU Operation Modes

Real Mode

Real Mode is a holdover from the original Intel 8086. The Intel 8086 accessed memory using 20-bit addresses. But, as the processor itself was 16-bit, Intel invented an addressing scheme that provided a way of mapping a 20-bit addressing space into 16-bit words. Today's x86 processors start in the so-called Real Mode, which is an operating mode that mimics the behavior of the 8086, with some very tiny differences, for backwards compatibility.

Protected Mode Flat Memory Model

If programming in a modern operating system (such as Linux, Windows), you are basically

programming in flat 32-bit mode. Any register can be used in addressing, and it is generally more efficient to use a full 32-bit register instead of a 16-bit register part. Additionally, segment registersare generally unused in flat mode, and it is generally a bad idea to touch them.

Multi-Segmented Memory Model

Using a 32-bit register to address memory, the program can access (almost) all of the memory in a modern computer. For earlier processors (with only 16-bit registers) the segmented memory model was used. The 'CS', 'DS', and 'ES' registers are used to point to the different chunks of memory. For a small program (small model) the CS=DS=ES. For larger memory models, these 'segments' can point to different locations.

Register Transfer Language And Micro Operations:Register Transfer language:

Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic

The modules are interconnected with common data and control paths to form a digitalcomputer system

The operations executed on data stored in registers are called microoperations

A microoperation is an elementary operation performed on the information stored in one ormore registers

Examples are shift, count, clear, and load

Some of the digital components from before are registers that implement microoperations

The internal hardware organization of a digital computer is best byspecifying

The set of registers it contains and their functions

The sequence of microoperations performed on the binary information stored

The control that initiates the sequence of microoperations

Use symbols, rather than words, to specify the sequence of microoperations The symbolic notation used is called a register transfer language

A programming language is a procedure for writing symbols to specify a given computational process Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations

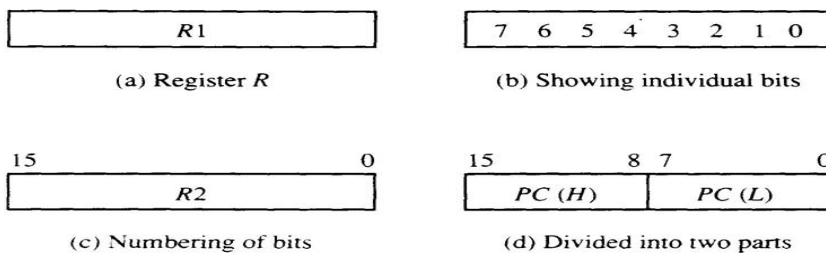
Register Transfer

Designate computer registers by capital letters to denote its function. The register that holds an address for the memory unit is called MAR.The program counter register is called PC.

IR is the instruction register and R1 is a processor register

The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1Refer to Figure 4.1 for the different representations of a register

Figure 4-1 Block diagram of register.



Designate information transfer from one register to another by R2 ⯇ R1
This statement implies that the hardware is available
The outputs of the source must have a path to the inputs of the destination
The destination register has a parallel load capability
If the transfer is to occur only under a predetermined control condition, designate it by
If (P = 1) then (R2 ⯇ R1) or, P: R2 ⯇ R1,where P is a control function that can be either 0 or 1
Every statement written in register transfer notation implies the presence of the required hardware construction
Arithmetic Micro-operations
There are four categories of the most common micro operations:
Register transfer: transfer binary information from one register to another
Arithmetic: perform arithmetic operations on numeric data stored in registers
Logic: perform bit manipulation operations on non-numeric data stored in registers
Shift: perform shift operations on data stored in registers

The basic arithmetic micro operations are addition, subtraction, increment, decrement, and shift Example of addition: R3 ⯇ R1+R2
Subtraction is most often implemented through complementation and addition
Example of subtraction: R3 ⯇ R1 +~~R2~~ + 1 (strikethrough denotes bar on top – 1�s complement of R2)Adding 1 to the 1‟s complement produces the 2‟s complement
Adding the contents of R1 to the 2‟s complement of R2 is equivalent to subtracting

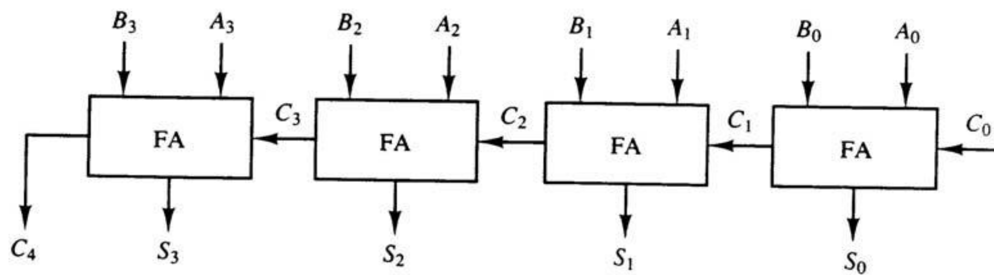


Figure 4-6 4-bit binary adder.

