Paper Name: Computer Organization and Architecture

SYLLABUS

1. Introduction to Computers
Basic of Computer, Von Neumann Architecture, Generation of Computer, Classification of Computers, Instruction Execution

2. Register Transfer and Micro operations
Register Transfer, Bus and Memory Transfers, Tree-State Bus Buffers, Memory Transfer, Micro-Operations, Register Transfer Micro-Operations, Arithmetic Micro-Operations, Logic Micro-Operations, Shift Micro-Operations.

3. Computer Arithmetic
Addition And Subtraction With Signed-Magnitude, Multiplication Algorithm, Booth Multiplication Algorithm, Array Multiplier, Division Algorithm, Hardware Algorithm, Divide Overflow, Floating-Point Arithmetic Operations, Basic Considerations, Register Configuration, Addition And Subtraction, Decimal Arithmetic Operations, BCD Adder, BCD Subtraction.

4. Programming the Basic Computer
Machine language, Assembly language, Assembler, First pass, Second pass, Programming Arithmetic and Logic operations, Multiplication Program, Double-Precision Addition, Logic operations, Shift operations.

5. Organization of a Computer
Central Progressing Unit (CPU), Stack Organization, Register Stack, Memory Stack, Reverse Polish Notation. Instruction Formats, Three- Address Instructions, Two – Address Instructions, One- Address Instructions, Zero-Address Instructions, RISC Instructions, Addressing Modes Reduced Instruction Set Computer, CISC Characteristics RISC Characteristics.

6. Input-Output Organization
Modes Of Transfer, Priority Interrupt, DMA, Input-Output Processor (IOP), CPU-IOP Communication.

7. Memory Organization
Memory Hierarchy, Main Memory, Auxiliary Memory, Cache Memory, Virtual Memory. Address Space and Memory Space, Associative Memory, Page Table, Page Replacement.

8. Parallel Processing and Vector Processing
Pipelining, Parallel Processing, Pipelining General Consideration, Arithmetic Pipeline Instruction Pipeline, Vector Operations, Matrix Multiplication, Memory Interleaving.

9. Multiprocessors
Characteristics of Multiprocessors, Interconnection Structure
Time-Shared Common Bus, Multi-Port Memory, Crossbar Switch, Multistage Switching Network, Hypercube Interconnection, Inter Processor Arbitration, Cache Coherence

References: -

1. "Computer System Architecture", John. P. Hayes.
2. "Computer Architecture and parallel Processing ", Hwang K. Briggs.
3. "Computer System Architecture", M.Morris Mano.

SYLLABUS

Paper Name: Computer Organization and Architecture

Introduction to Computers

Basic of Computer, Von Neumann Architecture, Generation Of Computer, Classification Of Computers, Instruction Execution.

Register Transfer and Micro operations

Register transfer, Bus and Memory Transfers, tree-state bus buffers, Memory transfer, Micro-Operations , Register transfer Micro-Operations, Arithmetic Micro-Operations , Logic Micro-Operations, Shift Micro-Operations.

Computer Arithmetic

Addition and subtraction with signed-magnitude, Multiplication algorithm, Booth multiplication algorithm, Array multiplier, Division algorithm, Hardware algorithm, Divide Overflow , Floating-point Arithmetic operations,
Basic considerations, Register configuration, Addition and subtraction, Decimal Arithmetic operations, BCD adder, BCD subtraction.

Programming the Basic Computer

Machine language, Assembly language, Assembler, First pass, Second pass, Programming Arithmetic and Logic operations, Multiplication Program, Double-Precision Addition, Logic operations, Shift operations,.

Organization of a Computer

Central Progressing Unit (CPU), Stack organization, Register stack, Memory stack, Reverse polish notation .Instruction Formats, Three- address Instructions, Two – address instructions, One- address instructions, Zero-address instructions, RISC Instructions, Addressing Modes Reduced Instruction Set Computer, CISC characteristics RISC characteristics.

Input-Output Organization

Paper Name: Computer Organization and Architecture

Modes of transfer, Priority interrupt, DMA, Input-Output Processor (IOP), CPU-IOP Communication.

Memory Organization

Memory hierarchy, Main memory, Auxiliary memory, Cache memory, Virtual memory. Address Space and Memory Space, Associative memory ,Page table
Page Replacement.

Introduction to Parallel Processing

Pipelining, Parallel processing , Pipelining general consideration, Arithmetic pipeline Instruction pipeline .

Vector Processing

Vector operations, Matrix multiplication, Memory interleaving.

Multiprocessors

Characteristics of multiprocessors, Interconnection structure
Time-shared common bus, Multi-port memory ,Crossbar switch ,Multistage switching network, Hypercube interconnection, Inter processor arbitration , Cache coherence

Paper Name: Computer Organization and Architecture

TABLE OF CONTENTS

Paper Name: Computer Organization and Architecture

- 3.2.3 Division algorithm
  - 3.2.3.1 Hardware algorithm
  - 3.2.3.2 Divide Overflow
- 3.4 Floating-point Arithmetic operations
  - 3.4.1 Basic considerations
  - 3.4.2 Register configuration
  - 3.4.3 Addition and subtraction
- 3.5 Decimal Arithmetic operations
  - 3.5.1 BCD adder
  - 3.5.2 BCD subtraction

Unit 4 : Programming the Basic Computer

- 4.1 Machine language
- 4.2 Assembly language
- 4.3 Assembler
  - 4.3.1 First pass
  - 4.3.2 Second pass
- 4.4 Programming Arithmetic and Logic operations
  - 4.4.1 Multiplication Program
  - 4.4.2 Double-Precision Addition
  - 4.4.3 Logic operations
  - 4.4.4 Shift operations

Unit 5 : Central Progressing Unit (CPU)

- 5.1 Stack organization
  - 5.1.1 Register stack
  - 5.1.2 Memory stack
  - 5.1.3 Reverse polish notation
- 5.2 Instruction Formats
  - 5.2.1 Three- address Instructions
  - 5.2.2 Two – address instructions
  - 5.2.3 One- address instructions
  - 5.2.4 Zero-address instructions
  - 5.2.5 RISC Instructions
- 5.3 Addressing Modes
- 5.4 Reduced Instruction Set Computer
  - 5.4.1 CISC characteristics
  - 5.4.2 RISC characteristics

Unit 6: Input-Output Organization

- 6.1 Modes of transfer
  - 6.1.1 Programmed I/O
  - 6.1.2 Interrupt-Initiated I/O
- 6.2 Priority interrupt
  - 6.2.1 Daisy-chaining priority
  - 6.2.2 Parallel priority interrupt
  - 6.2.3 Interrupt cycle

Paper Name: Computer Organization and Architecture

- 6.3 DMA
  - 6.3.1 DMA Controller
  - 6.3.2 DMA Transfer
- 6.4 Input-Output Processor (IOP)
  - 6.1.1 CPU-IOP Communication
  - 6.1.2 Serial Communication
  - 6.1.3 Character-Oriented Protocol
  - 6.1.4 Bit-Oriented Protocol
- 6.5 Modes of transfer

Unit-7 Memory Organization

- 7.1 Memory hierarchy
- 7.2 Main memory
  - 7.2.1 RAM and ROM chips
  - 7.2.2 Memory Address Map
- 7.3 Auxiliary memory
  - 7.3.1 Magnetic disks
  - 7.3.2 Magnetic Tape
- 7.4 Cache memory
  - 7.4.1 Direct Mapping
  - 7.4.2 Associative Mapping
  - 7.4.3 Set- associative Mapping
  - 7.4.4 Virtual memory
  - 7.4.5 Associative memory Page table
  - 7.4.6 Page Replacement

UNIT 8 : Introduction to Parallel Processing

- 8.1 Pipelining
  - 8.1.1 Parallel processing
  - 8.1.2 Pipelining general consideration
  - 8.1.3 Arithmetic pipeline
  - 8.1.4 Instruction pipeline

Unit 9: Victor Processing

- 9.1 Vector operations
- 9.2 Matrix multiplication
- 9.3 Memory interleaving

UNIT 10 : Multiprocess

- 10.1 Characteristics of multiprocessors
- 10.2 Interconnection structure
  - 10.2.1 Time-shared common bus
  - 10.2.2 Multi-port memory
  - 10.2.3 Crossbar switch
  - 10.2.4 Multistage switching network
  - 10.2.5 Hypercube interconnection

10.3 Inter processor arbitration
10.4 Cache coherence
10.5 Instruction Execution

Paper Name: Computer Organization and Architecture

UNIT 1

INTRODUCTION TO COMPUTERS

Computer: An Introduction

1.1 Introduction

Computer is one of the major components of an Information Technology network and gaining increasing popularity. Today, computer technology has permeated every sphere of existence of modern man. In this block, we will introduce you to the computer hardware technology, how does it work and what is it? In addition we will also try to discuss some of the terminology closely linked with Information Technology and computers.

1.2 WHAT IS COMPUTER?

Computer is defined in the Oxford dictionary as “An automatic electronic apparatus for making calculations or controlling operations that are expressible in numerical or

logical terms" . A device that accepts data1, processes the data according to the instructions provided by the user, and finally returns the results to the user and usually consists of input, output, storage, arithmetic, logic, and control units. The computer can store and manipulate large quantities of data at very high speed

The basic function performed by a computer is the execution of a **program**. A program is a sequence of instructions, which operates on data to perform certain tasks. In modern digital computers data is represented in binary form by using two symbols 0 and 1, which are called **binary digits** or bits. But the data which we deal with consists of numeric data and characters such as decimal digits 0 to 9, alphabets A to Z, arithmetic operators (e.g. +, -, etc.), relations operators (e.g. =, >, etc.), and many other special characters (e.g.;,@,{,],etc.). Thus, collection of eight bits is called a byte. Thus, one byte is used to represent one character internally. Most computers use two bytes or four bytes to represent numbers (positive and negative) internally. Another term, which is commonly used in computer, is a **Word**. A word may be defined as a unit of information, which a computer can process, or transfer at a time. A word, generally, is equal to the number of bits transferred between the central processing unit and the main memory in a single step. It ma also be defined as the basic unit of storage of integer data in a computer. Normally, a word may be equal to 8, 16, 32 or 64 bits .The terms like 32 bit computer, 64 bit computers etc. basically points out the word size of the computer.

1.3 VON NEUMANN ARCHITECTURE

Most of today's computer designs are based on concepts developed by John von Neumann referred to as the **VON NEUMANN ARCHITECTURE**. Von Neumann proposed that there should be a unit performing arithmetic and logical operation on the data. This unit is termed as Arithmetic Logic (**ALU**). One of the ways to provide instruction to such computer will be by connecting various logic components in such a fashion, that they produce the desired output for a given set of inputs. The process of connecting various logic components in specific configuration to achieve desired results is called **Programming**. This programming since is achieved by providing instruction within hardware by various connections is termed as **Hardwired**. But this is a very inflexible process of programming. Let us have a general configuration for arithmetic and logical functions. In such a case there is a need of a control signal, which directs the ALU to performed a specific arithmetic or logic function on the data. Therefore, in such a system, by changing the control signal the desired function can be performed on data.

---

[1] Representation of facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing by humans or by automatic means. Any representations such as characters or analog quantities to which meaning is or might be assigned

Any operation, which needs to be performed on the data, then can be obtained by providing a set of control signals. This, for a new operation one only needs to change the set of control signals.

But, how can these control signals by supplied? Let us try to answer this from the definition of a program. A program consists of a sequence of steps. Each of these steps, require certain arithmetic or logical or input/output operation to be performed on data. Therefore, each step may require a new set of control signals. Is it possible for us to provide a unique code for each set of control signals? Well the answer is yes. But what do we do with these codes? What about adding a hardware segment, which accepts a code and generates termed as Control Unit (CU). This, a program now consists of a sequence of codes. This machine is quite flexible, as we only need to provide a new sequence of codes for a new program. Each code is, in effect, and instruction, for the computer The hardware interprets each of these instructions and generates respective control signals,

The Arithmetic Logic Unit (ALU) and the Control Unit (CU) together are termed as the Central Processing Unit (CPU). The CPU is the moist important component of a computer's hardware. The ALU performs the arithmetic operations such as addition, subtraction, multiplication and division, and the logical operations such as: "Is A =B?" (Where A and B are both numeric or alphanumeric data), "Is a given character equal to M (for male) or F (for female)?" The control unit interprets instructions and produces the respective control signals.

All the arithmetic and logical Operations are performed in the CPU in special storage areas called registers. 'The size of the register is one of the important consideration in determining the processing capabilities of the CPU. Register size refers to the amount of information that can be held in a register at a time for processing. The larger the register size, the faster may be the speed o processing. A CPU's processing power is measured in Million Instructions Per Second (MIPS). The performance of the CPU was measured in milliseconds (one thousand of a second) on the first generation computers, in microseconds (one millionth of a second) on second-generation computers, and is expected to be measured in Pico-seconds (one 1000^{th} of a nano-second) in the later generations. How can the instruction and data be put into the computers? An external environment supplies the instruction and data, therefore, an input module is needed. The main responsibility of input module will be to put the data in the form of signals that can be recognized by the system. Similarly, we need another component, which will report the results in the results in proper format and form. This component is called output module. These components are referred together as input/output (I/O) components. In addition, to transfer the information, the computer system internally needs the system interconnections. Most common input/output devices are keyboard, monitor and printer, and the most common interconnection structure is the Bus structure.

Are these two components sufficient for a working computer? No, because input devices can bring instructions or data only sequentially and a program may not be executed sequentially as jump instructions are normally encountered in programming. In addition, more than one data elements may be required at a time. Therefore, a temporary storage area is needed in a computer to store temporarily the instructions and the data. This component is referred to as memory. It was pointed out by von-Neumann that the same memory can be used or storing data and instructions. In such cases the data can be treated as data on which processing can be performed, while instructions can be treated as data, which can be used for the generation of control signals.

The memory unit stores all the information in a group of memory cells, also called memory locations, as binary digits. Each memory location has a unique address and can be addressed independently. The contents of the desired memory locations are provided to the central processing unit by referring to the address of the memory location. The amount of information that can be held in the main memory is known as memory capacity. The capacity of the main memory s measured in Kilo Bytes (KB) or Mega Bytes (B). One-kilo byte stands for 2^{10} bytes, which are 1024 bytes (or approximately 1000 bytes). A mega byte stands for 2^{20} bytes, which is approximately little over one million bytes. When 64-bit CPU's become common memory will start to be spoken about in terabytes, petabytes, and exabytes.

- One kilobyte equals 2 to the 10^{th} power, or 1,024 bytes.
- One megabyte equals 2 to the 20^{th} power, or 1,048,576 bytes.
- One gigabyte equals 2 to the 30^{th} power, or 1,073,741,824 bytes.
- One terabyte equals 2 to the 40^{th} power, or 1,099511,627,776 bytes.
- One petabyte equals 2 to the 50^{th} power, or 1,125,899,906,842,624 bytes.
- One exabyte equals 2 to the 60^{th} power, or 1,152,921,504,606,846,976 bytes.
- One zettabyte equals 2 to the 70^{th} power, or 1,180,591,620,717,411,303,424
- One yottabyte equals 2 to the 80^{th} power, or 1,208,925,819,614,629,174,706,176

Note: There is some lack of standardization on these terms when applied to memory and disk capacity. Memory specifications tend to adhere to the definitions above whereas disk capacity specifications tend to simplify things to the 10^{th} power definitions (kilo=10^3, mega=10^6, giga=10^9, etc.) in order to produce even numbers.

Let us summarize the key features of a von Neumann machine.

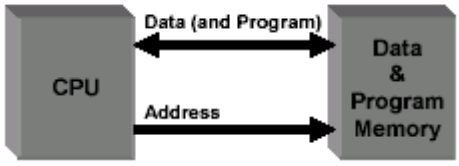
- The hardware of the von Neumann machine consists of a CPU, which includes an ALU and CU.
- A main memory system
- An Input/output system
- The von Neumann machine uses stored program concept, e.g., the program and data are stored in the same memory unit. The computers prior to this idea used

to store programs and data on separate memories. Entering and modifying these programs were very difficult as they were entered manually by setting switches and plugging and unplugging.

- Each location of the memory of von Neumann machine can be addressed independently.
- Execution of instructions in von Neumann machine is carried out in a sequential fashion (unless explicitly altered by the program itself) from one instruction to the next.

The following figure shows the basic structure of von Neumann machine. A von Neumann machine has only a single path between the main memory and control unit (CU). This feature/ constraint is refereed to as von Neumann bottleneck. Several other architectures have been suggested for modern computers

von Neumann Machine



- Attributed to John von Neumann
- Treats Program and Data equally
- One port to Memory . Simplified Hardware
- "von Neumann Bottleneck" (rate at which data and program can get into the CPU is limited by the bandwidth of the interconnect)

.

1.4 HISTORY OF COMPUTERS

Basic information about the technological development trends in computer in the past and its projections in the future. If we want to know about computers completely then we must start from the history of computers and look into the details of various technological and intellectual breakthrough. These are essential to give us the feel of how much work and effort has been done to get the computer in this shape.

The ancestors of modern age computer were the mechanical and electro-mechanical devices. This ancestry can be traced as back and 17th century, when the first machine capable of performing four mathematical operations, viz. addition, subtraction, division and multiplication, appeared.

1.4.1 MECHANICAL COMPUTERS

1.4.1.1. Pascaline:

Blaise Pascal made the very first attempt towards this automatic computing. He invented a device, which consisted of lots of gears and chains and used to perform repeated addition and subtractions. This device was called Pascaline. Later many attempts were made in this direction; we will not go in the details of these mechanical calculating devices. But we must discuss some details about the innovation by Charles babbage, the grandfather of modern computer. He designed two computers:

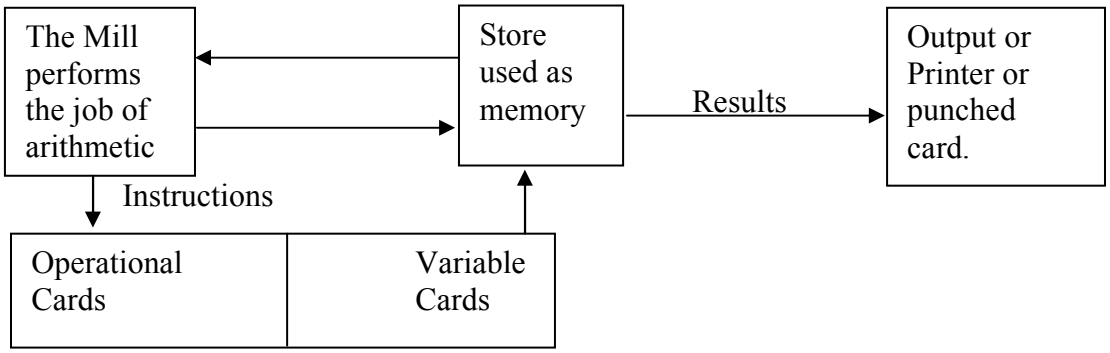
1.4.1.2. THE DIFFERENCE ENGINE

It was based on the mathematical principle of finite differences and was used to solve calculations on large numbers using a formula. It was also used for solving the polynomial and trigonometric functions.

1.4.1.3. THE ANALYTICAL ENGINE BY BABBAGE:

It was general purpose computing device, which could be used for performing any mathematical operation automatically. It consisted of the following components:

- **THE STORE:** A mechanical memory unit consisting of sets of counter wheels.
- **THE MILL:** An arithmetic unit, which is capable of performing the four basic arithmetic operations.
- **CARDS:** There are basically two types of cards:
    - **Operation Cards:** Selects one of four arithmetic operating by activating the mill to perform the selected function.
    - **Variable Cards:** Selects the memory locations to be used by the mill for a particular operation (i.e. the source of the operands and the destination of the results).
- **OUTPUT:** Could be directed to a printer or a cardpunch device.



Logical Structure of Babbage's Analytical Engine

The basic features of this analytical engine were:

- It was a general purpose programmable machine.
- It had the provision of automatic sequence control, thus enabling programs to alter its sequence of operations.
- The provision of sign checking of result existed.
- Mechanism for advancing or reversing of control card were permitted thus enabling exceution of any desired instruction. In other words, Babbage had deviced a conditional and branching instructions, the babbage machine is fundamentally the same as modern computer. Unfortunately Babbage work could not be completed. But as a tribute to Charles Babbage his Analytical Engine was completed in the last decade and is now on display at the science Museum at London.

Next notable attempts towards computer were electromechanical Zuse used electromechanical relays that could be either opened or closed automatically. Thus, the use of binary digits, rather than decimal numbers started.

1.4.1.4. HARVARD MARK I AND THE BUG

The next significant effort towards devising an electromechanical computer was made at the harvard University, jointly sponsered by IBM and the Department of UN Navy, Howard Aiken of Harvard University developed a system called Mark I in 1944. Mark I was decimal machine.

Some of you must have heard a term call **"bug"**. It is mainly used to indicate errors in computer programs. This term was coined, when one day, a program in Mark I did not run properly due to a short-circuiting the computer. Since then, the "bug" has been linked with errors or problems in computer programming. The process of eliminating error in a program is thus, known as **"debugging"**.

The basic drawback of these mechanical and elecromechanical computers were:

- Friction/inertia of moving components had limited the speed.
- The data movement using gears and liner was quite difficult and unreliable.
- The change was to have switching and storing mechanism with no moving parts and then the electronic switching technique "triode" vacuum tubes were used and hence born the first electronic computer.
- The evolution of digital computing is often divided into *generations*. Each generation is characterized by dramatic improvements over the previous generation in the technology used to build computers, the internal organization of computer systems, and programming languages. Although not usually associated with computer generations, there has been a steady improvement in algorithms, including algorithms used in computational science. The following history has been organized using these widely recognized generations as mileposts.

1.4.2 First Generation Electronic Computers (1937-1953)

Three machines have been promoted at various times as the first electronic computers. These machines used electronic switches, in the form of vacuum tubes, instead of electromechanical relays. In principle the electronic switches would be more reliable, since they would have no moving parts that would wear out, but the technology was still new at that time and the tubes were comparable to relays in reliability. Electronic components had one major benefit, however: they could ``open'' and ``close'' about 1,000 times faster than mechanical switches.

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. Eckert, Mauchly, and John von Neumann, a consultant to the ENIAC project, began work on a new machine before ENIAC was finished. The main contribution of EDVAC, their new project, was the notion of a *stored program*. There is some controversy over who deserves the credit for this idea, but none over how important the idea was to the future of general purpose computers. ENIAC was controlled by a set of external switches and dials; to change the program required physically altering the settings on these controls. These controls also limited the speed of the internal electronic operations. Through the use of a memory that was large enough to hold both instructions and data, and using the program stored in memory to control the order of arithmetic operations, EDVAC was able to run orders of magnitude faster than ENIAC. 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.

The trends, which were encountered during the era of first generation computer, were:

- The first generation computer control was centralized in a single CPU, and all operations required a direct intervention of the CPU.
- Use of ferrite-core main memory was started during this time.
- Concepts such as use of virtual memory and index register (you will know more about these terms in advanced courses).
- Punched cards were used as input device.
- Magnetic tapes and magnetic drums were used as secondary memory.
- Binary code or machine language was used for programming.
- Towards the end due to difficulties encountered in use of machine language as programming language, the use of symbolic language, which is now called assembly language, started.
- Assembler, a program, which translates assembly language programs to machine language, was made.
- Computer was accessible to only one programmer at a time (single user environment).

- Advent of Von-Neumann Architecture.

1.4.3 Second Generation (1954-1962)

The second generation saw several important developments at all levels of computer system design, 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-0 at MIT's Lincoln Laboratory.** Memory technology was based on magnetic cores which could be accessed in random order, as opposed to mercury delay lines, in which data was stored as an acoustic wave that passed sequentially through the medium and could be accessed only when the data moved by the I/O interface.

During this second generation many high level programming languages were introduced, including FORTRAN (1956), ALGOL (1958), and COBOL (1959). Important commercial machines of this era include the IBM 704 and its successors, the 709 and 7094. The latter introduced I/O processors for better throughput between I/O devices and main memory.
The second generation also saw the first two supercomputers designed specifically for numeric processing in scientific applications. The term ``supercomputer'' is generally reserved for a machine that is an order of magnitude more powerful than other machines of its era. Two machines of the 1950s deserve this title. The Livermore Atomic Research Computer (LARC) and the IBM 7030 (aka Stretch) were early examples of machines that overlapped memory operations with processor operations and had primitive forms of parallel processing

1.4.4 Third Generation (1963-1972)

The third generation brought huge gains in computational power. Innovations in this era include the use of integrated circuits, or ICs (semiconductor devices with several transistors built into one physical component), semiconductor memories starting to be used instead of magnetic cores, microprogramming as a technique for efficiently designing complex processors, the coming of age of pipelining and other forms of parallel processing, and the introduction of operating systems and time-sharing.

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. Computer designers began to take advantage of parallelism by using multiple functional units, overlapping CPU and I/O operations, and pipelining (internal parallelism) in both the

instruction stream and the data stream. The SOLOMON computer, developed by Westinghouse Corporation, and the ILLIAC IV, jointly developed by Burroughs, the Department of Defense and the University of Illinois, was representative of the first parallel computers.

1.4.5. Fourth Generation (1972-1984)

The next generation of computer systems saw the use of **large scale integration (LSI** - 1000 devices per chip) and **very large scale integration (VLSI** - 100,000 devices per chip) in the construction of computing elements. At this scale entire processors will fit onto a single chip, and for simple systems the entire computer (processor, main memory, and I/O controllers) can fit on one chip. Gate delays dropped to about 1ns per gate.

Semiconductor memories replaced core memories as the main memory in most systems; until this time the use of semiconductor memory in most systems was limited to registers and cache. A variety of parallel architectures began to appear; however, during this period the parallel computing efforts were of a mostly experimental nature and most computational science was carried out on vector processors. Microcomputers and workstations were introduced and saw wide use as alternatives to time-shared mainframe computers.

Developments in software include very high level languages such as FP (functional programming) and Prolog (programming in logic). These languages tend to use a *declarative* programming style as opposed to the *imperative* style of Pascal, C, FORTRAN, et al. In a **declarative style**, a programmer gives a mathematical specification of what should be computed, leaving many details of how it should be computed to the compiler and/or runtime system. These languages are not yet in wide use, but are very promising as notations for programs that will run on massively parallel computers (systems with over 1,000 processors). Compilers for established languages started to use sophisticated optimization techniques to improve code, and compilers for vector processors were able to vectorize simple loops (turn loops into single instructions that would initiate an operation over an entire vector).

Two important events marked the early part of the third generation: the development of the C programming language and the UNIX operating system, both at Bell Labs. In 1972, Dennis Ritchie, seeking to meet the design goals of CPL and generalize Thompson's B, developed the C language. Thompson and Ritchie then used C to write a version of UNIX for the DEC PDP-11. This C-based UNIX was soon ported to many different computers, relieving users from having to learn a new operating system each time they change computer hardware. UNIX or a derivative of UNIX is now a de facto standard on virtually every computer system.

1.4.6 Fifth Generation (1984-1990)

The development of the next generation of computer systems is characterized mainly by the acceptance of parallel processing. Until this time parallelism was limited to pipelining and vector processing, or at most to a few processors sharing jobs. The fifth generation saw the introduction of machines with hundreds of processors that could all be working on different parts of a single program.

Other new developments were the widespread use of computer networks and the increasing use of single-user workstations. Prior to 1985 large scale parallel processing was viewed as a research goal, but two systems introduced around this time are typical of the first commercial products to be based on parallel processing. The Sequent Balance 8000 connected up to 20 processors to a single shared memory module (but 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.

The Intel iPSC-1, nicknamed ``the hypercube'', took a different approach. Instead of using one memory module, Intel 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 bottleneck and large systems (using more processors) could be built. Toward the end of this period a third type of parallel processor was introduced to the market. In this style of machine, known as a *data-parallel* or SIMD, there are several thousand very simple processors. All processors work under the direction of a single control unit; i.e. if the control unit says ``add a to b'' then all processors find their local copy of a and add it to their local copy of b.

Scientific computing in this period was still dominated by vector processing. Most manufacturers of vector processors introduced parallel models, but there were very few (two to eight) processors in this parallel machines. In the area of computer networking, both wide area network (WAN) and local area network (LAN) technology developed at a rapid pace, stimulating a transition from the traditional mainframe computing environment toward a distributed computing environment in which each user has their own workstation for relatively simple tasks (editing and compiling programs, reading mail) but sharing large, expensive resources such as file servers and supercomputers. RISC technology (a style of internal organization of the CPU) and plummeting costs for RAM brought tremendous gains in computational power of relatively low cost workstations and servers. This period also saw a marked increase in both the quality and quantity of scientific visualization.

1.4.7. Sixth Generation (1990 -)

This generation is beginning with many gains in parallel computing, both in the hardware area and in improved understanding of how to develop algorithms to exploit diverse, massively parallel architectures. Parallel systems now complete with vector

processors in terms of total computing power and most expect parallel systems to dominate the future.

Combinations of parallel/vector architectures are well established, and one corporation (Fujitsu) has announced plans to build a system with over 200 of its high end vector processors. Workstation technology has continued to improve, with processor designs now using a combination of RISC, pipelining, and parallel processing. As a result it is now possible to purchase a desktop workstation for about $30,000 that has the same overall computing power (100 megaflops) as fourth generation supercomputers.

One of the most dramatic changes in the sixth generation will be the explosive growth of wide area networking. Network bandwidth has expanded tremendously in the last few years and will continue to improve for the next several years. T1 transmission rates are now standard for regional networks, and the national ``backbone'' that interconnects regional networks uses T3. Networking technology is becoming more widespread than its original strong base in universities and government laboratories as it is rapidly finding application in K-12 education, community networks and private industry.

1.5 CLASSIFICATION COMPUTERS

1.5.1 MICRO COMPUTER

A microcomputer's CPU is microprocessor. The microcomputer originated in late 1970s. the first microcomputers were built around 8-bit microprocessor chips. It means that the chip can retrieve instructions/data from storage, manipulate, and process an 8-bit data at a time or we can say that the chip has a built-in 8-bit data transfer path. An improvement on 8-bit chip technology was seen in early 1980s, when a series of 16-bit chips namely 8086 and 8088 were introduced by Intel Corporation, each one with an advancement over the other. 8088 is a 8/16 bit chip i.e. an 8-bit path is used to move data between chip and primary storage(external path), at a time, but processing is done within the chip using a 16-bit path(internal path) at a time. 8086 is a 16/16 bit chip i.e. the internal and external paths both are 16 bit wide. Both these chips can support a primary storage capacity of upto 1 mega byte (MB). These computers are usually divided into desktop models and laptop models. They are terribly limited in what they can do when compared to the larger models discussed above because they can only be used by one person at a time, they are much slower than the larger computers, and they cannot store nearly as much information, but they are excellent when used in small businesses, homes, and school classrooms. These computers are inexpensive and easy to use. They have become an indispensable part of modern life. Thus

- Used for memory intense and graphic intense applications
- Are single-user machines

1.5.2 MINI COMPUTER

Minicomputers are much smaller than mainframe computers and they are also much less expensive. The cost of these computers can vary from a few thousand dollars to several hundred thousand dollars. They possess most of the features found on mainframe computers, but on a more limited scale. They can still have many terminals, but not as many as the mainframes. They can store a tremendous amount of information, but again usually not as much as the mainframe. Medium and small businesses typically use these computers. Thus

- Fit somewhere between mainframe and PCs
- Would often be used for file servers in networks

1.5.3. MAINFRAME COMPUTER

Mainframe computers are very large, often filling an entire room. They can store enormous of information, can perform many tasks at the same time, can communicate with many users at the same time, and are very expensive. . The price of a mainframe computer frequently runs into the millions of dollars. Mainframe computers usually have many terminals connected to them. These terminals look like small computers but they are only devices used to send and receive information from the actual computer using wires. Terminals can be located in the same room with the mainframe computer, but they can also be in different rooms, buildings, or cities. Large businesses, government agencies, and universities usually use this type of computer. Thus

- Most common type of large computers
- Used by many people using same databases
- Can support many **terminals**
- Used in large company like banks and insurance companies

1.5.4. SUPER COMPUTER

The upper end of the state of the art mainframe machine is the supercomputer. These are amongst the fastest machines in terms of processing speed and use multiprocessing techniques, were a number of processors are used to solve a problem. Computers built to minimize distance between points for very fast operation. Used for extremely complicated computations. Thus

- Largest and most powerful
- Used by scientists and engineers
- Very expensive
- Would be found in places like Los Alamos or NASA

1.6 INSTRUCTION EXECUTION

Paper Name: Computer Organization and Architecture

We know that the basic function performed by a computer is the execution of a program. The program, which is to be executed, is a set of instructions, which are stored in memory. The central processing unit (CPU) executes the instructions of the program to complete a task. The instruction execution takes place in the CPU registers. Let us, first discuss few typical registers, some of which are commonly available in of machines.
These registers are:-

- **Memory Address Register (MAR):-** Connected to the address lines of the system bus. It specifies the address of memory location from which data or instruction is to be accessed (for read operation) or to which the data is to be stored (for write operation).
- **Memory Buffer Register (MBR):-** Connected to the data lines of the system bus. It specifies which data is to be accessed(for read operation) or to which data is to be stored (for write operation).
- **Program Counter (PC):-** Holds address of next instruction to be fetched, after the execution of an on-going instruction.
- **Instruction Register (IR):-** Here the instruction are loaded before their execution or holds last instruction fetched.

Instruction Cycle

The simplest model of instruction processing can be a two step process. The CPU reads (fetches) instructions (codes) from the memory one at a time, and executes. Instruction fetch involves reading of an instruction from a memory location to the CPU register. The execution of this instruction may involve several operations depending on the nature of the instruction. Thus to execute an instruction, a processor must go through two sub-cycles:

Paper Name: Computer Organization and Architecture

UNIT-2

REGISTER TRANSFER AND MICRO OPERATIONS

2.1 Introduction To Register Transfer

A micro operations is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may by transferred to another register.

The symbolic notation used to describe the micro operation transfer among registers is called a register transfer language. The term “register transfer” implies the availability of hardware logic circuits that can perform stated micro operation and transfer the results to the operation to the same or another register.
Register Transfer

We designate computer registers by capital letters to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register, represented by MAR. Other examples are PC (for program counter), IR (for instruction register) and R1 (for processor register). We show the individual flip-flops in an n-bit register by giving numbers them in sequence from 0 through n - 1, starting from 0 in the right most position and increasing the numbers toward the left.

A 16-bit register is divided into two halves. Low byte (Bits 0 through 7) is assigned the symbol L and high byte (Bits 8 through 15) is assigned the symbol H. The name of a 16-bit register is PC. The symbol PC(L) represents the low order byte and PC(H) designates the high order byte. The statement R_2 □ R_1 refers the transfer of the content of register R_1 into register R_2. It should be noted that the content of the source register R_1 does not

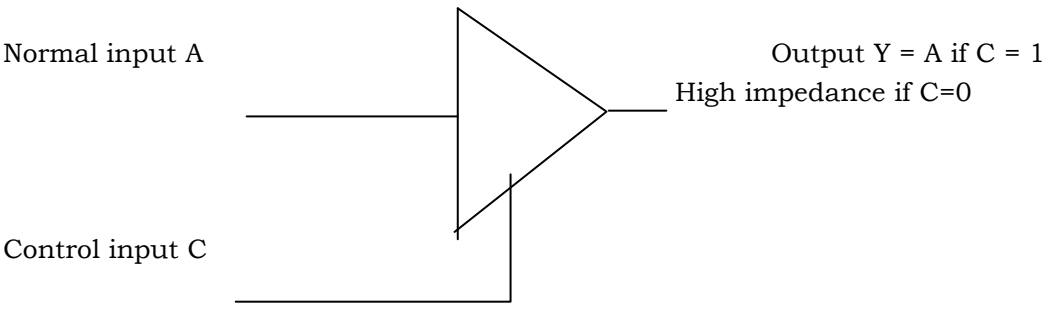
change after the transfer. In real applications, the transfer occurs only under a predetermined control condition. This can be shown by means of an "if-then" statement:
If P=1 then R_2 □□ R_1

where P is a control signal generated in the control section of the system. For convenience we separate the control variables from the register transfer operation by specifying a control function. A control function is a Boolean variable that is equal to 1 or 0. The control function is written as follows:
P: R_2 □□□ R_1
Bus

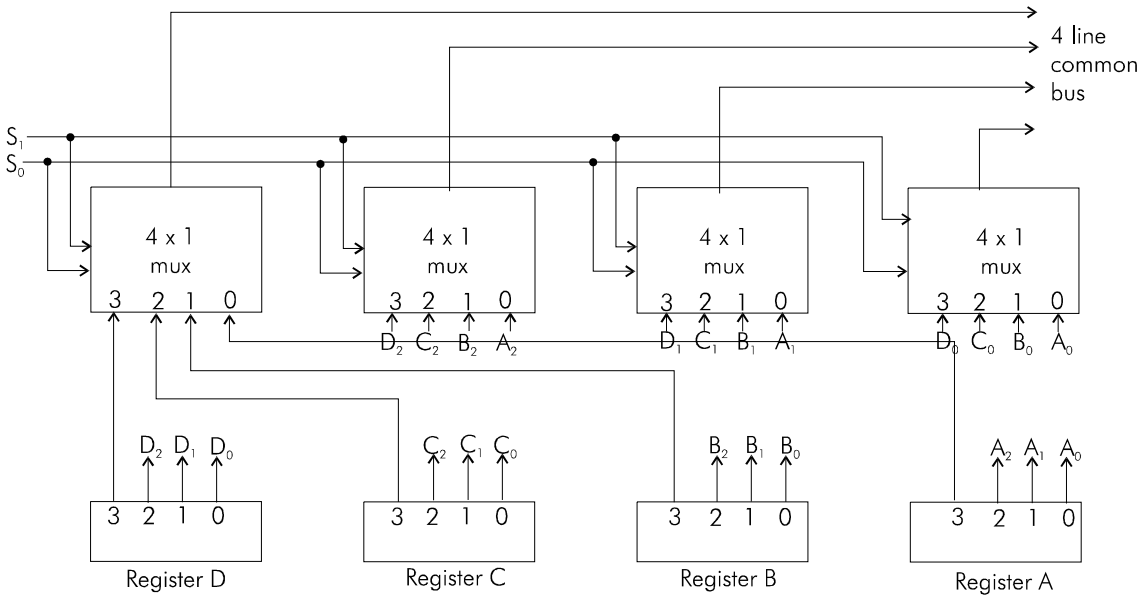
Since a computer has many registers, paths must be provided to transfer information from one register to another. If separate lines are used between each register and all other registers, number of wires will be excessive in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

A common bus system can be constructed using multiplexers. These multiplexers select the source register whose binary information is then placed on the bus. A bus system will multiplex registers of a bit each to produce an n-line common bus. The number of multiplexers required to construct the bus is equal to n, where n is the number of bits in each register. The size of each multiplexer must be k × 1 since it multiplexes k data lines. A bus system can be constructed with 'three-state gates' instead of multiplexers. A three-state gate is a digital circuit that shows three states. Two of the states are equivalent to logic 1 and 0. The third state is a high impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance. The one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a three state buffer gate is shown in the figure given below. The control input determines the output.



The construction of a bus system for four registers is shown in the figure in on the next page.



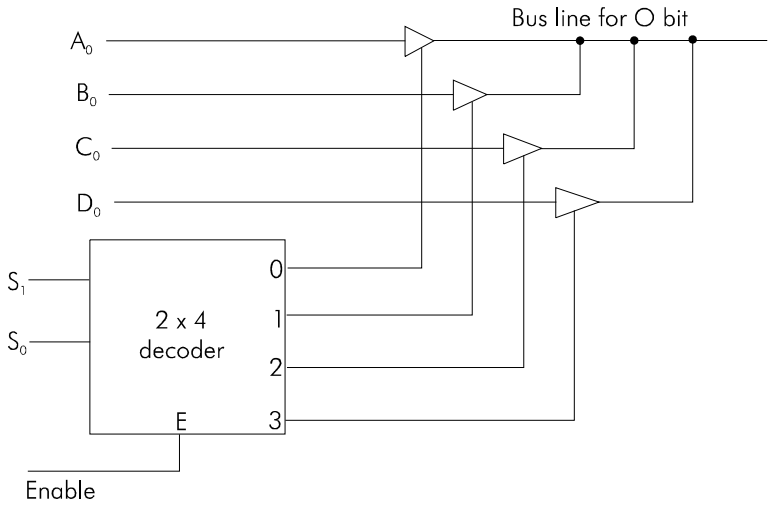
The function table of the above bus system is

S_1	S_0	Register collected
0	0	A
0	1	B
1	0	C
1	1	D

Three state table buffers

Three state table buffers: A bus system can be constructed with three state gates instead of multiplexers. A three states gate is digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state. The high-impedance states behaves like an open circuit, which means that the output is disconnected and does not have a logic, such as AND or NAND. However the one most commonly used in the design of a bus system is the buffer gate.

