DEPARTMEhT OF ELECTRONICS AND **COMMUNICATION ENGINEERING SYLLABUS (THEORY)**

COMPUTER ORGANIZATION & ARCHITECTURE

UNIT I COMPUTER ORGANIZATION & INSTRUCTIONS 9

Basics of a computer system: Evolution, Ideas, Technology, Performan ce, Power wall, Uniprocessu rs tu Multipro cessors. Address in g and addressing modes. Instructions: Operations and Operands, Representing instructions, Logical operations, control operations.

UNIT II ARITHMETIC 9

Fixed point Addition, Subtraction, Multiplication and Division. Floating Point arithmetic, High performance arithmetic, Sub word parallelism

UNIT III **THE** PROCESSOR 9

Introduction, Logic Design Conventions, Building a Data path A Simple Implementation scheme An Overview of Pipelining - Pipelined Data path and Control. Data Hazards: Forwarding versus Stalling, Co ntrol Ha zards, Exceptin n.s, Paralleli.sm via Instru ction.s.

UNIT IV MEMORY AN D I/O 0 RGANtZAT ION 9

Memory hierarchy, Memu i y Chip O rganization, Cache memo ry, Virtual memory. Parallel Bus Architectures, Internal Communication Meth odologies, Serial Bus A rchitectiires, Mass sto rage, In put and Output Devices.

UNIT V ADVANCED COMPUTER ARCHITECTURE 9

Parallel process ing architects res and challenges, Har‹tware multith reading, Multicore anlt shared memory multiprocessors, Intro du ction to Gra ph ics Pmce.ssing Unit.s, Clusters a nd Warehouse scale computers - Introduction to Multiprocessor network topologies.

TOTAL: 45 PERIODS

TEXT BOOKS:

1. David A. Paterson and John L. Hennessey, Computer 0 i ganization and Design ||, Fifth edition, Morgan Kauffman / Elsevier, 2014. (UNIT I -V)

2. Miles |. Murdocca and Vincent P. Heuring, Computer Architecture and Organization: An Integrated approach ||, Secon‹t editio n, Wiley lndia Pvt Ltd, 2015 (U N IT IV,V)

REFERENCES

1. V. Carl Hamacher, Zvonko fi. Varanesic and Safat fi. Zaky, -Computer Organiza tio n —, Fifth edition, Mc firaw-Hill Education India Pvt Ltd, 2014.2. William S t a l l i n g s Computer Organization a n d A r c h it e c t u r e I. S e v e nth E d i t i o n , P e a r s o n
Education, 2006.

3. Govindarajalu, -Comp uter Architecture and 0 rganization, Design Principles anlt Applications", Second cditinn, McGraw- H ill Ed ucation India Pvt Ltd, 2014.

UNIT – I

COMPUTER ORGANIZATION & INSTRUCTIONS

1.1 INTRODUCTION

Computer architecture acts as the interface between the hardware and the lowest level software. Computer architecture refers to:

- Attributes of a system visible to programmers like data type of variables.
- Attributes that have a direct impact on the execution of programs like clock cycle.

Computer Architecture is defined as study of the structure, behavior, and design of computers.

Computer Organization: It refers to the operational units and their interconnections that realize the architectural specifications. It describes the function of and design of the various units of digital computer that store and process information. The attributes in computer organization refers to:

- Control signals
- Computer/peripheral interface
- Memory technology

Computer hardware: Consists of electronic circuits, displays, magnetic and optical storage media, electromechanical equipment and communication facilities.

Computer Architecture: It is concerned with the structure and behavior of the computer. It includes the information formats, the instruction set and techniques for addressing memory. The attributes in computer architecture refers to the:

- Instruction set
- Data representation
- I/O mechanisms
- Addressing techniques

The basic **distinction between architecture and organization** is: the attributes of the former are visible to programmers whereas the attributes of the later describes how features are implemented in the system.

1.2 BASICS OF A COMPUTERSYSTEM

The modern day computer systems functional unit is given by Von Neumann Architecture.

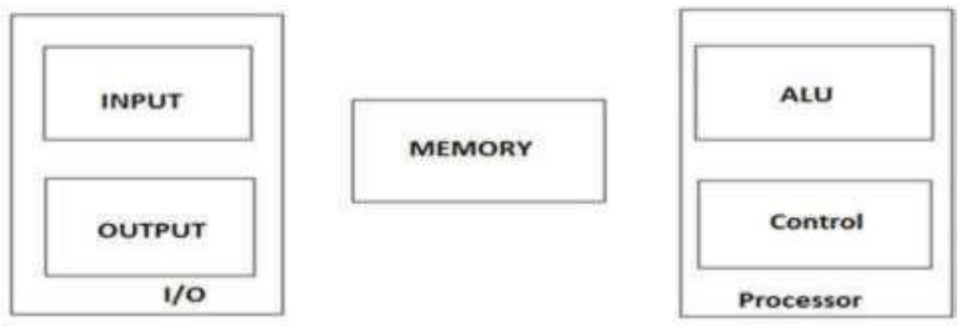


Fig 1.1: Von Neumann Architecture

Input Unit

Computers accepts the coded information through input unit. Computer must receive both data and program statements to function properly and must be able to solve problems. The method of feeding data and programs to a computer is accomplished by an input device. Input devices read data from a source, such as magnetic disks, and translate that data into electronic impulses for transfer into the CPU. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or the processor.

Central Processing Unit (CPU)

The CPU processes data transferred to it from one of the various input devices. It then transfers either an intermediate or final result of the CPU to one or more output devices. A central control section and work areas are required to perform calculations or manipulate data. The CPU is the computing center of the system. It consists of a control section, an arithmetic-logic section, and an internal storage section (memory unit). Each section within the CPU serves a specific function and has a particular relationship with the other sections within the CPU.

Memory Unit

It stores the programs and data. Memory unit is broadly classified into two types: Primary memory and Secondary memory.

1. **Primary Memory:**

It is a fast memory that operates at electronic speeds. Programs must be stored in the memory while they are being executed. The memory contains large no of semiconductor storage cells. Each cell carries 1 bit of information. The cells are processed in a group of fixed size called **Words**. To provide easy access to any word in a memory, a distinct address is associated with each word location. Addresses are numbers that identify successive locations. The number of bits in each word is called the **word length.** The word length ranges from 16 to 64 bits. There are 3 types of primary memory:

I. **RAM:** Memory in which any location can be reached in short and fixed amount of time after specifying its address is called RAM. Time required to access 1 word is called Memory Access Time.

II. **Cache Memory:** The small, fast, RAM units are called Cache. They are tightly coupled with processor to achieve high performance.

III. **Main Memory:** The largest and the slowest unit is the main memory.

Arithmetic & Logic Unit

Most computer operations are executed in ALU. The arithmetic-logic section performs arithmetic operations, such as addition, subtraction, multiplication, and division. Through internal logic capability, it tests various conditions encountered during processing and takes action based on the result. Data may be transferred back and forth between these two sections several times before processing is completed. Access time to registers is faster than access time to the fastest cache unit in memory.

Output Unit

Its function is to send the processed results to the outside world.

Control Unit

The operations of Input unit, output unit, ALU are co-ordinate by the control unit. The control unit is the Nerve centre that sends control signals to other units and senses their states. The control section directs the flow of traffic (operations) and data. It also maintains order within the computer. The control section selects one program statement at a time from the program storage area, interprets the statement, and sends the appropriate electronic impulses to the arithmetic-logic and storage sections so they can carry out the instructions. The control section does not perform actual processing operations on the data.

The control section instructs the input device on when to start and stop transferring data to the input storage area. It also tells the output device when to start and stop receiving data from the output storage area. Data transfers between the processor and the memory are controlled by the control unit through **timing signals**. Information stored in the memory is fetched, under program control into an arithmetic and logic unit, where it is processed.

1.2.1 Evolution of Computers

- The word ‛computer’ is an old word that has changed its meaning several times in the last few centuries.
- Today, the word computer refers to computing devices, whether or not they are electronic, programmable, or capable of ‛storing and retrieving’ data.

The Mechanical Era (1623-1945)

- Wilhelm Schick hard, Blaise Pascal, and Gottfried Leibnitz were among mathematicians who designed and implemented calculators that were capable of addition, subtraction, multiplication, and division during the seventeenth century.
- The first multi-purpose or programmable computing device was probably **Charles Babbage's Difference Engine**, which was begun in 1823 but never completed.
- In 1842, Babbage designed a more ambitious machine, called the **Analytical Engine** but unfortunately it also was only partially completed.
- Babbage, together with Ada Lovelace recognized several important programming techniques, including conditional branches, iterative loops and index variables.
- Babbage designed the machine which is the first to be used in computational science.
- In 1933, George Scheutz and his son, Edvard began work on a smaller version of the difference engine and by 1853 they had constructed a machine that could process 15-digit numbers and calculate fourth-order differences.
- The US Census Bureau was one of the first organizations to use the mechanical computers which used punch-card equipment designed by Herman Hollerith to tabulate data for the 1890 census.
- In 1911 Hollerith's company merged with a competitor to found the corporation which in 1924 became **International Business Machines** (IBM).

First Generation Electronic Computers (1937-1953)

- These devices used electronic switches, in the form of **vacuum tubes**, instead of electromechanical relays.
- The earliest attempt to build an electronic computer was by J. V. Atanasoff, a professor of physics and mathematics at Iowa State in 1937.
- Atanasoff set out to build a machine that would help his graduate students solve systems of partial differential equations.
- By 1941 he and graduate student Clifford Berry had succeeded in building a machine that could solve 29 simultaneous equations with 29 unknowns.
- However, the machine was not programmable, and was more of an electronic calculator.
- A second early electronic machine was Colossus, designed by Alan Turing for the British military in 1943.
- The first general purpose programmable electronic computer was the Electronic Numerical Integrator and Computer **(ENIAC),** built by J. Presper Eckert and John V. Mauchly at the University of Pennsylvania.
- ENIAC was controlled by a set of external switches and dials; to change the program required physically altering the settings on these controls.
- Research work began in 1943, funded by the Army Ordinance Department, which needed a way to compute ballistics during World War II.
- The machine was completed in 1945 and it was used extensively for calculations during the design of the hydrogen bomb.
- Eckert, Mauchly, and John von Neumann, a consultant to the ENIAC project, began work on a new machine before ENIAC was finished.
- The next development was **EDVAC**- Electronic Discrete Variable Computer.
- The main contribution of EDVAC, their new project, was the notion of a **stored program.**
- EDVAC was able to run orders of magnitude faster than ENIAC and by storing instructions in the same medium as data, designers could concentrate on improving the internal structure of the machine without worrying about matching it to the speed of an external control.

- Eckert and Mauchly later designed the first commercially successful computer, the **UNIVAC (**Universal Automatic Computer**)**; in 1952.
- Software technology during this period was very primitive.
- The instructions were written in machine language that could be executed directly.

Second Generation (1954-1962)

- The second generation witnessed several important developments at all levels of computer system design, ranging from the technology used to build the basic circuits to the programming languages used to write scientific applications.
- Electronic switches in this era were based on **discrete diode and transistor technology** with a switching time of approximately 0.3 microseconds.
- The first machines to be built with this technology include **TRADIC** at Bell Laboratories in 1954 and TX-とat M)T s Lincoln Laboratory.
- Index registers were designed for controlling loops and floating point units for calculations based on real numbers.
- A number of high level **programming languages** were introduced and these include FORTRAN (1956), ALGOL (1958), and COBOL (1959).
- **Batch processing systems** came to existence.
- Important commercial machines of this era include the IBM 704 and its successors, the 709 and 7094.
- In the 1950s the first two supercomputers were designed specifically for numeric processing in scientific applications.
- Multi programmed computers that serve many users concurrently came to existence. This is otherwise known as **time-sharing systems**.

Third Generation (1963-1972)

- Technology changes in this generation include the use of **integrated circuits**, or ICs.
- This generation led to the introduction of **semiconductor memories**, **micro programming** as a technique for efficiently designing complex processors and the introduction of **operating systems** and time-sharing.

1.7 Introduction

- The first ICs were based on small-scale integration (SSI) circuits, which had around 10 devices per circuit (or chip), and evolved to the use of medium-scale integrated (MSI) circuits, which had up to 100 devices per chip.
- Multilayered printed circuits were developed and core memory was replaced by faster, solid state memories.
- In 1964, Seymour Cray developed the CDC 6600, which was the first architecture to use **functional parallelism**.
- By using 10 separate functional units that could operate simultaneously and 32 independent memory banks, the CDC 6600 was able to attain a computation rate of one million floating point operations per second (Mflops).
- Five years later CDC released the 7600, also developed by Seymour Cray.
- The CDC 7600, with its pipelined functional units, is considered to be the first vector processor and was capable of executing at ten Mflops.
- The IBM 360/91, released during the same period, was roughly twice as fast as the CDC 660.
- Early in this third generation, Cambridge University and the University of London cooperated in the development of **CPL** (Combined Programming Language, 1963).
- CPL was an attempt to capture only the important features of the complicated and sophisticated ALGOL.
- However, like ALGOL, CPL was large with many features that were hard to learn.
- In an attempt at further simplification, Martin Richards of Cambridge developed a subset of CPL called **BCPL** (Basic Computer Programming Language, 1967).
- In 1970 Ken Thompson of Bell Labs developed yet another simplification of CPL called simply **B**, in connection with an early implementation of the UNIX operating system.

Fourth Generation (1972-1984)

- **Large scale integration** (LSI - 1000 devices per chip) and **very large scale integration** (VLSI - 100,000 devices per chip) were used in the construction of the fourth generation computers.
- Whole processors could now fit onto a single chip, and for simple systems the entire computer (processor, main memory, and I/O controllers) could fit on one chip.

- Gate delays dropped to about 1ns per gate. Core memories were replaced by semiconductor memories.
- Large main memories like CRAY 2 began to replace the older high speed vector processors, such as the CRAY 1, CRAY X-MP and CYBER.
- In 1972, Dennis Ritchie developed the **C language** from the design of the CPL and Thompson's B.
- Thompson and Ritchie then used C to write a version of UNIX for the DEC PDP-11.
- Other developments in software include very high level languages such as FP (functional programming) and Prolog (programming in logic).
- IBM worked with Microsoft during the 1980s to start what we can really call PC (Personal Computer) life today.
- IBM PC was introduced in October 1981 and it worked with the operating system (software) called 'Microsoft Disk Operating System (MS DOS) など.
- Development of **MS DOS** began in October 1980 when IBM began searching the market for an operating system for the then proposed IBM PC and major contributors were Bill Gates, Paul Allen and Tim Paterson.
- In 1983, the **Microsoft Windows** was announced and this has witnessed several improvements and revision over the last twenty years.

Fifth Generation (1984-1990)

- This generation brought about the introduction of machines with hundreds of processors that could all be working on different parts of a single program.
- The scale of integration in semiconductors continued at a great pace and by 1990 it was possible to build chips with a million components - and semiconductor memories became standard on all computers.
- Computer networks and single-user workstations also became popular. Parallel processing started in this generation.
- The Sequent Balance 8000 connected up to 20 processors to a single shared memory module though each processor had its own local cache.
- The machine was designed to compete with the **DEC VAX-780** as a general purpose UNIX system, with each processor working on a different user's job.

- However Sequent provided a library of subroutines that would allow programmers to write programs that would use more than one processor, and the machine was widely used to explore parallel algorithms and programming techniques.
- The **Intel iPSC-1**, also known as the hypercube connected each processor to its own memory and used a network interface to connect processors.
- This distributed memory architecture meant memory was no longer a problem and large systems with more processors (as many as 128) could be built.
- Also introduced was a machine, known as a **data-parallel** or SIMD where there were several thousand very simple processors which work under the direction of a single control unit.
- Both wide area network (WAN) and local area network (LAN) technology developed rapidly.

Sixth Generation (1990 -)

- Most of the developments in computer systems since 1990 have not been fundamental changes but have been gradual improvements over established systems.
- This generation brought about gains in parallel computing in both the hardware and in improved understanding of how to develop algorithms to exploit parallel architectures.
- Workstation technology continued to improve, with processor designs now using a combination of RISC, pipelining, and parallel processing.
- Wide area networks, network bandwidth and speed of operation and networking capabilities have kept developing tremendously.
- Personal computers (PCs) now operate with Gigabit per second processors, multi-Gigabyte disks, hundreds of Mbytes of RAM, color printers, high-resolution graphic monitors, stereo sound cards and graphical user interfaces.
- Thousands of software (operating systems and application software) are existing today and Microsoft Inc. has been a major contributor. Microsoft is said to be one of the biggest companies ever, and its chairman – Bill Gates has been rated as the richest man for several years.

- Finally, this generation has brought about **micro controller technology**. Micro controllers are embedded inside some other devices so that they can control the features or actions of the product.
- They work as small computers inside devices and now serve as essential components in most machines.

1.2.2 Great Ideas in Computer Architecture

The ideas that marked tremendous improvement in the field of computer architecture are briefly discussed here.

な. Moore’ s Law

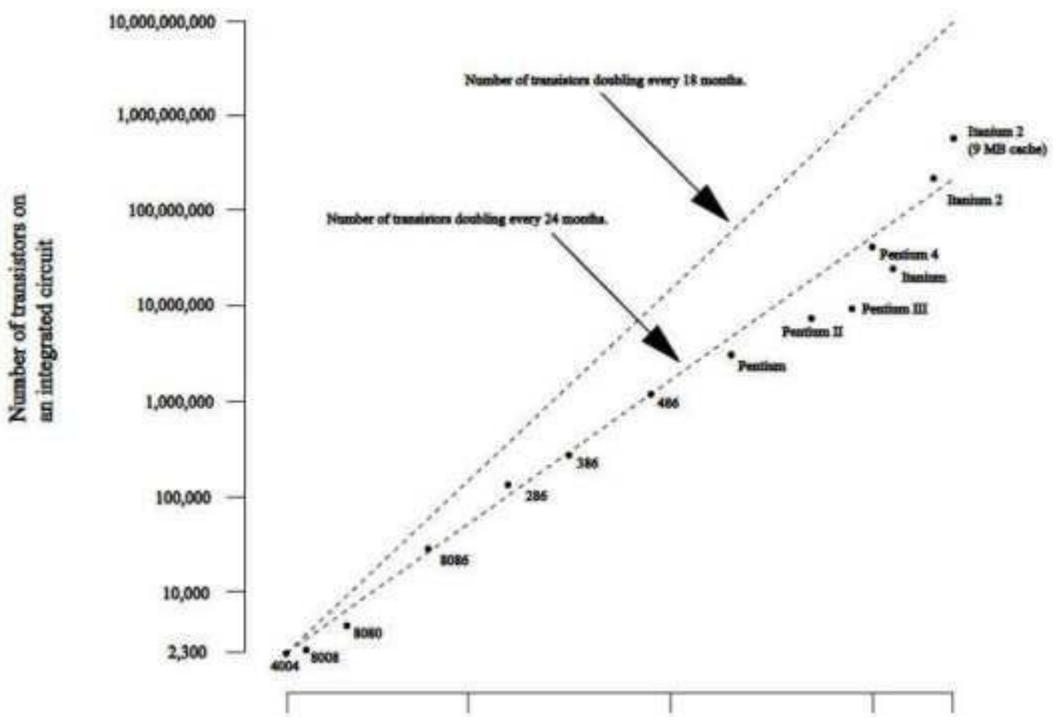


Fig な.に: Illustration of Moore’ s Law

Moore's law states that the numbers of transistors will double every 18 months.

It is an observation that the number of transistors in a dense integrated circuit doubles about every two years. It is an observation and projection of a historical trend and not a physical or natural law.

2. Abstract Design

It is a major productivity technique for hardware and software. Abstractions are used to represent the design at different levels of representation. The detailed lower-level design details from the higher levels.

3. Performance through parallelism

Parallelism executes programs faster by performing several computations at the same time. This requires hardware with multiple processing units. The overall performance of the system is significantly increased by performing operations in parallel.

4. Performance through Pipelining

Pipelining increases the CPU instruction throughput. **Throughput** is a performance metric which is the number of instructions completed per unit of time. But it does not reduce the execution time of an individual instruction. It increases the execution time of each instruction due to overhead in the pipeline control. The increase in instruction throughput means that a program runs faster and has lower total execution time.

5. Make the Common Case Fast

Making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance. In making a design trade-off, favor the frequent case over the infrequent case. Amdahl's Law can be used to quantify this principle. This also applies when determining how to spend resources, since the impact on making some occurrence faster is higher if the occurrence is frequent. This will:

- Helps performance
- Is simpler and can be done faster

6. Performance via prediction

The computer can perform better (on average) by making rational guesses on the decisions. Instead of wasting clock cycles for certain results, the computers can remarkably improve the performance

7. Hierarchy of memories

Programmers want memory to be fast, large, and cheap. The memory speed is a primary factor in determining the performance of the system. The memory capacity limits the size of problems that can be solved.

Architects have found that hierarchy of memories will be a solution for all these issues. The fastest, smallest, and most expensive memory per bit is placed the top of the hierarchy and the slowest, largest, and cheapest per bit is at the bottom. **Caches** give the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy.

8. Dependability via Redundancy

Computers need to be fast and dependable. Since any physical device can fail, we make systems dependable by including redundant components that can take over when a failure occurs and help detect failures. Restoring the state of the system is done by redundancy.

1.2.3 Technologies

Up until the early なひばどs computers used magnetic core memory, which was slow, cumbersome, and expensive and thus appeared in limited quantities. The situation improved with the introduction of transistor-based dynamic random-access memory (DRAM, invented at IBM in 1966) and static random-access memory (SRAM). A **transistor** is simply an on/off switch controlled by electricity. The **integrated circuit** (IC) combined dozens to hundreds of transistors into a simple chip. **Very large-scale integrated** (VLSI) circuit is a device containing hundreds of thousands to millions of transistors.

Manufacturing of IC:

Integrated circuits are chips manufactured on silicon wafers. Transistors are placed on wafers through a chemical etching process. Each wafer is cut into chips which are packed individually.

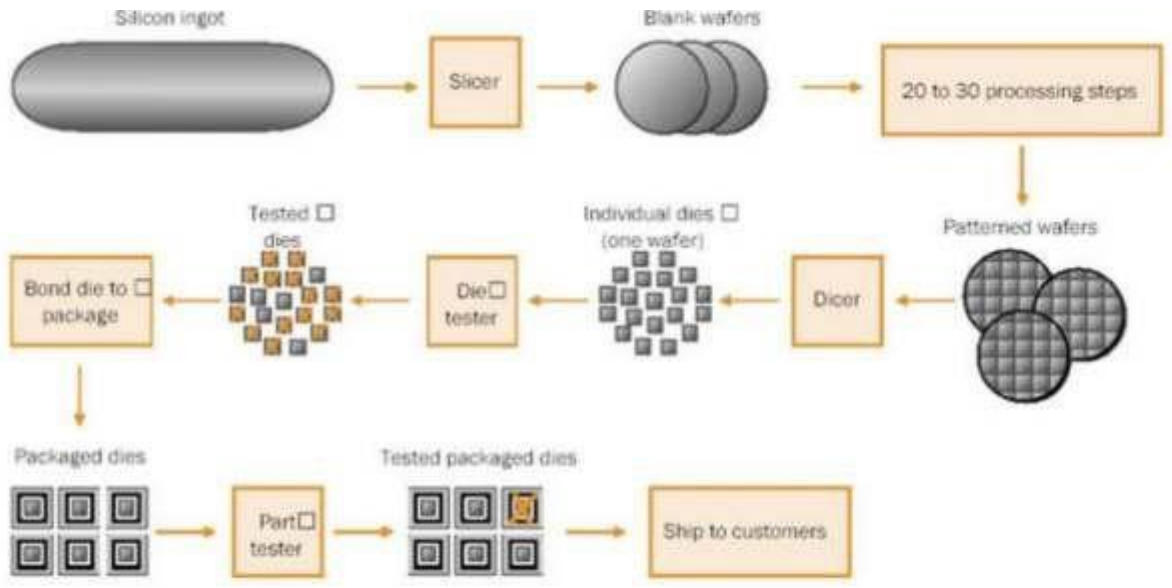


Fig 1.3: Chip manufacturing process

After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers. These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Then, the wafers are diced into dies. The good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers.

Cost of an IC is found from:

- Cost per die= (cost per wafer) / ((dies per wafer)*yield) Yield refers the fraction of dies that pass testing.
- Dies / wafer= wafer area / die area
- Yield=1 / (1 + (defects per area * die area)/2 $)^2$

Programmable Logic Device (PLD)

A programmable logic device (PLD) is an electronic component used to build reconfigurable digital circuits. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed, that is, reconfigured.

The major limitations of PLD:

- Consume space due to large number of switches for programmability
- Low speed due to the presence of many switches.

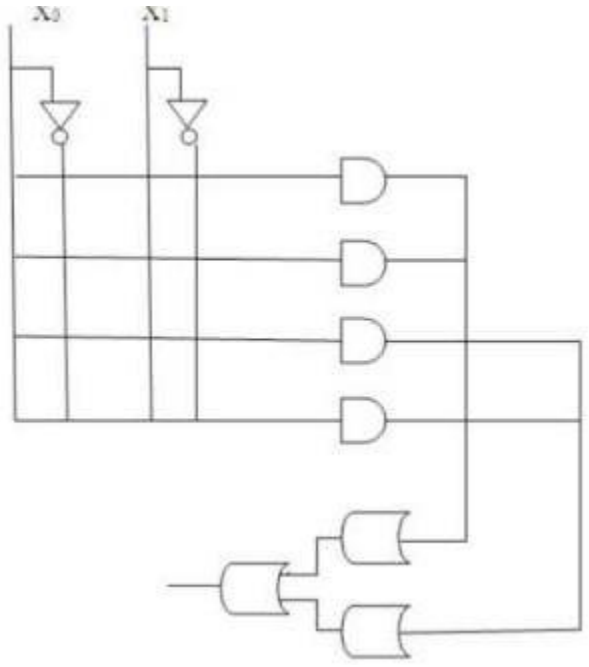


Fig 1.4: Programmable Logic Device

Custom chips

An Application-Specific Integrated Circuit (ASIC) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. Application-Specific Standard Products (ASSPs) are intermediate between ASICs and industry standard integrated circuits.

1.2.4 Performance

Elapsed time and throughput are two different ways of measuring speed.

- **Elapsed time** or wall-clock time or response time is the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead. It is the better measure for processor speed because it is less dependent on other system components.
- **CPU execution time** is the actual time the CPU spends computing for a specific task.
- The **User CPU time** is the CPU time spent in a program itself. **System CPU time** is the CPU time spent in the operating system performing tasks on behalf of the program.

- The **CPU Performance** equation (CPU Time) is the product of number of instructions executed, Average CPI of the program and CPU clock cycle.

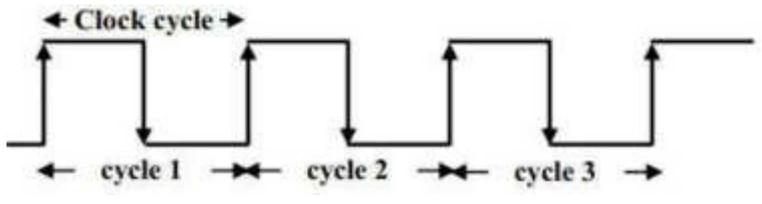
$$\textit{CPU Time} = \frac{\textit{Seconds}}{\textit{Program}} \times \frac{\textit{Instructions}}{\textit{Program}} \times \frac{\textit{Cycles}}{\textit{Instruction}} \times \frac{\textit{Seconds}}{\textit{Cycle}}$$

- Performance is inversely proportional to execution time. Performance ratios are inverted from time ratios.

$$\textit{Performance improve mentation} = \frac{\textit{Performance after change}}{\textit{Performance before change}} \times \frac{\text{Execution } \textit{time before change}}{\textit{Execution time after change}}$$

- Clock cycle is the time for one clock period, usually of the processor clock, which runs at a constant rate.
- Clock period is the length of each clock cycle.
- The CPU clock rate depends on CPU organization and hardware implementation.

$$\textit{Clock Rate} = \frac{1}{\textit{Clock Cycle}}$$



- **Cycles per Instruction (CPI)** is count of clock cycles taken by an instruction to complete its execution.

$$\textit{Instructions per Cycle (IPC)} = \frac{1}{\textit{Cycles per Instruction}}$$

- Performance is improved by reducing number of clock cycles, increasing clock rate and hardware designer must often trade off clock rate against cycle count.
- Workload is a set of programs run on a computer that is either the actual collection of applications run by a user or is constructed from real programs to approximate such a mix. A typical workload specifies both the programs as well as the relative frequencies.
- To evaluate two computer systems, a user would simply compare the execution time of the workload on the two computers.
- Alternatively, set of benchmarks containing several typical engineering or scientific applications can be used. A CPU benchmark (CPU benchmarking) is a series of tests designed to measure the performance of a computer or device CPU. A set of standards, or baseline measurements are used to compare the performance of different systems, using the same methods and circumstances.

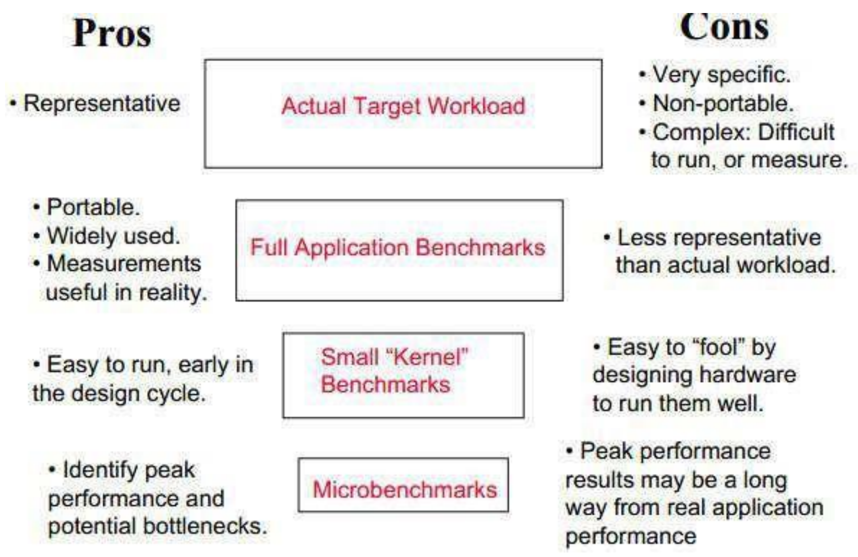

Pros
Cons
• Representative
Actual Target Workload
• Very specific.
• Non-portable.
• Complex: Difficult to run, or measure.
• Portable.
• Widely used.
• Measurements useful in reality.
Full Application Benchmarks
• Less representative than actual workload.
• Easy to run, early in the design cycle.
Small "Kernel" Benchmarks
• Easy to "fool" by designing hardware to run them well.
• Identify peak performance and potential bottlenecks.
Microbenchmarks
• Peak performance results may be a long way from real application performance

Fig 1.6: Types of Benchmark Programs

- The use of benchmarks whose performance depends on very small code segments encourages optimizations in either the architecture or compiler that target these segments.
- The arithmetic mean is proportional to execution time, assuming that the programs in the workload are each run an equal number of times.
- **Weighted arithmetic mean** is an average of the execution time of a workload with weighting factors designed to reflect the presence of the programs in a workload; computed as the sum of the products of weight.

Example 1.1: For a given program, the execution time on machine A is 1s and on B is 10s. Find the performance or speed up of the machines.

Execution $_A$= 1s

Execution $_B$=10s

$$Speedup = \frac{Performance\ of\ A}{Performance\ of\ B} x \frac{Execution\ of\ B}{Execution\ of\ A}$$

Speedup=10/1=10

The performance of machine A is 10 times faster than that of B.

Example 1.2: For a certain program with 1,00,00,000 instructions, find the execution time given the average CPI is 2.5 cycles/instruction and clock rate as 200MHz.

Number of instructions=1,00,00,000

Average CPI=2.5 cycles/ instruction

Clock rate=200MHz =200000000 Hz

Clock cycle=1/Clock rate=1/ 200000000= 5 x 10^{-9}s

$$CPU\ Time = \frac{Seconds}{Program} ⍰ \frac{Instructions}{Program} x \frac{Cycles}{Instruction} x \frac{Seconds}{Cycle}$$

CPU Time=10000000 x 2.5 x 5 x 10^{-9}

=0.125 s

Example 1.3: For a certain program with 1,00,00,000 instructions has an average CPI is 2.5 cycles/instruction and clock rate as 200MHz. When a new optimization complier is deployed, the instruction count was reduced to 95,00,000 with new CPI=3.0 cycles/instruction at modified clock rate of 300MHz. Find the speedup.

$$\text{Speedup} = \frac{\textit{Old Execution Time}}{\textit{New Execution Time}} \text{ x } \frac{I_{old} \text{ x } CPI_{old} \text{ x } \textit{Clock cycle}_{old}}{I_{new} \text{ x } CPI_{new} \text{ x } \textit{Clock Cycle}_{new}}$$

$(10000000 \text{ x } 2.5 \text{ x } 5 \text{ x } 10^{-9}) \;/\; (9500000 \text{ x } 3 \text{ x } 3.33 \text{ x } 10^{-9})$

 1.315

The new compiler is 1.315 times faster than the old one.

Example 1.4: A program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which with run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 time as many clock cycles as computer A for this program. What clack rate should we tell the designer to target?

Clock rate of B= Clock Cycles $_B$ / CPU Time $_B$

= 1.2 x Clock Cycles $_A$ / 6 Clock Cycles $_A$= CPU Time $_A$ x Clock Rate $_A$

 10 x 2 = 20 x 10^9

Clock Cycles $_B$ = 1.2 x 20 x 10^9 / 6

= 4 GHz

Example 1.5: Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

Computer A: Cycle Time = 250ps, CPI = 2.0

Computer B: Cycle Time = 500ps, CPI = 1.2

CPU Time=Instruction Count x CPI $_A$ x Cycle Time $_A$

= I x 2.0 x 250= I x 500

CPU Time=Instruction Count x CPI $_B$ x Cycle Time $_B$

= I x1.2x 500= Ix 600

$$\frac{CPU\ time_B}{CPU\ Time_A} = 1.2$$

1.2.5 Power wall

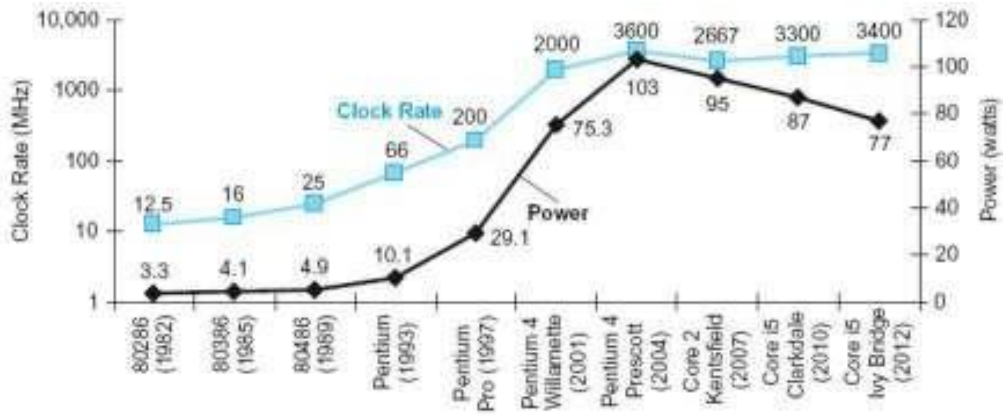


Fig 1.7: Clock rate and Power

- **Power wall** refers to the representational wall signifying the peak power constraint of a system.
- Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years is shown in Fig 1.7.
- The Pentium 4 made a dramatic jump in clock rate and power but less so in performance.
- The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip.
- Continuous technology scaling like reduction of the transistor feature sizes makes it possible to pack more transistors in a given chip die area.
- Reduced supply voltage, simultaneous switching of these transistor devices causes a tremendous increase in the power density, leading to the power wall disaster.

- An increase in the power density increases the chip temperature, which slows down the transistor switching rate and hence, the overall speed of the computer.
- Cooling solutions are very expensive, and hence, computer architects have focused on innovating device, circuit and architecture level techniques to combat power wall.
- **Dynamic voltage and frequency scaling** are solutions for these problems. Here the operating voltage and frequency of the chip are dynamically controlled based on the chip activity.
- In CMOS (complementary metal oxide semiconductor) IC technology

$$\boxed{\textbf{Power} = \textbf{Capacitive load} \times \textbf{Voltage}^2 \times \textbf{Frequency}}$$

Example 1.6: Suppose we developed a new, simpler processor that has 85% of the capacitive load of the more complex older processor. Further, assume that it has adjustable voltage so that it can reduce voltage 1 5% compared to processor B, which results in 15% shrink in frequency. What is the impact on dynamic power? Given: 85% of capacitive load of old CPU, 15% voltage reduction, 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{(C_{old} \times 0.85) \times (V_{old} \times 0.85)^2 \times (F_{old} \times 0.85)}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

The new processor uses 0.52 the power of the old processor.

1.2.6 from Uniprocessors to Multiprocessors

The performance of the computers has drastically increased when the technology has drifted from uniprocessor systems to multiprocessor system. As the core computing units were made more powerful, the performance of the processors also increased significantly.

Uniprocessor system is a type of architecture that is based on a single computing unit. All the operations were done sequentially on the same unit. Multiprocessor systems are based on executing instructions on multiple computing units.

The multiprocessor architectures, is based on Flynn Taxonomy.

Flynn's taxonomy is a classification of parallel computer architectures that are based on the number of concurrent instruction and data streams available in the architecture.

Single Instruction, Single Data (SISD):

- This is a uniprocessor machine which is capable of executing a single instruction, operating on a single data stream.
- The machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers.
- Most conventional computers have SISD architecture.
- All the instructions and data to be processed have to be stored in primary memory.
- The speed of the processing element in the SISD model is limited by the rate at which the computer can transfer information internally.

Multiple Instruction, Single Data (MISD):

- An MISD computing system is a multiprocessor machine capable of executing different instructions on different Processing Elements but all of them operating on the same dataset.

Single Instruction, Multiple Data (SIMD):

- This machine capable of executing the same instruction on all the CPUs but operating on different data streams.
- Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. So that the information can be passed to all the Processing Elements (PEs) organized data elements of vectors can be divided into multiple sets and each PE can process one data set.

Multiple Instruction, Multiple Data (MIMD):

- This is capable of executing multiple instructions on multiple data sets.
- Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable to any kind of application.
- Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.

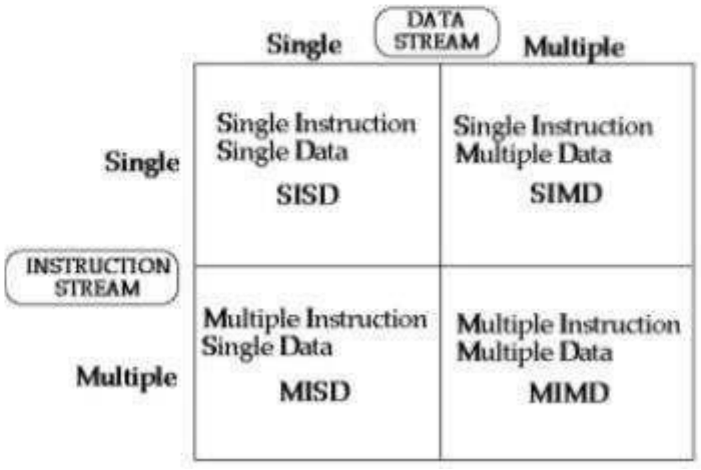

DATA STREAM
Single
Multiple
INSTRUCTION STREAM
Single
Multiple
Single Instruction Single Data
SISD
Single Instruction Multiple Data
SIMD
Multiple Instruction Single Data
MISD
Multiple Instruction Multiple Data
MIMD

Fig 1.8: Flynns Taxonomy

Apart from these architectures, MIPS Technologies developed a Microprocessor without Interlocked Pipeline Stages on Reduced Instruction Set Computer (RISC).

Concern for Power

- The power limit has forced a dramatic change in the design of microprocessors. Since 2002, the rate has slowed from a factor of 1.5 per year to a factor of 1.2 per year.
- Most of the desktop manufacturing companies are shipping microprocessors with multiple processors per chip, where the benefit increased throughput than on response time. This is done at the cost of increase in power.
- To reduce confusion between the words processor and microprocessor, companies refer to processors as cores and such microprocessors are generically called **multicore microprocessors.**
- A **quad core** microprocessor is a chip that contains four processors or four cores.
- In the past, programmers could rely on innovations in hardware, architecture, and compilers to double performance of their programs every 18 months without having to change a line of code.
- Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of multiple processors.

- Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve performance of their code as the number of cores increases.

1.3 ADDRESSING AND ADDRESSING MODES

Each instruction of a computer specifies an operation on certain data.

The different ways in which the location of an operand is specified in an instruction is called as Addressing mode.

Different operands will use different addressing modes. One or more bits in the instruction format can be used as mode field. The value of the mode field determines which addressing mode is to be used. The effective address will be either main memory address of a register.

The most common addressing modes are:

1. Immediate addressing mode
2. Direct addressing mode
3. Indirect addressing mode
4. Register addressing mode
5. Register indirect addressing mode
6. Displacement addressing mode
7. Stack addressing mode

□ Immediate Addressing:

- This is the simplest form of addressing. Here, the operand is given in the instruction.
- This mode is used to define constant or set initial values of variables.
- The advantage of this mode is that no memory reference other than instruction fetch is required to obtain operand.
- The disadvantage is that the size of the number is limited to the size of the address field because most instruction sets is small compared to word length.

- **Example:** ADD 3
- Adds 3 to contents of accumulator and 3 is the operand.

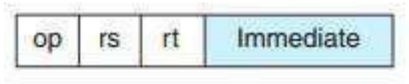


Fig 1.9: Immediate Mode

- **Direct Addressing:**
  - In direct addressing mode, effective address of the operand is given in the address field of the instruction.
  - It requires one memory reference to read the operand from the given location and provides only a limited address space.
  - Length of the address field is usually less than the word length.
  - **Example :** Move P, Ro

    Add Q, Ro

    Where P and Q are the address of operand, R_o is any register. Sometimes Accumulator (AC) is the default register. Then the instruction will look like:

    Add A

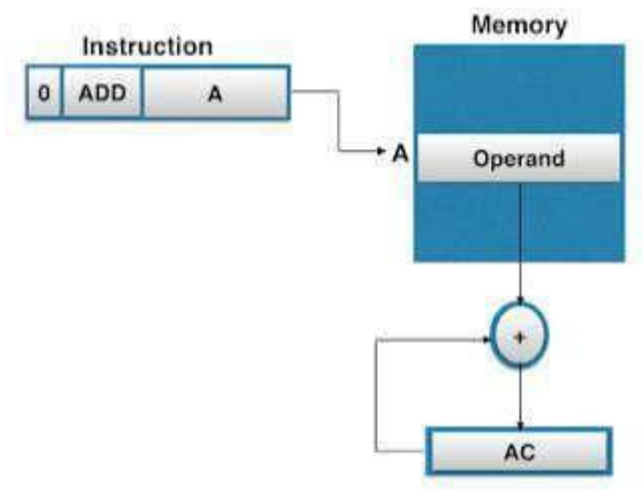


Fig 1.10: Direct Addressing modes

□ Indirect or Pseudo direct Addressing:

- Indirect addressing mode, the address field of the instruction refers to the address of a word in memory, which in turn contains the full length address of the operand.
- The address field of instruction gives the memory address where on, the operand is stored in memory.
- Control fetches the instruction from memory and then uses its address part to access memory again to read Effective Address.
- The advantage of this mode is that for the word length of N, an address space of 2N can be addressed.
- The disadvantage is that instruction execution requires two memory references to fetch the operand.
- Multilevel or cascaded indirect addressing can also be used.
- **Example:** Effective Address (EA) = (A).
- The operand will be present in the memory location A.

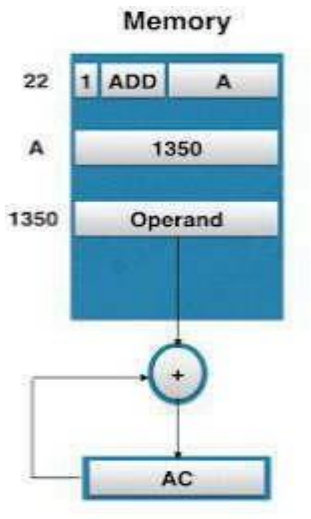


Fig 1.11: Indirect Addressing Modes

□ Register Addressing:

- Register addressing mode is similar to direct addressing. The only difference is that the address field of the instruction refers to a register rather than a memory location.

- 3 or 4 bits are used as address field in the instruction to refer 8 to 16 generate purpose registers (GPR).
- The operands are in registers that reside within the CPU.
- The instruction specifies a register in CPU, which contain the operand.
- There is no need to compute the actual address as the operand is in a register and to get operand there is no memory access involved.
- The advantages of register addressing are small address field is needed in the instruction and faster instruction fetch.
- The disadvantages includes very limited address space and usage of multiple registers helps in performance but it complicates the instructions.

- **Example:** MOV AX, BX

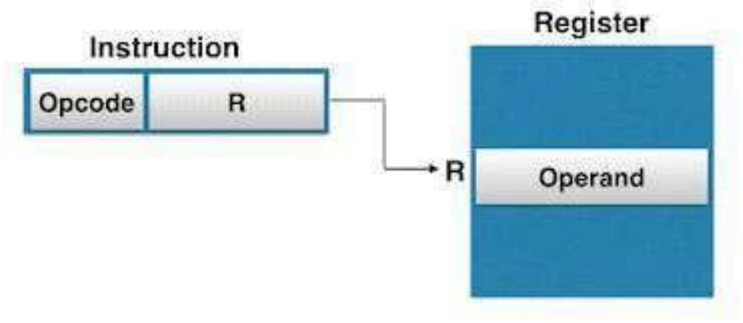


Fig 1.12: Register Mode

- **Register Indirect Addressing:**

- This mode is similar to indirect addressing. The address field of the instruction refers to a register.
- The instruction specifies a register in CPU whose contents give the operand in memory.
- The selected register contain the address of operand rather than the operand itself.
- The register contains the effective address of the operand. This mode uses one memory reference to obtain the operand.

- Control fetches instruction from memory and then uses its address to access Register and looks in Register(R) for effective address of operand in memory.
- The address space is limited to the width of the registers available to store the effective address.
- **Example: MOV AL, [BX]**

  Code example in Register:

  MOV BX, 1000H

  MOV 1000H, operand
- The instruction (MOV AL, [BX]) specifies a register [BX] which contain the address of operand (1000H) rather than address itself.

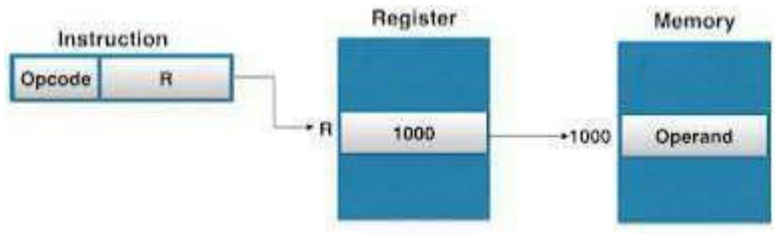


Fig 1.13: Register Indirect Mode

- **Displacement Addressing:**
  - It is a combination of direct addressing or register indirect addressing mode.
  - Displacement Addressing Modes requires that the instruction have two address fields, at least one of which is explicit means, one is address field indicate direct address and other indicate indirect address.
  - Value contained in one addressing field is A, which is used directly and the value in other address field is R, which refers to a register whose contents are to be added to produce effective address.
  - **Example:** EA=A+(R)

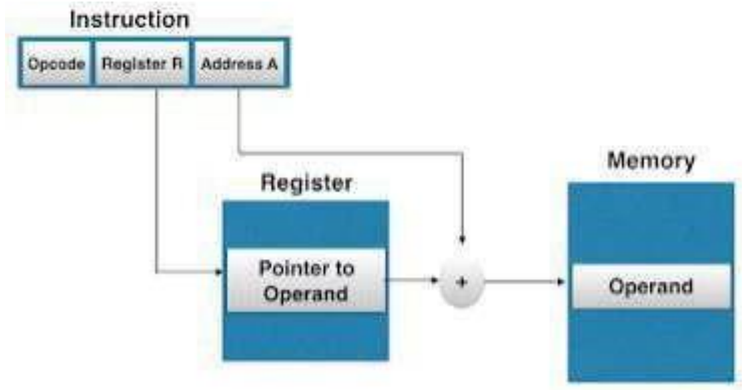


Fig 1.14 a): Displacement Addressing Modes

- In displacement addressing mode there are 3 types of addressing mode.
- **Relative addressing:**

  The contents of program counter is added to the address part of instruction to obtain the Effective Address. The address field of the instruction is added to implicitly reference register Program Counter to obtain effective address.

