NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

Moulding True Citizen
Since 1968

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIAL

CST 202 COMPUTER ORGANIZATION AND ARCHITECTURE

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Courses offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE OUTCOMES

SUBJECT CODE: C210		
COURSE OUTCOMES		
C210.1	K2	**Recognize** and express the relevance of basic components, I/O organization and pipelining schemes in a digital computer.
C210.2	K4	**Illustrate** the design of Arithmetic Logic Unit and explain the usage of registers in it
C210.3	K5	**Explain** the implementation aspects of arithmetic algorithms in a digital computer
C210.4	K3	**Demonstrate** the control signals required for the execution of a given instruction and **Develop** the control logic for a given arithmetic problem
C210.5	K5	**Explain** the types of memory systems and mapping functions used in memory systems

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software
products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

CO PO MAPPING

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C210.1	3	3	3	3	-	-	-	-	-	-	-	3
C210.2	3	3	3	3	-	-	-	-	-	-	3	3
C210.3	3	3	3	-	-	-	-	-	-	-	3	3
C210.4	3	3	3	3	-	-	-	-	-	-	3	3
C210.5	3	3	3	3	-	-	-	-	-	-	3	3
C210	**3**	**3**	**3**	**3**	-	-	-	-	-	-	**3**	**3**

CO PSO MAPPING

CO'S	PSO1	PSO2	PSO3
C210.1	-		-
C210.2	-	-	-
C210.3	-	-	-
C210.4	-	-	-
C210.5	-	-	-
C210	-	-	-

S:NO;	TOPIC
1	**Von Neumann architecture**
2	**Computer Organization and Architecture | Pipelining**

MODULE NOTES & QUESTION BANK

Module I

Qn No	Question	Knowledge Level	CO
1	Describe phases of instruction execution.	K3	CO1
2	List the memory operations	K2	CO1
3	Write short notes on single bus operations	K3	CO1
4	Explain functional units of computers	K5	CO1
5	Explain basic input output operations	K5	CO1
6	Explain single bus organization with neat diagram	K5	CO1
7	Write short notes on 1. MAR 2. MDR 3. PC	K3	CO1
8	Discuss different addressing modes	K4	CO1
9	Discuss Memory addressability	K4	CO1
10	Discuss Connection between processor and memory	K4	CO1
11	Differentiate between Big endian and Little endian addressing	K4	CO1
12	Describe basic instruction types	K3	CO1

Module II

Qn No	Question	Knowledge Level	CO
1	Explain the design of 4 bit Arithmetic circuit to perform basic arithmetic operations	K3	CO2
2	List the micro operations on	K2	CO2
3	Write short notes on Status registers	K3	CO2
4	Explain ALU design	K5	CO2
5	Explain Design of Logic Circuits	K5	CO2
6	Explain design of Accumulator with neat diagram	K5	CO2
7	Write short notes on Register transfer logic.	K3	CO2
8	Discuss design of Shifter.	K4	CO2
9	Discuss Binary incrementer with neat diagram.	K4	CO2
10	Discuss Connection between processor and memory.	K4	CO2
11	Describe different logic micro operations	K4	CO2
12	Describe basic shift operations	K3	CO2

Module III

Qn No	Question	Knowledge Level	CO
1	List and explain the different pipeline hazards and their possible solutions	K3	CO3
2	Give the logic used behind Booth's multiplication algorithm.	K2	CO3
3	Identify the appropriate algorithm available inside the system to perform the multiplication between -14 and -9. Also trace the algorithm for the above input.	K3	CO3
4	List and explain the different pipeline hazards and their possible solutions	K5	CO3
5	Design a combinational circuit for 3x2 multiplication.	K5	CO3
6	Draw the flowchart for Booth's Multiplication Algorithm	K5	CO3
7	Design a combinational circuit for 3x2 multiplication.	K3	CO3
8	Explain restoring method of Division	K4	CO3
9	List and explain the different pipeline hazards and their possible solutions	K4	CO3
10	Design 2x3 multiplier	K4	CO3
11	Explain restoring method of division.	K4	CO3
12	Describe basic shift operations	K3	CO3

Module IV

Qn No	Question	Knowledge Level	CO
1	List and explain the different pipeline hazards and their possible solutions	K3	CO3
2	Give the logic used behind Booth's multiplication algorithm.	K2	CO3
3	Identify the appropriate algorithm available inside the system to perform the multiplication between -14 and -9. Also trace the algorithm for the above input.	K3	CO3
4	List and explain the different pipeline hazards and their possible solutions	K5	CO3
5	Design a combinational circuit for 3x2 multiplication.	K5	CO3
6	Draw the flowchart for Booth's Multiplication Algorithm	K5	CO3
7	Design a combinational circuit for 3x2 multiplication.	K3	CO3
8	Explain restoring method of Division	K4	CO3
9	List and explain the different pipeline hazards and their possible solutions	K4	CO3
10	Design 2x3 multiplier	K4	CO3
11	Explain restoring method of division.	K4	CO3
12	Describe basic shift operations	K3	CO3

Module V

Qn No	Question	Knowledge Level	CO
1	List and explain the different pipeline hazards and their possible solutions	K3	CO3
2	Give the logic used behind Booth's multiplication algorithm.	K2	CO3
3	Identify the appropriate algorithm available inside the system to perform the multiplication between -14 and -9. Also trace the algorithm for the above input.	K3	CO3
4	List and explain the different pipeline hazards and their possible solutions	K5	CO3
5	Design a combinational circuit for 3x2 multiplication.	K5	CO3
6	Draw the flowchart for Booth's Multiplication Algorithm	K5	CO3
7	Design a combinational circuit for 3x2 multiplication.	K3	CO3
8	Explain restoring method of Division	K4	CO3
9	List and explain the different pipeline hazards and their possible solutions	K4	CO3
10	Design 2x3 multiplier	K4	CO3
11	Explain restoring method of division.	K4	CO3
12	Describe basic shift operations	K3	CO3

Module I

→ Basic Structure of Computers
* Functional Units
* Basic Operational Concepts
* Bus Structures
* Memory Locations & Memory Operations
* Instructions & Sequencing
* Addressing Modes

→ Basic Processing Unit
* Fundamental Concepts
* Instruction Cycle
* Exⁿ of Complete Instⁿ
* Single & Multiple Bus Organization.

Basic Structure of Computers

→ FUNCTIONAL UNITS

A computer consists of 5 finally independant main parts:

1. Input Unit
2. Memory Unit
3. ALU
4. Output Unit
5. Control Unit.

The input unit accepts coded information from human operators, from electromechanical devices such as key boards or from other computers. The information recieved is either stored in the computer's memory for later experience (reference) or immediately used by ALU to perform desired operations. The processing steps are determined by a pgm stored in the memory. Finally the results are sent back to the outside world through the unit. Fig 1.1 shows the functional units.

INPUT UNIT:

Computers accept coded inf^n through input units, which read the data. The most well-known i/p device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code & transmitted over a cable to either memory or the processor.

Eg: Joysticks, trackball & mouse Microphones

MEMORY UNIT

→ The function of memory unit is to store programs & data. There are 2 classes of storage called primary & secondary.

Primary storage is a fast memory that operates at electronic speeds. Programs must be stored in the memory while they are being executed. The memory contains a large no. of semiconductor storage cells, each capable of storing one bit of information. The memory is organized so that the contents of one word, containing n bits, can be stored or retrieved in one basic operation.

To provide easy access to any ~~location~~ word in the memory, a distinct address is associated with each word location. Addresses are no.s that identify successive locations.

The no. of bits in each word is often referred to as the word length of the computer. Typical wordlengths range from 16 to 64 bits.

Programs must reside in the memory during execution. Instructions & data can be written into the memory or read out under the ctrl of the processor. Memory in which any location can be reached in a short & fixed amount of time after specifying its address is called random-access Memory (RAM). The time used to access one word is called access time

ARITHMETIC and LOGIC UNIT

Most computer op^ns are executed in the arithmetic and logic unit (ALU) of the processor. Consider a typical example: Suppose two no.s located in the memory are to be added. They are brought into the processor, & the actual addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.

When operands are brought into the processor, they are stored in high-speed storage elmts called registers. Each register can store one word of data. Access times to registers are faster than access times to the fastest cache unit in the memory hierarchy.

The control & ALU are many times faster than other devices connected to a computer system.

OUTPUT UNIT

Its function is to send processed results to the outside world. Eg: printer, Graphic units.

CONTROL UNIT

The memory, arithmetic & logic and I/P & O/P units store & process inf^n and perform input & o/p operations. The op^n of these units must be coordinated by the control unit.

The op^n of a computer can be summarized as follows:

The computer accepts inf^n & data through an i/p unit and stored it in the memory.

→ Information stored in the memory is fetched, under pgm ctl, into an ALU, where it is processed.
→ Processed info leaves the computer through an o/p unit.
→ All activities inside the m/c are directed by the ctl unit.

BASIC Operational Concepts.

Activity in a computer is governed by instructions. To perform a given task, an appropriate pgm consisting of a list of inst^{n}s is stored in the memory. Individual instructions are brought from memory into the processor, which executes the specified ops. Data to be used as operands are also stored in the memory.

Eg: Add LOCA, R0

This inst^{n} adds the operand at mem. loc^{n} LOCA to the operand in a reg. in the processor, R0 & places the sum into reg. R0. First, the inst^{n} is fetched from the memory into the processor. Next, the operand at LOCA is fetched and added to the conts. of R0. Finally, the resulting sum is stored in reg. R0.

The preceding Add inst^{n} combines a memory access op^{n} with an ALU op^{n}. The effect of the above inst^{n} can be realized by the 2 inst^{n} sequence

Load LOCA, R1
Add R1, R0

The first of these inst^{n}s transfers the contents of memory location LOCA into processor reg. R1 and the second inst^{n} adds the conts of R1 & R0 & places the sum into R0.

Fig 1.2

The processor contains ALU, ctrl ckt & a no. of registers.

Instruction Register (IR) – holds the instⁿ that is currently being executed.

Programs Counter (PC) – It keeps track of the exⁿ of a pgm. It contains the memory address of the next instⁿ to be fetched & executed. During the exⁿ of an instⁿ, the conts. of the PC are updated to the address of the next instⁿ to be executed.

General Purpose Registers: R_0 through R_{n-1}

Memory Address Registers: holds the address of the location to be accessed.

Memory Data Register: contains the data to be written into or read out of the addressed location.

In addition to transferring data b/w the memory and the processor, the computer accepts data from input devices & sends data to output devices.

Interrupt Service Routine: Normal exⁿ of pgms may be preempted if some device requires urgent servicing. For eg:, a monitoring device in a computer-controlled industrial process may detect a dangerous situation immediately, the normal exⁿ of the current pgm must be interrupted. To do this, the device raises an interrupt s/l. An interrupt is a request from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate Interrupt Service Routine. The conts. of the PC, the general registers &

some control inf^{n} are stored in memory.

BUS STRUCTURES

To achieve a reasonable speed of op^{n}, a computer must be organized so that all its cmts can handle one full word of data at a given time. When a word of data is transferred b/w cmts, all its bits are transferred in parallel, i.e; the bits are transferred simultaneously over many wires or lines, one bit per line. A group of lines that serves as a connecting path for several devices is called a bus.

Fig 1.3. All cmts are connected to this bus. B/c the bus can be used for only one transfer at a time, only 2 cmts can actively use the bus at any given time. Systems that contain multiple buses achieve more concurrency in op^{n}s by allowing 2 or more transfers to be carried out at the same time.

Buffer Registers: to hold the inf^{n} during transfers.

Eg: Printer.

MEMORY LOCATIONS & ADDRESSING

Number, character operands & instructions are stored in memory. The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. B/c a single bit represents a very small amount of information, bits are

seldom handled individually. The memory is organized so that a group of n bits can be stored and retrieved in a single basic op^n. Each group of n bits is referred to as a word of info, & n is called the word length. Fig 2.5.

Accessing the memory to store or retrieve a single item of info, either a word or a byte requires distinct names or addresses for each item location. (Numbers from 0 through $2^k - 1$). k - suitable value. The 2^k addresses constitute the address space of a computer.

→ Byte Addressability

- 3 basic info quantities - bit, byte & word.
  (byte: 8 bits; word: 16 - 64 bits)

It is impractical to assign distinct addresses to individual bit locations in the memory.

- to have successive addresses refer to successive byte locations in the memory.

Byte addressable Memory: Byte locations have addresses 0, 1, 2 ... Thus, if the word length of the m/c is 32 bits, successive words are located at addresses 0, 4, 8 ... with each word consisting of 4 bytes.

Big-Endian & Little-Endian Assignments.

Two ways that byte addresses can be assigned across words. Fig 2.7.

The name big-endian is used when lower byte addresses are used for more significant bytes (leftmost bytes) of the word.

The name little-endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (rightmost bytes) of the word.

Fig : 2.7.

Words are said to be alined in memory if they begin at a byte address that is a multiple of the no. of bytes in a word.

Accessing Numbers, Characters & Character Strings.

→ A number usually occupies one word. It can be accessed in the memory by specifying its word address.

→ Individual characters can be accessed by their byte address. The beginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive characters of string. 2 ways to indicate the length of the string.

- end of string
- separate memory word location.

MEMORY OPERATIONS

Both pgm instns & data operands are stored in the memory. Operands & results must be moved b/w memory & the processor; instn to be transferred from the memory to the processor.

Two basic ops involving the memory are Load (or Read or Fetch) and Store (write).

The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a load operation, the processor sends the address of the desired location to the memory & requests that its contents be read. The memory reads the data stored at that address. & sends them to the processor.

The Store operation transfers an item of info from the processor to a specific mem. location, destroying the former conts. of that location. The processor sends the address of the desired location to the memory, together with the data to be written into that location.

INSTRUCTIONS & INSTRUCTION SEQUENCING

A computer must have instructions capable of performing 4 types of operations:

- → Data transfers b/w memory & the processor regs.
- → Arithmetic & logic ops on data
- → Pgm sequencing & ctrl
- → I/O transfers.

* Register Transfer Notation:

Transfers b/w – memory locations, processor registers, or registers in the I/O subsystem.
– Identify the location by a symbolic name for its hardware binary address.

For eg: names of the addresses of memory locations may be LOC, PLACE A, VAR2

The conts. of a location are denoted by placing square brackets around the name of the loc^n.

$R_1 \leftarrow [LOC]$

means that the conts. of memory location LOC are transferred into processor register R_1.

Eg: $R_3 \leftarrow [R_1] + [R_2]$

This type of notation is known as Register Transfer Notation (RTN).

* Assembly Language Notation

Eg: Move LOC, R_1

The conts. of LOC are unchanged by the exc^n of this inst^n, but the old conts. of register R_1 are overwritten.

Eg: Add R_1, R_2, R_3.

* Basic Instruction Types:

High level lang. instⁿ

C = A+B.

This stmt requires the action

C ← [A] + [B]

Single m/c instⁿ

Add A, B, C // 3 address Instⁿ

A & B → source operands

C → destination operand.

Add → operation to be performed.

Operation Source1, Source2, Destination

2 Address Instⁿ: Add A, B

B ← [A] + [B]

Move B, C

C ← [B]

1 Address Instruction:

Add A

→ Add conts. of mem. Location A to the the conts. of the accumulator reg. & place the sum back into the accumulator.

Load A, Store A.

↓ Copies A into accumulator

↓ Store accumulator into A.

Load A, R_i

R_i → General purpose registers.

Eg: Add R_i, R_j

* Instruction Execution & Straight-Line Sequencing:

Pgm: C ← [A] + [B]

Execution: The processor contains a reg. called program counter (PC), which holds the address of the inst^n to be executed next. To begin executing a pgm, the address of its inst^n (i in the eg) must be placed into the PC. Then the processor ctrl ckts use the inf^n in the PC to fetch & execute inst^{n}s, one at a time, in the order of increasing addresses. This is called straight-line sequencing. During the ex^n of each inst^n, the PC is incremented by 4 to point to point to the next inst^n. Thus after the Move inst^n at location i+8 is executed, the PC contains the value i+12, which is the address of the 1st inst^n of the next pgm segment.

Ex: Its a 2 phase procedure.

In the Instruction fetch: the inst^n is fetched from the memory location whose address is in PC. This inst^n is placed in the inst^n reg. IR. in the processor.

Instruction execute: the inst^n in the IR is examined to determine which op^n is to be

performed. The specified op^n is then performed by the processor. Ref. fig 2.

*Branching:

Eg: Adding a list of n numbers.

Instead of using a long list of Add inst^{n}s, it is possible place a single Add inst^n in a pgm loop. The loop is a straight-line sequence of inst^{n}s executed as many times as needed. It starts at location LOOP & ends at the inst^n Branch>0. During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to R_0.

The inst^n Decrement R_1 reduces the cont. of R_1 by 1 each time through the loop. Ex^n of the loop is repeated as long as the result of the decrmt op^n is greater than zero.

Branch inst^n: loads a new value into the pgm counter. As a result, the processor fetches & executes the inst^n at this new address called the branch target, instead of the inst^n at the loc^n that follows the branch inst^n in sequential address order.

Branch>0 LOOP

branch to location LOOP if the result of immediately preceding inst^n which is decremented value in reg. R_1, is >zero.

* Condition Codes

The processor keeps track of inf^n about the results of various op^{n}s for use by subsequent conditional branch inst^{n}s. This is accomplished by recording the required inf^n in individual bits, often called condition code flags.

Four commonly used flags are

N (negative) set to 1 if the result is negative otherwise cleared to 0.

Z (zero) set to 1 if the result is 0; otherwise cleared to 0.

V (overflow) set to 1 if arithmetic overflow occurs otherwise cleared to 0.

C (carry) set to 1 if a carry-out results from the op^n; otherwise cleared to 0.

* Generating Memory Addresses:

The purpose of the inst^n block at LOOP is to add a diff. no. from the list during each pass through the loop. Hence, the Add inst^n in that block must refer to a diff. address during each pass. The memory operand addr. cannot be given directly in a single Add Inst^n in the loop.

— Need for flexible ways to specify the addr. of an operand. The inst^n set of a computer typically provides a no. of such methods called addressing modes.

ADDRESSING MODES

The different ways in which the loc^n of an operand is specified in an inst^n are referred to as addressing modes.

* Implementation of Variables & Constants:

— can be accessed by specifying the name of the register or the addr. of mem. location where the operand is located. These 2 modes are:

1. Register Mode: The operand is the conts. of a processor register; the name (address) of reg. is given in the inst^n.

2. Absolute mode: The operand is in a mem. location; the addr. of this location is given explicitly in the inst^n.

The inst^n Move LOC, R2 uses these 2 modes.

Exp: Integer A, B; — allocate a mem. location to each of the variables A & B. (Absolute Mode)

3. Immediate Mode: The operand is given explicitly in the inst^n. For eg: Move 200, R0. (immediate)
places the value 200 in reg. R0. — specifies the value of a source operand.

```
Move  B, R1
Add   #6, R1
Move  R1, A
```

* Indirection and Pointers:

The inst^n doesn't give the operand or its address explicitly. It provides inf^n from which the mem. address of the operand can be determined. This is effective address (EA) of the operand.