The construction of a bus system with ***three state table buffers*** is shown in the following figure:



2.2 Bus And Memory Transfer

A read operation implies transfer of information to the outside environment from a memory word, whereas storage of information into the memory is defined as write operation. Symbolizing a memory word by the letter M, it is selected by the memory address during the transfer which is a specification for transfer operations. The address is specified by enclosing it in square brackets following the letter M.

For example, the read operation for the transfer of a memory unit M from an address register AR to another data register DR can be illustrated as:
Read: DR ←M[AR]

The write operation transfer the contents of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address in the AR. The write operation can be stated symbolic as follows:

Write: M[AR] ← R1
This cause a transfer on information from R1 into the memory word M selected by the address in AR.

2.3 Micro-Operations

A micro-operation is an elementary operation which is performed on the data stored in registers. We can classify the micro-operations into four categories:

1. Register transfer: transfer binary information from one register to another.
2. Arithmetic: perform arithmetic operations on numeric data stored in registers.

3. Logic: perform bit manipulation operation on non-numeric data stored in registers.
4. Shift: perform shift operations on data stored in registers.

2.3.1 Arithmetic Micro-operations

These micro-operations perform some basic arithmetic operations on the numeric data stored in the registers. These basic operations may be addition, subtraction, incrementing a number, decrementing a number and arithmetic shift operation. An 'add' micro-operation can be specified as:

$R_3 \square R_1 + R_2$

It implies: add the contents of registers R_1 and R_2 and store the sum in register R_3.
The add operation mentioned above requires three registers along with the addition circuit in the ALU.

Subtraction, is implemented through complement and addition operation as:

$R_3 \square R_1 – R_2$ is implemented as
$R_3 \square R_1 +$ (2's complement of R_2)
$R_3 \square R_1 +$ (1's complement of $R_2 + 1$)
$R_3 \square R_1 + R_2 + 1$

An increment operation can be symbolized as:

$R_1 \square R_1 + 1$

while a decrement operation can be symbolized as:

$R_1 \square R_1 – 1$

We can implement increment and decrement operations by using a combinational circuit or binary up/down counters. In most of the computers multiplication and division are implemented using add/subtract and shift micro-operations. If a digital system has implemented division and multiplication by means of combinational circuits then we can call these as the micro-operations for that system. An arithmetic circuit is normally implemented using parallel adder circuits. Each of the multiplexers (MUX) of the given circuit has two select inputs. This 4-bit circuit takes input of two 4-bit data values and a carry-in-bit and outputs the four resultant data bits and a carry-out-bit. With the different input values we can obtain various micro-operations.

Equivalent micro-operation	Micro-operation name
$R \square R_1 + R_2$	Add
$R \square R_1 + R_2 +1$	Add with carry
$R \square R_1 + R_2$	Subtract with borrow
$R \square R_1 +$ 2's	Subtract
$R \square R_1$	Transfer
$R \square R_1 + 1$	Increment
$R \square R_1 – 1$	Decrement

2.3.2 Logic Micro-operations

These operations are performed on the binary data stored in the register. For a logic micro-operation each bit of a register is treated as a separate variable.
For example, if R_1 and R_2 are 8 bits registers and

R_1 contains 10010011 and
R_2 contains 01010101
R_1 AND R_2 00010001

Some of the common logic micro-operations are AND, OR, NOT or complements. Exclusive OR, NOR, NAND.

We can have four possible combinations of input of two variables. These are 00, 01, 10 and 11. Now, for all these 4 input combination we can have 2^4 = 16 output combinations of a function. This implies that for two variables we can have 16 logical operations.

Logic Micro Operations

SELECTIVE SET
The selective-set operation sets to 1 the bits in register A where there are corresponding 1's in register B. it does not affect bit positions that have 0's in B. the following numerical example clarifies this operation:-

```
1010   A before
1100   B (logic operand)
----
1110   A after
```

SELECTIVE COMPLEMENT
The selective-complement operation complements bits in register A where there are corresponding 1's in register B. it does not affect bit positions that have 0's in B. the following numerical example clarifies this operation:-

```
1010          A before
1100   B (logic operand)
----
0110   A after
```

SELECTIVE CLEAR
The selective-clear operation clears to 0 the bits in register A only where there are corresponding 1's in register B. For example:-

1010 A before
1100 B (logic operand)
0010 A after

MASK OPERATION

The mask operation is similar to the selective-clear operation except that thebits of A are cleared only where there are corresponding 0's in B. the mask operation is an AND micro operation, for example:-

1010 A before
1100 B (logic operand)
1000 A after masking

INSERT OPERATION

The insert operation inserts a new value into a group of bits. This is done by first masking the bits and then Oring them with the required value. For example, suppose that an A register contains eight bits, 0110 1010. to replace the four leftmost bits by the value 1001 we first the four unwanted bits:-

0110 1010 A before
0000 1111 B (mask)
0000 1010 A after masking

and then insert the new value:-

0000 1010 A before
1001 0000 B (insert)
1001 1010 A after insertion

the mask operation is an AND microoperation and the insert operation is an OR microoperation.

CLEAR OPERATION

The clear operation compares the words in A and B and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as has own by the following example:

1010 A
1010 B
0000 A $\leftarrow$ A $\oplus$ B

When A and B are equal, the two corresponding bits are either both 0 or both 1. in either case the exclusive-OR operation produces a 0. the all-0's result is then checked to determine if the tow numbers were equal.

2.3.4 Shift Microoperations

Shift microoperation can be used for serial transfer of data. They are used generally with the arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. During a shift-right operation the serial input transfers a bit into the leftmost position. The serial input transfers a bit into the rightmost position during a shift-left operation. There are three types of shifts, logical, circular and arithmetic.

Logical shift

A logical shift operation transfers 0 through the serial input. We use the symbols *shl* and *shr* for logical shift left and shift right microoperations, e.g.

R1 ← shl R1

R2 ← shr R2

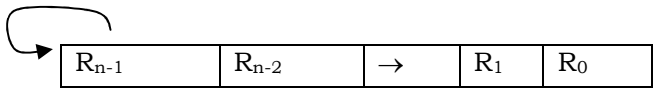
are the two micro operations that specify a 1-bit shift left of the content of register R1 and a 1- bit shift right of the content of register R2.

Circular shift

The circular shift is also known as rotate operation. It circulates the bits of the register around the two ends and there is no loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We use the symbols *cil* and *cir* for the circular shift left and circular shift right. E.g. suppose Q1 register contains 01101101 then after cir operation, it contains 0110110 and after cil operation it will contain 11011010.

Arithmetic Shift

An arithmetic shift micro operation shifts a signed binary number to the left or right. The effect of an arithmetic shift left operation is to multiply the binary number by 2. Similarly an arithmetic shift right divides the number by 2. Because the sign of the number must remain the same arithmetic shift-right must leave the sign bit unchanged, when it is multiplied or divided by 2. The left most bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Following figure shows a typical register of n bits.



Bit R_{n-1} in the left most position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bits) to the right. Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1}, and so on for other bits in the register.

Paper Name: Computer Organization and Architecture

Paper Name: Computer Organization and Architecture

UNIT 3

PROGRAMMING ELEMENTS

3.1 Computer Arithmetic

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations.

To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms.

In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

A processor has an arithmetic processor(as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2's complement. Most computers use the signed magnitude representation for the mantissa.

3.2 Addition and Subtraction with Signed –Magnitude Data

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 4.1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.
The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm).

Table 4.1: Addition and Subtraction of Signed-Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		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 – B)	+ (A – B)
(+ A) – (– B)	+ (A + B)			
(–A) – (+B)	– (A + B)			
(–A) – (–B)		– (A – B)	+ (B – A)	+ (A – B)

Algorithm

When the signs of A and B are same, add the two magnitudes and attach the sign of result is that of A. When the signs of A and B are not same, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A, if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result will be positive.

3.3 Multiplication Algorithms

Multiplication of two fixed-point binary numbers in signed magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example:

```
      23       10111  Multiplicand
      19   x   10011  Multiplier
    -------------------
           10111
          10111
         00000
        00000
       10111
-------------------------
437    110110101 Product
```

This process looks at successive bits of the multiplier, least significant bit first. If the multiplier bit is 1, the multiplicand is copied as it is; otherwise, we copy zeros. Now we shift numbers copied down one position to the left from the previous numbers. Finally, the numbers are added and their sum produces the product.

Hardware Implementation for signed-magnitude data

When multiplication is implemented in a digital computer, we change the process slightly. Here, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers, and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. Now, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

The hardware for multiplication consists of the equipment given in Figure 4.5. The multiplier is stored in the register and its sign in Q_s. The sequence counter SC is initially set bits in the multiplier. After forming each partial product the counter is decremented. When the content of the counter reaches zero, the product is complete and we stop the process.

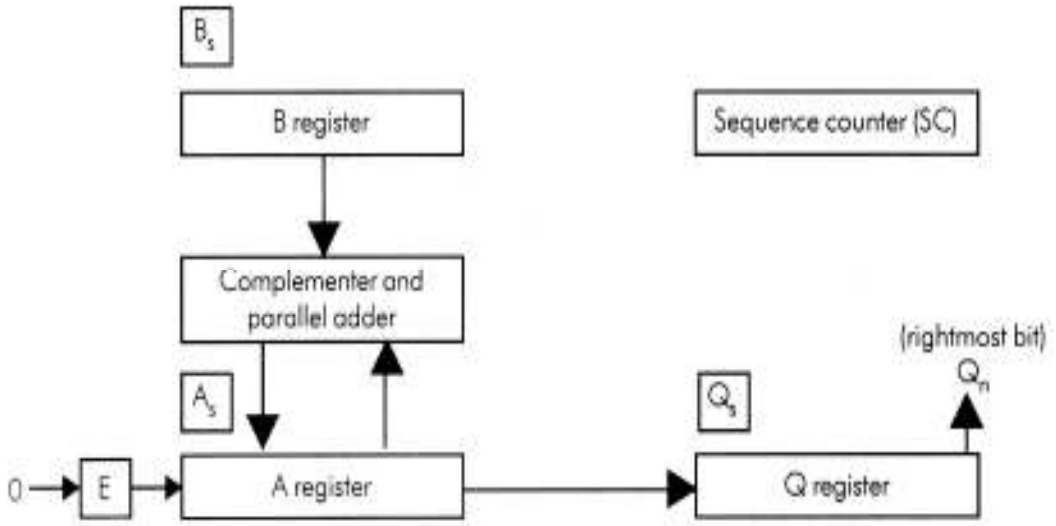


Figure 4.5: Hardware for Multiply Operation

3.3.1 Booth Multiplication Algorithm

If the numbers are represented in signed 2's complement then we can multiply them by using Booth algorithm. In fact the strings of 0's in the multiplier need no addition but just shifting, and a string of l's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$. For example, the binary number 001111 (+15) has a string of 1's from 2^3 to 2^0(k = 3, m = 0).

Table 4.2: Numerical Example for Binary Multiplier

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Q_n = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Q_n = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	1	00001	01100	011
Q_n = 0; shift right EAQ	0	01000	10110	010
Q_n = 0; shift right EAQ	0	00100	01011	001
Qn = 1; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	11011		

Final product in AQ = 0110110101

The number can be represented as $2^{k+1} - 2^m = 2^4 - 2^0 = 16 - 1 = 15$. Therefore, the multiplication M x 14, where M is the multiplicand and 14 the multiplier may be computed as M x 2^4 - M x 2^1. That is, the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Booth algorithm needs examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand added to the partial product, subtracted from the partial product, or left unchanged by the following rules:

1. The multiplicand is subtracted from the partial product when we get the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product when we get the first Q (provided that there was a previous 1) in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is the same as the previous multiplier bit.

The algorithm applies to both positive and negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of l's and the last operation will be a subtraction of the appropriate weight. For example, a multiplier equal to -14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 - 2^1 = -14$.

The hardware implementation of Booth algorithm requires the register configuration shown in Figure 4.7(a). Q_n represents the least significant bit of the multiplier in register QR. An extra flip-flop Q_{n+1} is appended to QR to provide a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Figure 4.7(b). AC and the appended bit Q_{n+1} are initially set to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Q_n and Q_{n+1} are inspected. If the two bits are 10, it means that the first 1 in a string of 1's has been encountered. This needs a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01. It means that the first 0 in a string of 0's has been encountered. This needs the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. Hence, the two numbers that are added always have opposite sign, a condition that excludes an overflow. Next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC same The sequence counter decrements and the computational loop is repeated n times.

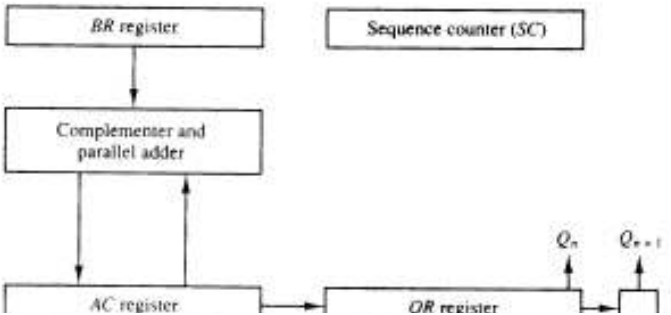


Figure 4.7(a): Hardware for Booth Algorithm

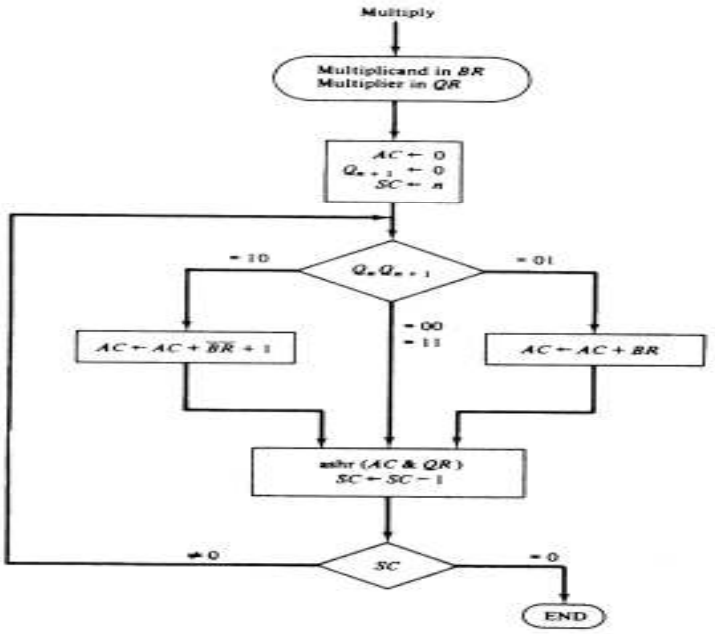


Figure 4.7(b)

A numerical example of Booth algorithm is given in Table 4.3 for n = 5. It gives the multiplication of (-9) x (-13) = +117. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC. The final value of Q_{n+1} is the original sign bit of the multiplier and should not be taken as part of the product.

Table 4.3: Example of Multiplication with Booth Algorithm

Qn Qn+1	BR = 10111 $\overline{BR + 1} = 01001$	AC	QR	Q_{n+1}	SC
1 0	Initial	00000	10011	0	101
	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			

ashr	0011	10101	1	000

3.3.2 Array Multiplier

To check the bits of the multiplier one at a time and forming partial products is a sequential operation requiring a sequence of add and shift micro-operations. The multiplication of two binary numbers can be done with one micro-operation by using combinational circuit that forms the product bits all at once.

This is a fast way since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and so it is not an economical unit for the development of ICs.

Now we see how an array multiplier is implemented with a combinational circuit. Consider the multiplication of two 2-bit numbers as shown in Fig. 4.8. The multiplicand bits are b_1 and b_0, the multiplier bits are a_1 and a_0, and the product is c_3 c_2 c_1 c_0. The first partial product is obtained by multiplying a_0 by b_1b_0. The multiplication of two bits gives a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can we implement it with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a_1 by b_1b_0 and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need j * k AND gates and (j – 1) k-bit adders to produce a product of j + k bits.

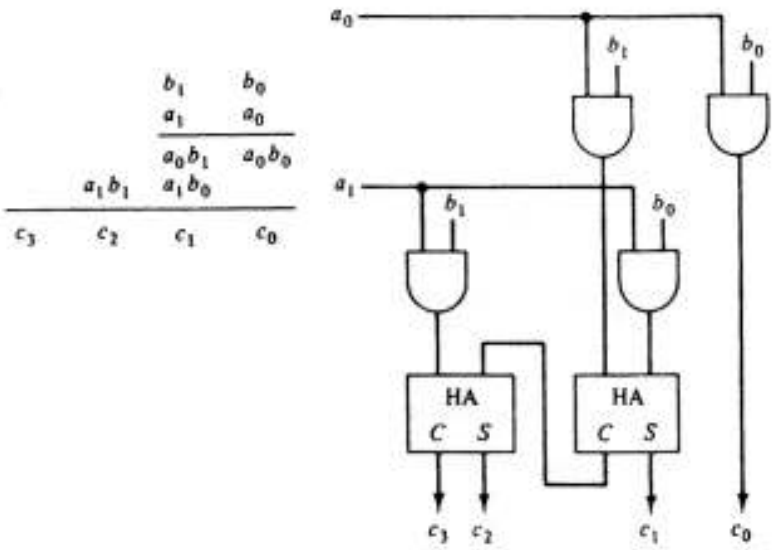


Figure 4.8: 2-bit by 2-bit array multiplier

As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by $b_3b_2b_1b_0$ and the multiplier by $a_2a_1a_0$. Since k=4 and j=3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Figure 4.9.

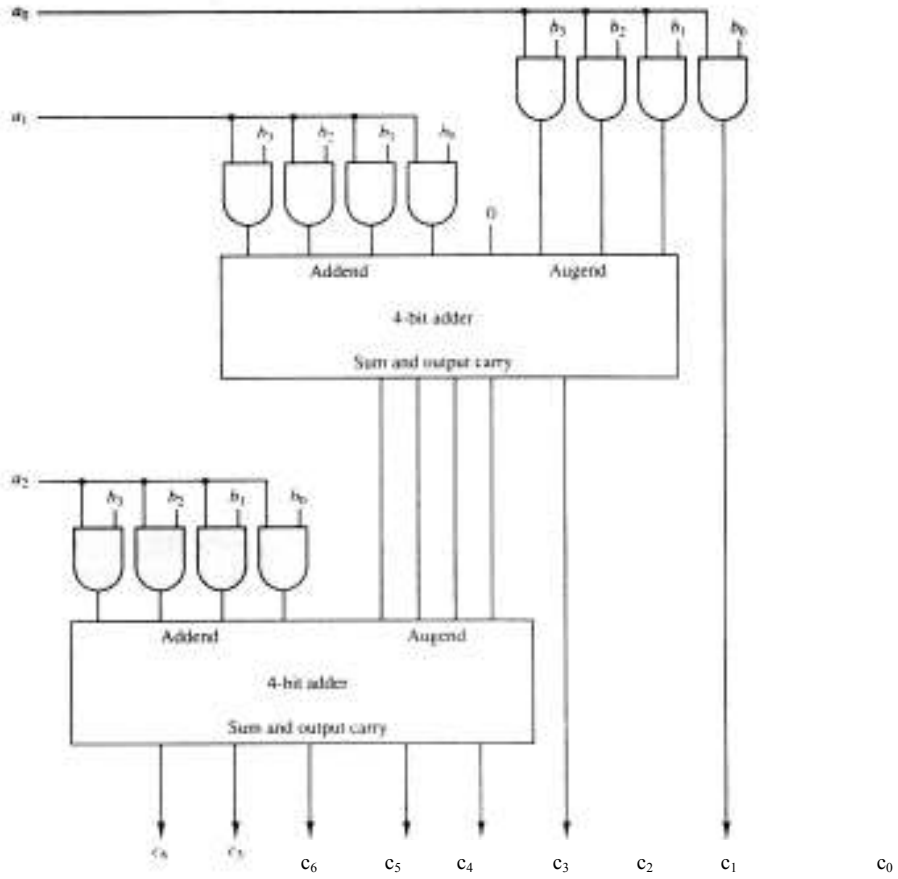


Figure 4.9: 4-bit by 3-bit array multiplier

3.3.3 Division Algorithms

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is described in Figure 4.10. The divisor B has five bits and the dividend A has ten.

Division: B = 10001	11010 ~~0111000~~00 01110 011100 - 10001 - 010110 - -10001 - - 001010 - - - 010100 - - - -10001 - - - -000110 - - - - -00110	Quotient = Q Dividend = A 5 bits of A < B, quotient has 5 bits 6 bits of A □□B Shift right B and subtract; enter 1 in Q 7 bits of remainder □ B Shift right B and subtract; enter 1 in Q Remainder < B; enter 0 in Q; shift right B Remainder □ B Shift right B and subtract; enter 1 in Q Remainder < B; enter 0 in Q Final remainder

Figure 4.10: Example of Binary Division

The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.

Hardware Implementation for Signed-Magnitude Data

In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.

The hardware required is identical to that of multiplication. Register EAQ is now shifted to the left with 0 inserted into Q_n and the previous value of E is lost. The example is given in Figure 4.10 to clear the proposed division process. The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. E

keeps the information about the relative magnitude. A quotient bit 1 is inserted into Q_n and the partial remainder is shifted to the left to repeat the process when E = 1. If E = 0, it signifies that A < B so the quotient in Q_n remains a 0 (inserted during the shift). To restore the partial remainder in A the value of B is then added to its previous value. The partial remainder is shifted to the left and the process is repeated again until we get all five quotient-bits. Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and A has the final remainder.
Before showing the algorithm in flowchart form, we have to consider the sign of the result and a possible overflow condition. The sign of the quotient is obtained from the signs of the dividend and the divisor. If the two signs are same, the sign of the quotient is plus. If they are not identical, the sign is minus. The sign of the remainder is the same as that of the dividend.

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

Figure 4.11: Example of Binary Division with Digital Hardware

3.3.3.1 Hardware Algorithm

Figure 4.6 is a flowchart of the hardware multiplication algorithm. In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

Now, the low order bit of the multiplier in Q_n is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. When SC = 0 we stops the process.

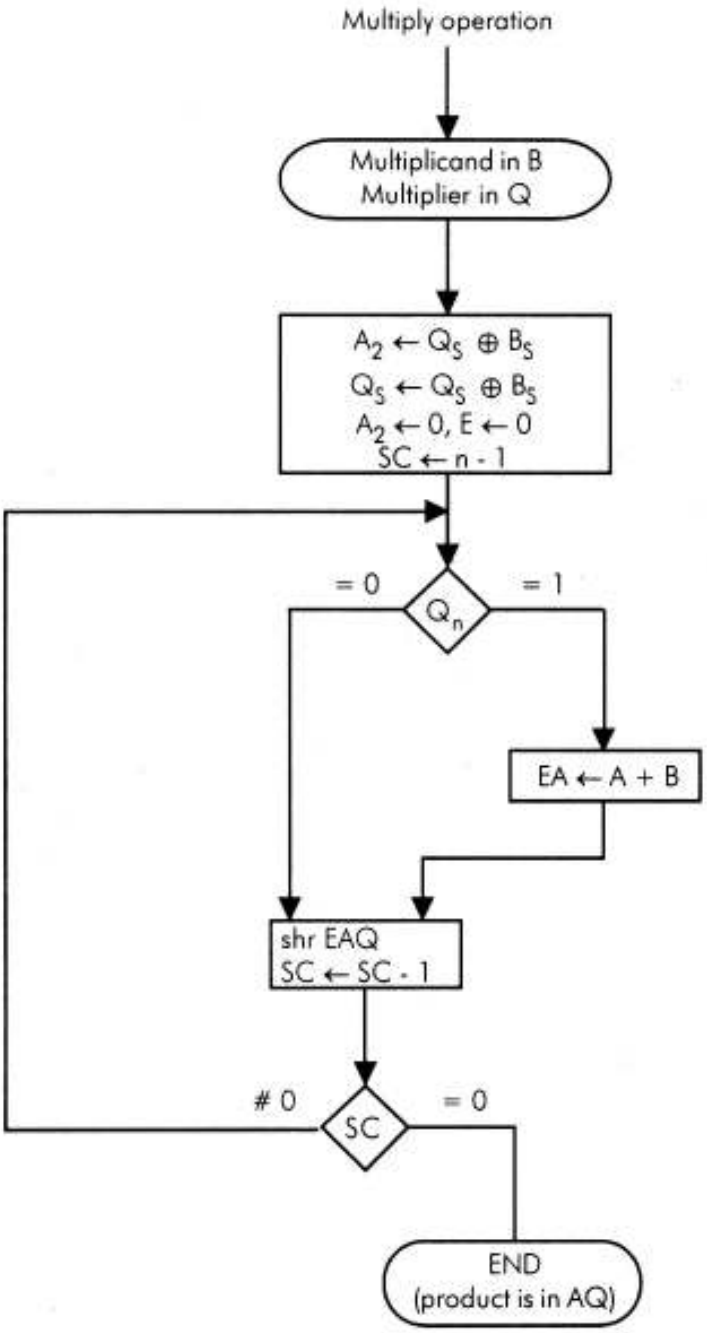


Figure 4.6: Flowchart for Multiply Operation

The hardware divide algorithm is given in Figure 4.12. A and Q contain the dividend and B has the divisor. The sign of the result is transferred into Q. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will have n-1 bits.

We can check a divide-overflow condition by subtracting the divisor (B) from half of the bits of the dividend stored (A). If A<B, the divide-overflow occur and the operation is terminated. If $A \geq B$, no divide overflow occurs and so the value of the dividend is restored by adding B to A.

The division of the magnitudes begins by shifting the dividend in AQ to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that EA>B because EA consists of 1 followed by n-1 bits while B consists of only n-1 bits. In this case, B must be subtracted from EA and 1 inserted into Q_n for the quotient bit. Since in register A, the high-order bit of the dividend (which is in E) is missing, its value is $EA - 2^{n-1}$. Adding to this value the 2's complement of B results in:
$(EA - 2^{n-1}) + (2^{n-1} - B) = EA - B$

If we want E to remain a 1, the carry from this addition is not transferred to E. If the shift-left operation inserts a 0 into E, we subtract the divisor by adding its 2's complement value and the carry is transferred into E. If E=1, it shows that A< B, therefore Q_n is set. If E = 0, it signifies that A < B and the original number is restored by B + A. In the latter case we leave a 0 in Q_n.
We repeat this process with register A holding the partial remainder. After n-1 loops, the quotient magnitude is stored in register Q and the remainder is found in register A. The quotient sign is in Qs and the sign of the remainder is in As.

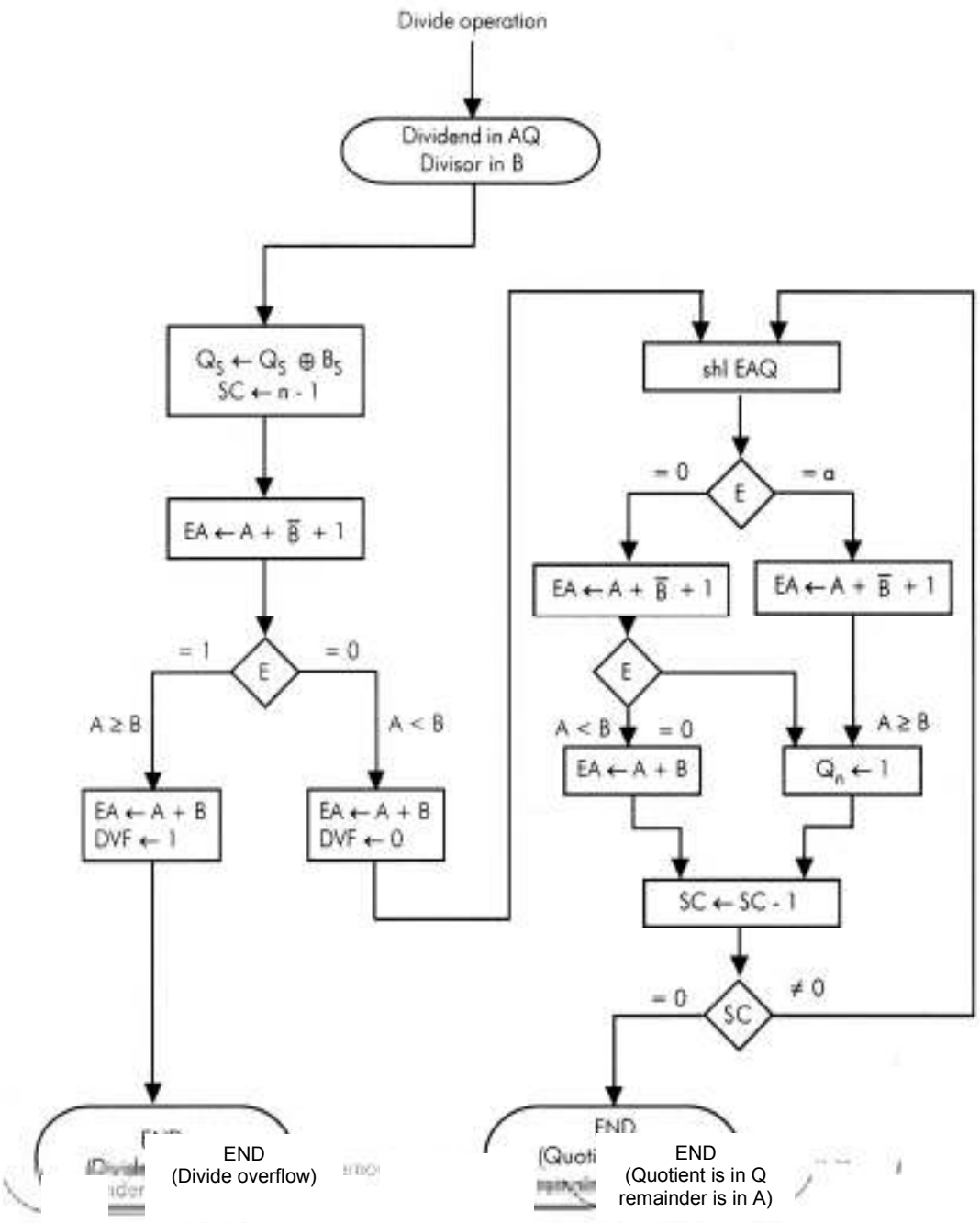


Figure : Flowchart for Divide Operation

3.3.3.2 Divide Overflow

An overflow may occur in the division operation, which may be easy to handle if we are using paper and pencil but is not easy when are using hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, let us consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example of Figure 4.12, the

quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit. This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers. Provisions to ensure that this condition is detected must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor, we can understand the condition for overflow as follows:

A divide-overflow occurs if the high-order half bits of the dividend makes a number greater than or equal to the divisor. Another problem associated with division is the fact that a division by zero must be avoided. The divide-overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor, which is equal to zero. Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

3.4 Floating-point Arithmetic operations

In many high-level programming languages we have a facility for specifying floating-point numbers. The most common way is by a real declaration statement. High level programming languages must have a provision for handling floating-point arithmetic operations. The operations are generally built in the internal hardware. If no hardware is available, the compiler must be designed with a package of floating-point software subroutine. Although the hardware method is more expensive, it is much more efficient than the software method. Therefore, floating- point hardware is included in most computers and is omitted only in very small ones.

3.4.1 Basic Considerations

There are two part of a floating-point number in a computer - a mantissa m and an exponent e. The two parts represent a number generated from multiplying m times a radix r raised to the value of e. Thus

$$m \times r^e$$

The mantissa may be a fraction or an integer. The position of the radix point and the value of the radix r are not included in the registers. For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating-point number

$$.53725 \times 10^3$$

A floating-point number is said to be normalized if the most significant digit of the mantissa in nonzero. So the mantissa contains the maximum possible number of significant digits. We cannot normalize a zero because it does not have a nonzero digit. It is represented in floating-point by all 0's in the mantissa and exponent.

Floating-point representation increases the range of numbers for a given register. Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be + (24^7 – 1), which is approximately + 101^4. The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be represented is

$$+ (1 - 2^{-35}) \times 2^{2047}$$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and because 2^{11}–1 = 2047. The largest number that can be accommodated is approximately 10^{615}. The mantissa that can accommodated is 35 bits (excluding the sign) and if considered as an integer it can store a number as large as (2^{35} –1). This is approximately equal to 10^{10}, which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating-point number. An 8-bit microcomputer uses four words to represent one floating-point number. One word of 8 bits are reserved for the exponent and the 24 bits of the other three words are used in the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers. Their execution also takes longer time and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. We do this alignment by shifting one mantissa while its exponent is adjusted until it becomes equal to the other exponent. Consider the sum of the following floating-point numbers:

$$.5372400 \times 10^2$$
$$+ .1580000 \times 10^{-1}$$

It is necessary to make two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When we store the mantissas in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. Now, the mantissas can be added.

$$\begin{array}{r} .\ 5372400 \times 10^2 \\ +.\ 0001580 \times 10^2 \\ \hline .\ 5373980 \times 10^2 \end{array}$$

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

$$\begin{array}{r} .56780 \times 10^5 \\ -\ .56430 \times 10^5 \\ \hline .00350 \times 10^5 \end{array}$$

An underflow occurs if a floating-point number that has a 0 in the most significant position of the mantissa. To normalize a number that contains an underflow, we shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. Here, it is necessary to shift left twice to obtain $.35000 \times 10^3$. In most computers a normalization procedure is performed after each operation to ensure that all results are in a normalized form.

Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

The operations done with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compared and incremented (for aligning the mantissas), added and subtracted (for multiplication) and division), and decremented (to normalize the result). We can represent the exponent in any one of the three representations - signed-magnitude, signed 2's complement or signed 1's complement.
A is a fourth representation also, known as a biased exponent. In this representation, the sign bit is removed from beginning to form a separate entity. The bias is a positive number that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive. The following example may clarify this type of representation. Consider an exponent that ranges from –50 to 49. Internally, it is represented by two digits (without a sign) by adding to it a bias of 50. The exponent register contains the number e + 50, where e is the actual exponent. This way, the exponents are represented in registers as positive numbers in the range of 00 to 99. Positive exponents in registers have the range of numbers from 99 to 50. The subtraction pf 50 gives the positive values from 49 to 0. Negative exponents are represented in registers in the range of –1 to –50.

Biased exponents have the advantage that they contain only positive numbers. Now it becomes simpler to compare their relative magnitude without bothering about their

signs. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

3.1.1.1 Register Configuration

The register configuration for floating-point operations is shown in figure 4.13. As a rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

The register organization for floating-point operations is shown in Fig. 4.13. Three registers are there, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part may use corresponding lower-case letter symbol.

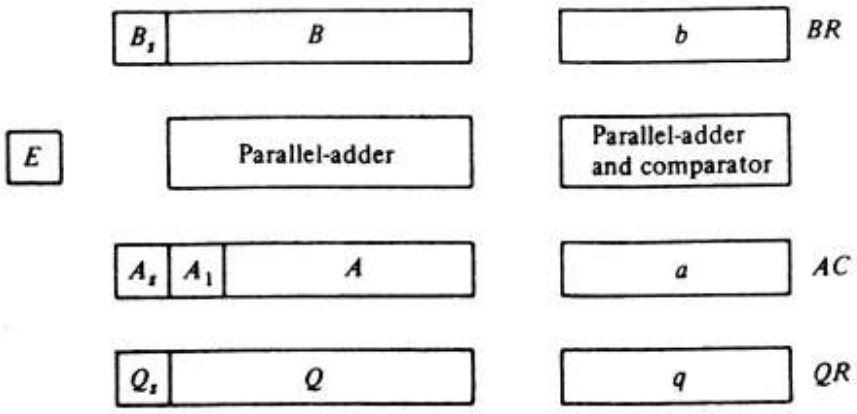


Figure 4.13: Registers for Floating Point arithmetic operations

Assuming that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in A_s, and a magnitude that is in A. The diagram shows the most significant bit of A, labeled by A_1. The bit in his position must be a 1 to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation of A_s, A and a.

In the similar way, register BR is subdivided into B_s, B, and b and QR into Q_s, Q and q. A parallel-adder adds the two mantissas and loads the sum into A and the carry into E. A separate parallel adder can be used for the exponents. The exponents do not have a district sign bit because they are biased but are represented as a biased positive quantity. It is assumed that the floating-point number are so large that the chance of an exponent overflow is very remote and so the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a fraction, so they binary point is assumed to reside to the left of the magnitude part. Integer representation for floating point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation.
The numbers in the registers should initially be normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands are always normalized.

3.4.1.2 Addition and Subtraction of Floating Point Numbers

During addition or subtraction, the two floating-point operands are kept in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas
4. Normalize the result

A floating-point number cannot be normalized, if it is 0. If this number is used for computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be un-normalized. The normalization procedure ensures that the result is normalized before it is transferred to memory.

For adding or subtracting two floating-point binary numbers, if BR is equal to zero, the operation is stopped, with the value in the AC being the result. If AC = 0, we transfer the content of BR into AC and also complement its sign we have to subtract the numbers. If neither number is equal it to zero, we proceed to align the mantissas.

The magnitude comparator attached to exponents a and b gives three outputs, which show their relative magnitudes. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until two exponents are equal.

The addition and subtraction of the two mantissas is similar to the fixed-point addition and subtraction algorithm presented in Fig. 4.14. The magnitude part is added or subtracted depends on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E = 1, the bit is transferred into A_1 and all other bits of A are shifted right. The exponent must be

incremented so that it can maintain the correct number. No underflow may occur in this case this is because the original mantissa that was not shifted during the alignment was already in a normalized position.

If the magnitudes were subtracted, there may be zero or may have an underflow in the result. If the mantissa is equal to zero the entire floating-point number in the AC is cleared to zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A_1, is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A_1 is checked again and the process is repeated until $A_1 = 1$. When $A_1 = 1$, the mantissa is normalized and the operation is completed.

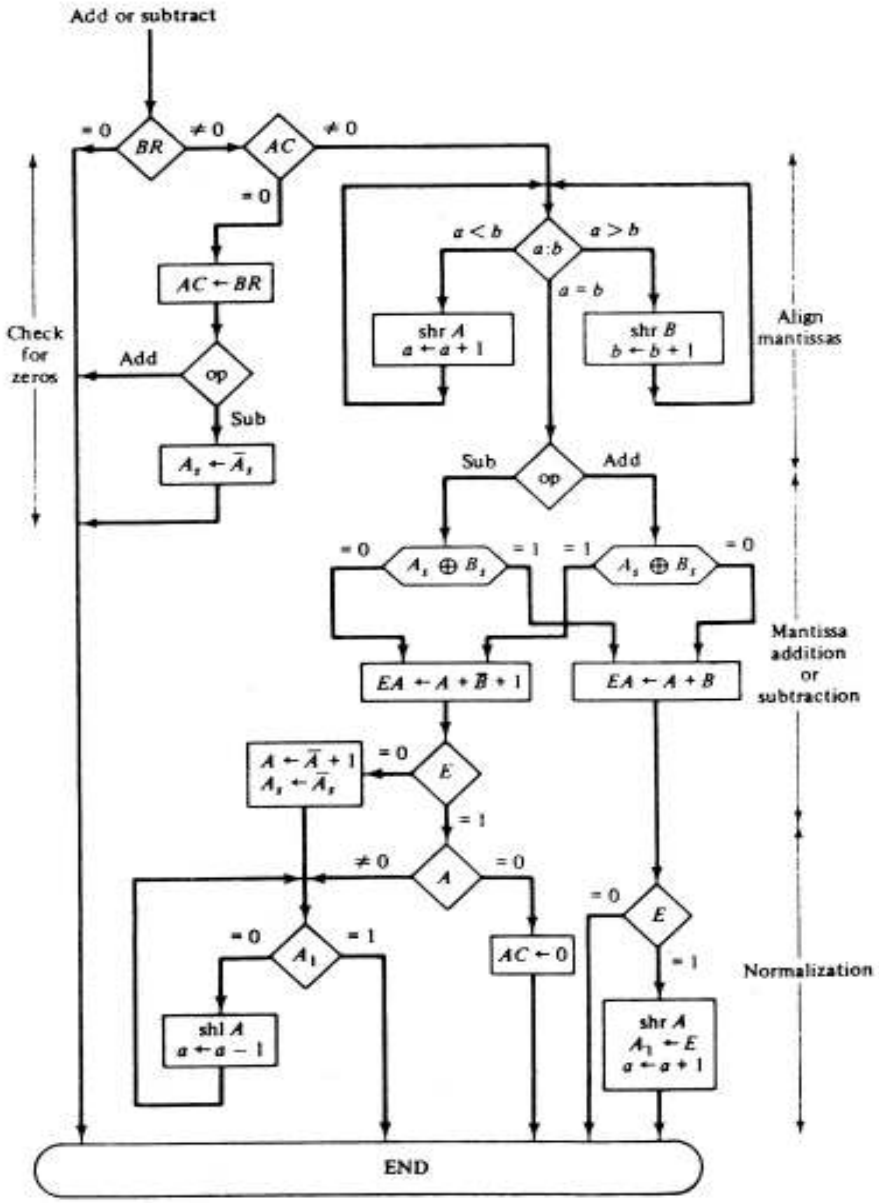


Figure Addition and Subtraction of floating –point numbers

3.4.2 Decimal Arithmetic operations

Decimal Arithmetic Unit

The user of a computer input data in decimal numbers and receives output in decimal form. But a CPU with an ALU can perform arithmetic micro-operations only on binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and output data. When the application calls for a large amount of input-output and a relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers. Computers that can do decimal arithmetic must store the decimal data in binary coded form. The decimal numbers are then applied to a decimal arithmetic unit, which can execute decimal arithmetic micro-operations.
Electronic calculators invariably use an internal decimal arithmetic unit since inputs and outputs are frequent. There does not seem to be a reason for converting the keyboard input numbers to binary and again converting the displayed results to decimal, this is because this process needs special circuits and also takes a longer time to execute. Many computers have hardware for arithmetic calculations with both binary and decimal data.