Multiply and divide are not included as micro operations
A micro operation is one that can be executed by one clock pulse
Multiply (divide) is implemented by a sequence of add and shift micro operations (subtractand shift)
To implement the add micro operation with hardware, we need the registers that hold the data and thedigital component that performs the addition
A full-adder adds two bits and a previous carry
A binary adder is a digital circuit that generates the arithmetic sum of two binary numbers of anylength
A binary added is constructed with full-adder circuits connected in cascadeAn n-bit binary adder requires n full-adders

The subtraction A-B can be carried out by the following stepsTake the 1‟s complement of B (invert each bit)
Get the 2‟s complement by adding 1Add the result to A
The addition and subtraction operations can be combined into one common circuit by including anXOR gate with each full-adder
The increment micro operation adds one to a number in a register
This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one
If the increment is to be performed independent of a particular register, then use half-adders connected in cascade
An n-bit binary incrementer requires n half-adders
Each of the arithmetic micro operations can be implemented in one composite arithmetic circuit The basic component is the parallel adder
Multiplexers are used to choose between the different operations
The output of the binary adder is calculated from the following sum: $D = A + Y + C_{in}$

Logic Microoperations

Logic operations specify binary operations for strings of bits stored in registers and treat each bitseparately
Example: the XOR of R1 and R2 is symbolized by
P: R1 ← R1⊕ R2
Example: R1 = 1010 and R2 = 1100

1010	Content of R1
1100	Content of R2
0110	Content of R1 after P = 1

Symbols used for logical microoperations:
OR: ∨
AND: ∧
XOR: ⊕
The + sign has two different meanings: logical OR and summation
When + is in a microoperation, then summation

When + is in a control function, then OR

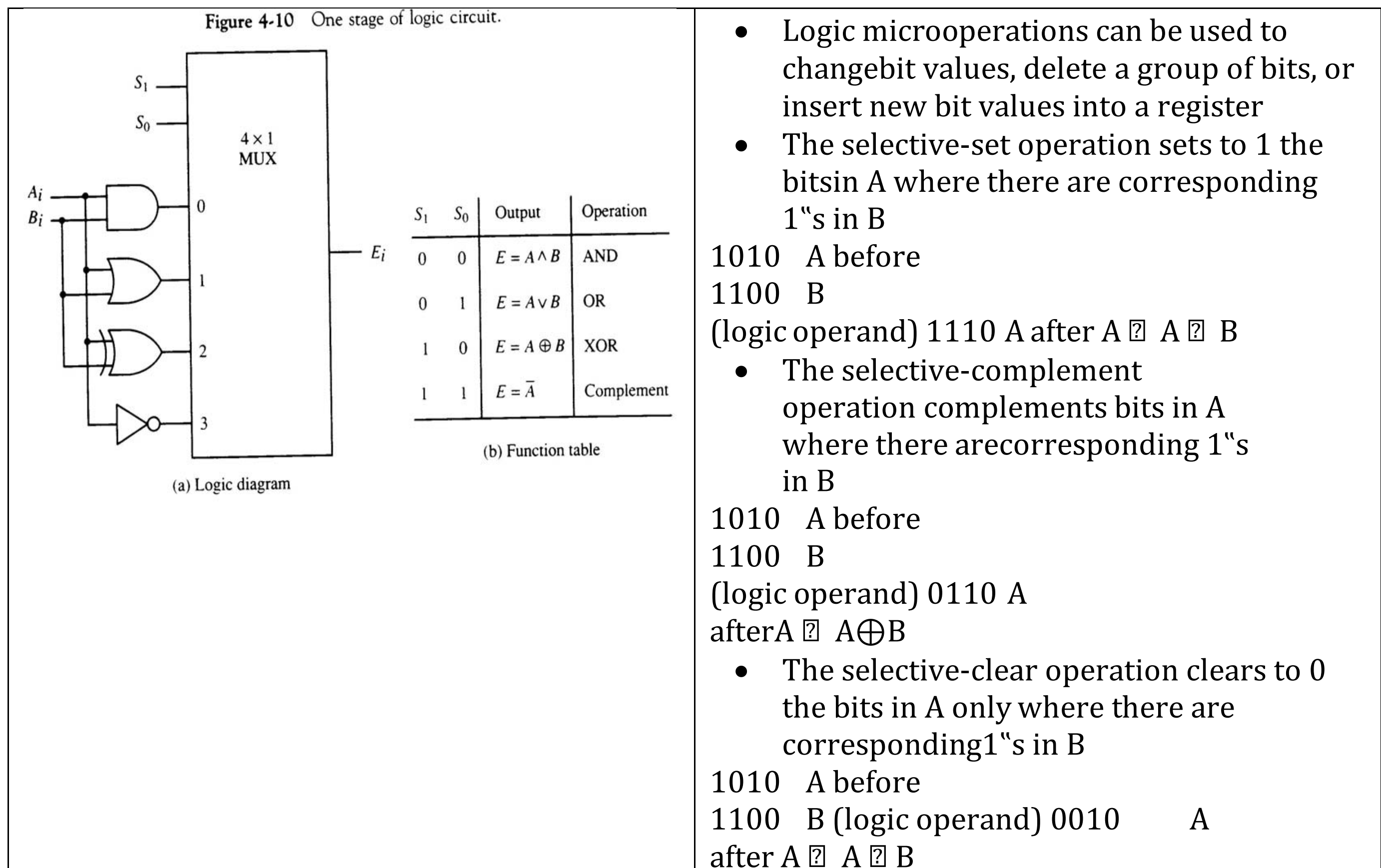
Example:

P + Q: R1 R2 + R3, R4 R5 R6

There are 16 different logic operations that can be performed with two binary variables

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers

All 16 microoperations can be derived from using four logic gates



S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

- Logic microoperations can be used to changebit values, delete a group of bits, or insert new bit values into a register
- The selective-set operation sets to 1 the bitsin A where there are corresponding 1"s in B

1010 A before
1100 B
(logic operand) 1110 A after A A B

- The selective-complement operation complements bits in A where there arecorresponding 1"s in B

1010 A before
1100 B
(logic operand) 0110 A
afterA A⊕B

- The selective-clear operation clears to 0 the bits in A only where there are corresponding1"s in B

1010 A before
1100 B (logic operand) 0010 A
after A A B

The mask operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0"s in B

1010 A before
1100 B
(logic operand) 1000 Aafter A A B

The insert operation inserts a new value into a group of bits

This is done by first masking the bits to be replaced and then Oring them with the bits to beinserted

0110 1010	A before
0000 1111	B (mask)
0000 1010	A after masking
0000 1010	A before
1001 0000	B (insert)
1001 1010	A after insertion

The clear operation compares the bits in A and B and produces an all 0‟s result if the two numberare equal
1010 A
1010 B
0000 A A⊕B

Shift Microoperations
Shift microoperations are used for serial transfer of data
They are also used in conjunction with arithmetic, logic, and other data- processing operations There are three types of shifts: logical, circular, and arithmetic
A logical shift is one that transfers 0 through the serial input
The symbols shl and shr are for logical shift-left and shift-right by one position R1 shlR
The circular shift (aka rotate) circulates the bits of the register around the two ends without loss of information
The symbols cil and cir are for circular shift left and right
The arithmetic shift shifts a signed binary number to the left or right. To the left is multiplying by 2, to the right is dividing by 2.
Arithmetic shifts must leave the sign bit unchanged.
A sign reversal occurs if the bit in R_{n-1} changes in value after the shift.This happens if the multiplication causes an overflow.
An overflow flip-flop V_s can be usedto detecttheoverflow $V_s = R_{n-1} \oplus R_{n-2}$
A bi-directional shift unit with parallel load could be used to implement this
Two clock pulses are necessary with this configuration: one to load the value and another toshift

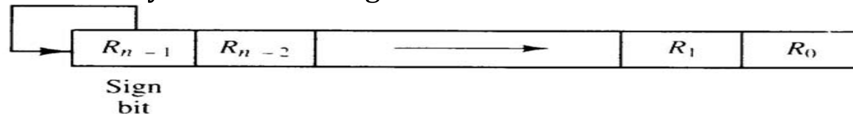


Figure 4-11 Arithmetic shift right.