Indirect mode — The effective address of the operand is the conts. of a reg. or mem. location whose addr. appears in the inst^n.

Indirection — placing the name of the reg. or the mem. address given in the inst^n in parantheses. The reg. or mem. location that contains the addr. of an operand is called a pointer. ~~Fig 2.12~~

* Indexing & Arrays

Index mode: The effective address of the operand is generated by adding a constant value to the conts. of a register. The reg. used may be either special reg. provided for this purpose or it may be any one of a set of general purpose regs. in the processor. It is called index reg.

$X(R_i)$ where X denotes the const. value contained in the inst^n & R_i is the name of the reg. involved. The effective address of the operand is given by $EA = X + [R_i]$.

Fig illustrates 2 ways of using the index mode. ~~The~~ In fig (a), the index reg, R_i contains the address of a mem loc^n & the value X defines an offset from this address to the location where the operand is found.

In fig. b, the constant X corresponds to a mem. address, the conts. of the index reg. defines the offset to the operand; the EA is sum of 2 values; one is given explicitly in the inst^n & the other is stored in a register.

* Relative Addressing.

The effective Address is determined by the Index mode using the PC in place of the general purpose register R_i.

eg: Branch>0 LOOP causes pgm exc^n to go to the branch target loc^n identified by the name LOOP if the Branch condition is satisfied. The location can be computed by specifying it as an offset from current value of the pgm counter.

* Auto increment Mode:

The effective address of the operand is the conts. of a reg. specified in the inst^n. After accessing the operand, the conts. of this reg. are automatically incremented to point to the next item in a list.

BASIC PROCESSING UNIT

The unit which executes m/c instns & coordinates the activities of other units, is called Instn set Processor (ISP).

Fundamental Concepts:

→ Program Counter
→ Instn Register

To execute an instn, the processor has to perform the follo. 3 steps.

1. Fetch the contents of the memory location pointed to by the PC. The contents of this locn are interpreted as an instn to be executed. Hence, they are loaded into the IR. Symbolically, this can be written as

$$IR \leftarrow [PC]$$

2. Assuming that the mem. is byte addressable, increment the contents of the PC by 4

that is $PC \leftarrow [PC] + 4$

3. Carry out the actions specified by the instn the IR.

In cases, where an instn occupies more than one word, steps 1 & 2 must be repeated as many times as necessary to fetch the complete instn. These 2 steps are usually referred to as the fetch phase; step 3 constitutes the exn phase.

Fig. shows an orgn in which the arithmetic & logic Unit (ALU) and all the Regs. are interconnected via a single common bus. This bus is internal to the processor.

Internal processor Bus: Data & Address lines connected to the internal processor bus through MAR & MDR.
Memory Data Reg. – 2 i/ps & 2 o/ps. Data may be loaded into MDR either from the mem. bus or from internal processor bus.
Memory Address Reg – i/p is connected to internal bus & its o/p is connected to o/p external bus.
The ctrl lines of the memory bus are connected to instⁿ decoder & ctrl logic Block.
The no. & use of the processor regs. R0 through $R_{(n-1)}$ vary from one processor to another.
The MUX (multiplexer) selects either the o/p of reg. Y or a const. value 4 to be provided as i/p A of the ALU. The const. 4 is used to increment the conts. of the pgm Counter.
ALU: As instⁿ exᵗ progresses, data are transferred from one reg. to another often passing through the ALU to perform some arithmetic or logic opⁿ. The instⁿ decoder & ctrl logic unit is responsible for implementing the actions specified by the instⁿ loaded in IR register. The decoder generates the ctrl s/ls needed to select the regs. involved & direct the transfer of data.

Instⁿ execution:

* Transfer a word of data from one processor reg. to another or to the ALU.
* Perform an arithmetic or a logic opⁿ & store the result in a processor reg.
* Fetch the conts. of a given mem. locⁿ & load them into processor reg.
* Store a word of data from a processor reg. into a given mem. locⁿ.

1. Register Transfers

Instⁿ exⁿ involves a sequence of steps in which data are transferred from one reg. to another. For each reg, 2 ctrl s/ls are used to place the conts. of that reg. on the bus & load the data on the bus into the reg.

Suppose that we wish to transfer the conts. of reg. R_1 to reg. R_4. This can be accomplished as

→ Enable the o/p of reg. R_1 by setting R_{1out} to 1. This places the conts. of R_1 on the processor bus.

→ Enable the i/p of reg. R_4 by setting R_{4in} to 1. This loads data from the processor bus into reg. R_4.

2. Performing an Arithmetic or Logic Op's.

The ALU is a combinational ckt that has no internal storage. It performs arithmetic & logic opⁿs on the 2 operands applied to its A & B i/ps.

Fig:

One of the operands is the o/p of the multiplexer MUX & the other operand is obtained directly from the bus. The result produced by the ALU is stored temporarily in register Z.

3. Fetching a word from Memory:

To fetch a word of infⁿ from memory, the processor has to specify the address of the memory location where this infⁿ is stored & request a Read opⁿ.

4. Storing a Word in Memory.

Writing word into a memory locⁿ follows a similar procedure. The desired address is loaded

WMFC - Word Mem-fn. completed

into MAR. Then, the data to be written are loaded into MDR. & a Write command is issued.

1. $R1_{out}$, MAR_{in}
2. $R2_{out}$, MDR_{in}, Write
3. MDR_{outE},

EXECUTING OF A COMPLETE INSTRUCTION

Eg: Add (R_3), R_1 ⇒ adds the conts. of a mem. location pointed to by R_3 to reg. R_1. Executing this inst^n requires the follo. actions:

1. Fetch the inst^n
2. Fetch the 1st operand
3. Perform the addition
4. Load the result into R_1.

Step	Action	
1	PC_{out}, MAR_{in}, Read, select4, Add, Z_{in}	The inst^n fetch op^n is initiated by loading the conts. of the PC into MAR & sending a read req. to memory. The select s/l is set to select4, which causes the multiplexer MUX to select the const. 4. This value is added to the operand at i/p B which is the conts. of PC & result is stored in Z.
2.	Z_{out}, PC_{in}, Y_{in}, WMFC	The updated value is moved from reg. Z back into the PC, during step 2, while waiting for the mem. to respond.

3	MDR_{out}, IR in	The word fetched from the memory is loaded into the IR.
4.	R_3out, MAR_{in}, Read	The inst^n decoding ckt interprets the conts. of the IR at the begining of step 4. This enables the ctrl ckt'ry to activate the ctrl s/ls for steps 4 through 7, which constitute the exc^n phase.
5	R_1out, Yin, WMFC	The conts. of reg - R_3 are transferred to the MAR in step 4, & a memory read op^n is initiated. Then the conts. of R_1 are transferred to reg. Y in step 5.
6.	MDR_{out}, SelectY, Add, Zin	When the Read op^n is

Module II

Register transfer logic: inter register transfer – arithmetic, logic and shift micro operations.

Processor logic design: - processor organization – Arithmetic logic unit - design of arithmetic circuit - design of logic circuit - Design of arithmetic logic unit - status register – design of shifter - processor unit – design of accumulator.

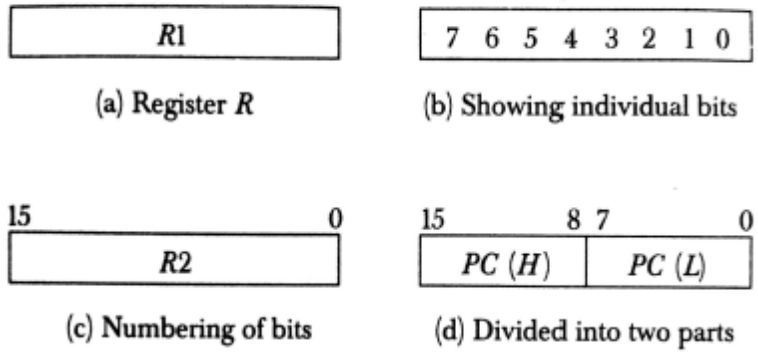
Register transfer logic

A digital system is an interconnection of digital hardware modules. The modules are registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system. Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called *microoperations*. A *microoperation* 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 be transferred to another register. Examples of microoperations are shift, count, clear, and load.

The symbolic notation used to describe the micro-operation transfer among registers is called RTL (Register Transfer Language). The use of ***symbols*** instead of a ***narrative explanation*** provides an organized and concise manner for listing the micro-operation sequences in registers and the control functions that initiate them.

Registers: Computer registers are designated by upper case letters (and optionally followed by digits or 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 and is designated by the name **MAR**. Other designations for registers are ***PC*** (for program counter), ***IR*** (for instruction register, and ***R1*** (for processor register). The individual flip-flops in an n-bit register are numbered in sequence from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 4-1 shows the representation of registers in block diagram form.

Figure 4-1 Block diagram of register.



The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 4-1(a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol *H* (for high byte). The name of the 16-bit register is *PC*. The symbol *PC (0-7)* or *PC (L)* refers to the low-order byte and PC (8-15) or *PC (H)* to the high-order byte**.**

Register Transfer:

Information transfer from one register to another is designated in symbolic form by means of a *replacement operator*. The statement **R2← R1** denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R 1 does not change after the transfer. If we want the transfer to occur only under a predetermined control condition then it can be shown by an if-then statement.

if (P=1) then R2← R1

P is the control signal generated by a control section. We can separate the control variables from the register transfer operation by specifying a ***Control Function.*** Control function is a Boolean variable that is equal to 0 or 1. control function is included in the statement as **P: R2← R1** Control condition is terminated by a colon implies transfer operation be executed by the

hardware only if P=1. Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 4-2 shows the block diagram that depicts the transfer from R1 to R2.

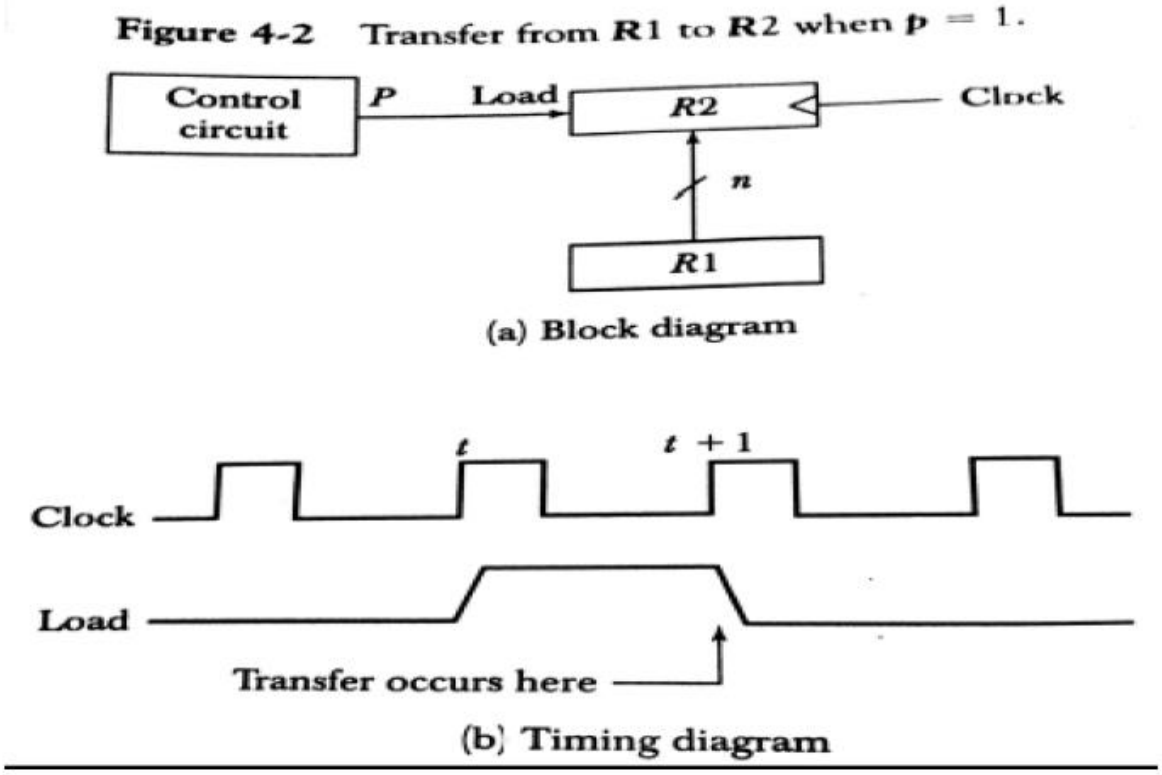


Figure 4-2 Transfer from R1 to R2 when $p = 1$.

The n outputs of register R1 are connected to the n inputs of register R2. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input that is activated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one

applied to the register. As shown in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time *t*. The next positive transition of the clock at time *t* + 1 finds the load input active and the data inputs of R2 are then loaded into the register in parallel.

Types of Micro-operations:

- *Register Transfer Micro-operations:* Transfer binary information from one register to another.
- *Arithmetic Micro-operations:* Perform arithmetic operation on numeric data stored in registers.
- *Logical Micro-operations:* Perform bit manipulation operations on data stored in registers.
- *Shift Micro-operations:* Perform shift operations on data stored in registers.
- Register Transfer Micro-operation doesn't change the information content when the binary information moves from source register to destination register.

Arithmetic Micro-operations:

The basic arithmetic micro-operations are

- Addition
- Subtraction
- Increment
- Decrement
- Shift

The arithmetic Micro-operation defined by the statement below specifies the add micro-operation.

R3 ← R1 + R2 . It states that the contents of R1 are added to contents of R2 and sum is transferred to R3. To implement this statement hardware requires 3 registers and digital component that performs addition.Subtraction is most often implemented through complementation and addition.

The subtract operation is specified by the following statement

R3 ← R1 + R2 + 1

- instead of minus operator, we can write as
- R2 is the symbol for the 1's complement of R2
- Adding 1 to 1's complement produces 2's complement
- Adding the contents of *R1* to the 2's complement of R2 is equivalent to R1-R2.

Binary Adder:

Digital circuit that forms the arithmetic sum of 2 bits and the previous carry is called **FULL ADDER.**

Digital circuit that generates the arithmetic sum of 2 binary numbers of any lengths is called **BINARY ADDER.** Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder.

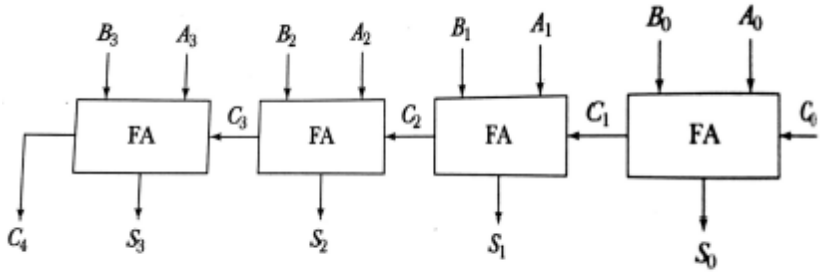


Figure 4-6 4-bit binary adder.

The augends bits of A and the addend bits of *B* are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is Co and the output carry is C4. The *S* outputs of the full-adders generate the required sum bits. An n-bit binary adder requires n full-adders.

Binary Adder – Subtractor:

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 4-7.

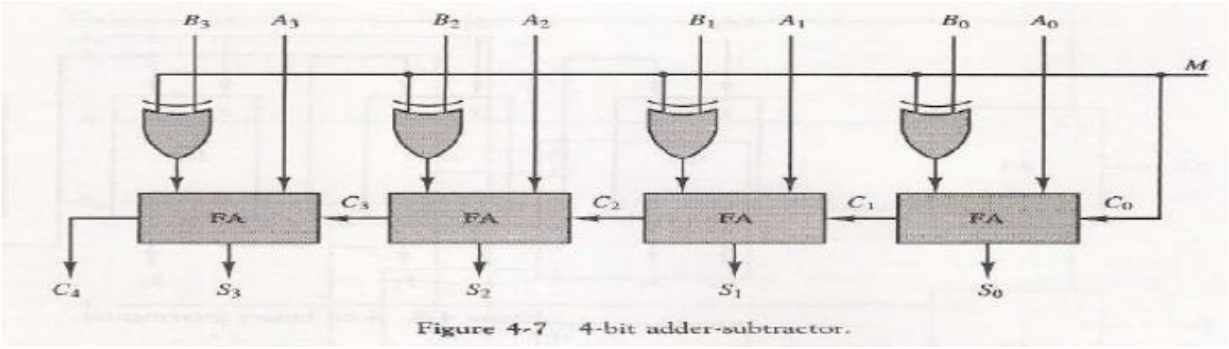


Figure 4-7 4-bit adder-subtractor.

The mode input M controls the operation. When M = 0 the circuit is an adder and when M = 1 the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B .

When M = 0, we have *B* xor 0 = B. The full-adders receive the value of *B*, the input carry is 0, and the circuit performs A plus B. When M = 1, we have *B xor 1 = B'* and Co = 1. The *B* inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of *B*.

Binary Incrementer:

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This can be accomplished by means of half-adders connected in cascade. The diagram of a 4-bit 'combinational circuit incrementer is shown in Fig. 4-8.

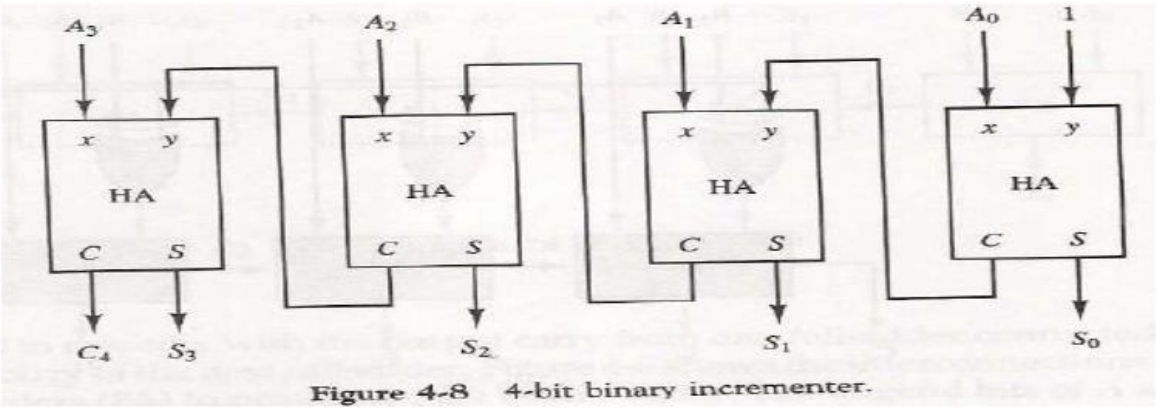


Figure 4-8 4-bit binary incrementer.

One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A0 through A3, adds one to it, and generates the incremented output in S0 through S3. The output carry *C4* will be 1 only after incrementing binary 1111. This also causes outputs S0 through S3 to go to 0.