Users can specify by programmed instructions whether they want the computer to does calculations with binary or decimal data.

A decimal arithmetic unit is a digital function that does decimal micro-operations. It can add or subtract decimal numbers. The unit needs coded decimal numbers and produces results in the same adopted binary code. A single-stage decimal arithmetic unit has of nine binary input variables and five binary output variables, since a minimum of four bits is required to represent each coded decimal digit. Each stage must have four inputs for the addend digit, four inputs for the addend digit, and an input-carry. The outputs need four terminals for the sum digit and one for the output-carry. Of course, there is a wide range of possible circuit configurations dependent on the code used to represent the decimal digits.

3.4.2.1 BCD Adder

Now let us see the arithmetic addition of two decimal digits in BCD, with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum

cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input-carry. Assume that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary and produce a result that may range from 0 to 19. These binary numbers are listed in Table 4.4 and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1. K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four its in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit binary adder.

The output sum of two decimal numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary column of the table. The problem is to find a simple rule so that the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column.

It is apparent that when the binary sum is equal to or less than 1001, no conversion is needed. When the binary sum is greater than 1001, we need to add of binary 6 (0110) to the binary sum to find the correct BCD representation and to produces output-carry as required.

Table 4.4: Derivation of BCD Adder

Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

One way of adding decimal numbers in BCD is to use one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum if the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. The second operation produces an output-carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal

to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until all decimal digits are added.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry K = 1. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_8. To differentiate them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

When C = 1, we need to add 0110 to the binary sum and provide an output-carry for the next stage.

A BCD adder is circuit that adds two BCD digits in parallel and generates a sum digit also in BCD. ABCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.

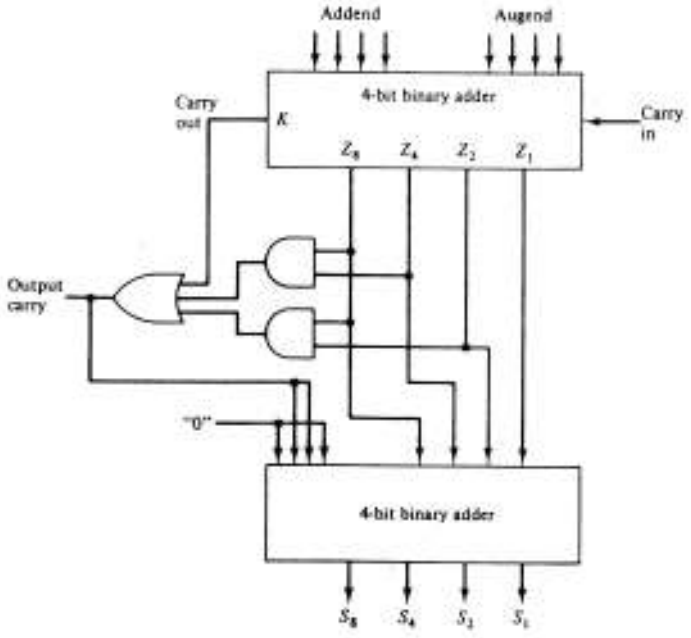


Figure 4.17: Block Diagram of BCD Adder

3.4.2.2 BCD Subtraction

Subtraction of two decimal numbers needs a subtractor circuit that is different from a BCD adder. We perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend. Since the BCD is not a self-complementing code, we cannot obtain the 9's complement by complementing each bit in the code. It must be formed using a circuit that subtracts each BCD digit from 9.

The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit but we have to include. There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit then we discard the carry after each addition. In the second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111(decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. Complementing each bit of 4-bit binary number N is identical to the subtraction of the number from 1111 (decimal 15). Adding the binary equivalent of decimal 10 gives 15 – N + 10 = 9 + 16. But 16 signifies the carry that is discarded, so the result is 9 – N as required. Adding the binary equivalent of decimal 6 and then complementing gives 15 – (N + 6) = 9 – N as required.

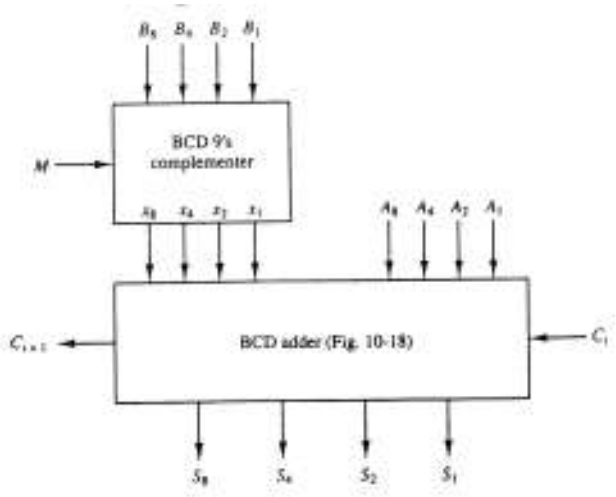
We can also obtain the 9's complement of a BCD digit through a combinational circuit. When this circuit is combined to a BCD adder, we get a BCD adder/subtractor. Let the subtrahend (or addend) digit be denoted by the four binary variables B_8, B_4, B_2, and B_1. Let M be a mode bit that controls the add/subtract operation. When M = 0, the two digits are added; when M = 1, the digits are subtracted. Let the binary variables x_8, x_4, x_2, and x_1 be the outputs of the 9's complement circuit. By an examination of the truth table for the circuit, it may be observed that B_1 should always be complemented; B_2 is always the same in the 9's complement as in the original digit; x4 is 1 when the exclusive OR of B_2 and B_4 is 1; and x_8 is 1 when $B_8B_4B_2 = 000$. The Boolean functions for the 9's complement circuit are

$$x_1 = B_1\, M' + B'_1\, M$$
$$x_2 = B_2$$
$$x_4 = B_4M' + (B'_4B_2 + B_4B'_2)M$$
$$x_8 = B_8M' + B'_8B_4'B'_2M$$

From these equations we see that x = B when M = 0. When M = 1, the x equals to the 9's complement of B.

One stage of a decimal arithmetic unit that can be used to add or subtract two BCD digits is given in Fig. 4.18. It has of a BCD adder and a 9's complementer. The mode M controls the operation of the unit. With M = 0, the S outputs form the sum of A and B. With M = 1, the S outputs form the sum of A plus the 9's complement of B. For numbers with n decimal digits we need n such stages. The output carries C_i+1 from one stage. to subtract the two decimal numbers let M = 1 and apply a 1 to the input carry

C_1 of the first stage. The outputs will form the sum of A plus the 10's complement of B, which is equivalent to a subtraction operation if the carry-out of the last stage is discarded.



One Stage of a decimal arithmetic unit

UNIT 4
PROGRAMMING THE BASIC COMPUTER

4.1 Machine Language

To write a program for a computer we need to specify, directly or indirectly, a sequence of machine instructions. Machine instructions inside the computer form a binary pattern, which is difficult to understand and manipulate. The instruction set of the basic computer, whose hardware organization was explored earlier used to program a computer. The 25 instructions of the basic computer are in Table 2.5 to provide an easy reference for the programming examples that follow.

Table 2.5: Computer Instructions

Symbol	Hexadecimal code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to m+1
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC

SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

A program is a list of instructions to tell the computer to do needed processing on the data. We can write a program for a computer in various types of programming languages, but this program can be executed by the computer only when it is in binary form. If it is written in some other language it has to be translated to the binary form so that it can be executed by the computer.

A symbolic instruction and its binary equivalent has a one-to-one relationship between them. An assembly language can be thought of a machine level language writing 16 bits for each instruction. Because there are several digits, it becomes tedious. By writing the equivalent hexadecimal code, we can reduce to four digits. We can go one step further and replace each hexadecimal address by a symbolic address and each hexadecimal opened by a decimal operand. We find it convenient because, we generally do not know exactly the numeric memory location of operands at the time of writing a program. As we know that there is a set of rules for a programming language, we must conform with all format and rules of the language if we want our programs to be translated correctly. A line of code is the fundamental unit of an assembly language program.

Table 2.6: Binary Program to Add Two Numbers

Location	Instruction Code			
0	0010	0000	0000	0100
1	0001	0000	0000	0101
10	0011	0000	0000	0110
11	0111	0000	0000	0001
100	0000	0000	0101	0011
101	1111	1111	1110	1001
110	0000	0000	0000	0000

4.2 Assembly Language

As we know that a programming language is defined by a set of rules. If users want their programs to be translated correctly they must conform to all formats and rules of the language. Every digital computer has its own separate assembly language. The rules for writing assembly language programs are available from the computer manufacturer in the form of documents and manuals.

A line of code is the fundamental unit of an assembly language program. The specific language is defined by a group of rules. This group describes the symbols that can be used. It also tells how a line of code can be made from them. We will now give the rules to write assembly language programs for the basic computer.

Rules for the Assembly Language

A line of code of an assembly language program is divided in three columns called fields. The fields describe the following information.

1. The label: may be empty or it may specify a symbolic address.
2. The instruction: specifies a machine instruction or a pseudo-instruction.
3. The comment: may be empty or it may include a comment.

A symbolic address has one, two, or three alphanumeric characters. But it cannot have more than three alphanumeric characters. The first character is an alphabet; the next two may be alphabets or numeric digits. We can choose the symbols arbitrarily. A symbolic address in the label field is terminated by a comma to make it a label. The instruction field specifies one of the following items:

1. A memory-reference instruction (MRI)
2. A register-reference (i.e. input-output instruction) (non-MRI)
3. A pseudo-instruction with or without an operand

A memory-reference instruction occupies two or three symbols. These symbols are separated by spaces. The first must be a three-letter symbol defining an MRI operation code. The second one is a symbolic address. The third symbol, which is optional, is the letter I. It is a direct address instruction, if I is missing otherwise it is an indirect address instruction.

A non-MRI is an instruction that does not have an address part. A non-MRI is found in the instruction field of a program by any one of the three-letter symbols for the register-reference and input-output instructions.

The following is an illustration of the symbols that may be placed in the instruction field of a program.

CLA	non-MRI
ADD OPR	direct address MRI
ADD PTR I	indirect address MRI

The first three-letter symbol in each line must be one of the instruction symbols of the computer. A memory-reference instruction, such as MUL, must be followed by a symbolic address. The letter I may or may not be present.

The memory location of an operand is determined by a symbolic address in the instruction field. This location is mentioned somewhere in the program by appearing again as a label in the first column. If we want to translate program from assembly language to a machine language, each symbolic address that is mentioned in the instruction field must occur again in the label field.

A pseudo-instruction is an instruction to the assembler giving information about some phase of the translation (it is not a machine instruction). Four pseudo-instructions that are recognized by the assembler are listed in Table 2.7. The assembler is informed by the origin (ORG) pseudo-instruction that the instruction or operand in the following line is to be placed in a memory location specified by the number next to ORG.

Table 2.7: Definition of Pseudo-instructions

Symbol	Information for the Assembler
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

To inform the assembler that the program is terminated the END symbol is placed at the end of the program. The radix is given by the other two pseudo-instructions. They also describe the operand and tell the assembler how to convert the listed number to a binary one.

We reserve the third field in a program for comments. A line of code may or may not have a comment. But if there is a comment, it must be preceded by a slash for the assembler to recognize the beginning of a comment field. Comments are useful for explaining the program and are helpful in understanding the step-by-step procedure taken by the program. Comments are used for explanation and are not are neglected during the binary translation process.

An Example

The program of Table 2.8 is an example of an assembly language program. The first line has the pseudo instruction ORG to define the origin of the program at memory location $(100)_{16}$.
The next six lines define machine instructions, and the last four have pseudo-instructions. Three symbolic addresses have been used and each is listed in column 1 as a label and in column 2 as an address of a memory-reference instruction. Three of

the pseudo-instructions specify operands, and the last one signifies the END of the program.

When the program is converted into the binary code and executed by the computer it perform a subtraction between two numbers. We can perform subtraction operation by adding the minuend to the 2's complement of the subtrahend. We know that subtrahend is a negative number, we convert it into a binary number as signed 2's complement representation because we dictate that all negatives numbers be in their 2's complement form. Thus, – 23 converts to +23 and the difference is 83+ (2's complement of –23) = 83+23=106.

Table 2.8: Language Program to Subtract Two Numbers

```
        ORG 100     /Origin of program is location 100
        LDA SUB     /Load subtrahend to AC
        CMA         /Complement AC
        INC         /Increment AC
        ADD MIN     /Add minuend to AC
        STA DIF     /Store difference
        HLT         /Halt computer
MIN,    DEC 83      /Minuend
SUB,    DEC -23     /Subtrahend
DIF,    HEX 0       /Difference stored here
        END         /End of symbolic program
```

4.3 Assembler

An assembler is a program that takes as input a symbolic language program and produces as output its binary machine language equivalent. The input is called the source program and the resulting binary program is called the object program. The assembler is a program that operates on character strings and produces an equivalent binary interpretation.

Memory Representation of Symbolic Program

The symbolic program must be stored in memory, before starting the assembly process. The user writes the symbolic program on a computer. This symbolic program is taken into memory with the help of a loader program. Since the program consists of symbols, its representation in memory must use an alphanumeric character code. Usually each character is represented by an 8-bit code, in the basic computer. The high-order bit is always 0 and the other seven bits are as described by ASCII code. Table 2.10 gives the hexadecimal equivalent of the character set. Each character is given two hexadecimal digits. So each character can be easily converted to their equivalent 8-bit code. The last entry in the table does not print a character, it looks after the physical movement of the cursor in the terminal. When the return key is depressed, the code for CR is produced. Therefore "carriage" is goes to its initial position and we can start typing a new line.

We store a line of code in consecutive memory locations. Two characters in each location. Since a memory word has a capacity of 16 bits we can store two characters

stored in each word. A comma delimits a label symbol. Now we see how the operation code and addresses are terminated. They are terminated with a space and the end of the line is recognized by the CR code. For example, the following line or code: PL3, LDA SUB I is stored in seven consecutive memory locations, as shown in Table 2.11. The label PL3 occupies two words and is terminated by the code for comma (2C). The instruction field in the line of code may have one or more symbols. Each symbol is terminated by the code for space (20) except for the last symbol, which is terminated by the code of carriage return (0D). If the line of code has a comment, the assembler recognizes it by the code for a slash (2F). The assembler neglects all characters in the comment field and keeps checking for a CR code. When this code is encountered, it replaces the space code after the last symbol in the line of code.

Table 2.10: Hexadecimal Character Code

Character	Code	Character	Code	Character	Code	
A	41	Q	51	6	36	
B	42	R	52	7	37	
C	43	S	53	8	38	
D	44	T	54	9	39	
E	45	U	55	space	20	
F	46	V	56	(	28	
G	47	W	57	)	29	
H	48	X	58	*	2A	
I	49	Y	59	+	2B	
J	4A	Z	5A	,	2C	
K	4B	0	30	−	2D	
L	4C	1	31	.	2E	
M	4D	2	32	/	2F	
N	4E	3	33	=	3D	
O	4F	4	34	CR	0D	(carriage return)
P	50	5	35			

Table 2.11: Computer Representation of the line of code: PL3, LDA SUB I

Memory word	Symbol	Hexadecimal code	Binary representation
1	P L	50 4C	0101 0000 0100 1100
2	3 ,	33 2C	0011 0011 0010 1100
3	L D	4C 44	0100 1100 0100 0100
4	A	41 20	0100 0001 0010 0000
5	S U	53 55	0101 0011 0101 0101
6	B	42 20	0100 0010 0010 0000
7	I CR	49 0D	0100 1001 0000 1101

The user's symbolic language program in ASCII is input for the assembler program. The assembler scans this input twice to produce the equivalent binary program. The binary program constitutes the output generated by the assembler. We will now describe

briefly the major tasks that must be performed by the assembler during the translation process.

4.3.1 First Pass

Entire symbolic program is scanned by a two-pass assembler twice. After the first pass, it generates a table that correlates all user-defined symbols with their equivalent value in binary. The binary translation is done during the second pass. To keep track of the location of instructions, the assembler uses a memory word called a location counter (abbreviated LC). The content of LC stores the value of the memory location assigned to the instruction or operand presently being processed. The ORG pseudo-instruction initializes the location counter to the value of the first location. Since instructions are stored in sequential locations, the content of LC is incremented by 1 after processing each line of code. To avoid ambiguity in case ORG is missing, the assembler sets the location counter to 0 initially.

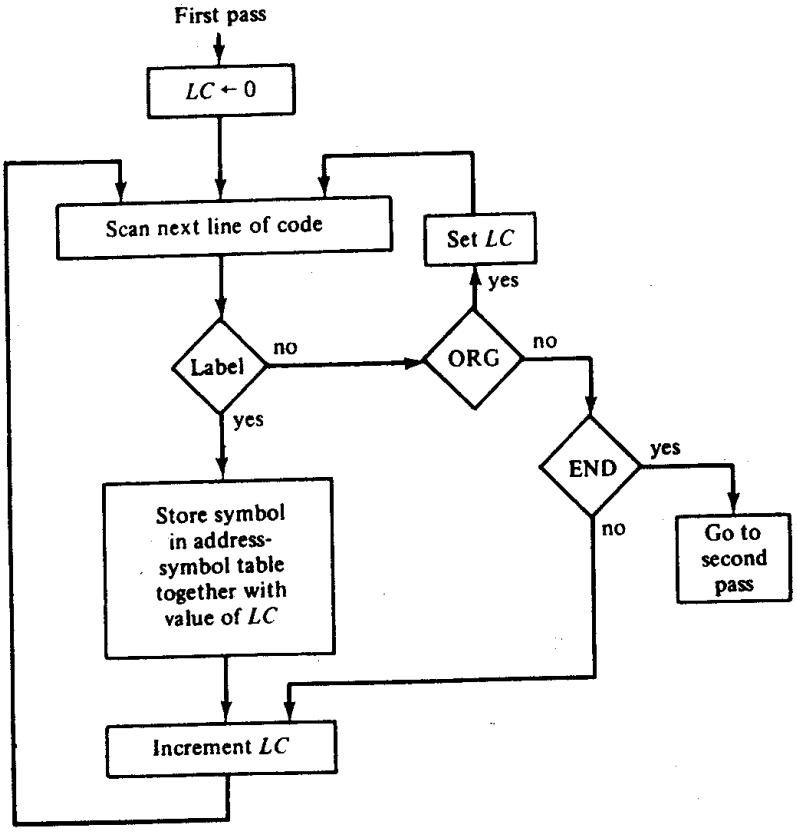


Figure 2.9: Flowchart for first pass of assembler

The flowchart of Fig. 2.9 describes the tasks performed by the assembler during the first pass. LC is initially set to 0. A line of symbolic code is analyzed to determine if it has a label (by the presence of a comma). If the line of code has no label, the assembler checks the symbol in the instruction field. If it contains an ORG pseudo-instruction, the assembler sets LC to the number that follows ORG and goes back to process the next

line. If the line has an END pseudo-instruction, the assembler terminates the first pass and goes to the second pass. (Note that a line with ORG or END should not have a label.) If the line of code contains a label, it is stored in the address symbol table together with its binary equivalent number specified by the content of LC. Nothing stored in the table if no label is encountered. LC is then incremented by 1 and a new line of code is processed.

The assembler generates the address symbol table listed in Table 2.12, for the program to Table 2.8. We store each label symbol in two memory locations and terminate it by a comma. If the label contains less than three characters, the memory locations are filled with the code for space. The value found in LC while the line was processed is stored in the next sequential memory location. The program has three symbolic addresses: MIN, SUB, and DIF. These symbols represent 12-bit addresses equivalent to hexadecimal 106 107 and 108, respectively. The address symbol table occupies three words for each label symbol encountered and constitutes the output data that the assembler generates during the first pass.

Table 2.12

Memory word	Symbol or (LC)*	Hexadecimal code	Binary representation
1	M I	4D 49	0100 1101 0100 1001
2	N ,	4E 2C	0100 1110 0010 1100
3	(LC)	01 06	0000 0001 0000 0110
4	S U	53 55	0101 0011 0101 0101
5	B ,	42 2C	0100 0010 0010 1100
6	(LC)	01 07	0000 0001 0000 0111
7	D I	44 49	0100 0100 0100 1001
8	F ,	46 2C	0100 0110 0010 1100
9	(LC)	01 08	0000 0001 0000 1000

* (LC) designates content of location counter.

4.3.2 Second Pass

With the help of table-lookup procedures, machine instructions are translated during the second pass. A take-lookup procedure is a search of table entries to find whether a specific item matches one of the items stored in the table. The assembler uses four tables. Any symbol that is encountered in the program must be available as an entry in one of these tables; otherwise, the symbol cannot be interpreted. We assign the following names to the four tables:

1. Pseudo-instruction table.
2. MRI table.
3. Non-MRI table.
4. Address symbol table.

The pseudo-instruction table has the four symbols ORG, END, DEC, and HEX. Each symbol refers the assembler to a subroutine that processes the pseudo-instruction when encountered in the program. The MRI table has the seven symbols of the memory-reference instructions and their 3-bit operation code equivalent. The non-MRI table has the symbols for the 18 register-reference and input-output instructions and their 16-bit binary code equivalent. In the first pass the address symbol table is created. In order to determine its binary value, the assembler searches these tables to find the symbol that it is currently processing. The tasks performed by the assembler during the second pass are described in the flowchart of Fig. 2.10. LC is initially set to 0. Lines of code are then analyzed one at a time. Labels are neglected during the second pass, so the assembler goes immediately to the instruction field and proceeds to check the first symbol encountered. It first checks the pseudo-instruction table. A match with ORG sends the assembler to a subroutine that sets LC to an initial value. A match with END terminates the translation process. An operand is placed in the memory location specified by the content of LC. The location counter is then incremented by 1 and the assembler continues to analyze the next line of code.

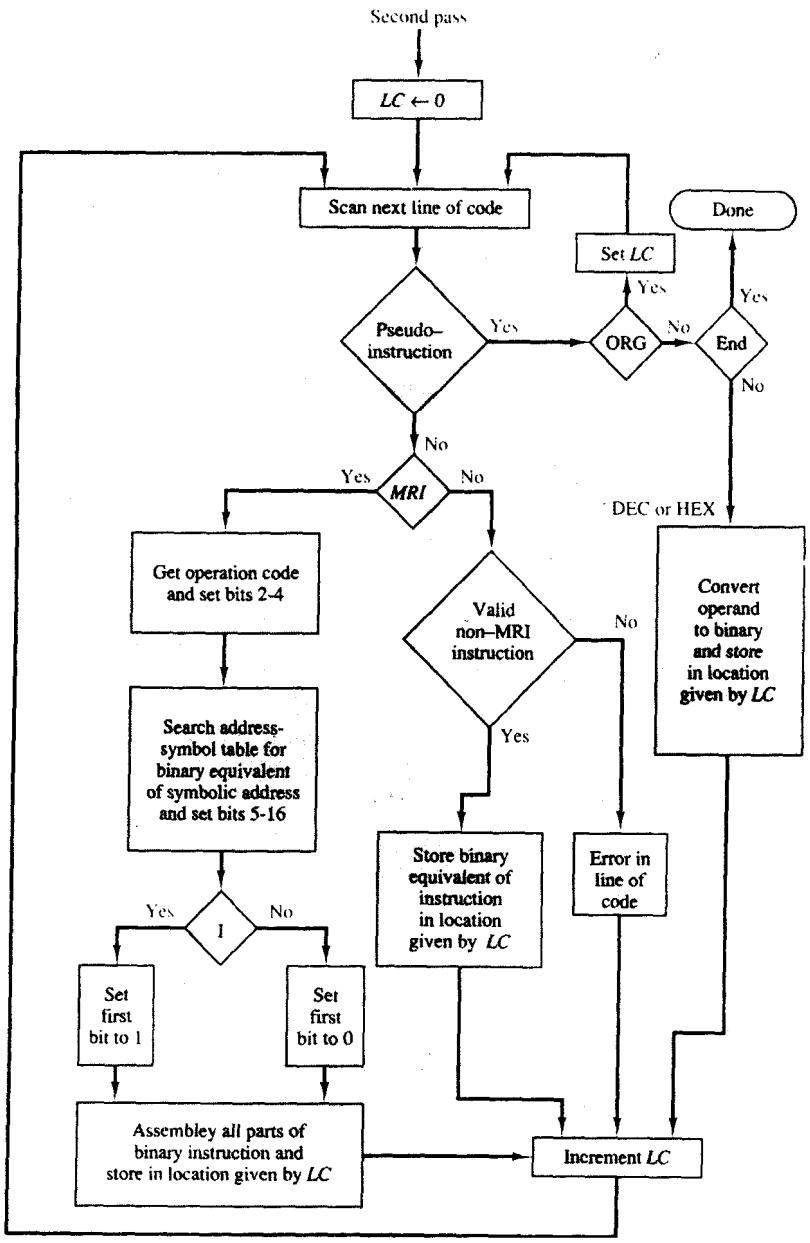


Figure 2.10: Flochart for second pass of assembler

If the symbol obtained is not a pseudo-instruction, the assembler goes to the MRI table. The assembler refers to the non-MRI table if the symbol is not found in MEI table. A symbol found in the non-MRI table corresponds to a register reference or input-output instruction. The assembler stores the 16-bit instruction code into the memory word designated by LC. The location counter is incremented and a new line analyzed.

When we get a symbol in the MRI table, the assembler extracts its equivalent 3-bit code and inserts it in bits 2 through 4 of a word. A memory reference instruction is designated by two or three symbols. The second symbol is a symbolic address and the third, which may or may not be present, is the letter I. By searching the address symbol

table the symbolic address is converted to binary. The first bit of the instruction is set to 0 or 1, depending on whether the letter I is absent or present. The three parts of the binary instruction code are assembled and then stored in the memory location specified by the content of LC. The location counter is incremented and the assembler continues to process the next line.

An important job of the assembler is to check for possible errors. We can call it "error diagnostics". One example of such an error may be an invalid machine code symbol which is detected by its being absent in the MRI and non-MRI tables. The assembler is unable to translate such a symbol because it does not know its binary equivalent value. In such a case, the assembler prints an error message to inform the programmer that his symbolic program has an error at a specific line of code. Another possible error may occur if the program has a symbolic address that did not appear also as a label. The assembler cannot translate the line of code properly because the binary equivalent of the symbol will not be found in the address symbol table generated during the first pass. Other errors may occur and a practical assembler should detect all such errors and print an error message for each.

4.4 Programming Arithmetic and Logic Operations

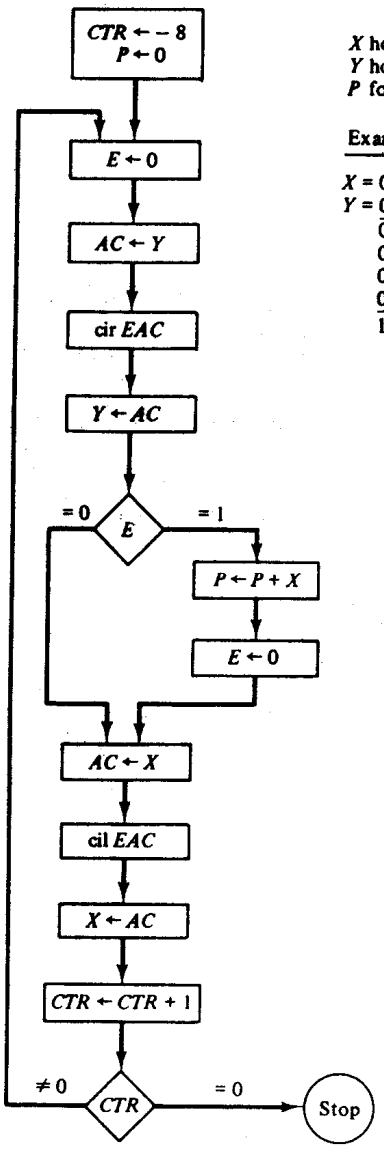
In a large system the number of instructions available in a computer may be a few hundred or a few dozen in a small one. Some computers execute a given operation with one machine instruction; some may require many machine instructions to perform the same operation. For example, consider the four basic arithmetic operations. Some computers have machine instructions to add, subtract, multiply, and divide. Others, such as the basic computer, have only one arithmetic instruction, such as ADD. A program must implement operations not included in the set of machine instructions.

We have shown in Table 2.8 a program for subtracting two numbers. Programs for the other arithmetic operations can be developed in a similar fashion.

If operations are implemented in a computer with one machine instruction, then it is said to be implemented by hardware. Operations implemented by a set of instructions that form a program are said to be implemented by software. Some computers provide an extensive set of hardware instructions designed so that common tasks can be performed efficiently. Others contain a smaller set of hardware instructions and depend more heavily on the software implementation of many operations. Hardware implementation is more costly because of the additional circuits needed to implement the operation. Software implementation results in long programs both in number of instructions and in execution time.

4.5 Multiplication Program

We use the conventional method of multiplying two numbers to write the program for multiplying two numbers. As shown in the example of Fig. 2.11, the multiplication process consists of checking the bits of the multiplier Y and adding the multiplicand X as many times as there are 1's in Y, provided that the value of X is shifted left from one line to the next. Since the computer can add only two numbers at a time, we reserve a memory location, denoted by P, to store intermediate sums. The intermediate sums are called partial products since they hold a partial product until all numbers are added. As shown in the numerical example under P, the partial product starts with zero. The multiplicand X is added to the content of P for each bit of the multiplier Y that is 1. The value of X is shifted left after checking each bit of the multiplier. The final value in P forms the product. The example has numbers with four significant bits. When multiplied, the product contains eight significant bits. The computer can use numbers with eight significant bits to produce a product of up to 16 bits. The flowchart of Fig. 2.11 shows the step-by-step procedure for programming the multiplication operation. The program has a loop that is traversed eight times, once for each significant bit of the multiplier. Initially, location X holds the multiplicand and location Y holds the multiplier. A counter CTR is set to -8 and location P is cleared to zero.



X holds the multiplicand
Y holds the multiplier
P forms the product

Example with four significant digits

		P
X =	0000 1111	
Y =	0000 1011	0000 0000
	0000 1111	0000 1111
	0001 1110	0010 1101
	0000 0000	0010 1101
	0111 1000	1010 0101
	1010 0101	

Figure 2.11: Flowchart for multiplication program

We can check multiplier bit if it is transferred to the E register. We do this by clearing E, loading the value of Y into the AC, circulating right E and AC and storing the shifted number back into location Y. This bit stored in E is the low-order bit of the multiplier. We now check the value of E. If it is 1, the multiplicand X is added to the partial product P. If it is 0, the partial product does not change. We then shift the value of X once to the left by loading it into the AC and circulating left E and AC. The loop is executed eight times by incrementing location CTR and checking when it reaches zero. When the counter reaches zero, the program exits from the loop with the product stored in location P.

The program in Table 2.14 gives the instructions for multiplication of two unsigned numbers. The initialization is not listed but should be included when the program is

loaded into the computer. The initialization consists of bringing the multiplicand and multiplier into locations X and Y, respectively; initializing the counter to -8; and initializing location P to zero.

Table 2.14: Program to Multiply Two Positive Numbers

```
        ORG 100
LOP,    CLE         /Clear E
        LDA Y       /Load multiplier
        CIR         /Transfer multiplier bit to E
        STA Y       /Store shifted multiplier
        SZE         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRO
ONE,    LDA X       /Load multiplicand
        ADD P       /Add to partial product
        STA P       /Store partial product
        CLE         /Clear E
ZRO,    LDA X       /Load multiplicand
        CIL         /Shift left
        STA X       /Store shifted multiplicand
        ISZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        HLT         /Counter is zero; halt
CTR,    DEC -8      /This location serves as a counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX 0       /Product formed here
        END
```

4.5.1 Double-Precision Addition

When we multiply two 16-bit unsigned numbers, the result is a 32-bit product and it must be stored in two memory words. A number is said to have double precision if it is stored in two memory words. When a partial product is computed, it is necessary that a double-precision number be added to the shifted multiplicand, which is also a double-precision number. For greater accuracy, the programmer may wish to employ double-precision numbers and perform arithmetic with operands that occupy two memory words. We now develop a program that adds two double-precision numbers.

We place one of the double-precision numbers in two consecutive memory locations, AL and AH, with AL holding the 16 low-order bits. The second number is placed in BL and BH. The program is listed in Table 2.15. The two low-order portions are added and the carry transferred into E. The AC is cleared and the bit in E is circulated into the least significant position of the AC.

The two high-order portions are then added to the carry and the double-precision sum is stored in CL and CH.

Table 2.15: Program to Add Two Double-Precision Numbers

```
        LDA AL    /Load A low
        ADD BL    /Add B low, carry in E
        STA CL    /Store in C low
        CLA       /Clear AC
        CIL       /Circulate to bring carry into AC(16)
        ADD AH    /Add A high and carry
        ADD BH    /Add B high
        STA CH    /Store in C high
        HLT
AL,     —         /Location of operands
AH,     —
BL,     —
BH,     —
CL,     —
CH,     —
```

4.5.2 Logic Operations

To perform logic operations: AND, CMA, and CLA, a basic computer has three machine instructions. The LDA instruction is considered as a logic operation that transfers a logic operand into the AC. We listed 16 different logic operations earlier, similarly all 16 logic operations can be implemented by software means because any logic function can be implemented using the AND and complement operations. For example, the OR operation is not available as a machine instruction in the basic computer. From DeMorgan's theorem we recognize the relation x + y = (x'y')'. The second expression contains only AND and complement operations. A program that forms the OR operation of two logic operands A and B is as follows:

```
LDA A     Load first operand A
CMA       Complement to get Ā
STA TMP   Store in a temporary location
LDA B     Load second operand B
CMA       Complement to get B̄
AND TMP   AND with Ā to get Ā ∧ B̄
CMA       Complement again to get A ∨ B
```

The other logic operations can be implemented by software in a similar fashion.

4.5.3 Shift Operations

In a basic computer, the circular-shift operations are machine instructions. The other interesting shifts are the logical shifts and arithmetic shifts. We can program these two shifts with a small number of instructions.

To perform the logical shift requires zeros are added to the extreme positions. This can be easily accomplished by clearing E and circulating the AC and E. Thus for a logical shift-right operation we need the two instructions

CLE

CIR

For a logical shift-left operation we need the two instructions
CLE
CIL
The arithmetic shifts depend on the type of representation of negative numbers. We adopt the signed-2's complement representation for the basic computer. For an arithmetic right-shift it is necessary that the sign bit in the leftmost position remain unchanged. But the sign bit itself is shifted into the high-order bit position of the number. The program for the arithmetic right-shift requires that we set E to the same value as the sign bit and circulate right, thus:

```
CLE  /Clear E to 0
SPA  /Skip if AC is positive; E remains 0
CME  /AC is negative; set E to 1
CIR  /Circulate E and AC
```

it is necessary for arithmetic shift-left that the added bit in the least significant position be 0. This can be done easily by clearing E prior to the circulate-left operation. The sign bit remains same during this shift. With a circulate instruction, the sign bit moves into E. It is then necessary to compare the sign bit with the value of E after the operation. If the two values are equal, the arithmetic shift has been correctly performed. If they are not equal, an overflow occurs. An overflow shows that the unshifted number was too large. When multiplied by 2 (by means of the shift), the number so obtained exceeds the capacity of the AC.

Paper Name: Computer Organization and Architecture

UNIT 5

CENTRAL PROGRESSING UNIT (CPU)

5.1 Stack Organization

The CPU of most computers comprises of a stack or last-in-first-out (LIFO) list wherein information is stored in such a manner that the item stored last is the first to be retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). A Stack Pointer (SP) is the register where the address for the stack is held because its value always points at the top item in the stack. The physical registers of a stack are always available for reading or writing unlike a stack of trays where the tray itself may be taken out or inserted because it is the content of the word that is inserted or deleted.

A stack has only two operations i.e. the insertion and deletion of items. The operation insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top. The deletion operation is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. In actual, nothing is exactly pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

5.1.1 Register Stack

There are two ways to place a stack. Either it can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. The organization of a 64-word register stack is exhibited in figure 5.3. A binary number whose value is equal to the address of the word that is currently on top of the stack is contained by the stack pointer register. Three items are placed in the stack - A, B and C in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by l, the result is 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The 1-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of micro-operations:

SP ← SP + 1	Increment stack pointer
M[SP] ← DR	Write item on top of the stack
If (SP= 0) then (FULL ← l)	Check if stack is full

The stack pointer is incremented so that it points to the address of the next-higher word. The word from DR is inserted into the top of the stack by the memory write operation. The M[SP] denotes the memory word specified by the address presently available in SP whereas the SP holds the address the top of the stack. The storage of the first item is done at address 1 whereas as the last item is store at address 0. If SP reaches 0, the stack is full of items, so FULL is set to 1. This condition is reached if the top item prior to the last push was in location 63 and after incrementing SP, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.

A new item is deleted from the stack if the stack is not empty (if EMTY <> 0). The pop operation consists of the following sequence of micro-operations:

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

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

5.1.2 Memory Stack

As shown in Fig. 5.3, stack can exist as a stand-alone unit or can be executed in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory. A portion of memory is assigned to a stack operation and a processor register is used as a stack pointer to execute stack in the CPU. Figure 5.4 shows a portion of computer memory partitioned into three segments - program, data, and stack. The address of the next instruction in the program is located by the program counter PC while an array of data is pointed by address register AR. The top of the stack is located by the stack pointer SP. The three registers are connected to a common address bus, which connects the three registers and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

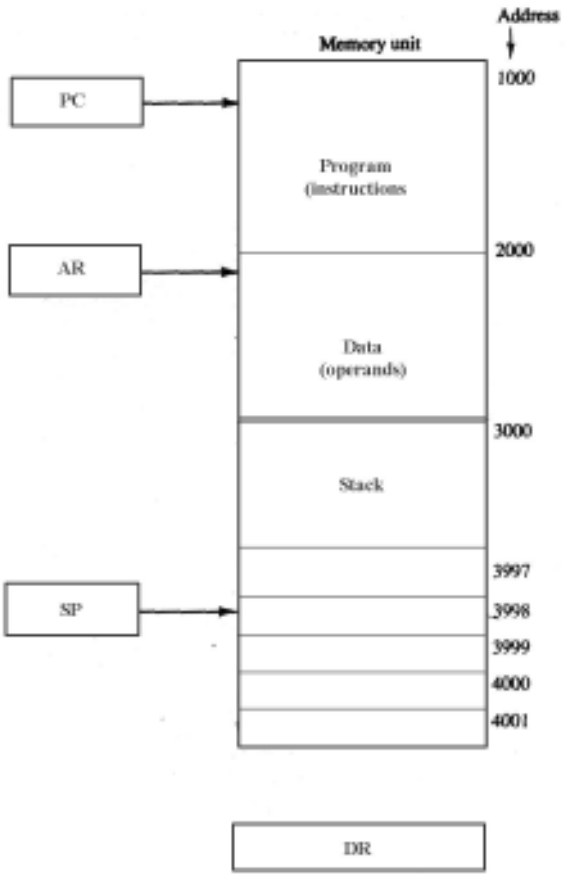


Figure 5.4: Computer memory with program, data, and slack segments.