  Example: EA=A+PC

  Assume that PC contains the value 825 and the address part of instruction contain the value 24, then the instruction at location 825 is read from memory during fetch phase and the Program Counter is then incremented by one to 826. Here both PC and instruction contains address. The effective address computation for relative address mode is 826+24=850

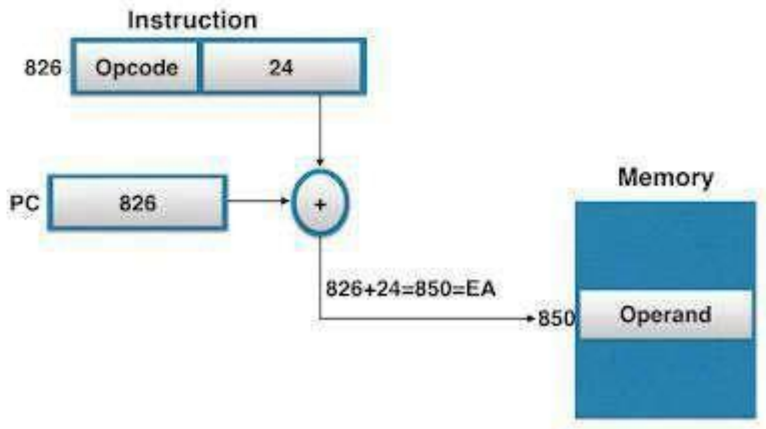


Fig 1.14 b): Relative addressing

 Base register addressing

The content of the Base Register is added to the direct address part of the instruction to obtain the effective address. The address field point to the Base Register and to obtain EA, the contents of Instruction Register, is added to direct address part of the instruction. This is similar to indexed addressing mode except that the register is now called as Base Register instead of Index Register.

Example: EA=A +Base

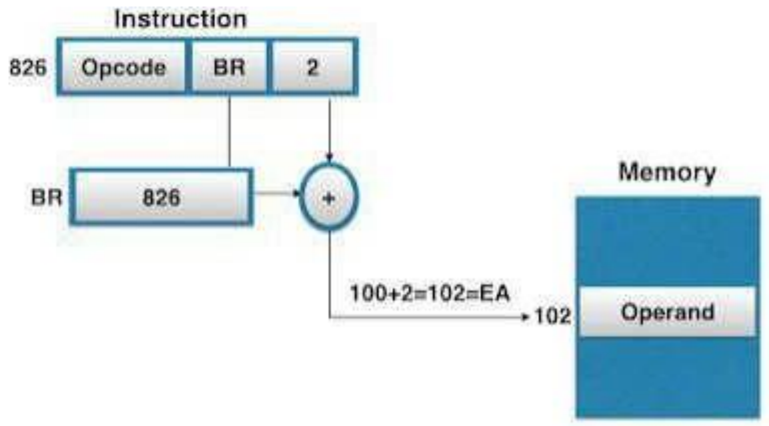


Fig 1.14 c): Base Register Addressing Mode

 Indexed addressing:

The content of Index Register is added to direct address part of instruction to obtain the effective address. The register indirect addressing field of instruction point to Index Register, which is a special CPU register that contain an Indexed value, and direct addressing field contain base address.

The data array is in memory and each operand in the array is stored in memory relative to base address. The distance between the beginning address and the address of operand is the indexed value stored in indexed register.

Any operand in the array can be accessed with the same instruction, which provided that the index register contains the correct index value i.e., the index register can be incremented to facilitate access to consecutive operands.

Example: EA=A+ Index

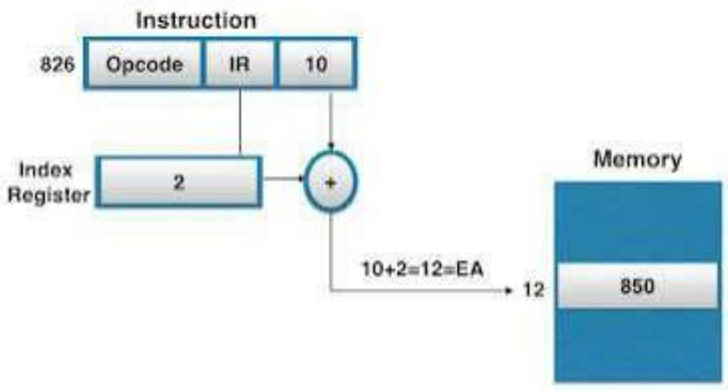


Fig 1.14d): Indexed Addressing

□ **Stack Addressing:**

- Stack is a linear array of locations referred to as last-in first out queue.
- The stack is a reserved block of location, appended or deleted only at the top of the stack.
- Stack pointer is a register which stores the address of top of stack location.
- This mode of addressing is also known as **implicit addressing**.
- Example: Add
- This instruction pops two items from the stack and adds.

Additional Modes:

There are two additional modes. They are:

- Auto-increment mode
- Auto-decrement mode

These are similar to Register indirect Addressing Mode except that the register is incremented or decremented after (or before) its value is used to access memory. These modes are required because when the address stored in register refers to a table of data in memory, then it is necessary to increment or decrement the register after every access to table so that next value is accessed from memory.

Auto-increment mode:

- Auto-increment Addressing Mode are similar to Register Indirect Addressing Mode except that the register is incremented after its value is loaded (or accessed) at another location like accumulator (AC).
- The Effective Address of the operand is the contents of a register in the instruction.
- After accessing the operand, the contents of this register is automatically incremented to point to the next item in the list.
- **Example:** (R) +.
- The contents in register R will be accessed and them it will be incremented to point the next item in the list.

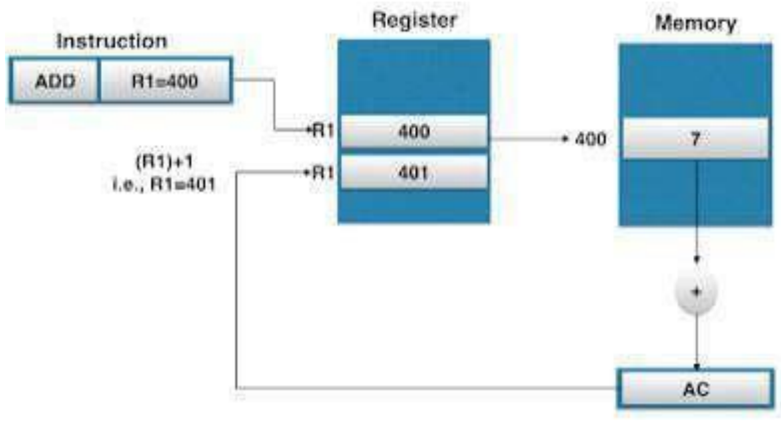


Fig 1.16: Auto-increment Mode

- The effective address is (R) =400 and operand in AC is 7. After loading R1 is incremented by 1, it becomes 401.

Auto-decrement mode:

- Auto-decrement Addressing Mode is reverse of auto-increment, as in it the register is decrement before the execution of the instruction.
- Effective address is equal to EA=(R) - 1
- The Effective Address of the operand is the contents of a register in the instruction.
- After accessing the operand, the contents of this register is automatically decremented to point to the next item in the list.

- **Example**: - (R)
- The contents in register R will be decremented and then it is accessed.

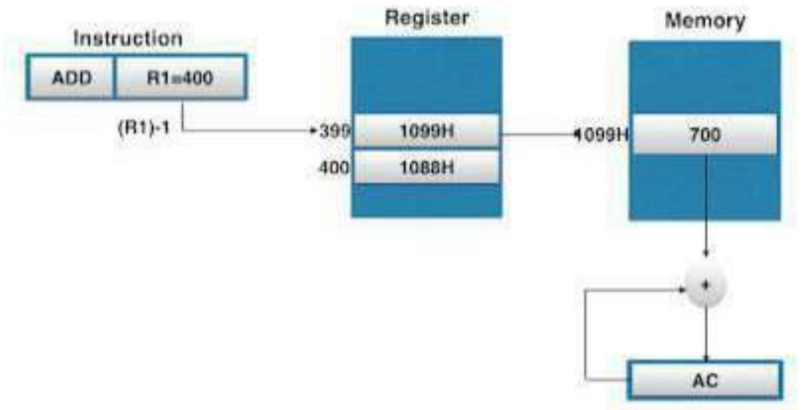


Fig 1.17: Auto Decrement Addressing Mode

1.4 INSTRUCTIONS

An instruction is a binary code, which specifies a basic operation for the computer.

- Operation Code (op code) defines the operation type. Operands define the operation source and destination.
- **Instruction Set Architecture** (ISA) describes the processor in terms of what the assembly language programmer sees, i.e. the instructions and registers.
- The op codes and operands follows Stores Program Concept.

Stored Program Concept is an idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

1.4.1 Operations

- The computer performs the arithmetic through operations.
- The MIPS arithmetic instruction performs only one operation and must always have exactly three variables.

  Example: Add a, b, c

  Adds b and c and stores the sum in a.

- The hardware for a variable number of operands is more complicated than hardware for a fixed number.
- It is always essential to design the instructions with same number of operands so as to simplify the hardware requirement.

1.4.2 Operands

- The operands of arithmetic instruction must be from specially built memory locations called **registers.**
- The registers are accessed as 32 bit groups termed as **words**. MIPS architecture supports 32 registers.

Memory Operands

- The operands are always stored in registers.
- Data transfer instruction is a command that moves data between memory and registers.
- Address of an operand is a value used to delineate the location of a specific data element within a memory array.
- The data transfer instruction that copies data from memory to a register is traditionally called **load (lw- load word)**.
- The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory.
- The sum of the constant portion of the instruction and the contents of the second register forms the memory address.
- **Store (sw- store word)** instruction copies data from a register to memory.
- The format of a store is the name of the operation, followed by the register to be stored, then offset to select the array element, and finally the base register.
- The MIPS address is specified in part by a constant and in part by the contents of a register.

- Many programs have more variables than computers have registers. The compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory.
- The process of putting less commonly used variables into memory is called **spilling registers.**

Constant or Immediate Operands

- Sometimes it is necessary to load a constant from memory to use one. The constants would have been placed in memory when the program was loaded.

  Example: add I$s3,$s3,10
- This instruction is interpreted as addition of content of $s3 and the value 10. The sum is stored in $s3. Add I means add immediate, since one of the operand is in immediate addressing mode.
- As per the design principle ¦Make common case faster¦, the constant operands must be loaded faster from the memory.
- Since constants occur more frequently in the instruction, they are mentioned in the instruction itself rather than to load from registers.

Name	**Example**	**Comments**
32 Registers	$S0, $S1… $t0, $t1…	They can be accessed quickly. In MIPS architecture, the data must be loaded into the register to perform arithmetic operation.
2^{30} memory words	Memory[0], Memory [1]…	The contents can be accessed only after data transfer instructions. MIPS use byte addressing.

Category	**Instruction**	**Operation**
Arithmetic	Add $s1, $s2, $s3	S1=s2+s3. There are three operands in this instruction. The data resides in the registers.
	Sub $s1, $s2, $s3	S1=s2-s3. There are three operands in this instruction. The data resides in the registers.

	Addi $s1, $s2, 50	S1=s2+50. This is add immediate instruction. It has two operands and one constant value, which is directly added to get the result.
Data Transfer	Lw $s1, 50($s2) Sw$s1,50($s2)	S1=memory [s2+50] Data is transferred from memory to registers. Memory[s2+50]=$s1 Data is transferred from register to memory.

1.4.3 Representation of Instructions

- Numbers are represented in computer hardware as a series of high and low electronic signals that are denoted as 1s and 0s. Hence they are considered base 2 numbers.

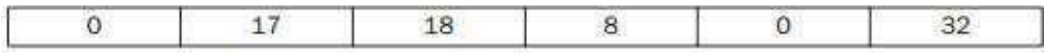
A bit or binary digit is a single digit of a binary number and is the smallest indivisible unit of computing.

- The binary digit may be used to denote high or low, on or off, true or false, or 1 or 0.

- Registers are part of every instruction, hence there must be a convention to map register names into numbers.

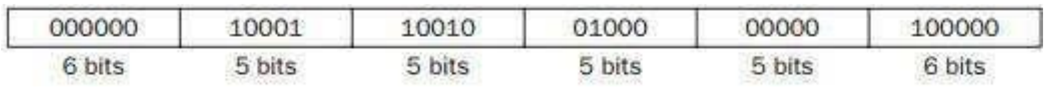
  Example: add $t0,$s1,$s2.

  This instruction is mapped to its equivalent decimal representation as: The binary



0	17	18	8	0	32

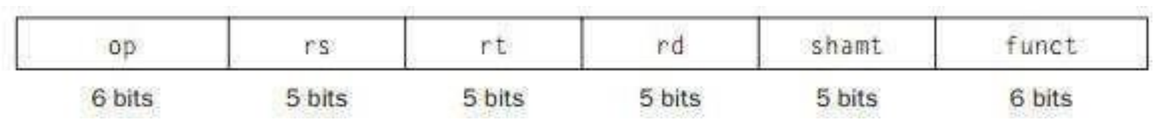
equivalent representation is given as:



000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Each cell is termed as a **field.**
- The binary representation used for communication within a computer system is termed as **Machine Language.**
- **Instruction Format** is a representative form an instruction of fields of binary numbers.

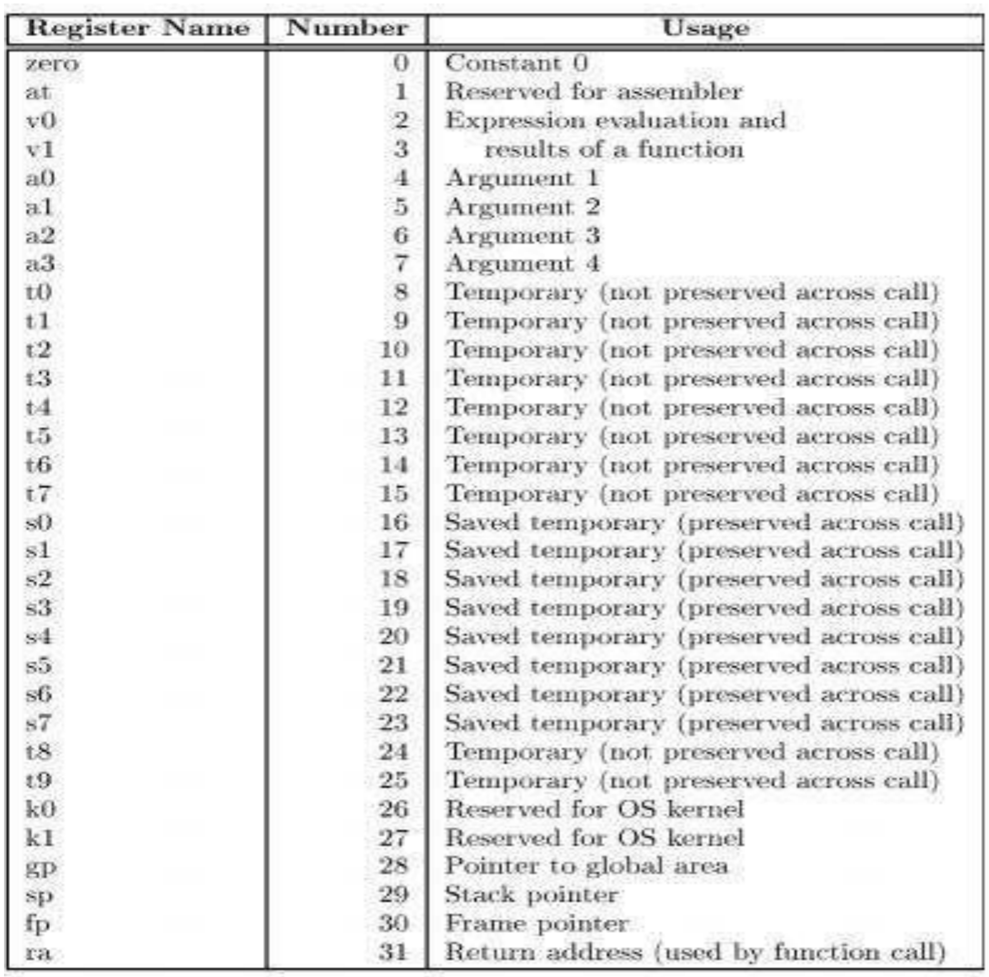
Fields in MIPS



op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Op code is the field that denotes the operation and format of an instruction.

- rs: The first register source operand.
- rt: The second register source operand.
- rd: The register destination operand. It gets the result of the operation.
- shamt: Shift amount. This is done to adds two zeros to the low-order end of the sign- extended offset field in calculating the address. This operation truncated the sign values.
- op field and is sometimes called the function code.
- The MIPS instructions are designed in the same format for easy manipulation this is in accordance with the design principle Good design demands good compromises.



Register Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and
v1	3	results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)
t3	11	Temporary (not preserved across call)
t4	12	Temporary (not preserved across call)
t5	13	Temporary (not preserved across call)
t6	14	Temporary (not preserved across call)
t7	15	Temporary (not preserved across call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved across call)
t9	25	Temporary (not preserved across call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

Fig 1.18: Mapping of register names and numbers

Op code values of MIPS instruction

In the MIPS instruction reg means a register number ranging from 0 and 31. Address means a 16-bit address, and not applicable (n.a.) means this field does not appear in this format. The add and sub instructions have the same value in the op field. The hardware uses the funct field to decide the whether it is addition or subtraction operation using: add (32) or subtract (34).

Instruction	Format	Op	Rs	Rt	Rd	Shamt	Funct	Address
Add	R	0	Reg	Reg	Reg	0	32_{10}	Na
Sub	R	0	Reg	Reg	Reg	0	$^{34}10$	Na
Add immediate	I	$^{8}10$	Reg	Reg	Na	Na	Na	Constant
Lw	I	35_{10}	Reg	Reg	Na	Na	Na	Address
Sw	I	43_{10}	Reg	Reg	Na	Na	Na	Address

1.4.4 Logical Operations

The following are the logical operations performed by the processor:

Logical Operations	MIPS Instructions
Shift left	sll
Shift right	srl
Bit by bit AND	and, andi
Bit by bit OR	or, ori
Bit by bit NOT	nor

The first class of such operations is called **shifts.** They move all the bits in a word to the left or right, filling the emptied bits with 0s.

0000 0000 0000 00000 000 0000 0000 0000 1001_2= 9_{10} After left shifting by four, the new value is 144.

0000 0000 0000 0000 0000 0000 0000 1001 0000_2= 144_{10}

- **Left shift:** Left shifting by i bits is equivalent to multiplying the number by 2^i.
- **Right Shift**: Right shifting by i bits is equivalent to dividing the number by 2^i.
- **AND:** This is used in masking of bits.
- **OR:** It is a bit-by-bit operation that places a 1 in the result if either operand bit is a 1
- **NOT:** A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.
- **NOR:** A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands.

Category	Instruction	Operation
AND	and $s1, $s2, $s3	S1=s2&s3
OR	or $s1, $s2, $s3	S1=s2|s3
NOR	nor $s1, $s2, $s3	S1=~(s2|s3)
NAND	nand $s1, $s2, $s3	S1=~(s2&s3)
AND immediate	andi $s1, $s2, 100	S1=s2&100
OR immediate	Ori $s1, $s2, $s3	S1=s2|100
Shift leftlogical	Sll $s1, $s2, 10	S1=s2<<10
Shift right logical	Srl $s1, $s2, 10	S1=s2>>10

1.4.5 Control Operations

Decision making and branching makes the computers more powerful.

Decision Making:

Decision making in MIPS assembly language includes two decision-making instructions **(conditional branches):**

i) Branch if Equal (BEQ):

beq register1, register2, L1

In this instruction, the go to the statement labeled L1 if the value in register1 is equal to the value in register2.

ii) Branch if not Equal (BNE): bne register1, register2, L1

In this instruction, the go to the statement labeled L1 if the value in register1 does not equal the value in register2.

Conditional branch is an instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

Example:

Consider the following statement,

if (i == j) f = g + h; else f = g – h;

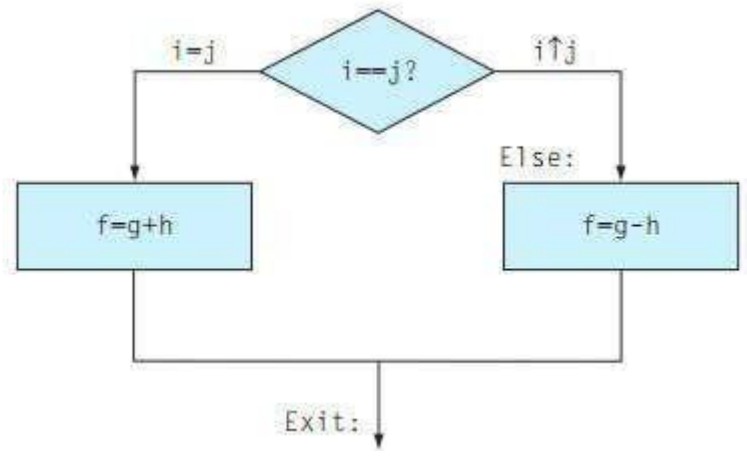


Fig 1.19: Flowchart for if (i == j) f = g + h; else f = g – h;

The instruction first compares for equality, using beq. In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent then part of if (the label Else is defined below):

bne $s3,$s4,Else # go to Else if i =j

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

add $s0,$s1,$s2 # f = g + h (skipped if i =j)

This instruction says that the processor always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is jump, abbreviated as j (the label Exit is defined below).

j Exit # go to Exit

The assignment statement in the else portion of if statement can again be compiled into a single instruction. We just need to append the label Else to this instruction. We also show the label Exit that is after this instruction, showing the end of the if-then-else compiled code:

Else: sub $s0, $ s1, $s2 # f = g – h (skipped if i = j)

Exit:

Compilers create branches and labels wherever necessary for maintaining flow of the program. Also, the assembler calculates the addresses and relieves the compiler and the assembly language programmer.

Looping:

When a set of statements has to be executed more number of times, looping statements are used.

Example:

while (save[i] == k)

i += 1;

i and k correspond to registers $s3 and $s5 and the base of the array save is in $s6. The MIPS instructions are:

- The first step is to load save[i] into a temporary register. This operation needs an address. Multiply the index i by 4 and add i to the base of array to obtain the address.
- Add the label Loop to it to branch back to that instruction at the end of the loop: *Loop:*

  *sll $t1,$s3,2 # Temp reg $t1 = 4 * i*
- To get the address of save[i] , add $t1 and the base of save in $s6:

  add $t1,$t1,$s6 # $t1 = address of save[i]
- Use that address to load save[i] into a temporary register:

  lw $t0,0($t1) # Temp reg $t0 = save[i]

D

- The next instruction performs the loop test, exiting if save[i] k: *bne $t0,$s5, Exit # go to Exit if save[i] k*
- The next instruction adds 1 to i :
  add $s3,$s3,1 # i = i + 1
- The end of the loop branches back to the while test at the top of the loop. Add the Exit label after it:
  j Loop # go to Loop

  Exit:

Grouping on instructions that makes compiling easy is through partitioning the assembly language instructions into basic blocks.

A sequence of instructions without branches except possibly at the end and without branch targets or branch labels except possibly at the beginning are called basic blocks.

Case / Switch Statements

These statements allow the programmers to select one among the many options. The simple way is to implement switch is through a sequence of conditional tests using a chain of if-then-else statements. The alternatives are encoded in **jump address table.** The program needs only to index into the table and then jump to the appropriate sequence.

A table of addresses of alternative instruction sequences is maintained in jump address table.

The jump table is an array of words containing addresses that correspond to labels in the code. MIPS include a **jump register instruction** (jr), to support the unconditional jump to the address specified in a register. The program loads the appropriate entry from the jump table into a register, and then it jumps to the proper address using a jump register.

		immediate)
Unconditional branch	J L	Goto L (Jump to target address L)

UNIT - II
ARITHMETIC

2.1 INTRODUCTION

Data is manipulated by using the arithmetic instructions in digital computers to give solution for the computation problems. The addition, subtraction, multiplication and division are the four basic arithmetic operations. **Arithmetic processing unit** is responsible for executing these operations and it is located in central processing unit.

The arithmetic instructions are performed on binary or decimal data**. Fixed-point numbers** are used to represent integers or fractions. These numbers can be signed or unsigned negative numbers. A wide range of arithmetic operations can be derived from the basic operations.

Signed and Unsigned Numbers:

Signed numbers:

These numbers require an arithmetic sign. The most significant bit of a binary number is used to represent the sign bit. If the sign bit is equal to zero, the signed binary number is positive; otherwise, it is negative. The remaining bits represent the actual number. The negative numbers may be represented either in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point

- Binary numbers signed magnitude
- Signed な|s complement
- Signed に|s complement

Unsigned binary numbers:

These are positive numbers and thus do not require an arithmetic sign. An m-bit unsigned number represents all numbers in the range 0 to 2^m ¦1. For example, the range of 16-bit unsigned binary numbers is from 0 to 65,53510 in decimal and from 0000 to FFFF16 in hexadecimal.

Signed Magnitude Representation:

The most significant bit (MSB) represents the *sign*. A 1 in the MSB bit position denotes a negative number and 0 denotes a positive number. The remaining *n* ・な bits are preserved and represent the magnitude of the number.

Examples:

Number	Signed Magnitude Representation
+3	0011
-3	1011
0	0000
-0	1011
5	0101
-5	1101

One’s Complement Representation:

)n one|s complement, positive numbers remain unchanged as before with the sign- magnitude numbers. Negative numbers are represented by taking the one|s complement (inversion, negation) of the unsigned positive number. Since positive numbers always start with a 0, the complement will always start with a 1 to indicate a negative number.

The one|s complement of a negative binary number is the complement of its positive counterpart, so to take the one|s complement of a binary number.

Number	One’s complement Representation
00001000 (+8)	11110111
10001000(-8)	01110111
00001100(+12)	11110011
10001100(-12)	01110011

Two’s Complement Representation:

)n two|s complement, the positive numbers are exactly the same as before for unsigned binary numbers. A negative number, is represented by a binary number, which when added to its corresponding positive equivalent results in zero.

2.4 **Arithmetic**

)n two|s complement form, a negative number is the に|s complement of its positive number with the subtraction of two numbers being A – B = A + ゅ に|s complement of B ょ using much the same process as before as basically, two|s complement is adding な to one|s complement of the number.

The main difference between 12 s complement and 22 s complement is that 12 s complement has two representations of 0 (+0): 00000000, and (-0): 11111111. In 22 s complement, there is only one representation for zero: 00000000 (0).

+0: 00000000

に|s complement of -0:

-0: 00000000 (Signed magnitude representation)

ななななななな ゅな|s complement representationょ

ななななななな + な= どどどどどどどど ゅに|s complement representationょ

These shows in に|s complement representation both +ど and -0 takes same value. This solves the **double-zero problem**, which existed in the な|s complement.

Example 2.1: Convert 2_{10} and -2_{10} to 32 bit binary numbers.

+2= **0**000 0000 0000 0010 (16 bits)

= 0000 0000 0000 0000 0000 0000 0000 0010 (32 bits)

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word.

2=0000 0000 00000010

-に=な|s complement of に +な

1111 1111 1111 11どな ゅな|s complement of にょ + 1

= **1**111 1111 1111 1110 (16 bits)

= 1111 1111 1111 1111 1111 1111 1111 1110 (32 bits)

To convert to 32 bit number copy the digit in the MSB of the 16 bit number for 16 times and fill the left half.

2.2 FIXED POINT ARITHMETIC

A fixed-point number representation is a real data type for a number that has a fixed number of digits after the radix point or decimal point.

This is a common method of integer representation is sign and magnitude representation. One bit is used for denoting the sign and the remaining bits denote the magnitude. With 7 bits reserved for the magnitude, the largest and smallest numbers represented are +127 and –127. Fixed-point numbers are useful for representing fractional values, usually in base 2 or base 10, when the executing processor has no floating point unit (FPU) or if fixed-point provides improved performance or accuracy for the application at hand. Most low-cost embedded microprocessors and microcontrollers do not have an FPU.

A value of a fixed-point data type is essentially an integer that is scaled by a specific factor. The scaling factor is usually a power of 10 (for human convenience) or a power of 2 (for computational efficiency). However, other scaling factors may be used occasionally, e.g. a time value in hours may be represented as a fixed-point type with a scale factor of 1/3600 to obtain values with one-second accuracy. The maximum value of a fixed-point type is the largest value that can be represented in the underlying integer type, multiplied by the scaling factor; and similarly for the minimum value.

Example:

The value 1.23 can be represented as 1230 in a fixed-point data type with scaling factor of 1/1000.

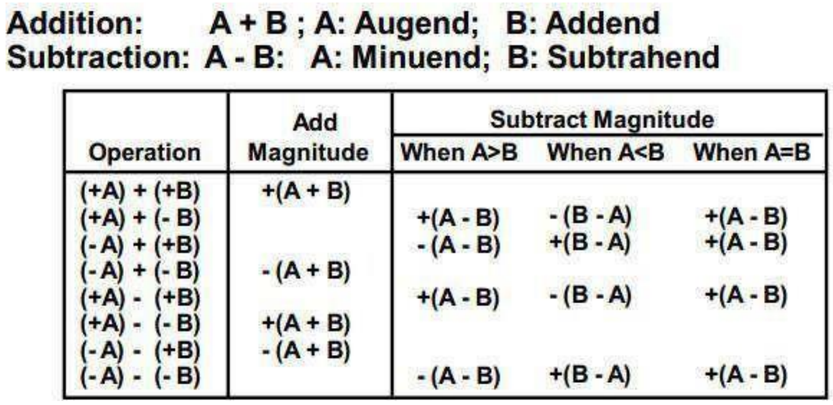
Precision loss and overflow

- The fixed point operations can produce results that have more bits than the operands there is possibility for information loss.
- In order to fit the result into the same number of bits as the operands, the answer must be rounded or truncated.
- Fractional bits lost below this value represent a precision loss which is common in fractional multiplication.
- If any integer bits are lost, however, the value will be radically inaccurate.
- Some operations, like divide, often have built-in result limiting so that any positive overflow results in the largest possible number that can be represented by the current format.

- Likewise, negative overflow results in the largest negative number represented by the current format. This built in limiting is often referred to as **saturation.**
- Some processors support a hardware overflow flag that can generate an exception on the occurrence of an overflow, but it is usually too late to salvage the proper result at this point.

2.2.1 Addition and Subtraction

In addition, the digits are added bit by bit from right to left, with carries passed to the next digit to the left. Subtraction operation is also done using addition: The appropriate operand is simply negated before being added.



Addition: A + B ; A: Augend; B: Addend
Subtraction: A - B: A: Minuend; B: Subtrahend

Operation	Add Magnitude	Subtract 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)

Fig 2.1: Addition and Subtraction operation

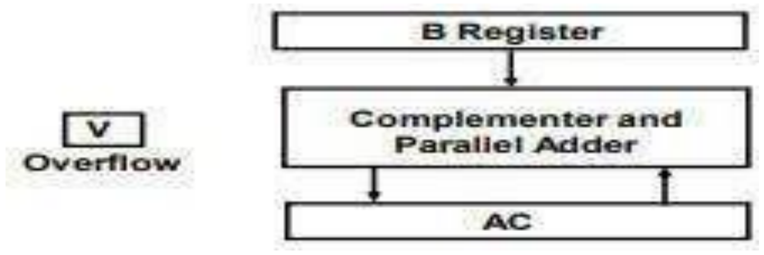
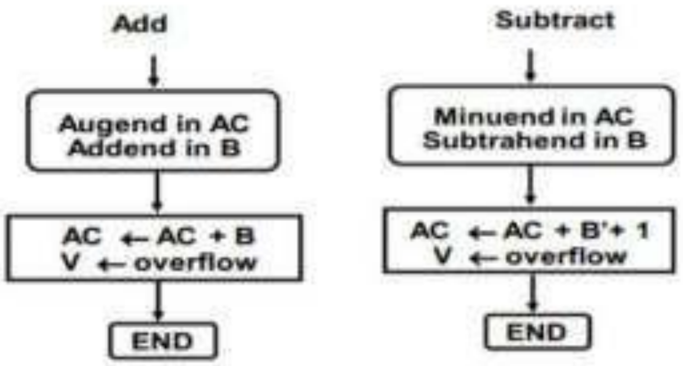


Fig 2.2: Hardware for addition / subtraction



a) Addition **b) Subtraction**

Fig 2.2: Addition and subtraction algorithm

Steps for addition:

- Place the addend in register B and augend in AC.
- Add the contents in B and AC and place the result in AC.
- V register will hold the overflow bits (if any).

Steps for subtraction:

- Place the minuend in AC and subtrahend in B.
- Add the contents of AC and 2's complemented B. Place the result in AC.
- V register will hold the overflow bits (if any).

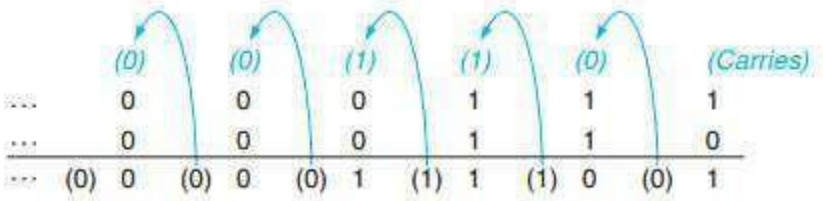


Fig 2.3: Manipulating carry

The figure 2.3 shows binary addition with carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry. If there is a carry at this bit, it will be stored in the overflow register.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. This means borrow occurred from the sign bit.

Operation	Operand A	Operand B	Result indicating overflow
A+B	>=0	>=0	<0
A+B	<0	<0	>=0
A-B	>=0	<0	<0
A-B	<0	>=0	>=0

Example 2.2: Add 6 and 7.

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten} \end{array}$$

Example 2.3: Subtract 6 from 7.

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

Example 2.4: Subtract 6 from 7 through 2's complement.

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

The MIPS instructions for addition and subtraction are given in the following table:

Instruction	Example	Operation
Add	Add $s1, $s2, $s3	S1=s2+s3Overflow detected
Subtract	Sub $s1, $s2, $s3	S1=s2-s3Overflow detected
Add Immediate	Addi $s1, $s2, 100	S1=s2+100Overflow detected
Add unsigned	Addu $s1, $s2, $s3	S1=s2+s3Overflow undetected
Subtract unsigned	Subu $s1, $s2, $s3	S1=s2-s3Overflow undetected
Add immediate unsigned	Addiu $s1, $s2, 100	S1=s2+100Overflow undetected

2.2.2 Multiplication

Multiplication is seen as repeated addition. The first operand is called the multiplicand and the second the multiplier. The final result is called the product. The number of digits in the product is larger than the number in either the multiplicand or the multiplier. The length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n + m bits long. The steps in multiplication are:

- Place a copy of the in the proper place if the multiplier digit is a 1
- Place 0 in the proper place if the digit is 0.

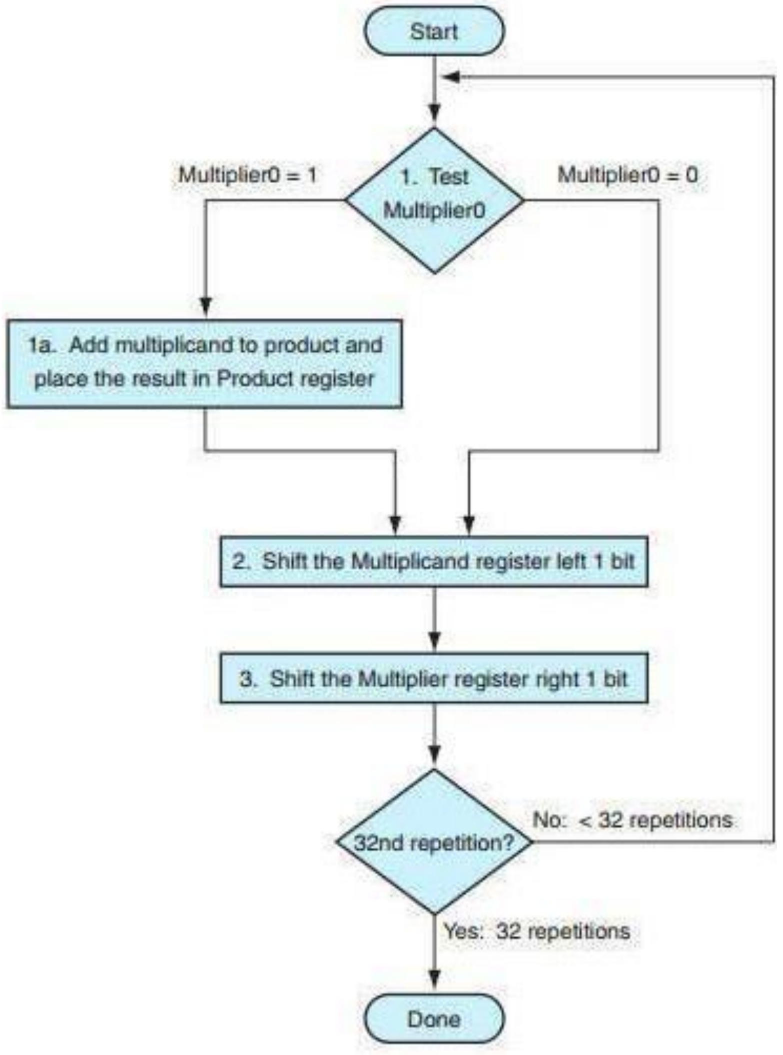

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Fig 2.4: Basic multiplication algorithm

Booth's Algorithm:

Booth algorithm gives a procedure for multiplying binary integers in signed- に|s complement representation.)t operates on the fact that strings of ど|s in the multiplier require no addition but just shifting, and a string of な|s in the multiplier from bit weight に^k to weight 2^m can be treated as 2^{k+1}- 2^m.

For example, the binary number 001110 (+14) has a string な|s from にぬ to にな ϕk=ぬ, m=1). The number can be represented as 2^{k+1}– 2^m = 24 – 21 = 16 – 2 = 14. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as M X 2^4 – M X 2^1. Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Booth algorithm requires examination of the multiplier bits and shifting of partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted

From the partial, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant な in a string of な|s in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 in a string of ど|s in the multiplier.
3. The partial product does not change when multiplier bit is identical to the previous multiplier bit.

The algorithm works for positive or negative multipliers in に|s complement representation. This is because a negative multiplier ends with a string of な|s and the last operation will be a subtraction of the appropriate weight. The two bits of the multiplier in on and Qn+1 are inspected. If the two bits are equal to 10, it means that the first 1 in a string of 1|s has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.)f the two bits are equal to どな, it means that the first ど in a string of ど|s has been encountered. This requires the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change.

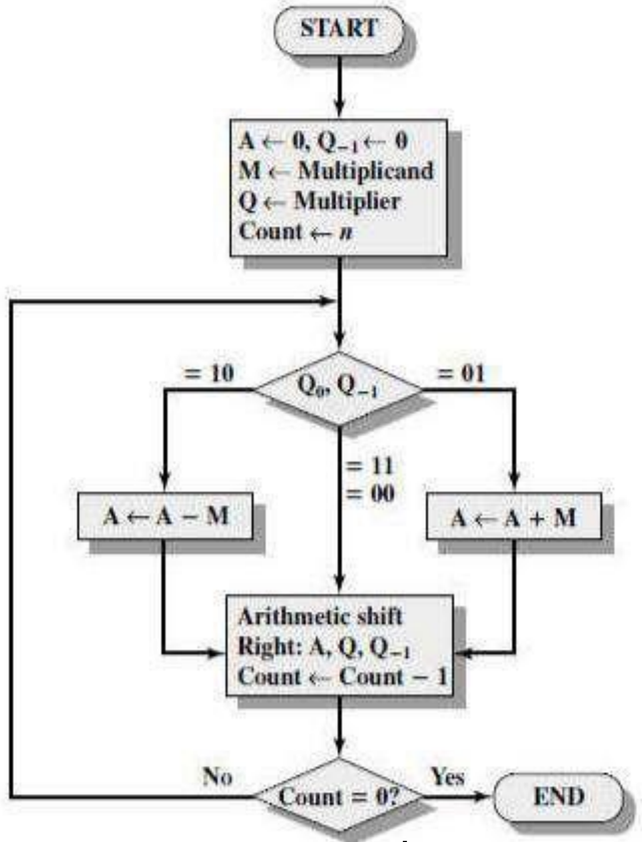
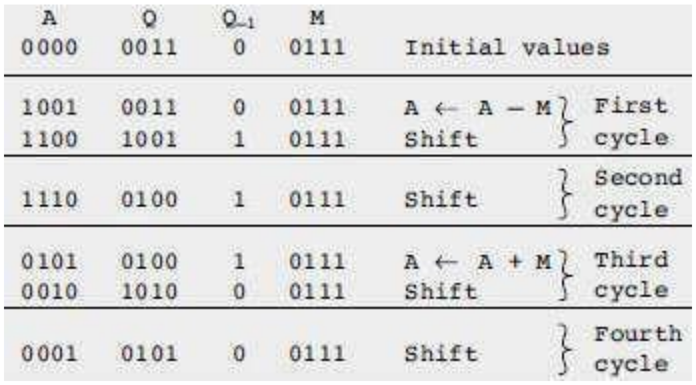


Fig 2.5: Flowchart for Booth' s algorithm

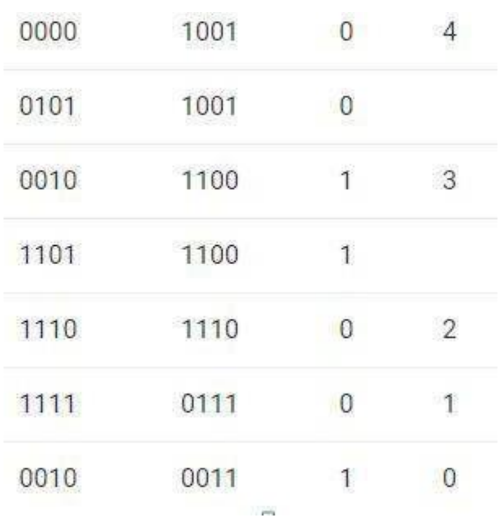
Example に.5: Multiply 7 and ぬ using Booth' s algorithm.



A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

The product is available in AQ.