The circuit of Fig. 4-8 can be extended to an *n* -bit binary incrementer by extending the diagram to include *n* half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

Logic Micro-operations:

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers RI and R2 is symbolized by the statement

$$P: \quad R1 \leftarrow R1 \oplus R2$$

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1.

List of Logic Microoperations:

 There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 4-5.

TABLE 4-5 Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 4-6. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B. The logic micro-operations listed in the second column represent a relationship between the binary content of two registers A and B.

Shift Microoperations:

Shift microoperations are used for serial transfer of data. The contents of a register can be shifted to the left or the right. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. There are three types of shifts: logical, circular, and arithmetic. The symbolic notation for the shift microoperations is shown in Table 4-7.

TABLE 4-7 Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

Logical Shift: A *logical* shift is one that transfers 0 through the serial input. The symbols *shl* and shr for logical shift-left and shift-right microoperations. The microoperations that specify a 1-bit shift to the left of the content of register R and a 1-bit shift to the right of the content of register R shown in table 4.7. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

Circular Shift: The *circular* shift (also known as a *rotate* operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols *cil* and *cir* for the circular shift left and right, respectively.

Arithmetic Shift: An *arithmetic shift* is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2.

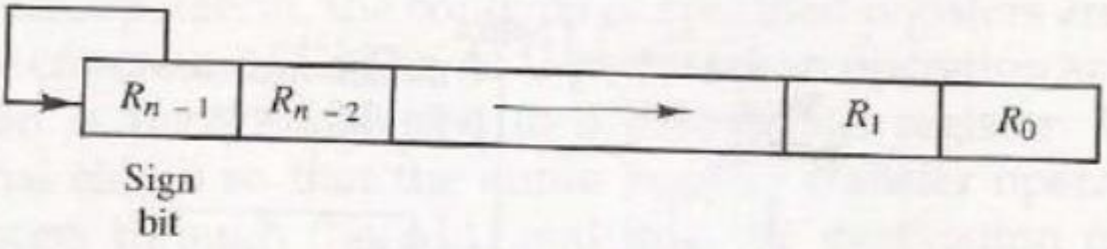


Figure 4-11 Arithmetic shift right.

Part 2

Processor logic design: - processor organization – Arithmetic logic unit - design of arithmetic circuit - design of logic circuit - Design of arithmetic logic unit - status register – design of shifter - processor unit – design of accumulator.

Processor organization

To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:

• **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).

• **Interpret instruction:** The instruction is decoded to determine what action is required.

• **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.

• **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.

• **Write data:** The results of an execution may require writing data to memory or an I/O module.

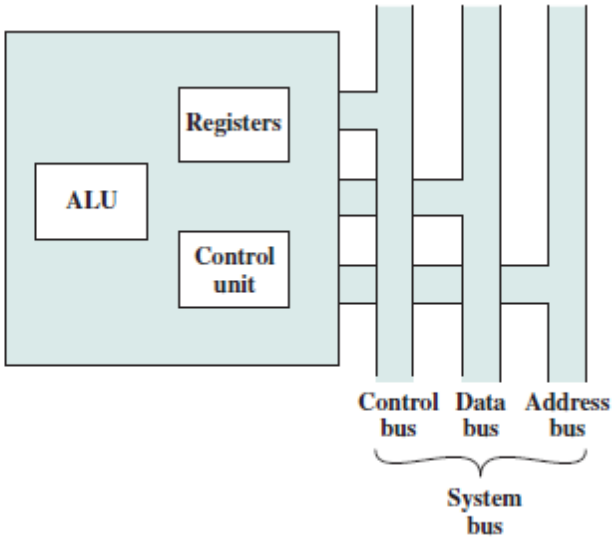


Figure 14.1 The CPU with the System Bus

Figure 14.1 is a simplified view of a processor, indicating its connection to the rest of the system via the system bus. The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers*. The data transfer and logic control paths are indicated, including an element labeled *internal processor bus*. This element is needed to transfer data

between the various registers and the ALU because the ALU in fact operates only on data in the internal processor memory. The figure also shows typical basic elements of the ALU. Note the similarity between the internal structure of the computer as a whole and the internal structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers) connected by data paths.

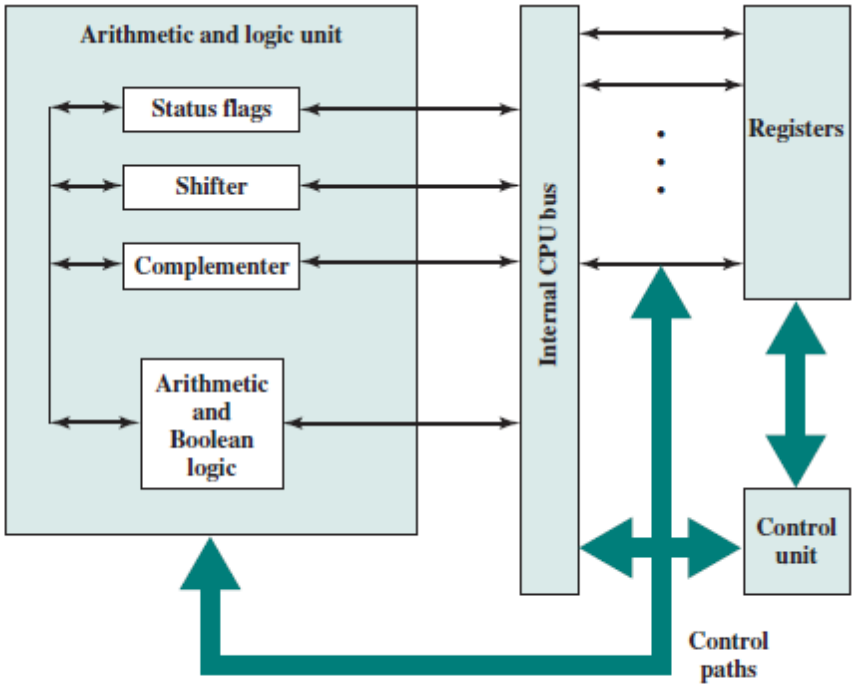


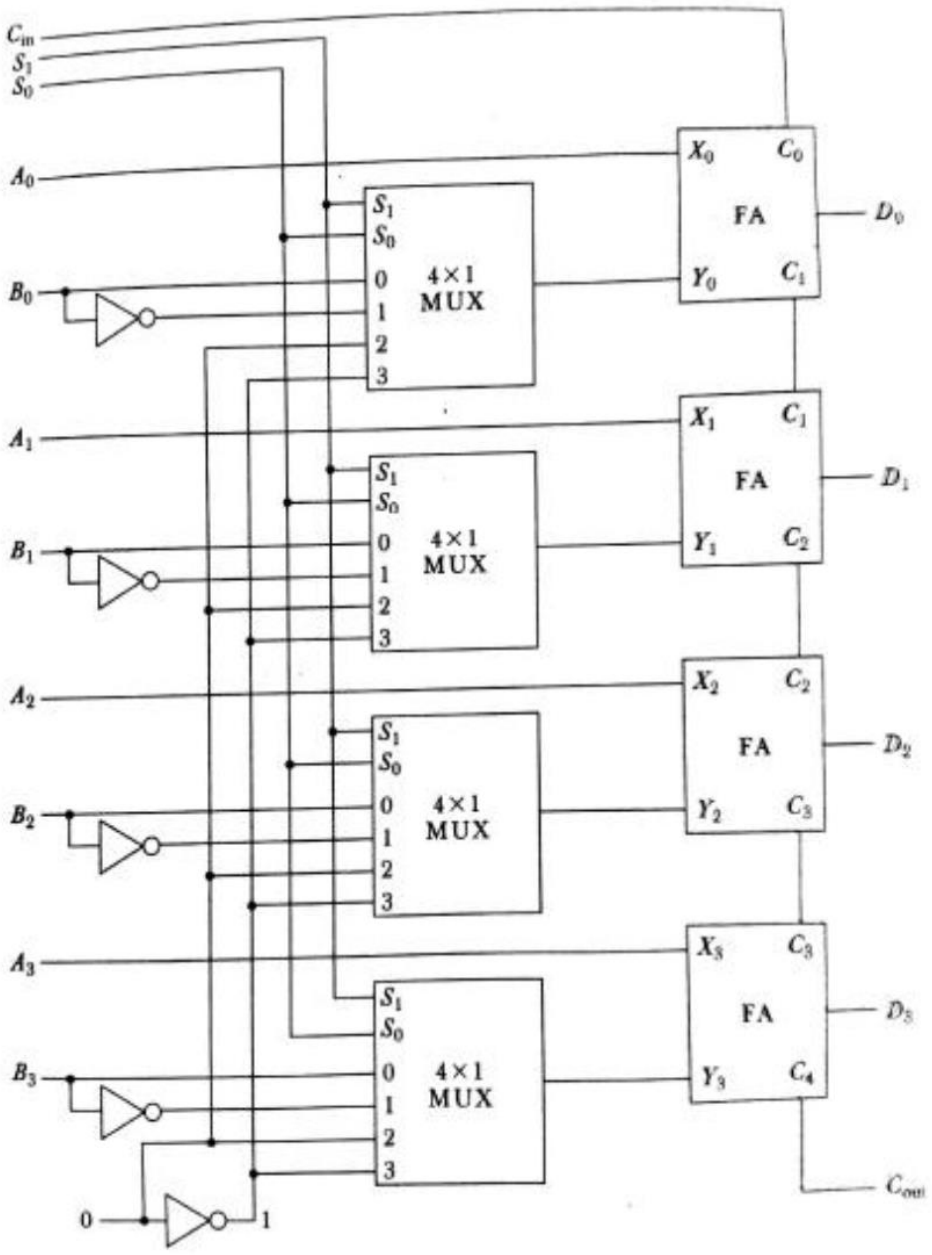
Figure 14.2 Internal Structure of the CPU

Design of Arithmetic Circuit

The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations. The diagram of a 4-bit arithmetic circuit is shown in Fig. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B are connected to the data inputs of the multiplexers. The multiplexers data inputs also receive the complement of B .

The other two data inputs are connected to logic-0 and logic-1 . Logic-0 i s a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through· an inverter

whose input is 0. The four multiplexers are controlled by two selection inputs, S1 and S0• The input carry C'" goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.



The output of the binary adder is calculated from the following arithmetic sum:

D = A + Y + C'"

where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. C'" is the input carry, which can be equal to 0 or 1. Note that the

symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S1 and So and making C'" equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed Table.

TABLE 4-4 Arithmetic Circuit Function Table

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\bar{B}$	$D = A + \bar{B}$	Subtract with borrow
0	1	1	$\bar{B}$	$D = A + \bar{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

When S1S0 = 00, the value of B is applied to the Y inputs of the adder. If Cin = O, the output D = A + B . If Cin = 1, output D = A + B + l. Both cases perform the add microoperation with or without adding the input carry.

When S1S0 = 01, the complement of B is applied to the Y inputs of the adder. If Cin = 1, then D = A + B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtraction of A - B. When Cm = 0, then D = A + B . This is equivalent to a subtract with borrow, that is, A - B - 1.

When S1S0 = 10, the inputs from B are neglected, and instead, all O's are inserted into the Y inputs. The output becomes D = A + 0 + Cm· This gives D = A when Cm = 0 and D = A + 1 when Cin = 1. In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

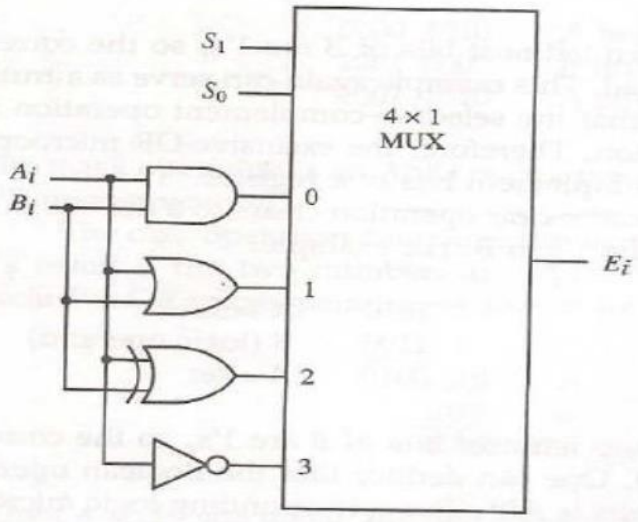
When S1So = 11, all 1' s are inserted into the Y inputs of the adder to produce the decrement operation D = A - 1 when Cm = 0. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces F = A + 2's complement of 1 = A - 1. When Cin = 1, then D = A - 1 + 1 = A, which causes a direct transfer from input A to output D. Note that the microoperation D = A is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

Design of Logic Circuit

The hardware implementation of logic rnicrooperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic rnicrooperations, most computers use only four-AND, OR, XOR (exclusive-OR), and complement from which all others can be derived. Figure shows one stage of a circuit that generates the four basic logic rnicrooperations . It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic.

The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs 51 and 50 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for i = 0, 1, 2, . . . , n - 1. The selection variables are applied to all stages. The function table in Fig. 4-10(b) lists the logic rnicrooperations obtained for each combination of the selection variables.

Figure 4-10 One stage of logic circuit.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

Some Applications:

Logic micro-operations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits or insert new bits values into a register. The following example shows how the bits of one register (designated

by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B).

- **Selective set**

The *selective-set* operation sets to 1 the bits in register A where there are corresponding l'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

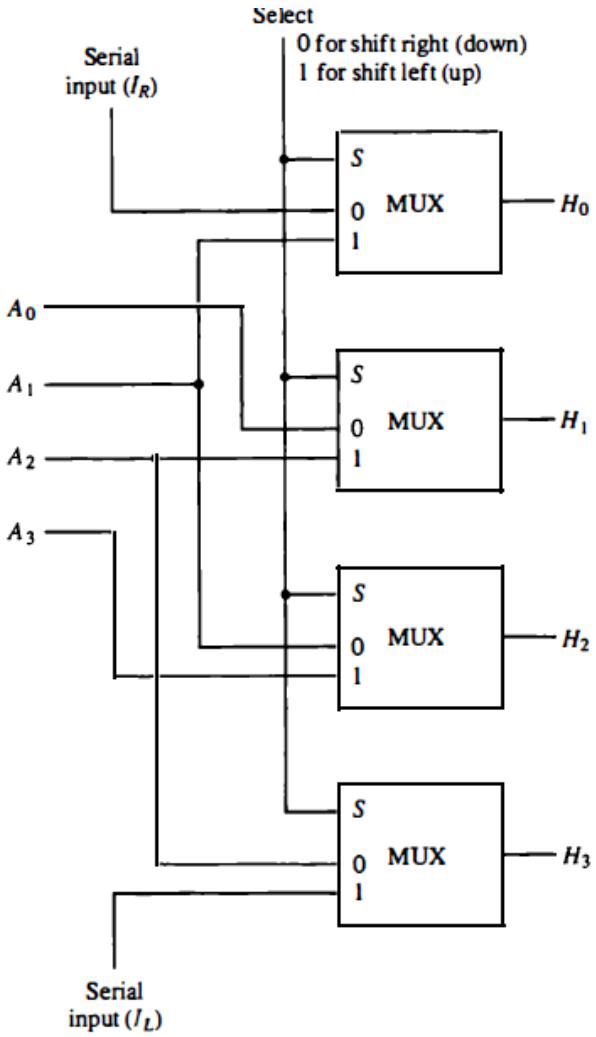
The *selective-complement* operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in *B*. For example:

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

Design of Shifter

A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 4-12. The 4-bit shifter has four data inputs, A0 through A3, and four data outputs, H0 through H3. There are two serial inputs, one for shift left (IL) and the other for shift right (IR). When the selection input *S=0* the input data are shifted right (down in the diagram). When *S* = 1, the input data are shifted left (up in the diagram). The function table in Fig. 4-12 shows which input goes to each output after the shift.

A shifter with n data inputs and outputs requires n multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4-12 4-bit combinational circuit shifter.

Status Registers

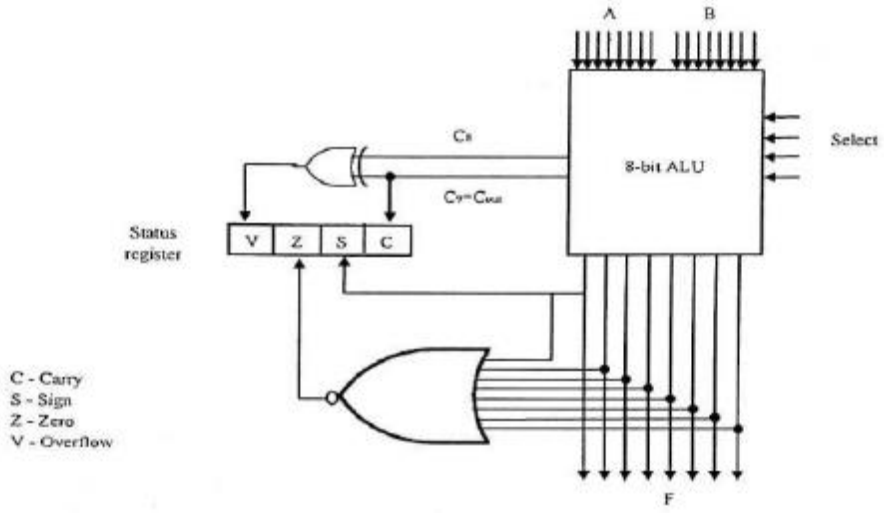
The relative magnitude of two numbers may be determined by subtracting one number from the other and then checking certain bit conditions in the resultant difference. This status bit conditions (often called condition-code bits or flag bits) are stored in a status register.

Status register is a 4 bit register. The four bits are C (carry), Z (zero),S (sign) and V (overflow). These bits are set or cleared as a result of an operation performed in the ALU.

Bit C is set if the output carry of an ALU is 1.

Bit S is set to 1 if the highest order bit of the result in the output of the ALU is 1. Bit Z is set to 1 if the output of the ALU contains all O's.

Bit V is set if the exclusive —OR of carries C8 and C9 is 1, and cleared otherwise. This is the condition for overflow when the numbers are in signed 2's complement representation. For an 8 bit ALU, V is set if the result is greater than 127 or less than -128.



Setting bits in a status register

After an ALU operation, status bits can be checked to determine the relationship that exist between the values of A and B.

If bit V is set after the addition two signed numbers, it indicates an overflow condition. If Z is set after an exclusive OR operation, it indicates that A=B. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit.

Relative magnitudes of A and B can be checked by compare operation. If A-B is performed for two unsigned binary numbers, relative magnitudes of A and B can be determined from the values transferred to the C and Z bits. If Z=1,we knows that A=B, since A-B=0. If Z=0, then we know that A is not equal to B. Similarly C=1 if A>=B and C=0 if A<B. The following table lists the various conditions.