Fig 5.4 displays the initial value of SP at 4001 and the growing of stack with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No checks are provided for checking stack limits.

We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:

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

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word form DR into the top of the stack. A new item is deleted with a pop operation as follows:

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

The top item is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

Most computers are not equipped with hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (40001 in this case). After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.

5.1.3 Reverse Polish Notation

Reverse Polish Notation is a way of expressing arithmetic expressions that avoids the use of brackets to define priorities for evaluation of operators. In ordinary notation, one might write
(3 + 5) * (7 - 2) and the brackets tell us that we have to add 3 to 5, then subtract 2 from 7, and multiply the two results together. In RPN, the numbers and operators are listed one after another, and an operator always acts on the most recent numbers in the list. The numbers can be thought of as forming a stack, like a pile of plates. The most recent number goes on the top of the stack. An operator takes the appropriate number of arguments from the top of the stack and replaces them by the result of the operation.

In this notation the above expression would be
3 5 + 7 2 - *
Reading from left to right, this is interpreted as follows:

- Push 3 onto the stack.
- Push 5 onto the stack. The stack now contains (3, 5).
- Apply the + operation: take the top two numbers off the stack, add them together, and put the result back on the stack. The stack now contains just the number 8.
- Push 7 onto the stack.
- Push 2 onto the stack. It now contains (8, 7, 2).
- Apply the - operation: take the top two numbers off the stack, subtract the top one from the one below, and put the result back on the stack. The stack now contains (8, 5).
- Apply the * operation: take the top two numbers off the stack, multiply them together, and put the result back on the stack. The stack now contains just the number 40.

Polish Notation was devised by the Polish philosopher and mathematician Jan Lucasiewicz (1878-1956) for use in symbolic logic. In his notation, the operators preceded their arguments, so that the expression above would be written as
* + 3 5 - 7 2

The 'reversed' form has however been found more convenient from a computational point of view.

5.2 Instruction Formats

- It is the function of the control unit within the CPU to interpret each instruction code
- The bits of the instruction are divided into groups called fields
- The most common fields are:
  - Operation code
  - Address field – memory address or a processor register
  - Mode field – specifies the way the operand or effective address is determined
- A register address is a binary number of *k* bits that defines one of 2^k registers in the CPU

- The instructions may have several different lengths containing varying number of addresses
- The number of address fields in the instruction format of a computer depends on the internal organization of its registers
- Most computers fall into one of the three following organizations:
  - Single accumulator organization
  - General register organization
  - Stack organization

- Single accumulator org. uses one address field

  ADD X : AC ← AC + M[X]
- The general register org. uses three address fields

  ADD R1, R2, R3: R1 ← R2 + R3
- Can use two rather than three fields if the destination is assumed to be one of the source registers
- Stack org. would require one address field for PUSH/POP operations and none for operation-type instructions

  PUSH X

  ADD
- Some computers combine features from more than one organizational structure

Example: X = (A+B) * (C + D)

5.2.1 Three-address instructions:

ADD	R1, A, B	R1 ← M[A] + M[B]
ADD	R2, C, D	R2 ← M[C] + M[D]
MUL	X, R1, R2	M[X] ← R1 * R2

5.2.2 Two-address instructions:

MOV	R1, A	R1 ← M[A]
ADD	R1, B	R1 ← R1 + M[B]
MOV	R2, C	R2 ← M[C]
ADD	R2, D	R2 ← R2 + D
MUL	R1, R2	R1 ← R1 * R2
MOV	X, R1	M[X] ← R1

5.2.3 One-address instructions:

LOAD	A	AC ← M[A]
ADD	B	AC ← AC + M[B]
STORE	T	M[T] ← AC
LOAD	C	AC ← M[C]
ADD	D	AC ← AC + M[D]
MUL	T	AC ← AC * M[T]
STORE	X	M[X] ← AC

5.2.4 Zero-address instructions:

PUSH	A	TOS ← A
PUSH	B	TOS ←B
ADD		TOS ← (A +B)
PUSH	C	TOS ← C
PUSH	D	TOS ← D
ADD		TOS ← (C + D)
MUL		TOS ← (C + D) * (A + B)
POP	X	M[X] ← TOS

5.2.5 RISC instructions:

LOAD	R1, A	R1 ← M[A]
LOAD	R2, B	R2 ← M[B]
LOAD	R3, C	R3 ← M[C]
LOAD	R4, D	R4 ← M[D]
ADD	R1, R1, R2	R1 ← R1 + R2
ADD	R3, R3, R4	R3 ← R3 + R4

```
MUL     R1, R1, R3    R1 ← R1 * R3
STORE  X, R1 M[X] ← R1
```

5.3 Addressing Modes

Many of the instructions which a computer actually executes during the running of a program concern the movement of data to and from memory. It is not possible simply to specify fixed addresses within each instruction, as this would require the location of data to be known at the time when the program was written. This is not possible for several reasons.

- When a program is read from disk, it will be put in memory in a position which cannot be predicted in advance. Hence, the location of any data in the program cannot be known in advance.
- Similarly, data which has been previously archived to files on disk or tape will be loaded into memory at a position which cannot be known in advance.
- If the data we wish to use will be read from an input device, then we cannot know in advance where in memory it will be stored.
- Many calculations involve performing the same operation repeatedly on a large quantity of data (for example, modifying an image which consists of over a million pixels). If each instruction operated on a fixed memory location, then the program would have to contain the same instruction many times, once for each pixel.

We therefore need different strategies for specifying the location of data.

Immediate addressing
The data itself, rather than an address, is given as the operand(s) of the instruction.

Direct or Absolute addressing
A fixed address is specified.

Implied addressing
The location of the data is implied by the instruction itself, so no operands need to be given. For example, a computer might have the instruction INCA, increment the accumulator.

Relative addressing
The location of the data is specified relative to the current value of the program counter. This is useful for specifying the location of data which is given as part of the program.

Indirect addressing

A memory location is given which holds another memory location. This second memory location holds the actual data. This mechanism solves the problems caused by reading data from file or an input device during program execution.

Indexed addressing

The location of the data is calculated as the sum of an address specified by one of the previous methods, and the value of an index register. This allows an array of data (for example, an image) to be accessed repeatedly by the same sequence of instructions.

- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced
- The decoding step in the instruction cycle determines the operation to be performed, the addressing mode of the instruction, and the location of the operands
- Two addressing modes require no address fields – the implied mode and immediate mode
- Implied mode: the operands are specified implicitly in the definition of the instruction – complement accumulator or zero-address instructions
- Immediate mode: the operand is specified in the instruction
- Register mode: the operands are in registers
- Register indirect mode: the instruction specifies a register that contains the address of the operand
- Auto increment or auto decrement mode: similar to the register indirect mode
- Direct address mode: the operand is located at the specified address given
- Indirect address mode: the address specifies the effective address of the operand
- Relative address mode: the effective address is the summation of the address field and the content of the PC
- Indexed addressing mode: the effective address is the summation of an index register and the address field
- Base register address mode: the effective address is the summation of a base register and the address field

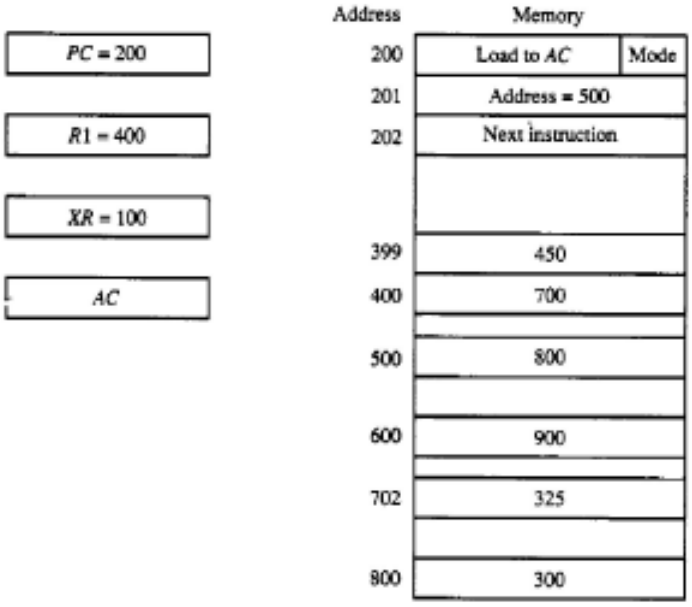


Figure 8-7 Numerical example for addressing modes.

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Data Transfer and Manipulation

- There is a basic set of operations that most computers include in their instruction set
- The opcode and/or symbolic code may differ for the same instruction among different computers

- There are three main categories of computer instructions:

- Data transfer
- Data manipulation
- Program control

- *Data transfer instructions*: transfer data from one location to another without changing the binary information content

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

- Some assembly language conventions modify the mnemonic symbol to differentiate between addressing modes
  LDI – load immediate
- Some use a special character to designate the mode

TABLE 8-6 Eight Addressing Modes for the Load Instruction

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

- *Data manipulation instructions*: perform arithmetic, logic, and/or shift operation

- Arithmetic instructions:

Increment	INC	Divide	DIV
Decrement	DEC	Add w/carry	ADDC
Add	ADD	Sub. w/borrow	SUBB
Subtract	SUB	Negate (2's comp)	NEG
Multiply	MUL		

- Some computers have different instructions depending upon the data type
  ADDI Add two binary integer numbers

ADDF	Add two floating point numbers
ADDD	Add two decimal numbers in BCD

- Logical and bit manipulation instructions:

Clear	CLR	Clear carry	CLRC
Complement	COM	Set carry	SETC
AND	AND	Comp. carry	COMC
OR	OR	Enable inter.	EI
Exclusive-OR	XOR	Disable inter.	DI

- Clear selected bits – AND instruction
- Set selected bits – OR instruction
- Complement selected bits – XOR instruction

- Shift instructions:

Logical shift right	SHR	Rotate right	ROR
Logical shift left	SHL	Rotate left	ROL
Arithmetic shift right	SHRA	ROR thru carry	RORC
Arithmetic shift left	SHLA	ROL thru carry	ROLC

OP REG TYPE RL COUNT

Program Control

- Program control instructions: provide decision-making capabilities and change the program path
- Typically, the program counter is incremented during the fetch phase to the location of the next instruction
- A program control type of instruction may change the address value in the program counter and cause the flow of control to be altered
- This provides control over the flow of program execution and a capability for branching to different program segments

Branch	BR	Return	RET
Jump	JMP	Compare	CMP
Skip	SKP	Test	TST
Call	CALL		

- TST and CMP cause branches based upon four status bits: C, S, Z, and V

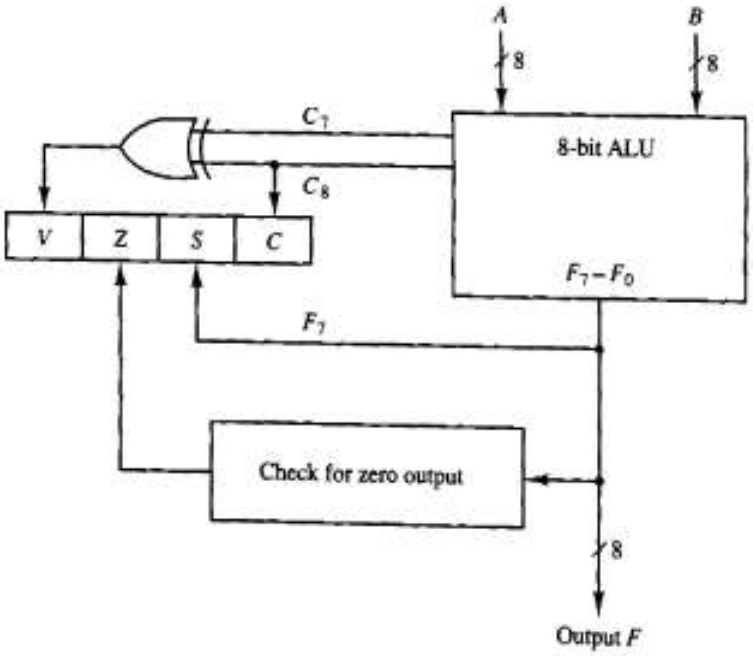


Figure 8-8 Status register bits.

TABLE 8-11 Conditional Branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

- A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine
- Execution of CALL:
  - Temporarily store return address
  - Transfer control to the beginning of the subroutine – update PC

```
SP      ← SP – 1
M[SP]   ← PC
PC      ← effective address
```

- Execution of RET:
  - Transfer return address from the temporary location to the PC

```
PC      ← M[SP]
SP      ← SP + 1
```

- *Program interrupt* refers to the transfer of program control to a service routine as a result of interrupt request

- Control returns to the original program after the service program is executed

- An interrupt procedure is similar to a subroutine call except:
    - The interrupt is usually initiated by an internal or external signal rather than an instruction
    - The address of the interrupt service routine is determined by the hardware rather than the address field of an instruction
    - All information necessary to define the state of the CPU is stored rather than just the return address

- The interrupted program should resume exactly as if nothing had happened
- The state of the CPU at the end of the execute cycle is determined from:
    - The content of the PC
    - The content of all processor registers
    - The content of certain status conditions

- The *program status word* (PSW) is a register that holds the status and control flag conditions
- Not all computers store the register contents when responding to an interrupt
- The CPU does not respond to an interrupt until the end of an instruction execution
- The control checks for any interrupt signals before entering the next fetch phase
- Three types of interrupts:
    - External interrupts
    - Internal interrupts
    - Software interrupts

- *External interrupts* come from I/O devices, timing devices, or any other external source
- *Internal interrupts* arise from illegal or erroneous use of an instruction or data, also called traps
- Internal interrupts are synchronous while external ones are asynchronous
- Both are initiated from signals that occur in the hardware of the CPU
- A *software interrupt* is initiated by executing an instruction

5.4 Reduced Instruction Set Computer

5.4.1 CISC characteristics

CISC, which stands for **Complex Instruction Set Computer**, is a philosophy for designing chips that are easy to program and which make efficient use of memory. Each instruction in a CISC instruction set might perform a series of operations inside the

processor. This reduces the number of instructions required to implement a given program, and allows the programmer to learn a small but flexible set of instructions.

Since the earliest machines were programmed in assembly language and memory was slow and expensive, the CISC philosophy made sense, and was commonly implemented in such large computers as the PDP-11 and the DECsystem 10 and 20 machines.

Most common microprocessor designs --- including the Intel(R) 80x86 and Motorola 68K series --- also follow the CISC philosophy.

As we shall see, recent changes in software and hardware technology have forced a re-examination of CISC. But first, let's take a closer look at the decisions which led to CISC.

5.4.2 CISCphilosophy1UseMicrocode

The earliest processor designs used dedicated (hardwire) logic to decode and execute each instruction in the processor's instruction set. This worked well for simple designs with few registers, but made more complex architectures hard to build, as control path logic can be hard to implement. So, designers switched tactics --- they built some simple logic to control the data paths between the various elements of the processor, and used a simplified microcode instruction set to control the data path logic. This type of implementation is known as a microprogrammed implementation.

In a microprogrammed system, the main processor has some built-in memory (typically ROM) which contains groups of microcode instructions which correspond with each machine-language instruction. When a machine language instruction arrives at the central processor, the processor executes the corresponding series of microcode instructions.

Because instructions could be retrieved up to 10 times faster from a local ROM than from main memory, designers began to put as many instructions as possible into microcode. In fact, some processors could be ordered with custom microcode which would replace frequently used but slow routines in certain application.

There are some real advantages to a microcoded implementation:

- since the microcode memory can be much faster than main memory, an instruction set can be implemented in microcode without losing much speed over a purely hard-wired implementation.
- new chips are easier to implement and require fewer transistors than implementing the same instruction set with dedicated logic, and...
- a microprogrammed design can be modified to handle entirely new instruction sets

quickly.

Using microcoded instruction sets, the IBM 360 series was able to offer the same programming model across a range of different hardware configurations.

Some machines were optimized for scientific computing, while others were optimized for business computing. However, since they all shared the same instruction set, programs could be moved from machine to machine without re-compilation (but with a possible increase or decrease in performance depending on the underlying hardware.)

This kind of flexibility and power made microcoding the preferred way to build new computers for quite some time.

CISCphilosophy2:Build"rich"instructionsets

One of the consequences of using a microprogrammed design is that designers could build more functionality into each instruction. This not only cut down on the total number of instructions required to implement a program, and therefore made more efficient use of a slow main memory, but it also made the assembly-language programmer's life simpler.
Soon, designers were enhancing their instruction sets with instructions aimed specifically at the assembly language programmer. Such enhancements included string manipulation operations, special looping constructs, and special addressing modes for indexing through tables in memory.

Forexample:

ABCDAddDecimalwithExtend
ADDAAddAddress
ADDXAddwithExtend
ASLArithmenticShiftLeft
CASCompareandSwapOperands
NBCDNegateDecimalwithExtend
EORILogicalExclusiveORImmediate
TAS Test Operand and Set

CISCphilosophy3:Buildhigh-levelinstructionsets

Once designers started building programmer-friendly instruction sets, the logical next step was to build instruction sets which map directly from high-level languages. Not only does this simplify the compiler writer's task, but it also allows compilers to emit fewer instructions per line of source code.

Modern CISC microprocessors, such as the 68000, implement several such instructions, including routines for creating and removing stack frames with a single call.

Forexample:

DBccTestCondition,DecrementandBranch
ROXLRotatewithExtendLeft
RTRReturnandRestoreCodes
SBCDSubtractDecimalwithExtend
SWAPSwapregisterWords
CMP2 Compare Register against Upper and Lower Bounds

TheriseofCISC

CISCDesignDecisions:

- usemicrocode
  - buildrichinstructionsets
  - buildhigh-levelinstructionsets

Taken together, these three decisions led to the CISC philosophy which drove all computer designs until the late 1980s, and is still in major use today. (Note that "CISC" didn't enter the computer designer's vocabulary until the advent of RISC --- it was simply the way that everybody designed computers.)

- The next lesson discusses the common characteristics that all CISC designs share, and how those characteristics affect the operation of a CISC machine.

The disadvantages of CISC

Still, designers soon realized that the CISC philosophy had its own problems, including:

- Earlier generations of a processor family generally were contained as a subset in every new version --- so instruction set & chip hardware become more complex with each generation of computers.

- So that as many instructions as possible could be stored in memory with the least possible wasted space, individual instructions could be of almost any length---this means that different instructions will take different amounts of clock time to execute, slowing down the overall performance of the machine.

- Many specialized instructions aren't used frequently enough to justify their existence --- approximately 20% of the available instructions are used in a typical program.

- CISC instructions typically set the condition codes as a side effect of the instruction. Not only does setting the condition codes take time, but programmers have to

remember to examine the condition code bits before a subsequent instruction changes them.

RISC characteristics

The design of the instruction set for the processor is very important in terms of computer architecture. It's the instruction set of a particular computer that determines the way that machine language programs are constructed. Computer hardware is improvised by various factors, such as upgrading existing models to provide more customer applications adding instructions that facilitate the translation from high-level language into machine language programs and striving to develop machines that move functions from software implementation into hardware implementation. A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated as CISC.

An important aspect of computer architecture is the design of the instruction set for the processor
The instruction set determines the way that machine language programs are constructed

- Many computers have instructions sets of about 100 - 250 instructions
- These computers employ a variety of data types and a large number of addressing modes – complex instruction set computer (CISC)
- A RISC uses fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often

- The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language
- The major characteristics of CISC architecture are:
    - Large number of instructions
    - Some instructions that perform specialized tasks and are used infrequently
    - Large variety of addressing modes
    - Variable length instruction formats
    - Instructions that manipulate operands in memory
- The goal of RISC architecture is to reduce execution time by simplifying the instructions set

RISC Characteristics
The essential goal of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

- Relatively few instructions.
- Relatively few addressing modes.
- Memory access limited to load and store instructions.

- All operations done within the registers of the CPU.
- Fixed length easily decoded instruction format.
- Single-cycle instruction execution.
- Hardwired rather than microprogrammed control.

A typical RISC processor architecture includes register-to-register operations, with only simple load and store operations for memory access. Thus the operand is code into a processor register with a load instruction. All computational tasks are performed among the data stored in processor registers and with the help of store instructions results are transferred to the memory. This architectural feature simplifies the instruction set and encourages the optimization of register manipulation. Almost all instructions have simple register addressing so only a few addressing modes are utilised. Other addressing modes may be included, such as immediate operands and relative mode. An advantage of RISC instruction format is that it is easy to decode.
An important feature of RISC processor is its ability to execute one instruction per clock cycle. This is done by a procedure referred to as pipelining. A load or store instruction may need two clock cycles because access to memory consumes more time than register operations. Other characteristics attributed to RISC architecture are:

- A relatively large number of register in the processor unit.
- Use of overlapped register windows to speed-up procedure call and return.
- Efficient instruction pipeline.
- Compiler support for efficient translation of high-level language programs into machine language programs.

- *Overlapped register windows* are used to pass parameters and avoids the need for saving and restoring register values during procedure calls
- Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window
- Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results

- Example: system with 74 registers and four procedures
  - Each procedure has a total of 32 registers while active
  - 10 global registers
  - 10 local registers
  - 6 low overlapping registers
  - 6 high overlapping registers
- Relationships of register windows
  - # of global registers = G
  - # of local registers in each window = L
  - # of common registers to two windows = C
  - # of windows = W
  - window size = L + 2C + G

- total # of registers = (L + C)W + G

The first CPUs had a so called *Complex Instruction Set Computer* (CISC). This means that the computer can understand many and complex instructions. The X86 instruction set, with its varying length from 8 to 120 bit, was originally developed for the 8086 with its mere 29000 transistors.

More instructions have been added within new generations of CPUs. The 80386 had 26 new instructions, the 486 added 6 and the Pentium another 8 new instructions. This meant, that programs had to be rewritten to use these new instructions. This happened for example with new versions of Windows . Hence, some programs require a 386 or a Pentium processor to function.

Paper Name: Computer Organization and Architecture

UNIT 6

INPUT-OUTPUT ORGANIZATION

6.1.1 Programmed I/O

The simplest strategy for handling communication between the CPU and an I/O module is *programmed I/O*. Using this strategy, the CPU is responsible for all communication with I/O modules, by executing instructions which control the attached devices, or transfer data.

For example, if the CPU wanted to send data to a device using programmed I/O, it would first issue an instruction to the appropriate I/O module to tell it to expect data. The CPU must then wait until the module responds before sending the data. If the module is slower than the CPU, then the CPU may also have to wait until the transfer is complete. This can be very inefficient.

Another problem exists if the CPU must read data from a device such as a keyboard. Every so often the CPU must issue an instruction to the appropriate I/O module to see if any keys have been pressed. This is also extremely inefficient. Consequently this strategy is only used in very small microprocessor controlled devices.

6.1.2 Interrupt Driven I/O

Virtually all computers provide a mechanism y which other modules (I/O, memory) may interrupt the normal processing of the CPU. Table 3.1 lists the most common classes of interrupts. The specific nature of these interrupts is examined later in this book, specially in chapters 6 and 11. However, we need to introduce the concept now in order to understand more clearly the nature of the instruction cycle and the implications of interrupts on the interconnection structure. The reader need not be concerned at this stage about the details of the generation and processing of interrupts, but only focus on the communication between modules that results from interrupts.
Interrupts are provided primarily as a way to improve processing efficiency. For example, most external devices are much slower than the processor. Support that

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor, This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure such as power failure or memory parity error.

The processor is transferring data to a printer using the instruction cycle scheme of Figure 3.3. After each write operation, the processor will have to pause and remain idle until the printer catches up. He length of this pause can be on the order of many hundreds or even thousands of instruction cycles that do not involve memory. Clearly, this is a very wasteful use of the processor. With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.

Figure 3.7a illustrates this state of affairs for the application referred to in the preceding paragraph. He user program performs a series of WRITE calls inter-leaved with processing. Code segments 1,2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are calls to an I/O program consists of three sections:

- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer, and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function. He program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.

- A sequence of instructions, labeled 5 in the figure, to complete the operation, This may include setting a flag indicating the success or failure of the operation.

Because the I/P operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at point of the WRITE call for some considerable period of time.

A more common strategy is to use *interrupt driven I/O*. This strategy allows the CPU to carry on with its other operations until the module is ready to transfer data. When the CPU wants to communicate with a device, it issues an instruction to the appropriate I/O module, and then continues with other operations. When the device is ready, it will *interrupt* the CPU. The CPU can then carry out the data transfer as before.

This also removes the need for the CPU to continually poll input devices to see if it must read any data. When an input device has data, then the appropriate I/O module can interrupt the CPU to request a data transfer.

An I/O module interrupts the CPU simply by activating a control line in the control bus. The sequence of events is as follows.

1. The I/O module interrupts the CPU.
2. The CPU finishes executing the current instruction.
3. The CPU acknowledges the interrupt.
4. The CPU saves its current state.
5. The CPU jumps to a sequence of instructions which will handle the interrupt.

The situation is somewhat complicated by the fact that most computer systems will have several peripherals connected to them. This means the computer must be able to detect which device an interrupt comes from, and to decide which interrupt to handle if several occur simultaneously. This decision is usually based on *interrupt priority*. Some devices will require response from the CPU more quickly than others, for example, an interrupt from a disk drive must be handled more quickly than an interrupt from a keyboard.

Many systems use multiple interrupt lines. This allows a quick way to assign priorities to different devices, as the interrupt lines can have different priorities. However, it is likely that there will be more devices than interrupt lines, so some other method must be used to determine which device an interrupt comes from.

Most systems use a system of *vectored interrupts*. When the CPU acknowleges an interrupt, the relevant device places a word of data (a vector) on the data bus. The vector identifies the device which requires attention, and is used by the CPU to look up the address of the appropriate interrupt handing routine.

Memory Mapped and Isolated I/O

Whether a system uses programmed or interrupt driven I/O, it must still periodically send instructions to the I/O modules. Two methods are used for to implement this: *memory-mapped I/O* and *isolated I/O*.

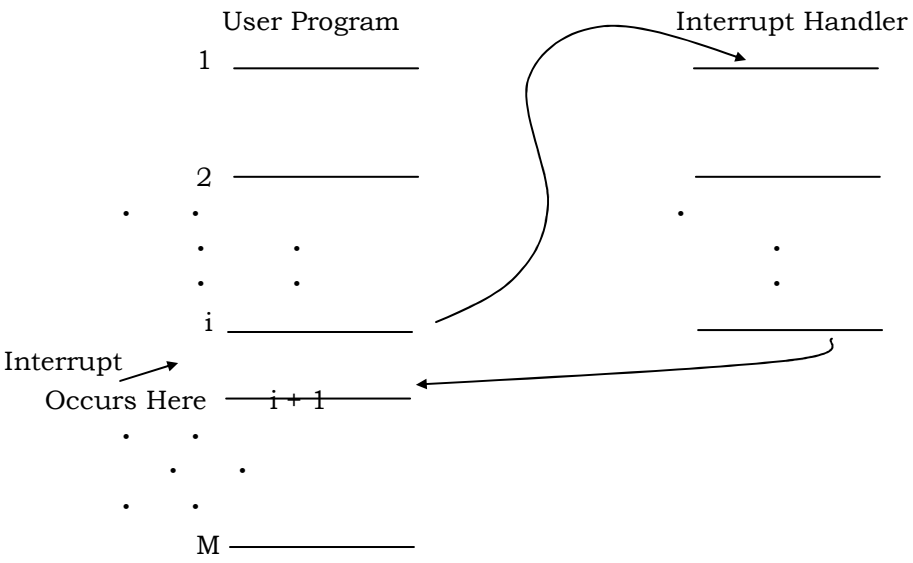
With memory-mapped I/O, the I/O modules appear to the CPU as though they occupy locations in main memory. To send instructions or transfer data to an I/O module, the CPU reads or writes data to these memory locations. This will reduce the available address space for main memory, but as most modern systems use a wide address bus this is not normally a problem.

With isolated I/O, the I/O modules appear to occupy their own address space, and special instructions are used to communicate with them. This gives more address space for both memory and I/O modules, but will increase the total number of different instructions. It may also reduce the flexibility with which the CPU may address the I/O modules if less addressing modes are available for the special I/O instructions.

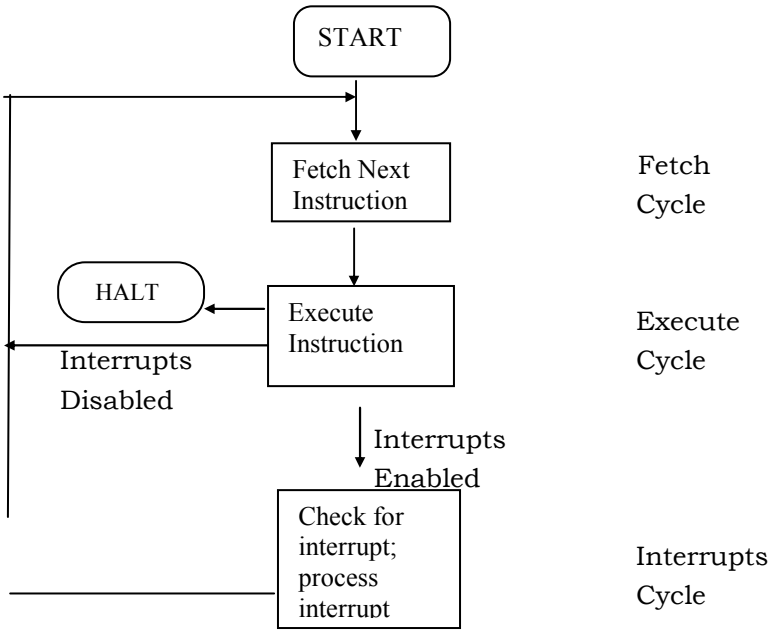
Interrupts and the Instruction Cycle

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 3.7b.As before , the user program reaches a point at which it makes a system call in the form of a WRITE call . The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.
When the external device becomes ready to be serviced, that is , when it is ready to accept more data from the processor , the I/O module for that external device sends an interrupt request signal to the processor. The processor responds by suspending operation of the current program , branching off to a program to service that particular I/O device, known as an interrupt handler, and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by an asterisk (*) in Figure.

From the point of view of the user program, an interrupt is just that : an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure). Thus the user program does not have to contain any special code to accommodate interrupts ; the processor and the operating system are responsible for suspending the user program and then resuming it at the same point.



6 Figure Transfer of control via interrupts



7 Figure Instruction cycle with interrupts

To accommodate interrupts, an interrupt cycle is added to the instruction cycle, as shown in Figure 3.9. In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next

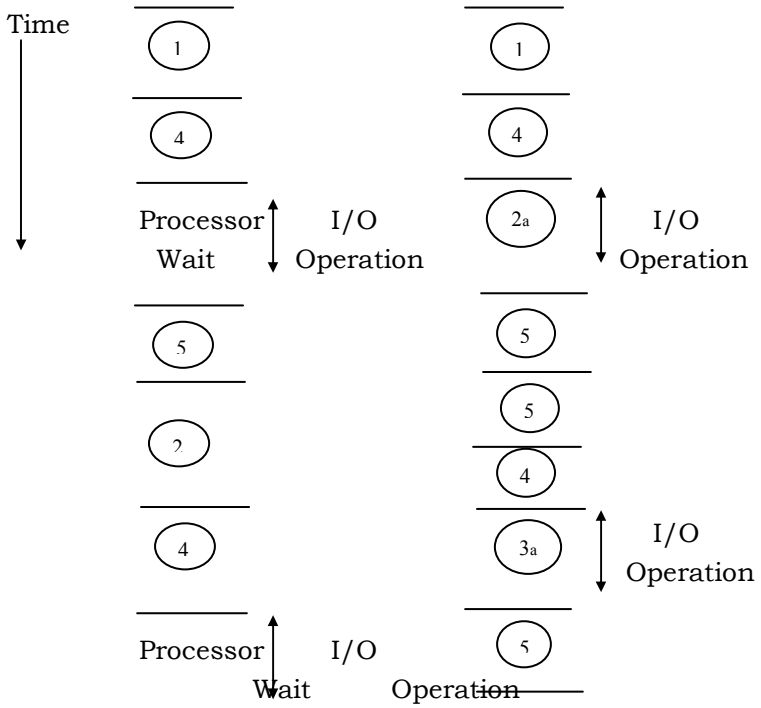
instruction of the current program. If an interrupt is pending, the processor does the following:

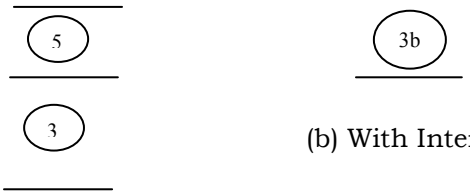
1. It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
2. It sets the program counter to the starting address of an interrupt handler routine.

The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this program determines the nature of the interrupt and performs whatever actions are needed.

For example, in the example we have been using, the handler determines which I/O module generated the interrupt, and may branch to a program that will write more data out to that I/O module. When the interrupt handler routine is completed, the processor can resume execution of he user program at the point of interruption.

It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt an to decide on the appropriate action. Nevertheless, because of the relatively large amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much ore efficiently with the use of interrupts.





(a) Without Interrupts

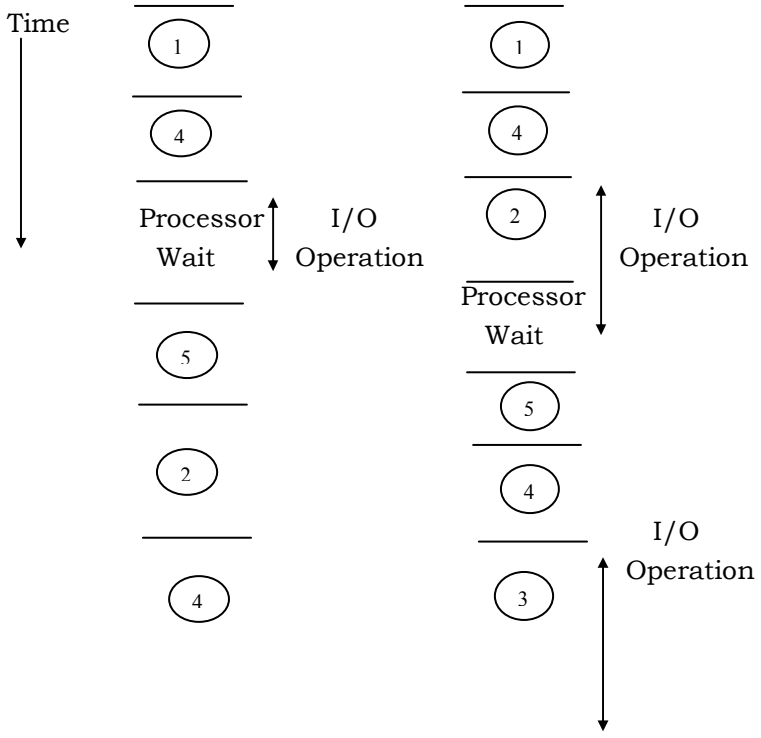
(b) With Interrupts

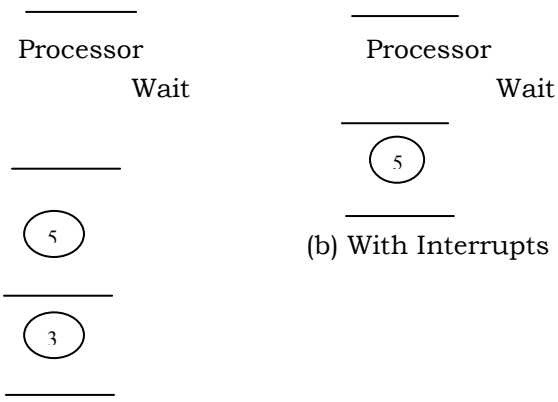
8 Figure Program timing; short I/O wait

To appreciate the gain in efficiency, consider Figure which is a timing diagram based on the flow of control in Figures.

Figure assume that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program. The more typical case, especially for a slow device such as a printer, is that the I/o operation will take much more time than executing a sequence of user instructions. Figure 3.7c indicates this state of affairs. In this case, the user program reaches the second RITE call before the I/O operation spawned by the first call is complete. The result is that the user program is hung up at that point. When the preceding I/O operation is completed, this new WRITE call may b processed, and a new I/O operation may be started. Figure 3.11 shows the timing for this situation with and without the use of interrupts. We can see that there is still a gain in efficiency because part of the time during which the I/O operation is under way overlaps with the execution of user instructions.

Figure shows a revised instruction cycle state diagram that includes interrupt cycle processing.





(a) Without Interrupts

(b) With Interrupts

9 Figure Program timing; short I/O wait

10 Multiple Interrupts

The discussion so far has only discussed the occurrence of a single interrupt. Suppose, however, that multiple interrupts can occur. For example, a program may be receiving data from a communications line and printing results. The printer will generate an interrupt every time that it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives. The unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communication interrupt to occur while a printer interrupt is being processed.

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A disabled interrupt simply means that the processor can and will ignore that interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has enabled interrupts. Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the ther interrupt handler routine completes, interrupts are enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is nice and simple, as interrupts are handled in strict sequential order (figure)

The drawback to the above approach is that it does not take into account relative priority or time-critical needs. For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more input. If the first batch of input has not been processed before the second batch arrives, data may be lost.

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be itself interrupted(figure).

As an example of this second approach, consider a system with three I/O devices: a printer, a disk , and a communications line, with increasing priorities of 2,4 and 5 respectively. Figure 3.14 illustrates a possible sequence. A user program begins at t = 0 . At t =10, a printer interrupt occur ; user information is placed on the system stack, and execution continues at the printer interrupt service routine (ISR). While this routine is still executing, at t = 15, a communications interrupt occurs. Since the communications line has higher priority than the printer, the interrupt is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing , a disk interrupt occurs (t = 20). Since this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.

When the communications ISR is complete (t = 25), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors he higher-priority disk interrupt and control transfers to the disk ISR. Only when that routine is complete (t = 35) is the printer ISR resumed. When that routine completes (t = 40), control finally returns to the user program.

6.2 Priority Interrupt

A priority interrupt establishes a priority to decide which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests, which if delayed or interrupted, could have serious consequences. Devices with high- speed transfers are given high priority, and slow devices receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first. Establishing the priority of simultaneous interrupts can be done by software or hardware. We can use a polling procedure to identify the highest-priority. There is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. We test the highest-priority source first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower-priority source is tested, and so on. Thus the initial service routine interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest-priority device among all devices that interrupted the computer.

6.2.1 Daisy-Chaining Priority

The daisy-chaining method has a serial connection of all devices that request an interrupt. The device with the highest priority is kept in the first position, followed by lower-priority devices and so on. This method of connection is shown in Fig.6.32. The interrupt request line is common to all devices and forms a wired logic connection. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts are pending, the interrupt line stays in the high-level state and as a result CPU does not recognize any interrupt. This is equivalent to a negative logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt. If device I has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.

A device with PI=0 input generates a 0 in its PO output to inform the next-lower-priority device that the acknowledge signal has been blocked. A device that makes a request for an interrupt and has a I in its Pi input will intercept the acknowledge signal by placing a 0 in its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output. Thus the device with PI = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

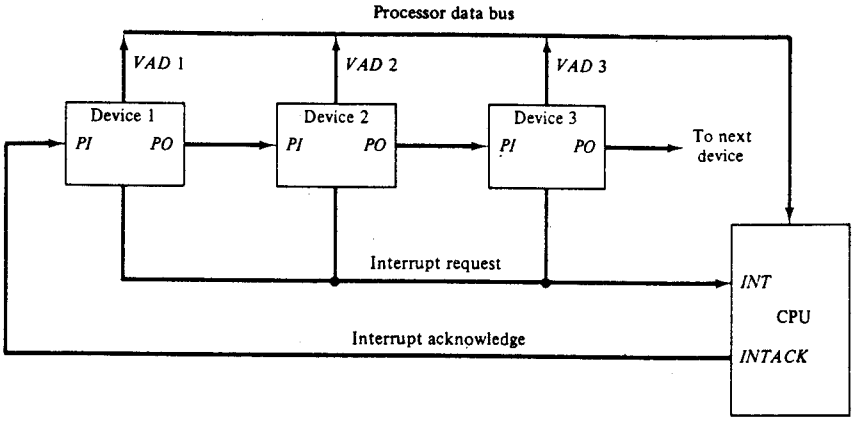


Figure 6.32: Daisy-chain priority interrupt

Figure 6.33 shows the internal logic that must be included within each device when connected in the daisy-chaining scheme. The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter (a circuit that provides the wired logic for the common interrupt line). If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF. If PI = 1 and RF= 0, then PO = 1 and the vector address is disabled. This condition passes the

acknowledge signal to the next device through PO. The device is active when PI = 1 and RF = 1. This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

6.2.2 Parallel Priority Interrupt

The method uses a register whose bits are set separately by the interrupt signal from each device. Now we establish the priority according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register whose purpose is to control the status of each interrupt request. The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the while a lower-priority device is being serviced.