Example 2.6 : Multiply -5 and -7 using Booth's algorithm



A	Q	Q-1	M
0000	1001	0	4
0101	1001	0	
0010	1100	1	3
1101	1100	1	
1110	1110	0	2
1111	0111	0	1
0010	0011	1	0

The product is available in AQ

2.2.3 Division

Division is repeated subtraction. The two operands (dividend and divisor) and the result (quotient) of divide are accompanied by a second result called the remainder. The following are the terminologies:

- Dividend: A number being divided.
- Divisor: A number that the dividend is divided by.
- Quotient: The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.
- Remainder: The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend

Dividend = Quotient * Divisor + Remainder

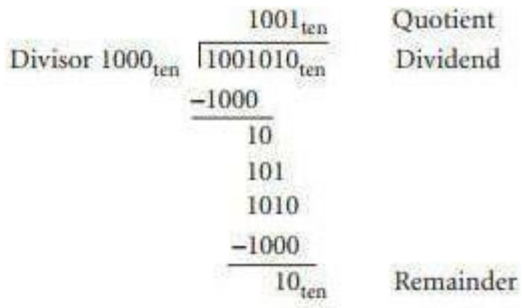


Fig 2.6: Division Terminologies

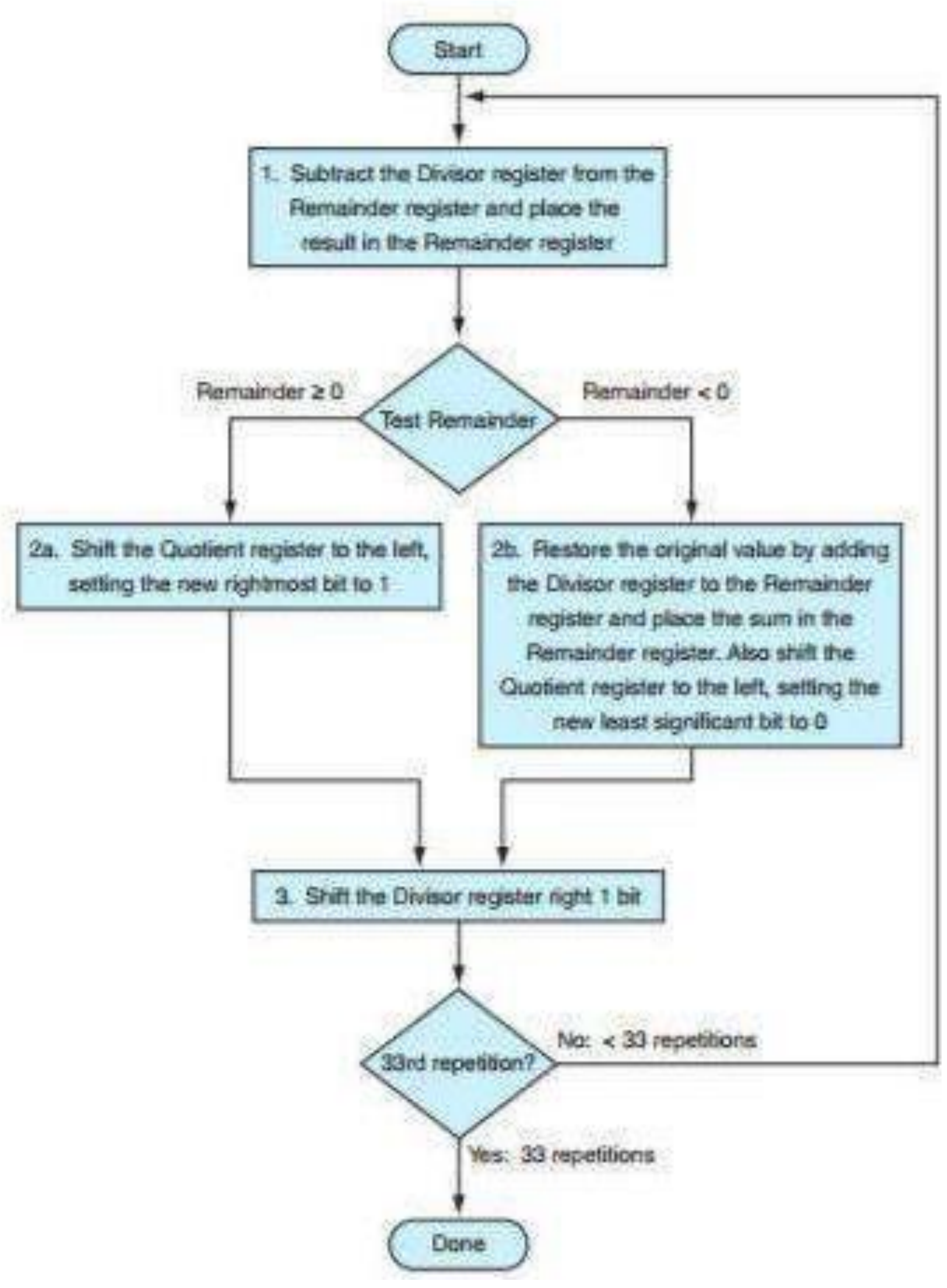

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

Fig 2.7: Basic division operation

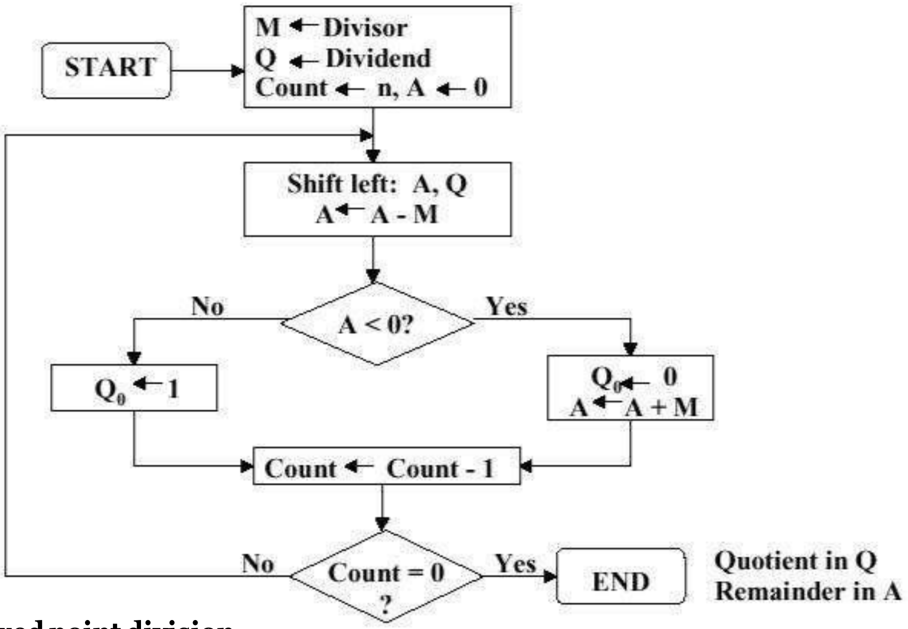
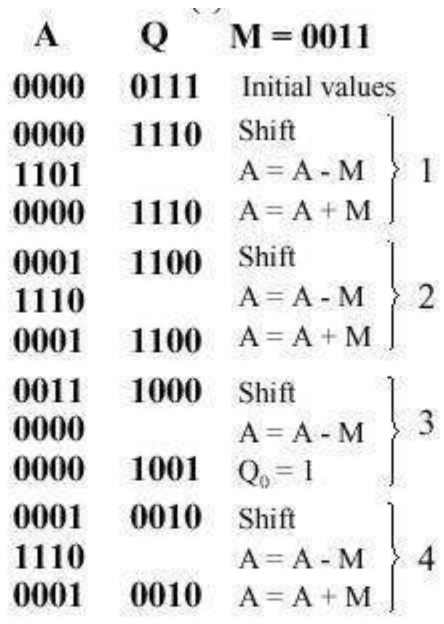


Fig 2.8: Fixed point division

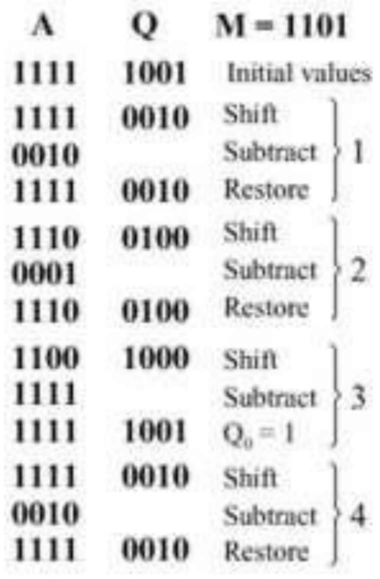
Example 2.7: Divide -7 by 3



A	Q	M = 0011	
0000	0111	Initial values	
0000	1110	Shift	1
1101		A = A - M	
0000	1110	A = A + M	
0001	1100	Shift	2
1110		A = A - M	
0001	1100	A = A + M	
0011	1000	Shift	3
0000		A = A - M	
0000	1001	$Q_0 = 1$	
0001	0010	Shift	4
1110		A = A - M	
0001	0010	A = A + M	

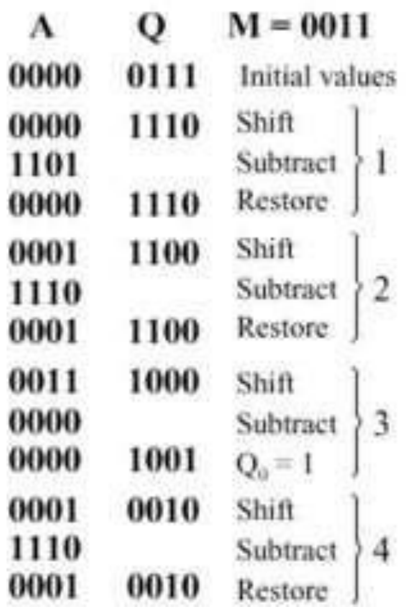
Quotient=0010Remainder=0001

Example 2.8: Divide -7 by -3



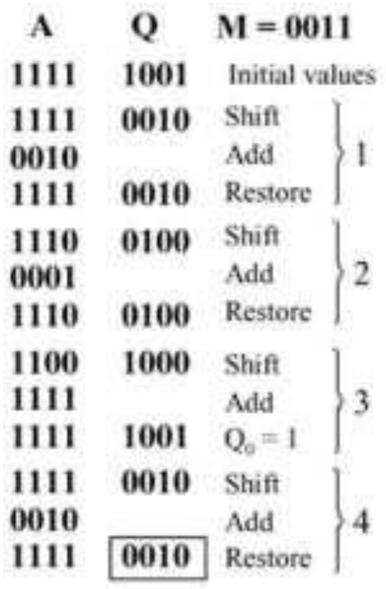
A	Q	M = 1101	
1111	1001	Initial values	
1111	0010	Shift	
0010		Subtract	1
1111	0010	Restore	
1110	0100	Shift	
0001		Subtract	2
1110	0100	Restore	
1100	1000	Shift	
1111		Subtract	3
1111	1001	$Q_0 = 1$	
1111	0010	Shift	
0010		Subtract	4
1111	0010	Restore	

Example 2.9: Divide 7 by 3



A	Q	M = 0011	
0000	0111	Initial values	
0000	1110	Shift	
1101		Subtract	1
0000	1110	Restore	
0001	1100	Shift	
1110		Subtract	2
0001	1100	Restore	
0011	1000	Shift	
0000		Subtract	3
0000	1001	$Q_0 = 1$	
0001	0010	Shift	
1110		Subtract	4
0001	0010	Restore	

Example 2.10: Divide -7 by 3



A	Q	M = 0011	
1111	1001	Initial values	
1111	0010	Shift	1
0010		Add	
1111	0010	Restore	
1110	0100	Shift	2
0001		Add	
1110	0100	Restore	
1100	1000	Shift	3
1111		Add	
1111	1001	$Q_0 = 1$	
1111	0010	Shift	4
0010		Add	
1111	$\boxed{0010}$	Restore	

MIPS instructions for multiplication and division

Category	Example	Description
Multiply	mult $s2, $s3	Hi, lo=s2 * s3 64 bit signed product in Hi, Lo
Multiply unsigned	multu $s2, $s3	Hi, lo=s2 * s3 64 bit signed product in Hi, Lo

Divide	div $s2, $s3	Lo=s2/s3 (Quotient)
		Hi=s2 mod s3 (Remainder)

Divide unsigned	divu $s2, $s3	Lo=s2/s3 (unsigned Quotient)
		Hi=s2 mod s3 (Remainder)

Move from Hi	mfhi $s1	S1=Hi Used to get a copy of Hi
Move from Lo	mflo $s1	S1=lo Used to get a copy of Lo

2.3 FLOATING POINT ARITHMETIC

To represent the fractional binary numbers (IEEE 754 floating point format), it is necessary to consider floating point. If the point is assumed to the right of the sign bit, we can represent the fractional binary numbers as given below:

$$\boxed{B = (b_0 * 2^0 + b_{-1} * 2^{-1} + b_{-2} * 2^{-2} + \ldots + b_{-(n-1)} * 2^{-(n-1)})}$$

With this fractional number system, we can represent the fractional numbers in the following range,

$$-1 < F < 1 - 2^{-(n-1)}$$

The binary point is said to be float and the numbers are called **floating point numbers**. The position of binary point in floating point numbers is variable and hence numbers must be represented in the specific manner is referred to as floating point representation. The floating point representation has three fields. They are:

- **Sign:** Sign bit is the first bit of the binary representation. ┼ な ┼implies negative number and ┼ど┼ implies positive number.

  Example: 11000001110100000000000000000001. This is negative number since it starts with 1.

- **Exponent:** It starts from bit next to the sign bit of the binary representation. The exponent field is needed to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, to express an exponent of zero, 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200"127), or ばぬ. The exponents of ┼なにば ゆall どsょ and +なにぱ ゆall なs) are reserved for special numbers.

  Double precision has an 11-bit exponent field, with a bias of 1023.**Example:** For 8 bit conversion: 8 =2^{3-1}-1=3. Bias=3.

  For 32 bit conversion: 32=2^{8-1}-1= 127. Bias=127.

- **Significant digits or Mantissa:** It is calculated from the remaining 23 bits of the binary representation.)t consists of ┼な┼and a fractional part. This represents the

Precision bits of the number. It is composed of an implicit leading bit (left of the radix point) and the fraction bits (to the right of the radix point). To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways.

Example: 50 can be represented as

1. 0.050×10^3
2. $.5000 \times 10^3$

- 5.000×10^1
- 50.00×10^0
- $5000. \times 10^{-2}$

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, 50 is represented as 5.000 × 101.

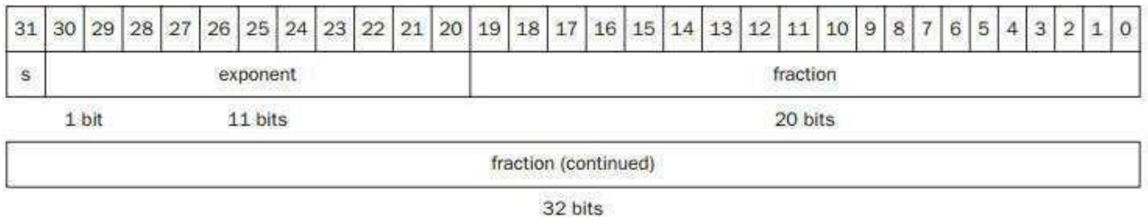


Fig 2.9: Parts of floating point number

Conversion of Decimal number to floating point:

- **Sign bit:** 1 implies negative number and 0 implies positive number.
- **Exponent:** To find the exponent value for binary representation, express the number by the nearest smaller or equal to 2^k number. The bias is determined by $2^{k-1}-1$, where k is the number of bits in exponent field. Add the bias with k value to express the exponent in binary form.
- **Mantissa:** Move the binary point so that there is only one bit from the left. Adjust the exponent of 2 so that the value does not change. This is normalizing the number. Now, consider the fractional part and represented as 23 bits by adding zeros.

Example 2.11. Find the decimal equivalent of the floating point number:

01000001110100000000000000000000

Sign=0

Exponent:

$10000011=131_{10}$

131-127=4

Exponent= 2^4=16

Mantissa:

Remaining 23 bits: 10100000000000000000000

=1*(1/2) + 0*(1/4) + 1*(1/8ょ + ど*ゅな/なはょ +……… = ど.はにの Decimal number= Sign * Exponent * Mantissa

=-1 * 16 *0.625 = -26

Example 2.11: Find the floating point equivalent of -17.

Sign=1 (-ve number)

Exponent:

Bias for 32 bit = 127 (2^{8-1} -1 = 127) 127 + 4 = $131=10000011_2$

Mantissa:

17 = 10001_2=1.0001 x 2^4

Fractional part=00010000000000000000000 -17 =1 10000011

$00010000000000000000000_2$

Terminologies:

- **Overflow:** A situation in which a positive exponent becomes too large to fit in the exponent field.
- **Underflow:** A situation in which a negative exponent becomes too large to fit in the exponent field.
- **Double precision:** A floating point value represented in two 32-bit words.

- **Single precision:** A floating point value represented in a single 32-bit word.

	Sign	Exponent	Fraction
Single Precision	1 [31]	8 [30–23]	23 [22–00]
Double Precision	1 [63]	11 [62–52]	52 [51–00]

Fig 2.10: Floating point formats

Example 2.12: The IEEE-754 32-bit floating-point representation pattern is 0 10000000 110 0000 0000 0000 0000 0000. What is the number?

Sign bit S = 0 (positive number)

Exponent E = $10000000_2 = 128_{10}$ (in normalized form)

Fraction is 1.11_2 (with an implicit leading 1) = $1 + 1\times2^{-1} + 1\times2^{-2} = 1.75_{10}$

The number is $+1.75 \times 2^{(128-127)} = +3.5_{10}$

Example 2.13: Suppose that IEEE-754 32-bit floating-point representation pattern is 1 01111110 100 0000 0000 0000 0000 0000. Find the decimal number.

Sign bit S = 1 (negative number)

E = $0111\ 1110_2 = 126_{10}$ (in normalized form)

Fraction is 1.1_2 (with an implicit leading 1) = $1 + 2^{-1} = 1.5_{10}$

The number is -1.5 × 2^ (126-127) = -0.75D

Example 2.14: Suppose that IEEE-754 32-bit floating-point representation pattern is 1 01111110 000 0000 0000 0000 0000 0001. What is the decimal number?

Sign bit S = 1 (negative number) E = $0111\ 1110_2 = 126_{10}$ (in normalized form) Fraction is 1.000 0000 0000 0000 0000 0001B (with an implicit leading 1) = $1 + 2^{-23}$

The number is $-(1 + 2^{-23}) \times 2^{(126-127)} = -0.500000059604644775390625$

Example 2.15: Express 85.125 in single and double precision.

85 = 1010101

0.125 = 001

85.125 = 1010101.001

=1.010101001 x 2^6

Sign = 0

1. Single precision:

Biased exponent 127+6=133

133 = 10000101

Normalized mantisa = 010101001

The IEEE 754 Single precision = 0 10000101 01010100100000000000000

2. Double precision:

Biased exponent 1023+6=1029

1029 = 10000000101

Normalized mantisa = 010101001

The IEEE 754 Double precision=

0 10000000101 0101010010000000000000000000000000000000000000000000

2.3.1 Floating point addition and subtraction

Floating-point numbers are coded as sign/magnitude, reversing the sign-bit inverses the sign. Consequently the same operator performs as well addition or subtraction according to the two operand's signs. The steps in floating point addition are:

- Rewrite the smaller number such that its exponent matches with the exponent of the larger number.
- Add the mantissas
- Renormalize the mantissa by shifting mantissa and adjusting the exponent.
- Check for overflow/underflow of the exponent after normalization.
- If the mantissa does not fit in the space reserved for it, it has to be rounded off.

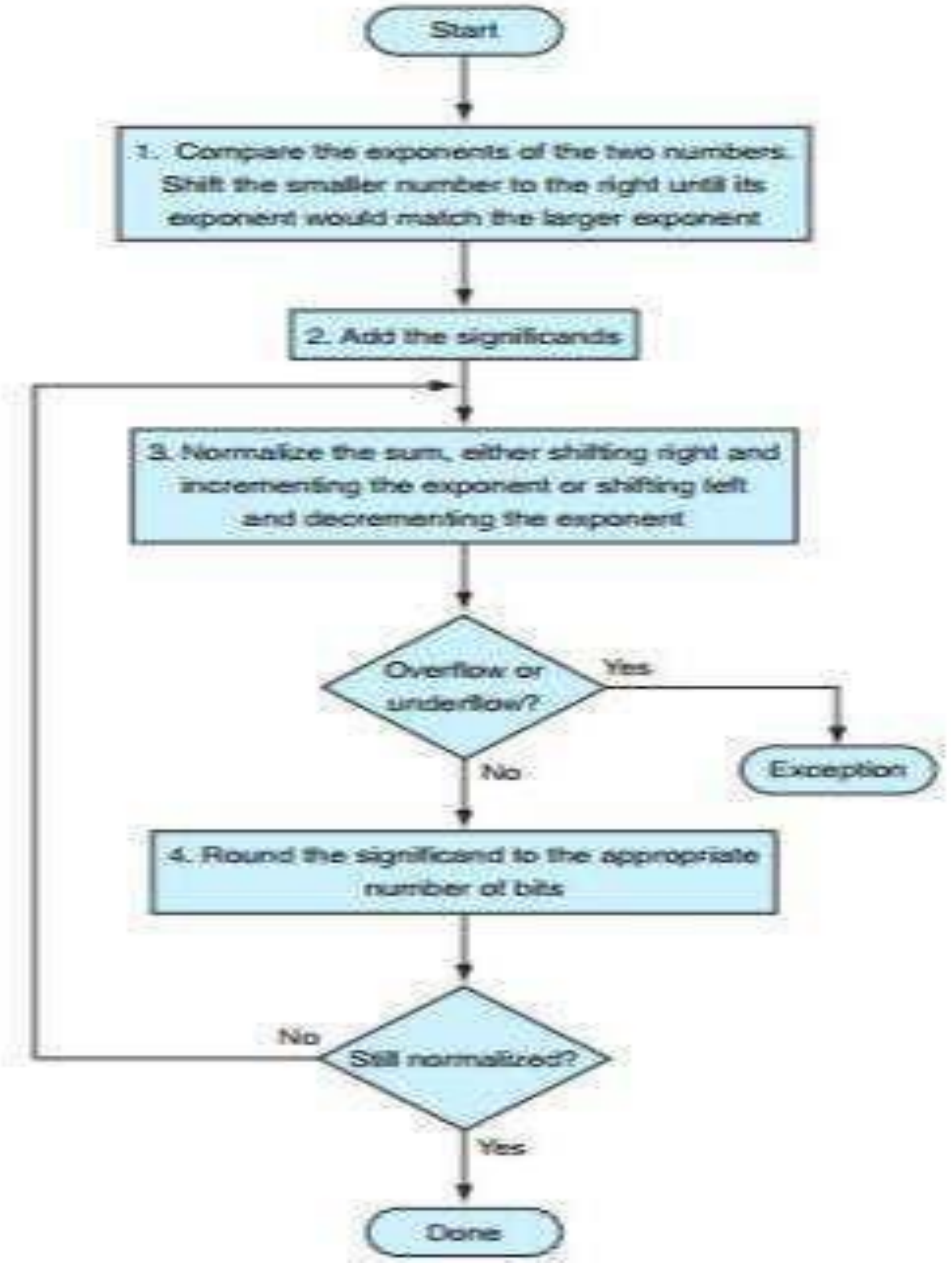

Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

Fig 2.11: Flowchart for floating point addition / subtraction

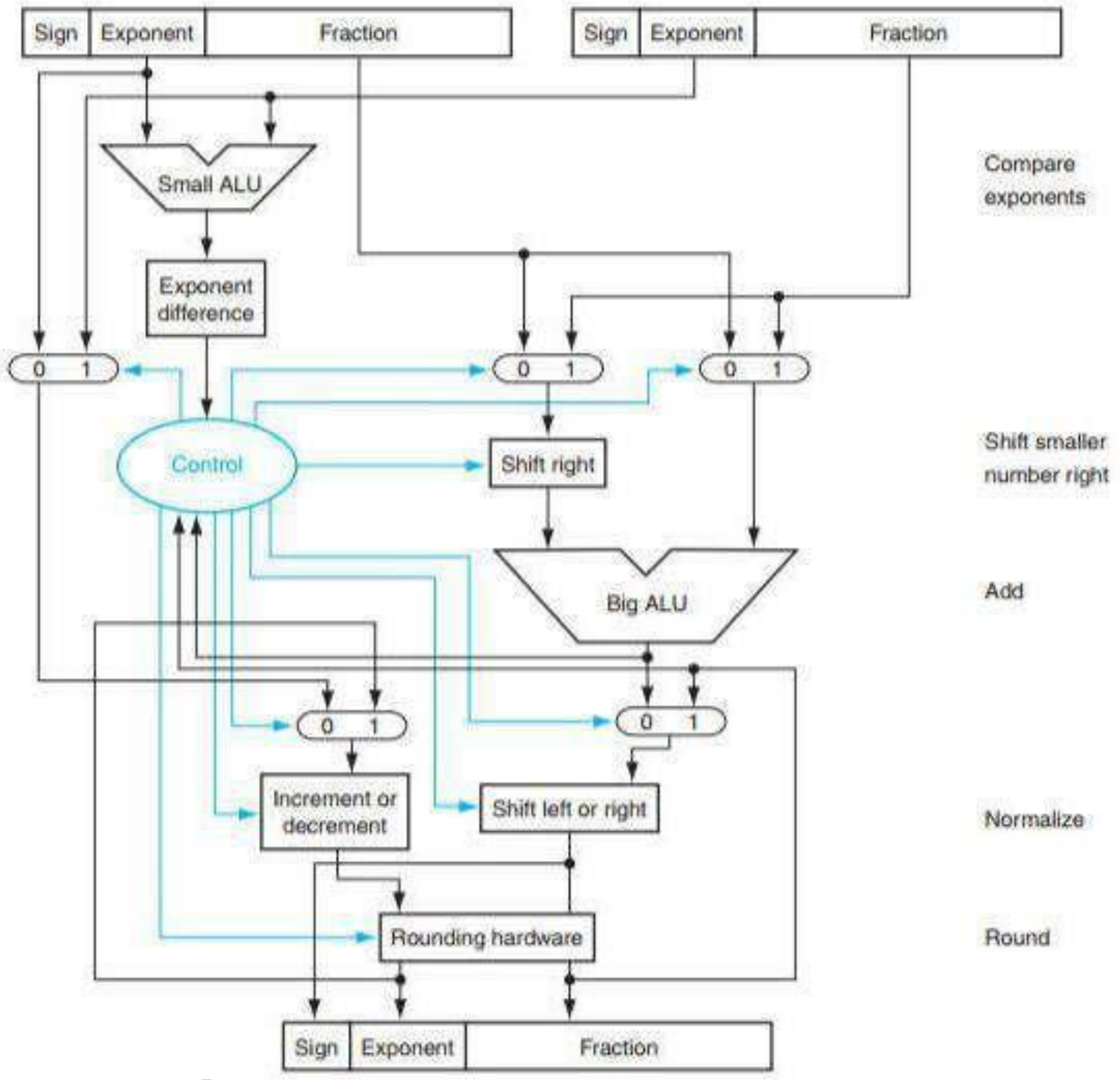


Fig 2.12: Hardware for floating point addition

The addition operation proceeds as the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significant of the smaller number, and the significant of the larger number. The smaller significant is shifted right, and then the significant are added together using the big ALU.

The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the final result.

Example 2.16: Add 0.5 + (-0.4375)

0.5 = 0.1 × 20 = 1.000 × 2-1 (normalized)

-0.4375 = -0.0111 × 20 = -1.110 × 2-2 (normalized)

Step 1: Rewrite the smaller number such that its exponent matches with the exponent of the larger number.

$-1.110 \times 2^{-2} = -0.1110 \times 2^{-1}$

Step 2: Add the mantissas

$$\begin{array}{r} 1.000 \times 2^{-1} + \\ -0.1110 \times 2^{-1} \\ \hline 0.001 \times 2^{-1} \end{array}$$

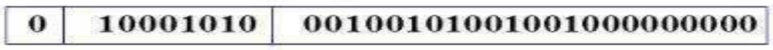
Step 3: Renormalize the mantissa by shifting mantissa and adjusting the exponent.s$0.001 \times 2^{-1} = 1.000 \times 2^{-4}$

-126 <= -4 <= 127 (-4 is within the range of -126 and 127).No overflow or underflow

Step 4: The sum fits in 4 bits so rounding is not required

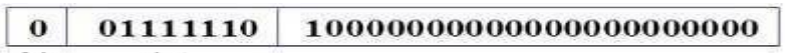
Example 2.17: Express the following numbers in IEEE 754 format and find their sum:

2345.125 and 0.75.Single precision format of 2345.125:



0	10001010	00100101001001000000000

Single precision format of 0.75:



0	01111110	10000000000000000000000

Exponent of 2345.125 > exponent of 0.75 10001010-01111110=00000110 = $(12)_{10}$
Shift 0.75 to 12 positions right: 0.00000000000110000000000 Add:
1. 00100101001001000000000 (1 is added before . since this is a positive number)
$\underline{+ \quad 0.00000000000110000000000}$ (0 is added before . since it is a negative number)

1. 00100101001111000000000

The sum is normalized. There is no underflow. The final sum is

0	10001010	0010010100111000000000

The result is +ve hence 0 is filled in the sign field. The exponent value of 2345.125 is copied in the exponent field of the result, since the 0.75 is adjusted to the exponent of 2345.125.

Example 2.18: Subtract -1.00000000000000010011010x2^{-1} from

1.00000000101100010001101x2^{-6} .

+1.00000000101100010001101x2^{-6}

-1.00000000000000010011010x2^{-1}

Change the +1.00000000101100010001101x2^{-6} into power of 2^{-6}.

0.000010000000001011000100 01101x2^{-1}

To perform subtraction take 2's complement of-1.00000000000000010011010x2^{-1} which is 1 0.11111111111111101100110 x 2^{-1}(Here first 1 is the overflow bit).

Now add both numbers

```
0   0.000010000000001011000100 01101x2^-1
1   0.11111111111111101100110       x 2^-1 1
-------------------------------------------
1.000010000000001000101010 01101x2^-1
```

2.3.2 Floating point multiplication

The following are the steps in floating point multiplication:

- Add the exponents
- Multiply the significant digits
- Normalize the product
- Round-off the product (if necessary)

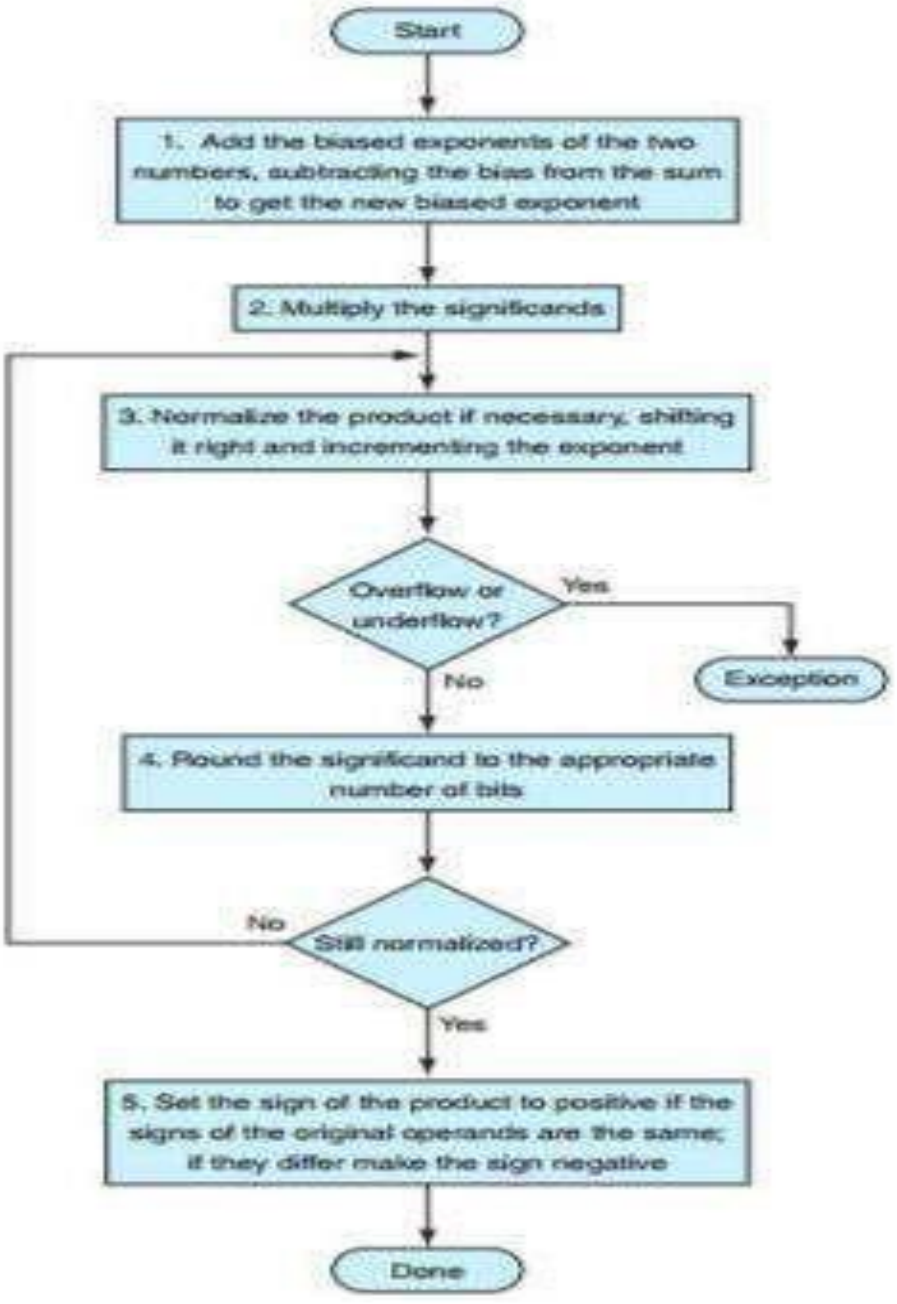

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Fig 2.13: Flowchart for Floating point multiplication

Example 2.19: Multiply 1.110 x 10^{10} by 9.200 x 10^{-5}. Express the product in 3 decimal places.

1. Add the exponents

Exponent of the product=10-5=5

 Multiply the significant digits 1.110 x 9.200=10.212000

 Normalize the product

10.212 x 10^{5}= 1.0212 x 10^{6}

4. Round-off

1.0212 x 10^{6}= 1.021 x 10^{6}

Example 2.20: Perform binary multiplication on 0.5 and -0.4375.

0.5= 1.000 x 2^{-1}

0.4375= -1.110 x 2^{-2}1.

Add the exponents

Exponent of the product=-1+-2=-3

 Multiply the significant digits 1.000 x -1.110=-1.110

 Normalize the product

-1.110 x 10^{-3} is already normalized.

Example 2.21: Multiply -1.110 1000 0100 0000 10101 0001 x 2^{-4} and 1.100 0000 0001 0000 0000 0000 x 2^{-2}.

1. Add the exponents

Exponent of the product=-4 + -2=-6 2. Multiply the significant digits

-1.110 1000 0100 0000 10101 0001 x 1.100 0000 0001 0000 0000 0000

= 10.10111000111110111111001100101000010000000000000

3. Normalize the product 1.010111000111110111111001100101000010000000000000 x 2^{-5}

4. Round-off (Only 23 fraction bits)

1.01011100011111011111100x2^{-5}

2.3.3 MIPS floating point instructions

MIPS provide several instructions for floating point numbers for performing the following operations:

- Arithmetic
- Data movement (memory and registers)
- Conditional jumps

Floating Point (FP) instructions work with a different bank of registers. Registers are named f0 to $f31. MIPS floating-point registers are used in pairs for double precision numbers and referred using even numbers. Single precision numbers end with .s and double precision numbers end with .d.

Category	**Example**	**Description**
FP add single	add.s $f2, $f4, $f6	f2=f4 + f6
FP subtract single	sub.s $f2, $f4, $f6	f2=f4 - f6
FP multiply single	mul.s $f2, $f4, $f6	f2=f4 * f6
FP divide single	div.s $f2, $f4, $f6	f2=f4 / f6
FP add double	add.d $f2, $f4, $f6	f2=f4 + f6
FP subtract double	sub.d $f2, $f4, $f6	f2=f4 - f6
FP multiply double	mul.d $f2, $f4, $f6	f2=f4 * f6
FP divide double	div.d $f2, $f4, $f6	f2=f4 -/f6
Load wordcopr,1	Lwcl $f1, 100 ($s2)	F1=memory[s2+100]32 bit data to FP register
Store word copr,1	Swcl $f1, 100 ($s2)	Memory[s2+100]=f132 bit data to memory
Branch on FP true	Bclt 25	If(cond==1) goto PC+4+100PC relative branch if cond is true
Branch on FP false	Bclt 25	If(cond==0) goto PC+4+100PC relative branch if cond is false

FP compare single (eq, ne, li, le, gt, ge)	C.lt.s \$f2, \$f4	If(f2 < f4) Cond=1; else cond=0
FP compare double (eq, ne, li, le, gt, ge)	C.lt.d \$f2, \$f4	If(f2 < f4) Cond=1; else cond=0

2.4 HIGH PERFROMANCE ARTHMETIC

The performance improvement in arithmetic operations like addition, multiplication and division will increase the overall computational speed of the machine.

2.4.1 High performance adders

The high performance adders takes an extra input namely the transit time.

The transmit time of a logical unit is used as a time base in comparing the operating speeds of different methods, and the number of individual logical units required is used in the comparison of costs.

The two multi-bit numbers being added together will be designated as A and B, with individual bits being A1, A2, B1, etc. The third input will be C. Outputs will be S (sum) R (carry), and T (transmit). The two multi bit numbers being added together will be designated asA and B, with individual bits being A1, A2, B1, etc. The third input will be C. Outputs will be S (sum) R (carry), and T (transmit).

The time required to perform an addition in conventional adder is dependent on the time required for a carry originating in the first stage to ripple through all intervening stages

to the S or R output of the final stage. Using the transit time of a logical block as a unit of time, this amounts to two levels to generate the carry in the first stage, plus two levels per stage for transit through each intervening stage, plus two levels to form the sum in the final stage, which gives a total of two times the number of stages.

$^{C}n^{=R}n-1$
$^{C}n^{=D}n-1 \;||\; ^{T}n-1 \; ^{R}n-2$
$^{C}n^{=D}n-1 \;||\; ^{T}n-1 \; ^{D}n-2 \;||\; ^{T}n-1^{T}n2 \; ^{R}n-3$

By allowing n to have successive values starting with one and omitting all terms containing a a resulting negative subscript, it may be seen that each stage of the adder will require one OR stage with n inputs and n AND circuits having one through n inputs, where N is the position number of the particular stage under consideration.

2.4.2 High performance Multiplication

Multiplication using variable length shift

- The multiplier and the partial product will always be shifted the same amount and at the same time.
- The multiplier is shifted in relation to the decoder, and the partial product with relation to the multiplicand.
- Operation is assumed starting at the low-order end of the multiplier, which means that shifting is to the right.
- If the lowest-order bit of the multiplier is a one, it is treated as though it had been approached by shifting across zeros.

Rules:

- When shifting across zeros (from low order end of multiplier), stop at the first one.

  a) If this one is followed immediately by a zero, add the multiplicand, then shift across all following zeros.

  b) If this one is followed immediately by a second one, subtract the multiplicand, then shift across all following ones.

2. When shifting across ones (from low order end of multiplier), stop at the first zero.

  a) If this zero is followed immediately by a one, subtract the multiplicand, then shift across all following ones.

  b) If this zero is followed immediately by a second zero, add the multiplicand, then shift across all following zeros.

- A shift counter or some equivalent device must be provided to keep track of the number of shifts and to recognize the completion of the multiplication.

- If the high-order bit of the multiplier is a one and is approached by shifting across ones, that shift will be to the first zero beyond the end of the multiplier, and that zero along with the bit in the next higher order position of the register will be decoded to determine whether to add or subtract.
- For this reason, if the multiplier is initially located in the part of the register in which the product is to be developed, it should be so placed that there will be at least two blank positions between the locations of the low-order bit of the partial product and the high-order bit of the multiplier.
- Otherwise the low-order bit of the product will be decoded as part of the multiplier.

Multiplication Using Uniform Shifts

- Multiplication which uses shifts of uniform size and permits predicting the number of cycles that will be required from the size of the multiplier is preferable to a method that requires varying sizes of shifts.
- The most important use of this method is in the application of carry-save adders to multiplication although it can also be used for other applications.

Uniform shifts of two

- Assume that the multiplier is divided into two-bit groups, an extra zero being added to the high-order end, if necessary, to produce an even number of bits.
- Only one addition or subtraction will be made for each group, and, using the position of the low-order bit in the group as a reference, this addition or subtraction will consist of either two times or four times the multiplicand.
- These multiples may be obtained by shifting the position of entry of the multiplicand into the adder one or two positions left from the reference position.
- The last cycle of the multiplication may require special handling.
- Following any addition or subtraction, the resulting partial product will be either correct or larger than it should be by an amount equal to one times the multiplicand.
- Thus, if the high-order pair of bits of the multiplier is 00 or 10, the multiplicand would be multiplied by zero or two and added, which gives a correct partial product.
- If the high-order pair of bits is 01 or 11, the multiplicand is multiplied by two or four,

not one or three, and added. This gives a partial product that is larger than it should be, and the next add cycle must correct for this.

- Following the addition the partial product is shifted left- two positions. This multiplies it by four, which means that it is now larger than it should be by four times the multiplicand.
- This may be corrected during the next addition by subtracting the difference between four and the desired multiplicand multiple.
- Thus, if a pair ends in zero, the resulting partial product will be correct and the following operation will be an addition.
- If a pair ends in a one, the resulting partial product will be too large, and the following operation will be a subtraction.
- It can now be seen that the operation to be performed for any pair of bits of the multiplier may be determined by examining that pair of bits plus the low-order bit of the next higher-order pair.
- If the bit of the higher-order pair is a zero, an addition will result; if it is one, a subtraction will result. If the low-order bit of a pair is considered to have a value of one and the high-order bit a value of two, then the multiple called for by a pair is the numerical value of the pair if that value is even and one greater if it is odd.
- If the operation is an addition, this multiple of the multiplicand is used. If the operation is a subtraction (the low-order bit of the next higher order pair a one), this value is combined with minus four to determine the correct multiple to use.
- The result will be zero or negative, with a negative result meaning subtract instead of add.

Multiplication Using Carry-Save Adders

- When successive additions are required before the final answer is obtained, it is possible to delay the carry propagation beyond one stage until the completion of all of the additions, and then let one carry-propagate cycle suffice for all the additions. Adders used in this manner are called **carry-save adders.**
- A carry-save adder consists of a number of stages, each similar to the full adder. It differs from the ripple-carry adder in that the carry (R) output is not connected directly

to the next-higher-order stage of the same adder, but goes to an intermediate register or other device in the same manner as the sum (S) output.

- A carry-save adder has three inputs which, as far as use is concerned, may be considered identical, and two outputs which are not identical and must be treated in different manners.
- The procedure for adding several binary numbers by using a carry-save adder would be as follows.
- Designate the inputs for the nth bit as A_n, B_n, and C, and the outputs for the same bit as S_n and R, where S_n is the sum output and R. is the carry output.
- In the first cycle enter three of the input numbers into A, B, and C.
- In the second cycle enter the S and R obtained from the previous cycle into A and B and the fourth input number into C.
- In this operation S_n goes into A_n, but R_n goes into B_{n+1}, where B_{n+1}isin the next higher-order bit position than B.
- This is continued until all of the input numbers have been entered into the adder.
- Each add cycle advances all carries one position, add cycles as already described may be continued with zeros being entered into the third input each time until the R outputs of all stages become zero.
- The alternative is to enter S and R into a carry-propagate adder and allow time for one cycle through it.
- This carry-propagate adder may be completely separate from the carry-save unit, or it may be a combined unit with a control line for selecting either carry-save or carry-propagate operation.

SUB WORD PARALLELISM

A sub word is a lower precision unit of data contained within a word. In sub word parallelism, multiple sub words are packed into a word and then process whole words.

With the appropriate sub word boundaries this technique results in parallel processing of sub words. Since the same instruction is applied to all sub words within the word, this is a form of SIMD(Single Instruction Multiple Data) processing. It is possible to apply sub word parallelism to noncontiguous sub words of different sizes within a word. In practical implementation is simple if sub words are same size and they are contiguous within a word. The data parallel programs that benefit from sub word parallelism tend to process data that are of the same size.

Example: If word size is 64bits and sub words sizes are 8,16 and 32 bits. Hence an instruction operates on eight 8bit sub words, four 16bit sub words, two 32bit sub words or one 64bit sub word in parallel.

Advantages of sub word parallelism

- Sub word parallelism is an efficient and flexible solution for media processing because algorithm exhibit a great deal of data parallelism on lower precision data.
- It is also useful for computations unrelated to multimedia that exhibit data parallelism on lower precision data.
- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors.
- One key advantage of sub word parallelism is that it allows general-purpose processors to exploit wider word sizes even when not processing high-precision data.
- The processor can achieve more sub word parallelism on lower precision data rather than wasting much of the word-oriented data paths and registers.

Support for sub word parallelism

- Data-parallel algorithms with lower precision data map well into sub word-parallel programs.
- The support required for such sub word-parallel computations then mirrors the needs of the data-parallel algorithms.
- To exploit data parallelism, we need sub word parallel compute primitives, which perform the same operation simultaneously on sub words packed into a word.
- These may include basic arithmetic operations like add, subtract, multiply, divide, logical, and other compute operations.

- Data-parallel computations also need

1. Data alignment before or after certain operations for sub words representing fixed-point numbers or fractions
2. Sub word rearrangement within a register so that algorithms can continue parallel processing at full clip
3. A way to expand data into larger containers for more precision in intermediate computations. Similarly, a way to contract it to a fewer number of bits after the computation's completion and before its output.
4. Conditional execution
5. Reduction operations that combine the packed sub words in a register into a single value or a smaller set of values.
6. A way to clip higher precision numbers to fewer bits for storage or transmission.
7. The ability to move data between processor registers and memory, as well as the ability to loop and branch to an arbitrary program location.

UNIT - III
THE PROCESSOR

3.1 INTRODUCTION

The key performance metrics of the computer systems are;

i. Instruction count: This depends on the compiler used and instruction set architecture.

ii. Clock cycle time: This depends on processor implementation.

iii. Clock cycles per instruction (CPI): This depends on processor implementation.

3.1.1 MIPS ARCHITECTURE

MIPS (Million Instructions Per Second) is a simple, streamlined, highly scalable RISC architecture with adopted by the industries.

The features that makes its widely useable are:

- Simple load and store with large number of register
- The number and the character of the instructions
- Better pipelining efficiency with visible pipeline delay slots
- Efficiency with compilers

These features make the MIPS architecture to deliver the highest performance with high levels of power efficiency. It is important to learn the architecture of MIPS to understand the detailed working of the processors.

Implementation of MIPS

IPS has 32 General purpose registers (GPR) or integer registers (64 bit) holding integer data. Floating point registers (FPR**)** are also available in MIPS capable of holding both single precision (32 bit) and double precision data (64 bit). The following are the data types available for MIPS:

Size	Name	Registers
8 bits	Byte	Integer register
bits	Half word	Integer register
bits	Word	Floating point register
bits	Double word	Floating point register

With these resources the MIPS performs the following operations:

- Memory referencing: load word (lw) and store word (sw)
- Arithmetic-logical instructions: add, sub, and, or, and slt
- Branch instructions: equal (beq) and jump (j)

i. Set the program counter (PC) to the address of the code and fetch the instruction from that memory.

ii. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, read only one register and for store word the processor has to operate on two registers.

The ALU operations are done and the result of the operation is stored in the destination register using store operation. When a branching operation is involved, then next address to be fetched must be changes based on the branch target.

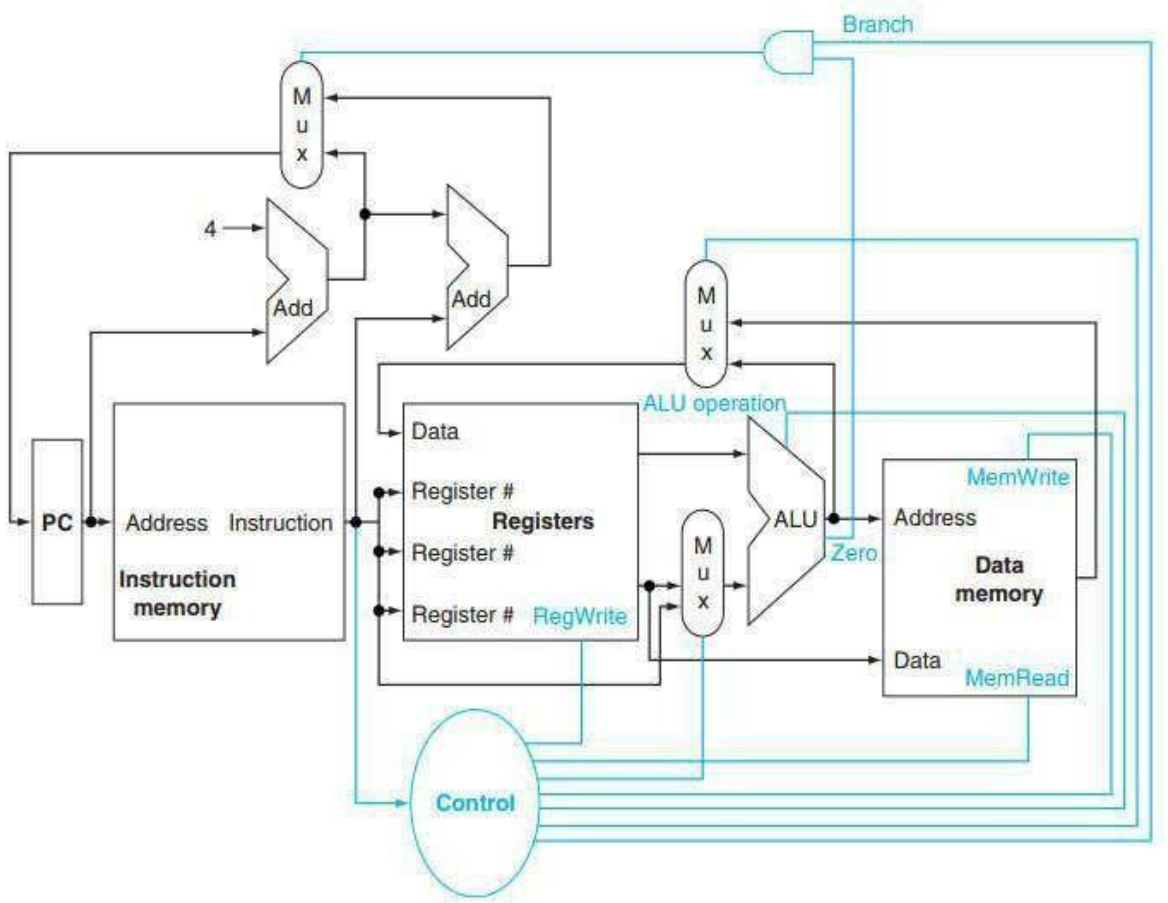


Fig 3.1: Implementation of MIPS architecture with multiplexers and control lines

Sequence of operations

- **Program Counter (PC):** This register contains the address (location) of the instruction currently getting executed. The PC is incremented to read the next instruction to be executed.
- The operands in the instruction are fetched from the registers.
- The ALU or branching operations are done. The results of the ALU operations are stored in registers. If the result is given in load and store forms, then the results are written to the memory address and from there they are transferred to the registers.
- In case of branch instructions, the result of the branch operation is used to determine the next instruction to be executed.

- The multiplexer (MUX1), selects one input control line from multiple inputs. This acts as a **data selector.**
- This helps to control several units depending on the type of instruction.
- The top multiplexor controls the target value of the PC. To execute next instruction the PC is set as PC+4. To execute a branch instruction set the PC to the branch target address.
- The multiplexor is controlled by the AND gate that operates on the zero output of the ALU and a control signal that indicates that the instruction is a branch.
- The multiplexor (MUX2) returns the output to the register file for loading the resultant data of ALU operation into the registers.
- MUX3 determines whether the second ALU input is from the registers or from the offset field of the instruction.
- The control lines determine the operation performed at the ALU. The control lines decide whether to read or write the data.

MIPS instruction format

There are only three instruction formats in MIPS. The instructions belong to any one of the following type:

- Arithmetic/logical/shift/comparison
- Control instructions (branch and jump)
- Load/store
- Other (exception, register movement to/from GP registers, etc.)

All the instructions are encoded in one of the following three formats:

I type: Load and store instructions

Op code	Rs	Rt	Immediate

R-type: Register to register operations

Op code	Rs	Rt	Rd	Shamt	Funct

J-Type: Jump instructions

Op code	Offset

The data and memory are well separated in MIPS implementation because:

- The instruction formats for the operations are not unique; hence the memory access will also be different.
- Maintaining separate memory area is less expensive.
- The operations of the processor are performed in single cycle. A single memory (for both data and memory access) will not allow for two different accesses within one cycle.

3.2 LOGIC DESIGN CONVENTIONS

The information in a computer system is encoded in binary form (0 or 1). The high voltage is encoded as 1 and low voltage as 0. The data is transmitted inside the processors through control wires / lines. These lines are capable of carrying only one bit at a time. So transfer of multiple data can be done through deploying multiple control lines or buses. The data should be synchronized with time by transferring it according to the clock pulses. All the internal operations inside the processor are implemented through logic elements. The logic elements are broadly classified into: **Combinatorial and Sequential elements.**

Differences between Combinatorial and Sequential elements

Combinatorial Elements	**Sequential Elements**
The output of the combinatorial circuit depends only on the current input.	The output depends on the previous stage outputs.
It has faster operation speed and easy implementation.	It has comparatively low operation speed I and tough implementation.
No feedback connections.	The output is connected with the input through feedback connections.
For a given set of inputs, combinatorial elements give the same output since there is no storage of past data.	The outputs vary based on previous outputs.

The basic building blocks are gates, which are time independent.	The basic building blocks are flip flops, which are time dependent.
It is used for Arithmetic and Logic operations.	It is used for data storage.
No need for trigger.	Triggering is needed to control the clock cycles.
No memory element.	Memory element is needed which is used to store the states.
Eg: Encoder, full adder, Decoder, Multiplier	**Eg:** Counters

Importance of state elements

The state elements characterize the machines. The contain state or status values so that the machine can be restored with the previous values by retaining the values in the state element. A state element has at least two inputs and one output. The required inputs are the data value to be written into the element and the clock, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. The following are the state elements in Fig 3.1: instructions, memories and registers.