In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit
The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register
This can be constructed with multiplexers

Arithmetic Logic Unit
The arithmetic logic unit (ALU) is a common operational unit connected to anumber of storage registers
To perform a microoperation, the contents of specified registers are placed in theinputs of the ALU
The ALU performs an operation and the result is then transferred to a destinationregister
The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register canbe performed during one clock pulse period

Micro Programmed Control

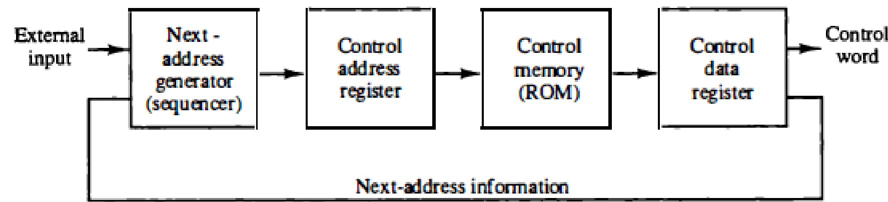
A control unit whose binary control variables are stored in memory is called a microprogrammed control unit. Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes a microprogram .Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).

A more advanced development known as dynamic microprogramming permits a microprogram to beloaded initially from an auxiliary memory such as a magnetic disk.

Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading.

A memory that is part of a control unit is referred to as a control memory.

Figure 7-1 Microprogrammed control organization.



The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory.

The control data register holds the present microinstruction while the next address is computed and read from memory.

The data register is sometimes called a pipeline register.

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established, there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory. It should be mentioned that most computers based on the reduced instruction set computer (RISC).

Address Sequencing

Microinstructions are stored in control memory in groups, with each group specifying a routine.

The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process.

A mapping procedure is a rule that transforms the instruction code into a control memory address

Incrementing of the control address register.

Unconditional branch or conditional branch, depending on status bit conditions.

A mapping process from the bits of the instruction to an address for control memory.

A facility for subroutine call and return

Conditional Branching

Special Bits : The branch logic provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions

Branch Logic : The branch logic hardware may be implemented in a variety of ways. The simplest wayis to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented. This can be implemented with a multiplexer.

Mapping of Instruction: A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction.

Microinstruction Format

The microinstruction format for the control memory is shown in Fig. The 20 bits of the microinstruction are divided into four functional parts. The three fields F1, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type of branch to beused. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 27 words.

15	14 – 11	10 – 0
I	Opcode	Address

(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from Fl.

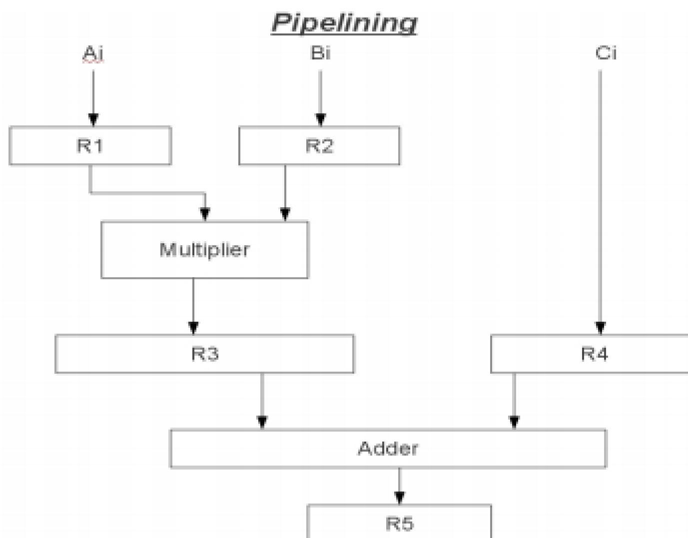
$$DR \leftarrow M[AR] \quad \text{with F2} = 100$$

$$\text{and} \quad PC \leftarrow PC + 1 \quad \text{with F3} = 101$$

Basic concepts of pipelining:
Performance of a computer can be increased by increasing the performance of the CPU.
This can be done by executing more than one task at a time. This procedure is referred to as pipelining. The concept of pipelining is to allow the processing of a new task even though the processing of previous task has not ended.
Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.