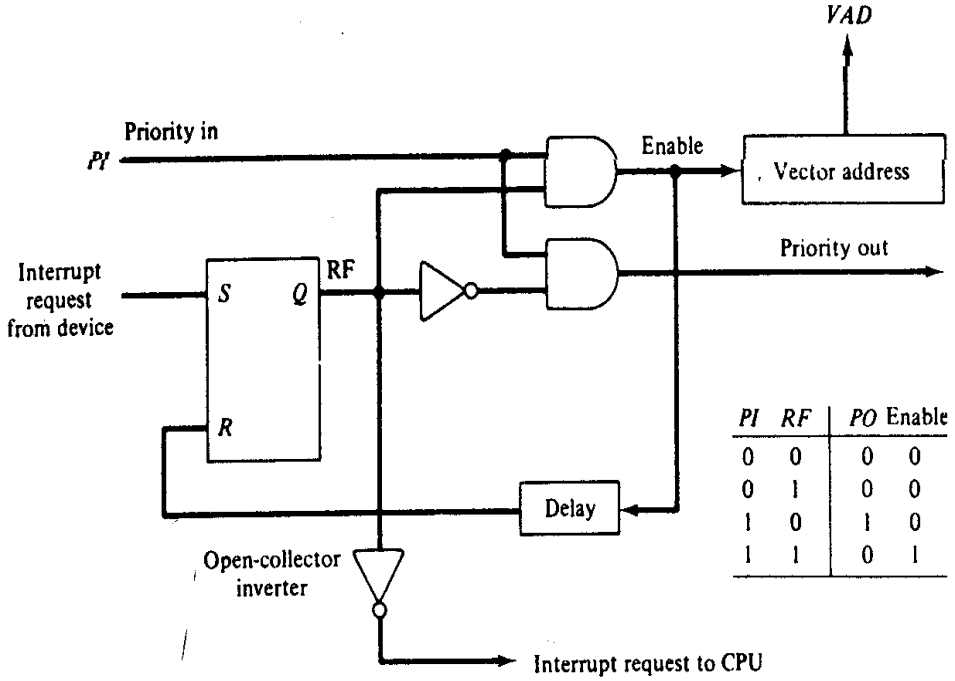


Figure 6.33: One stage of the daisy-chain priority arrangement

Fig. 6.34 shows the priority logic for a system of four interrupt sources. It has an interrupt register. The bits of this register are external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only

if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs. The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system. The outputs of 1ST AND with IEN provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal form the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus. We will now explain the priority encoder circuit and then discuss the interaction between the priority interrupt controller and the CPU.

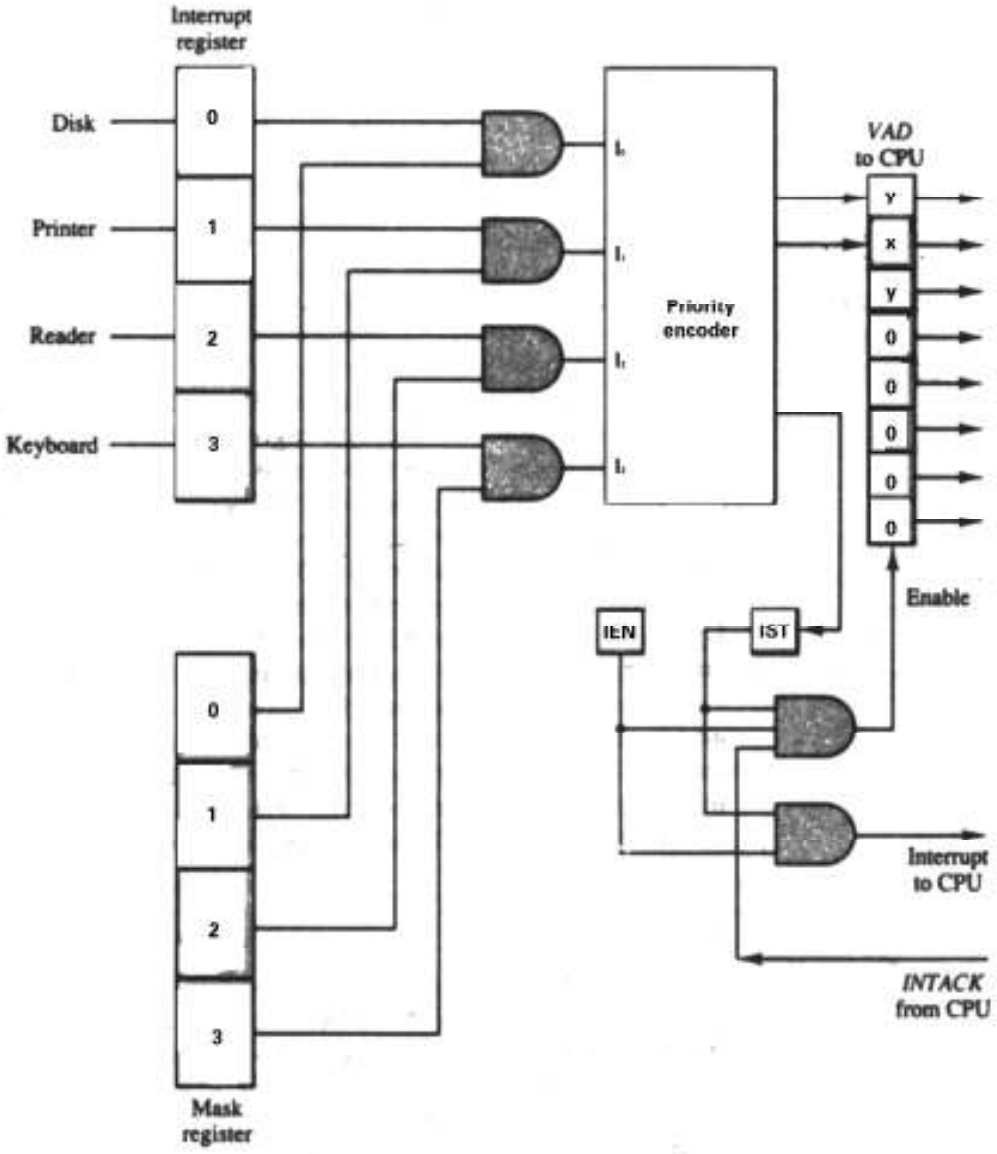


Figure 6.34: Priority interrupt hardware

6.2.3 Interrupt Cycle

The interrupt makes flip-flop IEN so that can be set or cleared by program instructions. When IEN is cleared, the interrupt request coming from 1ST is neglected by the CPU. The program-controlled IEN bit allows the programmer to choose whether to use the interrupt facility. If an instruction to clear IEN has been inserted in the program, it means that the user does not want his program to be interrupted. An instruction to set IEN indicates that the interrupt facility will be used while the current program is running. Most computers include internal hardware that clears IEN to 0 every time an interrupt is acknowledged by the processor.

CPU checks IEN and the interrupt signal from IST at the end of each instruction cycle the. If either 0, control continues with the next instruction. If both IEN and IST are equal to 1, the CPU goes to an interrupt cycle. During the interrupt cycle the CPU performs the following sequence of micro-operations:

SP ← SP – 1	Decrement stack pointer
M [SP] ← PC	Push PC into stack
INTACK ← 1	Enable interrupt acknowledge
PC ← VAD	Transfer vector address to PC
IEN ← 0	Disable further interrupts
Go to fetch next instruction	

The return address is pushed from PC into the stack. It then acknowledges the interrupt by enabling the INTACK line. The priority interrupt unit responds by placing a unique interrupt vector into the CPU data bus. The CPU transfers the vector address into PC and clears IEN prior to going to the next fetch phase. The instruction read from memory during the next fetch phase will be the one located at the vector address.
Software Routines

A priority interrupt system uses both hardware and software techniques. Now we discuss the software routines for this. The computer must also have software routines for servicing the interrupt requests and for controlling the interrupt hardware registers. Figure 6.35 shows the programs that must reside in memory for handling the interrupt system. Each device has its own service program that can be read through a jump

(JMP) instruction stored at the assigned vector address. The symbolic name of each routine represents the starting address of the service program. The stack shown in the diagram is used for storing the return address after each interruption.

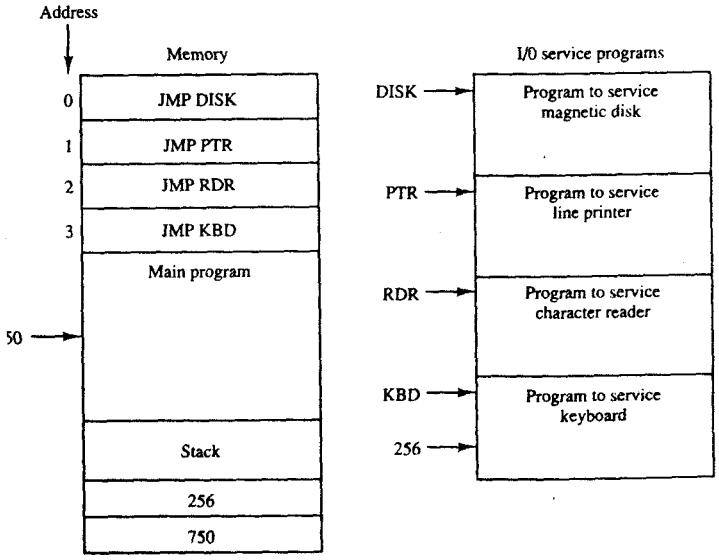


Figure 6.35: Programs stored in memory for servicing interrupts

Now we take an example to illustrate it. Let the keyboard sets interrupt bit while the CPU is executing the instruction in location 749 of main program. At the end of the instruction cycle, the computer goes to interrupt cycle. It stores the return address 750 in the stack and then accepts the vector address 00000011 from the bus and transfers it to PC. The instruction in location 3 is executed next, resulting in transfer of control to the KBD routine. Now suppose that the disk sets its interrupt bit when the CPU is executing the instruction at address 255 in the KBD program. Address 256 is pushed into the stack and control is transferred to the DISK service program. The last instruction in each routine is a return from interrupt instruction. When the disk service program is completed, the return instruction pops the stack and places 256 into PC. This returns control to the KBD routine to continue servicing the keyboard. At the end of the KBD program, the last instruction pops the stack and returns control to the main program at address 750. Thus, a higher-prior device can interrupt a lower-priority device. It is assumed that the time spent in servicing the high-priority interrupt is short compared to the transfer rate of the low-priority device so that no loss of information takes place.

Initial and Final Operations

We should remember that the interrupt enable IEN is cleared at the end of an interrupt cycle. This flip-flop must be set again to enable higher-priority interrupt requests, but not before lower-priority interrupts are disabled. The initial sequence of each interrupt

service routine must have instructions to control the interrupt hardware in the following manner:

1. Clear lower-level mask register bits.
2. Clear interrupt status bit 1ST.
3. Save contents of processor registers.
4. Set interrupt enable bit IEN.
5. Proceed with service routine.

The final sequence in each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

1. Clear interrupt enable bit IEN.
2. Restore contents of processor registers.
3. Clear the bit in the interrupt register belonging to the source that has been serviced.
4. Set lower-level priority bits in the mask register.
5. Restore return address into PC and set IEN.

6.3 Direct Memory Access

Although interrupt driven I/O is much more efficient than program controlled I/O, all data is still transferred through the CPU. This will be inefficient if large quantities of data are being transferred between the peripheral and memory. The transfer will be slower than necessary, and the CPU will be unable to perform any other actions while it is taking place.

6.3.1 DMA Controller

Many systems therefore use an additional strategy, known as direct memory access (DMA). DMA uses an additional piece of hardware - a DMA controller. The DMA controller can take over the system bus and transfer data between an I/O module and main memory without the intervention of the CPU. Whenever the CPU wants to transfer data, it tells the DMA controller the direction of the transfer, the I/O module involved, the location of the data in memory, and the size of the block of data to be transferred. It can then continue with other instructions and the DMA controller will interrupt it when the transfer is complete.

The CPU and the DMA controller cannot use the system bus at the same time, so some way must be found to share the bus between them. One of two methods is normally used.

Burst mode

The DMA controller transfers blocks of data by halting the CPU and controlling the system bus for the duration of the transfer. The transfer will be as quick as the weakest

link in the I/O module/bus/memory chain, as data does not pass through the CPU, but the CPU must still be halted while the transfer takes place.
Cycle stealing

The DMA controller transfers data one word at a time, by using the bus during a part of an instruction cycle when the CPU is not using it, or by pausing the CPU for a single clock cycle on each instruction. This may slow the CPU down slightly overall, but will still be very efficient.

1. Channel I/O

This is a system traditionally used on mainframe computers, but is becoming more common on smaller systems. It is an extension of the DMA concept, where the DMA controller becomes a full-scale computer system itself which handles all communication with the I/O modules.

2. I/O Interfaces

The interface of an I/O module is the connection to the peripheral(s) attached to it. The interface handles synchronisation and control of the peripheral, and the actual transfer of data. For example, to send data to a peripheral, the sequence of events would be as follows.

a) The I/O module sends a control signal to the peripheral requesting permission to send data.
b) The peripheral acknowledges the request.
c) The I/O module sends the data (this may be either a word at a time or a block at a time depending on the peripheral).
d) The peripheral acknowledges receipt of the data.

This process of synchronisation is known as *handshaking*.

The internal buffer allows the I/O module to compensate for some of the difference in the speed at which the interface can communicate with the peripheral, and the speed of the system bus.

I/O interfaces can be divided into two main types.

3. Parallel interfaces

There are multiple wires connecting the I/O module to the peripheral, and bits of data are transferred simultaneously, as they are over the data bus. This type of interface is used for high speed peripherals such as disk drives.

4. Serial interfaces

Only a single wire connects the I/O module to the peripheral, and data must be transferred one bit at a time. This is used for slower peripherals such as printers and keyboards.

5. I/O Function

Thus far, we have discussed the operation of the computer a s controlled by the CPU, and we have looked primarily at the interaction of CPU and memory. The discussion has only alluded to the role of the I/O component. This role is discussed in detail in chapter 6, but a brief summary is in order here.

An I/O module can exchange data directly with the CPU. Just as the CPU can initiate a read or write with memory, designating the address of a specific location, the CPU can also read data form or write data to an I/O module. In this latter case, the CPU identifies a specific device that is controlled by a particular I/O module. Thus, an instruction sequence similar in form to that of Figure 3.5 could occur, with I/O instructions rather than with memory-referencing instructions.

In some cases, it is desirable to allow I.O exchanges to occur directly with memory. In such a case, the CPU grants to an I/O module the authority to read from or write to memory, so that the I.O memory transfer can occur without tying up the CPU. During such a transfer, the I/O module issues read or write commands to memory, relieving the CPU of responsibility for the exchange. This operation is known as direct memory access (DMA), an it will be examined in detail in chapter 6. For now, all that we need to know is that the interconnection structure of the computer may need to allow for direct memory – I/O interaction.

This section introduces the concepts of input/output devices, modules and interfaces. It considers the various strategies used for communication between the CPU and I/O modules, and the interface between an I/O module and the device(s) connected to it. Some common I/O devices are considered in the last section.

6.3.2 DMA Transfer

There are three independent channels for DMA transfers. Each channel receives its trigger for the transfer through a large multiplexer that chooses from among a large number of signals. When these signals activate, the transfer occurs. The DMAxTSELx bits of the DMA Control Register 0 (DMACTL0). The DMA controller receives the trigger signal but will ignore it under certain conditions. This is necessary to reserve the memory bus for reprogramming and non-maskable interrupts etc. The controller also handles conflicts for simultaneous triggers. The priorities can be adjusted using the DMA Control Register 1 (DMACTL1). When multiple triggers happen simultaneously, they occur in order of module priority. The DMA trigger is then passed to the module whose trigger activated. The DMA channel will copy the data from the starting memory location or block to the destination memory location or block. There are many variations on this, and they are controlled by the DMA Channel x Control Register (DMAxCTL):

Single Transfer - each trigger causes a single transfer. The module will disable itself when DMAxSZ number of transfers have occurred (setting it to zero prevents transfer). The DMAxSA and DMAxDA registers set the addresses to be transferred to and from. The DMAxCTL register also allows these addresses to be incremented or decremented by 1 or 2 bytes with each transfer. This transfer halts the CPU.

Block Transfer - an entire block is transferred on each trigger. The module disables itself when this block transfer is complete. This transfer halts the CPU, and will transfer each memory location one at a time. This mode disables the module when the transfer is complete.

Burst-Block Transfer - this is very similar to Block Transfer mode except that the CPU and the DMA transfer can interleave their operation. This reduces the CPU to 20% while the DMA is going on, but the CPU will not be stopped altogether. The interrupt occurs when the block has completely transferred. This mode disables the module when the transfer is complete.

Repeated Single Transfer - the same as Single Transfer mode above except that the module is not disabled when the transfer is complete.

Repeated Block Transfer - the same as Block Transfer mode above except that the module is not disabled when the transfer is complete.

Repeated Burst-Block Transfer - the same as Burst Block Transfer mode above except that the module is not disabled when the transfer is complete.

Writing to flash requires setting the DMAONFETCH bit. If this is not done, the results of the DMA operation are "unpredictable." Also, the behavior and settings of the DMA module should only be modified when the module is disabled. The setting and triggers are highly configurable, allowing both edge and level triggering. The variety of settings is detailed in the DMA chapter of the users guide. Also, it is important to note that interrupts will not be acknowledged during the DMA transfer because the CPU is not active. Each DMA channel has its own flag, but the interrupt vector is shared with the DAC. This necessitates some software checking to handle interrupts with both modules enabled.

6.4 Input-output Processor (IOP)

The CPU or processor is the part that makes the computer smart. It is a single integrated circuit referred to as a microprocessor. The earlier microprocessors were Intel 8080 or 8086, they were very slow. Then came faster models from Intel such as 80286, 80386, 80486 and now Pentium processors. Each of these vary in speed of their operation. The AT compatibles – 80286 onwards, run in one of the two modes:

- Real mode

- Protected mode

The processor complex is the name of the circuit board that contains the main system processor and any other circuitry directly related to it, such as clock control, cache, and so forth. The processor complex design allows the user to easily upgrade the system later to a new processor type by changing one card. In effect, it amounts to a modular motherboard with a replaceable processor section.

Latest designs all incorporate the upgradable processor. Intel has designed all 486, Pentium, Pentium MMX, and Pentium Pro processors to be upgradable to faster (sometimes called OverDrive) processors in the future by simply swapping (or adding) the new processor chip. Changing only the processor chip for a faster one is the easiest and generally most cost-effective way to upgrade without changing the entire motherboard.

A computer may incorporate one or more external processors and assign them the task of communicating directly with all I/O devices. An input-output processor (IOP) may be classified as a processor with direct memory access capability that communicates with I/O devices. In this configuration, the computer system can be divided into a memory unit, and a number of processors comprised of the CPU and one or more IOPs. Each IOP takes care of input and output tasks, relieving the CPU from the housekeeping chores involved in I/O transfers.

The IOP is similar to a CPU except that it is designed to handle the details of I/O processing. Unlike the DMA controller that must be set up entirely by the CPU, the IOP can fetch and execute its own instructions. IOP instructions are specially designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks, such as arithmetic, logic, branching, and code translation.

The block diagram of a computer with two processors is shown in Figure 6.39. The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transfer of data between various peripheral deices and the memory unit.

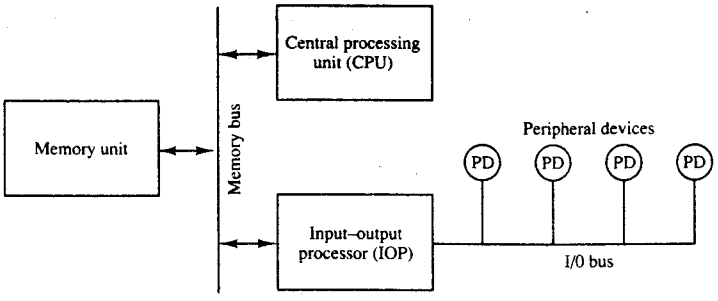


Figure 6.39: Block diagram of a computer with I/O processor

The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory. Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program. After the input data are assembled into a memory word, they are transferred from IOP directly into memory by "stealing" one memory cycle from the CPU. Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output device at the device rate and bit capacity.

The communication between the IOP and the devices attached to it is similar to the program control method of transfer. The way by which the CPU and IOP communicate depends on the level of sophistication included in the system. In most computer systems, the CPU is the master while the IOP is a slave processor. The CPU is assigned the task of initiating all operations, but I/O instructions are execute in the IOP. CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities. The IOP, in turn, typically asks for CPU attention by means of an interrupt. It also responds to CPU requests by placing a status word in a prescribed location in memory to be examined later by a CPU program. When an I/O operation is desired, the CPU informs the IOP where to find the I/O program and then leaves the transfer details to the IOP.

6.4.1 CPU-IOP Communication

There are many form of the communication between CPU and IOP. These are depending on the particular computer considered. In most cases the memory unit acts as a message center where each processor leaves information for the other. To appreciate the operation of a typical IOP, we will illustrate by a specific example the method by which the CPU and IOP communicate. This is a simplified example that omits many operating details in order to provide an overview of basic concepts.

The sequence of operations may be carried out as shown in the flowchart of Fig. 6.40. The CPU sends an instruction to test the IOP path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the IOP where to find its program.

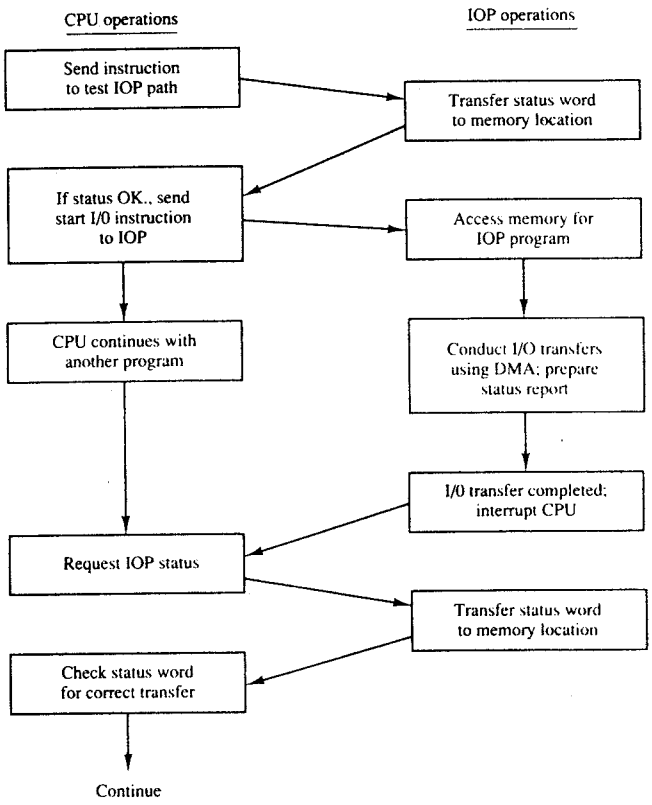


Figure 6.40: CPU-IOP communication

The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the IOP terminates the execution of its program, it sends an interrupt request to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The IOP responds by placing the contents of its status report into a specified memory location.

The IOP takes care of all data transfers between several I/O units and the memory while the CPU is processing another program. The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory.

IBM 370 I/O Channel

In the IBM 370, the I/O processor computer is known as a channel. A typical computer system configuration includes a number of channels with each channel attached to one or more I/O devices. There are three types of channels: multiplexer, selector, and block-multiplexer. The multiplexer channel can be connected to a number of slow- and medium-speed devices and is capable of operating with a number of I/O devices simultaneously. The selector channel is designed to handle one I/O operation at a time and is normally used to control one high-speed device.

The CPU communicates directly with the channels through dedicated control lines and indirectly through reserved storage areas in memory. Figure 6.41 shows the word formats associated with the channel operation. The I/O instruction format has three fields: operation code, channel address, and device address. The computer system may have a number of channels, and each is assigned an address. Similarly, each channel may be connected to several devices and each device is assigned an address. The operation code specifies one of eight I/O instructions: start I/O, start I/O fast release, test I/O, clear I/O, halt I/O, halt device, test channel, and store channel identification. The addressed channel responds to each of the I/O instructions and executes it. It also sets one of four condition codes in a processor register called PSW (processor status word). The CPU can check the condition code in the PSW to determine the result of the I/O operation. The meaning of the four condition codes is different for each I/O instruction. But, in general, they specify whether the channel or the device is busy, whether or not it is operational, whether interruptions are pending, if the I/O operation had started successfully, and whether a status word was stored in memory by the channel.

The format of the channel status word is shown in Fig. 6.41(b). It is always stored in location 64 in memory. The key field is a protection mechanism used to prevent unauthorized access by one user to information that belongs to another user or to the operating system. The address field in the status word gives the address of the last command word used by the channel. The count field gives the residual count when the transfer was terminated. The count field will show zero if the transfer was completed successfully. The status field identifies the conditions in the device and the channel and any errors that occurred during the transfer.

The difference between the start I/O and start I/O fast release instructions is that the latter requires less CPU time for its execution. When the channel receives one of these two instructions, it refers to memory location 72 force address of the first channel command word (CCW). The format of the channel command word is shown in Fig. 6.41(c). The data address field specifies first address of a memory buffer and the count field gives the number of involved in the transfer. The command field specifies an I/O operation flag bits provide additional information for the channel. The command and corresponds to an operation code that specifies one of six basic types of O operations:

1. Write. Transfer data from memory to 10 device.
2. Read. Transfer data from I/O device to memory.
3. Read backwards. Read magnetic tape with tape moving backward
4. Control. Used to initiate an operation not involving transfer of data, such as rewinding of tape or positioning a disk-access mechanism.
5. Sense. Informs the channel to transfer its channel status were memory location 64.

6. Transfer in channel. Used instead of a jump instruction. Here a word in missing address field specifies the address of the next command word to be executed by the channel.

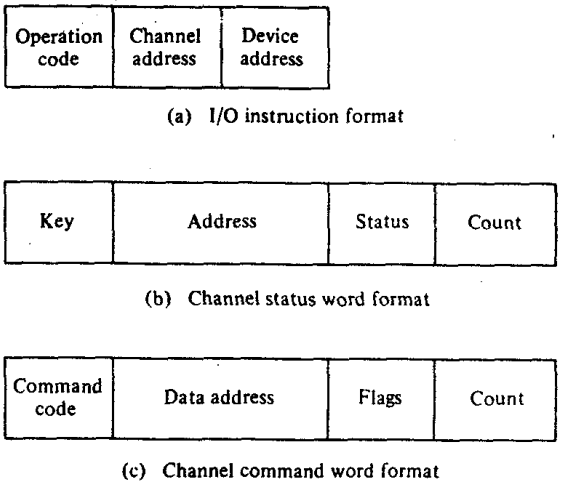


Figure 6.41: IBM 370 I/O related word formats.

An example of a channel program is shown in Table 6.7. It consists of three command words. The first causes a byte transfer into a magnetic tape from memory starting at address 4000. The next two command-words perform a similar function with a different portion of memory and byte count. The six flags in each control word specify certain interrelations between count. The first flag is set to 1 in the first command word to specify "data chaining." It results in combining the 60 bytes from the first command word with the word with the 20 bytes of its successor into one record of 80 bytes. The 80 bytes are written on tape without any separation or gaps ever though two memory sections were used. The second flag is set to 1 in the second command word to specify "command chaining." It informs the channel that the next command word will use the same I/O device, in this case, the tape, the channel informs the tape unit to start inserting a record gap on the tape and proceeds to read the next command word from memory. The 40 bytes at the third command word are then written on tape as a separate record. When all the flags are equal to zero, it signifies the end of I/O operations for the particular I/O device.

Table 6.7: BM-370 Channel Program Example

Command	Address	Flags	Count
Write tape	4000	100000	60
Write tape	6000	010000	20
Write tape	3000	000000	40

A memory map showing all pertinent information for I/O processing is illustrated in Fig 6.42. The operation begins when the CPU program encounters a start I/O instruction. The IOP then goes to memory location 72 to obtain a channel address word. This word

contains the starting address of the I/O channel program. The channel then proceeds to execute the program specified by the channel command words. The channel constructs a status word during the transfer and stores it in location 64. Upon interruption, the CPU can refer the memory location 64 for the status word.

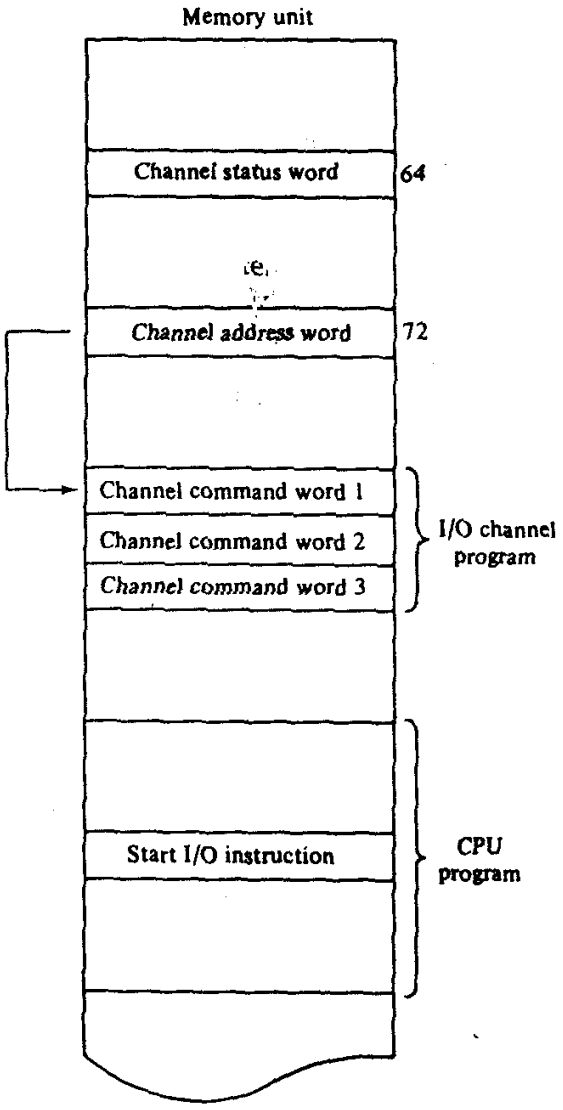


Figure 6.42: Location of Information in Memory for I/O Operation in the IBM 70

6.4.2 Serial Communication

Now we see the serial communication. A data communication processor is an I/O processor that distributes and collects data from many remote terminals connected through telephone and other communication lines. It is a specialized I/O processor

designed to communicate directly with data communication networks. A communication network may consist of any of a wide variety of devices, such as printers, interactive display devices, digital sensors, or a remote computing facility. With the use of a data communication processor, the computer can service fragments of each network demand in an interspersed manner and thus have the apparent behavior of serving many users at once. In this way the computer is able to operate efficiently in a time-sharing environment.

The main difference between an I/O processor and a data communication processor is in the way the processor communicates with the I/O devices. An I/O processor communicates with the peripherals through a common I/O bus that is comprised of many data and control lines. All peripherals share the common bus and use it to transfer information to and from the I/O processor. A data communication processor communicates with each terminal through a single pair of wires. Both data and control information are transferred in a serial fashion with the result that the transfer rate is much slower. The task of the data communication processor is to transmit and collect digital information to and from each terminal, determine if the information is data or control and respond to all requests according to predetermined established procedures. The processor, obviously, must also communicate, with the CPU and memory in the same manner as any I/O processor.

The way that remote terminals are connected to a data communication processor is via telephone lines or other public or private communication facilities. Since telephone lines were originally designed for voice communication and computers communicate in terms of digital signals, some form of conversion must be used. The converters are called data sets, acoustic couplers or modems (from "modulator-demodulator"). A modem converts digital signals into audio tones to be transmitted over telephone lines and also converts audio tones from the line to digital signals for machine use.

Synchronous transmission does not use start-stop bits to frame characters and therefore makes more efficient use of the communication link. High-speed devices use synchronous transmission to realize this efficiency. The modems used in synchronous transmission have internal clocks that are set to the frequency that bits are being transmitted in the communication line. For proper operation, it is required that the clocks in the transmitter and receiver modems remain synchronized at all times. The communication line, however, contains only the data bits from which the clock information must be extracted. Frequency synchronization is achieved by the receiving modem from the signal transitions that occur in the received data. Any frequency shift that may occur between the transmitter and receiver clocks is continuously adjusted by maintaining the receiver clock 'at the frequency of the incoming bit stream. The modem transfers the received data together with the clock to the interface unit.

Contrary to asynchronous transmission, where each character can be sent separately with its own start and stop bits, synchronous transmission must send a continuous

message in order to maintain synchronism. The message consists of a group of bits transmitted sequentially as a block of data. The entire block is transmitted with special control characters at the beginning and end of the block. The control characters at the beginning of the block supply the information needed to separate the incoming bits into individual characters.

In synchronous transmission, where an entire block of characters is transmitted, each character has a parity bit for the receiver to check. After the entire block is sent, the transmitter sends one more character as a parity over the length of the message. This character is called a longitudinal redundancy check (LRC) and is the accumulation of the exclusive-OR of all transmitted characters. The receiving station calculates the LRC as it receives characters and compares it with the transmitted LRC. The calculated and received LRC should be equal for error-free messages. If the receiver finds an error in the transmitted block, it informs the sender to retransmit the same block once again. Another method used for checking errors in transmission is the cyclic redundancy check (CRC). This is a polynomial code obtained from the message bits by passing them through a feedback shift register containing a number of exclusive-OR gates. This type of code is suitable for detecting burst errors occurring in the communication channel.
Data can be transmitted between two points in three different modes - simplex, half-duplex, and full-duplex. A simplex line carries information in one direction only. This mode is seldom used in data communication because the receiver cannot communicate with the transmitter to indicate the occurrence of errors. Examples of simplex transmission are radio and television broadcasting.
A half-duplex transmission system is one that is capable of transmitting in both directions but data can be transmitted in only one direction at a time. A pair of wires is needed for this mode.
A full-duplex transmission can send and receive data in both directions simultaneously. This can be achieved by means of a four-wire link, with a different pair of wires dedicated to each direction of transmission.

The communication lines, modems, and other equipment used in the transmission of information between two or more stations is called a data link. The orderly transfer of information in a data link is accomplished by means of a protocol. A data link control protocol is a set of rules that are followed by interconnecting computers and terminals to ensure the orderly transfer of information. The purpose of a data link protocol is to establish and terminate a connection between two stations, to identify the sender and receiver, to ensure that all messages are passed correctly without errors, and to handle all control functions involved in a sequence of data transfers. Protocols are divided into two major categories according to the message-framing, technique used. These are character-oriented protocol and bit-oriented protocol.

6.4.3 Character-Oriented Protocol

The character-oriented protocol is based on a character set. The code most commonly used is ASCII (American Standard Code for Information Interchange). It is a 7-bit code with an eighth bit used for parity. The code has 128 characters, of which 95 are graphic characters and 33 are control characters. The graphic characters include the upper- and lowercase letters, the ten numerals, and a variety of special symbols. A list of the ASCII characters can be found in Table 6.8. The control characters are used for the purpose of routing data, arranging the test in a desired format, and for the layout of the printed page. The characters that control the transmission are called communication control characters. These characters are listed in Table 6.8. Each character has a 7-bit code and is referred to by a three-letter symbol. The role of each character in the control of data transmission is stated briefly in the function column of the table.
The SYN character serves as synchronizing agent between the transmitter and receiver.

When the 7-bit ASCII code is used with an odd-parity bit in the most significant position, the assigned SYN character has the 8-bit code 00010110 which has the property that, upon circular shifting, it repeats itself only after a full 8-bit cycle. When the transmitter starts sending 8-bit characters, it sends a few characters first and then sends the actual message. The initial continuous string of bits accepted by the receiver is checked for a SYN character. In other words, with each clock pulse, the receiver checks the last eight bits received. If they do not match the bits of the SYN character, the receiver accepts the next bit, rejects the previous high-order bit, and again checks the last eight bits received for a SYN character. This is repeated after each clock pulse and bit received until a SYN character is recognized. Once a SYN character is detected, the receiver has framed a character. From here on the receiver counts every eight bits and accepts them as a single character. Usually, the receiver checks two consecutive SYN characters to remove any doubt that the first did not occur as a result of a noise signal on the line. Moreover, when the transmitter is idle and does not have any message characters to send, it sends a continuous string of SYN characters. The receiver recognizes these characters as a condition for synchronizing the line and goes into a synchronous idle state. In this state, the two units maintain bit and character synchronism even though no meaningful information is communicated.
Table 6.8: ASCII Communication Control Characters

Code	Symbol	Meaning	Function
0010110	SYN	Synchronous idle	Establishes synchronism
0000001	SOH	Start of heading	Heading of block message
0000010	STX	Start of text	Precedes block of text
0000011	ETX	End of text	Terminates block of text
0000100	EOT	End of transmission	Concludes transmission
0000110	ACK	Acknowledge	Affirmative acknowledgement
0010101	NAK	Negative acknowledge	Negative acknowledgement
0000101	ENQ	Inquiry	Inquire if terminal is on
0010111	ETB	End of transmission block	End of block of data
0010000	DLE	Data link escape	Special control character

We transmit messages through the data link with an established format consisting of a header field, a text field, and an error-checking field. A typical message format for a character-oriented protocol is shown in Fig. 6.43. The two SYN characters assure

proper synchronization at the start of the message. Following the SYN characters is the header, which starts with an SOH (start of heading) character. The header consists of address and control information. The STX character terminates the header and signifies the beginning of the text transmission. The text portion of the message is variable in length and may contain any ASCII characters except the communication control characters. The text field is terminated with the ETX character. The last field is a block check character (BCC) used for error checking. It is usually either a longitudinal redundancy check (LRC) or a cyclic redundancy check (CRC). The receiver accepts the message and calculates its own BCC. If the BCC transmitted does not agree with the BCC calculated by the receiver, the receiver responds with a negative acknowledge (NAK) character. The message is then retransmitted and checked again. Retransmission will be typically attempted several times before it is assumed that the line is faulty. When the transmitted BCC matches the one calculated by the receiver, the response is a positive acknowledgment using the ACK character.

Transmission Example

Let us illustrate by a specific example the method by which a terminal and the processor communicate. The communication with the memory unit and CPU is similar to any I/O processor.

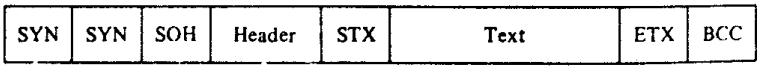


Figure 6.43: Typical message format for character-oriented protocol

A typical message that might be sent from a terminal to the processor is listed in Table 6.9. A look at this message reveals that there are a number of control characters used for message formation. Each character, including the control characters, is transmitted serially as an 8-bit binary code which consist of the 7-bit ASCII code plus an odd parity bit in the eighth most significant position. The two SYN characters are used to synchronize the receiver am transmitter. The heading starts with the SOH character and continues with two characters that specify the address of the terminal. In this particular example the address is T4, but in general it can have any set of two or more graphic characters. The STX character terminates the heading and signifies the beginning of the text transmission. The text data of concern here is "request balance of account number 1234." The individual characters for this message are not listed in the table because they will take too much space. It must be realized however, that each character in the message has an 8-bit code and that each bit is transmitted serially. The ETX control character signifies the termination of the text characters. The next character following ETX is a longitudinal redundancy check (LRC). Each bit in this character is a parity bit calculated from all the bits in the same column in the code section of the table.

The data communication processor receives this message and proceeds to analyze it. It recognizes terminal T4 and stores the text associated with the message. While receiving the characters, the processor checks the parity of each character and also computes the longitudinal parity. The computed LRC is compared with the LRC character received. If the two match, a positive acknowledgement (ACK) a sent back to the terminal. If a mismatch exists, a negative acknowledgment (NAK) is returned to the terminal, which would initiate a retransmission of the same block. If the processor finds the message without errors, it transfers the message into memory and interrupts the CPU. When the CPU acknowledges the interrupt, it analyzes the message and prepares a text message for responding to the request. The CPU sends an instruction to the data communication processor to send the message to the terminal.