3.2.1 Clocking Methodology

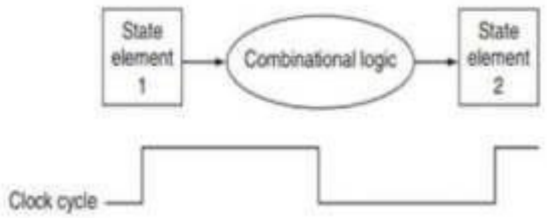


Fig 3.2 a: Combinatorial Logic

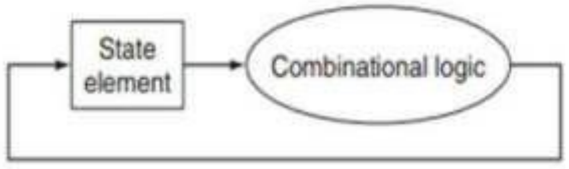


Fig 3.2 b: Edge triggered Logic

A clocking methodology is a set of rules for interconnecting components and clock signals that, when followed, guarantee proper operation of the resulting system.

The primary objective of clocking methodology is timing correlation.

Edge triggered clocking methodology

This allows the processor to read the register contents, send the value through some combinatorial logic and write that register in same clock cycle under the assumption that the state elements are controlled by implicit clock cycles.

- Here, the stored values are updated only on a clock edge.
- In combinatorial logic, the input must be read, processed and the output must be sent to the location, all in one single clock cycle (Fig 3.2 a).
- The driving force of this combinatorial circuit will be an explicit control signal.
- All the changes occur only when the clock signal is triggered.
- In edge-triggered methodology, the contents of a register are read and the value is sent through combinational logic, and written to that register in the same clock cycle.
- This prevents the access of inconsistent intermediate data
- Feedback cannot occur within 1 clock cycle because of the edge-triggered update of the state element.
- The clock cycle still must be long enough so that the input values are stable when the active clock edge occurs.

3.3 BUILDING A DATAPATH

A datapath is a representation of the flow of information (data and instructions) through the CPU, implemented using combinatorial and sequential circuitry.

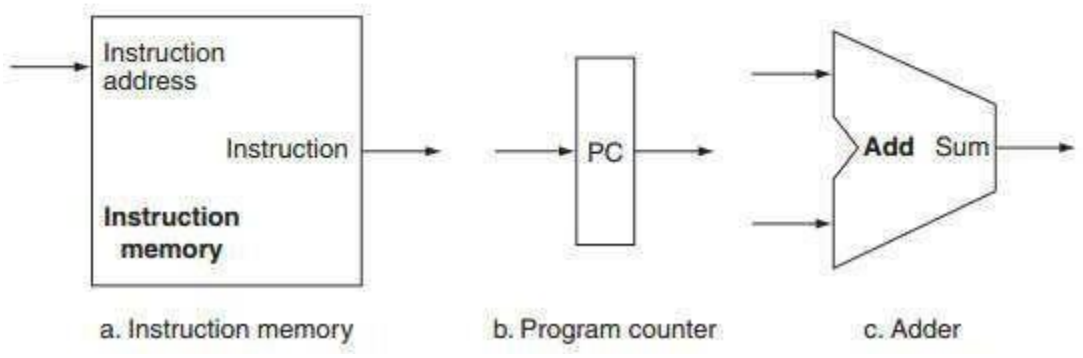


Fig: 3.3 Components of Data path

Data path is a functional unit that operates or hold data. In the MIPS implementation the data path elements includes instruction and data memories, the register file, the arithmetic logic unit (ALU), and adders. The functionalities of basic elements are listed below:

- **Instruction Memory:** It is a state element that provides read access because the data path do not perform write operation. This combinatorial memory always holds contents of location specified by the address.
- **Program Counter (PC):** This is a 32 bit state register containing the address of the current instruction that is being executed. It is updated after every clock cycle and do not require an explicit write signal.
- **Adder:** This is a combinatorial circuit that updates the value of PC after every clock cycle to get that address of the next instruction to be executed.

3.3.1 Instruction Fetch:

The fundamental operation in Instruction Fetch is to send the address in the PC to the instruction memory and obtain the specified instruction, and the increment the PC.

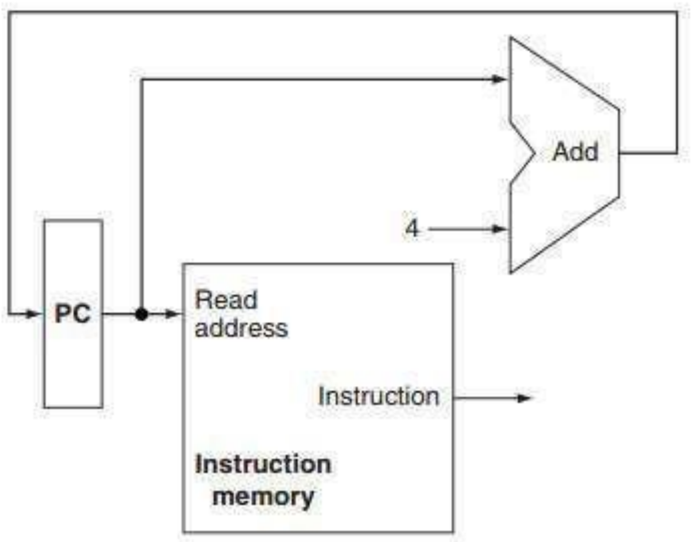


Fig: 3.4: Instruction Fetch

R type instructions:

- They all read two registers, perform an ALU operation on the contents of the registers and write the result.
- This instruction class includes add, sub, and, or, and slt.
- The processor's [illegible] general-purpose registers are stored in a structure called a **register file.**
- A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the machine.
- The R-format always performs ALU operation that has three register operands (2-read and 1-write).
- The register number must be specified in order to read the data from the register file. Also the output from a register file will contain the data that is read from the register.
- The write operation to a register has two inputs: the register number and the value to be written. This operation is edge triggered.

Load and Store instructions:

- The load and store instructions compute a memory address by adding the base register.
- If the instruction is a load, the value read from memory must be written into the register file in the specified register.
- The memory is computed by adding the address of base register and the16-bit signed offset field (which is a part of the instruction).
- If the instruction is a store, the value to be stored must also be read from the register.

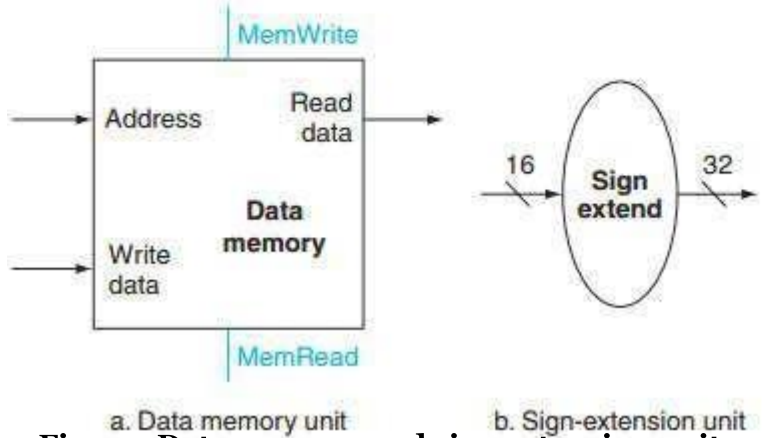


Fig 3.5: Data memory and sign extension unit

- The processor has a sign extension unit to **sign-extend** the 16-bit offset field in the instruction to a 32-bit signed value.
- The data memory unit is necessary to perform write operation of store instruction. So it has both read and write control signals, an address input and data input.

Branch Instructions:

Branch Target is the address specified in a branch, which is used to update the PC if the branch is taken. In the MIPS architecture the branch target is computed as the sum of the offset field of the instruction and the address of the instruction following the branch.

J. The beq instruction (branch instruction) has three operands, two registers that are compared for equality, and a 16-bit offset to compute the branch target address. **beq t1, t2, offset**

K. Thus, the branch data path must do two operations: compute the branch target address and compare the register contents.

L. **Branch Taken** is where the branch condition is satisfied and the program counter (PC) loads the branch target. All unconditional branches are taken branches.

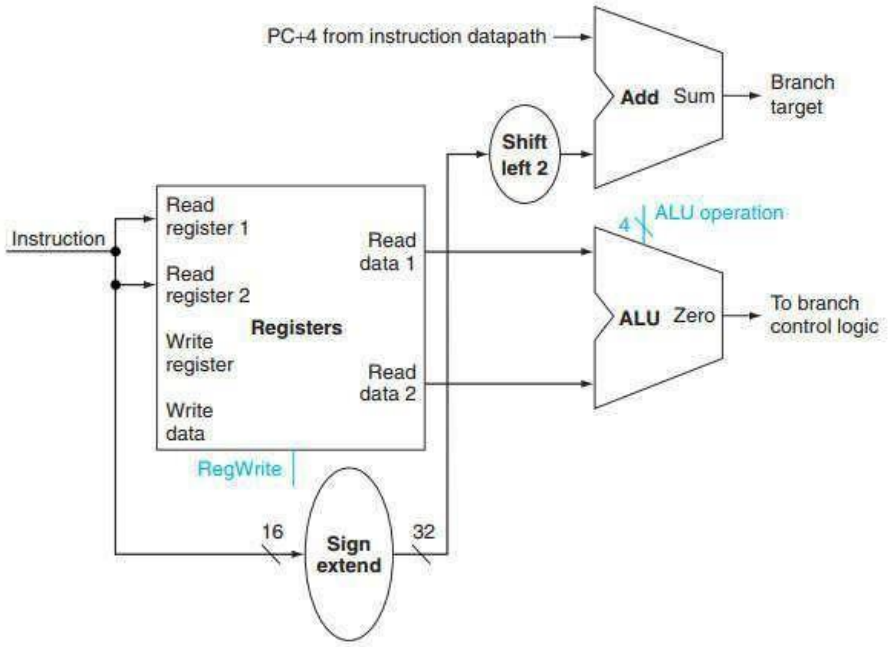
M. **Branch not Taken is** where the branch condition is false and the program counter (PC) loads the address of the instruction that sequentially follows the branch.

N. The branch target is calculated by taking the address of the net instruction after the branch instruction, since the PC value will be updated as PC+4 even before the branch decision is taken

O. The offset field is shifted left 2 bits to increase the effective range of the offset field by a factor of four.

Fig 3. 6: Data path of branch Instructions

JJ. The unit labelled Shift left 2 adds two zeros to the low-order end of the sign-extended offset



field. This operation truncated the sign values.

KK. The control logic decides whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

LL. The jump instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits. This shift is done by concatenating 00 to the jump offset.

MM. **Delayed branch** is where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

NN. MIPS architecture implements delayed branch (i.e.) the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

JJJ. When the condition is false, the execution looks like a normal branch.

KKK. When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address.

KKK. Delayed branches facilitate pipelining.

3.3.2 Creating a single Datapath

- A simple implementation of a single data path is to execute all operations within one clock cycle.
- The data path resources can be utilized only for one clock cycle. To facilitate this, some resources must be duplicated for simultaneous access while other resources will be shared.
- One example is having separate memory for instructions and memory.
- When a resource is used in shared mode, then multiple connections must be made. The selection of which control will access the resource will be decided by a multiplexer.

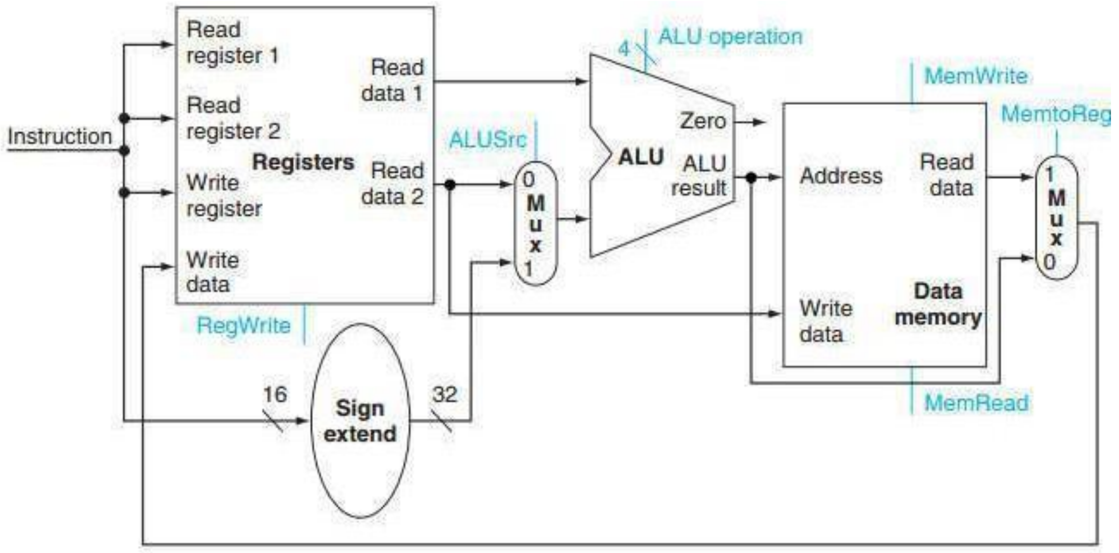


Fig: 3.7: Simple data path

- The data path illustrated in Fig 3.7 shows the assembling of individual elements into a simple data path.

- To implement branch instructions the data path must include an adder circuitry to compute branch target (Refer Fig: 3.6).
- The control unit for this data path must take inputs and generate a write signal for each state element. Apart from the inputs a selector control must be included for each multiplexor and the ALU control.
- The operations of arithmetic-logical (or R-type) instructions and the memory instructions data path are almost similar.
- The arithmetic-logical instructions use the ALU with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, but the second input is the sign-extended 16-bit offset field from the instruction.

3.4 SIMPLE IMPLEMENTATION SCHEME

The basic implementation includes a subset of the core MIPS instruction set:

- The memory-reference instructions load word (lw) and store word (sw).
- The arithmetic-logical instructions add, sub, AND, OR, and slt.
- The instructions branch equal (beq) and jump (j).

For any instruction, the following two steps are same:

- Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
- Read one or two registers, using fields of the instruction to select the registers to read.

Load instruction needs to read only one register, but most other instructions require reading two registers. The remaining actions required to complete the instruction depend on the instruction class. For the three instruction classes namely memory-reference, arithmetic-logical, and branches, the actions are mostly the same. This is due to the simplicity and regularity of the MIPS instruction set.

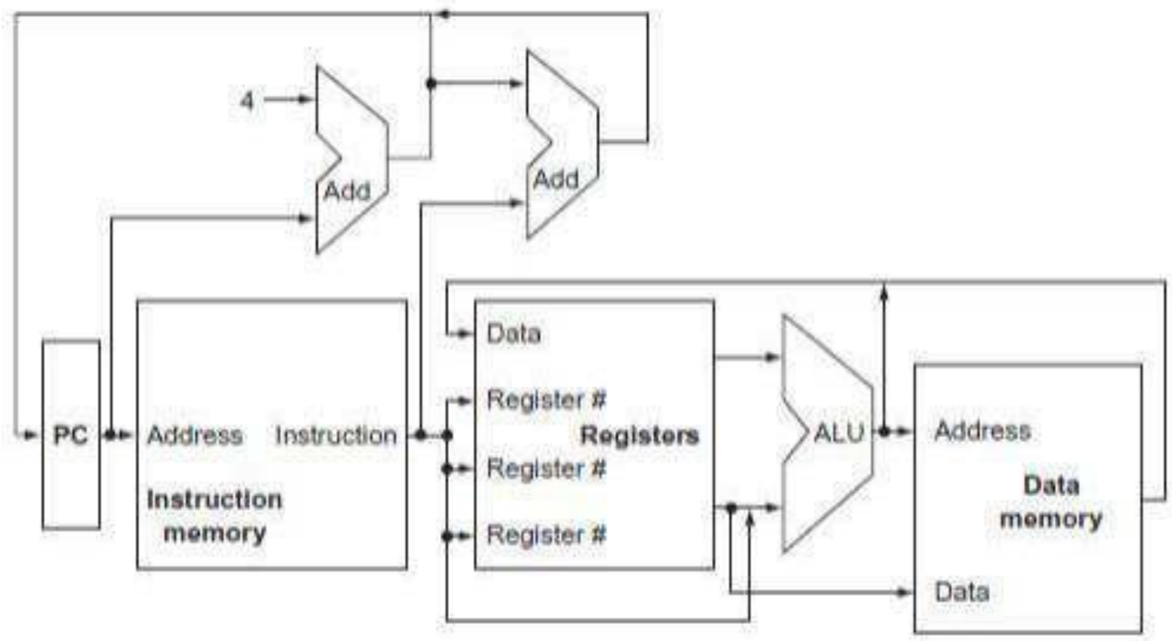


Fig 3.8: An abstract view of MIPS implementation Instruction Formats of MIPS

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

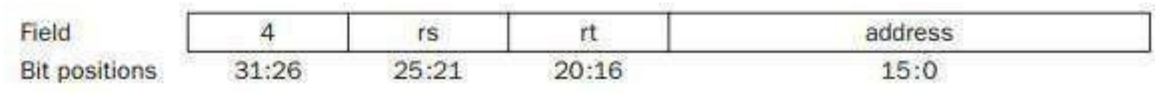
Fig 3.9: R-Format Instruction

Instruction format for R-format instructions have an op code of 0. These instructions have three register operands: sources: rs, rt, and destination: rd. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. This instruction type is used to implement are add, sub, and, or, and slt. The shamt field is for shifting operation.

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

Fig 3.10: Load or store instruction

Instruction format for load specified by op code = 35ten and store is specified by op code 43ten) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory.



Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

Fig 3.11: Branch Instructions

Instruction format for branch equal (op code = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign extended, shifted, and added to the PC to compute the branch target address.

- All instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.
- The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison.
- After using the ALU, the actions required to complete various instruction classes differ.
- A memory-reference instruction will need to access the memory either to read data for a load or write data for a store.
- An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register.
- Branch instruction need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction.
- All instructions start by using the program counter to supply the instruction address to the instruction memory.
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction.

- Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch).
- If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
- If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers.
- The result from the ALU or memory is written back into the register file.
- Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch off set are summed) or from an adder that increments the current PC by 4.
- The thick lines interconnecting the functional units represent buses, which consist of multiple signals.
- Fig.3.12shows the data path of Fig 3.8 with the three required multiplexors added, and control lines for the major functional units.
- A control unit, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors.
- The third multiplexor, which determines whether PC + 4 or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a beq instruction.

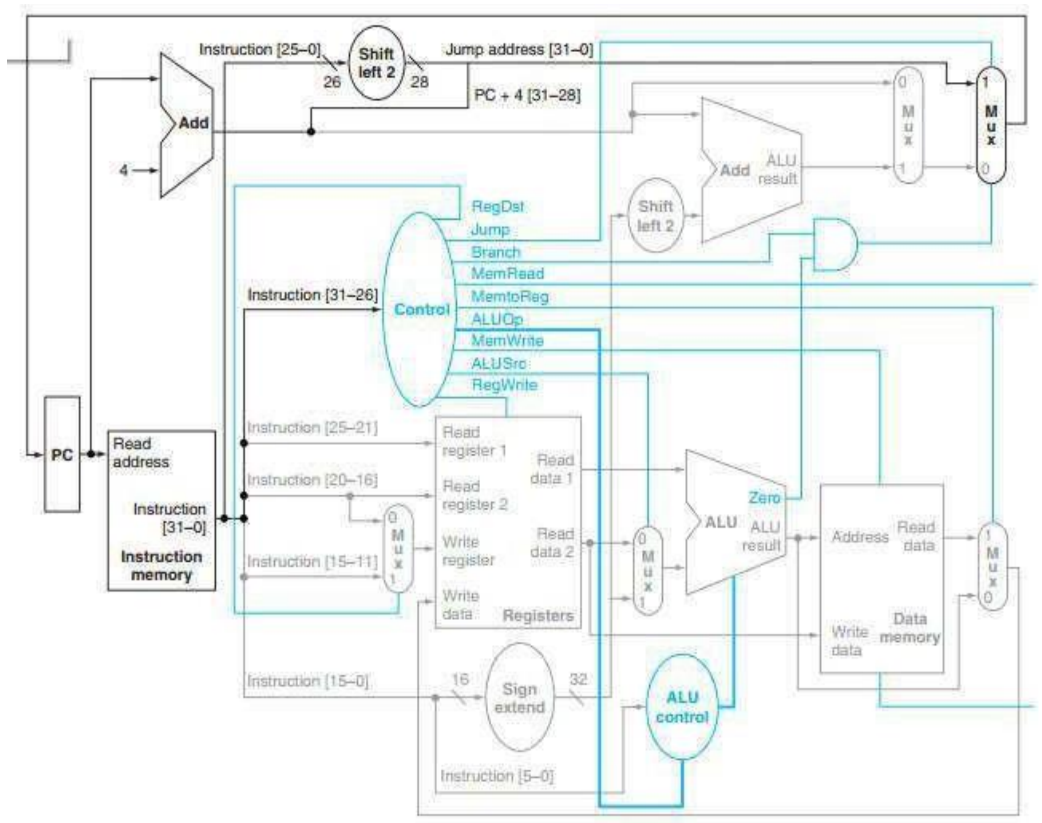


Fig 3.12: Implementation scheme with control lines

Operation of the Data path given in Fig 3.12:

Four steps to execute the instruction; these steps are ordered by the flow of information:

- The instruction is fetched, and the PC is incremented.
- Two registers, $t2 and $t3, are read from the register file. the main control unit computes the setting of the control lines during this step.
- The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.
- The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

Effect of the control signals

Signal	When deasserted	When asserted
RegDst	The register destination number for Write register comens from the rt field (bits 20:16)	The register destination number for Write register comes from the rd field (bits 15:11)
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second operand comes from the second register file output (Read data 2).	The second operand is the sign extended lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder after computing PC+4	The PC is replaced by the output of the adder after computing the branch target.
MemRead	None	Data memory contents designated by the address input are placed on the Read data input.
MemtoReg	The value given to Write data input is got from the ALU.	The value given to Write data input is got from the data memory.

The setting of the control lines is completely determined by the op code fields of the instruction as given below:

Instruc-tion	**Reg Dst**	**ALU Src**	**Memto Reg**	**Reg Write**	**Mem Read**	**Mem Write**	**Branch**	**ALU Op1**	**ALU Opo**
R-format	1	0	0	1	0	0	0	1	0
Lw	0	1	1	1	1	0	0	0	0
Sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

Finalizing the Controls

The logic values for a comprehensive control unit can be expressed as a single large truth table. This table combines all the outputs and uses the op code bits as inputs. It completely specifies the control function.

Input / Output	Signal	R-format Name	Lw	Sw	Beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	X
	RegWrite	1	1	0	0
		υ		υ	
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

3.5 PIPELINING

Pipelining is an implementation technique in which multiple instructions are executed simultaneously by overlapping them in execution to save time and resource. The previous instruction will be in the execution phase when the current instruction is fetched from the memory.

Need for Pipelining

Without a pipeline, a computer processor fetches the first instruction from memory, performs the operation mentioned in it, and then goes to fetch the next instruction from memory. While fetching the instruction, the arithmetic unit of the processor is idle. It must wait until it is loaded with next instruction.

With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor. The result is an increase in the number of instructions that can be performed during a given time period.

3.5.1 Stages in MIPS pipelining:

The following are the various stages in pipelining:

- **Instruction Fetch (IF):** Fetch instruction from memory.
- **Instruction Decode (RD):** Read registers while decoding the instruction. The format of MIPS instructions allows reading and decoding to occur simultaneously.
- **Execute:** Execute the operation or calculate an address. This involves ALU operations.
- **Memory access (MEM):** Access an operand in data memory.
- **Write Back (WB):** Write the result into a register.

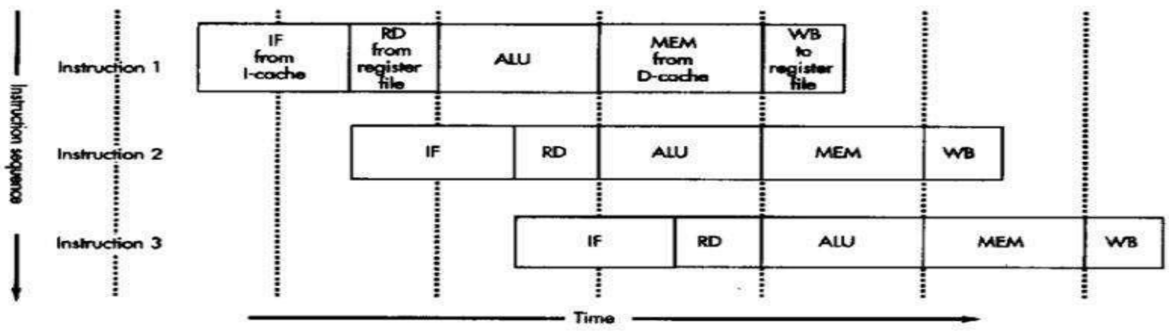


Fig 3.13: 5 stage pipelining of MIPS architecture

The pipelining speed can be manipulated using the expression:

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Pipelining improves performance by increasing instruction throughput. It is not decreasing the execution time of an individual instruction, but increases the number of instructions that complete its execution for a given time period. Thus the overall performance of the processor is improved both in terms of resource utilization and throughput.

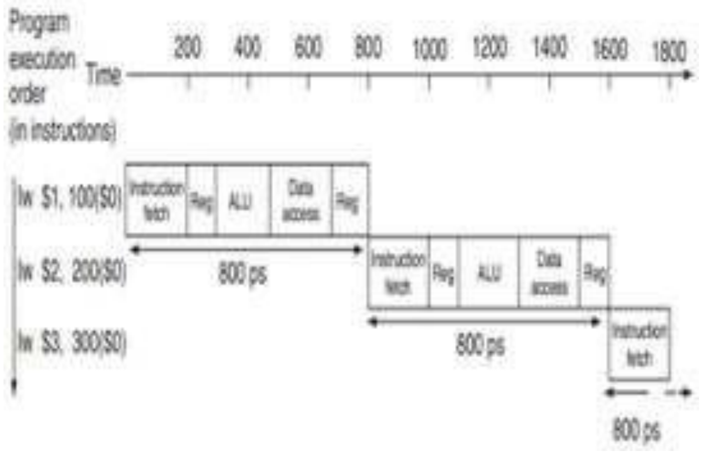


Fig 3.14 a) Non pipelined Execution

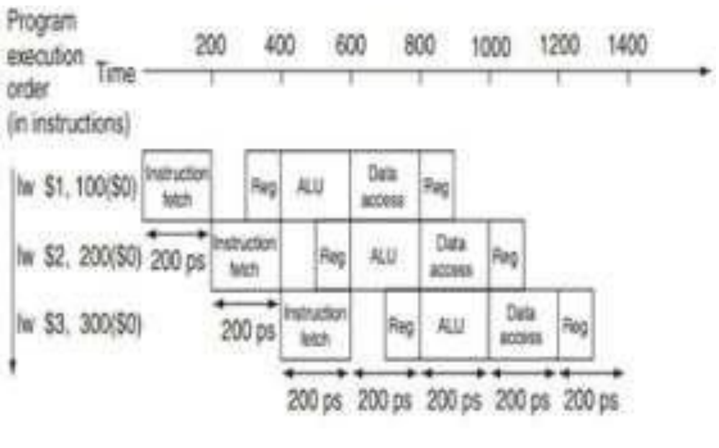


Fig 3.14 b) Pipelined Execution

Fig 3.14 shows the comparison of execution of instructions with and without pipelining on same hardware components. The timeline clearly indicates that there is a difference in execution time and resource utilization. The challenges in implementing pipelining may arise due to slowest resource.

3.5.2 Designing instruction sets for Pipelining

- The simplicity and generality of MIPS instructions are that they are of same length. This facilitates easy instruction fetching in the first stage of pipelining.
- MIPS has only a few instruction formats. In every instruction format, the source operand register is located at the same position in the instruction format.
- This symmetry eases the instruction decode stage by reading the register file simultaneously while the hardware is determining the type of instruction format.
- Also, the memory operands appear in only in load or store instruction type in MIPS. So that the execute stage can calculate the memory address and then access memory in the following stage.
- Operands must be aligned in memory. Hence, a single data transfer instruction requiring two data memory accesses can be done in a single pipeline stage.

3.5.3 Hazards in Pipelining

Hazards are situations that prevent the next instruction in the instruction cycle from being executing during its designated clock cycle. Hazards reduce the performance of the pipelining.

They are attempt to use same resource by two or more instruction at the same time.

Example: In case of single memory is used for instructions and data access and when two instructions are accessing the same register one at instruction fetch stage and other at memory access stage. This leads to inconsistent data access.

Types of hazard:

Structural Hazards: They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

- **Data Hazards:** They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- **Control Hazards**: They arise from the pipelining of branches and other instructions that change the PC. This is also known as **branch hazard.** The flow of instruction addresses is not what the pipeline had expected. This results in control hazard.

3.5.4 Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.

Data hazards occur in register files due to inconsistencies in file. This is an occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. In other words, data hazards occur when the pipeline must be stalled because one step must wait for another to complete. This is due to the data dependence.

Example: Consider the following instructions:

add $s0, $t0, $t1

sub $t2, $s0, $t3

Here the sub instruction uses the result of add instruction ($s0). The add instruction cannot not write its result until the fifth stage. This results in wasting three clock cycles in the pipeline. Since the stall occurs due to the non availability of data, this is termed as data hazards.

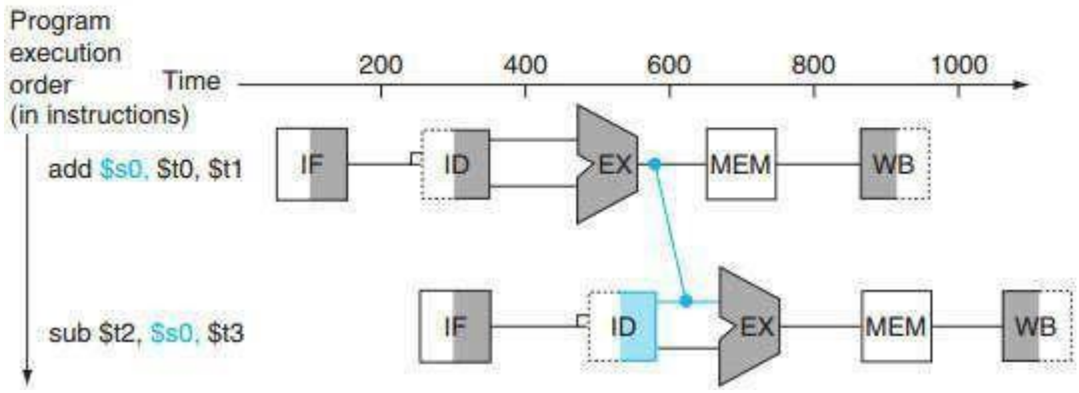


Fig 3.15: Data Hazard

Solution to resolve data hazard:

Forwarding or bypassing is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory. This can be done by adding extra memory element or hardware that acts as an internal buffer.

Forwarding cannot be a universal solution to solve data hazards. Consider the following instructions:

lw $s0, 20($t1)

sub $t2, $s0, $t3

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, there will be a hazard called as **load-use data hazard**.

A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested. This is Load-use data hazard.

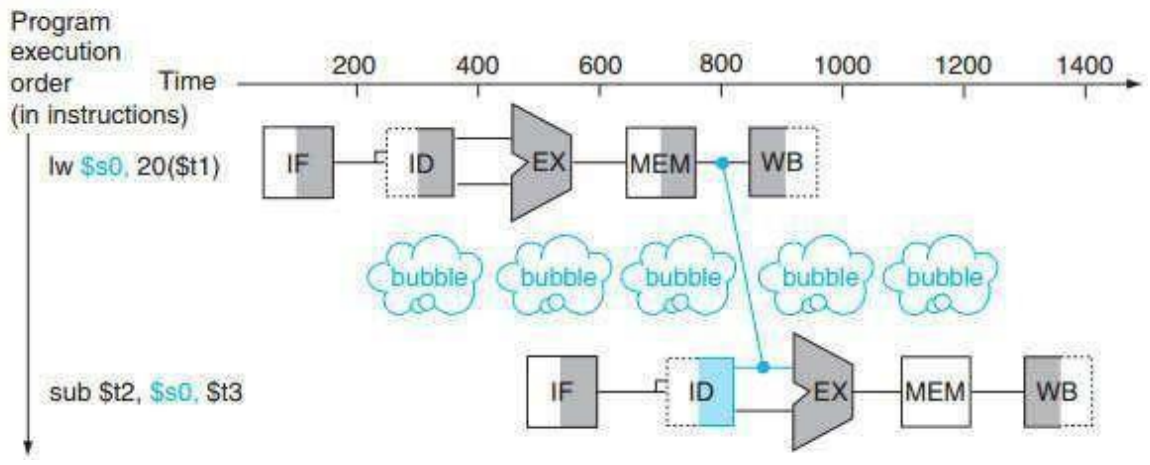


Fig 3.16: Load-Use data hazard

The stall mentioned in Fig 3.16 is called **bubble or pipeline stall.** A pipeline stall is a delay in execution of an instruction in order to resolve a hazard. During the decoding stage, the control unit will determine if the decoded instruction reads from a register that the instruction currently in the execution stage writes to.

Problem 3.1

Find the hazards in the following code segment and reorder the instructions to avoid any pipeline stalls.

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

Solution:

Both the add instructions have a hazard because of their dependence on the immediately preceding lw instruction. Bypassing eliminates several other potential hazards including the dependence of the first add on the first lw and any hazards for store instructions. Moving up the third lw instruction eliminates both hazards. This is possible since the lw instruction is independent of other operations:

```
lw $t1, 0($t0)
lw $t2, 4($t1)
lw $t4, 8($01)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

3.6 A PIPELINED DATAPATH

The stages of pipelined data path are:

- IF: Instruction fetch
- ID: Instruction decode and register file read

- EX: Execution or address calculation
- MEM: Data memory access
- WB: Write back

The two exceptions to the normal flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the data path.
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

  Data flowing from right to left does not affect the current instruction; only later instructions in the pipeline are influenced by these reverse data movements. Note that the first right-to-left arrow can lead to data hazards and the second leads to control hazards. One way to show what happens in pipelined execution is to pretend that each instruction has its own data path, and then to place these data paths on a time line to show their relationship.

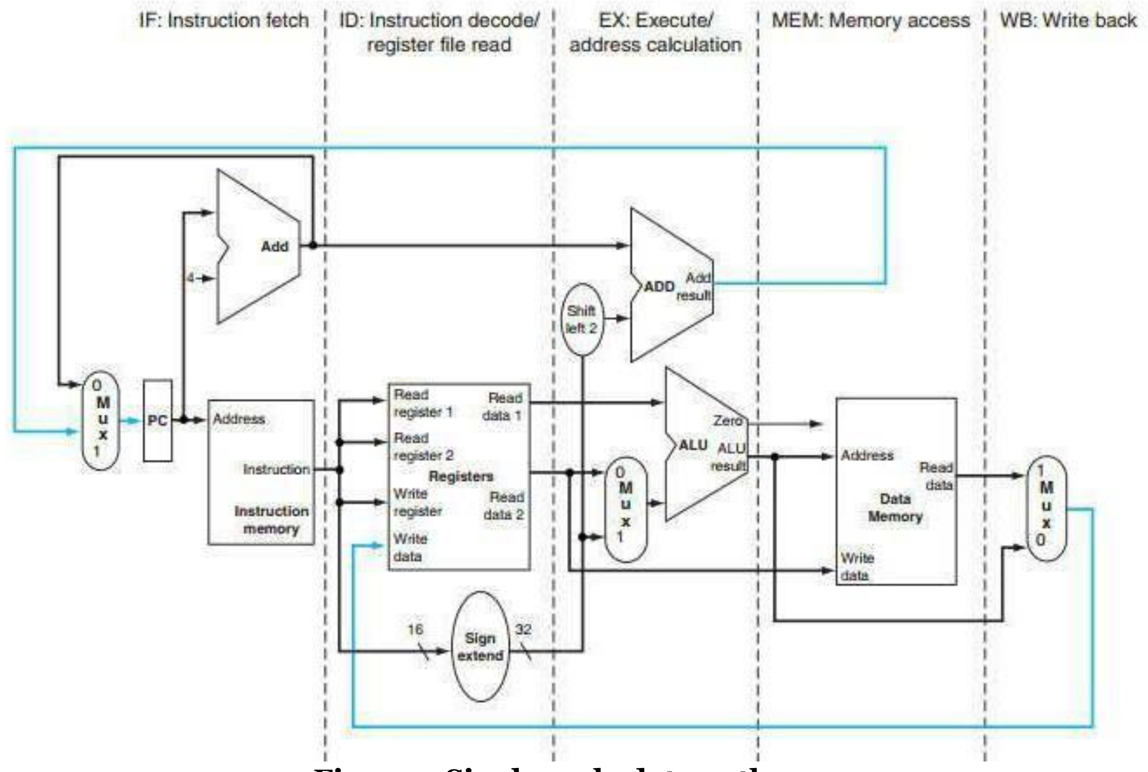


Fig 3.17: Single cycle data path

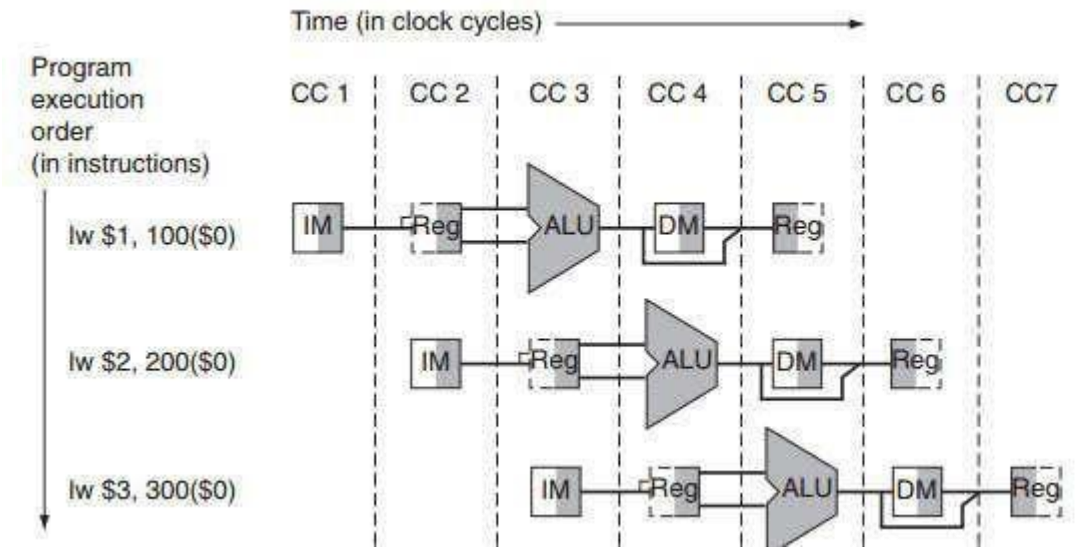


Fig 3.18: Instructions in single cycle data path

- The above fig shows that each instruction has its own data path, and each stage is labeled by the physical resource used in that stage, corresponding to the portions of the data path.
- IM represents the instruction memory and the PC in the instruction fetch stage, Reg stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on.
- To maintain proper time order, the data path breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB).
- This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read.
- As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

Operations in each stage of Pipeline:

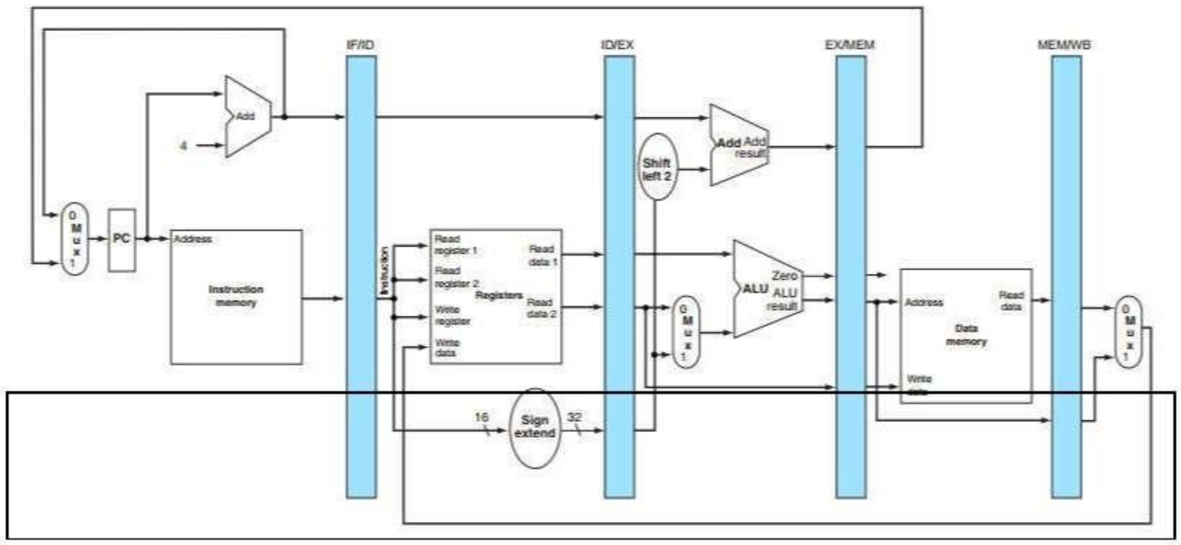


Fig 3.19: Five stages of Pipeline

 Instruction fetch:

The instruction is read from memory using the address in the PC and then placed in the IF/ID pipeline register.

The IF/ID pipeline register is similar to the Instruction register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle.

This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq.

The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

 Instruction decode and register file read:

The instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers.

All three values are stored in the ID/EX pipeline register, along with the incremented PC address.

Transfer everything that might be needed by any instruction during a later clock cycle.

These first two stages are executed by all instructions, since it is too early to know the type of the instruction.

- **Execute or address calculation:**

  The load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU.

  That sum is placed in the EX/MEM pipeline register.

- **Memory access:**

  The load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

  The register containing the data to be stored was read in an earlier stage and stored in ID/EX.

  The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

- **Write back:**

  This involves reading the data from the MEM/WB pipeline register and writing it into the register file.

3.7 PIPELINED CONTROL

This section describes the necessary control lines for implementing a pipelined data path. The control logic is needed for PC source, register destination number, and ALU control. A 6-bit funct field (function code) is needed for the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. These 6 bits are the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

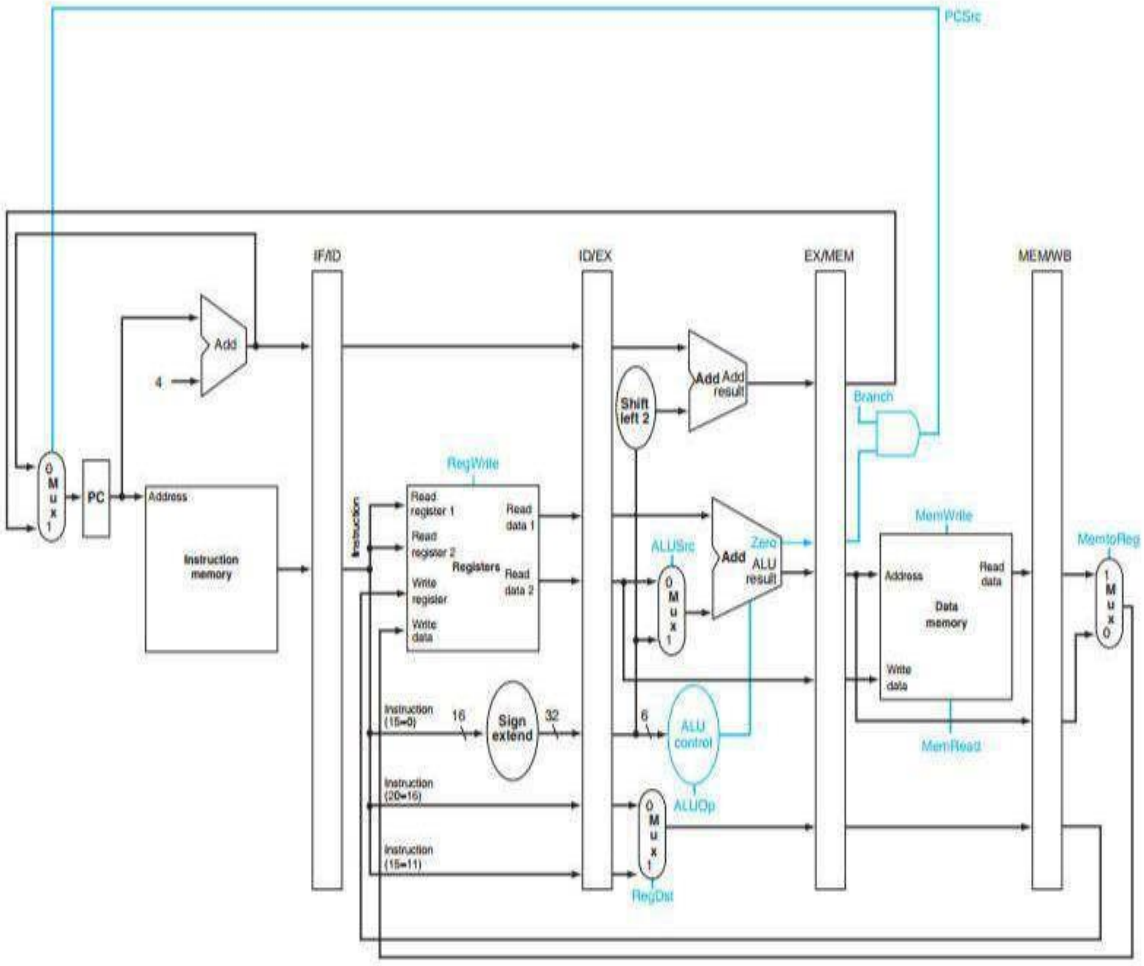


Fig 3.20: Control signals in single cycled data path

Sequence of operations:

- he PC is written on each clock cycle, so there is no separate write signal for the PC.
- There are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/ MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.
- To specify control for the pipeline, set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage.
- The control lines are also divided into five groups according to the pipeline stage:

- **Instruction fetch:** The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
- **Instruction decode/register file read:** As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
- **Execution/address calculation**: The signals to be set are Reg Dst, ALU Op, and ALU Src. The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.
- **Memory access:** The control lines set in this stage are Branch, Mem Read, and Mem Write. These signals are set by the branch equal, load, and store instructions, respectively.
- **Write back:** The two control lines are Mem to Reg, which decides between sending the ALU result or the memory value to the register file, and Reg Write, which writes the chosen value.

Implementing control means setting the nine control lines to these values in each stage for each instruction (explained in simple implementation scheme). The simplest way to do this is to extend the pipeline registers to include control information.

3.8 DATA HAZARDS

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.

Data hazards occur in register files due to inconsistencies in file. This is an occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. In other words, data hazards occur when the pipeline must be stalled because one step must wait for another to complete. This is due to the data dependence.

3.8.1 Forwarding or Bypassing

Forwarding or bypassing is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory. This can be done by adding extra memory element or hardware that acts as an internal buffer.

Forwarding cannot be a universal solution to solve data hazards. Consider the following instructions:

lw $so, 20($t1)

sub $t2, $so, $t3

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, there will be a hazard called as **load-use data hazard**.

A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested. This is Load-use data hazard.

Consider the following code:

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)

There are several dependences in this code fragment:

- The first instruction, SUB, stores a value into $2.
- That register is used as a source in the rest of the instructions this is no problem for 1-cycle and multi cycle data path.
- Each instruction executes completely before the next begins.
- This ensures that instructions 2 through 5 above use the new value of $2.

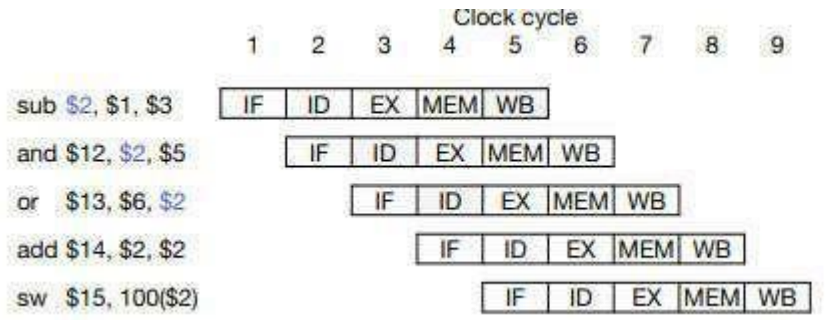


Fig 3.21: Pipelined diagram

- The SUB does not write to register $2 until clock cycle 5 causing 2 data hazards in our pipelined data path.
- The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this is the old value of $2
- The OR instruction uses register $2 in cycle 4, again before it's actually updated by SUB.

  To avoid data hazard, rewrite the instructions (sll means stall):

sub $2, $1, $3

sll $0, $0, $0

sll $0, $0, $0

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)

Since it takes two instruction cycles to get the value stored, one solution is for the assembler to insert no-ops or for compilers to reorder instructions to do useful work while the pipeline proceeds. Since the pipeline registers already contain the ALU result, we could just forward the value to later instructions, to prevent data hazards

- In clock cycle 4, the AND instruction can get the value of $1 - $3 from the EX/MEM pipeline register used by SUB.
- Then in cycle 5, the OR can get that same result from the MEM/WB pipeline register being used by SUB.

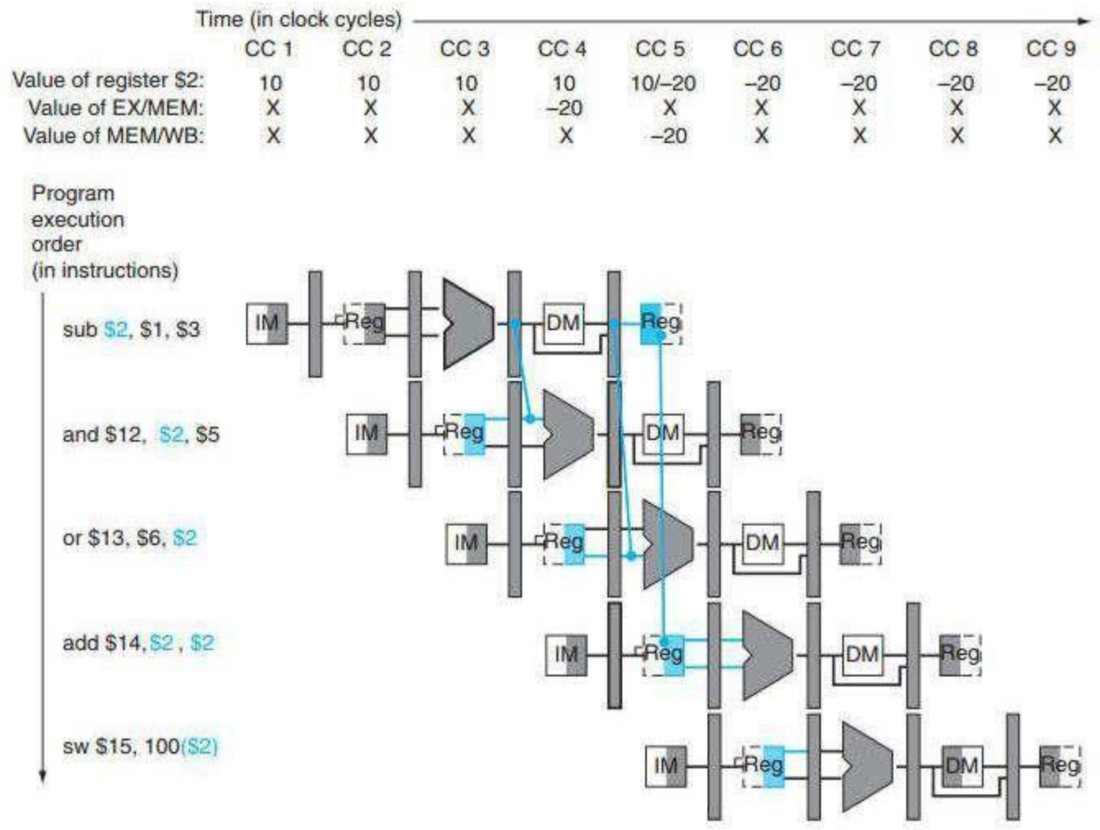

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Value of register $2: 10 10 10 10 10/–20 –20 –20 –20 –20
Value of EX/MEM: X X X –20 X X X X X
Value of MEM/WB: X X X X –20 X X X X
Program execution order (in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
IM
Reg
DM

Fig 3.22: Pipelined dependencies

Forward the data as soon as it is available to any units that need it before it is available to read from the register file. This is forwarding in data hazards.

When an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU. Te general format for specifying dependencies is given by:

Pipeline register. Field in the register

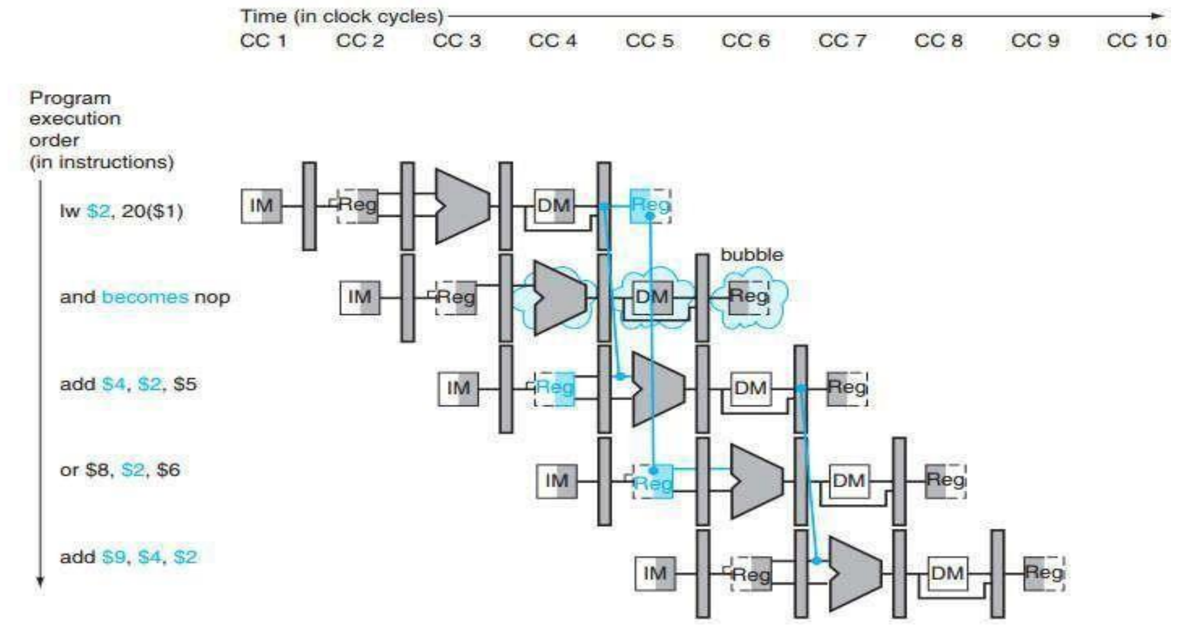
Example: ID/EX .Register Rs- refers that the value is found in the pipeline register ID/EX in the field Register Rs. The dependencies in the given example are:

- EX/MEM .Register Rd = ID/EX .Register Rs
- EX/MEM. Register Rd = ID/EX .Register Rt
- MEM/WB. Register Rd = ID/EX .Register Rs
- MEM/WB .Register Rd = ID/EX .Register Rt

The first hazard in the sequence is on register \$2, between the result of sub \$2,\$1,\$3 and the first read operand of and \$12,\$2,\$5. This hazard can be detected when the AND instruction is in the EX stage and the prior instruction is in the MEM stage.

EX/MEM .Register Rd = ID/EX .Register Rs = \$2

Forwarding the inputs to the ALU from any pipeline registers done by adding multiplexors to the input of the ALU and with the proper controls. By this the pipeline can be executed at full speed in the presence of these data dependences.



3.8.2 Stalling

Fig 3.23: Introducing stalls in pipelining

A bubble is inserted beginning in clock cycle 4, by changing the AND instruction to a nop (no operation). Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5. The or instruction is fetched in clock cycle 3, but its IF stage is delayed until clock cycle 5. After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

In short forwarding requires:

- Recognizing when a potential data hazard exists, and
- Revising the pipeline to introduce forwarding paths.

3.9 CONTROL HAZARDS

This occurs when there is a need for an instruction to take a decision based on the results of another instruction's result that has no yet completed its execution.

Control or branching hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

Instructions that disrupt the sequential flow of control present problems for pipelines are potential candidates for control hazards. The effects of these instructions cannot be exactly determined until late in the pipeline, so instruction fetch cannot continue unless it is explicitly managed. The following types of instructions can introduce control hazards:

- Unconditional branches
- Conditional branches
- Indirect branches
- Procedure calls
- Procedure returns

Example:

```
        ld      r2, 0(r4)       // r2 := memory at r4
        ld      r3, 4(r4)       // r3 := memory at r4+4
        sub     r1, r2, r3      // r1 := r2 - r3
        beqz    r1, L1          // if r1 is not 0, goto L1
        ldi     r1, 1           // r1 := 1
L1:     not     r1, r1          // r1 := not r1
        st      r1, 0(r5)       // store r1 to memory at r5
```

This code compares two memory locations and stores the result of that comparison (1 for equal, 0 for not equal) to another location. If the beqz branch is taken, then a 1 is stored; otherwise, a 0 is stored. The beqz instruction sources two hazards:

- When the beqz instruction is in the decode stage, the sub instruction is in the execute stage. The branch cannot read the output of the sub until it has been written to the register file; if it reads it early, it will read the wrong value.
- The instruction that is to be fetched after beqz is not known in advance. At this point, the status of the branch instruction is totally unknown whether it depends on the previous instruction or not. This is because it hasn't been de coded yet, so bypassing also can't help in resolving the hazard. Even if the decision is known, the location from where to fetch the instruction if the branch is taken is unknown because the effective address computation for branches do not happen until the EX stage.

Solutions for control hazards:

The following are solutions that can reduce control hazards:

- **Pipeline stall cycles:** Freeze the pipeline until the branch outcome and target are known, then proceed with fetch. Thus, every branch instruction incurs a penalty equal to the number of stall cycles. This solution is unsatisfactory if the instruction mix contains many branch instructions, and/or the pipeline is very deep.
- **Branch delay slots:** The instruction set architecture is constructed such that one or more instructions sequentially following a conditional branch instruction are executed whether or not the branch is taken. The compiler or assembly language writer must fill these branch delay slots with useful instructions or NOPs (no-operation op codes).
- **Branch prediction:** The outcome and target of conditional branches are predicted using some heuristic. Instructions are speculatively fetched and executed down the predicted path, but results are not written back to the register file until the branch is executed and the prediction is verified. When a branch is predicted, the processor enters a speculative mode in which results are written to another register file that mirrors the architected register file. Another pipeline stage called the commit stage is introduced to handle writing verified speculatively obtained results back into the real register file. Branch predictors can't be などど% accurate, so there is still a penalty for branches that is based on the branch mis prediction rate.

- **Indirect branch prediction:** Branches such as virtual method calls, computed goto and jumps through tables of pointers can be predicted using various techniques.
- **Return address stack (RAS):** Procedure returns are a form of indirect jump that can be perfectly predicted with a stack as long as the call depth doesn't exceed the stack depth. Return addresses are pushed onto the stack at a call and popped off at a return.

3.9.1 Static Branch Prediction

Branch prediction is a method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting for the actual outcome.

- In general, the bottoms of loops are branches that jump back to the top of the loop. These types of loops can easily be predicted as branch taken.
- The decision about a branch whether taken or not taken is arrived from the heuristics.
- **Dynamic hardware predictors,** guess the behavior of each branch and may change predictions for a branch over the life of a program.
- Dynamic prediction is performed by maintaining a history for each branch as taken or untaken, and then using the recent past behavior to predict the future.
- When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address.

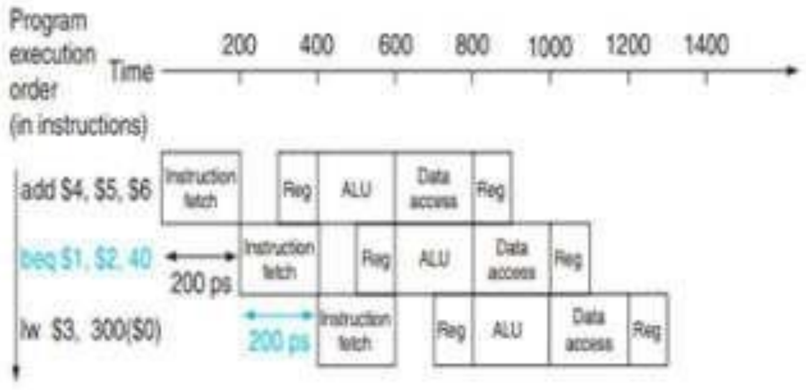


Fig 3.24 a) Branch not taken

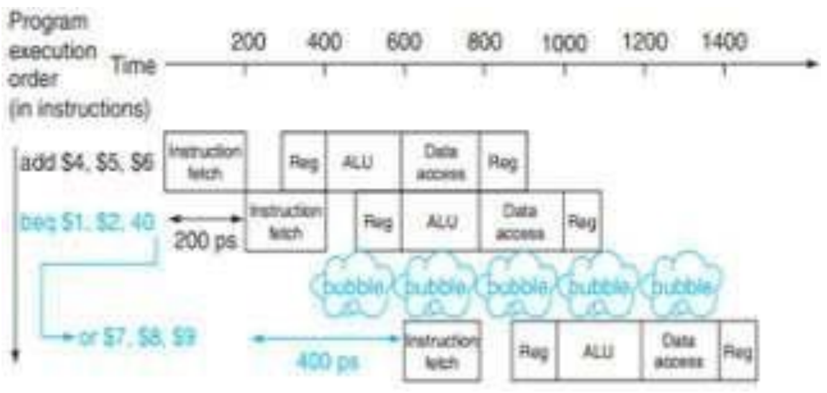


Fig 3.24 b) Branch taken

Branch Stalling

- This is stalling the instructions until the branch is complete is too slow.
- One improvement over branch stalling is to predict that the branch will not be taken and thus continue execution down the sequential instruction stream.
- If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.
- If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.
- To discard instructions, change the original control values to 0s.

Delayed Branches:

The delayed branch always executes the next sequential instruction, with the branch taking place after that one instruction delay. It is hidden from the MIPS assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer.

- One way to improve branch performance is to reduce the cost of the taken branch.
- The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty.

- Moving the branch decision up requires two actions to occur earlier:
  - Computing the branch target address
  - Evaluating the branch decision.
- The easy part of this change is to move up the branch address calculation.
- Despite these difficulties, moving the branch execution to the ID stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched.

3.9.2 Dynamic Branch Prediction

Prediction of branches at runtime using runtime information is called dynamic branch prediction.

- One implementation of that approach is a branch prediction buffer or branch history table.
- A **branch prediction buffer** is a small memory indexed by the lower portion of the address of the branch instruction.
- The memory contains a bit that says whether the branch was recently taken or not.

1 bit Prediction scheme

This scheme will be incorrect twice when not taken:

- Assume predict bit=0 to start (indicates branch not taken) and loop control is at the bottom of the code.
- First iteration in the loop, the predictor mispredict the branch since the branch is taken back to the top of the loop. Now invert the prediction bit (predict bit=1).
- Till the branch is taken, the prediction is correct.
- Exiting the loop, the predictor again mispredict the branch since this time the branch is not taken falling out of the loop. Now invert the prediction bit (**predict bit=0**).

Loop: first loop instruction

Second loopinstruction

--

--

--

Last loop instruction

Bne \$1,\$2, loop

Fall out instruction

2bit prediction scheme:

By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The two-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the midpoint of its range as the division between taken and not taken.

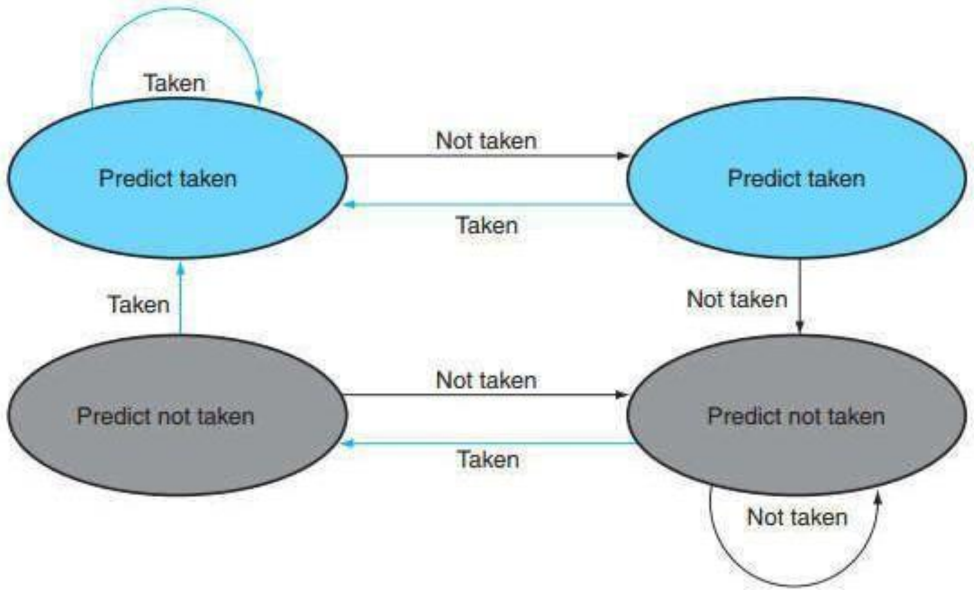


Fig 3.24: 2 bit prediction scheme

Branch delay slot:

- The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

- The limitations on delayed branch scheduling arise from the restrictions on the instructions that are scheduled into the delay slots the ability to predict at compile time whether a branch is likely to be taken or not.
- Delayed branching was a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle.
- As processors go to both longer pipelines and issuing multiple instructions per clock cycle, the branch delay becomes longer, and a single delay slot is insufficient.
- Hence, delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches.

3.10 EXCEPTIONS

Control is the most challenging aspect of processor design: One of the hardest parts of control is implementing exceptions and interrupts events other than branches or jumps that change the normal flow of instruction execution. They were initially created to handle unexpected events from within the processor, like arithmetic overflow. The term exception refer to any unexpected change in control flow without distinguishing whether the cause is internal or external. Interrupt is when the event is externally caused. The following are the causes of exceptions:

- R-type arithmetic overflow
- Executing undefined instruction
- I/O device request
- OS service request
- Hardware malfunction

Event	**Location**	**MIPS term**
I/O device request	External	Interrupt
OS service request	Internal	Exception
R-type arithmetic overflow	Internal	Exception
Executing undefined instruction	Internal	Exception
Hardware malfunction	Either	Exception / Interrupt

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a processor, which determines the clock cycle time and performance.

Exception Handling in the MIPS Architecture:

- The two types of exceptions that MIPS implementation can generate are execution of an undefined instruction and an arithmetic overflow.

A pipelined implementation treats exceptions as another form of control hazard. For example, suppose there is an arithmetic overflow in an add instruction. Flush the instructions that follow the add instruction from the pipeline and begin fetching instructions from the new address. This is done by turning the IF stage into a nop. Because of careful planning, the overflow exception is detected during the EX stage; hence, we can use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage. The final step is to save the address of the off ending instruction in the exception program counter (EPC). In reality, we save the address +4, so the exception handling the software routine must first subtract 4 from the saved value.

3.11 PARALLELISM VIA INSTRUCTIONS

The simultaneous execution of multiple instructions from a program is called Instruction Level Parallelism (ILP). It is a measure of how many of the instructions in a computer program can be executed simultaneously.

The ILP increases the depth of the pipeline to overlap more instructions. This is facilitated by adding extra hardware resources to replicate the internal component of the computer, so that it can launch multiple instructions in every pipeline stages. This is called **multiple issue.**

In Multiple Issue technique, multiple instructions are launched in one clock cycle.

This will improve the performance of the processor. The pipelined performance is estimated from the given formula (CPI-Cycles Per Instruction):

Pipeline CPI = Ideal CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stall

Launching multiple instructions per stage allows the instruction execution rate (CPI) to be less than 1. To obtain substantial increase in performance, we need to exploit parallelism across multiple basic blocks.

Implementing multiple issue processor

- **Static multiple issue processor:** Here the decisions are made by the compiler before execution.
- **Dynamic multiple issue processor:** Here the decisions are made during the execution by the processor.

The challenges in implementing a multiple issue pipeline are:

- **Packaging instructions into issue slots**: Issue slots are the positions from which instructions could be issued in a given clock cycle. To find the exact location of the current issue slot is the greatest challenge. So the process partially handled by the compiler. ; In dynamic issue designs, it is normally dealt with at runtime by the processor.
- **Dealing with data and control hazards:** In static issue processors, the consequences of data and control hazards are handled statically by the compiler. In dynamic issue processors, use hardware techniques to mitigate the control and data hazard.

3.11.1 Speculation

Speculation is an approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

- This allows the execution of complete instructions or parts of instructions before being certain whether this execution should take place.
- A commonly used form of speculative execution is control flow speculation where instructions past a control flow instruction are executed before the target of the control flow instruction is determined.
- Speculation may be done in the compiler or by the hardware.
- The uses speculation to reorder instructions, moving an instruction across a branch or a load across a store. The compiler usually inserts additional instructions that check the accuracy of the speculation and provide a fix-up routine to use when the speculation was incorrect.
- The processor hardware can perform the same transformation at runtime using techniques. The processor usually buffers the speculative results until it knows they are no longer speculative. If the speculation was correct, the instructions are completed by allowing the contents of the buffers to be written to the registers or memory. If the speculation was incorrect, the hardware flushes the buffers and reexecutes the correct instruction sequence.

Issue in Speculation:

Speculating on certain instructions may **introduce exceptions** that were formerly not present. The result would be that an exception that should not have occurred will occur. In

Compiler-based speculation, such problems are avoided by adding special speculation support that allows such exceptions to be ignored until it is clear that they really should occur. In hardware-based speculation, exceptions are simply buffered until it is clear that the instruction causing them is no longer speculative and is ready to complete; at that point the exception is raised, and normal exception handling proceeds.

3.11.2 Static Multiple Issue

The set of instructions that issues together in 1 clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

Static multiple-issue processors use compiler to assist with packaging instructions and handling hazards. The issue packet is treated as one large instruction with multiple operations. This is otherwise termed as **Very Long Instruction Word (VLIW).**Since the Intel IA-64 architecture supports this approach, it is known as **Explicitly Parallel Instruction Computer (EPIC**).

Loop unrolling is a technique used by compiler to solve static multiple issue.

Loop Unrolling is a technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different it rations are scheduled together.

Loop unrolling is a compiler optimization applied to certain kinds of loops to reduce the frequency of branches and loop maintenance instructions. It is easily applied to sequential array processing loops where the number of iterations is known prior to execution of the loop. After unrolling, there is more ILP available by overlapping instructions from different iterations.

- During the unrolling process, the compiler introduced additional registers, since multiple copies of the loop body are made.
- Augmenting new registers in loop unrolling is called **register renaming**. This is done to eliminate dependences that are not true data dependences, but may lead to potential hazards or may prevent the compiler from scheduling the code.
- To identify the independent instructions, it is necessary to trace the data dependencies.
- If there is no data values flow between the instructions, it is termed as **anti-dependence or name dependence.** This is an ordering forced purely by the reuse of a name.

- Renaming the registers during the unrolling process allows the compiler to the independent instructions for better code schedule.
- An **instruction group** is a sequence of consecutive instructions with no register data dependences among them.
- All the instructions in a group could be executed in parallel if sufficient hardware resources existed and if any dependences through memory were preserved.
- The compiler must explicitly indicate the boundary between one instruction group and another. This boundary is indicated by placing a stop between two instructions that belong to different groups.
- An explicit indicator of a break between independent and dependent instructions is termed as **stop**.
- **Predication** is a technique that can be used to eliminate branches by making the execution of an instruction dependent on a predicate, rather than dependent on a branch.
- Speculation and Predication improves ILP. Branches reduce the opportunity to exploit ILP by restricting the movement of code.
- Branches within a loop cannot be eliminated by loop unrolling. Predication eliminates this branch, by allowing more flexible exploitation of parallelism.
- **Speculation** consists of separate support for control speculation, which deals with deferring exceptions for speculated instructions, and memory reference speculation, which supports speculation of load instructions.
- Deferred exception handling is supported by adding speculative load instructions, which, when an exception occurs, tag the result as poison.
- **Poison** is the result generated when a speculative load yields an exception, or an instruction uses a poisoned operand. When a poisoned result is used by an instruction, the result is also poison, the software can then check for a poisoned result when it knows that the execution is no longer speculative.
- The speculation on memory references can be made by moving loads earlier than stores on which they may depend. This is done with an advanced load instruction.

- Advanced load is speculative load instruction with support to check for aliases that could invalidate the load. This demands the use of a special table to track the address that the processor loaded from.
- A subsequent instruction must be used to check the status of the entry after the load is no longer speculative.

3.11.2 Dynamic Multiple-Issue Processors

- Dynamic multiple issue processors are implemented using **superscalar processors** that are capable of executing more than one instruction per clock cycle.
- The compiler must schedule the instructions to the processors without any dependencies.
- To facilitate this, **dynamic pipeline scheduling** is performed by providing hardware support for reordering the order of instruction execution so as to avoid stalls.

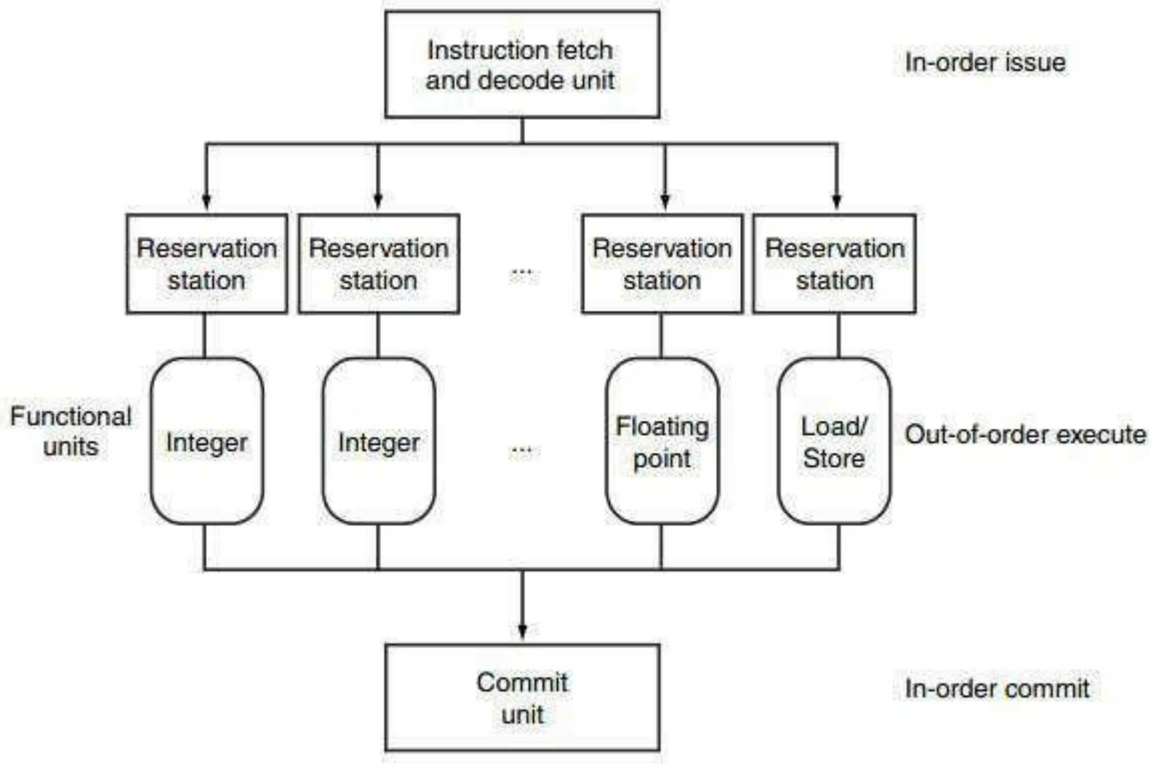


Fig 3.25: Units of dynamic scheduling pipeline

The following are the important components of dynamic scheduling pipelines:

- **Instruction Fetch Unit: This** unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution.
- **Functional unit:** They have buffers, called **reservation stations** that hold the operands and the operation. As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit.
- **Commit Unit:** This buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, called the **reorder buffer**, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

Operation of dynamic scheduling pipeline:

- When an instruction issues, if either of its operands is in the register file or the reorder buffer, it is copied to the reservation station immediately, where it is buffered until all the operands and an execution unit are available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.
- If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers.

Dynamic scheduling is often extended by including hardware-based speculation, especially for branch outcomes. By predicting the direction of a branch, a dynamically scheduled processor can continue to fetch and execute instructions along the predicted path.

UNIT - IV
MEMORYAND I/O ORGANIZATION

4.1 INTRODUCTION

Memory unit enables us to store data inside the computer. The computer memory always had here|s to **principle of locality.**

Principle of locality or locality of reference is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.

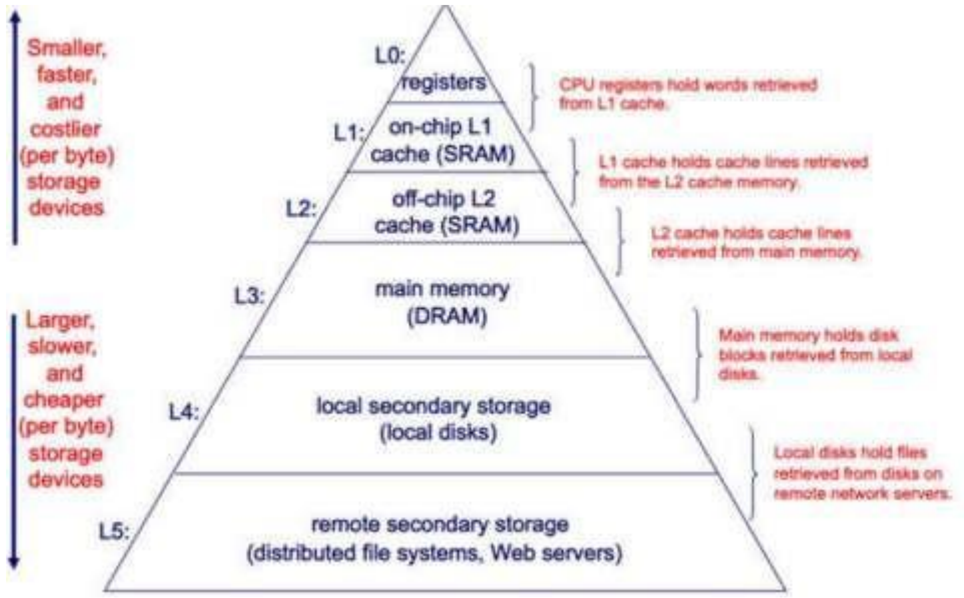
Two different types of locality are:

- **Temporal locality:** The principle stating that if a data location is referenced then it will tend to be referenced again soon.
- **Spatial locality:** The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

The locality of reference is useful in implementing the memory hierarchy.

Memory hierarchy is a structure that uses multiple levels of memories; as the distance from the CPU increases, the size of the memories and the access time both increase.

A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus smaller.


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0: registers
L1: on-chip L1 cache (SRAM)
L2: off-chip L2 cache (SRAM)
L3: main memory (DRAM)
L4: local secondary storage (local disks)
L5: remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from L1 cache.
L1 cache holds cache lines retrieved from the L2 cache memory.
L2 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

- Main memory is implemented from Dynamic Random Access Memory (DRAM).
- The levels closer to the processor (caches) use Static Random Access Memory (SRAM).
- DRAM is less costly per bit than SRAM, although it is substantially slower.
- For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- The computer programs tend to access the data at level k more often that at level k+1.
- The storage at level at k+1 can be slower

Cache memory (CPU memory) is high-speed SRAM that a computer Microprocessor can access more quickly than it can access regular RAM. This memory is typically integrated directly into the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU.

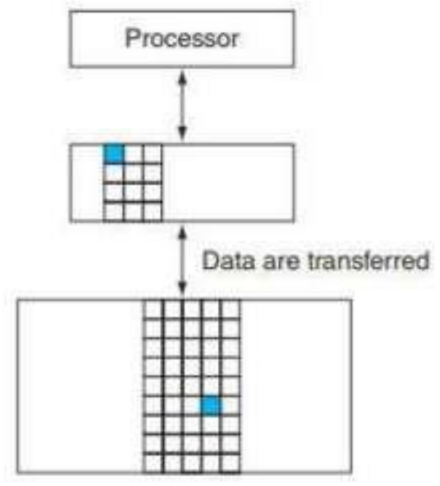


Fig 4.2: Data access by processor

The data transfer between various levels of memory is done through blocks. The minimum unit of information is called a **block.** If the data requested by the processor appears in some block in the upper level, this is called a **hit.** If the data is not found in the upper level, the request is called a **miss.** The lower level in the hierarchy is then accessed to retrieve the block containing the requested data.

The fraction of memory accesses found in a cache is termed as hit rate or hit ratio.

Miss rate is the fraction of memory accesses not found in a level of the memory hierarchy. Hit time is the time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

Miss penalty is the time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, and insert it in the level that experienced the miss.

Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty.

4.2 MEMORY HIERARCHY

A memory unit is a collection of semi-conductor storage cells with circuits to access the data stored in them. The data storage in memory is done in words. The number of bits in a word depends on the architecture of the computer. Generally a word is always multiple of 8. Memory is accessed through unique system assigned address. The accessing of data from memory is based on principle of locality.

4.2.1 Principle of Locality

The locality of reference or the principle of locality is the term applied to situations where the same value or related storage locations are frequently accessed. There are three basic types of locality of reference:

- **Temporal locality:** Here a resource that is referenced at one point in time is referenced again soon afterwards.
- **Spatial locality:** Here the likelihood of referencing a storage location is greater if a storage location near it has been recently referenced.
- **Sequential locality:** Here storage is accessed sequentially, in descending or ascending order. The locality or reference leads to memory hierarchy.

4.2.2 Need for memory hierarchy

Memory hierarchy is an approach for organizing memory and storage systems. It consist of multiple levels of memory with different speeds and sizes. The following are the reasons for such organization:

- Fast storage technologies cost more per byte and have less capacity
- Gap between CPU and main memory speed is widening
- Well-written programs tend to exhibit good locality.

The memory hierarchy is shown in Fig 4.1. The entire memory elements of the computer fall under the following three categories:

- **Processor Memory:**

  This is present inside the CPU for high-speed data access. This consists of small set of registers that act as temporary storage. This is the costliest memory component.

- **Primary memory:**

  This memory is directly accessed by the CPU. All the data must be brought inside main memory before accessing them. Semiconductor chips acts as main memory.

- **Secondary memory:**

  This is cheapest, large and relatively slow memory component. The data from the secondary memory is accessed by the CPU only after it is loaded to main memory.

There is a trade-off among the three key characteristics of memory namely-

- Cost
- Capacity
- Access time

Terminologies in memory access

- **Block or line:** The minimum unit of information that could be either present or totally absent.
- **Hit**: If the requested data is found in the upper levels of memory hierarchy it is called hit.

- **Miss:** If the requested data is not found in the upper levels of memory hierarchy it is called miss.
- **Hit rate or Hit ratio:** It is the fraction of memory access found in the upper level .It is a performance metric.

Hit Ratio = Hit/ (Hit + Miss)

- **Miss rate:** It is the fraction of memory access not found in the upper level (1-hit rate).
- **Hit Time:** The time required for accessing a level of memory hierarchy, including the time needed for finding whether the memory access is a hit or miss.
- **Miss penalty:** The time required for fetching a block into a level of the memory hierarchy from the lower level, including the time to access, transmit, insert it to new level and pass the block to the requestor.
- **Bandwidth:** The data transfer rate by the memory.
- **Latency or access time:** Memory latency is the length of time between the memory's receipt of a read request and its release of data corresponding with the request.
- **Cycle time:** It is the minimum time between requests to memory.

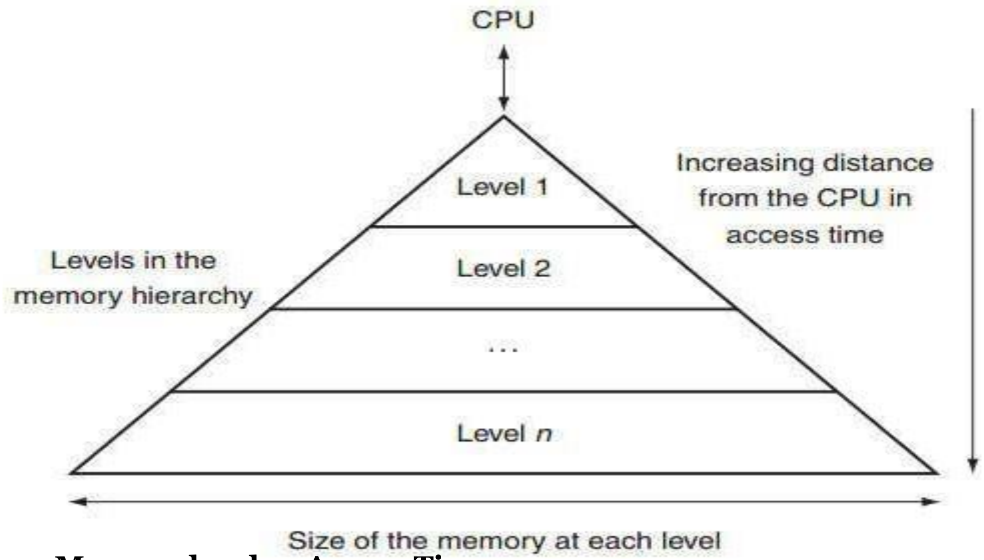

CPU
Level 1
Level 2
...
Level n
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

Fig 4.2: Memory level vs Access Time

The memory access time increases as the level increases. Since the CPU registers are located in very close proximity to the CPU they can be accessed very quickly and they are the more costly. As the level increases, the memory access time also increases thereby decreasing the costs.

4.2.3 Levels in Memory Hierarchy

The following are the levels in memory hierarchy:

- **CPU Registers:**

  They are at the top most level of this hierarchy, they hold the most frequently used data. They are very limited in number and are the fastest. They are often used by the CPU and the ALU for performing arithmetic and logical operations, for temporary storage of data.

- **Static Random Access Memory (SRAM):**

  Static Random Access Memory (Static RAM or SRAM) is a type of RAM that holds data in a static form, that is, as long as the memory has power. SRAM stores a bit of data on four transistors using two cross-coupled inverters. The two stable states characterize 0 and 1. During read and write operations another two access transistors are used to manage the availability to a memory cell.

- **Main memory or Dynamic Random Access Memory (DRAM):**

  Dynamic random access memory (DRAM) is a type of memory that is typically used for data or program code that a computer processor needs to function. In other words it is said to be the main memory of the computer. Random access allows processor to access any part of the memory directly rather than having to proceed sequentially from a starting place. The main advantages of DRAM are its simple design, speed and low cost in comparison to alternative types of memory. The main disadvantages of DRAM are volatility and high power consumption relative to other options.

- **Local Disks (Local Secondary Storage):**

  A local drive is a computer disk drive that is installed directly within the host or the local computer.)t is a computer's native hard disk drive (HDD), which is directly accessed by the computer for storing and retrieving data. It is a cheaper memory with more memory access time.

- **Remote Secondary Storage:**

  This includes Distributed file system (DFS) and online storage like cloud. The storage area is vast with low cost but larger access time.

Distinction between Static RAM and Dynamic RAM

SRAM	DRAM
Stores data till the power is supplied. Uses nearly 6 transistors for each memory cell.	Stored data only for few milliseconds irrespective of the power supply. Uses single transistor and capacitor for each memory cell.
Do not refresh the memory cell.	Refreshing circuitry is needed.
Faster data access.	Slower access.
Consumes more power.	Low power consumption.
Cost pet bit is high.	Comparatively lower costs.
They are made of more number of components per cells.	They are made of less number of components per cells.

4.3 CLASSIFICATION O MEMORY

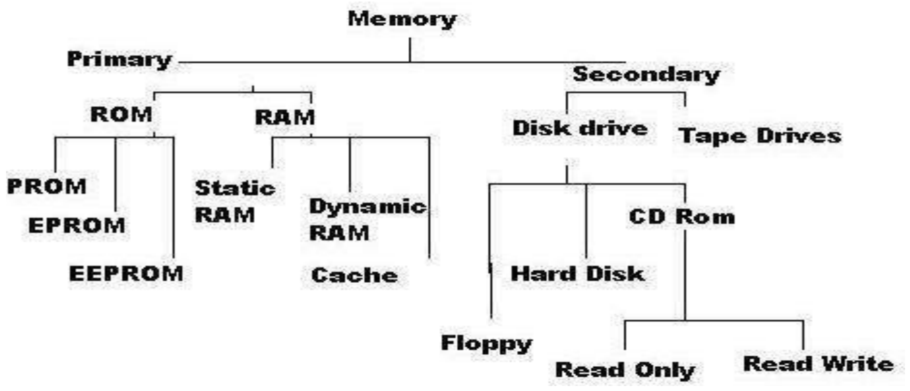


Fig 4.3: Classification of Memory

The instructions and data are stored in memory unit of the computer system are divided into following main groups:

- Main or Primary memory
- Secondary memory.

Primary Memory:

Primary memory is the main area in a computer in which data is stored for quick access by the computer¦s processor.)t is divided into two parts:

i) Random Access Memory (RAM):

RAM is a type of computer primary memory. It accessed any piece of data at any time. RAM stores data for as long as the computer is switched on or is in use. This type of memory is volatile. The two types of RAM are:

- **Static RAM:** This type of RAM is static in nature, as it does not have to be refreshed at regular intervals. Static RAM is made of large number of flip-flops on IC. It is being costlier and having packing density.
- **Dynamic RAM:** This type of RAM holds each bit of data in an individual capacitor in an integrated circuit. It is dynamic in the sense that the capacitor charge is repeatedly refreshed to ensure the data remains intact.

ii) Read Only Memory (ROM):

The ROM is nonvolatile memory. It retains stored data and information if the power is turned off.)n ROM, data are stored permanently and can¦t alter by the programmer. There are four types of ROM:

- □ **MROM (mask ROM):** MROM (mask ROM) is manufacturer-Programmed ROM in which data is burnt in by the manufacturer of the electronic equipment in which it is used and it is not possible for a user to modify programs or data stored inside the ROM chip.
- □ **PROM (programmable ROM):** PROM is one in which the user can load and store ¦read-only¦ programs and data.)n PROM the programs or data are stored only fast time and the stored data cannot modify the user.
- □ **EPROM (erasable programmable ROM):** EPROM is one in which is possible to erase information stored in an EPROM chip and the chip can be reprogrammed to store new information. When an EPROM is in use, information stored in it can only be ¦read¦ and the information remains in the chip until it is erased.
- □ **EEPROM (electronically erasable and programmable ROM):** EEPROM is one type of EPROM in which the stored information is erased by using high voltage electric pulse. It is easier to alter information stored in an EEPROM chip.

Secondary Memory:

Secondary memory is where programs and data are kept on a long time basis. It is cheaper from of memory and slower than main or primary memory. It is non-volatile and cannot access data directly by the computer processor. It is the external memory of the computer system.

Example: hard disk drive, floppy disk, optical disk/ CD-ROM.

4.4 MEMORY CHIP ORGANISATION

A memory consists of cells in the form of an array. The basic element of the semiconductor memory is the **cell**. Each cell is capable of storing one bit of information. Each row of the cells constitutes a memory words and all cells of a row are connected to a common line referred to as a **word line.** AW×b memory has w words, each word having 'b' number of bits.

The basic memory element called cell can be in two states (0 or 1). The data can be written into the cell and can be read from it.

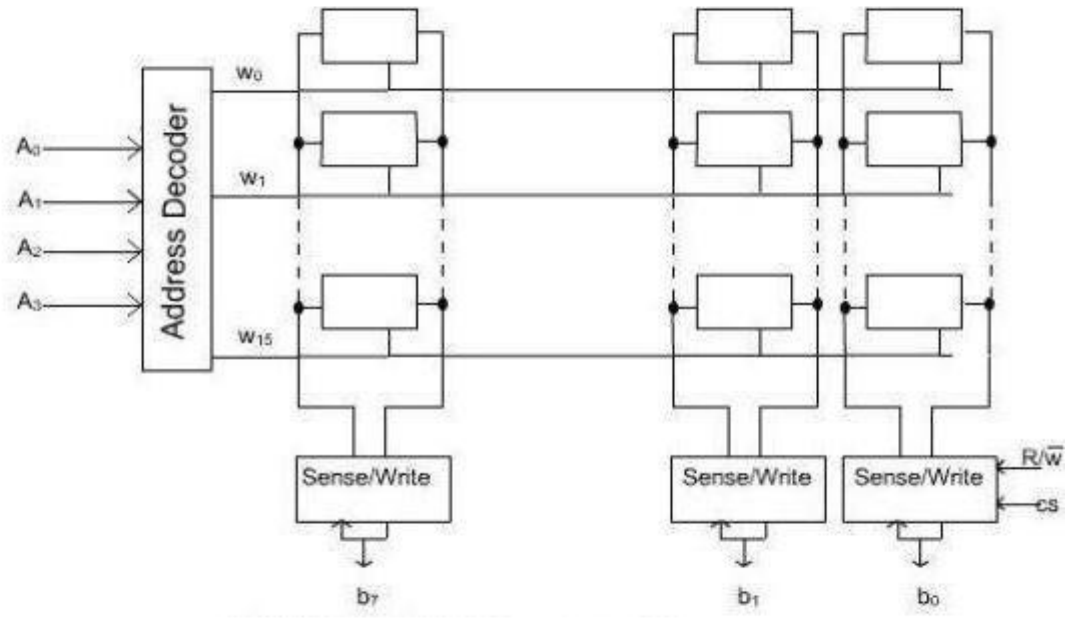


Fig 4.4: Organization of 16 x 8 memory

- In the above diagram there are 16 memory locations named as w_0, w_1, w_3...w_{15}. Each location can store at most 8 bits of data (b_0, b_1, b_3... b_7). Each location (w_n) is the word line. The word line of Fig 4.4 is 8.
- Each row of the cell is a memory word. The memory words are connected to a common line termed as word line. The word line is activated based on the address it receives from the address bus.
- An address decoder is used to activate a word line.
- The cells in the memory are connected by two **bit lines** (column wise). These are connected to data input and data output lines through sense/ write circuitry.
- **Read Operation:** During read operation the sense/ write circuit reads the information by selecting the cell through word line and bit lines. The data from this cell is transferred through the output data line.
- **Write Operation:** During write operation, the sense/ write circuitry gets the data and writes into the selected cell.
- The data input and output line of sense / write circuit is connected to a bidirectional data line.
- It is essential to have n bus lines to read 2^nwords.

Organization of 1M x 1 memory chip:

The organization of 1024 x 1 memory chips, has 1024 memory words of size 1 bit only. The size of data bus is 1 bit and the size of address bus is 10 bits. A particular memory location is identified by the contents of memory address bus. A decoder is used to decode the memory address.

Organization of memory word as a row:

- The whole memory address bus is used together to decode the address of the specified location.

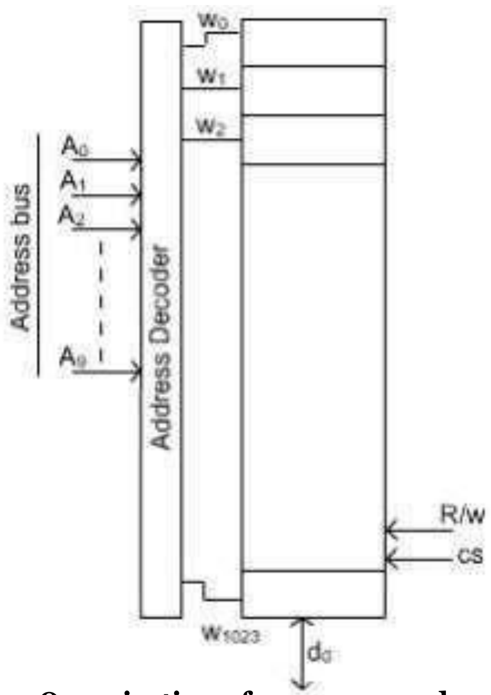


Fig 4.5: Organization of memory word as row

Organization of several memory words in row:

- One group is used to form the row address and the second group is used to form the column address.
- The 10-bit address is divided into two groups of 5 bits each to form the row and column address of the cell array.
- A row address selects a row of 32 cells, all of which could be accessed in parallel.
- Regarding the column address, only one of these cells is connected to the external data line via the input output multiplexers

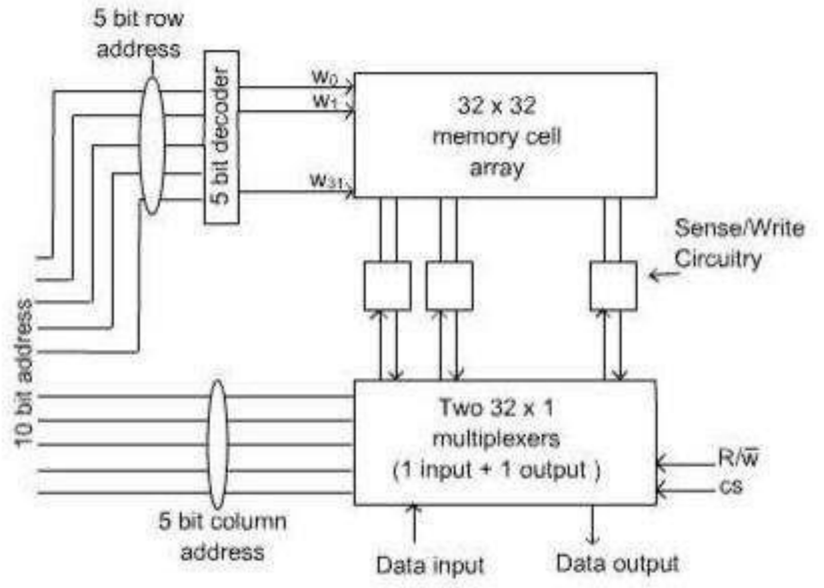


Fig 4.6: Organization of several memory words in row

Signals used in memory chip:

- A memory unit of 1MB size is organized as 1M x 8 memory cells.
- It has got220 memory location and each memory location contains 8 bits of information.
- The size of address bus is20 and the size of data bus is 8.
- The number of pins of a memory chip depends on the data bus and address bus of the memory module.
- To reduce the number of pins required for the chip, the cells are organized in the form of a square array.
- The address bus is divided into two groups, one for column address and other one is for row address.
- In this case, high- and low-order 10 bits of 20-bitaddress constitute of row and column address of a given cell, respectively.
- In order to reduce the number of pin needed for external connections, the row and column addresses are multiplexed on tenpins.

- During a Read or a Write operation, the row address is applied first. In response to a signal pulse on the Row Address Strobe (RAS) input of the chip, this part of the address is loaded into the row address latch.
- All cell of this particular row is selected. Shortly after the row address is latched, the column address is applied to the address pins.
- It is loaded into the column address latch with the help of Column Address Strobe (CAS) signal, similar to RAS.
- The information in this latch is decoded and the appropriate Sense/Write circuit is selected.

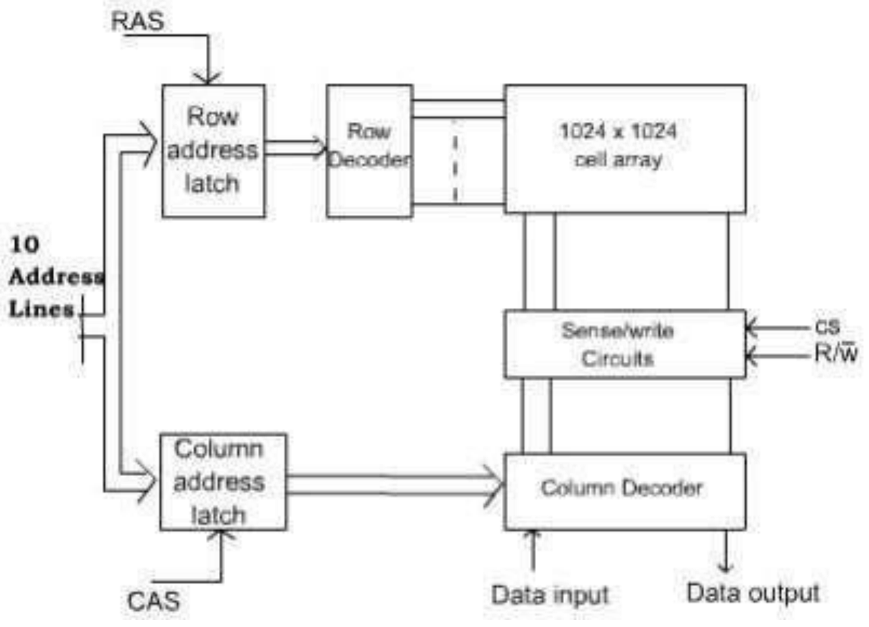


Fig 4.7: Signals in accessing the memory

- Each chip has a control input line called Chip Select (CS). A chip can be enabled to accept data input or to place the data on the output bus by setting its Chip Select input to 1.
- The address bus for the 64K memory is 16 bits wide.
- The high order two bits of the address are decoded to obtain the four chip select control signals.

- The remaining 14 address bits are connected to the address lines of all the chips.
- They are used to access a specific location inside each chip of the selected row.
- The R/ W inputs of all chips are tied together to provide a common read / write control.

4.3 CACHE MEMORY

The cache memory exploits the locality of reference to enhance the speed of the processor.

Cache memory or CPU memory, is high-speed SRAM that a processor can access more quickly than a regular RAM. This memory is integrated directly into the CPU chip or placed on a separate chip that has a separate bus interconnect with the CPU.

The cache memory stores instructions and data that are more frequently used or data that is likely to be used next. The processor looks first in the cache memory for the data. If it finds the instructions or data then it does perform a more time-consuming reading of data from larger main memory or other data storage devices.

The processor do not need to know the exact location of the cache. It can simply issue read and write instructions. The cache control circuitry determines whether the requested data resides in the cache.

- **Cache and temporal reference:** When data is requested by the processor, the data should be loaded in the cache and should be retained till it is needed again.
- **Cache and spatial reference:** Instead of fetching single data, a contiguous block of data is loaded into the cache.

Terminologies in Cache