Relation	Condition of Status bits	Boolean function
A>B	C=1 and Z=0	CZ'
A≥B	C=1	C
A<B	C=0	C'
A≤B	C=0 or Z=1	C' + Z
A=B	Z=1	Z
A≠B	Z=0	Z'

Status bits after the subtraction of unsigned numbers (A-B)

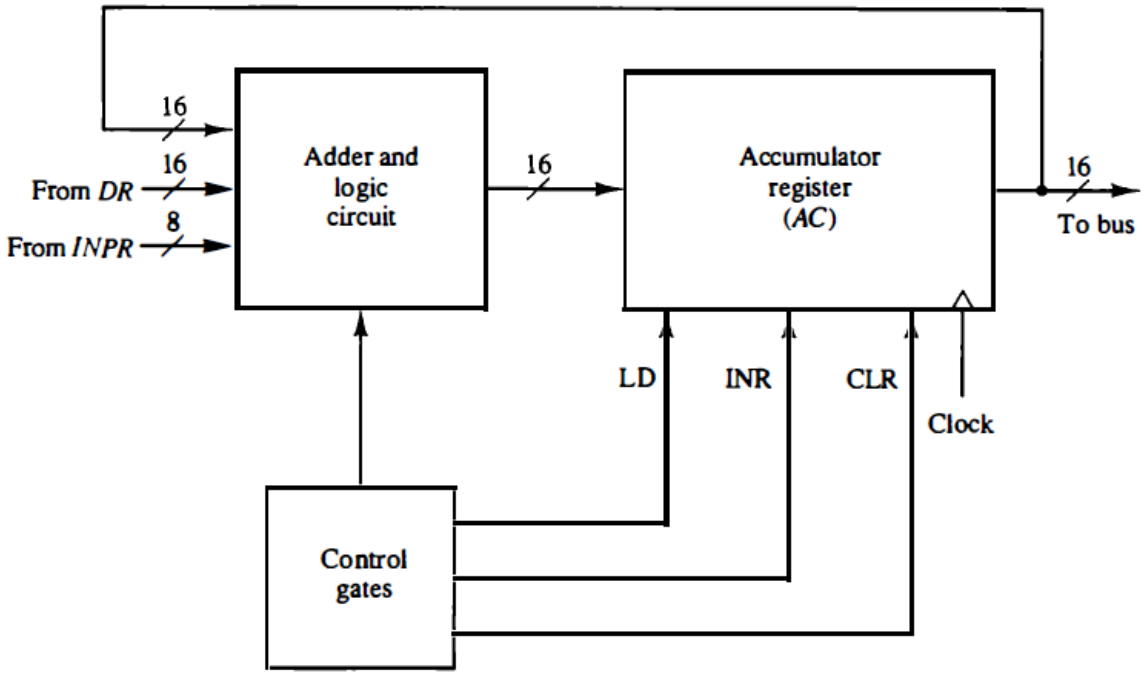
Design of Accumulator Logic

The circuits associated with the AC register are shown in Fig. 5-19. The adder and logic circuit has three sets of inputs. One set of 16 inputs comes from the outputs of AC . Another set of 16 inputs comes from the data register DR . A third set of eight inputs comes from the input register INPR . The outputs of the adder and logic circuit provide the data inputs for the register. In addition, it is necessary to include logic gates for controlling the LD, INR, and CLR in the register and for controlling the operation of the adder and logic circuit. In order to design the logic associated with AC, it is necessary to go over the register transfer statements in Table 5-6 and extract all the statements that change the content of AC .

D_0T_5:	$AC \leftarrow AC \wedge DR$	AND with *DR*
D_1T_5:	$AC \leftarrow AC + DR$	Add with *DR*
D_2T_5:	$AC \leftarrow DR$	Transfer from *DR*
pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$	Transfer from *INPR*
rB_9:	$AC \leftarrow \overline{AC}$	Complement
rB_7:	$AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E$	Shift right
rB_6:	$AC \leftarrow \text{shl } AC,\ AC(0) \leftarrow E$	Shift left
rB_{11}:	$AC \leftarrow 0$	Clear
rB_5:	$AC \leftarrow AC + 1$	Increment

From this list we can derive the control logic gates and the adder and logic circuit.

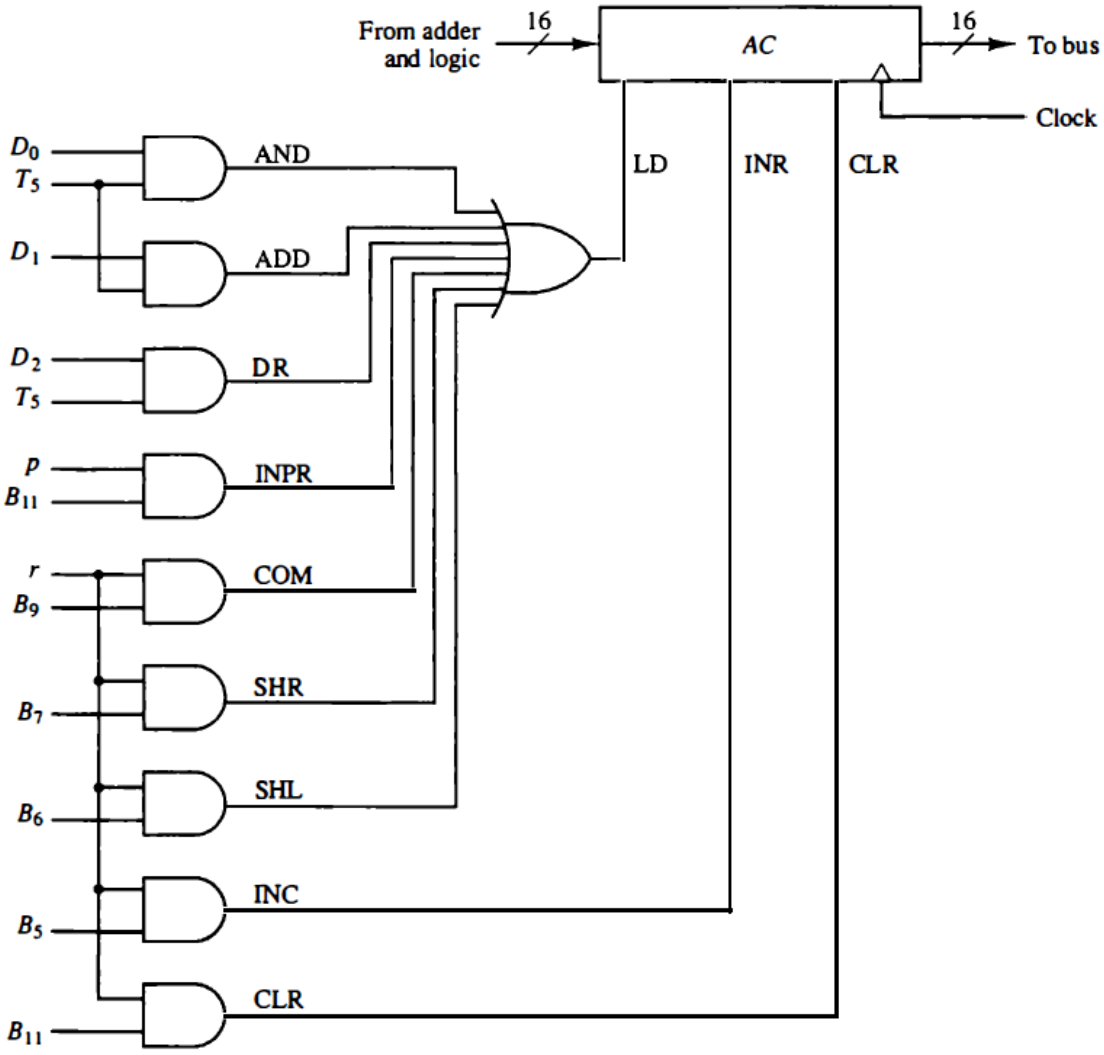
Figure 5-19 Circuits associated with AC.



Control of AC Register

The gate structure that controls the LD, INR, and CLR inputs of AC is shown in Fig. 5-20. The gate configuration is derived from the control functions in the list above. The control function for the clear micro operation is rBn, where r = D7l ' T3 and Bn = IR (ll). The output of the AND gate that generates this control function is connected to the CLR input of the register. Similarly, the output of the gate that implements the increment micro operation is connected to the INR input of the register. The other seven micro operations are generated in the adder and logic circuit and are loaded into AC at the proper time. The outputs of the gates for each control function is marked with a symbolic name. These outputs are used in the design of the adder and logic circuit.

Figure 5-20 Gate structure for controlling the LD, INR, and CLR of AC.

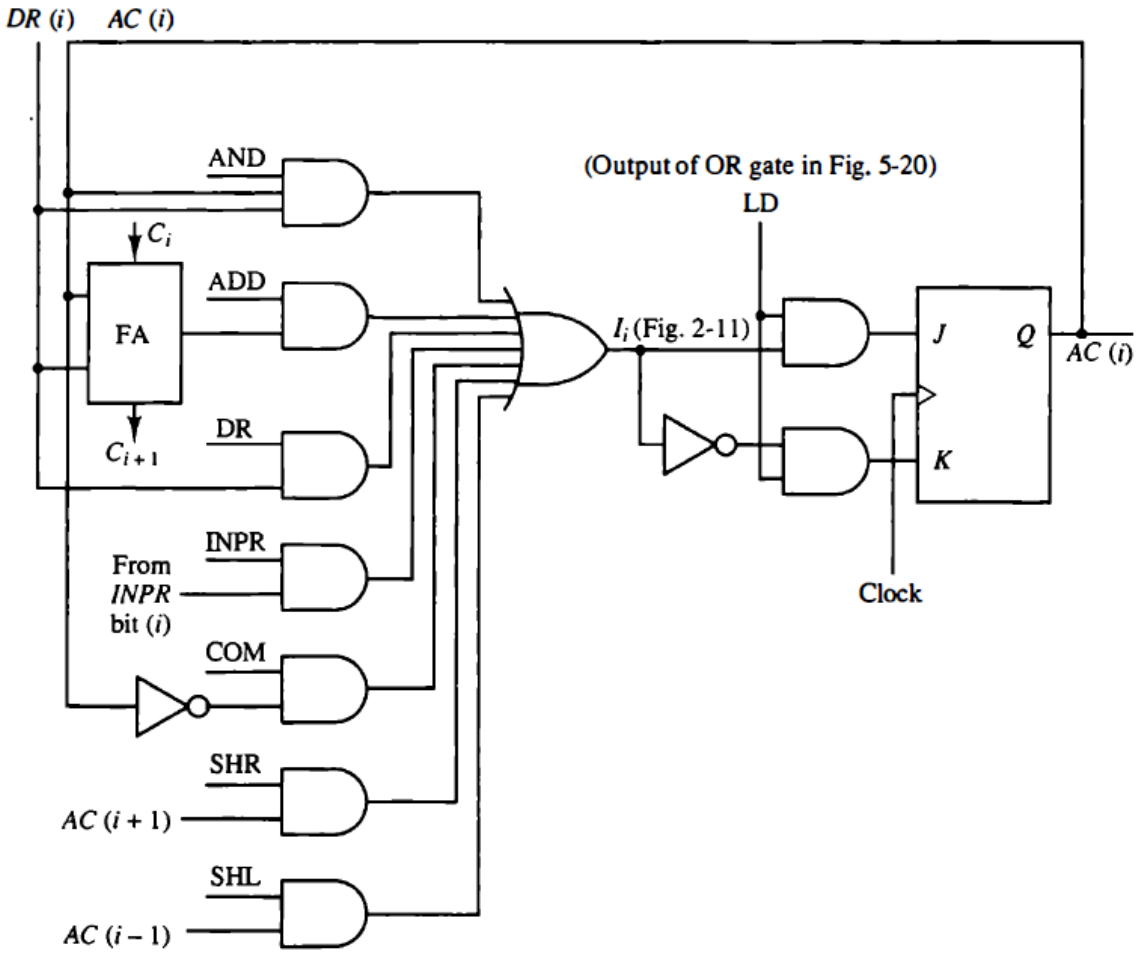


Adder and Logic Circuit

The adder and logic circuit can be subdivided into 16 stages, with each stage corresponding to one bit of AC . The load (LD) input is connected to the inputs of the AND gates. Figure 5-21 shows one such AC register stage (with the OR gates removed). The input is labeled I; and the output AC(i). When the LD input is enabled, the 16 inputs I, for i = 0, 1, 2, . . . , 15 are transferred to AC (0-15).

One stage of the adder and logic circuit consists of seven AND gates, one OR gate and a full-adder (FA), as shown in Fig. 5-21. The inputs of the gates with symbolic names come from the outputs of gates marked with the same symbolic name in Fig. 5-20. For example, the input marked ADD in Fig. 5-21 is connected to the output marked ADD in Fig. 5-20.

Figure 5-21 One stage of adder and logic circuit.



The AND operation is achieved by AND ing AC(i) with the corresponding bit in the data register DR(i). full-adder with the corresponding input and output carries. The transfer from INPR to AC is only for bits 0 through 7. The complement micro operation is obtained by inverting the bit value in AC. The shift-right operation transfers the bit from AC(i + 1), One stage of the adder uses a full-adder with the corresponding input and the shift-left operation transfers the bit from AC(i - 1). The complete adder and logic circuit consists of 16 stages connected together.

Module 3

Arithmetic algorithms: Algorithms for multiplication and division (restoring method) of binary numbers. Array multiplier , Booth's multiplication algorithm.

Pipelining: Basic principles**,** classification of pipeline processors, instruction and arithmetic pipelines (Design examples not required), hazard detection and resolution.

Multiplication Algorithms

The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product. The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is positive. If they are unlike, the sign of the product is negative.

```
23      10111    Multiplicand
19    × 10011    Multiplier
        10111
       10111
      00000   +
     00000
    10111
437 110110101    Product
```

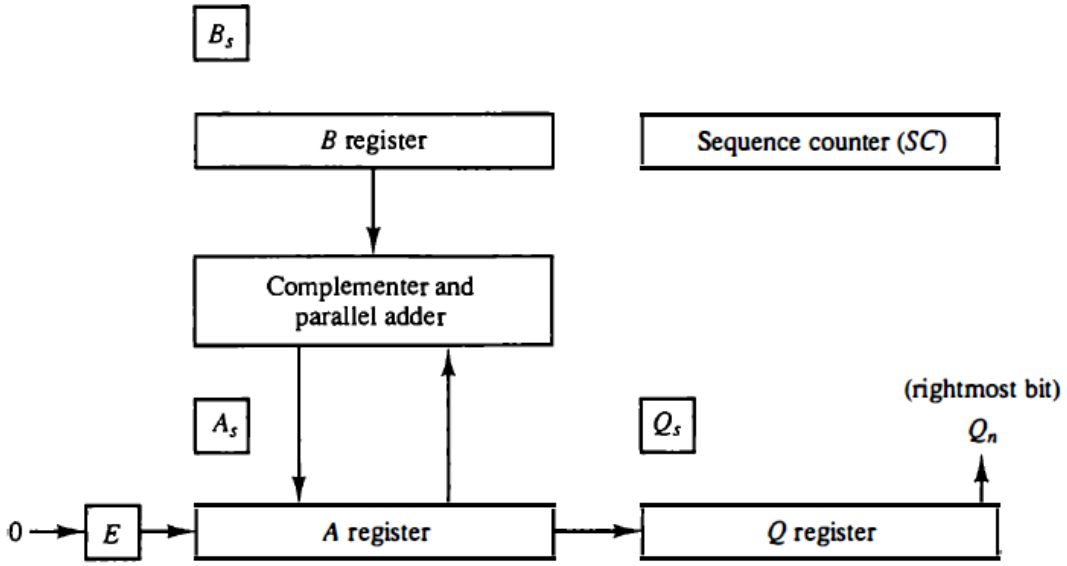
When multiplication is implemented in a digital computer, it is convenient to change the process slightly. First, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. Third, 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 multiplier is stored in the Q register and its sign in Q_s The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1

after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

Initially, the multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig. 10-5. The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by Q_n will hold the bit of the multiplier, which must be inspected next.

Figure 10-5 Hardware for multiply operation.

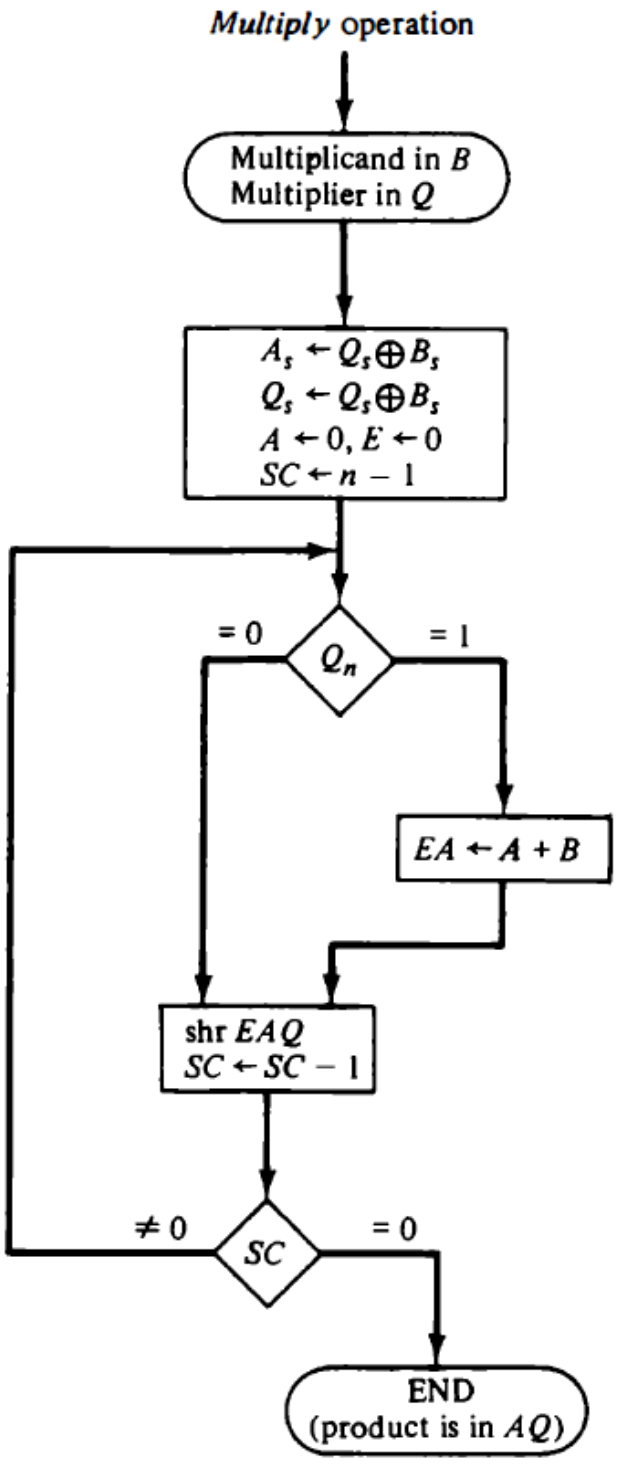


Hardware Algorithm

Figure 10-6 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s respectively. The signs are compared, and both A and Q are set to correspond to the 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 a number equal to the number of bits of the multiplier. We are

assuming here 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 consist of n - 1 bits.