Table 6.9: Typical Transmission from a Terminal to Processor

Code	Symbol	Comments
0001 0110	SYN	First sync character
0001 0110	SYN	Second sync character
0000 0001	SOH	Start of heading
0101 0100	T	Address of terminal is T4
0011 0100	4	
0000 0010	STX	Start of text transmission
0101 0010		
0100 0101	request	Text sent is a request to respond with the balance of
.	balance	account number 1234
.	of account	
.	No. 1234	
1011 0011		
0011 0100		
1000 0011	ETX	End of text transmission
0111 0000	LRC	Longitudinal parity character

A typical response from processor to terminal is listed in Table 6.9. After two SYN characters, the processor acknowledges the previous message with an ACK character. The line continues to idle with SYN character waiting for the response to come. The message received from the CPU is arranged in the proper format by the processor by inserting the required control characters before and after the text. The message has the heading SOH and the address of the terminal T4. The text message informs the terminal that the balance is $100. An LRC character is computed and sent to the terminal. If the terminal responds with a NAK character, the processor retransmits the message.

Table 6.10: Typical Transmission from Processor to Terminal

Code	Symbol	Comments
0001 0110	SYN	First sync character
0001 0110	SYN	Second sync character
1000 0110	ACK	Processor acknowledges previous message
0001 0110	SYN	Line is idling
.	.	
.	.	
.	.	
0001 0110	SYN	Line is idling
0000 0001	SOH	Start of heading
0101 0100	T	Address of terminal is T4
0011 0100	4	
0000 0010	STX	Start of text transmission
1100 0010		
1100 0001	balance	Text sent is a response from the computer giving the
.	is	balance of account
.	$100.00	
.		
.		
1011 0000		
1000 0011	ETX	End of text transmission
1101 0101	LRC	Longitudinal parity character

6.4.4 Bit-Oriented Protocol

This protocol does not use characters in its control field and is independent of any particular code. It allows the transmission of serial bit stream of any length without the implication of character boundaries. Messages are organized in a specific format called a frame. In addition to the information field, a frame contains address, control, and error-checking fields The frame boundaries are determined from a special 8-bit number called a flag Examples of bit-oriented protocols are SDLC (synchronous data link control) used by IBM, HDLC (high-level data link control) adopted by the International Standards Organization, and ADCCP (advanced data communication control procedure) adopted by the American National Standards Institute.

Any data communication link involves at least two participating stations The station that has responsibility for the data link and issues the commands to control the link is called the primary station. The other station is a secondary station. Bit-oriented protocols assume the presence of one primary station and one or more secondary stations. All communication on the data link is from the primary station to one or more secondary stations, or from a secondary station to the primary station.

The frame format for the bit-oriented protocol is shown in Fig. 6.44. A frame starts with-the 8-bit flag 01111110 followed by an address and control sequence. The information field is not restricted in format or content and can be of any length. The frame check field is a CRC (cyclic redundancy check sequence used for detecting errors in transmission. The ending flag indicate; to the receiving station that the 16 bits just received constitute the CRC bits The ending frame can be followed by another frame, another flag, or a sequence of consecutive Vs. When two frames follow each other, the intervening flag is simultaneously the ending flag of the first frame and the beginning flag of the next frame. If no information is exchanged, the transmitter sends a series of

flags to keep the line in the active state. The line is said to be in the idle state with the occurrence of 15 or more consecutive l's. Frames with certain control messages are sent without an information field. A frame must have a minimum of 32 bits between two flags to accommodate the address, control, and frame check fields. The maximum length depends on the condition of the communication channel and its ability to transmit long messages error-free.

To prevent a flag from occurring in the middle of a frame, the bit-oriented protocol uses a method called zero insertion. It requires a 0 be inserted by the transmitting station after any succession of five continuous l's. The receiver always removes a 0 that follows a succession of five l's. Thus the bit pattern 0111111 is transmitted as 01111101 and restored by the receiver to it original value by removal of the 0 following the five l's. As a consequence, no pattern of 01111110 is ever transmitted between the beginning and ending flags.

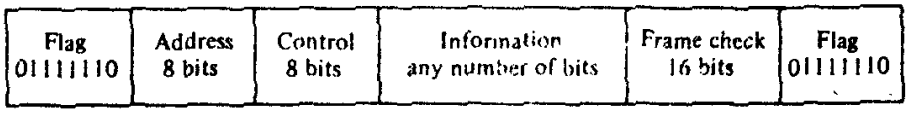


Figure 6.44: Frame Format for Bit-oriented Protocol

Following the flag is the address field, which is used by the primary station to designate the secondary station address. An address field of eight bits can specify up to 256 addresses. Some bit-oriented protocols permit the use of an extended address field. To do this, the least significant bit of an address byte is set too if another address byte follows. A 1 in the least significant bit of a byte is use to recognize the last address byte.

Following the address field is the control field. The control field comes in three different formats, as shown in Fig. 6.45. The information transfer format is used for ordinary data transmission. Each frame transmitted in this format contains send and receive counts. A station that transmits sequence frames counts and numbers each frame. This count is given by the send count N_s. A station receiving sequenced frames counts each error-free frame that receives. This count is given by the receive count N_r. The N_r count advance when a frame is checked and found to be without errors. The receiver confirms accepted numbered information frames by returning its N_r count to the transmitting station.

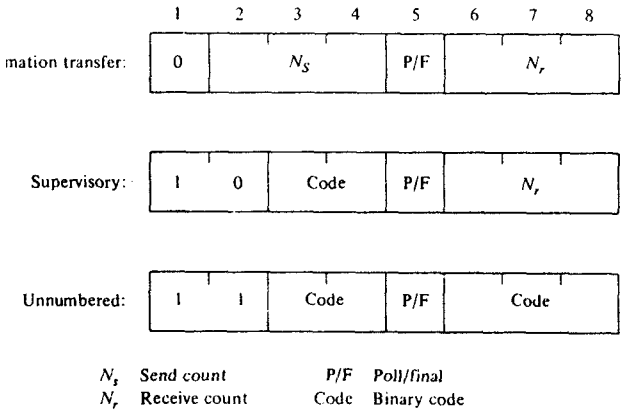


Figure 6.45: Control Field Format in Bit-oriented Protocol

The P/F bit is used by the primary station to poll a secondary station to request that it 'initiate transmission. It is used by the secondary station to indicate the final transmitted frame. Thus the P/F field is called P (poll) when the primary station is transmitting but is designated as F (final) when a secondary station is transmitting. Each frame sent to the secondary station from the primary station has a P bit set to 0. When the primary station is finished and ready for the secondary station to respond, the P bit is set to 1. The secondary station then responds with a number of frames in which the F bit is set to 0. When the secondary station sends the last frame, it sets the F bit to 1. Therefore, the P/F bit is used to determine when data transmission from a station is finished.

The supervisory format of the control field is recognized from the first two bits being 1 and 0. The next two bits indicate the type of command. The frames of the supervisory format do not carry an information field. They are used to assist in the transfer of information in that they confirm the acceptance of preceding frames carrying information, convey ready or busy conditions, and report frame numbering errors.

Paper Name: Computer Organization and Architecture

UNIT 7

MEMORY ORGANIZATION

Introduction

Memory in a computer system can be divided into two main classes: *main store* and *secondary store*. Main store is the high speed memory used to hold the programs and data currently in use. Secondary store is the memory used for long term storage of data, e.g. a disk drive. This section discusses main store - the next deals with secondary store.

Computer memory is made of a number of *cells* (one per bit), each of which can exist in two distinct states corresponding to the value of some physical property. Cells are often grouped together to form words. Each cell has an *address* which allows it to be uniquely specified.
Different physical properties can be used to store the information. The following are commonly used methods.

- Electrical with feedback (e.g. flip-flops)
- Electrical with stored charge (based on capacitance)
- Magnetic (e.g. disk drives)
- Structural (e.g. compact disks).

Computer memory can also be classed according to the method of access to the cells. *Random access memory* is arranged so that the time taken to read or write any cell is the same, regardless of its location. Main memory is random-access. Note however, that

the term random access memory (RAM) is often used to refer only to the read-writable main memory used for short term data storage. *Serial access memory* is arranged so that the time taken to access a particular cell is dependent is dependent on the physical location of the cell, and usually depends on the position of the last cell accessed. Tape drives are serial access devices.

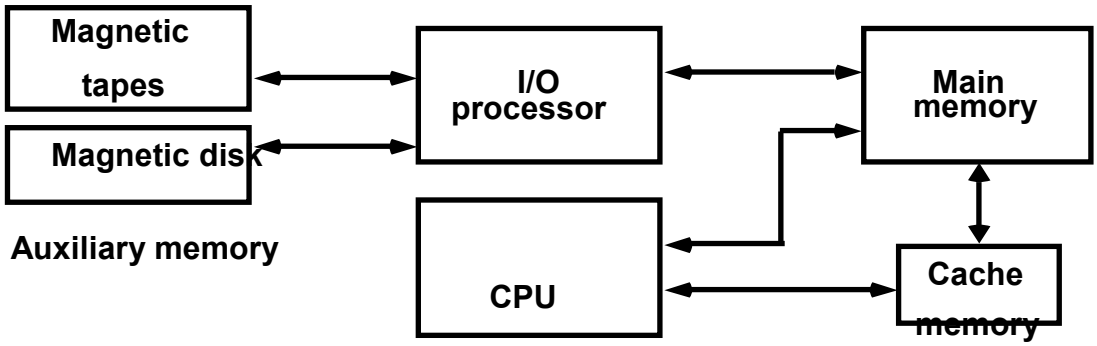
We can also classify memory according to whether or not we can modify the cell contents. *Read only memory* (ROM) is memory whose contents cannot be modified. This may either be semiconductor memory, or a read-only device such as an ordinary optical disk (CD-ROM). The core of a computer's operating system is often stored in semiconductor ROM.

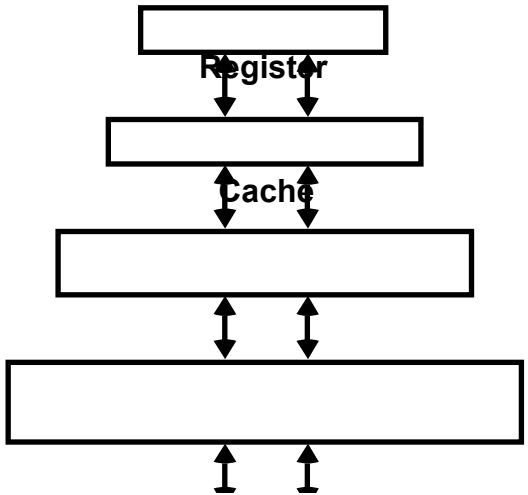
7.1 Memory Hierarchy

Memory Hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system

Computer systems always combine several different types of the memory devices discussed above. This is because none alone can provide all the required characteristics. Ideally computer memory should be the following.

a) Very fast
b) small
c) Consume low power
d) Robust and non-volatile (remember its contents even when switched off)
e) Cheap





Paper Name: Computer Organization and Architecture

Main Memory

Magnetic Disk

Unfortunately these aims conflict, and different types of memory device offer different benefits and drawbacks.

Internal CPU memory
This is very fast. However, it is bulky, expensive, consumes a lot of power, and the contents are lost when power is removed.

Main store
Relatively fast, but still bulky, expensive and volatile.

Magnetic disk
These can store large quantites of data cheaply and in a small space. Furthermore, these can be used for permament/semi-permanent storage, as the data is not lost when power is removed. However, the access time is much slower than main memory.

Magnetic tape and optical storage
These are both very cheap and can store huge quantities of data in a small space. Typically they use removable media, and so are ideal for permanent storage of data. However, access times are extremely long.

By combining different types of memory in a single system, the designer can get the best of all worlds and build a relatively low cost system with a high capacity and a speed almost that of a huge main memory.

Semiconductor (main) Memory
All of the memory used as main store in a modern computer is implemented as semiconductors fabricated on wafers of silicon. Semiconductor memory is fast and easy to use. To fulfill the needs of modern computer systems it is becoming increasingly dense (more bits per chip) and cheap.
A semiconductor memory chip consists of a large number of cells organised into an array, and the logic necessary to access any array in the cell easily. Semi-conductor memory may be classed according to the mechanism used by each cell to store data.

The simplest type of memory is called *static memory*. In static memory each cell uses a flip-flop made from four or six transistors. The data in each cell is remembered until the

power is switched off. Static memory is easy to use and reliable, but is relatively bulky, slow and expensive.
Most computer systems therefore use *dynamic memory* as their main store. Dynamic memory uses just a single transistor per cell, and is therefore denser, faster and cheaper. Unfortunately each cell gradually forgets the data stored in it, and so extra circuitry must be used to continually refresh the cells.

Memory Organisation
This section looks at how the various components of a computer's main memory are arranged.
An *n*-bit address bus allows us to uniquely refer to up to 2^n different memory locations. Often these memory locations are imagined laid out in a column, known as the *address space*. Various regions of the address space are then grouped into blocks to form a *memory map*. The blocks may refer to hardware (i.e. physical RAM and ROM devices), or to logical entities (i.e. programs, data, etc.).

Address Decoding
The need for memory address decoding arises from the fact that the main memory of a computer system is not contructed from a single component which uniquely addresses each possible memory location.
Imagine a situation where two 1M memory chips are connected to a 32-bit address bus to make 2M of memory available. Each memory chip will need twenty address lines to uniquely identify each location in it. If the address lines of each memory chip were simply connected to the first twenty CPU address lines, then both memory chips would be accessed simultaneously whenever the CPU referred to any address. We must find a strategy that allows us to individually address each memory chip. As each memory chip will have an enable input, this can rely on selectively enabling the one which we want to access.

a) Partial Address Decoding
This is the simplest and least expensive form of address decoding. In the above example, we could connect the chip select input of one memory chip to the last CPU address line, and the chip select input of the other to the same address line but via an inverter. In this way the two chips would never be accessed simultaneously.

However, this is a very inefficient scheme. Eleven of the address lines are not used, and one of the two memory chips is always selected. The usable address space of the computer has been reduced from 4G to 2K. Partial address decoding is used in small dedicated systems where low cost is the most important factor. The penalty paid is that not all the address space can be used, and future expansion will be difficult.

Full Address Decoding
A computer system is said to have full address decoding when each addressable location within a memory component corresponds to a single address on the CPU's

address bus. That is, every address line is used to specify each physical memory location, through a combination of specifying a device and a location within it.

Full address decoding makes the most efficient use of the available address space, but is often impracticable to use because of the excessive hardware needed to implement it. This is particularly true where devices with a small number of addressable locations (for example memory-mapped I/O devices) are used.

b) Block Address Decoding

Block address decoding is a compromise between partial address decoding and full address decoding. The memory space is divided into a number of blocks. For example, in a system with a 32-bit address bus, the memory space could be divided into 4096 blocks of 1M. This could be implemented using simple decoding devices.

Many real systems employ a combination of the above decoding techniques. For example, several small devices may reside in the same block by using partial address decoding within that block.

Cache Memory

Although semiconductor memory is fast, typically it takes longer for the CPU to access main memory than to execute simple instructions. We can get around this problem by taking advantage of *locality of reference*. Typically when executing a program, memory accesses over a short interval tend to be to similar memory locations. Most modern computer systems therefore use a small amount of very fast memory between the CPU and main memory, known as *cache* memory. Whenever the CPU needs to access main memory, the cache is first checked to see if the data is stored there. If it is, then main memory need not be accessed. If it is not, then it is fetched from main memory and placed in the cache, together with the data nearby, in the hope that this will also be needed soon. In this way, accesses to main memory can be greatly reduced (typically reduced to less than 10%).

We have already seen that CPU memory is faster than main memory, but is very expensive, and takes up a lot of room on a chip which is already likely to be fairly densely packed. Many systems therefore use a two-stage cache design, using a small amount of on-chip cache memory, and a larger external cache.

7.2 Main Memory

- A *memory unit* is a collection of storage cells together with associated circuits to transfer information in and out of storage
- The memory stores binary data in groups of bits called words
- A word can represent an instruction code or alphanumeric characters
- Each word in memory is assigned an address from 0 to $2^k - 1$, where k is the number of address lines
- A *decoder* inside the memory accepts an address opens the paths needed to select the bits of the specified word

- The memory capacity is stated as the total number of bytes that can be stored
- Refer to the number of bytes using one of the following
  - K (kilo) = 2^{10}
  - M (mega) = 2^{20}
  - G (giga) = 2^{30}
- 64K = 2^{10}, 2M = 2^{21}, and 4G = 2^{32}
- In *random-access memory* (RAM) the memory cells can be accessed for information from any desired random location
- The process of locating a word in memory is the same and requires an equal amount of time no matter where the cells are located physically in memory
- Communication between memory and its environment is achieved via data input and output lines, address selections lines, and control lines
- The *n* data input lines provide the information to be stored in memory
- The *n* data output lines supply the information coming out of memory
- The *k* address lines provide a binary number of *k* bits that specify a specific word or location
- The two control lines specify the direction of transfer – either read or write

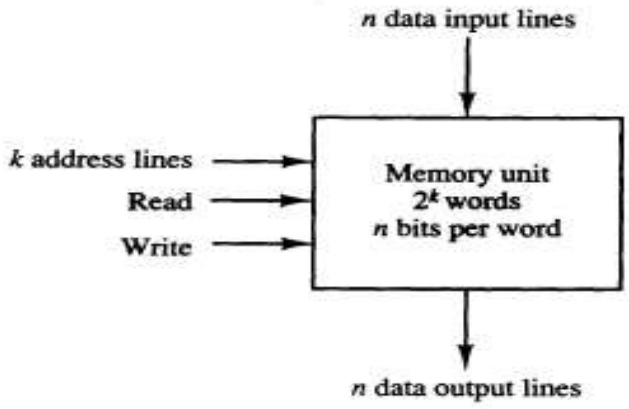


Figure 2-12 Block diagram of random access memory (RAM).

- Steps to *write* to memory:
  - Apply the binary address of the desired word into the address lines
  - Apply the data bits that are to be stored in memory on the data lines
  - Activate the write input
- Steps to *read* from memory:
  - Apply the binary address of the desired word into the address lines
  - Activate the read input

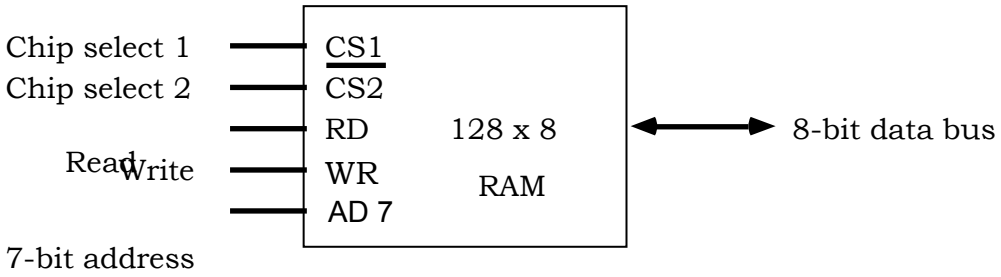
- A *read-only memory* (ROM) is a memory unit that performs the read operation only – there is no write capability

- The binary information stored in a ROM is permanent during the hardware production
- RAM is a general-purpose device whose contents can be altered
- The information in ROM forms the required interconnection pattern
- ROMs come with special internal electronic fuses that can be programmed for a specific configuration
- An *m* x *n* ROM is an array of binary cells organized into *m* words of *n* bits each
- A ROM has *k* address lines to select one of *m* words in memory and *n* output lines, one for each bit of the word
- May have one or more enable inputs for expansion
- The outputs are a function of only the present input (the address), so it is a combinational circuit constructed of decoders and OR gates

7.2.1 RAM and ROM Chips

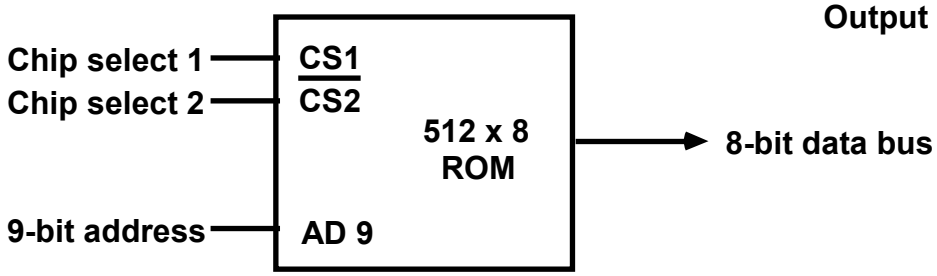
RAM and ROM Chips

Typical RAM chip



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

Typical ROM chip



- When used as a memory unit, it stores fixed programs that are not to be altered and for tables of constants that will not change
- When used in the design of control units for digital computers, it stores coded information that represents the sequence of internal control variables to enable the various operations
- A control unit that utilizes a ROM is called a *microprogrammed control unit*
- The required paths may be programmed in three different ways
- *Mask programming* is done by the semiconductor company based upon a truth table provided by the manufacturer
- *Programmable read-only memory* (PROM) is more economical. PROM units contain all fuses intact and are blown by users
- *Erasable PROM* (EPROM) can be altered using a special ultraviolet light *Electrical erasable PROM* (EEPROM) can be erased with electrical signals

7.2.2 Memory Address Map

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

To demonstrate with particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips to be used are specified in fig .The memory address map for rhis configuration is shown in table 12-1. The Compnent coloum specifies whether a RAM or ROM chip is used. The hexadecimal sddress coloumn assign a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other are 16 lines in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address lines.The ROM chip has 512 bytes and needs 9 address lines. The x's are alwayes assigned to the low arder bus lines: lines 1 through 7 for Ram andlines 1 through 9 for the ROM.

Component	Hexadecimal address	address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000-007F	0	0	0	X	X	X	X	X	X	X
RAM 2	0080-00FF	0	0	1	X	X	X	X	X	X	X
RAM 3	0100-017F	0	1	0	X	X	X	X	X	X	X

RAM 4	0180-01FF	0	1	1	X	X	X	X	X	X	X
ROM	0200-03FF	1	X	X	X	X	X	X	X	X	X

7.3 Auxiliary Memory

The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks.

7.3.1 Magnetic Disks

Of the various types of Auxiliary Storage, the types used most often involve some type of **magnetic disk**. These come in various sizes and materials, as we shall see. This method uses magnetism to store the data on a magnetic surface.

Advantages:

high storage capacity
reliable
gives direct access to data

A drive spins the disk very quickly underneath a read/write head, which does what its name says. It reads data from a disk and writes data to a disk.

Types of Magnetic Disks

Diskette / Floppy Disk

Sizes:

5¼"

3½"

Both sizes are made of mylar with an oxide coating. The oxide provides the magnetic quality for the disk. The "floppy" part is what is inside the diskette covers - a very floppy piece of plastic (i.e. the mylar)

Other Removable Media:

Several other kinds of removable magnetic media are in use, such as the popular Zip disk. All of these have a much higher capacity than floppy disks. Some kinds of new computers come without a floppy disk drive at all.

Each type of media requires its own drive. The drives and disks are much more expensive than floppy drives and disks, but then, you are getting much larger capacities.

Hard Disks:

These consist of 1 or more metal platters which are sealed inside a case. The metal is one which is magnetic. The hard disk is usually installed inside the computer's case, though there are removable and cartridgetypes,also.

Technically the hard drive is what controls the motion of the hard disks which contain the data. But most people use "hard disk" and "hard drive" interchangeably. They don't make that mistake for floppy disks and floppy drives. It is clearer with floppies that the drive and the disk are separate things.

Physical Characteristics of Disks:

1. The storage capacity of a single disk ranges from 10MB to 10GB. A typical commercial database may require hundreds of disks.
2. Figure 10.2 shows a moving-head disk mechanism.
    - Each disk *platter* has a flat circular shape. Its two surfaces are covered with a magnetic material and information is recorded on the surfaces. The platter of *hard disks* are made from rigid metal or glass, while *floppy disks* are made from flexible material.
    - The disk surface is logically divided into *tracks*, which are subdivided into *sectors*. A sector (varying from 32 bytes to 4096 bytes, usually 512 bytes) is the smallest unit of information that can be read from or written to disk. There are 4-32 sectors per track and 20-1500 tracks per disk surface.
    - The arm can be positioned over any one of the tracks.
    - The platter is spun at high speed.
    - To read information, the arm is positioned over the correct track.
    - When the data to be accessed passes under the head, the read or write operation is performed.
3. A disk typically contains multiple platters (see Figure 10.2). The read-write heads of all the tracks are mounted on a single assembly called a *disk arm*, and move together.
    - Multiple disk arms are moved as a unit by the actuator.
    - Each arm has two heads, to read disks above and below it.
    - The set of tracks over which the heads are located forms a cylinder.
    - This cylinder holds that data that is accessible within the disk latency time.

- It is clearly sensible to store related data in the same or adjacent cylinders.

4. Disk platters range from 1.8" to 14" in diameter, and 5"1/4 and 3"1/2 disks dominate due to the lower cost and faster seek time than do larger disks, yet they provide high storage capacity.
5. A *disk controller* interfaces between the computer system and the actual hardware of the disk drive. It accepts commands to r/w a sector, and initiate actions. Disk controllers also attach *checksums* to each sector to check read error.
6. *Remapping of bad sectors*: If a controller detects that a sector is damaged when the disk is initially formatted, or when an attempt is made to write the sector, it can logically map the sector to a different physical location.
7. SCSI (*Small Computer System Interconnect*) is commonly used to connect disks to PCs and workstations. Mainframe and server systems usually have a faster and more expensive bus to connect to the disks.
8. Head crash: why cause the entire disk failing (?).
9. A *fixed dead disk* has a separate head for each track -- very many heads, very expensive. *Multiple disk arms*: allow more than one track to be accessed at a time. Both were used in high performance mainframe systems but are relatively rare today.

Performance Measures of Disks

The main measures of the qualities of a disk are *capacity*, *access time*, *data transfer rate*, and *reliability*,

1. *access time*: the time from when a read or write request is issued to when data transfer begins. To access data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates. The time for repositioning the arm is called seek time, and it increases with the distance the arm must move. Typical seek time range from 2 to 30 milliseconds.

   Average seek time is the average of the seek time, measured over a sequence of (uniformly distributed) random requests, and it is about one third of the worst-case seek time.

   Once the seek has occurred, the time spent waiting for the sector to be accesses to appear under the head is called rotational latency time. Average rotational latency time is about half of the time for a full rotation of the disk. (Typical rotational speeds of disks ranges from 60 to 120 rotations per second).

   The access time is then the sum of the seek time and the latency and ranges from 10 to 40 milli-sec.
2. *data transfer rate*, the rate at which data can be retrieved from or stored to the disk. Current disk systems support transfer rate from 1 to 5 megabytes per second.

3. *reliability*, measured by the *mean time to failure*. The typical mean time to failure of disks today ranges from 30,000 to 800,000 hours (about 3.4 to 91 years).

Optimization of Disk-Block Access

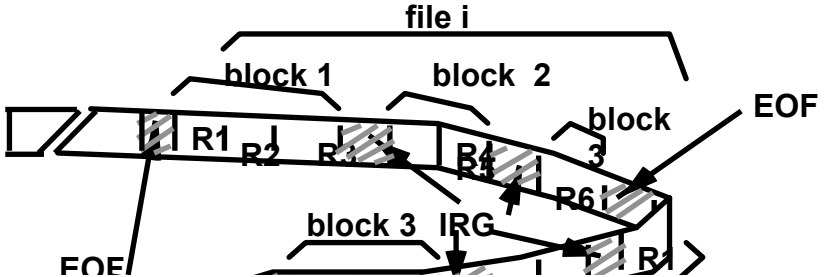
1. Data is transferred between disk and main memory in units called blocks.
2. A block is a contiguous sequence of bytes from a single track of one platter.
3. Block sizes range from 512 bytes to several thousand.
4. The lower levels of file system manager covert block addresses into the hardware-level cylinder, surface, and sector number.
5. Access to data on disk is several orders of magnitude slower than is access to data in main memory. Optimization techniques besides buffering of blocks in main memory.
    - Scheduling: If several blocks from a cylinder need to be transferred, we may save time by requesting them in the order in which they pass under the heads. A commonly used disk-arm scheduling algorithm is the *elevator algorithm*.
    - File organization. Organize blocks on disk in a way that corresponds closely to the manner that we expect data to be accessed. For example, store related information on the same track, or physically close tracks, or adjacent cylinders in order to minimize seek time. IBM mainframe OS's provide programmers fine control on placement of files but increase programmer's burden.

      UNIX or PC OSs hide disk organizations from users. Over time, a sequential file may become fragmented. To reduce fragmentation, the system can make a back-up copy of the data on disk and restore the entire disk. The restore operation writes back the blocks of each file continuously (or nearly so). Some systems, such as MS-DOS, have utilities that scan the disk and then move blocks to decrease the fragmentation.

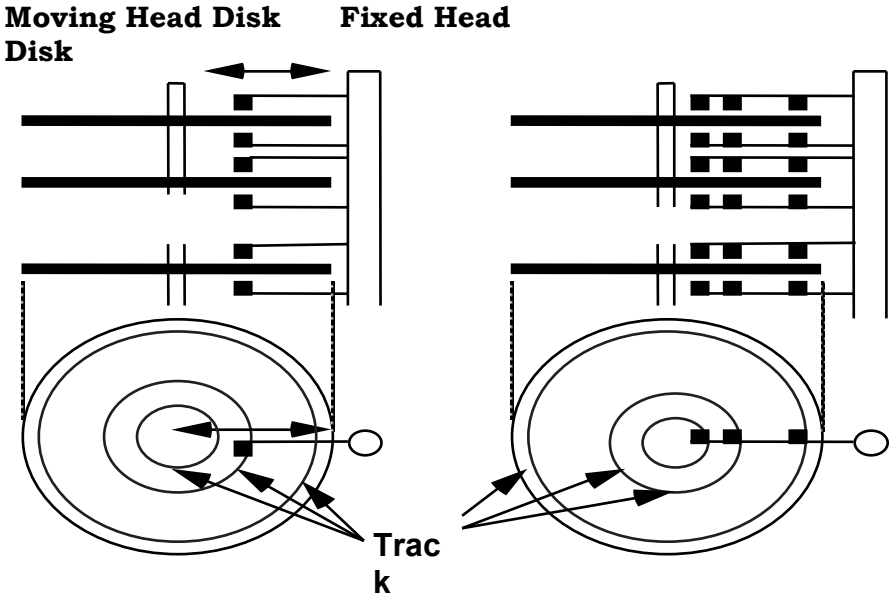
      Nonvolatile write buffers. Use *nonvolatile RAM* (such as *battery-back-up RAM*) to speed up disk writes drastically (first write to nonvolatile RAM buffer and inform OS that writes completed).
    - Log disk. Another approach to reducing write latency is to use a *log disk*, a disk devoted to writing a sequential log. All access to the log disk is sequential, essentially eliminating seek time, and several consecutive blocks can be written at once, making writes to log disk several times faster than random writes.

7.3.2 Magnetic Tape

Information Organization on Magnetic Tapes



Organization of Disk Hardware



A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic-tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters. Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound. They cannot be started or stopped fast enough between individual characters.

7.4 Cache Memory

RAM memory can be accessed incredibly fast. How fast? Well, standard SDRAM has a clock pulse of 133MHz. That means a word of memory can be accessed

133000000 times a second. Or in other words, it takes about 0.0000000075 seconds to do a memory read. Sounds fast. But actually the speed may be more like 60 ns (0.00000006 seconds) because of latency.

Problem is that CPU's now run at about 1GHz, or on a clock pulse about 1000000000 times a second or every 0.000000001 second (1 ns). That means that if the CPU is working with memory most of the time it will be sitting around doing nothing (98% of the time). For that reason we set up a smaller amount of SRAM called **cache memory** that the CPU works with directly instead of the slower (and larger) SDRAM.

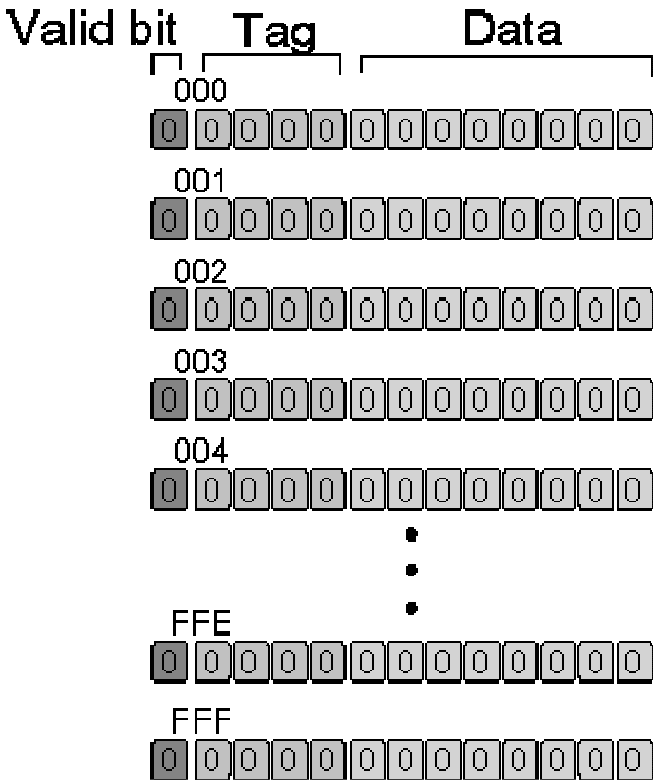
SRAM (built using combinatorial logic gates instead of capacitors) has an access speed of about 10 ns.

So what is the essential point here? Make sure that the memory that the CPU needs to access is already in the cache when it needs it. We will look at methods of using cache memory and its interaction with the rest of RAM in this chapter.

7.4.1 Direct Mapping

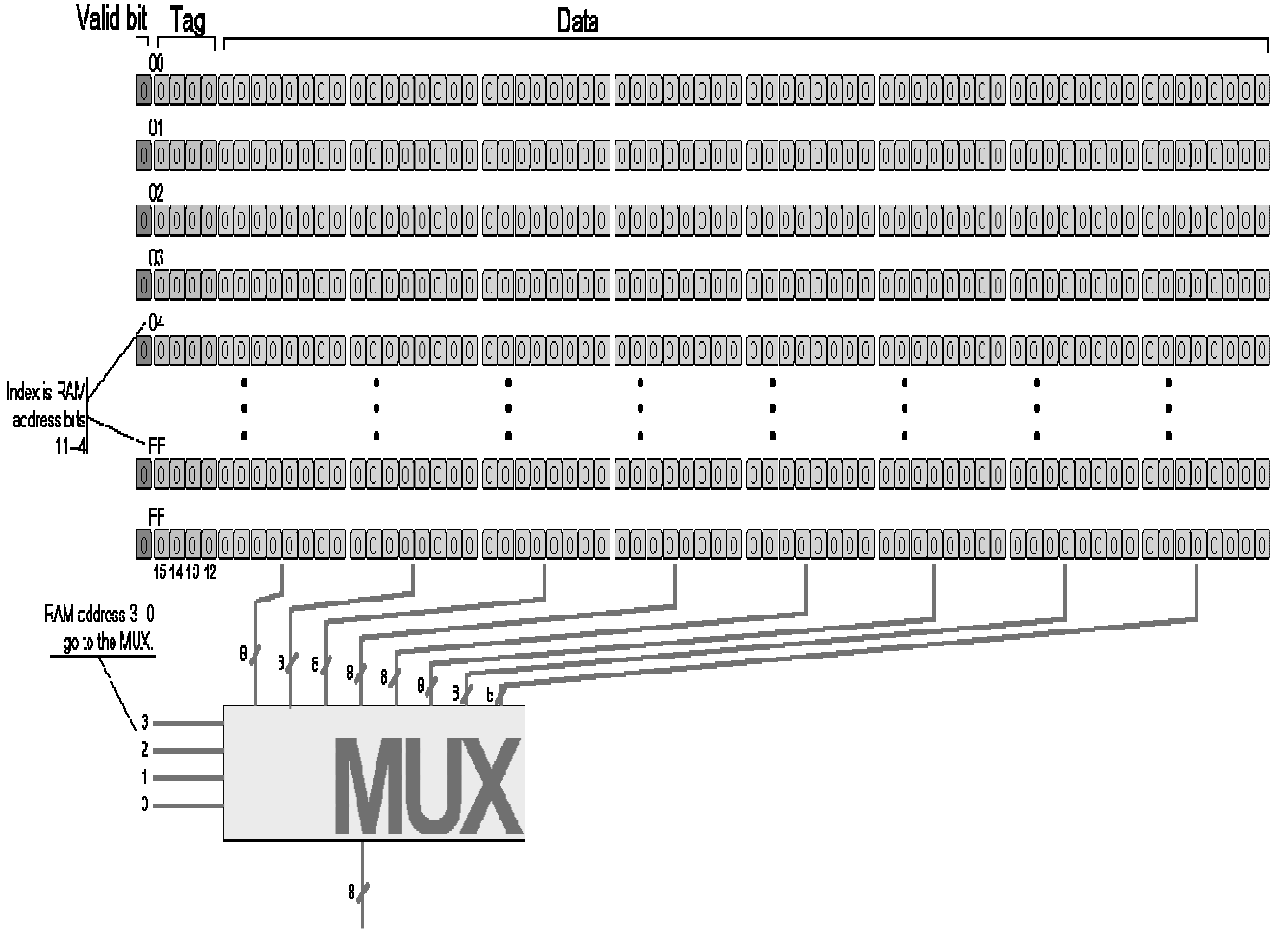
The significant overhead of associative memory and all those addresses can be avoided by not storing addresses. Instead, we will store data in the cache just like we do in RAM. But then we need some way to map lots of RAM addresses to much fewer addresses in the cache. Some kind of arithmatic. This is called **direct mapping**. In this scheme all addresses with the same lower order bits will map to the same address in the cache. If we have only 1K of cache, we will use only the last 10 lower order bits of the RAM address for the cache address, for 4K we will use the last 12 bits. Let's say we do that, we have 4K cache for a 64K RAM of the relatively simple CPU. In that case, how many RAM addresses will map to the same address in the cache? How will we resolve the collisions?

In direct mapping, we resolve the collisions by storing the rest of the RAM address (the last 4 higher order bits in this case) with the word of data, Something Like this:



The left over bits stored with the data are called the **tag**. To access a word in memory the CPU first checks to see if it is in the cache by going to the address of the last 10 bits of RAM (the **index**) and then checking the tag to make sure it matches the higher order bits, and also checking the valid bit to make sure it is valid. What if the tag does not match?