- **Split cache:** It has separate data cache and a separate instruction cache. The two caches work in parallel, one transferring data and the other transferring instructions.
- **A dual or unified cache:** The data and the instructions are stored in the same cache. A combined cache with a total size equal to the sum of the two split caches will usually have a better hit rate.
- **Mapping Function:** The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

- **Cache Replacement:** When the cache is full and a memory word 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 is the replacement algorithm.

4.3.1 Cache performance:

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache. If the processor finds that the memory location is in the cache, a **cache hit** has said to be occurred. If the processor does not find the memory location in the cache, a **cache miss** has occurred. When a cache miss occurs, the cache replacement is made by allocating a new entry and copies in data from main memory. The performance of cache memory is frequently measured in terms of a quantity called **Hit ratio.**

Hit ratio = hit / (hit + miss) = Number of hits/ Total accesses to the cache

Miss penalty or cache penalty is the sum of time to place a bock in the cache and time to deliver the block to CPU.

Miss Penalty = time for block replacement + time to deliver the block to CPU

Cache performance can be enhanced by using higher cache block size, higher associativity, reducing miss rate, reducing miss penalty, and reducing the time to hit in the cache. CPU execution Time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services.

CPU execution time=(CPU clock cycles + memory stall cycles (if any))
x Clock cycle time

The **memory stall cycles** are a measure of count of the memory cycles during which the CPU is waiting for memory accesses. This is dependent on caches misses and cost per miss (cache penalty).

Memory stall cycles = number of cache misses x miss penalty

 Instruction Count x (misses/ instruction) x miss penalty

 Instruction Count (IC) x (memory access/ instruction) x miss penalty

 IC x Reads per instruction x Read miss rate X Read miss penalty + IC x Write per instruction x Write miss rate X Write miss penalty

Misses / instruction = (miss rate x memory access)/ instruction

Issues in Cache memory:

- **Cache placement:** where to place a block in the cache?
- **Cache identification:** how to identify that the requested information is available in the cache or not?
- **Cache replacement:** which block will be replaced in the cache, making way for an incoming block?

4.3.2 Cache Mapping Policies:

These policies determine the way of loading the main memory to the cache block. Main memory is divided into equal size partitions called as **blocks or frames.** The cache memory is divided into fixed size partitions called as **lines.** During cache mapping, block of main memory is copied to the cache and further access is made from the cache not from the main memory.

Cache mapping is a technique by which the contents of main memory are brought into the cache memory.

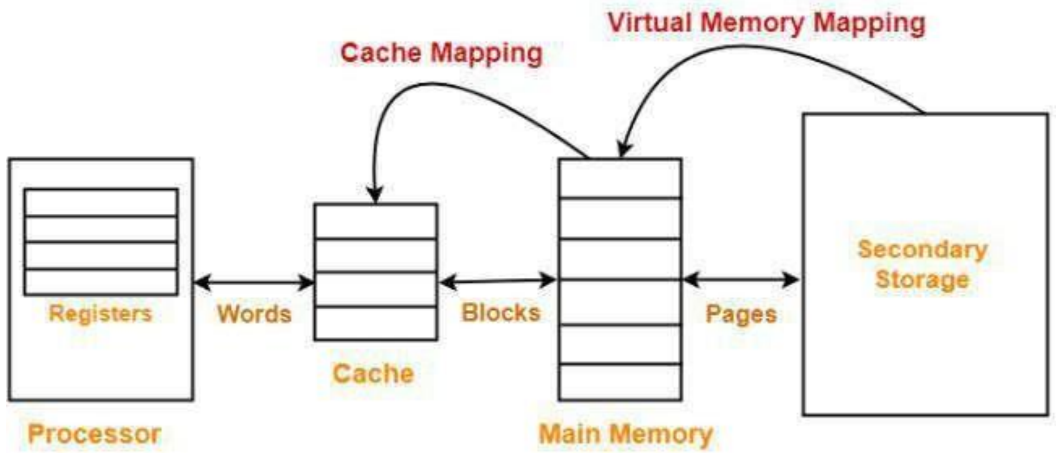


Fig 4.8: Cache mapping

There are three different cache mapping policies or mapping functions:

- Direct mapping
- Fully Associative mapping
- Set Associative mapping

Direct Mapping

- The simplest technique is direct mapping that maps each block of main memory into only one possible cache line.
- Here, each memory block is assigned to a specific line in the cache.
- If a line is previously taken up by a memory block and when a new block needs to be loaded, then the old block is replaced.
- Direct mapping s performance is directly proportional to the (it ratio.

The direct mapping concept is if the i^th^ block of main memory has to be placed at the j^th^ block of cache memory j = i % (number of blocks in cache memory)

- Consider a 128 block cache memory. Whenever the main memory blocks 0, 128, 256 are loaded in the cache, they will be allotted cache block 0, since j= (0 or 128 or 256) % 128 is zero).
- Contention or collision is resolved by replacing the older contents with latest contents.
- The placement of the block from main memory to the cache is determined from the 16 bit memory address.
- The lower order four bits are used to select one of the 16 words in the block.
- The 7 bit block field indicates the cache position where the block has to be stored.
- The 5 bit tag field represents which block of main memory resides inside the cache.
- This method is easy to implement but is not flexible.
- **Drawback:** The problem was that every block of main memory was directly mapped to the cache memory. This resulted in high rate of conflict miss. Cache memory has to be very frequently replaced even when other blocks in the cache memory were present as empty.

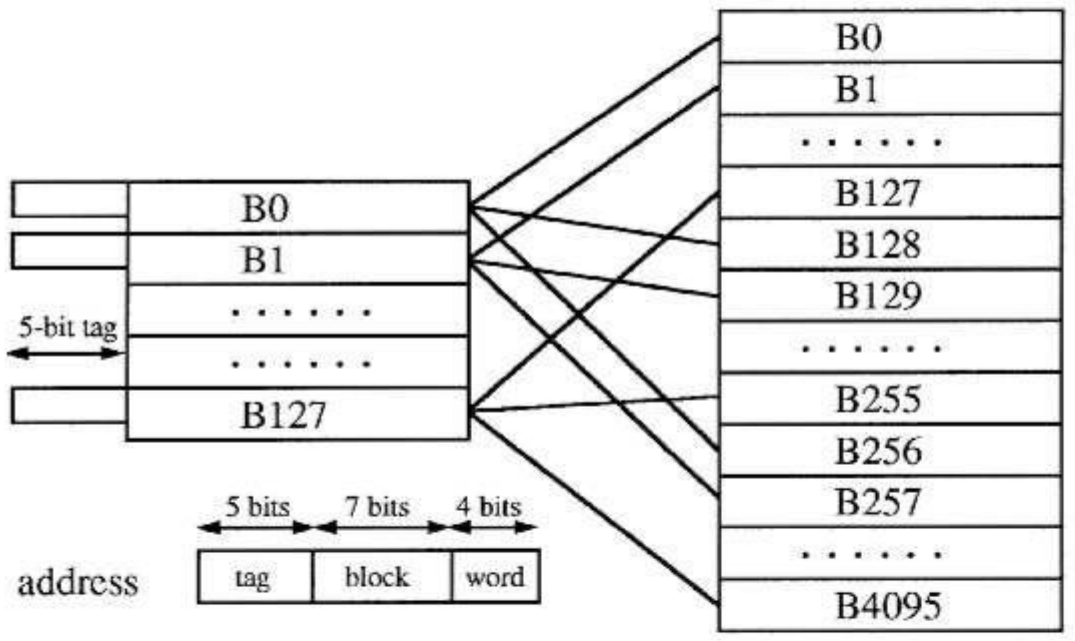


Fig 4.9: Direct memory mapping

Associative Mapping:

- The associative memory is used to store content and addresses of the memory word.
- Any block can go into any line of the cache. The 4 word id bits are used to identify which word in the block is needed and the remaining 12 bits represents the tag bit that identifies the main memory block inside the cache.
- This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form.
- The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to check, if the desired block is present. Hence it is known as Associative Mapping technique.
- Cost of an associated mapped cache is higher than the cost of direct-mapped because of the need to search all 128 tag patterns to determine whether a block is in cache.

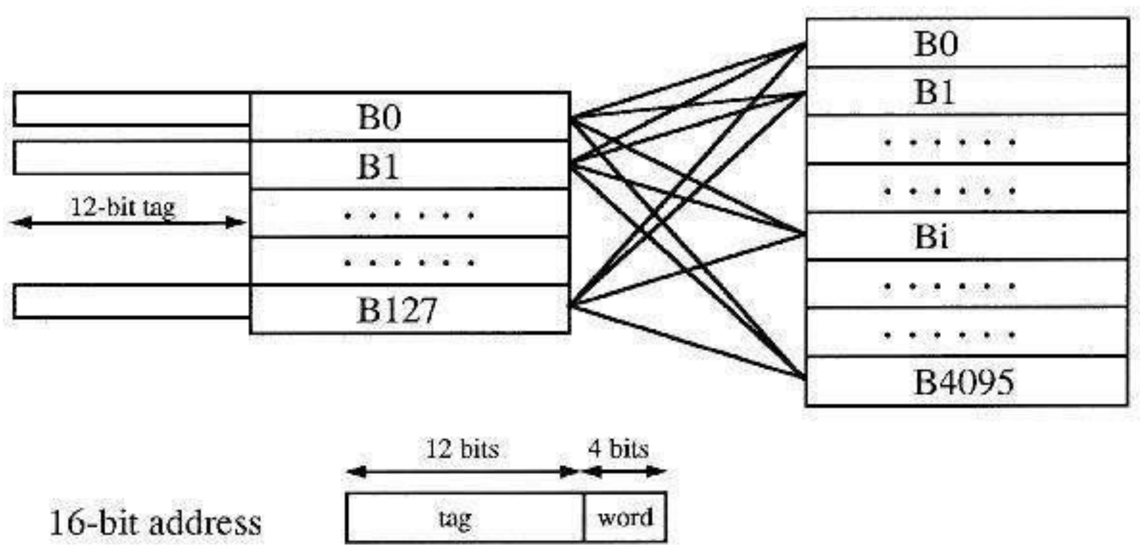


Fig 4.10: Associative Mapping

Set associative mapping:

- It is the combination of direct and associative mapping technique.
- Cache blocks are grouped into sets and mapping allow block of main memory to reside into any block of a specific set.
- This reduces contention problem (issue in direct mapping) with low hardware cost (issue in associative mapping).
- Consider a cache with two blocks per set.)n this case, memory block ど, はね, なにぱ,…..,ねどぬに map into cache set 0 and they can occupy any two block within this set.
- It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a set. Then a block in memory can map to any one of the lines of a specific set.
- The 6 bit set field of the address determines which set of the cache might contain the desired block. The tag bits of address must be associatively compared to the tags of the two blocks of the set to check if desired block is present.

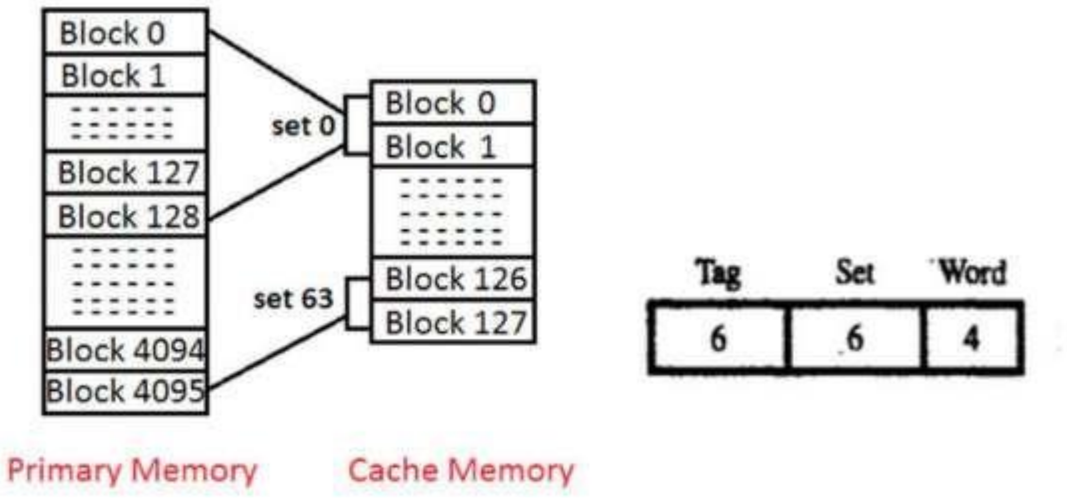


Fig 4.11: Set associative mapping

4.3.3 Handling Cache misses:

When a program accesses a memory location that is not in the cache, it is called a cache miss. The performance impact of a cache miss depends on the latency of fetching the data from the next cache level or main memory. The cache miss handling is done with the processor control unit and with a separate controller that initiates the memory access and refills the cache. The following are the steps taken when a cache miss occurs:

- Send the original PC value (PC - 4) to the memory.
- Instruct main memory to perform a read and wait for the memory to complete its access.
- Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
- Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.

4.3.4 Writing to a cache:

- Suppose on a store instruction, the data is written into only the data cache (without changing main memory); then, after the write into the cache, memory would have a different value from that in the cache. This leads to inconsistency.

- The simplest way to keep the main memory and the cache consistent is to always write the data into both the memory and the cache. This scheme is called write-through.

Write through is a scheme in which writes always update both the cache and the memory, ensuring that data is always consistent between the two.

- With a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time.
- A potential solution to this problem is deploying write buffer.
- A write buffer stores the data while it is waiting to be written to memory.
- After writing the data into the cache and into the write buffer, the processor can continue execution.
- When a write to main memory completes, the entry in the write buffer is freed.
- If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer.
- If the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help because writes are being generated faster than the memory system can accept them.

Write buffer is a queue that holds data while the data are waiting to be written to memory.

iii) The rate at which writes are generated may also be less than the rate at which the memory can accept them, and yet stalls may still occur. To reduce the occurrence of such stalls, processors usually increase the depth of the write buffer beyond a single entry.

iv) Another alternative to a write-through scheme is a scheme called write-back. When a write occurs, the new value is written only to the block in the cache.

v) The modified block is written to the lower level of the hierarchy when it is replaced.

vi) Write-back schemes can improve performance, especially when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than write-through.

Write-back is a scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

4.3.5 Cache Replacement Algorithms

When a main memory block needs to be brought into the cache while all the blocks are occupied, then one of them has to be replaced. This selection of the block to be replaced is using cache replacement algorithms. Replacement algorithms are only needed for associative and set associative techniques. The following are the common replacement techniques:

- **Least Recently Used (LRU):** This replaces the cache line that has been in the cache the longest with no references to it.
- **First-in First-out (FIFO):** This replaces the cache line that has been in the cache the longest.
- **Least Frequently Used (LFU):** This replaces the cache line that has experienced the fewest references.
- **Random:** This picks a line at random from the candidate lines.

Example 4.1: Program P runs on computer A in 10 seconds. Designer says clock rate can be increased significantly, but total cycle count will also increase by 20%. What clock rate do we need on computer B for P to run in 6 seconds? (Clock rate on A is 100 MHz). The new machine is B. We want CPU Time_B = 6 seconds.
We know that Cycles count_B = 1.2 Cycles count_A. Calculate Cycles count_A. CPU Time_A = 10 sec. = ; Cycles count_A = 1000 x 106 cycles Calculate Clock rate_B:

CPU Time_B = 6 sec. = ; Clock rate_B = = 200 MHz
Machine B must run at twice the clock rate of A to achieve the target execution time.

Example 4.2: We have two machines with different implementations of the same ISA. Machine A has a clock cycle time of 10 ns and a CPI of 2.0 for program P; machine B has a clock cycle time of 20 ns and a CPI of 1.2 for the same program. Which machine is faster? Let IC be the number of instructions to be executed. Then Cycles count_A = 2.0 IC

Cycles count_B = 1.2 IC
calculate CPU Time for each machine:
CPU Time_A = 2.0 IC x 10 ns = 20.0 IC ns
CPU Time_B = 1.2 IC x 20 ns = 24.0 IC ns
» Machine A is 20%faster.

Example 4.3: Consider an implementation of MIPS ISA with 500 MHz clock and

– each ALU instruction takes 3 clock cycles,

– each branch/jump instruction takes 2 clock cycles,

– each sw instruction takes 4 clock cycles,

– eachlw instruction takes 5 clock cycles.
Also, consider a program that during its execution executes:

– x=200 million ALU instructions

– y=55 million branch/jump instructions

– z=25 million sw instructions

– w=20 million lw instructions
Find CPU time. Assume sequentially executing CPU.

Clock cycles for a program = (3x + 2y + 4z + 5w)

= 910 x 10^6 clock cycles CPU_time = Clock cycles for a program / Clock rate

= 910 x 10^6 / 500 x 10^6 = 1.82 sec

Example 4.4: Consider another implementation of MIPS ISA with 1 GHz clock and

– each ALU instruction takes 4 clock cycles,

– each branch/jump instruction takes 3 clock cycles,

– each sw instruction takes 5 clock cycles,

– eachlw instruction takes 6 clock cycles.
Also, consider the same program as in Example 1.
Find CPI and CPU time. Assume sequentially executing CPU.

CPI = (4x + 3y + 5z + 6w) / (x + y + z + w)

= 4.03 clock cycles/ instruction

CPU time = Instruction count x CPI / Clock rate

= (x+y+z+w) x 4.03 / 1000 x 10^6

= 300 x10^6 x 4.03 /1000 x 10^6

= 1.21 sec

4.3 VIRTUAL MEMORY

Virtual memory is a memory management capability of an operating system that uses hardware and software to allow a computer to compensate for physical memory shortages by temporarily transferring data from RAM to disk storage.

The concept of virtual memory in computer organization is allocating memory from the hard disk and making that part of the hard disk as a temporary RAM. In other words, it is a technique that uses main memory as a cache for secondary storage. The motivations for virtual memory are:

- To allow efficient and safe sharing of memory among multiple programs
- To remove the programming burdens of a small, limited amount of main memory.

Virtual memory provides an illusion to the users that the PC has enough primary memory left to run the programs. Sometimes the size of programs to be executed may sometimes very bigger than the size of primary memory left, the user never feels that the system needs a bigger primary storage to run that program. When the RAM is full, the operating system occupies a portion of the hard disk and uses it as a RAM. In that part of the secondary storage, the part of the program which not currently being executed is stored and all the parts of the program that are executed are first brought into the main memory. This is the theory behind **virtual memory.**

Terminologies:

- **Physical address** is an address in main memory.
- **Protection** is a set of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, with one another by reading or writing each other's data.
- Virtual memory breaks programs into fixed-size blocks called **pages.**
- **Page fault** is an event that occurs when an accessed page is not present in main memory.
- **Virtual address** is an address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.
- **Address translation or address mapping** is the process by which a virtual address is mapped to an address used to access memory.

Working mechanism

- In virtual memory, blocks of memory are mapped from one set of addresses (virtual addresses) to another set (physical addresses).

- The processor generates virtual addresses while the memory is accessed using physical addresses.
- Both the virtual memory and the physical memory are broken into pages, so that a virtual page is really mapped to a physical page.
- It is also possible for a virtual page to be absent from main memory and not be mapped to a physical address, residing instead on disk.
- Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.
- Virtual memory also simplifies loading the program for execution by providing relocation. **Relocation** maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory. This relocation allows us to load the program anywhere in main memory.

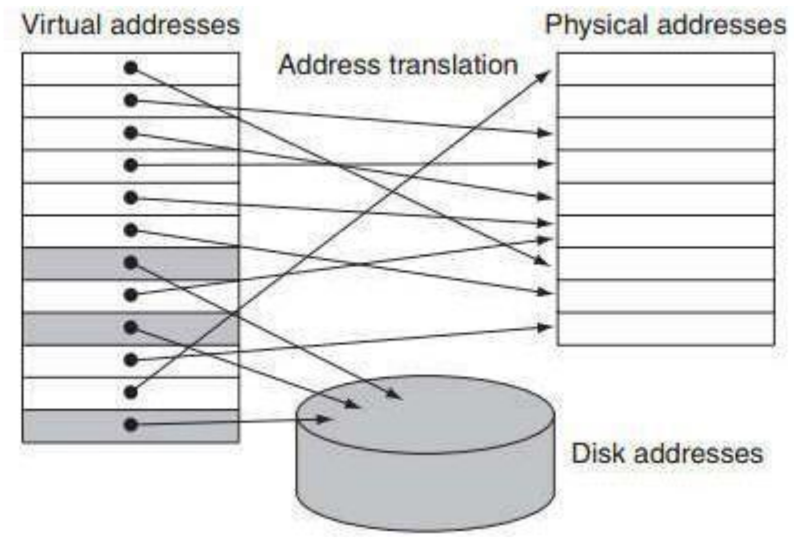


Fig 4.12: Mapping of virtual and physical memory

4.3.1 Addressing in virtual memory

- A virtual address is considered as a pair (p,d) where lower order bits give an offset d within the page and high-order bits specify the page p.
- The job of the Memory Management Unit (MMU) is to translate the page number p to a frame number f.

- The physical address is then (f,d), and this is what goes on the memory bus.
- For every process, there is a page and page-number p is used as an index into this array for the translation.
- The following are the entries in page tables:
    1. Validity bit: Set to 0 if the corresponding page is not in memory
    2. Frame number: Number of bits required depends on size of physical memory
    3. Protection bits: Read, write, execute accesses
    4. Referenced bit is set to 1 by hardware when the page is accessed: used by page replacement policy
    5. Modified bit (dirty bit) set to 1 by hardware on write-access: used to avoid writing when swapped out.

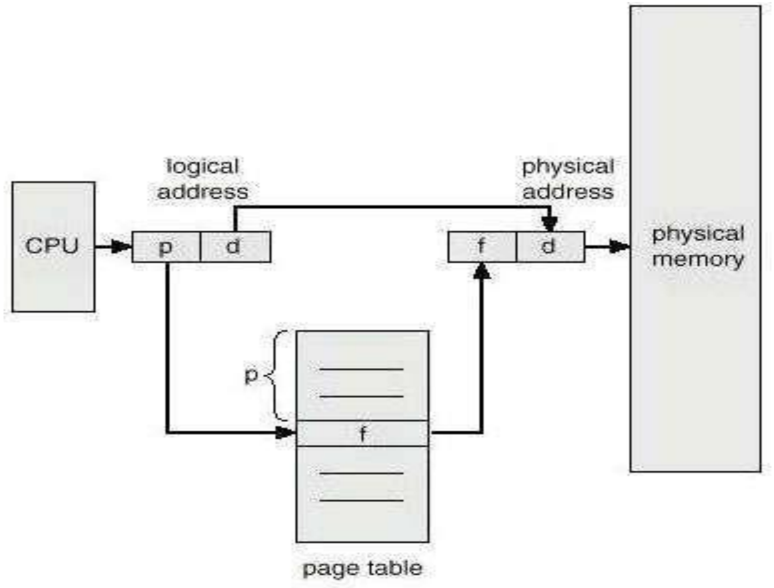


Fig 4.13: Conversion of logical address to physical address

Role of control bit in page table

The control bit (v) indicates whether the page is loaded in the main memory. It also indicates whether the page has been modified during its residency in the main memory. This information is crucial to determine whether to write back the page to the disk before it is removed from the main memory during next page replacement.

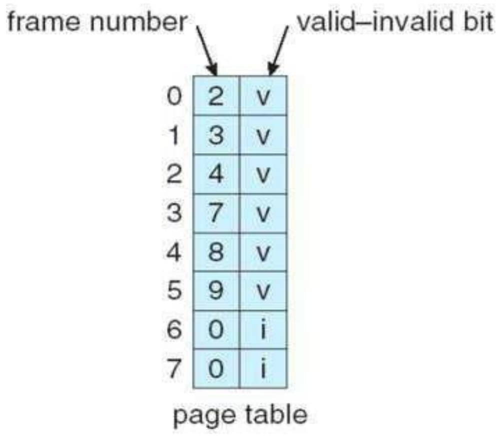


Fig 4.14: Page table

4.3.2 Page faults and page replacement algorithms

A page fault occurs when a page referenced by the CPU is not found in the main memory. The required page has to be brought from the secondary memory into the main memory. A page that is currently residing in the main memory, has to be replaced if all the frames of main memory are already occupied.

Page replacement is a process of swapping out an existing page from the frame of a main memory and replacing it with the required page.

Page replacement is done when all the frames of main memory are already occupied and a page has to be replaced to create a space for the newly referenced page. A good replacement algorithm will have least number of page faults.

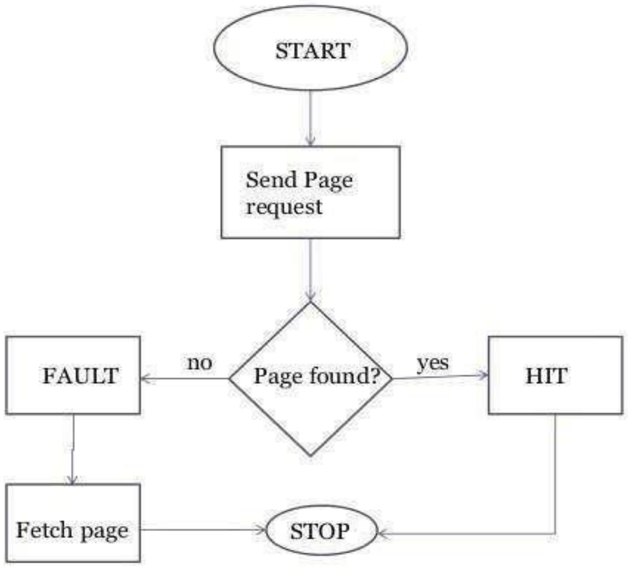


Fig 4.14: Occurrence of page fault

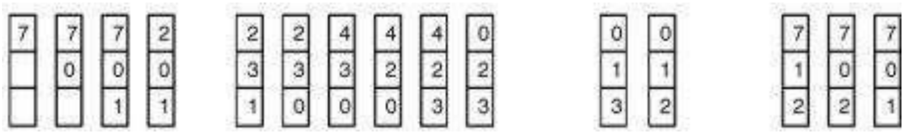
The following are the page replacement algorithms:

1. FIFO Page Replacement Algorithm
2. LIFO Page Replacement Algorithm
3. LRU Page Replacement Algorithm
4. Optimal Page Replacement Algorithm
5. Random Page Replacement Algorithm

1. **First In First Out (FIFO) page replacement algorithm**

It replaces the oldest page that has been present in the main memory for the longest time. It is implemented by keeping track of all the pages in a queue.

Example 4.5. Find the page faults when the following pages are requested to be loaded in a page frame of size 3: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



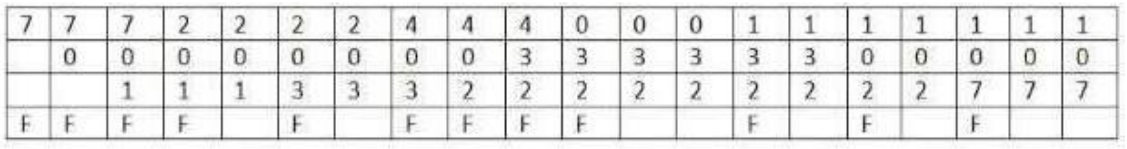
Page faults= 15

2. **Last In First Out (LIFO) page replacement algorithm**

It replaces the newest page that arrived at last in the main memory. It is implemented by keeping track of all the pages in a stack.

3. **Least Recently Used (LRU) page replacement algorithm** The new page will be replaced with least recently used page.

Example 4.6: Consider the following reference string. Calculate the number of page faults when the page frame size is 3 using LRU policy.7, 0, 1, 2, 0, 3, 4, 2, 3, 0, 3, 2,1,2,0,1,7,0,1



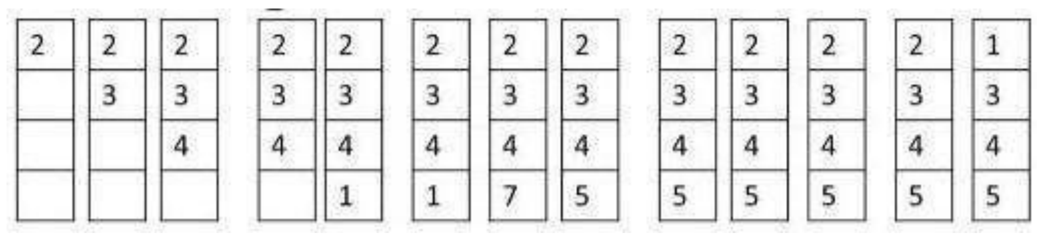
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
F	F	F	F		F		F	F	F	F			F		F		F		

Page faults= 12 (F bit indicates the occurrence of page faults)

4. **Optimal page replacement algorithm**

In this method, pages are replaced which would not be used for the longest duration of time in the future.

ample 4.7: Find the number of misses and hits while using optimal page replacement algorithm on the following reference string with page frame size as 4: 2, 3, 4, 2, 1, 3, 7, 5, 4, 3, 2, 3, 1.



Page fault=13Number of page hit= 6Number of page misses=7

5. **Random page replacement algorithms**

Random replacement algorithm replaces a random page in memory. This eliminates the overhead cost of tracking page references.

4.3.3 Translation Look aside Buffer (TLB)

A translation look aside buffer (TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval.

The page tables are stored in main memory and every memory access by a program to the page table takes longer time. This is because it does one memory access to obtain the physical address and a second access to get the data. The virtual to physical memory address translation occurs twice. But a TLB will exploit the locality of reference and can reduce the memory access time.

TLB hit is a condition where the desired entry is found in translation look aside buffer.
If this happens then the CPU simply access the actual location in the main memory.

If the entry is not found in TLB (TLB miss) then CPU has to access page table in the main memory and then access the actual frame in the main memory. Therefore, in the case of TLB hit, the effective access time will be lesser as compare to the case of TLB miss.

If the probability of TLB hit is P% (TLB hit rate) then the probability of TLB miss (TLB miss rate) will be (1-P) %. The effective access time can be defined as

$$\boxed{\textit{Effective access time} = P\,(t + m) + (1 - p)\,(t + k.m + m)}$$

Where, p is the TLB hit rate, t is the time taken to access TLB, m is the time taken to access main memory. K indicates the single level paging has been implemented.

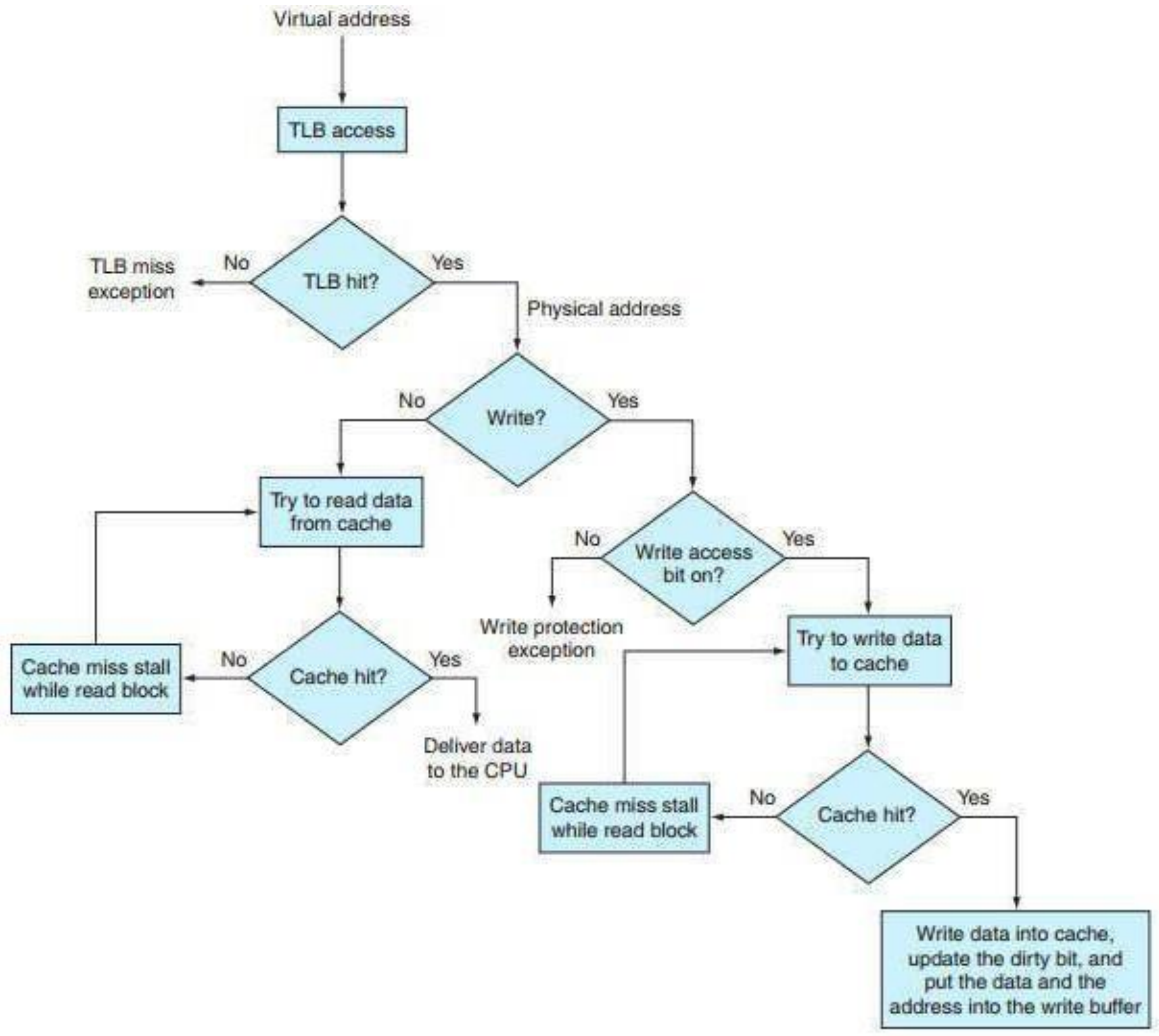

Virtual address
TLB access
TLB hit?
No
TLB miss exception
Yes
Physical address
Write?
No
Yes
Try to read data from cache
Write access bit on?
No
Yes
Write protection exception
Try to write data to cache
Cache hit?
No
Cache miss stall while read block
Yes
Deliver data to the CPU
Cache hit?
No
Cache miss stall while read block
Yes
Write data into cache, update the dirty bit, and put the data and the address into the write buffer

Fig 4.15: Cache access levels

4.3.5 Protection in Virtual memory

- Virtual memory allows sharing of main memory by multiple processes. So protection mechanisms, while providing memory protection.
- The protection mechanism must ensure one process cannot write into the address space of another user process or into the operating system.
- Memory protection can be done at two levels: hardware and software levels.

Hardware Level:

Memory protection at hardware level is done in three methods:

- The machine should support two modes: supervisor mode and user mode. This indicates whether the current running process is a user or supervisory process. The processes running in supervisor or kernel mode is an operating system process.
- Include user / supervisor bit in TLB to indicate whether the process is in user or supervisor mode. This is an access control mechanism imposed on the user process only to read from the TLB and not write to it.
- The processors can switch between user and supervisor mode. The switching from user to system mode is done through system calls that transfers control to a dedicated location in supervisor code space.

System call is a special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process.

4.4 PARALLEL BUS ARCHITECTURES

Single bus architectures connect multiple processors with their own cache memory using shared bus. This is a simple architecture but it suffers from latency and bandwidth issues. This naturally led to deploying parallel or multiple bus architectures. Multiple bus multiprocessor systems use several parallel buses to interconnect multiple processors with multiple memory modules. The following are the connection schemes in multi bus architectures:

1. Multiple-bus with full bus–memory connection (MBFBMC)

This has all memory modules connected to all buses. The multiple-bus with single bus memory connection has each memory module connected to a specific bus. For N processors with M memory modules and B buses, the number of connections requires are: B(N+M) and the load on each bus will ne N+M.

2. **Multiple bus with partial bus–memory connection (MBPBMC)**

   The multiple-bus with partial bus–memory connection, has each memory module connected to a subset of buses.

3. **Multiple bus with class-based memory connection (MBCBMC)**

   The multiple-bus with class-based memory connection (MBCBMC), has memory modules grouped into classes whereby each class is connected to a specific subset of buses. A class is just an arbitrary collection of memory modules.

4. **Multiple bus with single bus memory connection (MBSBMC)**

   Here, only single bus will be connected to single memory, but the processor can access all the buses. The numbers of connections:

$$BN + \sum_{j=1}^{k} M_j(j + B - k)$$

   And load on each bus is given by

$$N + \sum_{j=\max(i+k-B,1)}^{k} M_j,\ 1 \le i \le B$$

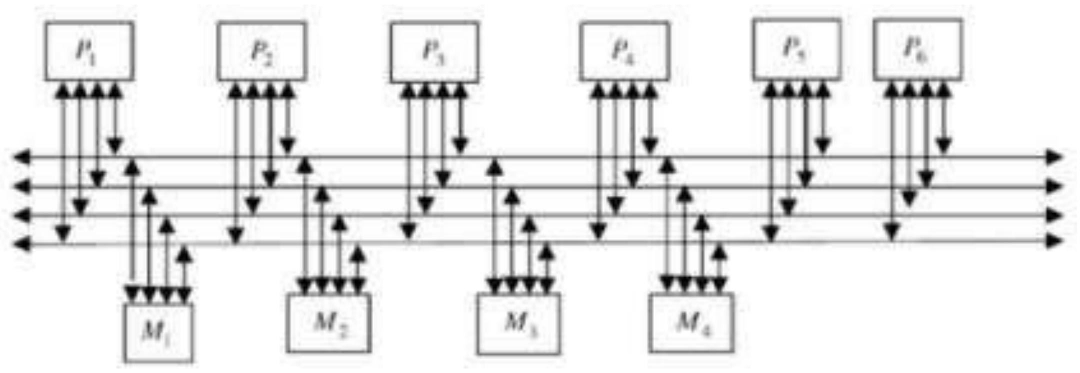


Fig 4.16 a) Multiple-bus with full bus–memory connection (MBFBMC)

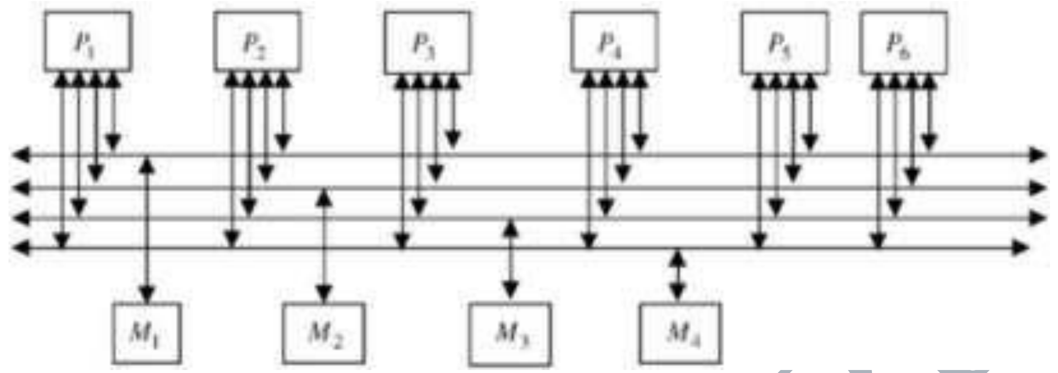


Fig 4.16 b) Multiple bus with single bus memory connection (MBSBMC)

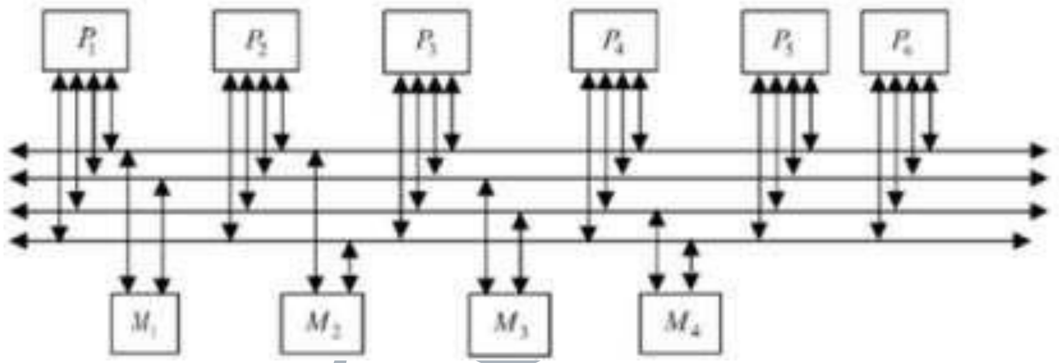


Fig 4.16 c) Multiple bus with partial bus–memory connection (MBPBMC)

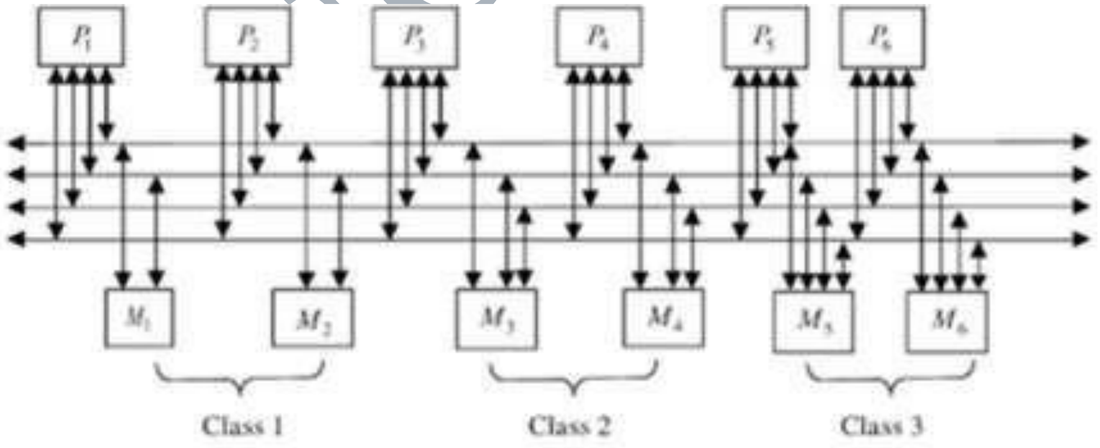


Fig 4.16 d) Multiple bus with class-based memory connection (MBCBMC)

4.4.1 Bus Synchronization:

- In a single bus multiprocessor system, bus arbitration is required in order to resolve the bus contention that takes place when more than one processor competes to access the bus.

- A bus can be classified as synchronous or asynchronous. The time for any transaction over a synchronous bus is known in advance. Asynchronous bus depends on the availability of data and the readiness of devices to initiate bus transactions.

- The processors that want to use the bus submit their requests to bus arbitration logic. The latter decides, using a certain priority scheme, which processor will be granted access to the bus during a certain time interval (bus master).

- The process of passing bus mastership from one processor to another is called **handshaking,** which requires the use of two control signals: bus request and bus grant.

- Bus request indicates that a given processor is requesting mastership of the bus.
- Bus grant: indicates that bus mastership is granted.
- Bus busy: is usually used to indicate whether or not the bus is currently being used.
- In deciding which processor gains control of the bus, the bus arbitration logic uses a predefined priority scheme.
- Among the priority schemes used are random priority, simple rotating priority, equal priority, and least recently used (LRU) priority.
- After each arbitration cycle, in simple rotating priority, all priority levels are reduced one place, with the lowest priority processor taking the highest priority. In equal priority, when two or more requests are made, there is equal chance of any one request being processed.
- In the LRU algorithm, the highest priority is given to the processor that has not used the bus for the longest time.

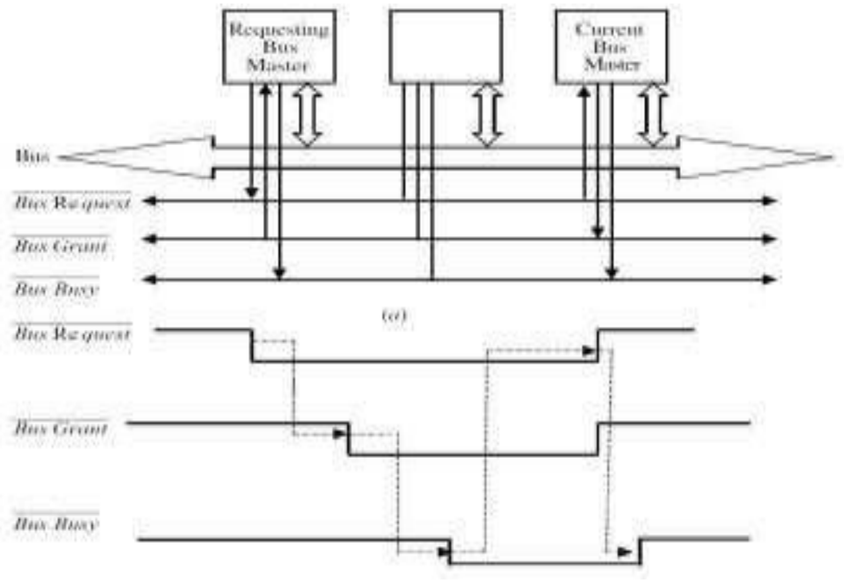


Fig 4.17: Bus synchronization

4.5 INTERNAL COMMUNICATION METHODOLOGIES

CPU of the computer system communicates with the memory and the I/O devices in order to transfer data between them. The method of communication of the CPU with memory and I/O devices is different. The CPU may communicate with the memory either directly or through the Cache memory. However, the communication between the CPU and I/O devices is usually implemented with the help of interface. There are three types of internal communications:

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access (DMA)

4.5.1 Programmed I/O

- Programmed I/O is implicated to data transfers that are initiated by a CPU, under driver software control to access Registers or Memory on a device.
- With programmed I/O, data are exchanged between the processor and the I/O module.

- The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data.
- When the processor issues a command to the I/O module, it must wait until the I/O operation is complete.
- If the processor is faster than the I/O module, this is wasteful of processor time. With interrupt-driven I/O, the processor issues I/O command, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work.
- With both programmed and interrupt I/O, the processor is responsible for extracting data from main memory for output and storing data in main memory for input.
- The alternative is known as direct memory access. In this mode, the I/O module and main memory exchange data directly, without processor involvement.
- With programmed I/O, the I/O module will perform the requested action and then set the appropriate bits in the I/O status register.
- The I/O module takes no further action to alert the processor.
- When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. In particular, it does not interrupt the processor.
- It is the responsibility of the processor periodically to check the status of the I/O module. Then if the device is ready for the transfer (read/write).
- The processor transfers the data to or from the I/O device as required. As the CPU is faster than the I/O module, the problem with programmed I/O is that the CPU has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data.
- The CPU, while waiting, must repeatedly check the status of the I/O module, and this process is known as **Polling.**
- The level of the performance of the entire system is severely degraded.

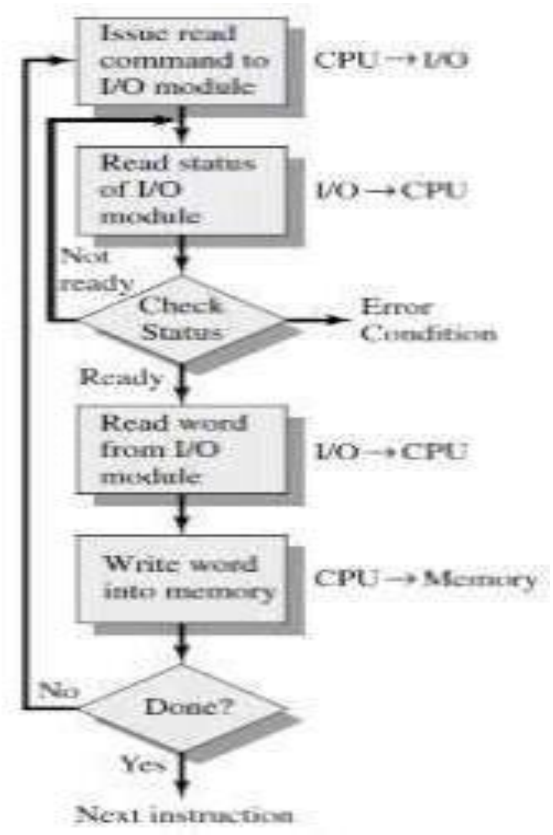


Fig 4.18: Workflow in programmed I/O

4.5.2 Interrupt Driven I/O

- The CPU issues commands to the I/O module then proceeds with its normal work until interrupted by I/O device on completion of its work.
- For input, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports, memory mapping.
- For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.
- Although Interrupt relieves the CPU of having to wait for the devices, but it is still inefficient in data transfer of large amount because the CPU has to transfer the data word by word between I/O module and memory.
- Below are the basic operations of Interrupt:

1. CPU issues read command
2. I/O module gets data from peripheral whilst CPU does other work
3. I/O module interrupts CPU
4. CPU requests data
5. I/O module transfers data

4.5.3 Direct Memory Access (DMA)

- Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement.
- DMA module controls exchange of data between main memory and the I/O device.
- Because of DMA device can transfer data directly to and from memory, rather than using the CPU as an intermediary, and can thus relieve congestion on the bus.
- CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

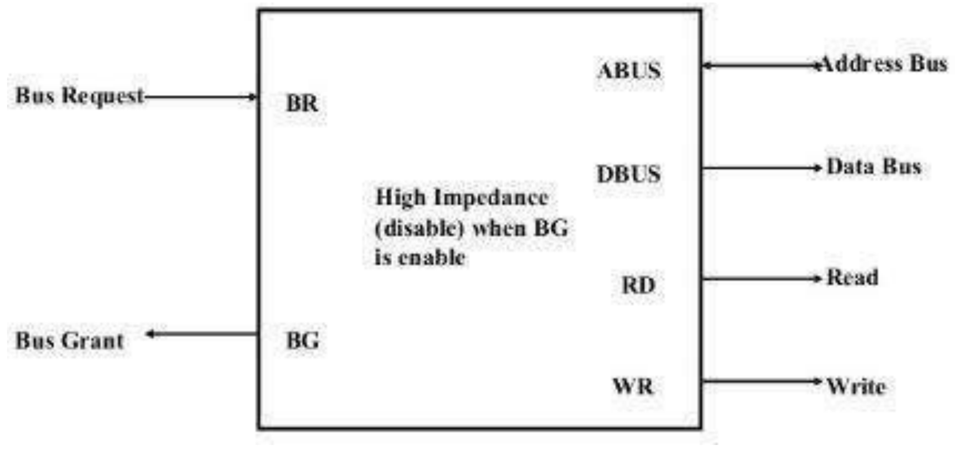


Fig 4.19: CPU bus signals for DMA transfer

- The CPU programs the DMA controller by setting its registers so it knows what to transfer where.
- It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum.

 When valid data are in the disk controller's buffer, DMA can begin. The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller.

 This read request looks like any other read request, and the disk controller does not know whether it came from the CPU or from a DMA controller.

 The memory address to write to is on the bus address lines, so when the disk controller fetches the next word from its internal buffer, it knows where to write it.

 The write to memory is another standard bus cycle.

 When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus.

 The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0.

 At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete.

 When the operating system starts up, it does not have to copy the disk block to memory; it is already there.

 The DMA controller requests the disk controller to transfer data from the disk controller's buffer to the main memory. In the first step, the CPU issues a command to the disk controller telling it to read data from the disk into its internal buffer.

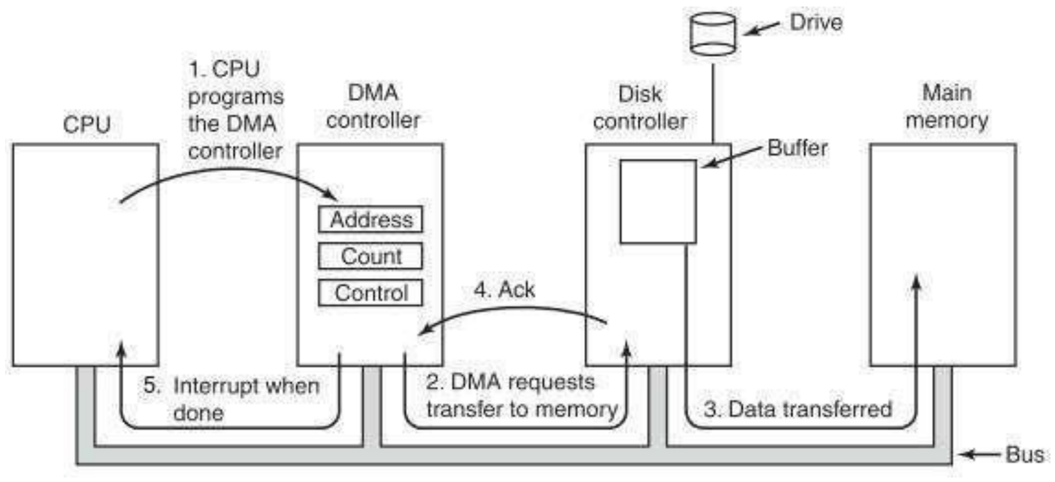


Fig 4.20: Operations in DMA

4.6 SERIAL BUS ARCHITECTURES

The peripheral devices and external buffer that operate at relatively low frequencies communicate with the processor using serial bus. There are two popular serial buses: Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C).

4.6.1 Serial Peripheral Interface (SPI)

Serial Peripheral Interface (SPI) is an interface bus designed by Motorola to send data between microcontrollers and small peripherals such as shift registers, sensors, and SD cards. It uses separate clock and data lines, along with a select line to choose the device.

- A standard SPI connection involves a master connected to slaves using the serial clock (SCK), Master Out Slave In (MOSI), Master In Slave Out (MISO), and Slave Select (SS) lines.
- The SCK, MOSI, and MISO signals can be shared by slaves while each slave has a unique SS line.
- The SPI interface defines no protocol for data exchange, limiting overhead and allowing for high speed data streaming.
- Clock polarity ⁅CPOL⁆ and clock phase ⁅CP(A⁆ can be specified as ¦ど¦ or ¦な¦ to form four unique modes to provide flexibility in communication between master and slave.
-)f CPOL and CP(A are both ¦ど¦ ⁅defined as Mode ど⁆ data is sampled at the leading rising edge of the clock. Mode 0 is by far the most common mode for SPI bus slave communication.
-)f CPOL is ¦な¦ and CP(A is ¦ど¦ ⁅Mode に⁆, data is sampled at the leading falling edge of the clock.
- Likewise, CPOL = ¦ど¦ and CP(A = ¦な¦ ⁅Mode な⁆ results in data sampled at on the trailing falling edge and CPOL = ¦な¦ with CP(A = ¦な¦ ⁅Mode ぬ⁆ results in data sampled on the trailing rising edge.

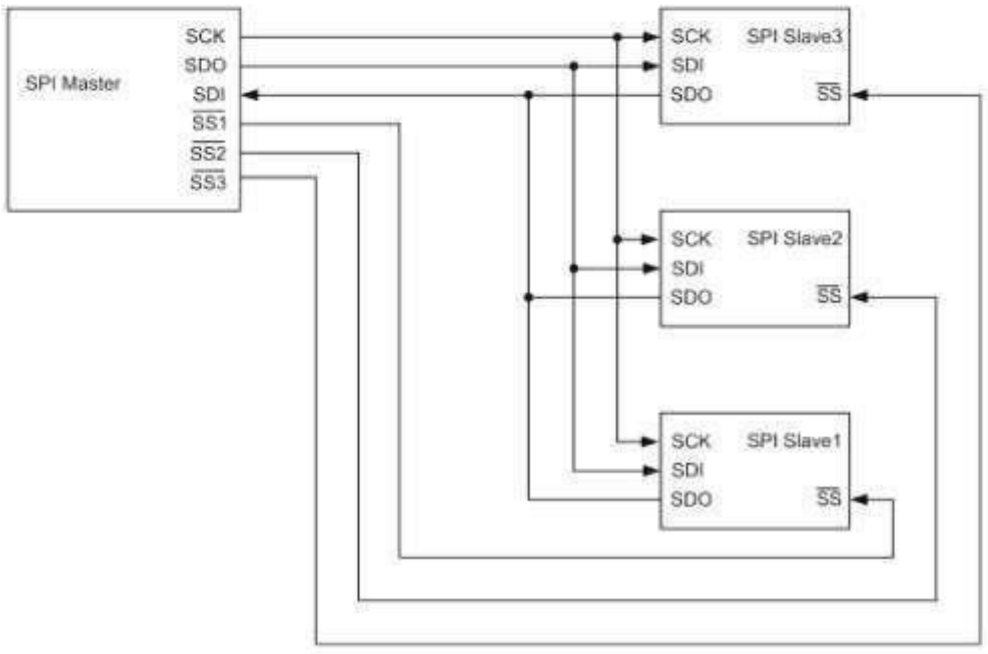


Fig 4.21: SPI master with three slaves

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Fig 4.22: Modes in SPI

- In addition to the standard 4-wire configuration, the SPI interface has been extended to include a variety of IO standards including 3-wire for reduced pin count and dual or quad I/O for higher throughput.
- In 3-wire mode, MOSI and MISO lines are combined to a single bidirectional data line.
- Transactions are half-duplex to allow for bidirectional communication. Reducing the number of data lines and operating in half-duplex mode also decreases maximum possible throughput; many 3-wire devices have low performance requirements and are instead designed with low pin count in mind.

- Multi I/O variants such as dual I/O and quad I/O add additional data lines to the standard for increased throughput.
- Components that utilize multi I/O modes can rival the read speed of parallel devices while still offering reduced pin counts. This performance increase enables random access and direct program execution from flash memory (execute-in-place).

4.6.2 Inter-Integrated Circuit (I^2C)

An inter-integrated circuit (Inter-IC or I2C) is a multi-master serial bus that connects low-speed peripherals to a motherboard, mobile phone, embedded system or other electronic devices.

- Philips Semiconductor created I^2C with an intention of communication between chips reside on the same Printed Circuit Board (PCB).
- It is a multi-master, multi-slave protocol.
- It is designed to lessen costs by streamlining massive wiring systems with an easier interface for connecting a central processing unit (CPU) to peripheral chips in a television.
- It had a battery-controlled interface but later utilized an internal bus system.
- It is built on two lines
- SDA (Serial Data) – The line for the master and slave to send and receive data
- SCL (Serial Clock) – The line that carries the clock signal.
- Devices on an I2C bus are always a master or a slave. Master is the device which always initiates a communication and drives the clock line (SCL). Usually a microcontroller or microprocessor acts a master which needs to read data from or write data to slave peripherals.
- Slave devices are always responds to master and won't initiate any communication by itself. Devices like EEPROM, LCDs, RTCs acts as a slave device. Each slave device will have a unique address such that master can request data from or write data to it.
- The master device uses either a 7-bit or 10-bit address to specify the slave device as its partner of data communication and it supports bi-directional data transfer.

Working of I²C

- The I2C, data is transferred in messages, which are broken up into frames of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted.
- The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame.
- The following are the bits in data frames:

1. **Start Condition:** The SDA line switches from a high voltage level to a low voltage level before the SCL line switches from high to low.
2. **Stop Condition:** The SDA line switches from a low voltage level to a high voltage level after the SCL line switches from low to high.
3. **Address Frame:** A 7 or 10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.
4. **Read/Write Bit**: A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).
5. **ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

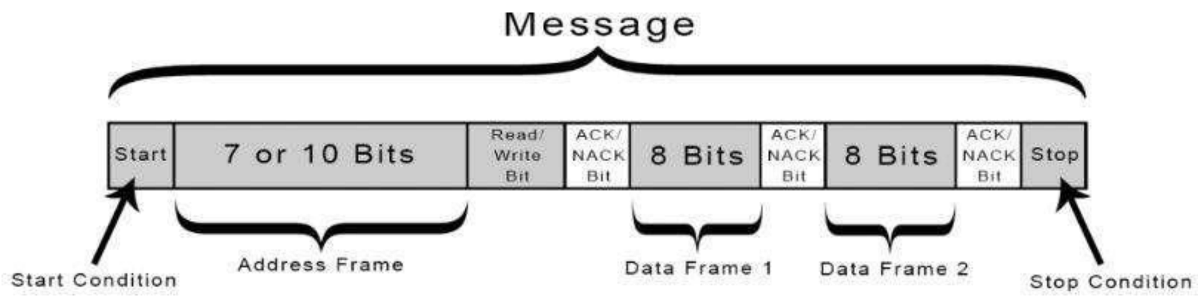


Fig 4.23: I²C Message Format

Addressing:

- I²C doesn't have slave select lines like SP), so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by addressing. The address frame is always the first frame after the start bit in a new message.

- The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address.
- If the address matches, it sends a low voltage ACK bit back to the master. If the address doesn't match, the slave does nothing and the SDA line remains high.

Read/Write Bit

- The address frame includes a single bit at the end that informs the slave whether the master wants to write data to it or receive data from it. If the master wants to send data to the slave, the read/write bit is a low voltage level. If the master is requesting data from the slave, the bit is a high voltage level.