Figure 10-6 Flowchart for multiply operation.



After the initialization, the low-order bit of the multiplier in Q, is tested. If it is a 1, the multiplicand in B is added to the present partial product in A . If it is a 0 , nothing i s done.

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. The process stops when SC = 0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

TABLE 10-2 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	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

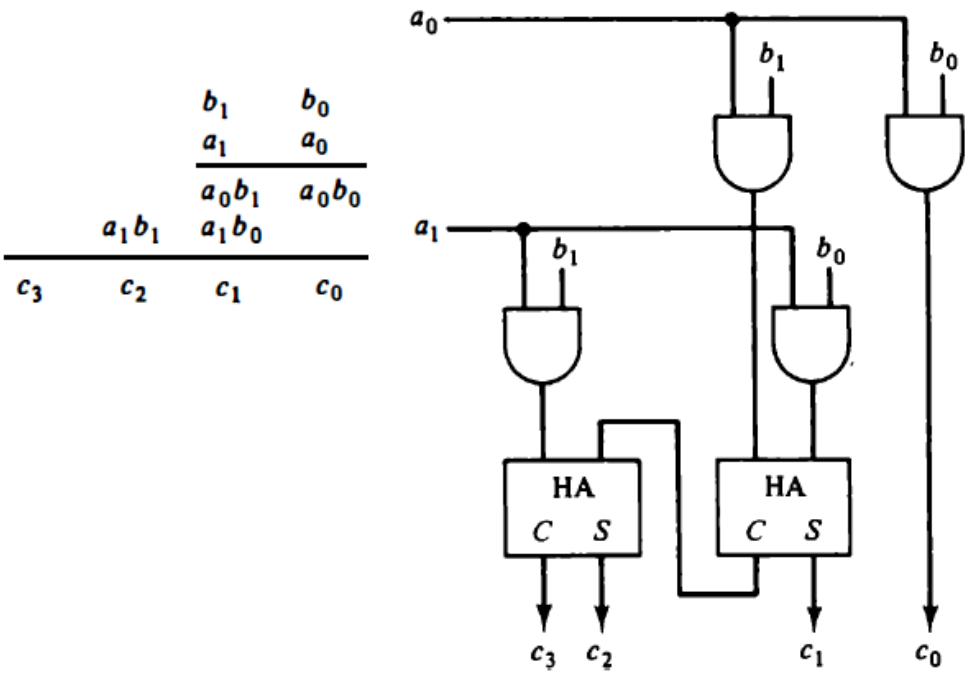
Array Multiplier

Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift microoperations. The multiplication of two binary numbers can be done with one microoperation by means of a combinational circuit that forms the product bits all at once. This is a fast way of multiplying two numbers 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 for this reason it was not economical until the development of integrated circuits. To see how an array multiplier can be

implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 10-9.

The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3 c2 c1 c0• The first partial product is formed by multiplying a0 by b1 b0• The multiplication of two bits such as a0 and b0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented 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 a1 by b1 b0 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.

Figure 10-9 2-bit by 2-bit array multiplier.



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 of 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 x k AND gates and (j - 1) k-bit adders to produce a product of j + k bits. 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 Fig. 10-10.

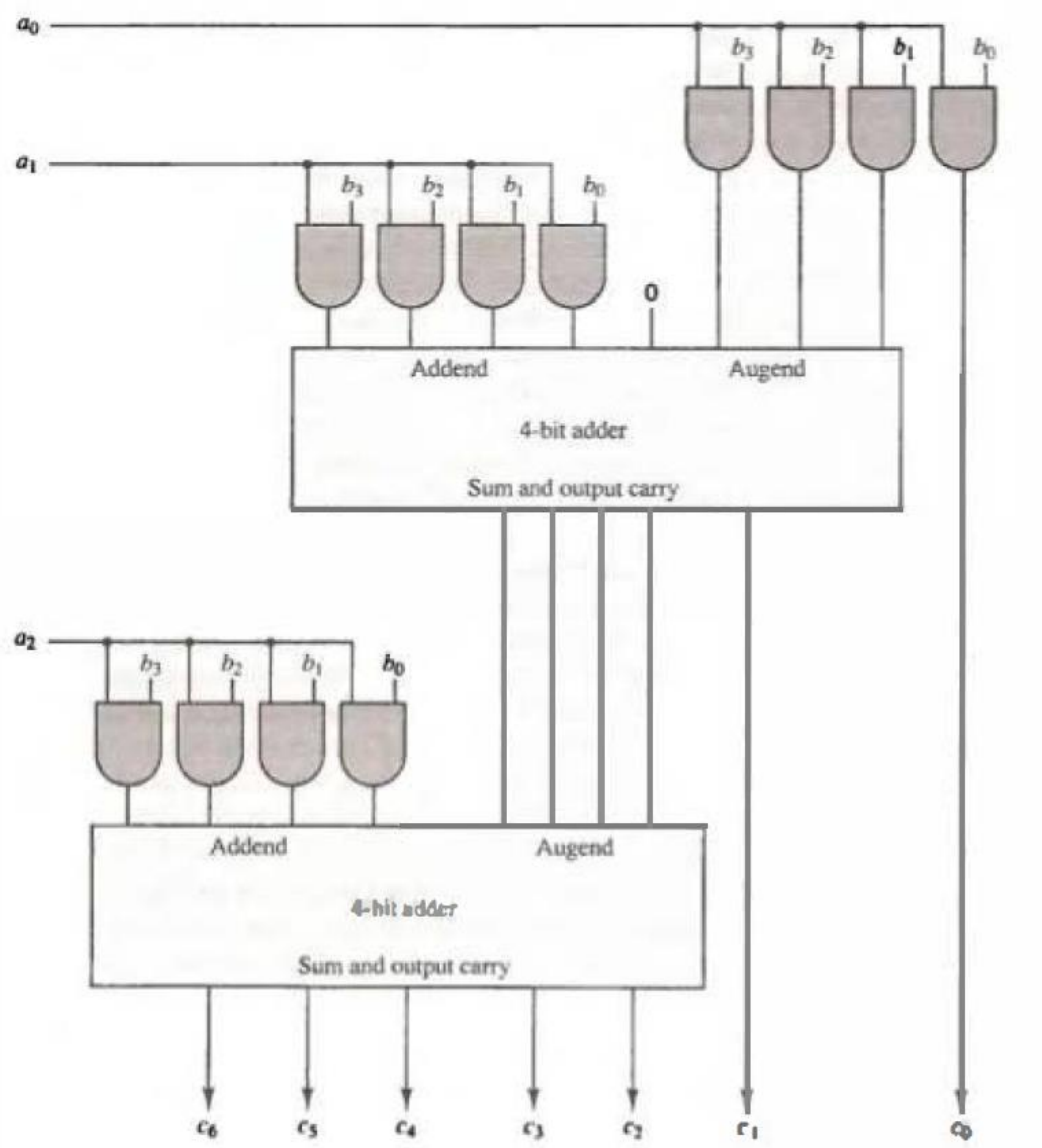
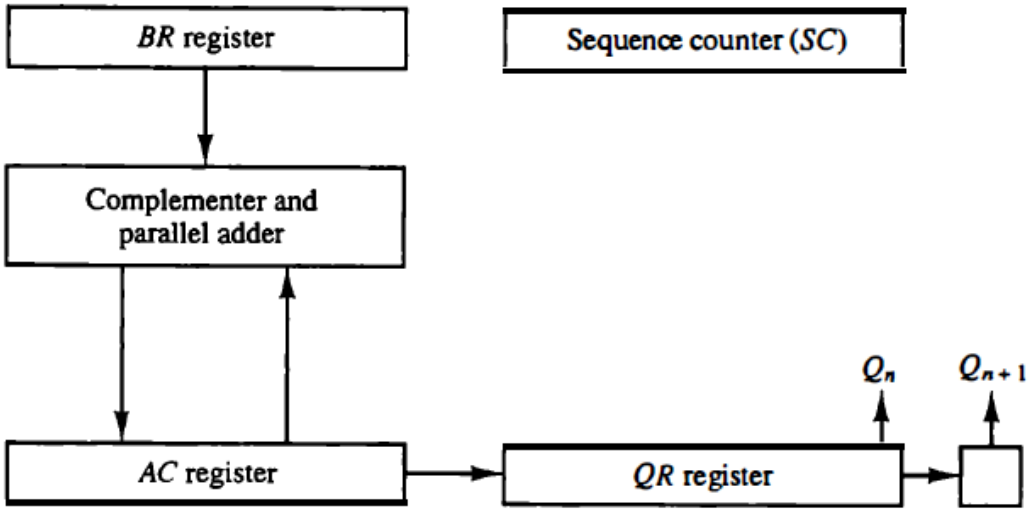


Figure 10-10 4-bit by 3-bit array multiplier.

Booth Multiplication Algorithm

The hardware implementation of Booth algorithm requires the register configuration shown in Fig. 10-7. This is similar to Fig. 10-5 except that the sign bits are not separated from the rest of the registers. To show this difference, we rename registers A, B, and Q, as AC, BR, and QR, respectively. Q_n designates the least significant bit of the multiplier in register QR . An extra flip-flop Q_{n+1} is appended to QR to facilitate a double bit inspection of the multiplier.

Figure 10-7 Hardware for Booth algorithm.



The flowchart for Booth algorithm is shown in Fig. 10-8. AC and the appended bit Q_{n+1} 1 are initially cleared 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 equal to 10, it means that the first 1 in a string of 1' s has been encountered. This requires a subtraction of the multiplicand from the partial product in AC . If the two bits are equal to 01, it means that the first 0 in a string of 0' s has been encountered. This requires the addition of the multiplicand to the partial product in AC . When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow. The 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 unchanged. The sequence counter is decremented and the computational loop is repeated n times. A numerical example of Booth algorithm is shown in Table 10-3 for n = 5.

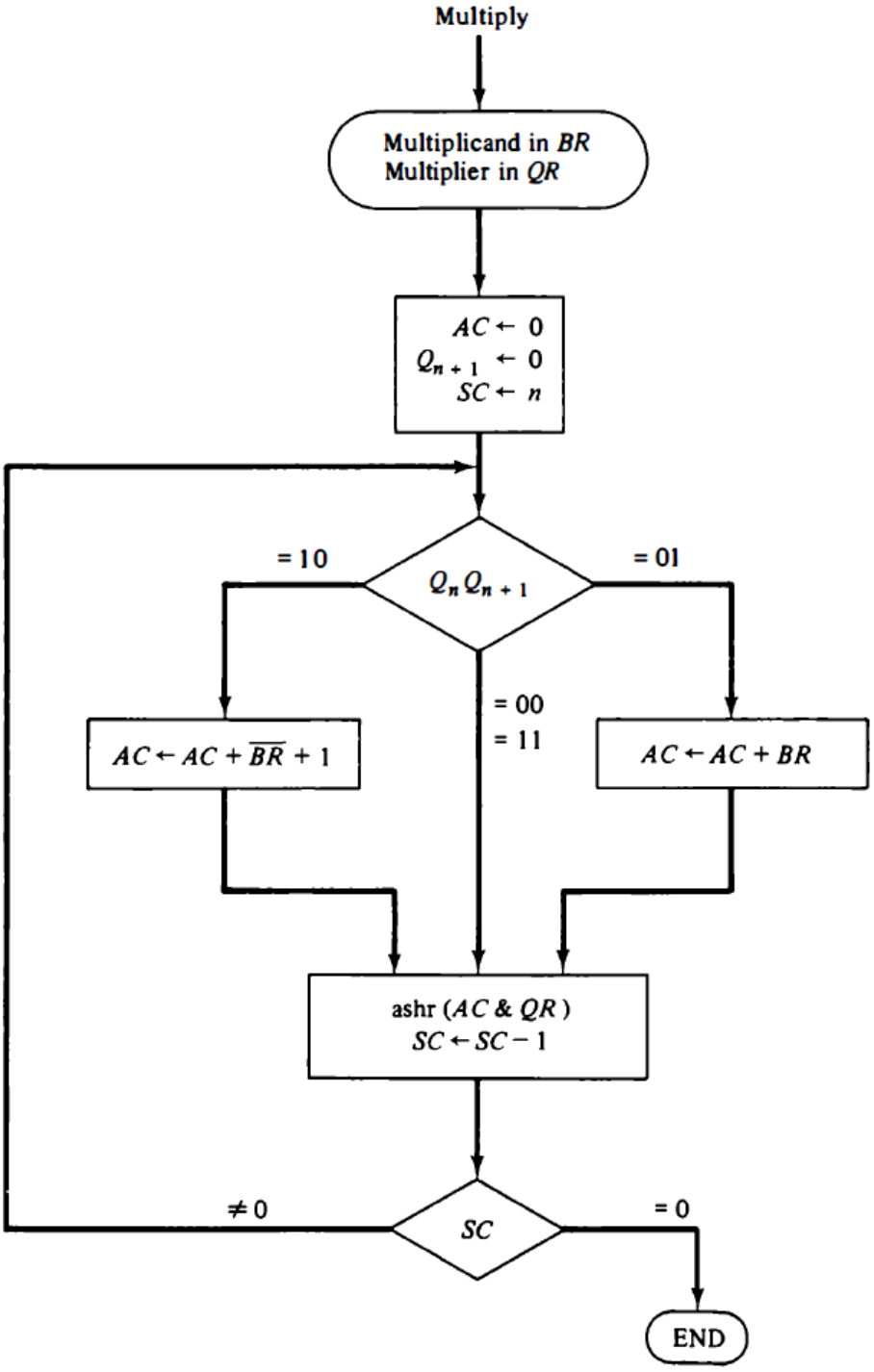


Figure 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

It shows the step-by-step 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 and QR and is positive. 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 10-3 Example of Multiplication with Booth Algorithm

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	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	00011	10101	1	000

Division

The divisor B consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bit number is smaller than B, we try again by taking the six most significant bits of A and compare this number with B. The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The divisor is then shifted once to the right and subtracted from the dividend. The difference is called a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. 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. Note that the result gives both a quotient and a remainder.

Figure 10-11 Example of binary division.

Divisor:		
B = 10001	11010	Quotient = Q
	10111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position. Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end-carry.

The hardware for implementing the division operation is identical to that required for multiplication and consists of the components shown in Fig. 10-5. Register EAQ is now shifted to the left with 0 inserted into Q, and the previous value of E lost. The numerical example is repeated in Fig. 10-12 to clarify 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. The information about the relative magnitude is available in E. If E = 1, it signifies that A >= B. A quotient bit 1 is inserted into Q, and the partial remainder is shifted to the left to repeat the process. If E = 0, it signifies that A < B so the quotient in Q, remains a 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed. 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 the final remainder is in A . 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 determined from the signs of the dividend and the divisor. If the two signs are alike, the sign of the quotient is plus. If they are unalike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

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

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\bar{B} + 1$		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 10-12 Example of binary division with digital hardware.

Divide Overflow

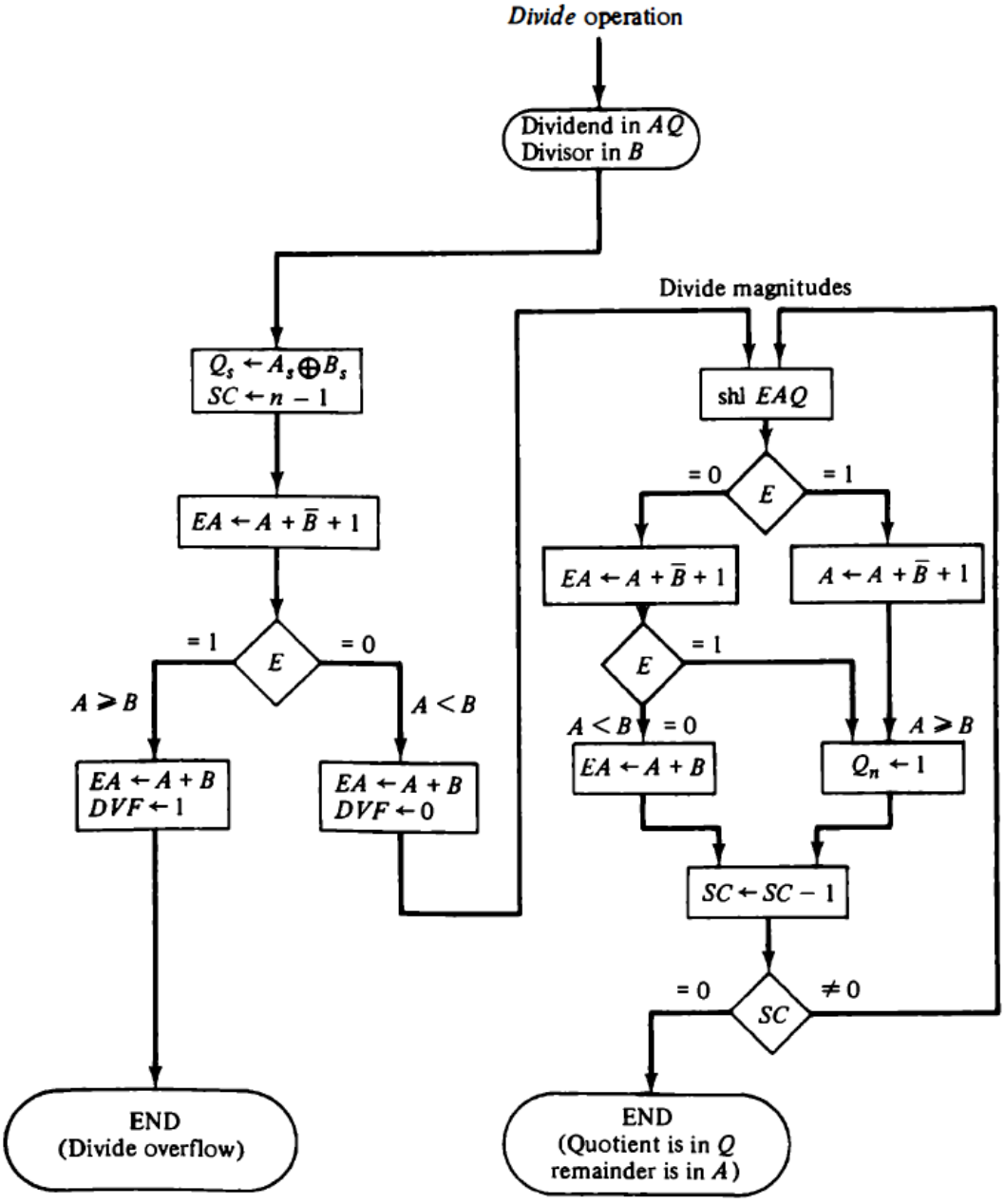
The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, 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 Fig. 10-11 we note that 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, the condition for overflow can be stated as follows: A divide-overflow condition occurs if the high-order half bits of the dividend constitute 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.