Of course direct-mapped cache can be built with lines as well, something like

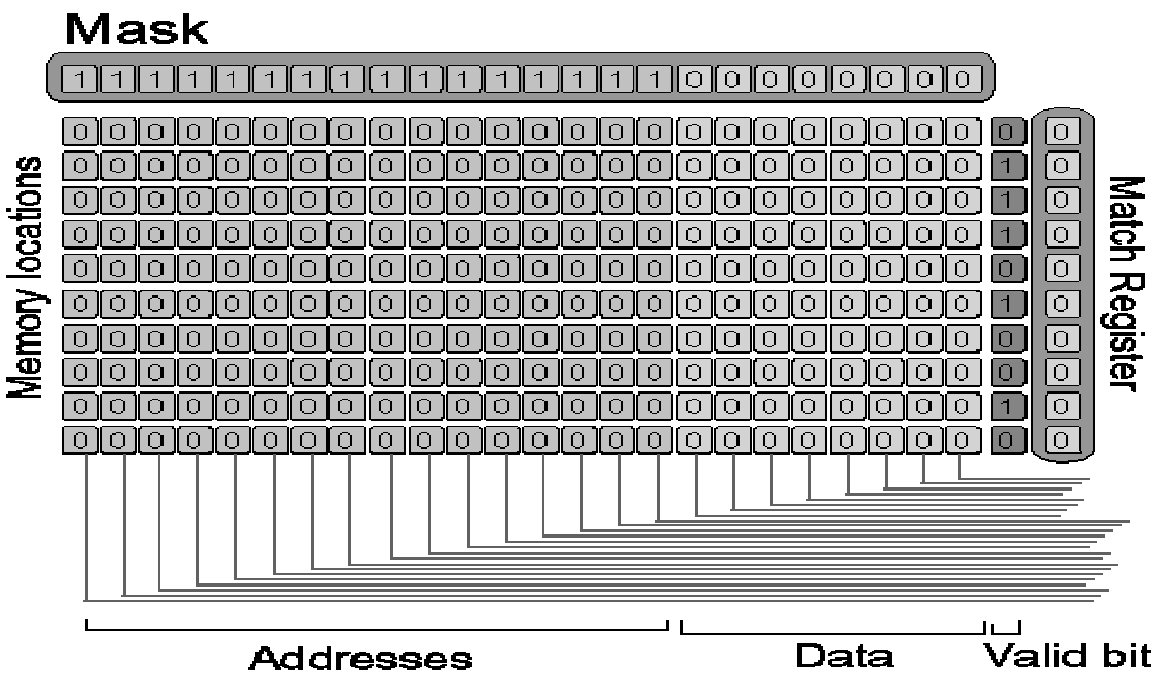


7.4.2 Associative Memory Cache

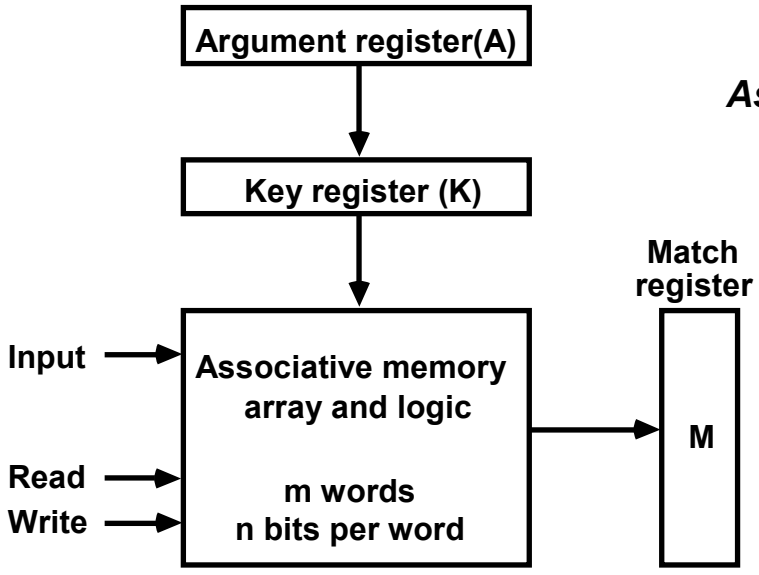
You have to remember that cache memory is by definition only a subset of the total RAM. Only a select few words of memory can fit into the cache. So which ones get to fit in the cache? That depends on what is needed.

Well, how will we address this memory. Remember, that on the relatively simple CPU, RAM memory is addressed with a 16 bit address, something like this:
where each byte is accessed with a specific 16 bit address like we discussed in the lecture on memory location. But that will not work for the cache because different words of memory will be smushed together in no particular order.

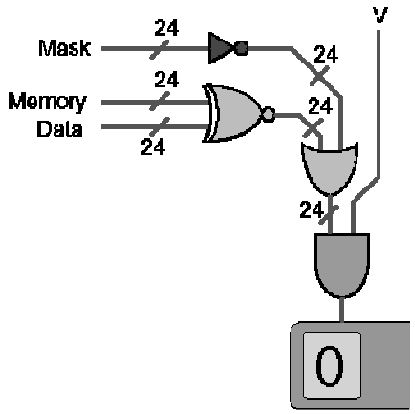
So, the first method of storage we will try for cache is to include the RAM memory address for each byte of RAM that is now being stored in the cache, something like this:



This is called **associative cache**. Notice that there is a great deal of overhead here since each byte needs an additional 2 bytes of storage for its address, along with an extra bit of information indicating that this byte is a valid piece of RAM. With associative memory, access to a byte is determined by the data input to the circuit. It is tested against the **mask** register to see which bits are the address. Those need to match exactly with the data. If the byte is also valid then a bit is set for that location in the **match register** according to the circuit shownbelow:



Associative Memory

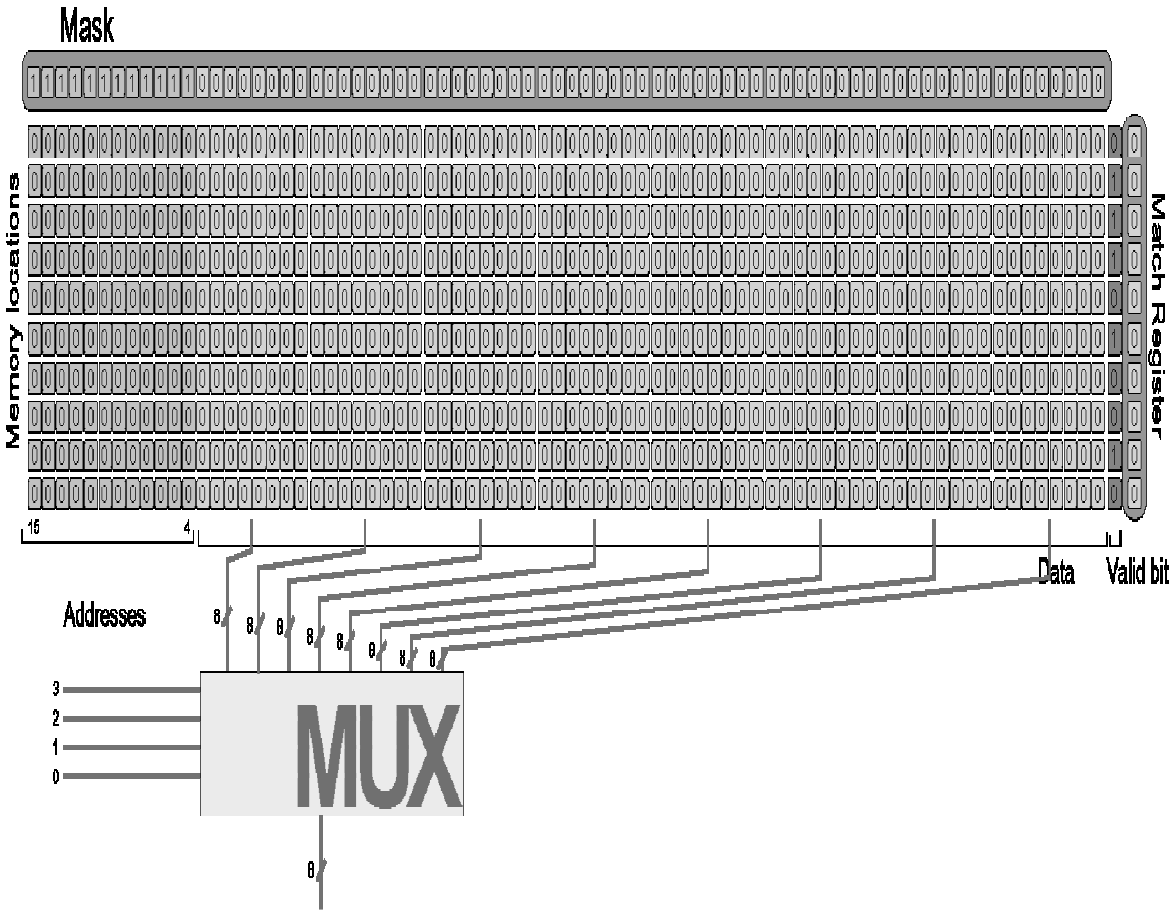


This circuit is repeated for each word in memory. In this way more than one word could be marked as matching in the match register. But in pactice, with cache memory, each word from RAM would only have one copy in the cache so only one bit in the match register would be set at a time.
Compare each word in CAM in parallel with the
content of A(Argument Register)
- If CAM Word[i] = A, M(i) = 1
- Read sequentially accessing CAM for CAM Word(i) for M(i) = 1
- K(Key Register) provides a mask for choosing a
particular field or key in the argument in A
(only those bits in the argument that have 1's in
their corresponding position of K are compared)

By the way, for a 4K associative memory cache, what is the size of the match register?

Associative memory with lines

As you can see the overhead for associative memory is significant. One way to lessen the overhead to to group together bytes of memory into chunks with only one address for the whole chunk. These "chunks" of memory are called **lines**. The associative memory with lines can be built like this:

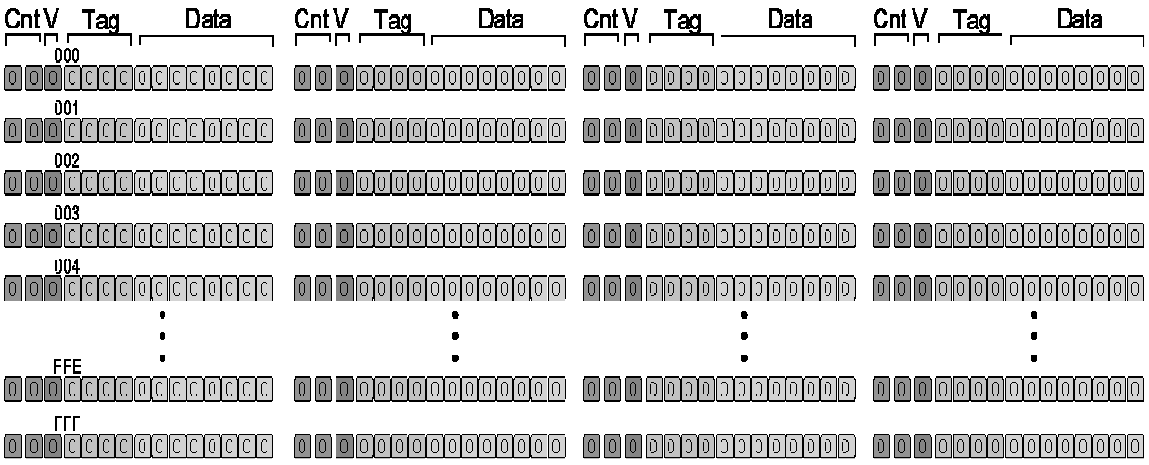


Obviously, this saves considerably on overhead of storage. What is another advantage of using lines of data?

7.4.3 Set-Associative Cache

What is the inherent drawback of direct-mapping?

To overcome the problems with direct-mapping designers thought of a third scheme called **Set-associative mapping**. This version of the cache memory is really just another type of direct-mapping. But instead of a single word of memory being able to occupy a given index location there is room for two or more words. In this example, four **ways** are stored for each index.



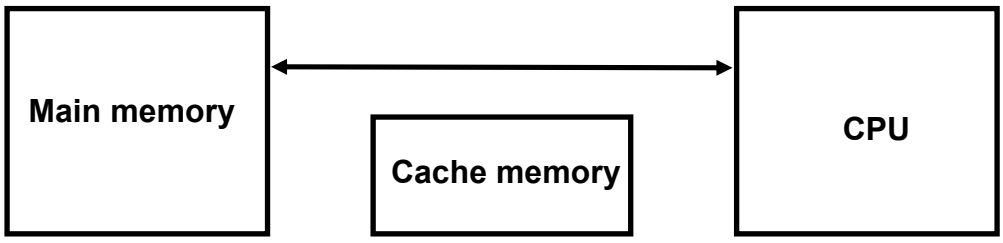
The tag still has to be matched in order to find the word we want from RAM but now we get to look at four canidates to see if one matches. So you have a 4 times greater chance of finding what you are looking for.

Locality of Reference

- The references to memory at any given time
  interval tend to be confined within a localized areas
- This area contains a set of information and
  the membership changes gradually as time goes by
- *Temporal Locality* The information which will be used in near future
  is likely to be in use already(e.g. Reuse of information in loops)
- *Spatial Locality:*If a word is accessed, adjacent(near) words are likely
  accessed soon(e.g. Related data items (arrays) are usually stored together; instructions are executed sequentially)

Cache

- The property of Locality of Reference makes the
  Cache memory systems work
- Cache is a fast small capacity memory that should hold those information
  which are most likely to be accessed



7.4.4 Virtual Memory

Storage allocation has always been an important consideration in computer programming due to the high cost of main memory and the relative abundance and lower cost of secondary storage. Program code and data required for execution of a process must reside in main memory to be executed, but main memory may not be large enough to accomodate the needs of an entire process. Early computer programmers divided programs into sections that were transferred into main memory for a period of processing time. As the program proceeded, new sections moved into main memory and replaced sections that were not needed at that time. In this early era of computing, the programmer was responsible for devising this overlay system.

As higher level languages became popular for writing more complex programs and the programmer became less familiar with the machine, the efficiency of complex programs suffered from poor overlay systems. The problem of storage allocation became more complex.
Two theories for solving the problem of inefficient memory management emerged -- static and dynamic allocation.

Static allocation assumes that the availability of memory resources and the memory reference string of a program can be predicted.
Dynamic allocation relies on memory usage increasing and decreasing with actual program needs, not on predicting memory needs.

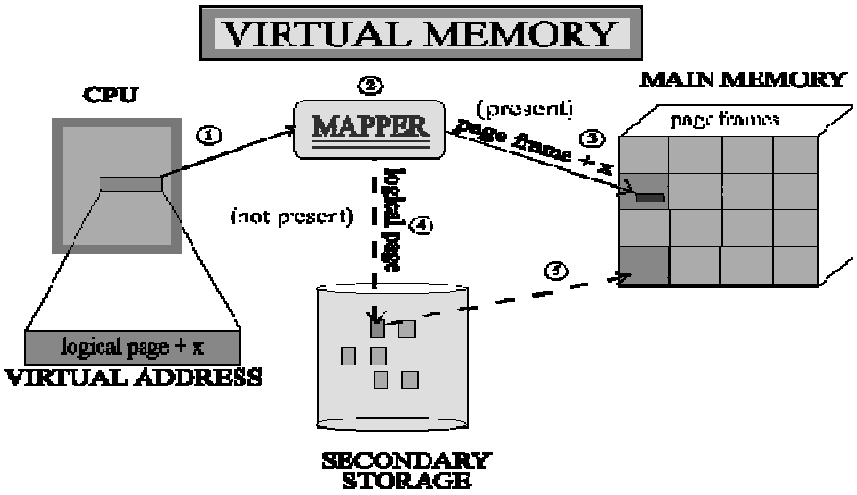
Program objectives and machine advancements in the '60s made the predictions required for static allocation difficult, if not impossible. Therefore, the dynamic allocation solution was generally accepted, but opinions about implementation were still divided. One group believed the programmer should continue to be responsible for storage allocation, which would be accomplished by system calls to allocate or de-allocate memory. The second group supported **automatic storage allocation** performed by the operating system, because of increasing complexity of storage allocation and emerging importance of multiprogramming. In 1961, two groups proposed a one-level memory store. One proposal called for a very large main memory to alleviate any need for storage allocation. This solution was not possible due to very high cost. The second proposal is known as virtual memory.

7.4.5 Associative memory Page Table Space

Virtual memory is a technique that allows processes that may not be entirely in the memory to execute by means of automatic storage allocation upon request. The term virtual memory refers to the abstraction of separating LOGICAL memory--memory as seen by the process--from PHYSICAL memory--memory as seen by the processor. Because of this separation, the programmer needs to be aware of only the logical

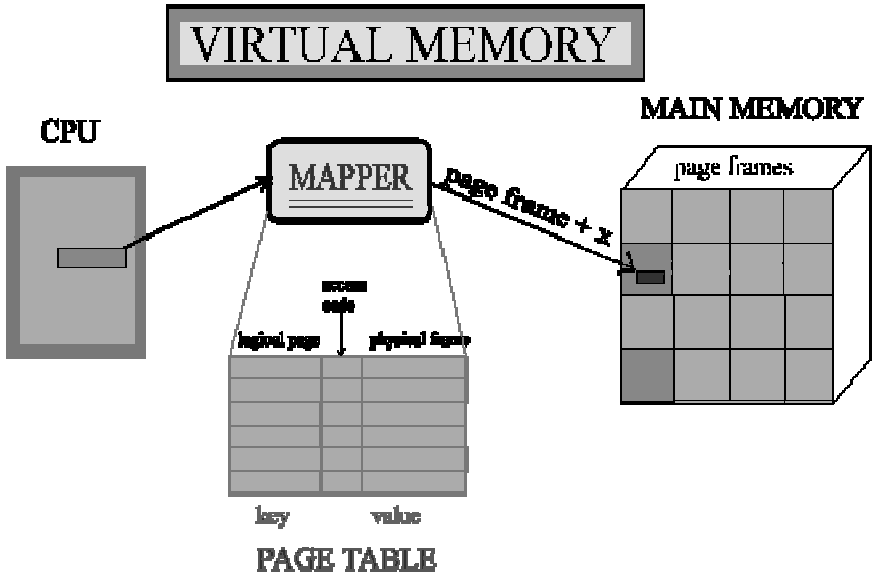
memory space while the operating system maintains two or more levels of physical memory space.

A random-access memory page table is inefficient with respect to storage utilization. The virtual memory abstraction is implemented by using secondary storage to augment the processor's main memory. Data is transferred from secondary to main storage as and when necessary and the data replaced is written back to the secondary storage according to a predetermined replacement algorithm. If the data swapped is designated a fixed size, this swapping is called **paging**; if variable sizes are permitted and the data is split along logical lines such as subroutines or matrices, it is called **segmentation**. Some operating systems combine segmentation and paging.



The diagram illustrates that a program generated address (1) or "logical address" consisting of a logical page number plus the location within that page (x) must be interpreted or "mapped" onto an actual (physical) main memory address by the operating system using an address translation function or mapper (2). If the page is present in the main memory, the mapper substitutes the physical page frame number for the logical number (3). If the mapper detects that the page requested is not present in main memory, a fault occurs and the page must be read into a frame in main memory from secondary storage (4,5)

7.4.5 Associative Memory and the Page Table



The implementation of the page table is vital to the efficiency of the virtual memory technique, for each memory reference must also include a reference to the page table. The fastest solution is a set of dedicated registers to hold the page table but this method is impractical for large page tables because of the expense. But keeping the page table in main memory could cause intolerable delays because even only one memory access for the page table involves a slowdown of 100 percent and large page tables can require more than one memory access. The solution is to augment the page table with special high-speed memory made up of associative registers or **translation lookaside buffers (TLBs)** which are called **ASSOCIATIVE MEMORY.**

Demonstation of the operation of Virtual Memory.

Each of these associative memory registers contains a key and a value. The keys to associative memory registers can all be compared simultaneously. If a match is found, the value corresponding to the key is output. This returned value contains the physical address of the logical page and an access code to indicate its presence in main memory.

Associative memory registers are very expensive so only the most frequently accessed pages should be represented by them. How many associative memory registers are required for a virtual memory implementation to run efficiently? The percentage of times a page is found in the associative memory registers is called HIT RATIO. The effective access time of the virtual memory system can be computed by multiplying the hit ratio by the access time using associative memory and adding (1 - the hit ratio) times the access time using the main memory page table. (Remember, using the page table requires an extra access to main memory.) This total is then compared to the time for a simple access to the main memory.

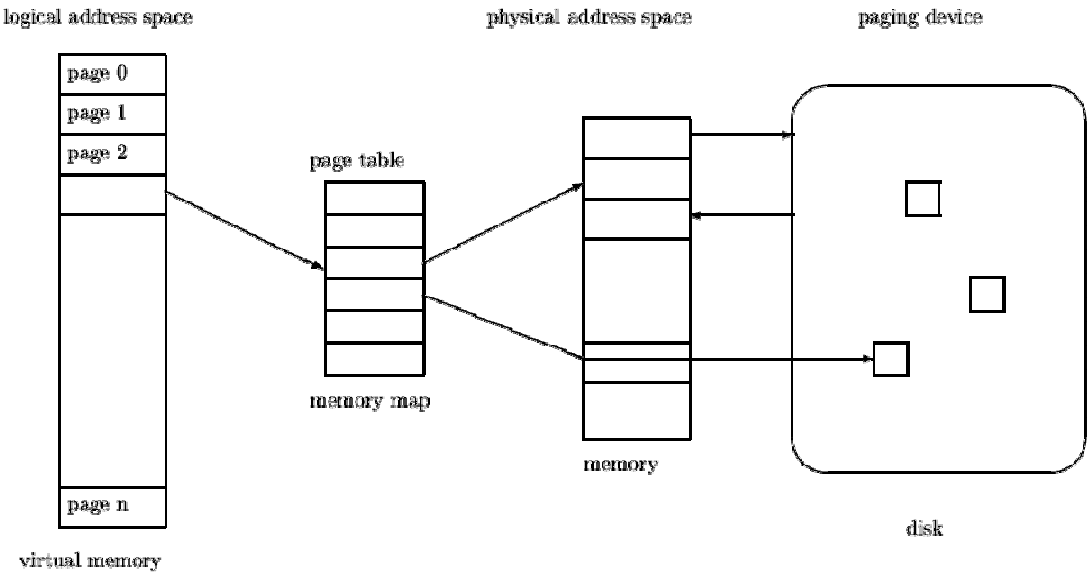
For example: If the access time for main memory is 120 nanoseconds and the access time for associative memory is 15 nanoseconds and the hit ratio is 85 percent, then

access time = .85 x (15 + 120) + (1 - .85) x (15 + 120 +120) = 153 nanoseconds. Since the simple access time is 120 nanoseconds, this represents a slowdown of 27 percent compared to the simple main memory access.

Virtual Memory up to now: all of a process in main memory (somewhere)

- partitions, pages, segments
- now: virtual memory
- allow execution of processes which may be partially in memory
- benefits:
- programs can be large and memory can be small
- increased multiprogramming IMPLIES better performance
- less I/O for loading/swapping programs
- why programs don't need to be entirely in memory:
- code for unusual error conditions
- more memory allocated than is needed
- some features of program rarely used
- VM is the separation of user logical memory from physical memory

Page Table



- logical address vs. physical address
- demand paging: only ``necessary'' pages are brought into memory

7.5.6 Page Replacement

A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide

1) Which page in main memory ought to be removed to make room for a new page

2) When a new page is to be transferred from auxiliary memory to main memory
3) Where the page is to be transferred from auxiliary memory to main mapping mechanism and the memory management software together constitute the architecture of virtual memory.
Consider the following virtual page reference sequence: page 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3. This indicates that these particular pages need to be accessed by the computer in the order shown. Consider each of the following 4 algorithm-frame combinations:

- LRU with 3 frames
- FIFO with 3 frames
- LRU with 4 frames
- FIFO with 4 frames

Print a copy of this page. For each of the 4 combinations, below, move from left to right as the virtual page numbers are requested in sequence. Put each virtual page into one of the frames by writing its number there (initially while empty frames remain, load them from top down). When all frames are already occupied by other pages, choose the right page to displace according to the applicable algorithm (LRU or FIFO) and mark the event with an F for Fault. (Do not count a fault when loading a missing page at a time when there is a frame unoccupied, in other words on the first 3 or 4 loads.) When finished, total the number of page faults and write it in where indicated.
Submit the printout. The assignment will be graded on 8 items: the 4 final page configuration figures at the extreme right (correct or incorrect), and the 4 page fault totals written (correct or incorrect). Please work carefully.

THREE Page Frames

Least-recently-used (LRU) method:

1 2 3 4 2 1 5 6 2 1 2 3

Number of page faults for LRU/3:

First-in-First-out (FIFO) method:

1 2 3 4 2 1 5 6 2 1 2 3

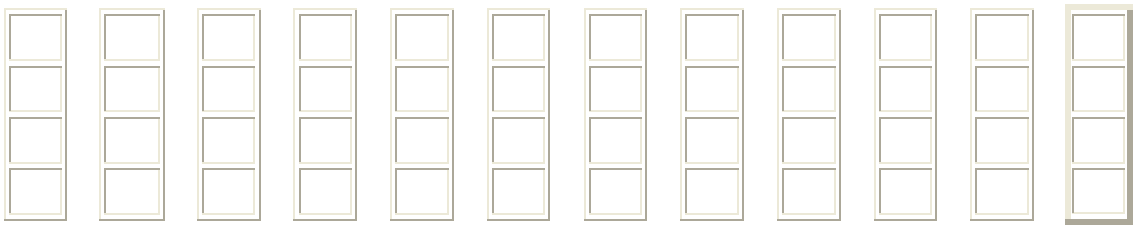
Paper Name: Computer Organization and Architecture

Number of page faults for FIFO/3:

FOUR Page Frames

Least-recently-used (LRU) method:

1 2 3 4 2 1 5 6 2 1 2 3



Number of page faults for LRU/4:

First-in-First-out (FIFO) method:

1 2 3 4 2 1 5 6 2 1 2 3

Number of page faults for FIFO/4:

Paper Name: Computer Organization and Architecture

UNIT 8

INTRODUCTION TO PARALLEL PROCESSING

8.1 .1 Parallel Processing

PARALLEL PROCESSING is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time. For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more ALUs and be able to execute two or more instructions at the same time. Furthermore, the system may have two or more processors operating concurrently. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of increase with parallel processing, and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. Shift registers operate in serial fashion one bit at a time, while registers with parallel load operate with all the bits of the word simultaneously, parallel processing at a higher level of complexity cab be achieved by have a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic logic and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

There are a variety of ways that parallel processing can be classified. It cab be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One

classification introduced by M.J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor. The sequence of instructions read from memory constitutes an *instruction stream*. The operations performed on the data in the processor constitute a data stream. Parallel processing may occur in the instruction stream, in the data stream, or in both. Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)
Single instruction stream, multiple data stream (SIMD)
Multiple instruction streams, single data stream (MISD)
Multiple instruction streams, multiple data stream (MIMD)

SISD represents the organizations of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously. MISD structure is only of theoretical interest since no practical system has been constructed using this organization. MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multi-computer systems can be classified in this category.

Flynn's classification depends on the distinction between the performance of the control unit and the data processing unit. It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining.

Here we are considering parallel processing under the following main topics:

1. Pipeline processing
2. Vector processing
3. Array processors

Pipeline processing is an implementation technique where arithmetic sub operations or the phases of a computer instruction cycle overlap in execute vector-processing deals with computations involving large vectors and matrices. Array processors computations on large arrays of data.

8.1.2 Pipelining general consideration

Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in special dedicated segments that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments. The name "pipeline" implies the flow of information analogous to an industrial assembly line. It is character of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computations is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit? The register holds the data and the combinational circuit performs the sub operation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.

The pipeline organization will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operation with a stream of numbers.

$A_i * B_i + C_i$ for I = 1, 2, 3, ……,7

Each sup operation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in fig. 9.2. R1 through R5 are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The sub operations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$ Input A_i and B_i

$R3 \leftarrow R1 * R2$, $R4 \leftarrow C$, Multiply and input C_i

$R5 \leftarrow R3 + R4$ Add C_i to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers A_1 and B_1 into R_1 and R2. The second clock pulse transfers the product of R1 and R2 into R3 and C_1 into R4. The same clock pulse transfers A_2 and B_2 into R1 and R2.The third clock pulse operates on all three

segments simultaneously. It places A_3 and B_3 into R1 and R2, transfers the product of R1 and R2 into R3, transfers C_2 into R4, and places the sum of R3 and R4 into R5. It takes three clock pulses to fill up the pipe and retrieve the first output from R5. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

Figure Example of pipeline processing

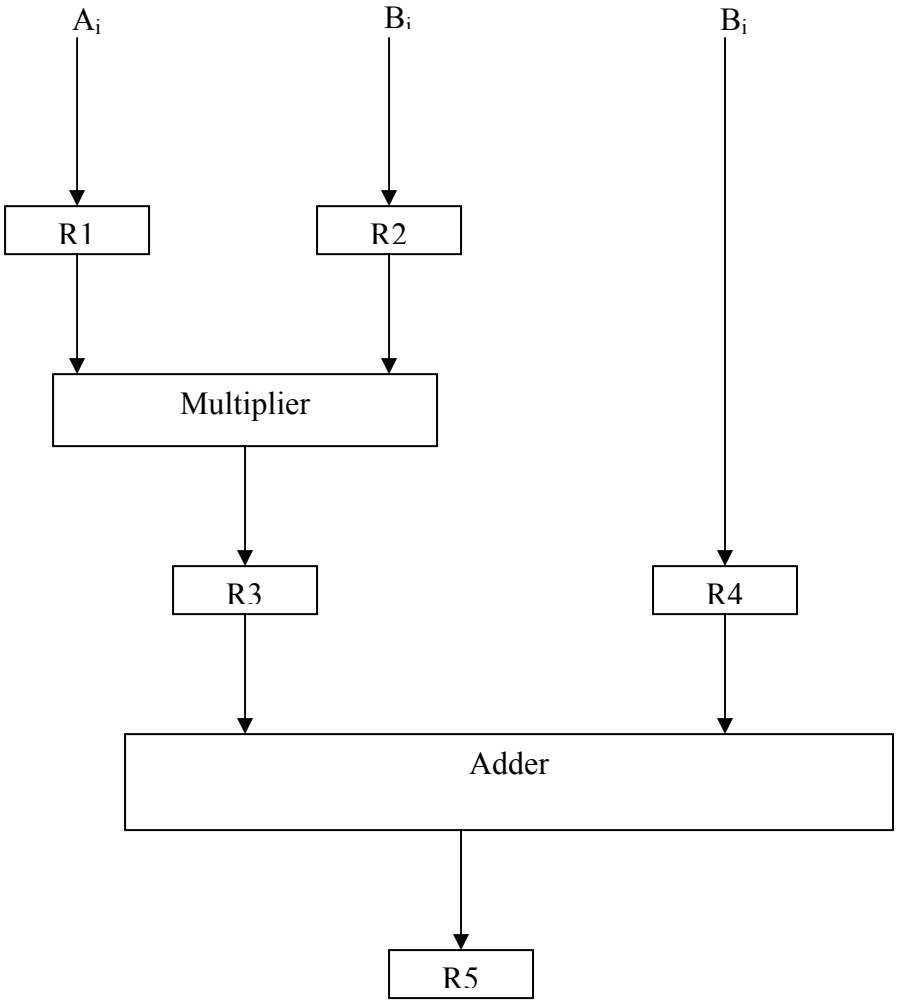


TABLE Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	- -		-
2	A_2	B_2	A_1 * B_1	C_1	-
3	A_3	B_3	A_2 * B_2	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	A_3 * B_3	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	A_4 * B_4	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	A_5 * B_5	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	A_6 * B_6	C_6	$A_5 * B_5 + C_5$

8	-	-	A_7 * B_7	C_7	$A_6 * B_6 + C_6$
9	-	-	-	-	$A_7 * B_7 + C_7$

8.1.3 Arithmetic pipeline

Arithmetic Pipeline: Pipeline arithmetic units are usually found in very high-speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. A pipeline multiplier is essentially an array multiplier as described in Fig. 10-10, with special adders designed to minimize the carry propagation time through the partial products. Floating-point operations are easily decomposed into sub operations as demonstrated in Sec. 10-5. We will now show an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$X = A \times 2^a$

$Y = B \times 2^b$

A and B are two fractions that represents the mantissas and a and b are the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 9-6. The registers labeled R are placed between the segments to store intermediate results. The sub operations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas
4. Normalize the result.

This follows the procedure outlined in the flowchart of Fig. 10-15 but with some variations that are used to reduce the execution time of the sub operations. The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This procedures and alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

The following numerical example may clarify the sub operations performed in each segment. For simplicity, we use decimal numbers, although Fig. 9-6 refers to binary numbers. Consider the two normalized floating-point numbers:

$X = 0.9504 \times 10^3$

$Y = 0.8200 \times 10^2$

The two exponents are subtracted in the first segment to obtain 3 – 2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$X = 0.9504 \times 10^3$

$Y = 0.0820 \times 10^3$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 procedures the sum

$Z = 1.0324 \times 10^3$

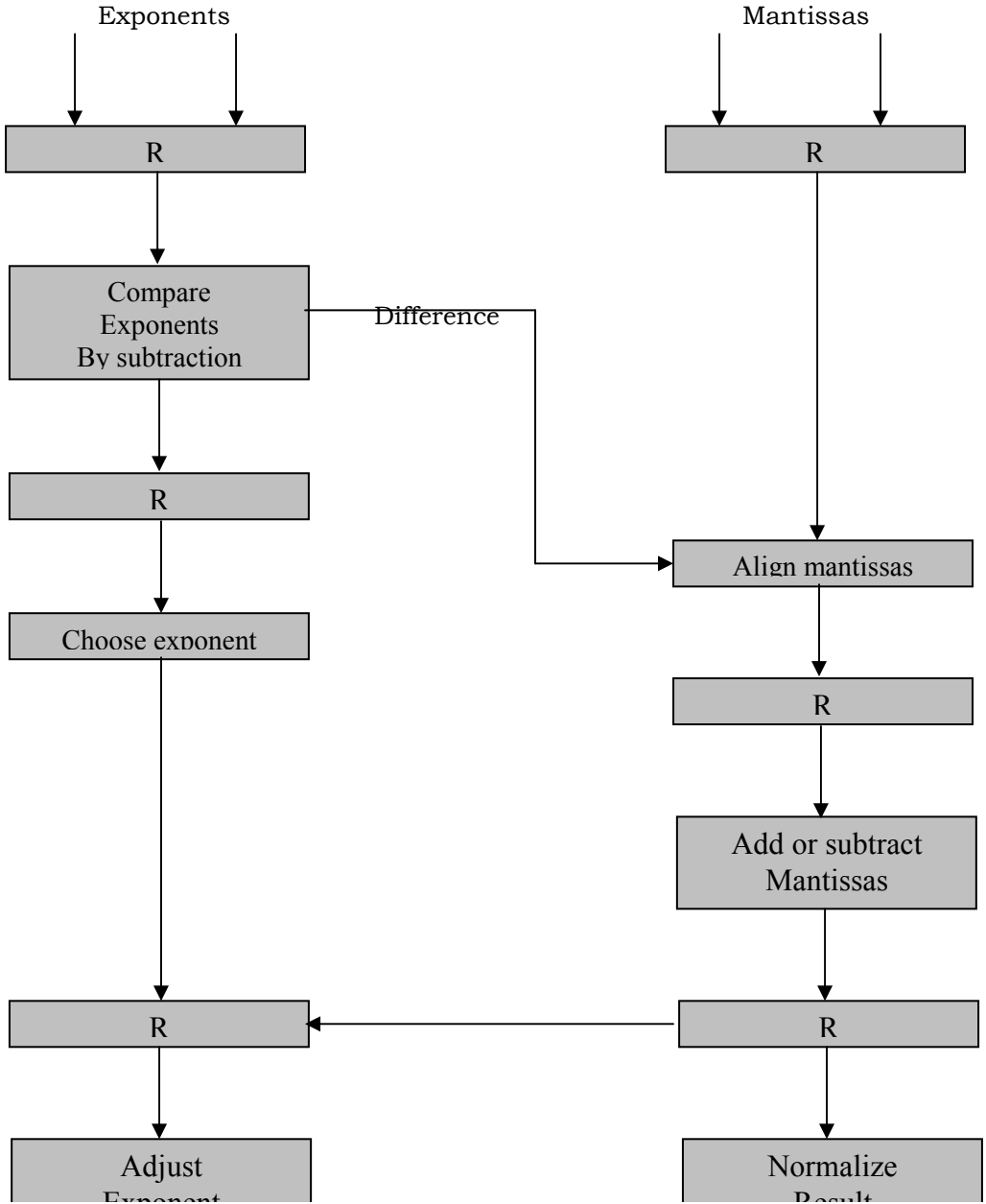


Fig. Pipeline for floating point addition and subtraction

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.
$Z = 0.10324 \text{ X } 10^4$

The comparator, shifter, adder subtraction, incrementer, and decremented in the floating point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are t_1 = 60ns, t_2 = 70ns, t_3 = 100ns, t_4 = 80ns, and the interface register have a delay of t_r = 10ns. The clock circle is chosen to be $t_p = t_3 + t_r$ = 110ns. An equivalent no pipeline floating point adder subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r$ = 320 ns. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the nonpipelined adder.

8.1.4 Instruction pipeline

Instruction Pipeline: Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This cause the instructions fetch and execute phases to overlap and perform simultaneous operations. One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

Paper Name: Computer Organization and Architecture

Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline. The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack. Whenever the execution unit is not using memory, the control increments the program counter and uses its address value to read consecutive instructions form memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis. Thus an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment. The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions. Whenever there is space in the FIFO buffer, the control unit initiates the next instruction fetch phase. The buffer acts as a queue form which control then extracts the instructions for the execution unit.

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction with the following sequence of steps.

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.

The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way it is.

Paper Name: Computer Organization and Architecture

UNIT 9

VECTOR PROCESSING

9.1 Vector operations
9.2 Matrix multiplication
9.3 Memory interleaving

9.1 Vector operations

There are certain computational problems that can not be resolved by a are beyond the capabilities of a conventional computer on the basis of the fact that they require a vast number of computations that will take a conventional computer days or even weeks to complete. In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that tend themselves to vector processing. Computers with vector processing capabilities are very much required in specialized applications. The following are representative application areas where vector processing is of the utmost importance:

- Long-range weather forecasting
- Petroleum explorations
- Seismic data analysis
- Medical diagnosis
- Aerodynamics and space flight simulations
- Artificial intelligence and expert systems
- Mapping the human genome
- Image processing

Vector and parallel processing techniques should be applied to achieve result of high performance in case of unavailability of complex computers to speed up the execution time of instructions.

Many scientific problems require arithmetic operations on large arrays of numbers. These numbers are usually formulated as vectors and matrices of relating point numbers. To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortan Do loop:

```
DO  20  I = 1, 100
20: C (I) = B(I) + A (I)
```

This is a program for adding two vectors A and B of length 100 produces a vector C.
This is implemented in machine language by the following sequence of operations:

```
        Initialize I = 0
20      Read A (I)
        Read B (I)
```

Store C (I) = A(I) +B (I)
Increment I = I +1
If I < 100 go to 20 continue

This constitutes a program loop that reaches a pair of operations from arrays A and B and perform a floating point addition. The loop control variable is then updated and the steps repeat 100 times.

A computer capable of vector processing eliminates the overheads associated with the time it takes to fetch and execute the instruction in the program loop. It allows operations to be specified with a single vector instructions of the form

C (1:100) = A (1: 100) +B (1: 100)

The vector instructions include the initial address of the operands, the length of the vectors and the operation to be performed, all in one composite instruction. The addition is done with a pipelined floating point adder. A possible instruction format for a vector instructions is shown Figure 5.14.

Operation Code	Base address source 1	Base address source 2	Base address destination	Vector Length

Figure 5.14

This is essentially a three address instruction with three fields specifying the base address of the operands and an additional field that gives the length of the data items in the vectors.

Example

9.2 Matrix Multiplication

Matrix multiplication is one of the most computational intensive operations performed in computers with vector processors. The multiplication of two n x n matrices consists of n^2 inner products or n^3 multiply-add operations. An n x m matrix of numbers has n rows and m columns and may be considered as constituting a set of n row vectors or a set of m column vectors. Consider, for example, the multiplication of two 3 x 3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix C is a 3 x 3 matrix whose elements are related to the elements of A and B by the inner product:

$$C_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

For example, the number in the first row and first column of matrix C is calculated by letting i = 1, j = 1, to obtain

$C_{11} = a_{11}\, b_{11} + a_{12}\, b_{21} + a_{13}\, b_{31}$

This requires three multiplications and (after initializing c_{11} to 0) three additions. The total number of multiplications or additions required to compute the matrix product is 9 x 3= 27. If we consider the linked multiply-add operation c = a x b as a cumulative operation, the product of two n x n matrices requires n^2 multiply-add operations. The computation consists of n^2 inner products, with each inner product requiring n multiply-add operations, assuming that c is initialized to zero before computing each element in the product matrix.