Consider the following operation: **Result=(A+B)*C**
First the A and B values are Fetched which is nothing but a "Fetch Operation".
The result of the Fetch operations is given as input to the Addition operation, which is an Arithmetic operation.
The result of the Arithmetic operation is again given to the Data operand C which is fetched from the memory and using another arithmetic operation which is Multiplication in this scenario is executed. Finally the Result is again stored in the "Result" variable.

In this process we are using up-to 5 pipelines which areFetch Operation (A), Fetch Operation(B)
Addition of (A & B), Fetch Operation(C)Multiplication of ((A+B), C)
Load ((A+B)*C)



Now consider the case where a k-segment pipeline with a clock cycle time t, is used to execute n tasks. The first task T1 requires a time equal to k t, to complete its operation since there are k segments in the pipe. The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they willbe completed after a time equal to (n - 1)t, . Therefore, to complete
n tasks using a k-segment pipeline requires k + (n - 1) clock cycles. For example, the diagram of Fig. shows four segments and six tasks.
The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles, as indicated in thediagram.

TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Throughput and Speedup
Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of inaeasing the computational speed of a computer system. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput.
Throughput: Is the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it, the cost of the system increases.
However, technological developments have reduced hardware costs to the point where parallel processing techniques a.re economically feasible.
Speedup of a pipeline processing: The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

$$S = T_{seq} / T_{pipe} = n*m / (m+n-1)$$

the maximum speedup, also called ideal speedup, of a pipeline processor with m stages over an equivalent nonpipelined processor is m. In other words, the ideal speedup is equal to the number of pipeline stages. That is, when n is very large, a pipelined processor can produce output approximately m times faster than a nonpipelined processor. When n is small, the speedup decreases.

Pipeline Hazards
There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three differenttypes.

Hazards

The first hazard is called a **structural hazard**. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn‟t put clothes away. Our carefullyscheduled pipeline plans would then be foiled.

As we said above, the MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories. If the pipeline in Figure 4.27 had a fourth instruction, we would see that in the same clock cycle the fi rst instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipelinecould have a structural hazard.

Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find thematch. Obviously, while you are doing the search, loads must wait that have completed drying and are ready to fold as well as those that have finished washing and are ready to dry.

In a pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

add $s0, $t0, $t1sub $t2, $s0, $t3

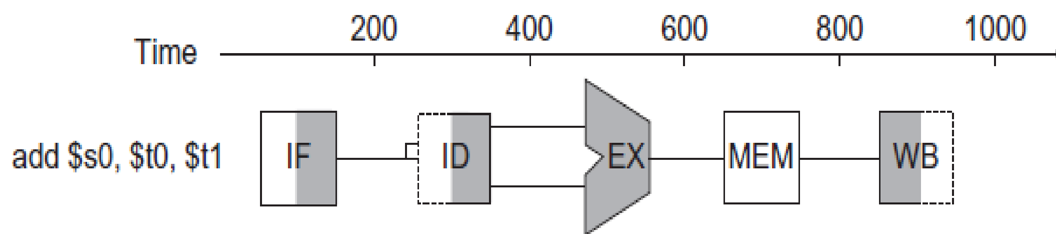


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn‟t write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too oft en and the delay is just too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don‟t need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.

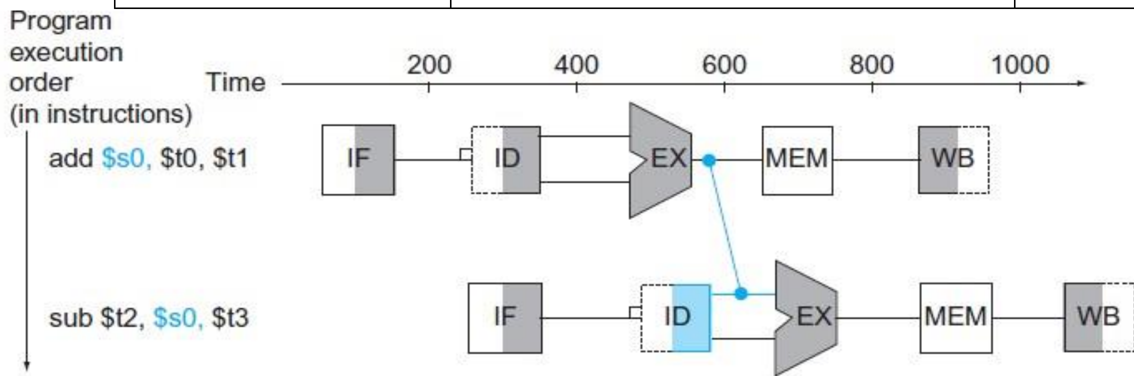


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path

It cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of $s0 instead of an add. As we can imagine from looking at Figure 4.29, the desired data would be available only *after* the fourth stage of the first instruction in the dependence, which is too late for the *input* of the third stage of sub. Hence, even with forwarding, we would have to stall one stage for a **load-use data hazard**, as Figure 4.30 shows. This figure shows an important pipeline concept, officially called a **pipeline stall**, but oft en given the nickname **bubble**. We shall see stalls elsewhere in the pipeline.