The occurrence of a divide overflow can be handled in a variety of ways. In some computers it is the responsibility of the programmers to check if DVF is set after each divide instruction. They then can branch to a subroutine that takes a corrective measure such as rescaling the data to avoid overflow. In some older computers, the occurrence of a divide overflow stopped the computer and this condition was referred to as a divide stop . Stopping the operation of the computer is not recommended because it is time consunting. The procedure in most computers is to provide an interrupt request when DVF is set. The interrupt causes the computer to suspend the current program and branch to a service routine to take a corrective measure. The most common corrective measure is to remove the program and type an error message explaining the reason why the program could not be completed. It is then the responsibility of the user who wrote the program to rescale the data or take any other corrective measure. The best way to avoid a divide overflow is to use floatingpoint data. We will see in Sec. 10-5 that a divide overflow can be handled very simply if numbers are in floating-point representation.

Figure 10-13 Flowchart for divide operation.



Hardware Algorithm

The hardware divide algorithm is shown in the flowchart of Fig. 10-13 . The dividend is in A and Q and the divisor in B . The sign of the result is transferred into Q_s to be part of the quotient. 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 o f n bits. Since a n operand must b e stored with its sign, one bit o f the word will be occupied by the sign and the magnitude will consist of n - 1 bits. A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A . If A ;, B, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely. If A < B, no divide overflow occurs so the value of the dividend is restored by adding B to A .

Pipelining

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 execu· tion time. For example, while an instruction is being executed in the ALU, the next instruction can be read from memory.

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

Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

A, • B, + C, for i = 1, 2, 3, . . . , 7

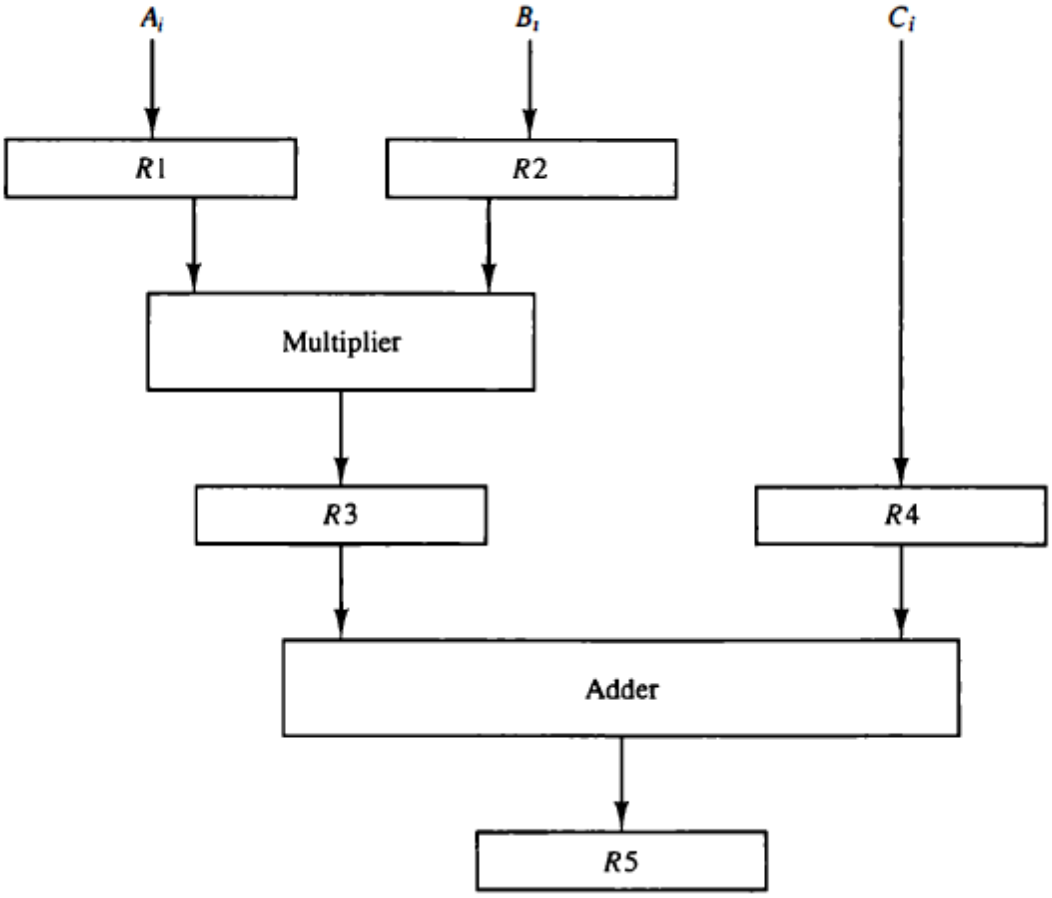
Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2. R 1 through RS are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

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

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

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

Figure 9-2 Example of pipeline processing.



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 A1 and 81 into R 1 and R2.

TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

The second dock pulse transfers the product of R 1 and R2 into R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R 1 and R2. The third clock pulse operates on all three segments simultaneously. It places A, and B, into R1 and R2, transfers the product of R1 and R2 into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS. From there on, each dock 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.

Any operation that can be decomposed into a sequence of sub operations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with clifferent sets of data. The general structure of a four-segment pipeline is illustrated in Fig. 9-3. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S; that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers R; that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously.

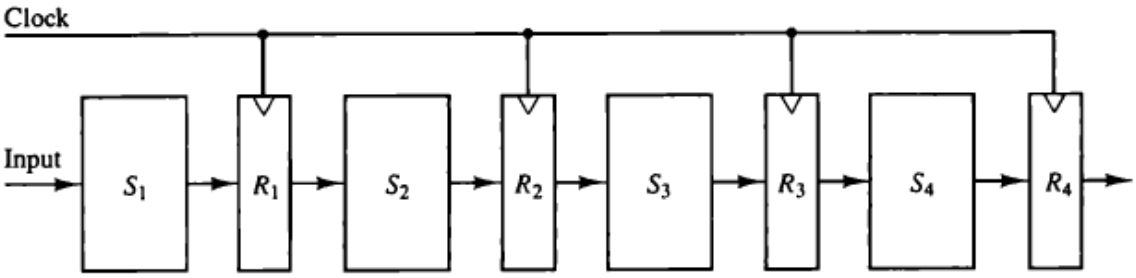


Figure 9-3 Four-segment pipeline.

The behavior of a pipeline can be illustrated with a space-time diagram. This is a cliagram that shows the segment utilization as a function of time. The space-time cliagram of a four-segment pipeline is demonstrated in Fig. 9-4. The horizontal axis clisplays the time in clock cycles and the vertical axis gives the segment number. The diagram shows six tasks T1 through T6 executed in four segments. Initially, task 1i is handled by segment 1. After the first clock, segment 2 is busy with T,, while segment 1 is busy with task T2• Continuing in this manner, the first task T1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

Figure 9-4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

There are two areas of computer design where the pipeline organization is applicable. An arithmetic pipeline divides an arithmetic operation into suboperations for execution in the pipeline segments. An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle. The two types of pipelines are explained in the following sections.

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.

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 represent 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 suboperations 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.

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 produces an 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.

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 produces the sum

$$Z = 1.0324 \times 10^3$$

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.

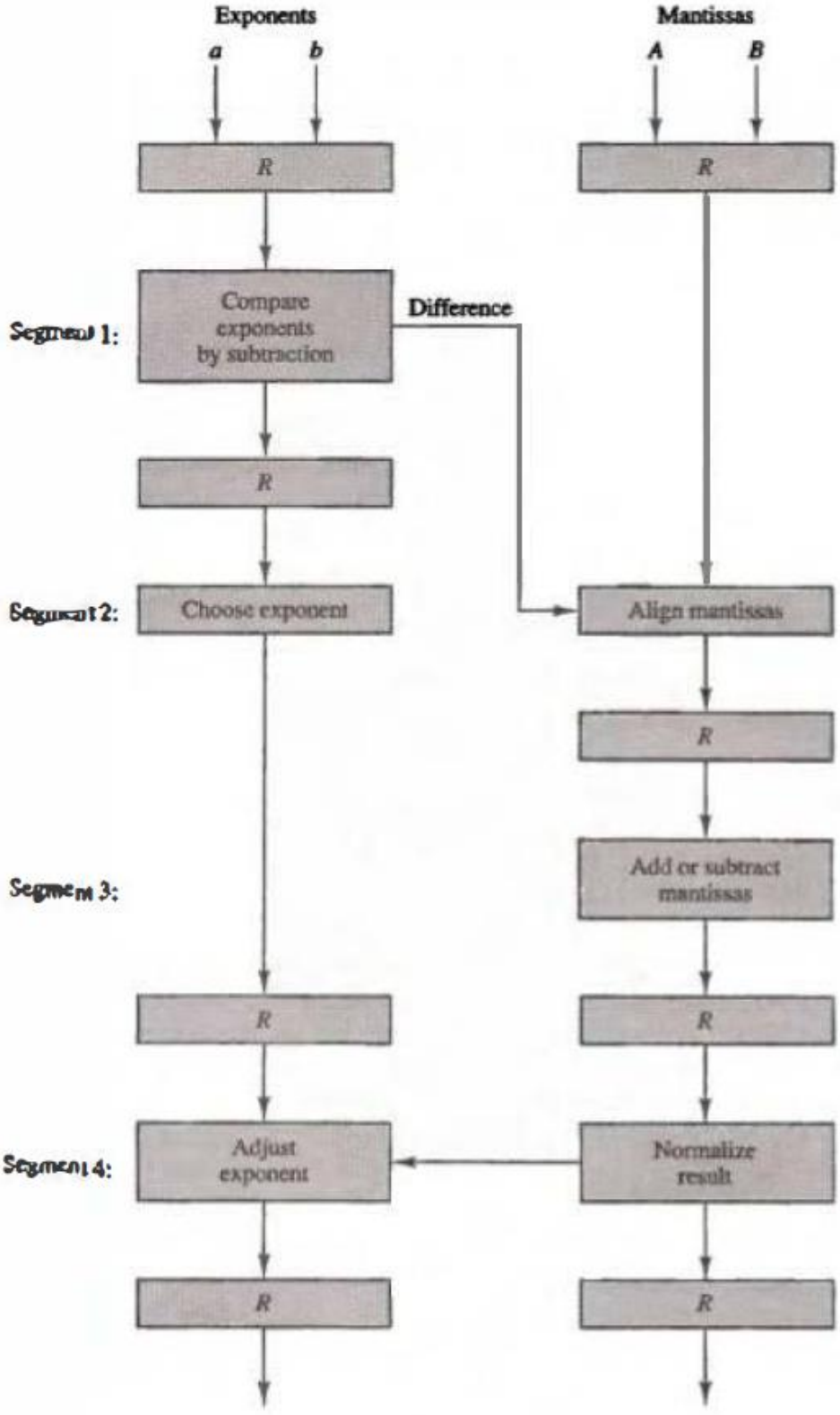


Figure 9-6 Pipeline for floating-point addition and subtraction.

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the timedelaysof the four segments are t, = 60 ns, t2 = 70 ns, t3 = 100ns, t4 = 80 ns, and the interface registers have a delay of t, = 10 ns. The dock cycle is chosen to be t, = t3 + t, = 110 ns. An equivalent nonpipeline floating point adder-subtractor will have a delay time t, = t, + t2 + t, + t4 + t, = 320ns. In this case the pipelined adder has a speedup of 32011 10 = 2. 9 over the non pipelined adder.

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 causes the instruction fetch and execute phases to overlap and perform simultaneous operations. Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each 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 instructioncycle 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 executed.

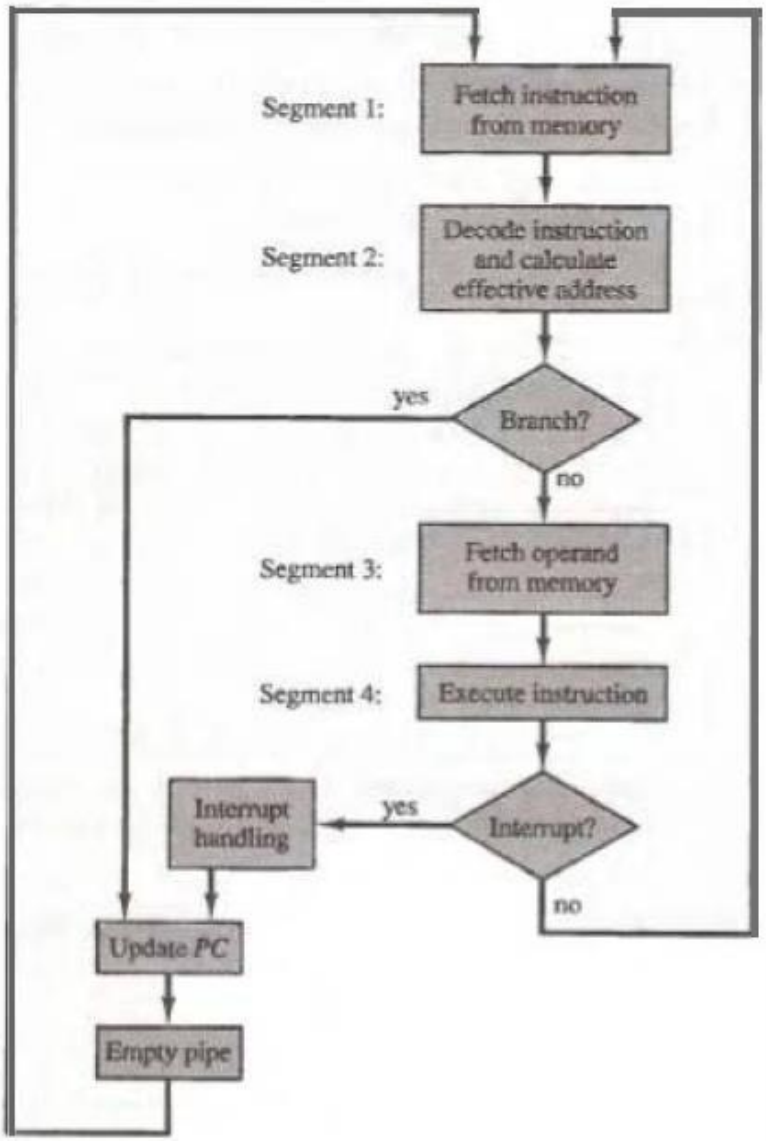


Figure 9-7 Four-segment CPU pipeline.

Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment. Assume further that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.

Figure 9-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3. The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO. Thus up to four sub operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 9-8 Timing of instruction pipeline.

Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence. In that case the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

Figure 9-8 shows the operation of the instruction pipetine. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. Fl is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories

so that the operation in Fl and FO can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FL Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. Resource conflicts caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. Branch difficulties arise from branch and other instructions that change the value of PC.

Data Dependency

A difficulty that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address. A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations. A data dependency occurs when an instruction needs data that are not yet available. For example, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for data to become available by the first instruction. Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. For example, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register. Therefore, the operand

access to memory must be delayed until the required address is available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.The most straightforward method is to insert hardware interlocks . An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.

Another technique called operand forwarding uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register ffie. This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.

A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program. The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting delayed load no-operation instructions. This method is referred to as delayed load. An example of delayed load is presented in the next section.

Handling of Branch Instructions

One of the major problems in operating an instruction pipeline is the occurrence of branch instructions. A branch instruction can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading the program counter with the target address. In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied. As mentioned previously, the branch instruction breaks the norrnal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline. One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction. An extension of this procedure is to continue fetching

instructions from both places until the branch decision is made. At that time control chooses the instruction stream of the correct program flow.

Another possibility is the use of a branch target buffer or BTB. The BTB is an associative memory included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch. It also stores the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction. If it is in the BTB, the instruction is available directly and prefetch continues from the new path. If the instruction is not in the BTB, the pipeline shifts to a new instruction stream and stores the target instruction in the BTB. The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

A variation of the BTB is the loop buffer. This is a small very high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches. The program loop can be executed directly without having to access memory until the loop mode is removed by the final branching out.

Another procedure that some computers use is branch prediction. A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. A correct prediction eliminates the wasted time caused by branch penalties.

A procedure employed in most ruse processors is the deli<yed branch . In this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions. An example of delayed branch is the insertion of a no-operation instruction after a branch instruction. This causes the computer to fetch the target instruction during the execution of the no operation instruction, allowing a continuous flow of the pipeline.

Detection and Resolution of Pipeline Hazards

pipeline hazards are simply any obstruction, condition or we can say any situation that is obstructing pipelines to work or act normally.

Generally, there are mainly three **types of hazards**:

1. Structural Hazard
2. Data Hazard
3. Control Hazard

We will discuss all the **hazards** one by one, let's start with structural hazard

1) Structural Hazard

When we try to do multiple or two different things using the same hardware in the same clock cycle this prevents the pipeline to work properly this is known as structural hazard.

To avoid this situation processor can use stalling in the pipelining.

Stall of one cycle will shift the pipeline to the one clock cycle until hazard can fully be avoided or eliminated.

This situation or hazard will not occur if we had separate data cache and instruction cache.

2) Data Hazard

In data hazard, read and write operations of shared variables by different instructions in a pipeline may lead to different kind of data dependencies such as,

a. Read after write hazard
b. Write after reading hazard
c. Write after write hazard

They arise when an instruction depends on the result of previous instruction is a way i.e exposed by overlapping of instructions in the pipelining.

To avoid data hazard we can do

a. Internal forwarding
b. Stalling

Internal forwarding- for avoiding the condition of reading after write hazard the latch which is placed right after ALU unit will be used as internal forwarding for transferring the output to the next instruction to the latch which is placed before the decoding unit.

Note: These are some other important terms related to data hazard:

i. **Data dependency**
ii. **Flow dependency** - Register R1 is loaded by I1 and then used by I2. Hence, the result of one register after executing instruction may affect the operant of that register.
iii. **Output dependency** - when two instructions want to write the same time.

3) Control Hazard

Control hazard occurs when we need to find the main root of any branch instruction, and cannot proceed further with another instruction till we get to root of that instruction.

Dependencies in a pipelined processor

There are mainly three types of dependencies possible in a pipelined processor. These are :

1) Structural Dependency

2) Control Dependency

3) Data Dependency

These dependencies may introduce stalls in the pipeline.

Stall : A stall is a cycle in the pipeline without new input.

Structural Hazard

This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.

Example:

ICycle	1	2	3	4	5
I1	IF(Mem)	ID	EX	Mem	

I2		IF(Mem)	ID	EX			
I3			IF(Mem)	ID	EX		
I4				IF(Mem)	ID		

In the above scenario, in cycle 4, instructions I1 and I4 are trying to access same resource (Memory) which introduces a resource conflict.

To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available. This wait will introduce stalls in the pipeline as shown below:

Cycle	1	2	3	4	5	6	7	8
I1	IF(Mem)	ID	EX	Mem	WB			
I2		IF(Mem)	ID	EX	Mem	WB		
I3			IF(Mem)	ID	EX	Mem	WB	
I4				–	–	–	IF(Mem)	

Solution for structural dependency

To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.

Renaming : According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory(CM) and Data memory(DM) respectively. CM will contain all the instructions and DM will contain all the operands that are required for the instructions.