In general, the inner product consists of the sum of k product terms of the form

$C = A_1B_1 + A_2\, B_2 + A_3 + B_3 + A_4 + B_4 + \ldots + A_k\, B_k$

In a typical application k may be equal to 100 or even 1000. The inner product calculation on a pipeline vector processor is shown in Fig. 5.15. The values of A and B are either in memory or in processor registers. The floating-point multiplier pipeline and the floating-point adder pipeline are assumed to have four segments each. All segment registers in the multiplier and adder are initialized to 0. Therefore, the output of the adder is 0 for the first eight cycles until both pipes are full. A_i and B_i pairs are brought in and-multiplied at a rate of one pair per cycle. After the first four cycles, the products begin to be added to the output of the adder. During the next four cycles 0 is added to the products entering the adder pipeline. At the end of the eighth cycle, the first four products $A_1\ B_1$ through $A_4\ B_4$ are in the four adder segments, and the next four products, A_5B_5 through $A_8\ E_8$ are in the multiplier segments. At the beginning of the ninth cycle, the output of the adder is A_1B_1 and the output of the multiplier is $A_5\ B_5$. Thus the ninth cycle starts the addition $A_1\ B_1 + A_5\ B_5$ in the adder pipeline. The tenth cycle starts the addition $A_2\ B_2 + A_6\ B_6$, and so on. This pattern breaks down the summation into four sections as follows:

$$
\begin{aligned}
C = {} & A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \cdots \\
& + A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \cdots \\
& + A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \cdots \\
& + A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \cdots
\end{aligned}
$$

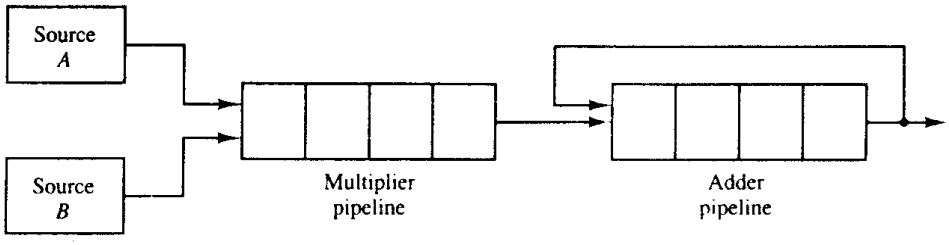


Figure 5.15: Pipeline for calculating an inner product

9.3 Memory interleaving

Interleaving is an advanced technique used by high-end motherboards/chipsets to improve memory performance. Memory interleaving increases bandwidth by allowing simultaneous access to more than one chunk of memory. This improves performance because the processor can transfer more information to/from memory in the same amount of time, and helps alleviate the processor-memory bottleneck that is a major limiting factor in overall performance.

Interleaving works by dividing the system memory into multiple blocks. The most common numbers are two or four, called *two-way* or *four-way* interleaving, respectively. Each block of memory is accessed using different sets of control lines, which are merged together on the memory bus. When a read or write is begun to one block, a read or write to other blocks can be overlapped with the first one. The more blocks, the more that overlapping can be done. As an analogy, consider eating a plate of food with a fork. Two-way interleaving would mean dividing the food onto two plates and eating with both hands, using two forks. (Four-way interleaving would require two more hands. :^)) Remember that here the processor is doing the "eating" and it is much faster than the forks (memory) "feeding" it (unlike a person, whose hands are generally faster.)

In order to get the best performance from this type of memory system, consecutive memory addresses are spread over the different blocks of memory. In other words, if you have 4 blocks of interleaved memory, the system doesn't fill the first block, and then the second and so on. It uses all 4 blocks, spreading the memory around so that the interleaving can be exploited.
Interleaving is an advanced technique that is not generally supported by most PC motherboards, most likely due to cost. It is most helpful on high-end systems, especially servers, that have to process a great deal of information quickly. The Intel Orion chipset is one that does support memory interleaving.

Paper Name: Computer Organization and Architecture

UNIT 10

MULTIPROCESSORS

10.1 Introduction to MULTIPROCESSORS

Characteristics of Multiprocessors: A multiprocessors system is an interconnection of two or more CPUs with memory and input-output equipment. The term "processor" in *multiprocessor* can mean either a central processing unit (CPU) or an input-output processor (IOP).

Computers are interconnected with each other by means of communication lines to form a *computer network*. The network consists of several autonomous computers that may or may not communicate with each other. A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.

Multiprocessing improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to function correctly with perhaps some loss in efficiency.

The benefit derived from a multiprocessors organization is an improved system performance. The system derives its high performance from the fact that computations can proceed in parallel in one of two ways.

1. Multiple independent jobs can be made to operate in parallel.
2. A single job can be partitioned into multiple parallel tasks.

10.2. Interconnection Structures:

The components that form a multiprocessors system are CPUs, IOPs connected to input-output devices, and a memory unit that may be partitioned into a number of separate modules. The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system. There are several physical forms available for establishing an interconnection network. Some of these schemes are presented in this section:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

10.2.1 Time-shared Common Bus:

A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in fig. 13-1. Only one processor can communicate with the memory or another processor at any given time. Transfer operations are conducted by the processor that is in control of the bus at the time. Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination unit to initiate the transfer. A command is issued to inform the destination unit what operation is to be performed. The receiving unit recognizes its address in the bus and responds to the control signals from the sender, after which the transfer is initiated. The system may exhibit transfer conflicts since one common bus is shared by all processors. These conflicts must be resolved by incorporating a bus controller that establishes priorities among the requesting units.

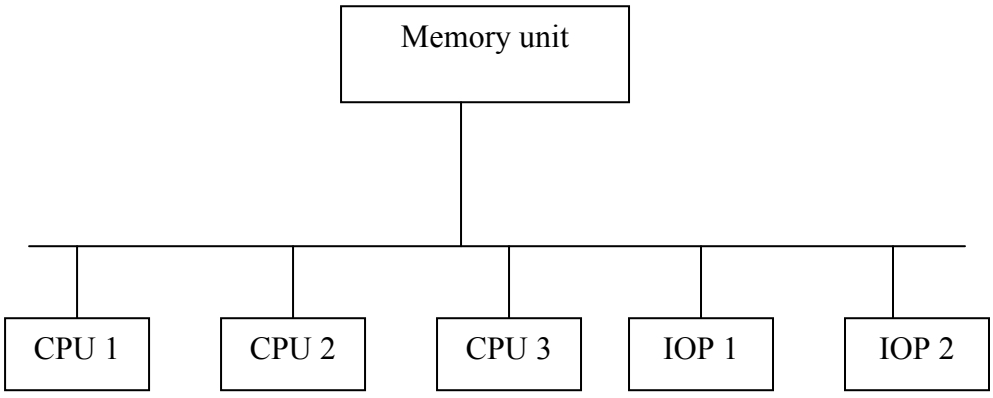
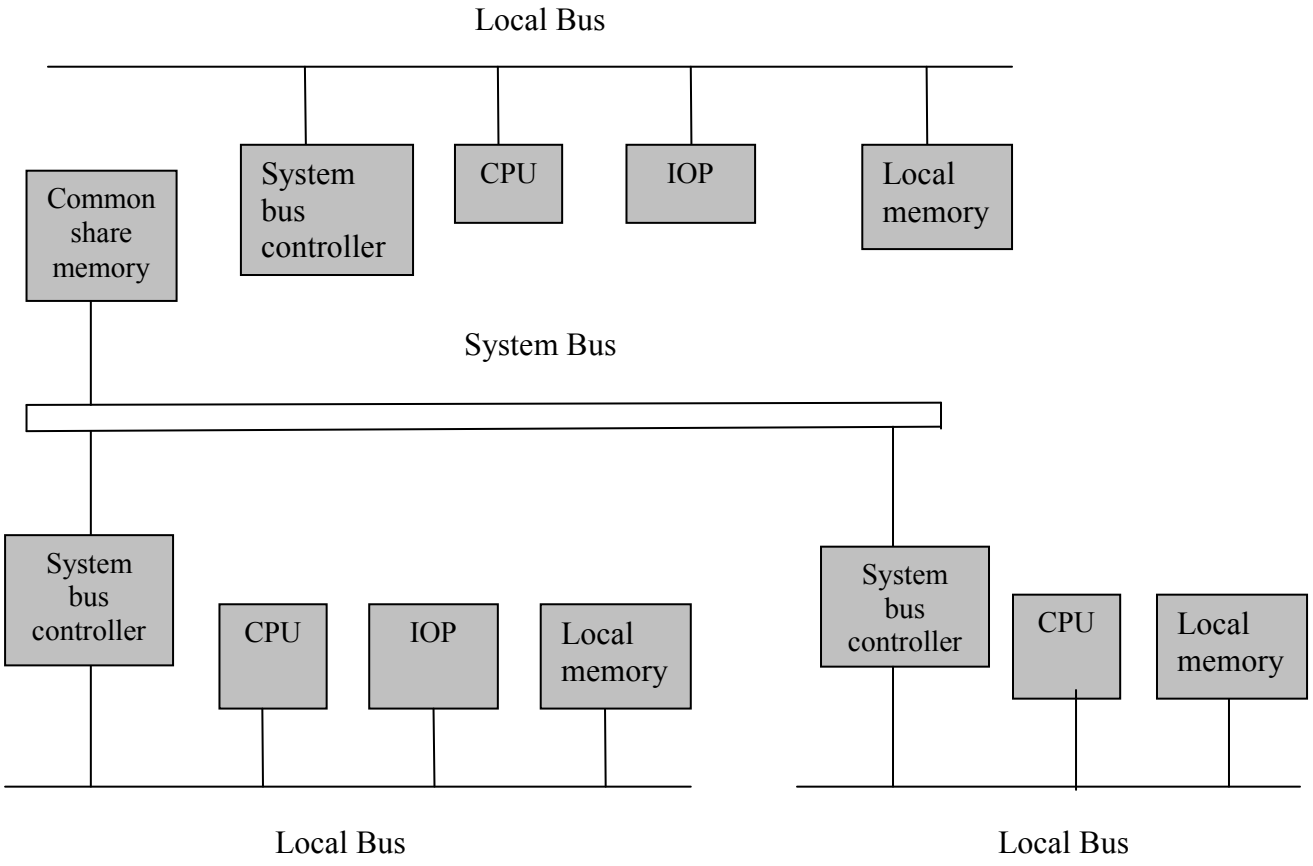


Figure 13-1 Time-shared common bus organization.

A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path. The processors in the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers. However, this increases the system cost and complexity.

A more economical implementation of a dual bus structure is depicted in Fig. Here we have a number of local buses each connected to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combination of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus, the I/O devices attached to it may be designed as a cache memory attached to the CPU (see Sec. 12-6). In this way, the average access time of the local memory can be made to approach the cycle time of the CPU to which it is attached.



10.2.2 Multi-port Memory:

A multiport memory system employs separate buses between each memory module and each CPU. This is shown in Fig. 13-3 for four CPUs and four memory modules (MMs). Each processor bus is connected to each memory module. A processor bus consists of the address, data and control lines required to communicate with memory. The memory module is said to have four ports and each port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port. The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module. Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.

The advantage of the multiport memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory. The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors. As a consequence, this interconnection structure is usually appropriate for systems with a small number of processors.

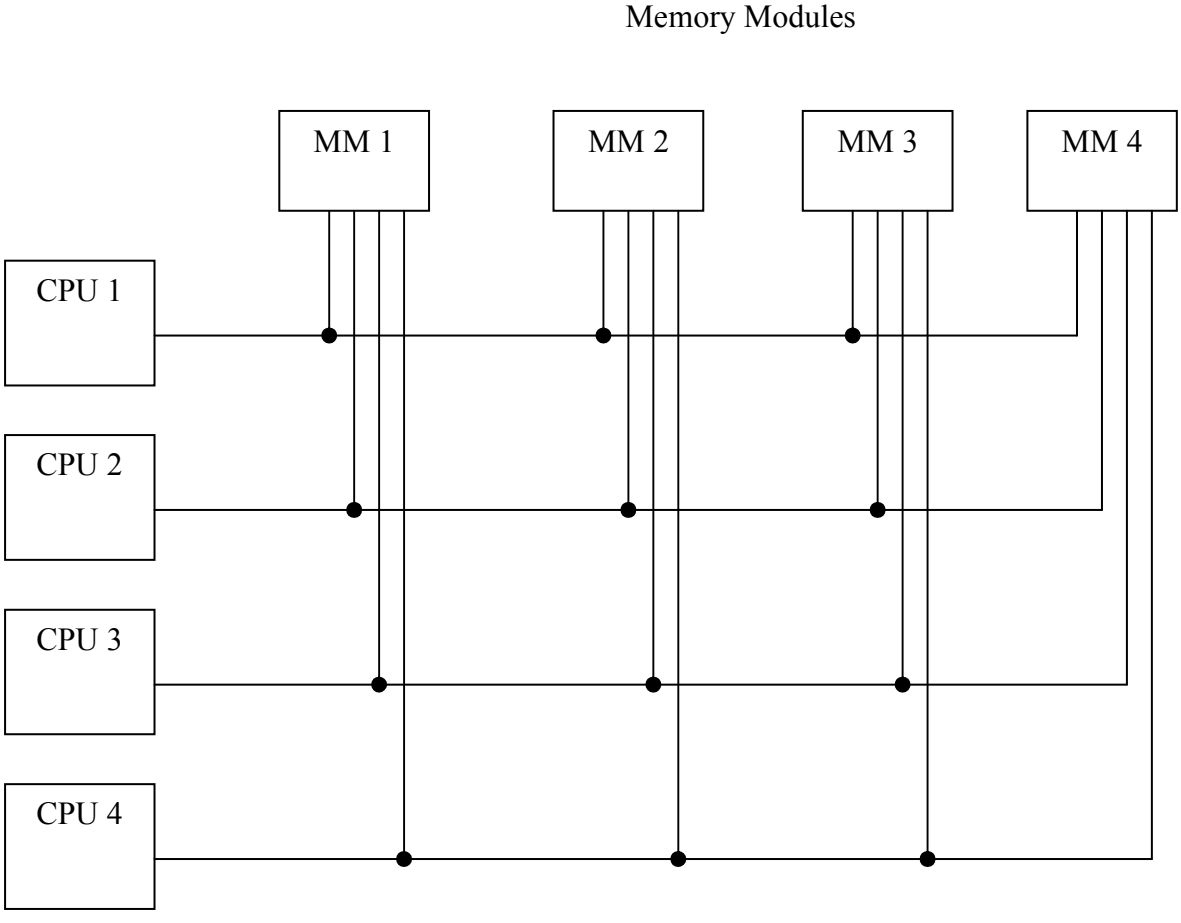


Figure 13-3 Multiport memory organization.

10.2.3 Crossbar Switch

The crossbar switch organization consists of a number of cross prints that are placed at intersections between processor buses and memory module paths. Figure shows a crossbar switch interconnection between four CPUs and four memory modules. The small square in each cross point is a switch that determines the path from a processor to a memory module. Each switch point has control logic to set up the transfer path between a processor and memory. It examines the address that is placed in the bus to determine whether its particular module is being addressed. It also resolves multiple requests for access to the same memory module on a predetermined priority basis.

Figure shows the functional design of a crossbar switch connected to one memory module. The circuit consists of multiplexes that select the data, address, and control from one CPU for communication with the memory module. Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory. The multiplexes are controlled with the binary code that is generated by a priority encoder within the arbitration logic.

A crossbar switch organization supports simultaneous transfers from all memory modules because there is a separate path associated with each module. However, the hardware required to implement the switch could be quite large and complex.

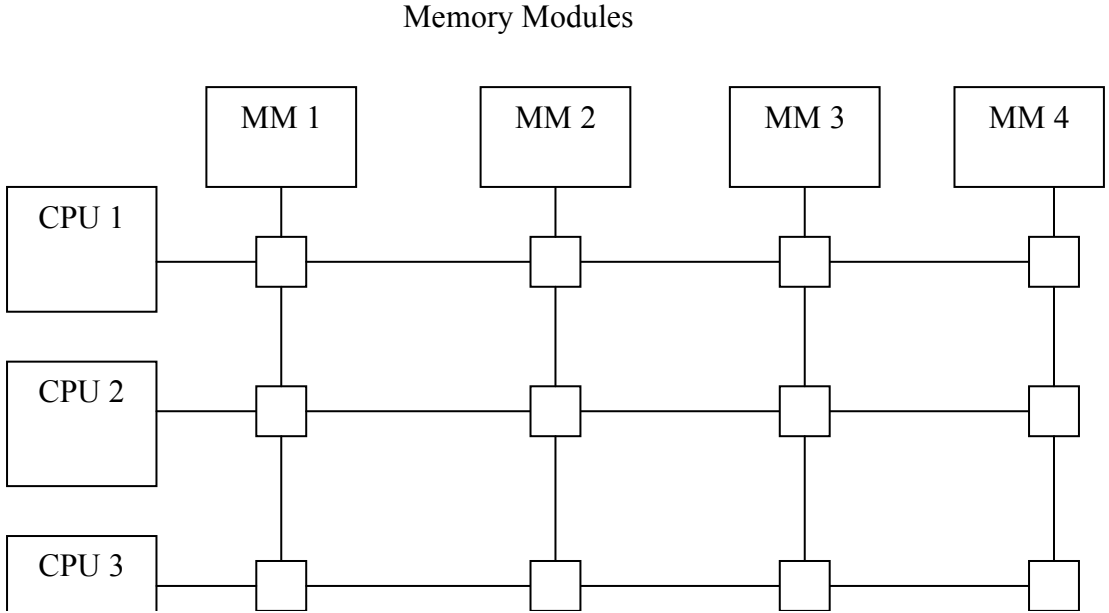


Figure 13-4 Crossbar Switch

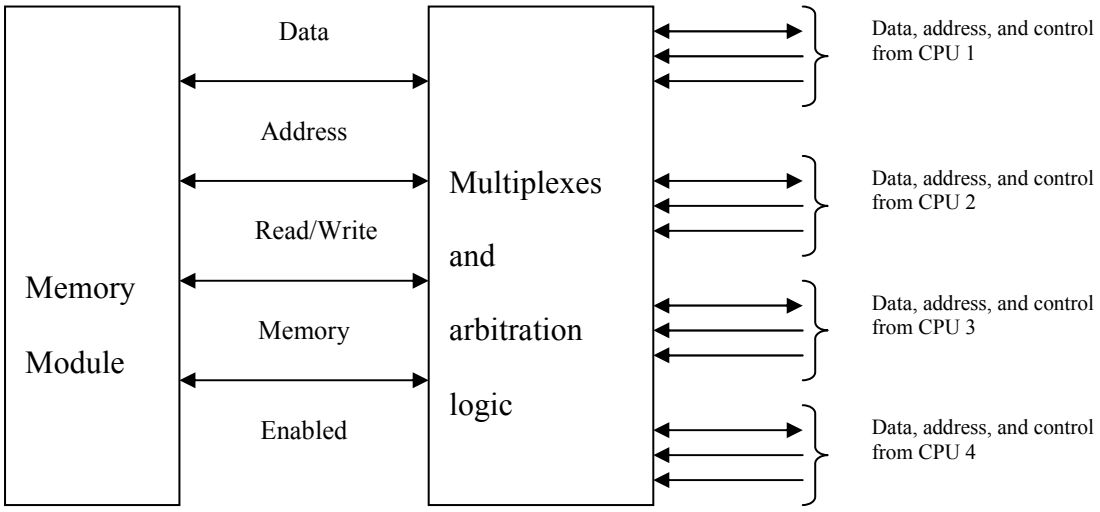


Figure Block diagram of crossbar switch

10.2.4 Multistage Switching Network:

The basic component of a multistage network is a two-input, two-output interchange switch. As shown in fig. the 2 X 2 switch has two inputs, labeled A and B, and two outputs, labeled 0 and 1. There are control signals (not shown) associated with the switch that establish the interconnection between the input and output terminals. The switch has the capability of connecting input A to either of the outputs. Terminal B of the switch behaves in a similar fashion. The switch also has the capability to arbitrate between conflicting requests. If inputs A and B both request the same output terminal, only one of them will be connected; the other will be blocked.

Using the 2 X 2 switch as a building block, it is possible to build a multistage network to control the communication between a number of source and destinations.

To see how this is done, consider the binary tree shown Fig. 13-7. The two processors P_1 and P_2 are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to a destination is determined from the binary bits of the destination number. The first bit of the destination number determines the switch output in the first level. The second bit specifies the output of the switch in the second level, and third bit specifies the output of the switch in the third level. For example, to connect P_1 to memory 101, it is necessary to form a path from P_1 to output 1 in the first level switch, output 0 in the second-level switch, and output 1 in the third-level switch. It is clear that either P1 or P2 can be connected to any one of the eight memories. Certain request patterns, however, cannot be satisfied simultaneously. For example, if P1 is connected to one of the destinations 000 through 011,P2 can be connected to only one of the destinations 100 through 111.

Many different topologies have been proposed for multistage switching networks to control processor-memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system. One such topology is the omega-switching network shown in Fig. 13-8. In this configuration, there is exactly one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

Figure Operation of 2 X 2 interchange switch.

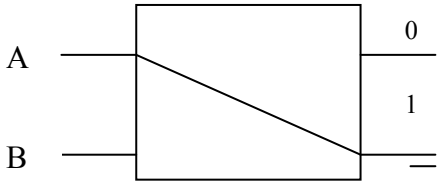
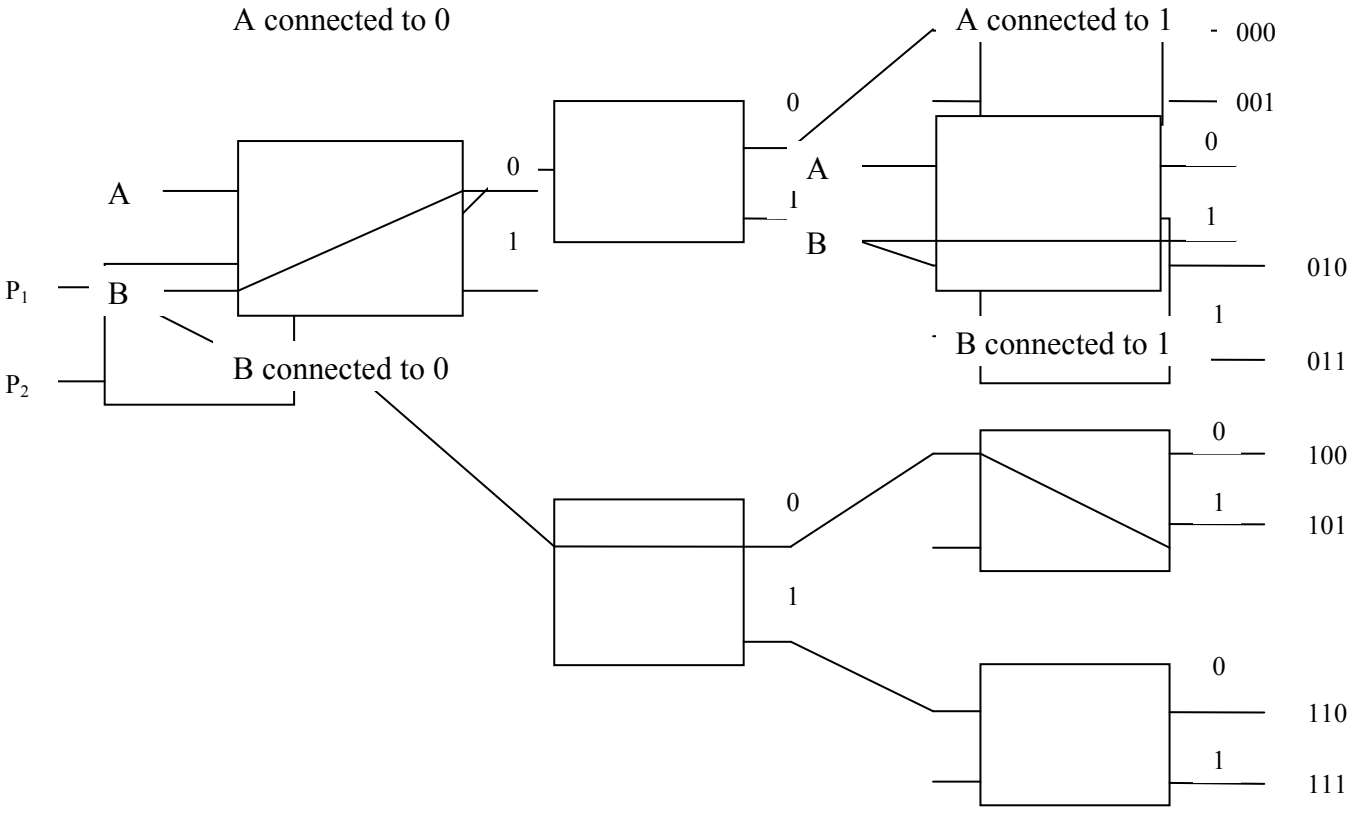
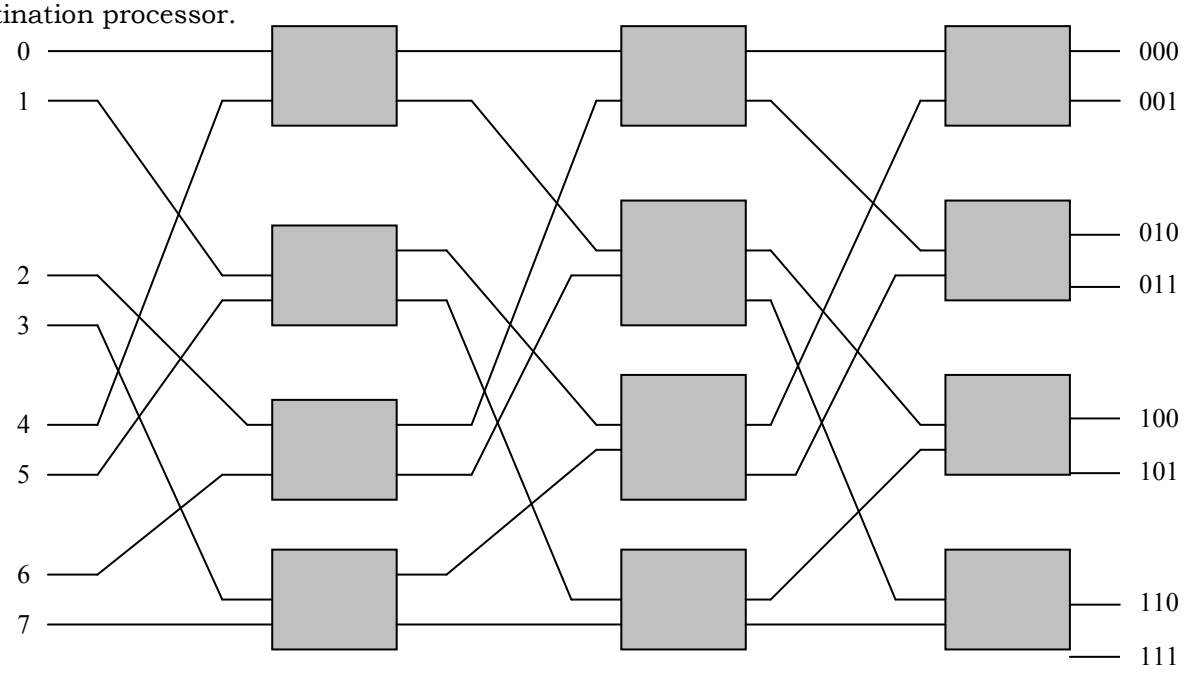


Figure Binary tree with 2 X 2 switches.



A particular request is initiated in the switching network by the source, which sends a 3- bit pattern representing the destination number. As the binary pattern moves through the network, each level examines a different bit to determine the 2 X 2 switch setting. Level 1 inspects the most significant bit, level 2 inspects the middle bit, and level 3 inspects the least significant bit. When the request arrives on either input of the 2 X 2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if the bit is 1.

In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module. The first pass through the network sets up the path. Succeeding passes are used to transfer the address into memory and then transfer the data in either direction, depending on whether the request is a read or a write. In a loosely coupled multiprocessor system, both the source and destination are processing elements. After the path is established, the source processor transfers a message to the destination processor.

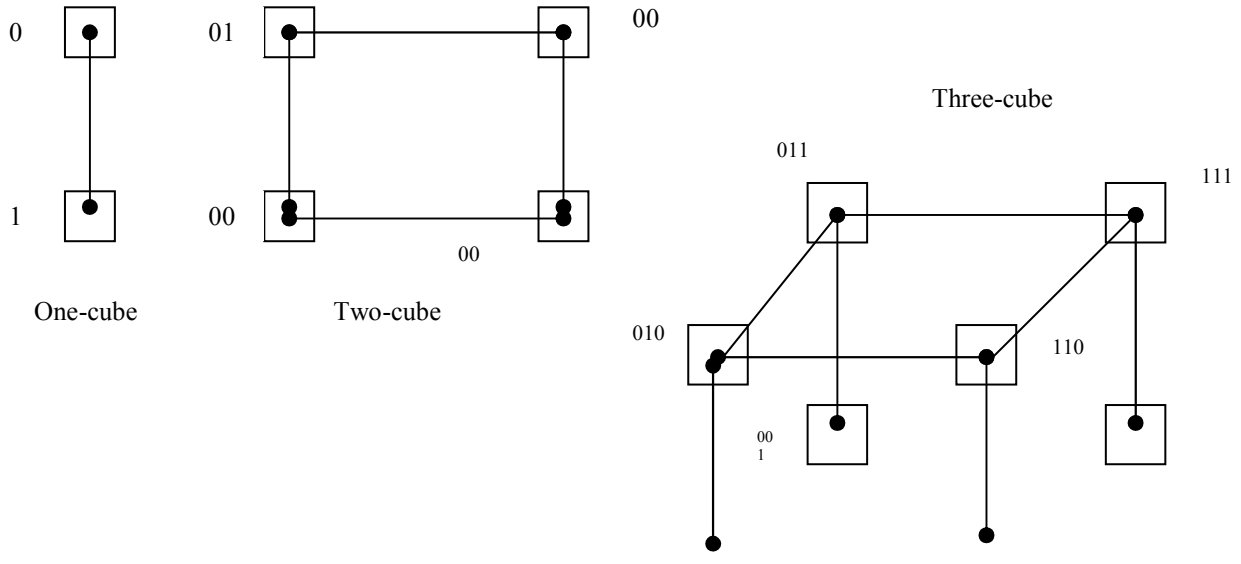


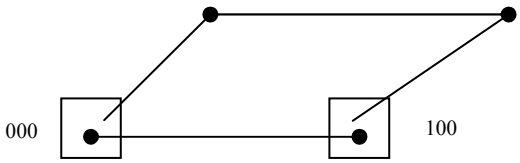
10.2.5 Hypercube Interconnection

The hypercube or binary n – cube multiprocessor structure is a loosely coupled system composed of $N = 2^n$ processor interconnected in an n- dimensional binary cube. Each processor forms a node of the cube. Although it is customary to refer to each node as having a processor, in effect it contains not only a CPU but also local memory and I/O interface. Each processor has direct communication paths to n other neighbor processors. These paths correspond to the edges of the cube. There are 2^n distinct n- bit binary addresses that can be assigned to the processors. Each processor address differs from that of each of its n neighbors by exactly one bit position.

Figure 13-9 shows the hypercube structure for n = 1,2, and 3. A one-cube structure has n = 1 and 2" = 2. It contains two processor interconnected by a single path. A two-cube structure has n= 2 and 2" = 4. It contains four nodes interconnected as a square. A three- cube structure has eight nodes interconnected as a cube. An n – cube structure has 2" nodes with a processor residing in each node. Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position. For example, the three neighbors of the node with address 100 in a three – cube structure are 000, 110,and 101 each of these binary numbers differs from address 100 by one bit value.

Routing message through an n-cube structure may take from one to n links from a source node to a destination node. For example, in a three- cube structure, node to a destination node. For example, in a three- cube structure, node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011 (from 000 to 001 to 011 or from 000 to 010 to 011). It is necessary to go through at least three links to communicate from node 000 to node 111. A routing procedure can be developed by computing the exclusive –OR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes. For example, in a three- cube structure, a message at 010 going to 001 produces an exclusive- OR of the two-address equal to 011. The message can be sent along the second axis to 000 and then through the third axis to 001.





A representative of the hypercube architecture is the Intel Ipsc computer complex. It consists of 128 (n = 7) microcomputers connected through communication channels. Each node consists of CPU, a floating-point processor, local memory, and serial communication interface units. The individual nodes operate independently on data stored in local memory according to resident programs. The data and programs to each node com 000 ough a message- passing system from other nodes or from a cube manager. Application program are developed and complied on the cube manager and then download to the individual nodes. Computations are distributed through the system and executed concurrently.

10.3 Interprocessor Arbitration

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. The CPU contains a number of internal buses for transferring information between processor registers and ALU. A memory bus consists of lines for transferring data, address, and read/write information. An I/O bus is used to transfer information to and from input and output devices. A bus that connects major components in a multiprocessor system, such as CPUs, IOPs, and memory, is called a system bus. The physical circuits of a system bus are contained in a number of identical printed circuit boards. Each board in the system belongs to a particular module. The board consists to circuits connected in parallel through connectors. Each pin of each circuit connector is connected by a wire to the corresponding pin of all other connectors in other boards. Thus any board can be plugged into a slot in the back plane that forms the system bus.

The processors in a shared memory multiprocessor system request access to common memory or other common resources through the system bus. If no other processor is currently utilization the bus, the requesting processor may be granted access immediately. However, the requesting processor must wait if another processor is currently utilizing the system bus. Furthermore, other processors may request the system bus at the same time. Arbitration must then be performed to resolve this multiple contention for the shared resources. The arbitration logic would be part of the system bus controller placed between the local bus and the system bus as shown in fig. 13 – 2.

10.4 Cache Coherence

The primary advantage of cache is its ability to reduce the average access time in uniprocessors. When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer. If the operation is to write, there are two commonly used procedures to update memory. In the write – through policy, both cache and main memory are updated with every write operation. In the write-back policy, only the cache is updated and the location is marked so that it can be copied later into main memory.

In a shared memory multiprocessor system, all the processors share a common memory. In addition, each processor may have a local memory, part or all of which may be a cache. The compelling reason for having separate caches for each processor is to reduce the average access time in each processor. The same information may reside in a number of copies in some caches and main memory. To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical. This requirement imposes a cache coherence problem. A memory scheme is coherent if the value returned on a load instruction is always the value given by the cache coherence problem, caching cannot be used in bus – oriented multiprocessors with two or more processors.

Conditions for incoherence

Cache coherence problems exist in multiprocessors with private caches because of the need to share writ able data. Read – only data can safely be replicated without cache coherence enforcement mechanisms. To illustrate the problem consider the three – processor configuration with private caches shown in Fig. 13 – 12. Sometime during the operation an element X from main memory is loaded into the three processors, P1, P2 and P3.As a consequence, it is also copied into the private caches of the three processors. For simplicity, we assume as that X contains the value of 52. The load on X to the three processors results in consistent copied in the caches and main memory.

If one of the processor performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value. Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache. This is shown in fig. 13-13. A store to X (of the value of 120) into the cache of processor P_1 updates memory to the new value in a write – through policy. A write- through policy maintains consistency between memory and the originating cache, but the other caches are inconsistent since they still hold the old value. In a write – back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory is inconsistent. Memory is updated eventually when the modified data in the cache are copied back into memory.

Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus. In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMA output, memory locations may be read

before they are updated from the cache when using the write back policy. I/O – based memory incoherence can be over- come by making the IOP a participant in the cache coherent solution that is adopted in the system.

Solutions to the Cache Coherence Problem

Various schemes have been proposed to solve the cache coherence problem in shared memory multiprocessors. We discuss some of these schemes briefly here. See references 3 and 10 fro more detailed discussions.

A simple scheme is to disallow private caches for each processor and have a shared cache memory associated with main memory. Every data access is made to the shared cache. This method violates the principle of closeness of CPU to cache and increases the average memory access time. In effects, this scheme solves the problem by avoiding it.

For performance considerations it is desirable to attach as private cache to each processor. One scheme that has been used allows only non-shared and read only data to be stored in caches. Such items are called cacheable. Shared writable data are non-cacheable. The compiler must tag data as either cacheable or non-cacheable, and the system hardware makes sure that only cacheable data are stored in caches. The non-cacheable data remain memory. This method restricts the type of data stored in caches and introduces an extra software overhead that may degrade performance.

A scheme that allows writable data to exist in at least one cache is a method that employs a *centralized global table* in its compiler. The status of memory blocks is stored in the central global table. Each block is identified *read-only* (RO) or *read and write* (RW). All caches can have copies of block identified as RO. Only one cache can have a copy of an RW block. Thus if the data are updated in the cache with an RW block, the other caches are not affected because they do not have a copy of this block.

The cache coherence problem can be solved by means of a combination of software and hardware or by means of hardware-only schemes. The two methods mentioned previously use software-based procedures that require the ability to tag information in order to disable caching of shared writable data. Hardware-only solutions are handled by the hardware automatically and have the advantage of higher speed and program transparency. In the hardware solution, the cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs. All caches attached to the bus constantly monitor the network for possible write operations. Depending on the method used, they must then either update or invalidate their own cache copies when a match is detected. The bus controller that monitors this action is referred to as a *snoopy cache controller.* This is basically a hardware unit designed to maintain a bus-watching mechanism over all the caches attached to the bus.

Various schemes have been proposed to solve the cache coherence problem by means of snoopy cache protocol. The simplest method is to adopt write-through policy and use the following procedure. All the snoopy controllers watch the bus for memory store operations. When a word in a cache is updated by writing into it, the corresponding location in main memory is also updated. The local snoopy controllers in all other caches check the memory to determine if they have a copy of the word that has been overwritten. If a copy already exists in a remote cache, that location is marked invalid. Because all caches snoop on all bus writes, whenever a word is written, the net effect is to update it in the original cache and main memory and remove it from all other caches. If al some future time a processor accesses the invalid item from its cache, the response is equivalent to a cache miss, and the updated item is transferred from main memory. In this way, inconsistent versions are prevented.

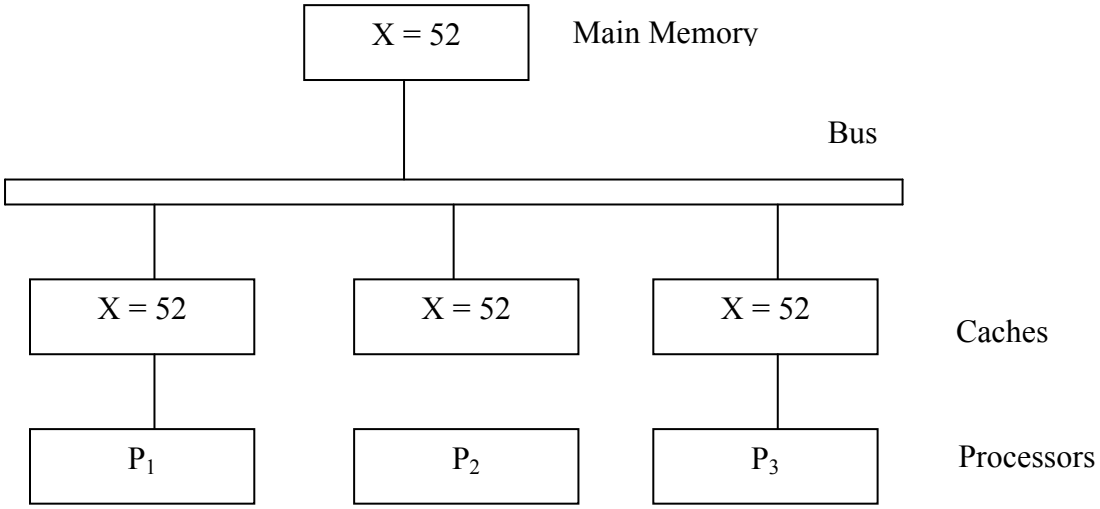
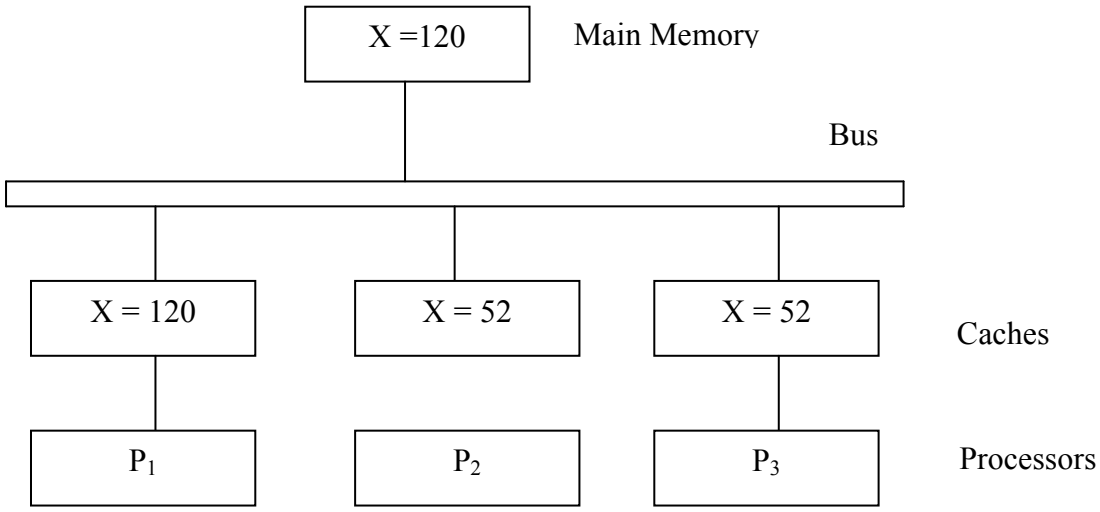


Figure Cache configurations after a store to X by processor P_1.



(a) With write-through cache policy