Data Frame

- After the master detects the ACK bit from the slave, the first data frame is ready to be sent.
- The data frame is always 8 bits long, and sent with the most significant bit first.
- Each data frame is immediately followed by an ACK/NACK bit to verify that the frame has been received successfully.
- The ACK bit must be received by either the master or the slave (depending on who is sending the data) before the next data frame can be sent.
- After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission.
- The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

Steps in Data transmission

1. The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level before switching the SCL line from high to low.
2. The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit.

3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave's own address, the slave leaves the SDA line high.
4. The master sends or receives the data frame.
5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame.
6. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high.

Advantages

- It uses two wires.
- This supports multiple masters and multiple slaves.
- ACK/NACK bit gives confirmation that each frame is transferred successfully.
- Well known and widely used protocol

Disadvantages

- Slower data transfer rate than SPI.
- The size of the data frame is limited to 8 bits
- More complicated hardware needed to implement than SPI.

4.7 MASS STORAGE

Mass storage refers to various techniques and devices for storing large amounts of data. Mass storage is distinct from memory, which refers to temporary storage areas within the computer. Unlike main memory, mass storage devices retain data even when the computer is turned off.

The mass storage medium includes:

- solid-state drives (SSD)
- hard drives
- external hard drives
- optical drives

- tape drives
- RAID storage
- USB storage
- flash memory cards

Solid State Devices

- Solid-state devices are electronic devices in which electricity flows through solid semiconductor crystals like silicon, gallium arsenide, and germanium rather than through vacuum tubes.
- It do not involve any moving parts or magnetic materials.
- RAM is a solid state device that consists of microchips that store data on non-moving components, providing for fast retrieval of that data.
- Transistors are the most important solid state devices. The transistors contain two p– n junctions, have three contacts or terminals.
- They require the action of perpendicular electrical fields, their behavior is more difficult to understand than that of diodes.
- The different types of transistors are: bipolar junction transistor (BJT) where the current is amplified, while in the field effect transistor (FET) a voltage controls a current.
- In a solid-state component, the current is confined to solid elements and compounds engineered specifically to switch and amplify it.
- Current flows in two forms: as negatively charged electrons, and as positively charged electron deficiencies called holes.
- In some semiconductors, the current consists mostly of electrons; in other semiconductors, it consists mostly of holes. Both the electron and the hole are called charge carriers.

Hard Drives

- A hard disk drive is a non-volatile memory hardware device that permanently stores and retrieves data on a computer.
- A hard drive is a secondary storage device that consists of one or more platters to which data is written using a magnetic head, all inside of an air-sealed casing.

- Internal hard disks reside in a drive bay, connect to the motherboard using an ATA, SCSI, or SATA cable, and are powered by a connection to the power supply unit.

External Hard Drives

- An external hard drive is a portable storage device that can be attached to a computer through a USB or FireWire connection, or wirelessly.
- External hard drives typically have high storage capacities and are often used to back up computers or serve as a network drive.

Optical Drives

- An Optical Drive refers to a computer system that allows users to use DVDs, CDs and Blu-ray optical drives.
- The drive contains some lenses that project electromagnetic waves that are responsible for reading and writing data on optical discs.
- An optical disk drive uses a laser to read and write data. A laser in this context means an electromagnetic wave with a very specific wavelength within or near the visible light spectrum.
- An optical drive that works with all types of discs will have two separate lenses: one for CD/DVD and one for Blu-ray.
- An optical drive has a rotational mechanism to spin the disc. Optical drives were originally designed to work at a constant linear velocity (CLV) (i.e.) the disc spins at varying speeds depending on where the laser beam is reading, so the spiral groove of the disc passes by the laser at a constant speed.
- An optical drive also needs a loading mechanism: A **tray-loading mechanism,** where the disc is placed onto a motorized tray, which moves in and out of the computer case and **slot-loading mechanism,** where the disc is slid into a slot and motorized rollers are used to move the disc in and out.

Tape disks

- A tape drive is a device that stores computer data on magnetic tape, especially for backup and archiving purposes.

- Tape drives work either by using a traditional helical scan where the recording and playback heads touch the tape, or linear tape technology, where the heads never actually touch the tape.
- Drives can be rewinding, where the device issues a rewind command at the end of a session, or non-rewinding.
- Rewinding devices are most commonly used when a tape is to be unmounted at the end of a session after batch processing of large amounts of data.
- Non-rewinding devices are useful for incremental backups and other applications where new files are added to the end of the previous session's files.
- The different types of tapes are audio, video and data storage tape.

Redundant Array of Inexpensive Disks (RAID) Storage

- RAID is a way of storing the same data in different places on multiple hard disks to protect data in the case of a drive failure.
- RAID works by placing data on multiple disks and allowing input/output (I/O) operations to overlap in a balanced way, improving performance. Because the use of multiple disks increases the mean time between failures (MTBF), storing data redundantly also increases fault tolerance.
- A RAID controller can be used as a level of abstraction between the OS and the physical disks, presenting groups of disks as logical units. Using a RAID controller can improve performance and help protect data in case of a crash.
- Levels in RAID:

1. **RAID 0 (Disk striping):**

   RAID 0 splits data across any number of disks allowing higher data throughput. An individual file is read from multiple disks giving it access to the speed and capacity of all of them. This RAID level is often referred to as striping and has the benefit of increased performance.

2. **RAID 1 (Disk Mirroring):**

   RAID 1 writes and reads identical data to pairs of drives. This process is often called data mirroring and it's a primary function is to provide redundancy. If any of the disks in the array fails, the system can still access data from the remaining disk(s).

3. **RAID 5 (Striping with parity):**

   RAID 5 stripes data blocks across multiple disks like RAID 0, however, it also stores parity information (Small amount of data that can accurately describe larger amounts of data) which is used to recover the data in case of disk failure. This level offers both speed (data is accessed from multiple disks) and redundancy as parity data is stored across all of the disks.

4. **RAID 6 (Striping with double parity):**

   Raid 6 is similar to RAID 5, however, it provides increased reliability as it stores an extra parity block. That effectively means that it is possible for two drives to fail at once without breaking the array.

5. **RAID 10 (Striping + Mirroring):**

   RAID 10 combines the mirroring of RAID 1 with the striping of RAID 0. Or in other words, it combines the redundancy of RAID 1 with the increased performance of RAID 0. It is best suitable for environments where both high performance and security is required.

Universal Serial Bus (USB) Devices

- USB is a system for connecting a wide range of peripherals to a computer, including pointing devices, displays, and data storage and communications products.
- The Universal Serial Bus is a network of attachments connected to the host computer.
- These attachments come in two types known as **Functions and Hubs.**
- Functions are the peripherals such as mice, printers, etc.
- Hubs basically act like a double adapter does on a power-point, converting one socket, called a port, into multiple ports.
- Hubs and functions are collectively called devices.
- When a device is attached to the USB system, it gets assigned a number called its address. The address is uniquely used by that device while it is connected.
- Each device also contains a number of endpoints, which are a collection of sources and destinations for communications between the host and the device.
- The combination of the address, endpoint number and direction are what is used by the host and software to determine along which pipe data is travelling.

Flash Drives

- A flash drive stores data using flash memory. Flash memory uses an electrically erasable programmable read-only (EEPROM) format to store and retrieve data.
- Flash drives are non-volatile, which means they do not need a battery backup.
- Most computers come equipped with USB ports, which detect inserted flash drives and install the necessary drivers to make the data retrievable.
- Computer users can store and retrieve data once the operating system has detected a connection to the USB port.
- Flash drives have a USB mass storage device classification, which means they do not require additional drivers.
- The computer's operating system recognizes a block-structured logical unit, which means it can use any file system or block addressing system to read the information on the flash drive.
- A flash drive enters emulation mode, or acts a hard drive, once it has connected to the USB port. This makes it easier to transfer data between the flash drive and the computer.
- Flash memory is known as a solid state storage device, meaning there are no moving parts — everything is electronic instead of mechanical.

4.9 INPUT AND OUTPUT DEVICES

The common input and output devices are discussed here:

4.9.1 Input Devices

Keyboard

- A keyboard has its own processor and circuitry that carries information to and from that processor.
- A large part of this circuitry makes up the **key matrix which is arranged in rows and columns.**
- The key matrix is a grid of circuits underneath the keys.
- In all keyboards each circuit is broken at a point below each key. When a key is presses, it presses a switch, completing the circuit and allowing a tiny amount of current to flow through.

- The mechanical action of the switch causes some vibration, called **bounce,** which the processor filters out.
- If the key is pressed and held continuously, the processor recognizes it as the equivalent of pressing a key repeatedly.
- Another type of keyboard has three layers: top plasticized layer with key positions marked on the top surface and conducting traces on another side; middle layer made of rubber with hole for key positions; bottom metallic layer with raised bumps for key positions.
- When a key is pressed the trace underneath the top layer comes in contact with the bump in the last layer, thus completing an electrical circuit. The current flow is sensed by the microcontroller.

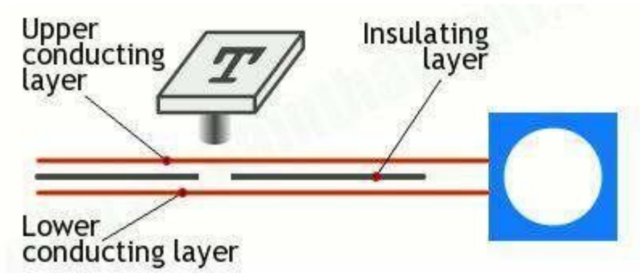


Fig 4.24: Layers in keyboard

Mouse

- A computer mouse is a hand-held pointing device that detects two-dimensional motion relative to a surface.
- This motion is typically translated into the motion of a pointer on a display, which allows a smooth control of the graphical user interface.
- There are two main kinds of mice: rolling rubber ball mouse or optical mouse.
- As the mouse is moved, the ball rolls under its own weight and pushes against two plastic rollers linked to thin wheels.
- One of the wheels detects movements in an up-and-down direction (y-axis) and the other detects side-to-side movements (x-axis).
- If the mouse is moved straight up, only the y-axis wheel turns. If the mouse is moved to the right, only the x-axis wheel turns.

- The optical mouse shines a bright light down onto the desk from an LED mounted on the bottom of the mouse.
- The light bounces straight back up off the desk into a photocell also mounted under the mouse, a short distance from the LED.
- The photocell has a lens in front of it that magnifies the reflected light, so the mouse can respond more precisely to your hand movements.
- As the mouse is pushed, the pattern of reflected light changes, and the chip inside the mouse uses this to figure out the motion.

Trackball, Joystick and Touch pad

- A trackball can also be used as an alternative to a mouse. This device also has buttons similar to those on a mouse.
- It holds a large moving ball on the top. The body of the trackball is not moved. The ball is rolled with fingers. The position of the cursor on the screen is controlled by rotating the ball.
- The main benefit of the trackball over a mouse is that it takes less space to move. The trackball is often included in laptop computers. The standard desktop computer also uses a trackball operated as a separate input device.
- A touchpad is a small, plane surface over which the user moves his finger. The user controls the movement of the cursor on the screen by moving his fingers on the touchpad. It is also known as a track pad.
- A touchpad also has one or more buttons near it. These button work like mouse buttons. Touchpads are commonly used with notebook computers.
- A joystick consists of a base and a stick. The stick can be moved in several directions to shift an object anywhere on the computer screen.
- A joystick can perform a similar function to a mouse or trackball. It is often considered less comfortable and efficient. The most common use of a joystick is for playing computer games.

Scanners

- Scanners operate by shining light at the object or document being digitized and directing the reflected light onto a photosensitive element.
- In most scanners, the sensing medium is an electronic, light-sensing integrated circuit known as a charged coupled device (CCD).
- Light-sensitive photo sites arrayed along the CCD convert levels of brightness into electronic signals that are then processed into a digital image.
- A scanner consists of a flat transparent glass bed under which the CCD sensors, lamp, lenses, filters and also mirrors are fixed.
- The document has to be placed on the glass bed. There will also be a cover to close the scanner.
- The lamp brightens up the text to be scanned. Most scanners use a cold cathode fluorescent lamp (CCFL).
- A stepper motor under the scanner moves the scanner head from one end to the other. The movement will be slow and is controlled by a belt.
- The scanner head consists of the mirrors, lens, CCD sensors and also the filter. The scan head moves parallel to the glass bed and that too in a constant path.
- As the scan head moves under the glass bed, the light from the lamp hits the document and is reflected back with the help of mirrors angled to one another.
- According to the design of the device there may be either 2-way mirrors or 3-way mirrors.
- The mirrors will be angled in such a way that the reflected image will be hitting a smaller surface.
- In the end, the image will reach a lens which passes it through a filter and causes the image to be focused on CCD sensors.
- The CCD sensors convert the light to electrical signals according to its intensity.
- The electrical signals will be converted into image format inside a computer.

4.9.2 Output Devices

Video Displays

- The CRT monitors were fundamental output display device.
- The CRT or cathode ray tube, is the picture tube of a monitor.
- The back of the tube has a negatively charged cathode.
- The electron gun shoots electrons down the tube and onto a charged screen.
- The screen is coated with a pattern of dots using phosphor that glow when struck by the electron stream.
- The image on the monitor screen is usually made up from at least tens of thousands of such tiny dots glowing on command from the computer.
- The closer together the pixels are, the sharper the image on screen.
- The distance between pixels on a computer monitor screen is called its dot pitch and is measured in millimeters. Most monitors have a dot pitch of 0.28 mm or less.
- There are two electromagnets around the collar of the tube which deflect the electron beam.
- The beam scans across the top of the monitor from left to right, is then blanked and moved back to the left-hand side slightly below the previous trace (on the next scan line), scans across the second line and so on until the bottom right of the screen is reached.
- The beam is again blanked, and moved back to the top left to start again.
- This process draws a complete picture, typically 50 to 100 times a second.
- The number of times in one second that the electron gun redraws the entire image is called the refresh rate and is measured in hertz (cycles per second).
- It is common, particularly in lower priced equipment, for all the odd-numbered lines of an image to be traced, and then all the even-numbered lines; the circuitry of such an interlaced display need to be have only half the speed of a non-interlaced display.
- An interlaced display, particularly at a relatively low refresh rate, can appear to some observers to flicker, and may cause eye strain and nausea.

- The intensity or strength of the electron beam is controlled by setting the voltage levels.
- The number of electrons that hits the screen determines the light emitted by the screen. When the voltage is varied in the electron gun, the brightness of the display also varies.
- The focusing hardware focuses the beam at all positions on the screen.
- The deflection of electron beam is controlled by electric or magnetic fields.
- Two pairs of coils mounted on the CRT to produce the necessary defection.
- The coils are placed in such a way that, the magnetic field produced by them results in traverse deflection force that is perpendicular to the magnetic field and electron beam.

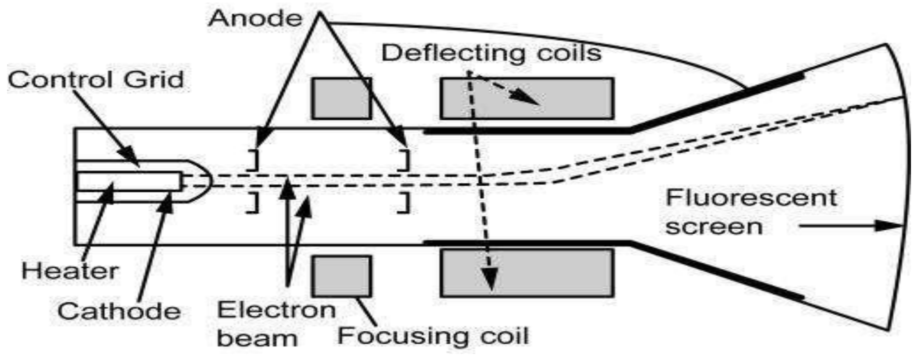


Fig 4.25: CRT Monitor

- An LED screen is an LCD screen, but instead of having a normal CCFL backlight, it uses light-emitting diodes (LEDs) as a source of light behind the screen.
- An LED is more energy efficient and a lot smaller than a CCFL, enabling a thinner television screen.

Printers

- A printer is an electromechanical device which converts the text and graphical documents from electronic form to the physical form.
- They are the external peripheral devices which are connected with the computers or laptops through a cable or wirelessly to receive input data and print them on the papers.
- Quality of printers is identified by its features like color quality, speed of printing, resolution etc. Modern printers come with multipurpose functions i.e. they are combination of printer, scanner, photocopier, fax, etc.
- Broadly printers are categorized as impact and non impact printers.

Daisy Wheel Printers

- Daisy wheel printers print only characters and symbols and cannot print graphics. They are generally slow with a printing speed of about 10 to 75 characters per second.
- A circular printing element is the heart of these printers that contains all text, numeric characters and symbols mould on each petal on the circumference of the circle.
- The printing element rotates rapidly with the help of a servo motor and pauses to allow the printing hammer to strike the character against the paper.

Dot Matrix Printers

- It is a popular computer printer that prints text and graphics on the paper by using tiny dots to form the desired shapes.
- It uses an array of metal pins known as print head to strike an inked printer ribbon and produce dots on the paper.
- These combinations of dots form the desired shape on the paper.
- The key component in the dot matrix printer is the 'print head' which is about one inch long and contains a number of tiny pins aligned in a column varying from 9 to 24.
- The print head is driven by several hammers which force each pin to make contact with the paper at the certain time. These hammers are pulled by small electromagnet which is energized at a specific time depending on the character to be printed.
- The timings of the signals sent to the solenoids are programmed in the printer for each character.

Inkjet printers

- Inkjet printers are most popular printers for home and small scale offices as they have a reasonable cost and a good quality of printing as well.
- An inkjet printer is made of the following parts:

i) Print head – It is the heart of the printer which holds a series a nozzles which sprays the ink drops over the paper.

ii) Ink cartridge – It is the part that contains the ink for printing. Generally monochrome (black & white) printers contain a black colored ink cartridges and a color printer contains two cartridges – one with black ink and other with primary colors (cyan, magenta and yellow).

iii) Stepper motor – It is housed in the printer to move the printer head and ink cartridges back and forth across the paper.

iv) Stabilizer bar – A stabilizer bar is used in printer to ensure the movement of print head is précised and controlled over the paper.

v) Belt – A belt is used to attach the print head with the stepper motor.

vi) Paper Tray – It is the place where papers are placed to be printed.

vii) Rollers – Printers have a set of rollers that helps to pull paper from the tray for printing purpose.

viii) Paper tray stepper motor- another stepper motor is used to rotate the rollers in order to pull the paper in the printer.

ix) Control Circuitry – The control circuit takes the input from the computer and by decoding the input controls all mechanical operation of the printer.

Laser Printers

- Laser printers are the most popular printers that are mainly used for large scale qualitative printing.
- They are among the most popularly used fastest printers available in the market.
- A laser printer uses a slight different approach for printing. It does not use ink like inkjet printers, instead it uses a very fine powder known as Toner.
- The control circuitry is the part of the printer that talks with the computer and receives the printing data.
- A Raster Image Processor (RIP) converts the text and images in to a virtual matrix of dots.
- The photo conducting drum which is the key component of the laser printer has a special coating which receives the positive and negative charge from a charging roller.
- A rapidly switching laser beam scans the charged drum line by line. When the beam flashes on, it reverses the charge of tiny spots on the drum, respecting to the dots that are to be printed black.

- As soon the laser scans a line, a stepper motor moves the drum in order to scan the next line by the laser.
- A developer roller plays the vital role to paste the tonner on the paper. It is coated with charged tonner particles.
- As the drum touches the developer roller, the charged tonner particles cling to the discharged areas of the drum, reproducing your images and text reversely.
- Meanwhile a paper is drawn from the paper tray with help of a belt. As the paper passes through a charging wire it applies a charge on it opposite to the toners charge.
- When the paper meets the drum, due to the opposite charge between the paper and toner particles, the toner particles are transferred to the paper.
- A cleaning blade then cleans the drum and the whole process runs smoothly continuously.
- Finally paper passes through the fuser which is a heat and presser roller, melts the toner and fixes on the paper perfectly.

UNIT - V
ADVANCED COMPUTERARCHITECTURE

5.1 PARALLEL PROCESSING ARCHITECTURES

Parallel computing architectures breaks the job into discrete parts that can be executed concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both. Parallel systems are more difficult to program than computers with a single processor because the architecture of parallel computers varies accordingly and the processes of multiple CPUs must be coordinated and synchronized. The crux of parallel processing are the CPUs.

Flynn's taxonomy is a specific classification of parallel computer architectures that are based on the number of concurrent instruction (single or multiple) and data streams (single or multiple) available in the architecture.

Parallelism in computer architecture is explained used Flynn's taxonomy. This classification is based on the number of instruction and data streams used in the architecture. The machine structure is explained using streams which are sequence of items. The four categories in Flynn's taxonomy based on the number of instruction streams and data streams are the following:

- (SISD) single instruction, single data
- (MISD) multiple instruction, single data
- (SIMD) single instruction, multiple data
- (MIMD) multiple instruction, multiple data

SISD (Single Instruction, Single Data stream)

- Single Instruction, Single Data (SISD) refers to an Instruction Set Architecture in which a single processor (one CPU) executes exactly one instruction stream at a time.
- It also fetches or stores one item of data at a time to operate on data stored in a single memory unit.

2.2.2 Most of the CPU design is based on the von Neumann architecture and the follow SISD.

2.2.3 The SISD model is a non-pipelined architecture with general-purpose registers, Program Counter (PC), the Instruction Register (IR), Memory Address Registers (MAR) and Memory Data Registers (MDR).

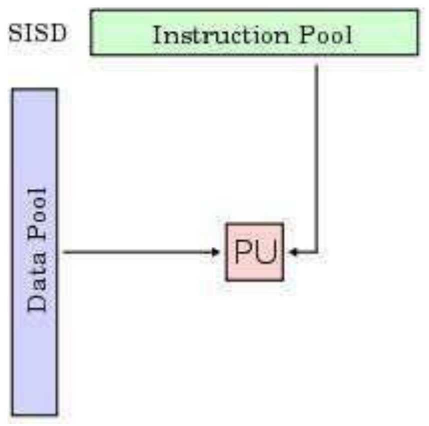


Fig 5.1: Single Instruction, Single Data Stream

SIMD (Single Instruction, Multiple Data streams)

- Single Instruction, Multiple Data (SIMD) is an Instruction Set Architecture that have a single control unit (CU) and more than one processing unit (PU) that operates like a von Neumann machine by executing a single instruction stream over PUs, handled through the CU.
- The CU generates the control signals for all of the PUs and by which executes the same operation on different data streams.
- The SIMD architecture is capable of achieving data level parallelism.

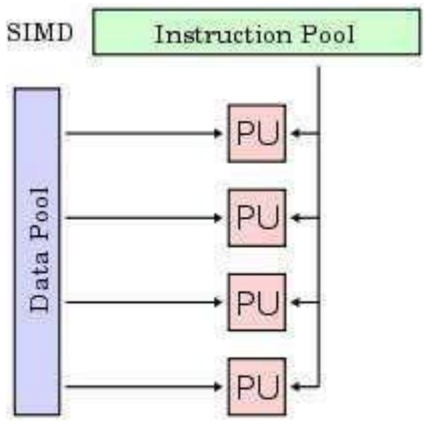


Fig 5.2: Single Instruction, Multiple Data streams MISD (Multiple Instruction, Single Data stream)

- Multiple Instruction, Single Data (MISD) is an Instruction Set Architecture for parallel computing where many functional units perform different operations by executing different instructions on the same data set.
- This type of architecture is common mainly in the fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors.

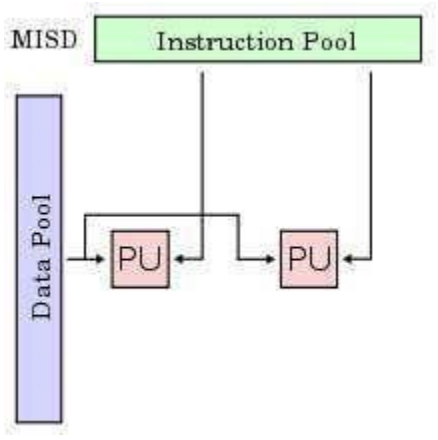


Fig 5.3: Multiple Instruction, Single Data stream

MIMD (Multiple Instruction, Multiple Data streams)

- Multiple Instruction stream, Multiple Data stream (MIMD) is an Instruction Set Architecture for parallel computing that is typical of the computers with multiprocessors.
- Using the MIMD, each processor in a multiprocessor system can execute asynchronously different set of the instructions independently on the different set of data units.
- The MIMD based computer systems can used the shared memory in a memory pool or work using distributed memory across heterogeneous network computers in a distributed environment.
- The MIMD architectures is primarily used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modelling, and communication switches etc.

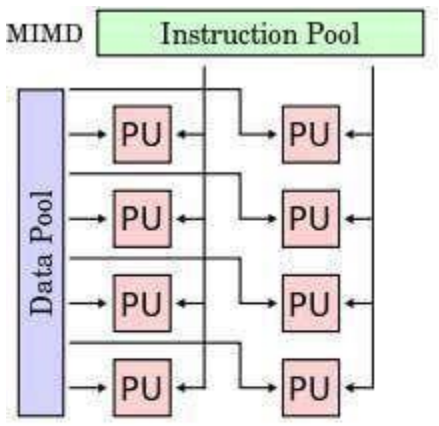


Fig 5.4:Multiple Instruction, Multiple Data streams

	Single	**Multiple**
Single	**SISD** Von Neumann Single	**MISD** May be pipelined computers
Multiple	**SIMD** Vector processors Fine grained Data Parallel computers	**MIMD** Multi computers Multiprocessors

Fig 5.5: Comparison of Flynn's taxonomy

5.1.1 Challenges in Parallelism

The following are the design challenges in parallelism:

- Available parallelism.
- Load balance: Some processors work while others wait due to insufficient parallelism or unequal size tasks.
- Extra work.
- Managing parallelism
- Redundant computation
- Communication

4. HARDWARE MULTITHREADING

Multithreading enables the processing of multiple threads at one time, rather than multiple processes. Since threads are smaller, more basic instructions than processes, multithreading may occur within processes. Threads are instruction stream with state (registers and memory). The register state is also called thread context. Threads could be part of the same process or from different programs. Threads in the same program share the same address space and hence consume fewer resources.

The terms multithreading, multiprocessing and multitasking are used interchangeably. But each has its unique meaning:

- **Multitasking:** It is the process of executing multiple tasks simultaneously. In multitasking, when a new thread needs to be executed, old thread's context in hardware written back to memory and new thread's context loaded.
- **Multiprocessing:** It is using two or more CPUs within a single computer system.
- **Multithreading:** It is executing several parts of a program in parallel by dividing the specific operations within a single application into individual threads.

Granularity: The threads are categorized based on the amount of work done by the thread. This is known as granularity. When the hardware executes from the hardware contexts determines the granularity of multithreading.

Hardware vs Software multithreading

Software Multithreading	**Hardware Multithreading**
Execution of concurrent threads is supported by OS.	Execution of concurrent threads is supported by CPU.
Large number of threads can be span.	Very limited number of threads can span.
Context switching is heavy. It involves more operations.	Light / immediate context switching with limited operations.

Hardware multithreading is having multiple threads contexts to span in same processor. This is supported by the CPU.

The following are the objectives of hardware multithreading:

- To tolerate latency of memory operations, dependent instructions, branch resolution by utilizing processing resources more efficiently. When one thread encounters a long-latency operation, the processor can execute a useful operation from another thread.
- To improve system throughput By exploiting thread-level parallelism by improving superscalar processor utilization
- To reduce context switch penalty

Advantages of hardware multithreading:

- Latency tolerance
- Better hardware utilization
- Reduced context switch penalty

Cost of hardware multithreading:

- Requires multiple thread contexts to be implemented in hardware.
- Usually reduced single-thread performance
- Resource sharing, contention
- Switching penalty (can be reduced with additional hardware)

5.2.1 Types of hardware multithreading

The hardware multithreading is classified based on the granularity of the threads as:

- Fine grained
- Coarse grained
- Simultaneous

Fine Grained Multithreading

- Here, the CPU switch to another thread at every cycle such that no two instructions from the thread are in the pipeline at the same time. Hence it is also known as **interleaved multithreading.**

Fine grained multithreading is a mechanism in which switching among threads happen despite the cache miss or stall caused by the thread instruction.

- The threads are executed in a round-robin fashion in consecutive cycles.
- The CPU checks every cycle if the current thread is stalled or not.
- If stalled, a hardware scheduler will change execution to another thread that is ready to run.
- Since the hardware is checking every cycle for stalls, all stall types can be dealt with, even single cycle stalls.
- This improves pipeline utilization by taking advantage of multiple threads
- It tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Fine-grained parallelism is best exploited in architectures which support fast communication.
- Shared memory architecture which has a low communication overhead is most suitable for fine-grained parallelism.
- This requires more threads to keep the CPU busy.

Advantages:

- No need for dependency checking between instructions since only one instruction in pipeline from a single thread.

- No need for branch prediction logic.
- The bubble cycles used for executing useful instructions from different threads.
- Improved system throughput, latency tolerance, utilization.

Disadvantages:

- Extra hardware complexity because of implementation of multiple hardware contexts and thread selection logic.
- Reduced single thread performance as one instruction fetched every N cycles.
- Resource contention between threads in caches and memory.
- Dependency checking logic between threads remains.

Coarse grained multithreading

- In this type, the instructions of other threads are executed successively until an event in current execution thread cause latency. This delay event induces a context switch.
- When a thread is stalled due to some event, the CPU switch to a different hardware context. This is known as Switch-on-event multithreading or **blocked multithreading.**

Coarse grained multithreading is a mechanism in which the switch only happens when the thread in execution causes a stall, thus wasting a clock cycle.

- This is less efficient that fine grained multithreading but requires only few threads to improve CPU utilization.
- The events that causes latency or stalls are: Cache misses, Synchronization events and floating point operations.
- Resource sharing in space and time always requires fairness considerations. This is implemented by considering how much progress each thread makes.
- The time allocated to each thread affects both fairness and system throughput. The allocation strategies depends on the answers to the following questions:
    - When do we switch?
    - For how long do we switch?
    - When do we switch back?

- How does the hardware scheduler interact with the software scheduler for fairness?
- What is the switching overhead vs. benefit?
- Where do we store the contexts?

- A trade off must be done between fairness and system throughput: Switch not only on miss, but also on data return.
- This has a severe problem because switching has performance overhead as it requires flushing of pipeline and window; reduced locality and increased resource contention.
- One possible solution is to estimate the slowdown of each thread compared to when run alone. Then enforce switching when slowdowns become significantly unbalanced.

Advantages:

- Simpler to implement, can eliminate dependency checking and branch prediction logic completely
- Switching need not have any performance overhead.
- Higher performance overhead with deep pipelines and large windows

Disadvantages

- Low single thread performance: each thread gets 1/Nth of the bandwidth of the pipeline

Simultaneous Multithreading (SMT)

- Here instructions can be issued from multiple threads in any given cycle.
- Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor. Thus, the wide superscalar instruction issue is combined with the multiple-context approach.
- In fine-grained and coarse-grained architectures, multithreading can start execution of instructions from only a single thread at a given cycle.
- Execution unit or pipeline stage utilization can be low if there are not enough instructions from a thread to dispatch in one cycle

- Unused instruction slots, which arise from latencies during the pipelined execution of single-threaded programs by a microprocessor, are filled by instructions of other threads within a multithreaded processor.
- The executions units are multiplexed among those thread contexts that are loaded in the register sets.
- Underutilization of a superscalar processor due to missing instruction-level parallelism can be overcome by simultaneous multithreading, where a processor can issue multiple instructions from multiple threads in each cycle.
- Simultaneous multithreaded processors combine the multithreading technique with a wide-issue superscalar processor to utilize a larger part of the issue bandwidth by issuing instructions from different threads simultaneously.

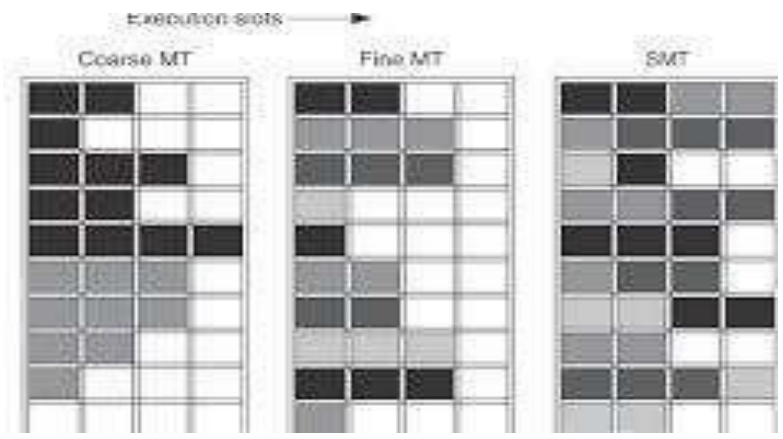


Fig 5.6: Hardware multithreading

5.3 MULTICORE AND SHARED MEMORY MULTIPROCESSORS

A multi-core processor is a single computing component with two or more independent processing units called cores, which read and execute program instructions. A shared-memory multiprocessor is a computer system composed of multiple independent processors that execute different instruction streams.

- Multi-core is usually the term used to describe two or more CPUs working together on the same chip. It is a type of architecture where a single physical processor contains the core logic of two or more processors.