ICycle	1	2	3	4	5	6	7
I1	IF(CM)	ID	EX	DM	WB		
I2		IF(CM)	ID	EX	DM	WB	
I3			IF(CM)	ID	EX	DM	WB
I4				IF(CM)	ID	EX	DM
I5					IF(CM)	ID	EX

I6	IF(CM)	ID
I7		IF(CM)

Control Dependency (Branch Hazards)

This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.

Consider the following sequence of instructions in the program:

100: I1

101: I2 (JMP 250)

102: I3

.

250: BI1

Expected output: I1 -> I2 -> BI1

NOTE: Generally, the target address of the JMP instruction is known after ID stage only.

Instruction/ Cycle	1	2	3	4	5	6
I1	IF	ID	EX	MEM	WB	
I2		IF	ID (PC:250)	EX	Mem	WB
I3			IF	ID	EX	Mem
BI1				IF	ID	EX

Output Sequence: I1 -> I2 -> I3 -> BI1

So, the output sequence is not equal to the expected output, that means the pipeline is not implemented correctly.

To correct the above problem we need to stop the Instruction fetch until we get target address of branch instruction. This can be implemented by introducing delay slot until we get the target address.

Instruction/ Cycle	1	2	3	4	5	6
I1	IF	ID	EX	MEM	WB	
I2		IF	ID (PC:250)	EX	Mem	WB
Delay	–	–	– –	–	–	
BI1				IF	ID	EX

Output Sequence: I1 -> I2 -> Delay (Stall) -> BI1

As the delay slot performs no operation, this output sequence is equal to the expected output sequence. But this slot introduces stall in the pipeline.

Solution for Control dependency Branch Prediction is the method through which stalls due to control dependency can be eliminated. In this at 1st stage prediction is done about which branch will be taken.For branch prediction Branch penalty is zero.

Branch penalty : The number of stalls introduced during the branch operations in the pipelined processor is known as branch penalty.

NOTE : As we see that the target address is available after the ID stage, so the number of stalls introduced in the pipeline is 1. Suppose, the branch target address would have been present after the ALU stage, there would have been 2 stalls. Generally, if the target address is present after the kth stage, then there will be (k – 1) stalls in the pipeline.

Total number of stalls introduced in the pipeline due to branch instructions = Branch frequency * Branch Penalty

Data Dependency (Data Hazard)

Example: Let there be two instructions I1 and I2 such that:

I1 : ADD R1, R2, R3

I2 : SUB R4, R1, R2

When the above instructions are executed in a pipelined processor, then data dependency condition will occur, which means that I2 tries to read the data before I1 writes it, therefore, I2 incorrectly gets the old value from I1.

Instruction / Cycle	1	2	3	4
I1	IF	ID	EX	DM
I2		IF	ID(Old value)	EX

To minimize data dependency stalls in the pipeline, operand forwarding is used.

Operand Forwarding : In operand forwarding, we use the interface registers present between the stages to hold intermediate output so that dependent instruction can access new value from the interface register directly. For avoiding the condition of reading after write hazard the latch which is placed right after ALU unit will be used as internal forwarding for transferring the output to the next instruction to the latch which is placed before the decoding unit.

Considering the same example:

I1 : ADD R1, R2, R3

I2 : SUB R4, R1, R2

ICycle	1	2	3	4
I1	IF	ID	EX	DM
I2		IF	ID	EX

Data Hazards

Data hazards occur when instructions that exhibit data dependence, modify data in different stages of a pipeline. Hazard cause delays in the pipeline. There are mainly three types of data hazards:

1) RAW (Read after Write) [Flow/True data dependency]

2) WAR (Write after Read) [Anti-Data dependency]

3) WAW (Write after Write) [Output data dependency]

Let there be two instructions I and J, such that J follow I. Then,

RAW hazard occurs when instruction J tries to read data before instruction I writes it.

Eg:

I: R2 <- R1 + R3

J: R4 <- R2 + R3

WAR hazard occurs when instruction J tries to write data before instruction I reads it.

Eg:

I: R2 <- R1 + R3

J: R3 <- R4 + R5

WAW hazard occurs when instruction J tries to write output before instruction I writes it.

Eg:

I: R2 <- R1 + R3

J: R2 <- R4 + R5

WAR and WAW hazards occur during the out-of-order execution of the instructions.

Module 4

Control Logic Design: Control organization – Hard_wired control-microprogram control – control of processor unit - Microprogram sequencer, micro programmed CPU organization - horizontal and vertical micro instructions.

The function of the control unit in a digital computer is to initiate sequences of micro operations. The number of different types of micro operations that are available in a given system is finite. The complexity of the digital system is derived from the number of sequences of micro operations that are performed. When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired. Microprogramming is a second alternative for designing the control unit of a digital computer. The principle of microprogramming is an elegant and systematic method for controlling the micro operation sequences in a digital computer. The control variables at any given time can be represented by a string of l's and O's called a control word. As such, control words can be programmed to perform various operations on the components of the system.

FUNCTIONS OF CONTROL UNIT

- The control unit directs the entire computer system to carry out stored program instructions.
- The control unit must communicate with both the arithmetic logic unit (ALU) and main memory.
- The control unit instructs the arithmetic logic unit that which logical or arithmetic operation is to be performed.
- The control unit co-ordinates the activities of the other two units as well as all peripherals and auxiliary storage devices linked to the computer.

Design of Control Unit

The Control Unit is classified into two major categories:

1. Hardwired Control
2. Microprogrammed Control

Hardwired Control

The Hardwired Control organization involves the control logic to be implemented with gates, flip-flops, decoders, and other digital circuits. This organization is very complicated if we have a large control unit. In this organization, if the design has to be modified or changed, requires changes in the wiring among the various components. Thus the modification of all the combinational circuits may be very difficult.

ADVANTAGES

- Hardwired Control Unit is fast because control signals are generated by combinational circuits.
- The delay in generation of control signals depends upon the number of gates.

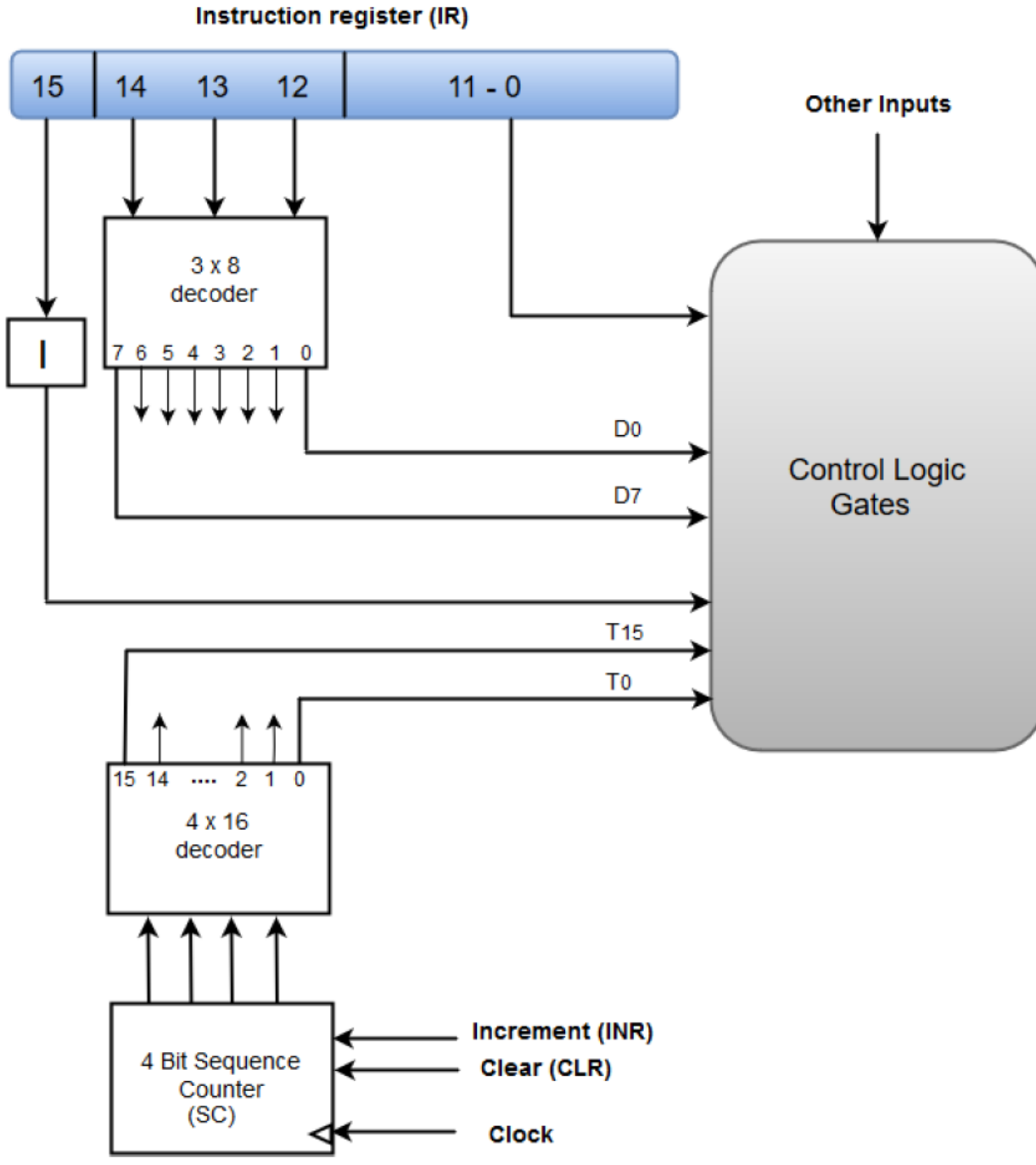
DISADVANTAGES

- More is the control signals required by CPU; more complex will be the design of control unit.
- Modifications in control signal are very difficult. That means it requires rearranging of wires in the hardware circuit.

 It is difficult to correct mistake in original design or adding new feature in existing design of control unit.

The following image shows the block diagram of a Hardwired Control organization.

Control Unit of a Basic Computer:



- A Hard-wired Control consists of two decoders, a sequence counter, and a number of logic gates.

- An instruction fetched from the memory unit is placed in the instruction register (IR).

The component of an instruction register includes; I bit, the operation code, and bits 0 through 11. The instruction register is divided into three parts: the I bit, operation code, and address part. First 12-bits (0-11) to specify an address, next 3-bits specify the operation code (opcode) field of the instruction and last left most bit specify the addressing mode I.I = 0 for direct address I = 1 for indirect address

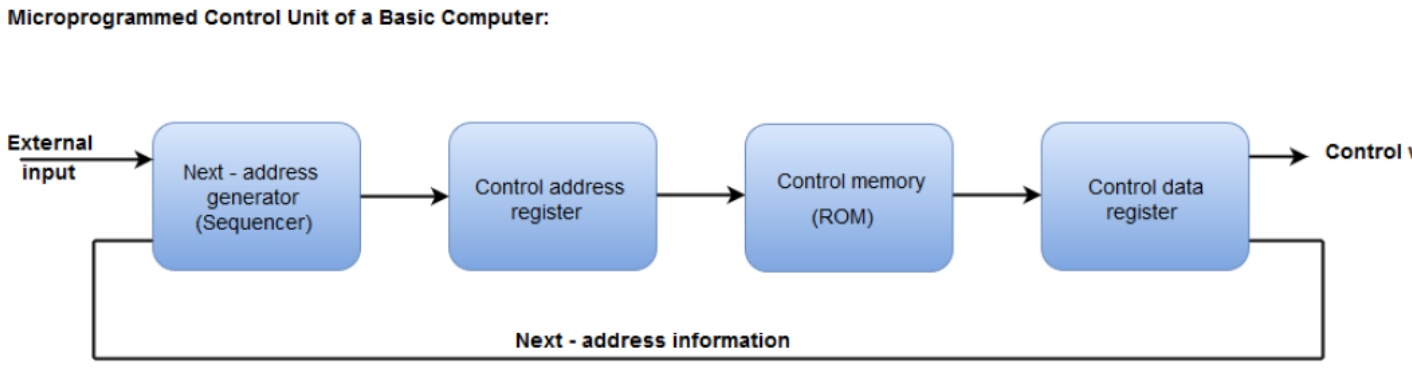
- The operation code in bits 12 through 14 are coded with a 3 x 8 decoder.
- The outputs of the decoder are designated by the symbols D0 through D7.
- The operation code at bit 15 is transferred to a flip-flop designated by the symbol I.
- The operation codes from Bits 0 through 11 are applied to the control logic gates.
- The Sequence counter (SC) can count in binary from 0 through 15.

Micro-programmed Control

The Microprogrammed Control organization is implemented by using the programming approach.

In Microprogrammed Control, the micro-operations are performed by executing a program consisting of micro-instructions.

The following image shows the block diagram of a Microprogrammed Control organization.



A memory that is part of a control unit is referred to as a control memory. A computer that employs a micro programmed control unit will have two separate memories: a main memory and a control memory. The main memory is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data. In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of microinstructions that specify various internal

control signals for execution of register microoperations. Each machine instruction initiates a series of microinstructions in control memory. These microinstructions generate the microoperations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction. The control memory is assumed to be a ROM, within which all control information is permanently stored. The control memory address register specifies the address o f the microinstruction, and the control data register holds the microinstruction read from memory.

The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the microoperations are being executed, the next address is

computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory. The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.

The control data register holds the present microinstruction while the next address is computed and read from memory. The data register is sometimes called a pipeline register. It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established, there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

- The Control memory address register specifies the address of the micro-instruction.
- The Control memory is assumed to be a ROM, within which all control information is permanently stored.
- The control register holds the microinstruction fetched from the memory.
- The micro-instruction contains a control word that specifies one or more micro-operations for the data processor.

- While the micro-operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
- The next address generator is often referred to as a micro-program sequencer, as it determines the address sequence that is read from control memory.

- Hardwired control units are generally faster than microprogrammed designs. In hardwired control, we saw how all the control signals required inside the CPU can be generated using a state counter and a PLA circuit.
- A microprogrammed control unit is a relatively simple logic circuit that is capable of (1) sequencing through microinstructions and (2) generating control signals to execute each microinstruction.

Hardwired Control Unit	**Microprogrammed Control Unit**
Hardwired control unit generates the control signals needed for the processor using logic circuits	Micrprogrammed control unit generates the contro of micro instructions stored in control memory
Hardwired control unit is faster when compared to microprogrammed control unit as the required control signals are generated with the help of hardwares	This is slower than the other as micro instructions a signals here
More costlier as everything has to be realized in terms of logic gates	Less costlier than hardwired control as only micro ins for generating control signals
It cannot handle complex instructions as the circuit design for it becomes complex	It can handle complex instructions
Only limited number of instructions are used due to the hardware implementation	Control signals for many instructions can be generate
Used in computer that make suse of Reduced Instruction Set Computers(RISC)	Used in computer that makes use of Complex Instruc Computers(CISC)

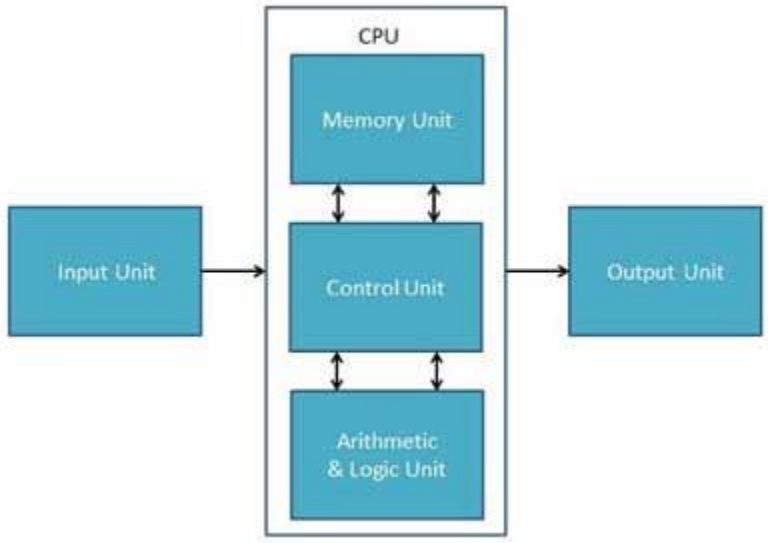
CPU(Central Processing Unit)

Central Processing Unit (CPU) consists of the following features −

- CPU is considered as the brain of the computer.
- CPU performs all types of data processing operations.
- It stores data, intermediate results, and instructions (program).
- It controls the operation of all parts of the computer.

CPU itself has following three components.

- Memory or Storage Unit
- Control Unit
- ALU(Arithmetic Logic Unit)



Memory or Storage Unit

This unit can store instructions, data, and intermediate results. This unit supplies information to other units of the computer when needed. It is also known as internal storage unit or the main memory or the primary storage or Random Access Memory (RAM).

Its size affects speed, power, and capability. Primary memory and secondary memory are two types of memories in the computer. Functions of the memory unit are −

- It stores all the data and the instructions required for processing.

- It stores intermediate results of processing.
- It stores the final results of processing before these results are released to an output device.
- All inputs and outputs are transmitted through the main memory.

Control Unit

This unit controls the operations of all parts of the computer but does not carry out any actual data processing operations.

Functions of this unit are −

- It is responsible for controlling the transfer of data and instructions among other units of a computer.
- It manages and coordinates all the units of the computer.
- It obtains the instructions from the memory, interprets them, and directs the operation of the computer.
- It communicates with Input/Output devices for transfer of data or results from storage.
- It does not process or store data.

ALU (Arithmetic Logic Unit)

This unit consists of two subsections namely,

- Arithmetic Section
- Logic Section

Arithmetic Section

Function of arithmetic section is to perform arithmetic operations like addition, subtraction, multiplication, and division. All complex operations are done by making repetitive use of the above operations.

Logic Section

Function of logic section is to perform logic operations such as comparing, selecting, matching, and merging of data.

Micro programmed sequencer for a control memory Microprogram sequencer:

The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address. The address selection part is called a microprogram sequencer. A microprogram sequencer can be constructed with digital functions to suit a particular application. To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be

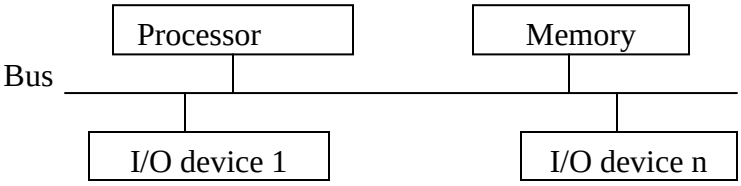
adapted to a wide range of applications. The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed. Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls. Some sequencers provide an output register which can function as the address register for the control memory. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register CAR. The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit. The output from CAR provides the address for the control memory. The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine registers SBR. The other three inputs to multiplexer 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction. Although the figure 4.6 shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time. The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field goes to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit.

Module: 5	**I/O ORGANIZATION –** Accessing I/O devices – interrupts - interrupt hardware – Direct Memory Access. **THE MEMORY SYSTEM -** Basic concepts – Semiconductor RAMs – Memory system considerations – Semiconductor ROMs –Content Addressable memory – Cache Memory -Mapping Functions.