Control Hazards

The third type of hazard is called a **control hazard**, arising from the need to make a decision based on the results of one instruction while others are executing. Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until aft er the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

Here is the first of two solutions to control hazards in the laundry room and its computer equivalent. *Stall:* Just operate sequentially until the first batch is dry and then repeat until you have the right formula.

This conservative option certainly works, but it is slow.

Parallel Processors

Introduction to parallel processors:

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of in a easing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

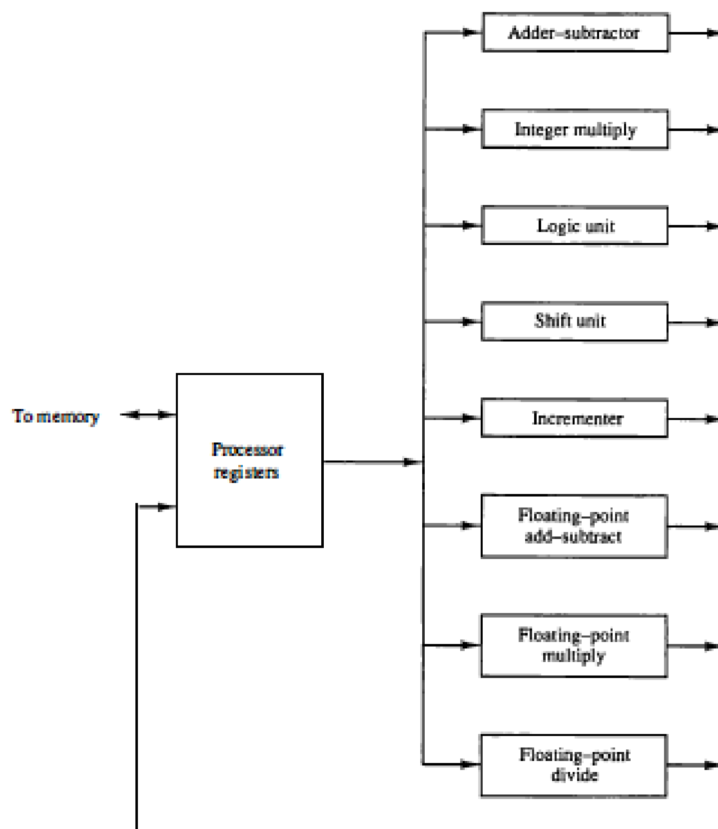
The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques a.re economically feasible.

Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. Shift registers operate in serial fashion one bit at a time, while registers with parallel load operate with all the bits of the word simultaneously.

Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

Figure 9-1 shows one possible way of separating the execution unit into eight functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers.

Figure 9-1 Processor with multiple functional units.



There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One classification introduced by M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor.
The sequence of instructions read from memory constitutes an instruction stream . The operations performed on the data in the processor constitutes a data stream . Parallel processing may occur in the instruction stream, in the data stream, or in both.

Flynn's classification divides computers into four major groups as follows: Single instruction stream, single data stream (SISD)
Single instruction stream, multiple data stream (SIMD) Multiple instruction stream, single data stream (MISD) Multiple instruction stream, multiple data stream (MIMD)

SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.
SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.
MISD structure is only of theoretical interest since no practical system has been constructed using this organization.
MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multicomputer systems can be classified in this category.

Concurrent access to memory and cache coherency:
The primary advantage of cache is its ability to reduce the average access time in uniprocessors. When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer. If the operation is to write, there are two commonly used procedures to update memory.
Write-through policy: In the write-through policy, both cache and main memory are updated with every write operation.
Write-back policy: In the write-back policy, only the cache is updated and the location is marked so that it can be copied later into main memory.
In a shared memory multiprocessor system, all the processors share a common memory. In addition, each processor may have a local memory, part or all of which may be a cache. The compelling reason for having separate caches for each processor is to reduce the average access time in each processor. The same information may reside in a number of copies in some caches and main memory.
To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical.
This requirement imposes a cache coherence problem. A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address. Without a proper solution to the cache coherence problem, caching cannot be used in bus- oriented multiprocessors with two or more processors.