- Shared Memory Processor (SMP) follows multiple-instruction multiple-data (MIMD) architecture.
- The processors share a common memory address space and communicate with each other via memory. All the processors will have dedicated cache memory.
- In a multiprocessor system all processes on the various CPUs share a unique logical address space, which is mapped on a physical memory that can be distributed among the processors.
- Each process can read and write a data item simply using load and store operations, and process communication is through shared memory.
- It is the hardware that makes all CPUs access and use the same main memory.
- Since all CPUs share the address space, only a single instance of the operating system is required.
- When a process terminates or goes into a wait state for whichever reason, the O.S. can look in the process table for another process to be dispatched to the idle CPU.
- On the contrary, in systems with no shared memory, each CPU must have its own copy of the operating system, and processes can only communicate through message passing.
- The basic issue in shared memory multiprocessor systems is memory itself, since the larger the number of processors involved, the more difficult to work on memory efficiently.
- All modern OS support symmetric multiprocessing, with a scheduler running on every processor the ready to run processes can be inserted into a single queue, that can be accessed by every scheduler, alternatively there can be a ¦ready to run¦ queue for each processor.
- When a scheduler is activated in a processor, it chooses one of the ready to run processes and dispatches it on its processor.

Load Balancing:

- A distinct feature in multiprocessor systems is load balancing.

- It is useless having many CPUs in a system, if processes are not distributed evenly among the cores.
- With a single ready-to-run queue, load balancing is usually automatic: if a processor is idle, its scheduler will pick a process from the shared queue and will start it on that processor.
- Modern OSs designed for SMP often have a separate queue for each processor to avoid the problems associated with a single queue.
- There is an explicit mechanism for load balancing, by which a process on the wait list of an overloaded processor is moved to the queue of another, less loaded processor.

5.3.1 Types of shared memory multiprocessors

There are three types of shared memory multiprocessors:

- Uniform Memory Access (UMA)
- Non Uniform Memory Access (NUMA)
- Cache Only Memory Access (COMA)

- **Uniform Memory Access (UMA)**
  - Here, all the processors share the physical memory in a centralized manner with equal access time to all the memory words.
  - Each processor may have a private cache memory. Same rule is followed for peripheral devices.
  - When all the processors have equal access to all the peripheral devices, the system is called a **symmetric multiprocessor.**
  - When only one or a few processors can access the peripheral devices, the system is called an **asymmetric multiprocessor.**
  - When a CPU wants to access a memory location, it checks if the bus is free, then it sends the request to the memory interface module and waits for the requested data to be available on the bus.
  - Multicore processors are small UMA multiprocessor systems, where the first shared cache is actually the communication channel.

- Shared memory can quickly become a bottleneck for system performances, since all processors must synchronize on the single bus and memory access.

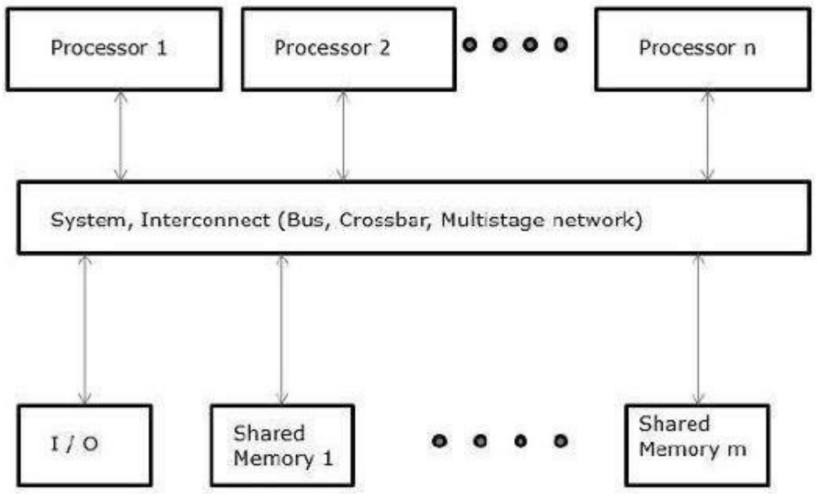


Fig 5.7: Uniform memory access model

Non-uniform Memory Access (NUMA)

- In NUMA multiprocessor model, the access time varies with the location of the memory word.
- Here, the shared memory is physically distributed among all the processors, called local memories.
- The collection of all local memories forms a global address space which can be accessed by all the processors.
- NUMA systems also share CPUs and the address space, but each processor has a local memory, visible to all other processors.
- In NUMA systems access to local memory blocks is quicker than access to remote memory blocks.
- Programs written for UMA systems run with no change in NUMA ones, possibly with different performances because of slower access times to remote memory blocks.
- Single bus UMA systems are limited in the number of processors, and costly hardware is necessary to connect more processors.

- Current technology prevents building UMA systems with more than 256 processors.
- To build larger processors, a compromise is mandatory: not all memory blocks can have the same access time with respect to each CPU.
- Since all NUMA systems have a single logical address space shared by all CPUs, while physical memory is distributed among processors, there are two types of memories: local and remote memory.

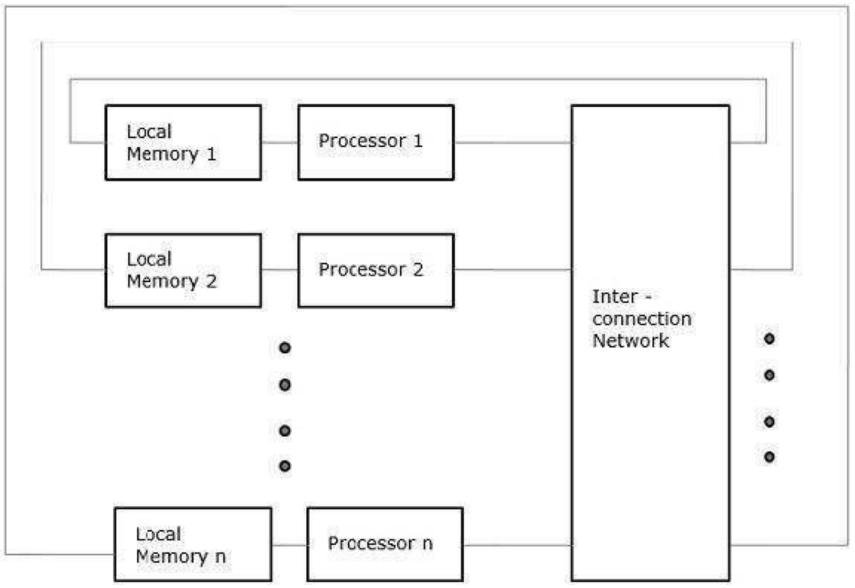


Fig 5.8:Non-uniform Memory Access model

There are two types of NUMA systems: Non-Caching NUMA (NC-NUMA) Cache-Coherent NUMA (CC-NUMA).

Non-Caching NUMA (NC-NUMA):

- In a NC-NUMA system, processors have no local cache. Each memory access is managed with a modified MMU, which controls if the request is for a local or for a remote block; in the latter case, the request is forwarded to the node containing the requested data.

- Obviously, programs using remote data will run much slower than what they would, if the data were stored in the local memory. In NC-NUMA systems there is no cache coherency problem, because there is no caching at all: each memory item is in a single location.
- Remote memory access is however very inefficient. For this reason, NC-NUMA systems can resort to special software that relocates memory pages from one block to another, just to maximize performances.

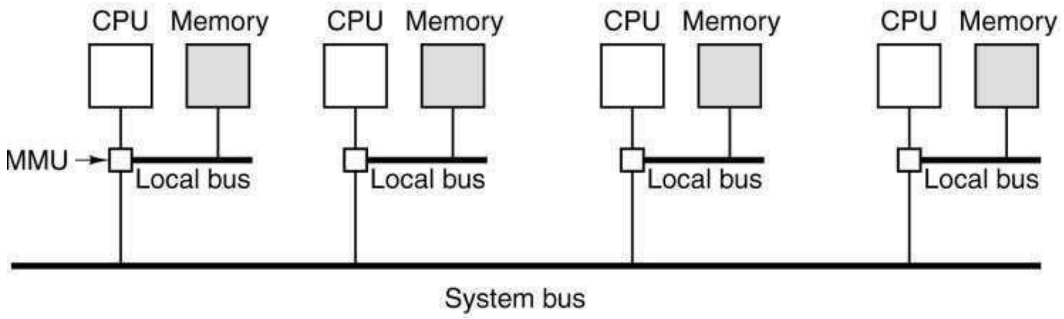


Fig 5.9:Non-Caching NUMA

Cache-Coherent NUMA (CC-NUMA):

- Caching can alleviate the problem due to remote data access, but brings the cache coherency issue.
- A method to enforce coherency is obviously bus snooping, but this techniques gets too expensive beyond a certain number of CPUs, and it is much too difficult to implement in systems that do not rely on bus-based interconnections.
- The common approach in CC-NUMA systems with many CPUs to enforce cache coherency is the directory-based protocol.
- The basic idea is to associate each node in the system with a directory for its RAM blocks: a database stating in which cache is located a block, and what is its state.
- When a block of memory is addressed, the directory in the node where the block is located is queried, to know if the block is in any cache and, if so, if it has been changed respect to the copy in RAM.

- Since a directory is queried at each access by an instruction to the corresponding memory block, it must be implemented with very quick hardware, as an instance with an associative cache, or at least with static RAM.

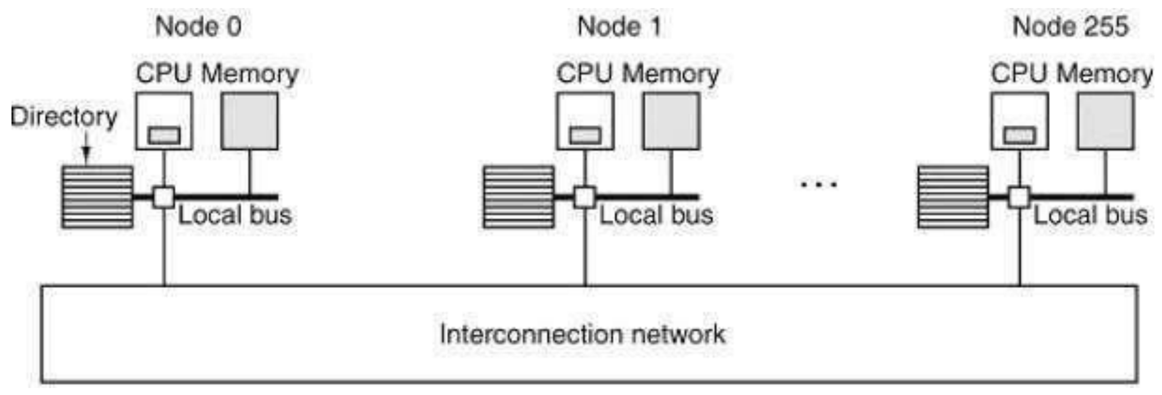


Fig 5.10:Cache-Coherent NUMA

iii) Cache Only Memory Access (COMA)

- The COMA model is a special case of the NUMA model. Here, all the distributed main memories are converted to cache memories.
- In a mono processor architecture and in shared memory architectures each block and each line are located in a single, precise position of the logical address space, and have therefore an address called home address.
- When a processor accesses a data item, its logical address is translated into the physical address, and the content of the memory location containing the data is copied into the cache of the processor, where it can be read and/or modified.
- In the last case, the copy in RAM will be eventually overwritten with the updated copy present in the cache of the processor that modified it.
- This property turns the relationship between processors and memory into a critical one, both in UMA and in NUMA systems:
- In NUMA systems, distributed memory can generate a high number of messages to move data from one CPU to another, and to maintain coherency in home address values. Remote memory references are much slower than local memory ones.

- In CC-NUMA systems, this effect is partially hidden by the caches.
- In UMA systems, centralized memory causes a bottleneck, and limit its the interconnection between CPU and memory, and its scalability.
- In COMA, there is no longer a home address, and the entire physical address space is considered a huge, single cache.
- Data can migrate within the whole system, from a memory bank to another, according to the request of a specific CPU, that requires that data.

5.5 GRAPHICS PROCESSING UNITS

A Graphics Processing Unit (GPU) is a single-chip processor primarily used to manage and boost the performance of video and graphics. It is a dedicated parallel processor for accelerating graphical and deeper computations.

GPU is designed to lessen the work of the CPU and produce faster video and graphics. GPU can be thought as an extension of CPU with thousands of cores. A GPU is extensively used in a PC on a video card or motherboard, mobile phones, display adapters, workstations and game consoles. They are mainly used for offloading computation intensive application. This is also known as a **visual processing unit (VPU).**

Differences between CPU and GPU

GPU	CPU
They facilitate highly parallel operations.	This supports serial execution of programs.
This has more number of cores (in thousands).	This has less number of cores.
They need special faster interfaces to facilitate faster data transfers.	No such special interfaces are required.
They have deeper pipelines. pipelines.	They have comparatively shallow

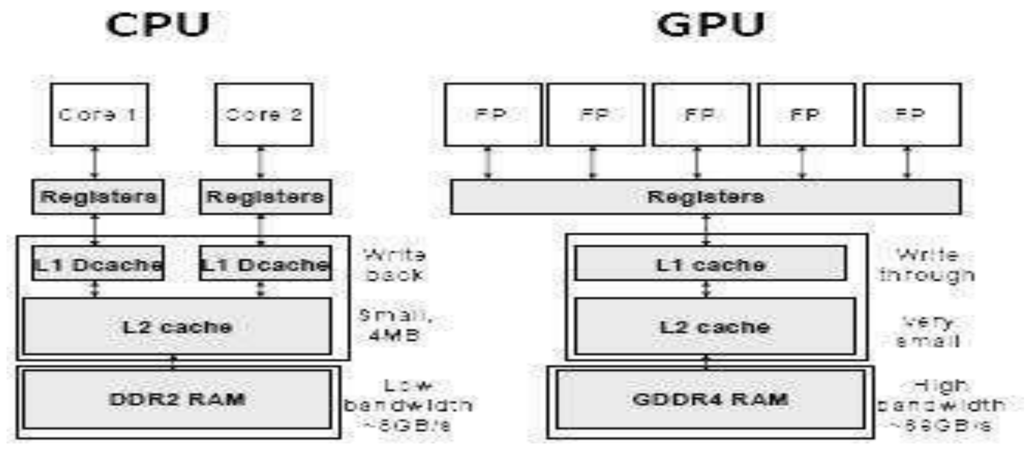


Fig 5.11: CPU vs GPU architecture

5.5.1 GPU features

The following are prominent features of GPU:

- 2-D or 3-D graphics
- Digital output to flat panel display monitors
- Texture mapping
- Application support for high-intensity graphics software such as AutoCAD
- Rendering polygons
- Support for YUV color space
- Hardware overlays
- MPEG decoding

5.5.2 Development of GPU

- The first GPU was developed by NVidia in 1999 and named as GeForce 256.
- This GPU model could process 10 million polygons per second and had more than 22 million transistors.
- This is a single-chip processor with integrated transform, drawing and BitBLT support, lighting effects, triangle setup /clipping and rendering engines.

- The GPU is connected to the CPU and is completely separate from the motherboard.
- The RAM is connected through the Accelerated Graphics Port (AGP) or the PCI express bus.
- Sometimes, GPUs are integrated into the north bridge on the motherboard and use the main memory as a digital storage area, but these GPUs are slower and have poorer performance.
- The accelerated memory in GPU is used for mapping vertices and can also supports programmable shade implementing textures, mathematical vertices and accurate color formats.
- Applications such as Computer-Aided Design (CAD) can process over 200 billion operations per second and deliver up to 17 million polygons per second.
- The main configurations of GPU processor are: Graphics coprocessor which is independent of CPU and Graphics accelerator that is based on commands from CPU.

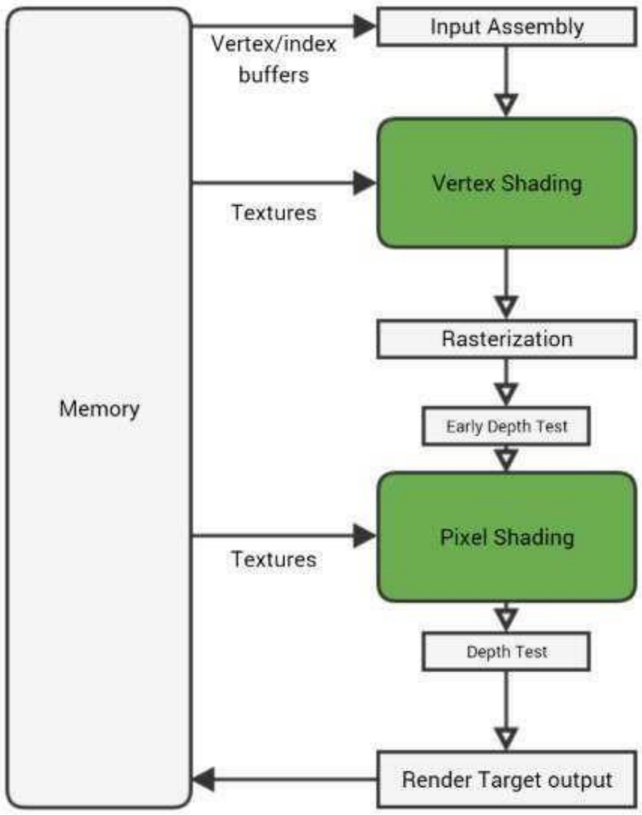


Fig 5.12: GPU Pipeline

Input Assembler stage

- This stage is the communication bridge between the CPU and GPU.
- It receives commands from the CPU and also pulls geometry information from system memory.
- It outputs a stream of vertices in object space with all their associated information.

Vertex Processing

- This processes vertices performing operations like transformation, skinning and lighting.
- A vertex shade takes a single input vertex and produces a single output vertex.

Pixel Processing

- Each pixel provided by triangle setup is fed into pixel processing as a set of attributes which are used to compute the final color for this pixel.
- The computations taking place here include texture mapping and math operations

Output Merger Stage

- The output-merger stage combines various types of output data to generate the final pipeline result.

5.6 CLUSTERS AND WAREHOUSE SCALE COMPUTERS

A cluster is a collection of desktop computers or servers connected together by a local area network to act as a single larger computer. A warehouse-scale computer (WSC) is a cluster comprised of tens of thousands of servers

Warehouse-scale computers form the foundation of internet services. The present days WSCs act as one giant machine. The main parts of a WSC are the building with the electrical and cooling infrastructure, the networking equipment and the servers.

WSCs as Servers

The following features of WSCs that makes it work as servers:

- **Cost-performance**: Because of the scalability, the cost-performance becomes very critical. Even small savings can amount to a large amount of money.

- **Energy efficiency:** Since large numbers of systems are clustered, lot of money is invested in power distribution and for heat dissipation. Work done per joule is critical for both WSCs and servers because of the high cost of building the power and mechanical infrastructure for a warehouse of computers and for the monthly utility bills to power servers. If servers are not energy-efficient they will increase
    - cost of electricity
    - cost of infrastructure to provide electricity
    - cost of infrastructure to cool the servers.
- **Dependability via redundancy:** The hardware and software in a WSC must collectively provide at least 99.99% availability, while individual servers are much less reliable. Redundancy is the key to dependability for both WSCs and servers. WSC architects rely on multiple cost-effective servers connected by a low cost network and redundancy managed by software. Multiple WSCs may be needed to handle faults in whole WSCs. Multiple WSCs also reduce latency for services that are widely deployed.
- **Network I/O:** Networking is needed to interface to the public as well as to keep data consistent between multiple WSCs.
- **Interactive and batch-processing workloads**: Search and social networks are interactive and require fast response times. At the same time, indexing, big data analytics etc. create a lot of batch processing workloads also. The WSC workloads must be designed to tolerate large numbers of component faults without affecting the overall performance and availability.

Differences between WSCs and data centers

Data Centers	**WSCs**
Data centres hosts services for multiple providers.	WSCs are run by only one client.
There will be little commonality between hardware and software.	Homogenous hardware and software management.
Third party software solutions.	In-house middle ware.

WSC are not servers:

The following features of WSCs make them different from servers:

- **Ample parallelism:**
  - Servers need not to worry about the parallelism available in applications to justify the amount of parallel hardware.
  - But in WSCs most jobs are totally independent and exploit request-level parallelism.
  - **Request-Level parallelism (RLP)** is a way of representing tasks which are set of requests which are to be to run in parallel.
  - Interactive internet service applications, the workload consists of independent requests of millions of users.
  - Also, the data of many batch applications can be processed in independent chunks, exploiting data-level parallelism.
- **Operational costs count:**
  - Server architects normally design systems for peak performance within a cost budget.
  - Power concerns are not too much as long as the cooling requirements are maintained. The operational costs are ignored.
  - WSCs, however, have a longer life times and the building, electrical and cooling costs are very high.
  - So, the operational costs cannot be ignored. A
  - ll these add up to more than 30% of the costs of a WSC in 10 years.
  - Power consumption is a primary, not secondary constraint when designing the WSC system.
- **Scale and its opportunities and problems**:
  - The WSCs are massive internally, so it gets volume discounts and economy of scale, even if there are not too many WSCs.
  - On the other hand, customized hardware for WSCs can be very expensive, particularly if only small numbers are manufactured.

- The economies of scale lead to cloud computing, since the lower per-unit costs of WSCs lead to lower rental rates.
- Even if a server had a Mean Time To Failure (MTTF) of twenty five years, the WSC architect should design for five server failures per day.

5.6.1 Architecture of WSC

The height of the servers is measured by **rack units.** A typical rack is 42 rack units. But the standard dimension to hold the servers is 48.26 cm.

1 rack unit (U)=1.75 inches or 44.45 mm.

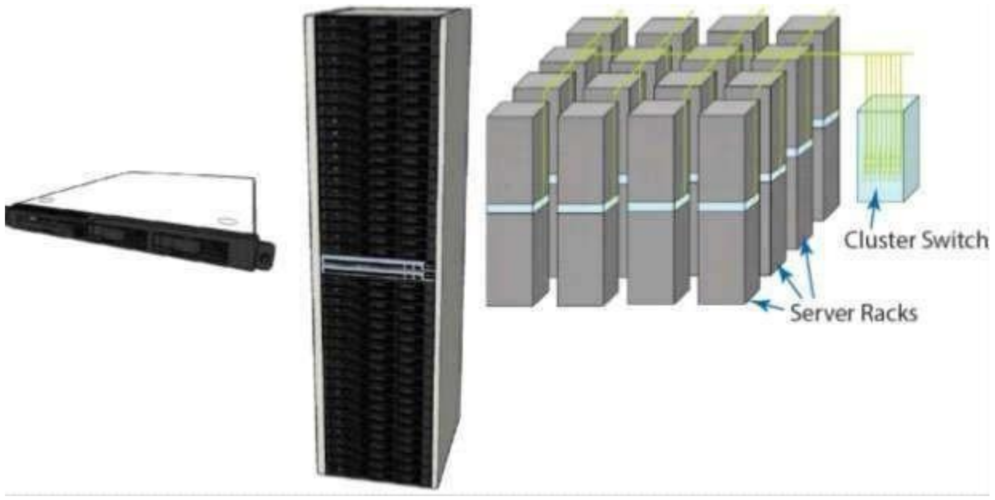


Fig 5.13: Architecture of WSCs

The fig 5.13 shows a WSC system with 1Unit server, 7 inch rack with an Ethernet switch. This figure shows a high end server. But low end servers are of 1U size mounted within a rack and connected with Ethernet switch. These rack level switches use 1 or 10 Gbps links with a number of uplink connections to cluster level switches. The second level switching can span more than 10,000 individual servers.

5.6.2 Programming model for WSC

There is a high variability in performance between the different WSC servers because of:

- varying load on servers
- file may or may not be in a file cache
- distance over network can vary
- hardware anomalies

A WSC will start backup executions on other nodes when tasks have not yet completed and take the result that finishes first. Rely on data replication to help with read performance and availability. A WSC also has to cope with variability in load. Often WSC services are performed with in-house software to reduce costs and optimize for performance.

5.6.2 Storage of WSC

- A WSC uses local disks inside the servers as opposed to network attached storage (NAS).
- The Google file system (GFS) uses local disks and maintains at least three replicas to improve dependability by covering not only disk failures, but also power failures to a rack or a cluster of racks by placing the replicas on different clusters.
- A read is serviced by one of the three replicas, but a write has to go to all three replicas.
- Google uses a relaxed consistency model in that all three replicas have to eventually match, but not all at the same time.

5.6.3 WSC networking

- A WSC uses a hierarchy of networks for interconnection.
- The standard rack holds 48 servers connected by a 48-port Ethernet switch. A rack switch has 2 to 8 uplinks to a higher switch.
- So the bandwidth leaving the rack is 6 (48/8) to 24 (48/2) times less than the bandwidth within a rack.
- There are array switches that are more expensive to allow higher connectivity.
- There may also be Layer 3 routers to connect the arrays together and to the Internet.
- The goal of the software is to maximize locality of communication relative to the rack.

5.6.4 Performance

Power Utilization Effectiveness (PUE) is widely used metric to estimate the performance of WSCs.

$$PUE = \frac{Total\_utility\_power}{IT\_equipment\_power}$$

Bandwidth is an important metric as there may be many simultaneous user requests or metadata generation batch jobs. Latency is also equally important metric as it is seen by users when they make requests. Users will use a search engine less as the response time increases. Also users are more productive in responding to interactive information when the system response time is faster as they are less distracted.

5.7 MULTIPROCESSOR NETWORK TOPOLOGIES

Multiprocessor system consists of multiple processing units connected via some interconnection network plus the software needed to make the processing units work together. There are two major factors used to categorize such systems:

- the processing units
- the interconnection network

A number of communication styles exist for multiprocessing networks. These can be broadly classified according to the communication model as shared memory (single address space) versus message passing (multiple address spaces).

5.7.1 Design Issues of Interconnection Networks

The important issue in the design of multiprocessor systems is how to cope with the problem of an adequate design of the interconnection network in order to achieve the desired performance at low cost. The choice of the interconnection network may affect several characteristics of the system such as node complexity, scalability and cost etc. The following are the issues which should be considered while designing an interconnection network.

- **Dimension and size of network:** It should be decided how many processing element are there in the network and what the dimensionality of the network is i.e. with how many neighbors, each processor is connected.

- **Symmetry of the network:** It is important to consider whether the network is symmetric or not i.e., whether all processors are connected with same number of processing elements or the processing elements of corners or edges have different number of adjacent elements.
- **Message Size:** Message size is dependent on the amount of data that can be transferred in one unit time.
- **Data transfer Time:** The time taken for a message to reach to another processor, Whether this time is a function of link distance between two processors or it depends upon the number of nodes coming in between are chief factors
- **Startup Time:** It is the time of initiation of the process.

5.7.2 Performance parameters

- **Number of nodes (N):** The number of nodes in a multiprocessor network plays a dynamic role by virtue of which the performance of the system is evaluated. Higher number of nodes means higher complexity but higher is the system performance. Therefore, number of processors should be optimal.
- **Node degree (D):** The node degree of the network is defined as the number of edges connected with the nodes. It is the connectivity among different nodes in a network. The connectivity of the nodes determines the complexity of the network. The greater number of links in the network means greater is the complexity. If the edge carries data from the node, it is called out degree and if this carries data into the node then it is called in degree.
- **Diameter (D):** The network diameter is defined as the maximum shortest path between the source and destination node. The path length is measured by the number of links traversed. This virtue is important in determining the distance involved in communication and hence the performance of parallel systems. The low diameter is always better because the diameter puts a lower bound on the complexity of parallel algorithms requiring communication between arbitrary pairs of nodes.
- **Cost (C):** It is defined as the product of the diameter and the degree of the node for a symmetric network.

$$\text{Cost (C)} = \text{Diameter} * \text{Degree} = D * d$$

Greater number of nodes means greater the cost of the network. It is good creation to measure the hardware cost and the performance of the multiprocessor network and gives more insight to design a cost-effective parallel system.

- **Extensibility**

It is virtue which facilitates large sized system out of small ones with minimum changes in the configuration of the nodes. It is the smallest increment by which the system can be expanded in a useful way. A network with large number of links or a large node degree tends to increase the hardware cost. Expandability is an important parameter to evaluate the performance of a multiprocessor system. The feasibility to extend a system while retaining its topological characteristics enables to design large scale parallel systems.

5.7.4 Network Topologies

The multiprocessor networks are classified in two broad categories based on their topological properties. These are given below:

- Cube based network
- Linearly Extensible Network

Cube Based Network

- The cube based architectures are widely used networks in parallel systems. They have good topological properties such as symmetry, scalability and possess a rich interconnection topology. The types of cube based networks are:
- **Binary hypercube or n-cube:**
  - This is a loosely coupled parallel multiprocessor based on the binary n-cube network.
  - An n-dimensional hypercube contains 2^n nodes and has n edges per node.
  - In hypercube, the number of communication links for each node is a logarithmic function of the total number of nodes.
  - The hypercube organization has low diameter and high bisection width at the expense of the number of edges per node and the length of the longest edge.

- The length of the longest edge in a hypercube network increases as the number of nodes in the network increases.
- The node degree increases exponentially with respect to the dimension, making it difficult to consider the hypercube a scalable architecture.
- The major drawback of the hypercube is the increase in the number of communication links for each node with the increase in the total number of nodes.

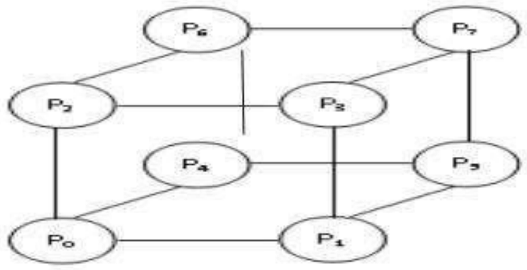


Fig 5.14: Hypercube

Cube Connected Cycle (CCC)

- The CCC architecture is an attractive parallel computation network suitable for VLSI implementation while preserving all the desired features of hypercube.
- The CCC is constructed from the n- dimensional hypercube by replacing each node in hypercube with a ring containing n node.
- Each node in a ring then connects to a distinct node to one of the n dimensions.
- The advantage of the cube- connected cycles is that node's degree is always 3, independent of the value of n. This architecture is modified from hypercube i.e. a 3-cube is modified to form a 3-cube-connected cycles (CCC) restricted the node degree to 3.
- The idea is to replace the corner nodes (vertices) of the 3-cube with a ring of 3-nodes.
- In general one can construct k-cube-connected cycles from a k-cube with n=2^k rings nodes.

Folded Hyper Cube (FHC)

- The FHC is the variation of the hypercube network and constructed by introducing some extra links to the hypercube.
- Halved diameter, better average distance, shorter delay in communication links, less message traffic density, lower cost make it very promising.
- e hardware overhead is almost 1/n, n being the dimensionality of the hypercube, which is negligible for large n.
- Optimal routing algorithms are developed are developed and proven to be remarkably more efficient than those of the conventional n-cube.
- A folded hypercube of dimension n is called FHC (n).
- The FHC (n) is a regular network of node connectivity (n+1)and the hypercube of degree 3 is converted to FHC (n) network.
- Extended versions of FHC (n) is called Extended Folded Cube (EFC). The EFC has better properties than the other variations of basic hypercube in terms of parameter.
- It has constant node degree, smaller diameter, and lower cost and also it maintains several numerous desirable characteristics including symmetry, hierarchical, expansive, recursive.

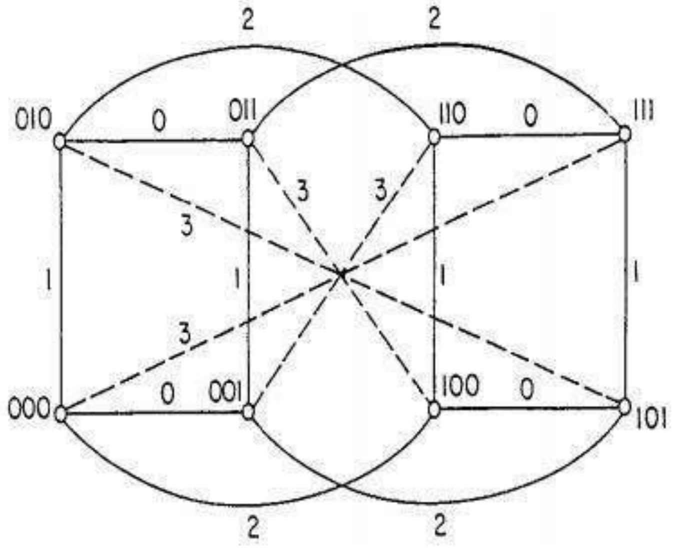


Fig 5.15: Folded Hypercube

Crossed Cube

- The Crossed Cube (CC) has the same node and link complexity as the hypercube and has most of its desirable properties including regularity, recursive structure, partition ability, strong connectivity and ability to simulate other architectures.
- Its diameter is only half of the diameter of the hypercube.
- Mean distance between vertices is smaller and it can simulate a hypercube through dilation 2 embedding.
- The basic properties of the CC, optimal routing and broadcasting algorithms are developed.
- The CC is derived from a hypercube by changing the way of connection of some hypercube links.
- The diameter of CC is almost half of that of its corresponding hypercube.

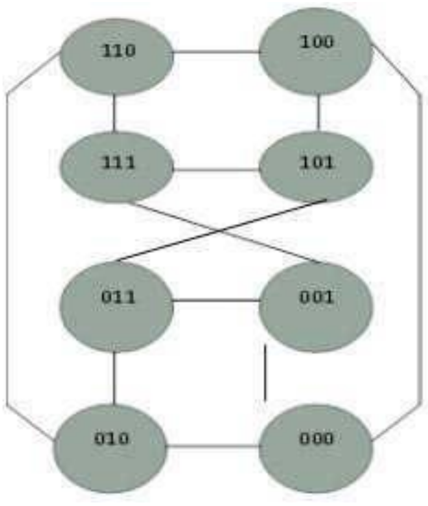


Fig 5.16: Crossed Cube

Reduced Hypercube (RHC)

- The RH (k, m) is obtained from the n- dimensional hypercube by reducing node edges in hypercube by following rules where k+2m= n.
- The lower VLS) complexity of R(¦s permit the construction of systems with more processing elements than are found in conventional hypercube.
- There are clusters and each cluster is a conventional k- dimensional hypercube.
- Of the higher n-k=2m dimensions, a node has only one direct connection is decided by the leftmost m bits in the k-bits field, i.e., the (2i + k) dimension, where i is the value of the m-bit binary number.

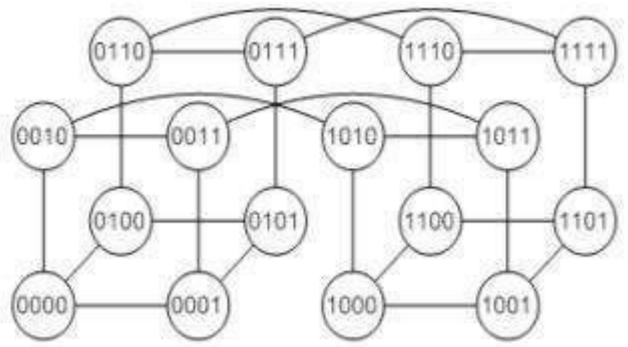


Fig 5.17: Reduced Hypercube

Hierarchal Cube Network (HCN)

- The Hierarchical Cube Network (HCN) is interconnection network for large-scale distributed memory multiprocessors.
- HCN has about three-fourths the diameter of a comparable hypercube, although it uses about half as many links per node-a fact that has positive ramifications on the implementation of HCN-connected systems.
- The HCN (n, n)has 2n clusters, where each cluster is an n-cube.
- Each node in the HCN (n, n) has n+1 links connected to it. n links are used inside the cluster. The additional links are used to connect nodes among clusters.
- The advantage of HCN is that the number of links required is reduced approximately to half as many links per node and the diameter is reduced to about three-fourth of a corresponding hypercube.

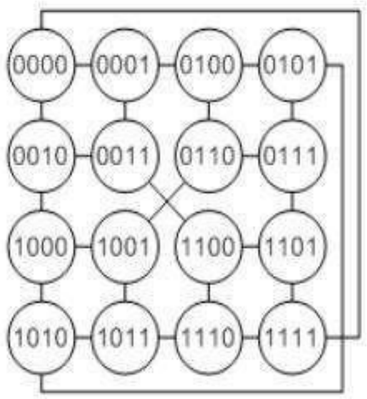


Fig 5.18: Hierarchical Cube Network

Dual Cube (DC)

- The DC is a new interconnection topology for large-scale distributed memory Multiprocessors that reduces the problem of increasing number of links in the large-scale hypercube network.
- This preserves most of the topological properties of the hypercube network.
- The DC shares the desired properties of the hypercube, however increases tremendously the total number of nodes in the system with limited links per node.
- The key properties of hypercube are also true in the dual-cube: each node can be represented by unique binary number such that two nodes are connected by an edge if and only if the two binary numbers differ in one bit only.
- However, the size of the dual-cube can be as large as eight thousands with up to eight links per node.
- A dual-cube uses binary hypercube as basic components. Each such hypercube component is referred to as a **cluster.**
- Assume that the number of nodes in a cluster is 2^m. In a dual cube, there are two
- Classes with each class consisting of 2^m clusters.
- The total number of nodes is 2^m or 2^{m+1}. Therefore, the nodes address has 2m+1 bits
- The leftmost bit is used to indicate the type of the class (class 0 and class 1).
- For the class 0, the rightmost m bits are used as the node ID within the cluster.
- Each node in cluster of class 0 has one and only one extra connection to a node in a cluster of class 1.

Meta Cube (MC)

- The MC is an interconnection topology for a very large parallel system. Meta cube network has two level cube structures. An MC (k, m) network can connect $2k+m^{2k}$ nodes with (k+m) links per node where k is the dimension of the high-level cubes (classes) and m is the dimension of the low-level cubes (clusters).
- In this network, the number of nodes is much larger than the hypercube with a small number of links per node.

 An MC network is a symmetric network with short diameter, easy and efficient routing.

- Similar to that of the hypercube.
- The meta cube has tremendous potential to be used as an interconnection network for very large scale parallel computers since the meta cube can connect hundreds of millions nodes with up to six links per node and it keeps some desired properties of the hypercube that are useful efficient communication among the nodes.

Folded Dual Cube (FDC)

- The FDC is a new cube based Interconnection topology for parallel systems with reduced diameter, cost and constructed from DC and FHC.
- The FDC is a graph Fr (V, E), where V represents a set of vertices and E represent a set of links.
- The FDC is to be slightly greater than Dual cube but quite less than HC and FHC.
- Diameter of FDC is found to be smaller than that of Dual cube and with the comparison of Dual cube, HC and FHC.
- FDC exhibits quite a good improvement in broadcast time over its parent networks with millions of nodes.
- The cost of the FDC topology is found to be less. The FDC will help to speed up the overall operation of large scale parallel systems.

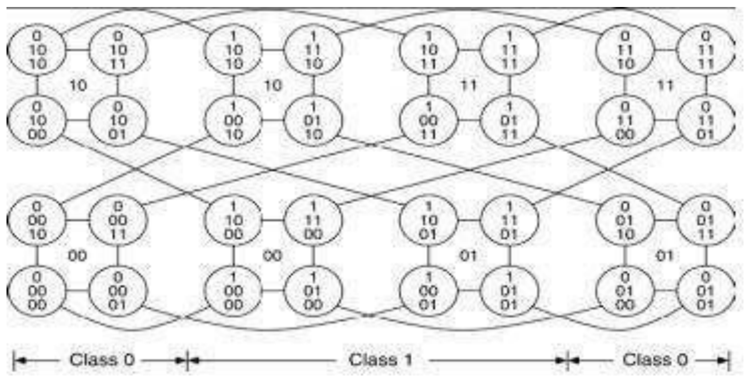


Fig 5.19: Folded Dual Cube

Folded Meta cube (FMC)

- The FMC is an efficient large scale parallel interconnection topology with better features such as reduced diameter, cost, improved broadcast time and constructed from MC.

- The FMC is a graph G (V, E), where V represents a set of vertices and E represent a set of links.
- The FMC is to be slightly greater than Meta cube but quite less than HC and FHC.
- Diameter of FMC is found to be smaller than that of Meta cube.
- FMC exhibits quite a good improvement in broadcast time over its parent network while connecting millions of nodes.
- The cost of the FMC is found to be less and will help to speed the overall operation of large scale parallel systems.

Necklace Hypercube (NH)

- NH is an array of processors attached to each two adjacent nodes of the hypercube network.
- It is highly scalable architecture while preserving most of the desirable properties of hypercube such as logarithmic diameter, fault tolerance etc.
- It has also some other properties such as hardware scalability and efficient VLSI layout that make it more attractive than an equivalent hypercube network.
- The Necklace-Hypercube is an undirected graph which has a necklace of processors to each edge of hypercube.
- The necklace length may be fixed or variable for different edge necklaces.

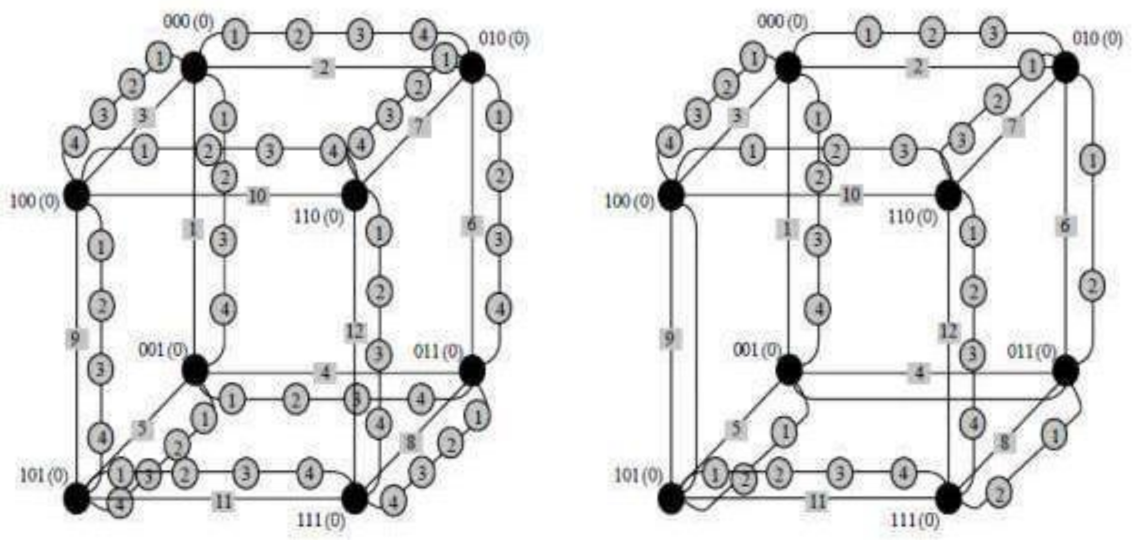


Fig 5.20: Necklace Hypercube

Linearly Extensible Network

The Linearly Extensible Networks is another class of multiprocessor architectures which reduces some of the drawbacks of HC architectures. The complexity of these networks is lesser as they do not have exponential expansion. Besides the scalability, other parameters to evaluate the performance of such networks are degree, number of nodes, diameter, bisection width and fault tolerance. Selection of a better interconnection network may have several applications with lesser complexities and improved power-efficiency.

Linear Array (LA)

- It is one dimensional network having the simplest topology with n-nodes having n-1 communication links.
- The internal nodes have degree 2 and the termination nodes have degree1.
- The diameter is n-1, which is long for large n and the bisection width is 1.
- It is asymmetric network. Linear array are the simplest connection topology.
- As the diameter increases linearly with respect to n, it should not be used for large n. For every small n, it is rather economical to implement a linear array.

Binary Tree (BT)

- A binary tree is either empty or consists of node called the root together with two binary trees called left sub tree and the right sub tree.
- When h is equal to height of a binary tree then maximum leaves are equal to 2^h and maximum nodes are $2^{h+1}-1$.
- In a binary tree network there is only one path between any two nodes.
- The binary tree is scalable architecture with a constant node degree and constant bisection width. In general, an n-level, complexity balanced binary tree should have N=2n-1 nodes.
- The maximum node degree is 3 and the diameter is 2(n-1). But has a poor bisection width of 1.

Ring (R)

- This is a simple linear array where the end nodes are connected. It is equivalent to mesh with wrap around connections.
- The data transfer in a ring is normal one direction. A ring is obtained by connecting the two terminal nodes of a linear array with one extra link.
- A ring network can be uni-or bidirectional and it is symmetric with a constant.
- It has a constant node degree of d=2, the diameter is N/2for a bidirectional ring and N for unidirectional ring.
- A ring network has a constant width 2.

Linearly Extensible Tree (LET)

- The Linearly Extensible Tree (LET) architecture exhibits better connectivity, lesser number of nodes over cube based networks.
- The LET network has low diameter, hence reduce the average path length traveled by all message and contains a constant degree per node.
- The LET network grows linearly in a binary tree like shape.
- In a binary tree the number of nodes at level n is 2n whereas in LET network the number is (n+1).

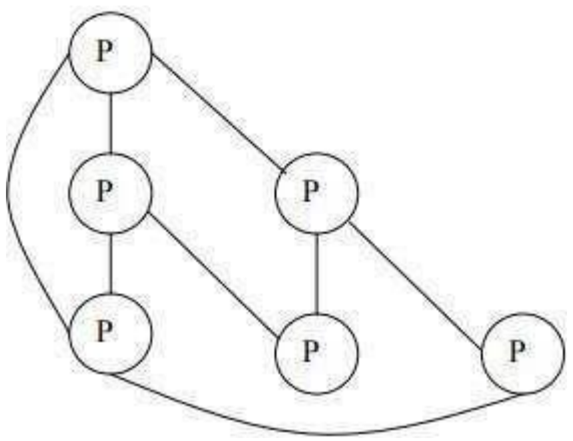


Fig 5.21: Linearly Extensible Tree

Linearly Extensible Cube (LEC)

- The LEC network grows linearly and possesses some of the desirable topological properties topological properties such as small diameter, high connecting constant node degree with high scalability.
- It has constant expansion of only two processors at each level of the extension while preserving all the desirable topological properties.
- The LEC network can maintain a constant node degree regardless of the increase in size in a network.
- The number of nodes in LEC network is 2 * n for n > 0 whereas the number of nodes in the hypercube is 2n. The diameter of network is N. It has a constant node degree 4.

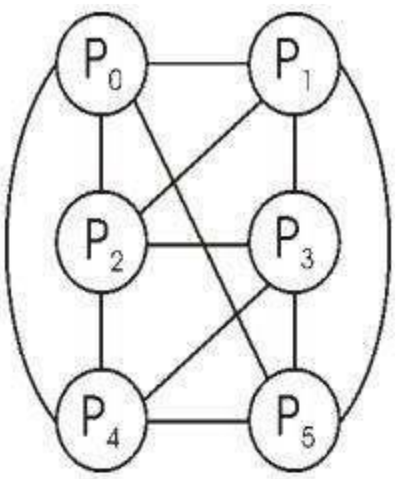


Fig 5.22: Linearly Extensible Cube