Conditions for Incoherence

Cache coherence problems exist in multiprocessors with private caches because of the need to sharewritable data. Read-only data can safely be replicated without cache coherence enforcement mechanisms.

To illustrate the problem, consider the three-processor configuration with private caches shown in Fig. 13-12. Sometime during the operation an element X from main memory is loaded into the three processors, P1, P2, and P3. As a consequence, it is also copied into the private caches of the three processors. For simplicity, we assume that X contains the value of 52. The load on X to the three processors results in consistent copies in the caches and main memory. If one of the processors performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value. Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache.

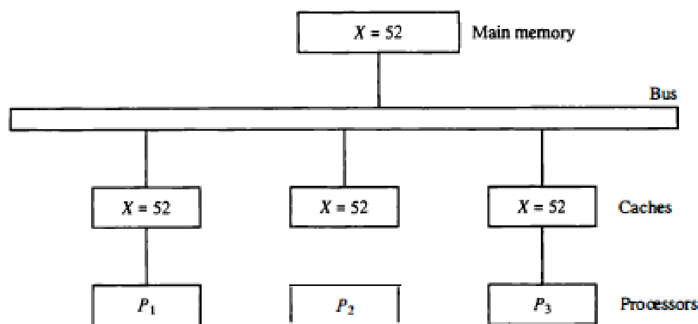
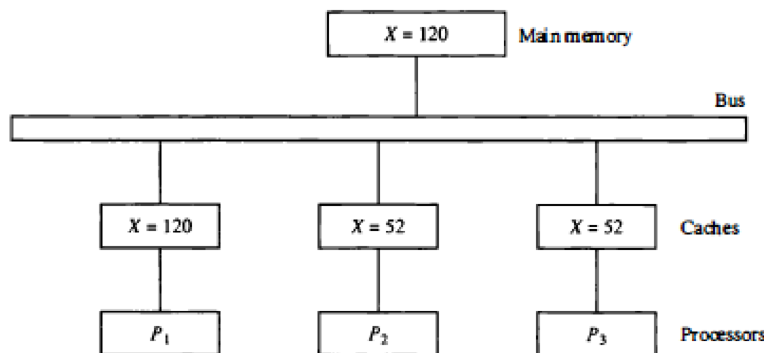


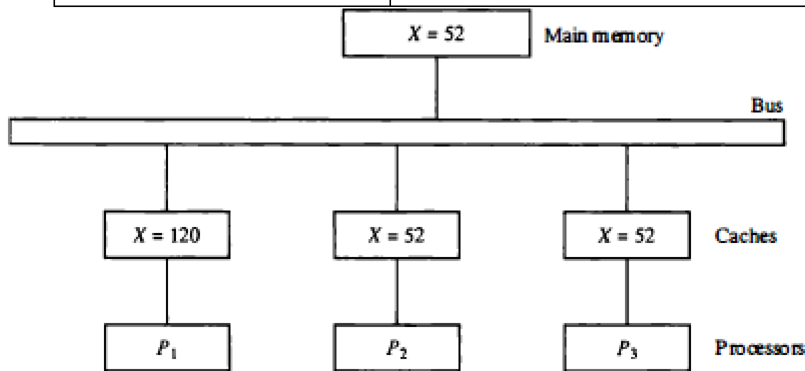
Figure 13-12 Cache configuration after a load on X.

This is shown in Fig. 13-13. A store to X (of the value of 120) into the cache of processor P1 updates memory to the new value in a write-through policy. A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value. In a write-back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory are inconsistent. Memory is updated eventually when the modified data in the cache are copied back into memory.

Figure 13-13 Cache configuration after a store to X by processor P_1.



(a) With write-through cache policy

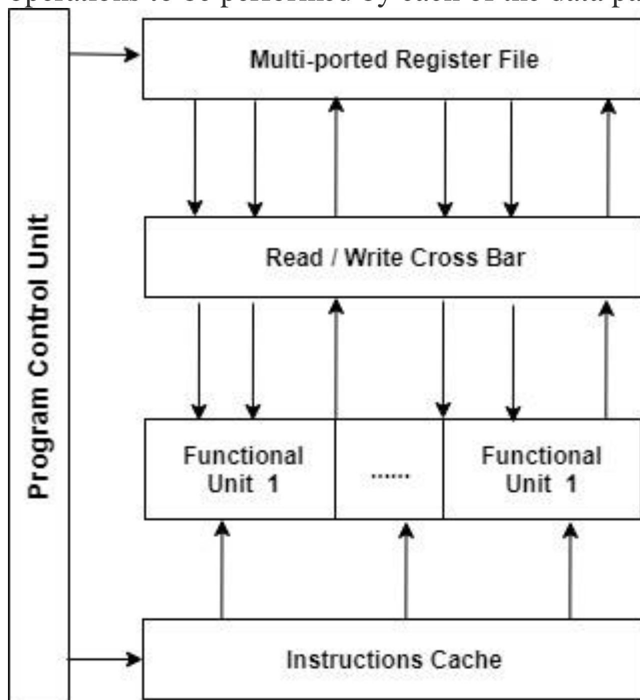


(b) With write-back cache policy

Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus. In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMAoutput, memory locations may be read before they are updated from the cache when using a write-back policy. VO-based memory incoherence can be overcome by making the IOP a participant in the cache coherent solution that is adopted in the system.

Very Long Instruction Word (VLIW) architecture in P-DSPs (programmable DSP) increases the number of instructions that are processed per cycle. It is a concatenation of several short instructions and requires multiple execution units running in parallel, to carry out the instructions in a single cycle. A language compiler or pre-processor separates program instructions into basic operations and places them into VLWI processor which then disassembles and transfers each operation to an appropriate execution unit.

VLIW P-DSPs have a number of processing units (data paths) i.e. they have a number of ALUs, MAC units, shifters, etc. The VLIW is accessed from memory and is used to specify the operands and operations to be performed by each of the data paths.



VLIW Architecture

As shown in figure, the multiple functional units share a common multiported register file for fetching the operands and storing the results. Parallel random access by the functional units to the register file is

facilitated by the read/write cross bar. Execution of the operations in the functional units is carried out concurrently with the load/ store operation of data between a RAM and the register file.

The performance gains that can be achieved with VLIW architecture depends on the degree of parallelism in the algorithm selected for a DSP application and the number of functional units. The throughput will be higher only if the algorithm involves execution of independent operations. For example, in convolution by using eight functional units, the time required can be reduced by a factor of 8 compared to the case where a single functional unit is used.

However, it may not always be possible to have independent stream of data for processing. The number of functional units is also limited by the hardware cost for the multi-ported register file and cross bar switch.

Advantages of VLIW architecture

Increased performance.

Potentially scalable i.e. more execution units can be added and so more instructions can be packed into the VLIW instruction.

Disadvantages of VLIW architecture

New programmer needed.

Program must keep track of Instruction scheduling.

Increased memory use.

High power consumption.

Why to use VLIW?

The key to higher performance in microprocessors for a broad range of applications is the ability to exploit fine-grain, instruction-level parallelism. Some methods for exploiting fine-grain parallelism include:

pipelining

multiple processors

superscalar implementation

specifying multiple independent operations per instruction

Pipelining is now universally implemented in high-performance processors. Little more can be gained by improving the implementation of a single pipeline. Using multiple processors improves performance for only a restricted set of applications. Superscalar implementations can improve performance for all types of applications. Superscalar (super: beyond; scalar: one dimensional) means the ability to fetch, issue to execution units, and complete more than one instruction at a time.

Superscalar implementations are required when architectural compatibility must be preserved, and they will be used for entrenched architectures with legacy software, such as the x86 architecture that dominates the desktop computer market. Specifying multiple operations per instruction creates a very-long instruction word architecture or VLIW.

A VLIW implementation has capabilities very similar to those of a superscalar processor—issuing and completing more than one operation at a time—with one important exception: the VLIW hardware is not responsible for discovering opportunities to execute multiple operations concurrently. For the VLIW implementation, the long instruction word already encodes the concurrent operations. This explicit encoding leads to dramatically reduced hardware complexity compared to a high-degree superscalar implementation of a RISC or CISC. The big advantage of VLIW, then, is that a highly concurrent (parallel) implementation is much simpler and cheaper to build than equivalently concurrent RISC or CISC chips. **VLIW is a simpler way to build a superscalar microprocessor**.