ACCESSING I/O DEVICES

- A simple arrangement to connect I/O devices to a computer is to use a single busstructure. It consists of three sets of lines to carry
  - Address
  - Data
  - Control Signals.
- When the processor places a particular address on address lines, the devices that recognize this address responds to the command issued on the control lines.
- The processor request either a read or write operation and the requested data are transferred over the data lines.
- When I/O devices & memory share the same address space, the arrangement is called **memory mapped I/O.**

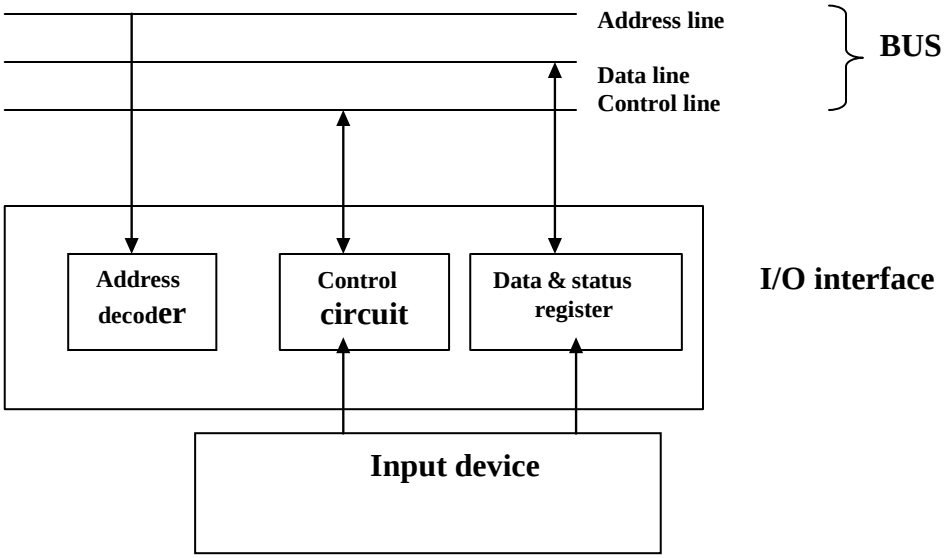
Single Bus Structure



Eg:-

Move DATAIN, R_o - Reads the data from DATAIN then into processor register R_o.
Move R_o, DATAOUT - Send the contents of register Ro to location DATAOUT.
DATAIN - Input buffer associated with keyboard.
DATAOUT - Output data buffer of a display unit / printer.

Fig: I/O Interface for an Input Device



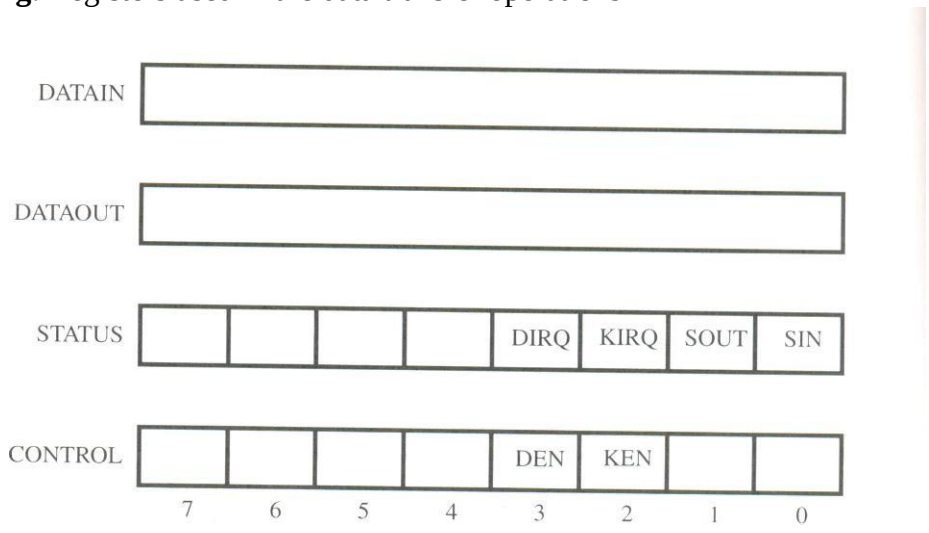
Address Decoder:

- It enables the device to recognize its address when the address appears on addresslines.

Data register - It holds the data being transferred to or from the processor.
Status register - It contains information relevant to the operation of the I/O devices.

- The address decoder, data & status registers and the control circuitry required to co-ordinate I/O transfers constitute the device's I/O interface circuit.
- For an input device, SIN status flag is used SIN = 1, when a character is entered at the keyboard, SIN = 0, once the char is read by processor.
- For an output device, SOUT status flag is used.

Eg: Registers used in the data transfer operations



DIRQ - Interrupt Request for display.
KIRQ - Interrupt Request for keyboard.
KEN - Keyboard enable.
DEN - Display Enable.
SIN, SOUT - Status flags.

The data from the keyboard are made available in the DATAIN register & the data sent to the display are stored in DATAOUT register.

Program:

```
        Move      #LINE,R0          Initialize memory pointer.
WAITK   TestBit   #0,STATUS         Test SIN.
        Branch=0  WAITK             Wait for character to be entered.
        Move      DATAIN,R1         Read character.
WAITD   TestBit   #1,STATUS         Test SOUT.
        Branch=0  WAITD             Wait for display to become ready.
        Move      R1,DATAOUT        Send character to display.
        Move      R1,(R0)+          Store charater and advance pointer.
        Compare   #$0D,R1           Check if Carriage Return.
        Branch≠0  WAITK             If not, get another character.
        Move      #$0A,DATAOUT      Otherwise, send Line Feed.
        Call      PROCESS           Call a subroutine to process the
                                      the input line.
```

Figure 4.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

EXPLANATION:

- This program, reads a line of characters from the keyboard & stores it in a memory buffer starting at locations LINE.
- Then it calls the subroutine "PROCESS" to process the input line.
- As each character is read, it is echoed back to the display.
- Register Ro is used as a pointer to memory buffer area. The contents of R0 are updated using Auto – increment mode so that successive characters are stored in successive memory location.
- Each character is checked to see if there is carriage return (CR), char, which has the ASCII code 0D (hex).
- If it is, a line feed character (ASCII character 0A) is sent to move the cursor one line down on the display & subroutine PROCESS is called. Otherwise, the program loops back towait for another character from the keyboard.

PROGRAM CONTROLLED I/O

In the above example, the processor repeatedly checks a status flag to achieve the required synchronization between Processor & I/O device. (ie) the processor polls the device.

There are 2 mechanisms to handle I/o operations. They are,

- **Interrupt** - Synchronization is achieved by having I/O device send special signal over the bus where is ready for data transfer operation.

- **DMA** - It is a technique used for high speed I/O device. Here, the device interface transfer data directly to or from the memory without continuous involvement by the processor.

INTERRUPTS

When a program enters a wait loop, it will repeatedly check the device status. During this period, the processor will not perform any function. There are many situations where other tasks can be performed while waiting for an I/O device to become ready. To allow this to happen, we can arrange for the I/O device to alert the processor when it becomes ready.

It can do so by sending a hardware signal called an **interrupt** to the processor. At least one of the bus control lines called an **interrupt request line** is usually dedicated for this purpose.

Since the processor is no longer required to continuously check the status of external devices, it can use the waiting period to perform other useful functions. Indeed by using interrupts such waiting periods can ideally be eliminated.

The routine executed in response to an interrupt request is called **Interrupt Service Routine.**

Fig: Transfer of control through the use of interrupts

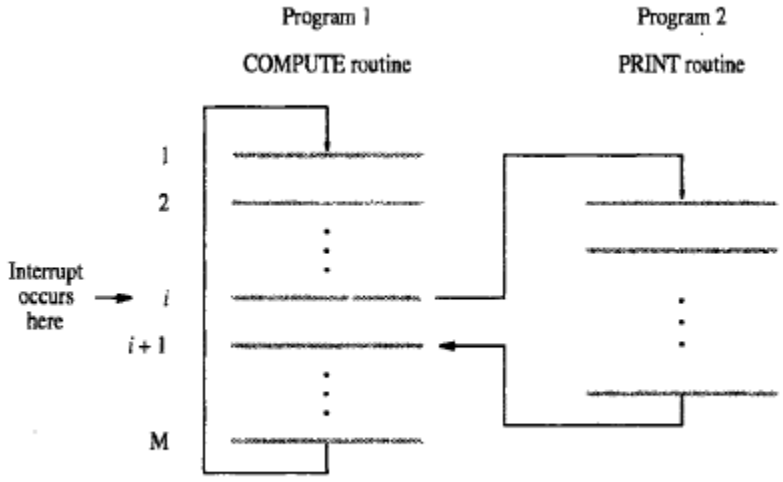


Figure 4.5 Transfer of control through the use of interrupts.

- Assume that an interrupt requests arrives during the execution of instruction i.
- The processor first completes the execution of instruction i. Then it loads the PC (Program Counter) with the address of the first instruction of the ISR.
- After the execution of ISR, the processor has to come back to instruction i + 1.
- Therefore, when an interrupt occurs, the current contents of PC which point to i+1 is put in temporary storage in a known location.
- A return from interrupt instruction at the end of ISR reloads the PC from that temporary storage location, causing the execution to resume at instruction i+1.
- When the processor is handling the interrupts, it must inform the device that itsrequest has been recognized so that it remove its interrupt requests signal.
- This may be accomplished by a special control signal called the **interruptacknowledge signal.**
- The task of saving and restoring the information can be done automatically by theprocessor.

- The processor saves only the contents of **program counter & status register** ie; it saves only the minimal amount of information to maintain the integrity of the program execution.
- Saving registers also increases the delay between the time an interrupt request is received and the start of the execution of the ISR. This delay is called the **Interrupt Latency.**
- Generally, the long interrupt latency in unacceptable.
- The concept of interrupts is used in Operating System and in Control Applications, where processing of certain routines must be accurately timed relative to external events. This application is also called as **real-time processing**.

Interrupt Hardware:

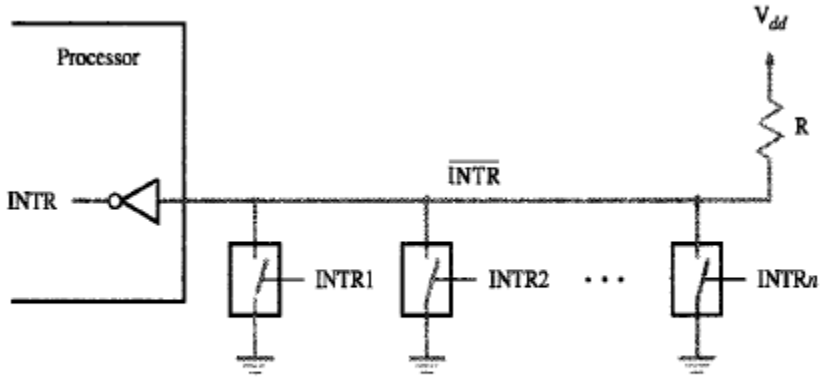


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

- A single interrupt request line may be used to serve n devices. All devices are connected to the line via switches to ground.
- To **request** an interrupt, a device **closes** its associated switch, the voltage on INTR line drops to 0**(zero).**
- If all the interrupt request signals (INTR1 to INTRn) are **inactive**, all switches are open and the voltage on INTR line is equal to **Vdd**..
- When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing INTR request signal received by the processor to go to 1.
- Since closing one or more switches will cause line voltage to drop to 0, the value of INTR is the logical OR of the requests from individual devices. ie;

INTR = INTR1+… +INTRn

$\overline{INTR}$ - It is used to name the INTR signal on common line it is active in the low voltage state.

In figure special gates called,

- **Open collector** (bipolar ckt) or **Open drain** (MOS circuits) is used to drive $\overline{INTR}$ line. The Output of the Open collector (or) Open drain control is equal to a switch to the ground that is open when gates input is in „0" state and closed when the gates input is in „1" state.
- Resistor „R" is called a **pull-up resistor** because it pulls the line voltage upto the high voltage state when the switches are open.

Enabling and Disabling Interrupts:

- The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program & start the execution of another because the interrupt may alter the sequence of events to be executed.
- INTR is active during the execution of **Interrupt Service Routine**.
- There are 3 mechanisms to solve the problem of infinite loop which occurs due to successive interruptions of active INTR signals.
- The following are the typical scenario.

  - The device raises an interrupt request.
  - The processor interrupts the program currently being executed.
  - Interrupts are disabled by changing the control bits is PS (Processor Status register)
  - The device is informed that its request has been recognized & in response, it deactivates the INTR signal.
  - The actions are enabled & execution of the interrupted program is resumed.

Edge-triggered:

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

Handling Multiple Devices:

- When several devices requests interrupt at the same time, it raises some questions. They are.

  - How can the processor recognize the device requesting an interrupt?
  - Given that the different devices are likely to require different ISR, how can the processor obtain the starting address of the appropriate routines in each case?
  - Should a device be allowed to interrupt the processor while another interrupt is being serviced?
  - How should two or more simultaneous interrupt requests be handled?

Polling Scheme:

- If two devices have activated the interrupt request line, the ISR for the selected device (first device) will be completed & then the second request can be serviced.
- The simplest way to identify the interrupting device is to have the ISR polls all the encountered with the IRQ bit set is the device to be serviced
- IRQ (Interrupt Request) -> when a device raises an interrupt requests, the status
- register IRQ is set to 1.

Advantage: It is easy to implement.
Disadvantages: The time spent for interrogating the IRQ bits of all the devices that may not be requesting any service.

Vectored Interrupt:

- Here the device requesting an interrupt may identify itself to the processor by sending a special code over the bus & then the processor start executing the ISR.
- The code supplied by the processor indicates the starting address of the ISR for the device.
- The code length ranges from 4 to 8 bits.
- The location pointed to by the interrupting device is used to store the staring address to ISR.
- The processor reads this address, called the interrupt vector & loads into PC.
- The interrupt vector also includes a new value for the Processor Status Register.
- When the processor is ready to receive the interrupt vector code, it activate the interrupt acknowledge (INTA) line.

Interrupt Nesting:
Multiple Priority Scheme:

- In multiple level priority scheme, we assign a priority level to the processor that can be changed under program control.
- The priority level of the processor is the priority of the program that is currently being executed.
- The processor accepts interrupts only from devices that have priorities higher than its own.
- At the time the execution of an ISR for some device is started, the priority of the processor is raised to that of the device.
- The action disables interrupts from devices at the same level of priority or lower.

Privileged Instruction:

- The processor priority is usually encoded in a few bits of the Processor Status word. It can also be changed by program instruction & then it is write into PS. These instructions are called **privileged instruction.** This can be executed only when the processor is in supervisor mode.
- The processor is in supervisor mode only when executing OS routines.
- It switches to the user mode before beginning to execute application program.

Privileged Exception:

- User program cannot accidently or intentionally change the priority of the processor & disrupts the system operation.

- An attempt to execute a privileged instruction while in user mode, leads to aspecial type of interrupt called the privileged exception.

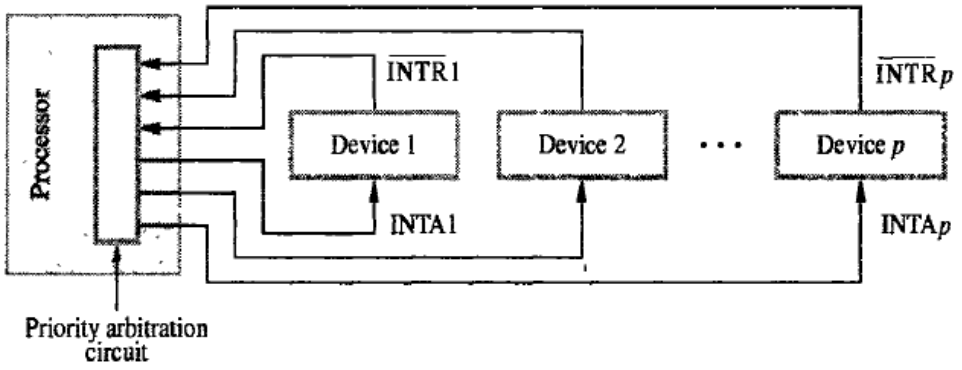
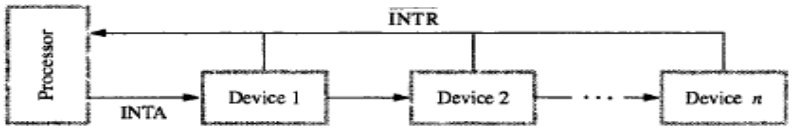


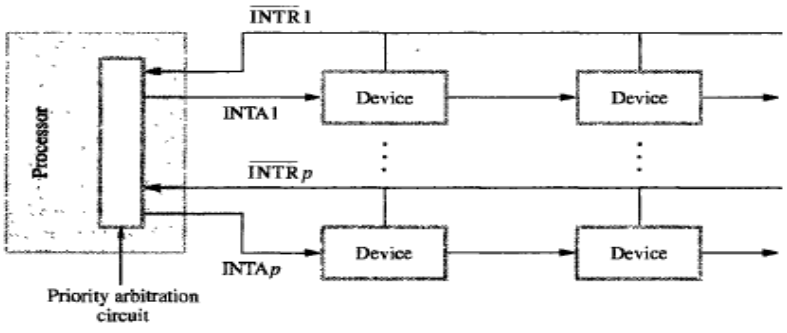
Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

- Each of the interrupt request line is assigned a different priority level.
- Interrupt request received over these lines are sent to a priority arbitration circuit in the processor.
- A request is accepted only if it has a higher priority level than that currently assigned to the processor,

Simultaneous Requests:
Daisy Chain:



(a) Daisy chain



(b) Arrangement of priority groups

Figure 4.8 Interrupt priority schemes.

- The interrupt request line INTR is common to all devices. The interrupt acknowledge line INTA is connected in a daisy chain fashion such that INTA signal propagates serially through the devices.
- When several devices raise an interrupt request, the INTR is activated & the processor responds by setting INTA line to 1. this signal is received by device.

- Device1 passes the signal on to device2 only if it does not require any service.
- If devices1 has a pending request for interrupt blocks that INTA signal & proceeds to put its identification code on the data lines.
- Therefore, the device that is electrically closest to the processor has the highest priority.

Merits:

It requires fewer wires than the individual connections.

Arrangement of Priority Groups:

- Here the devices are organized in groups & each group is connected at a different priority level.
- Within a group, devices are connected in a daisy chain.

Controlling Device Requests:

KEN	□	Keyboard Interrupt Enable
DEN	□	Display Interrupt Enable
KIRQ / DIRQ	□	Keyboard / Display unit requesting an interrupt.

- There are two mechanism for controlling interrupt requests.
- At the devices end, an interrupt enable bit in a control register determines whether the device is allowed to generate an interrupt requests.
- At the processor end, either an interrupt enable bit in the PS (Processor Status) or a priority structure determines whether a given interrupt requests will be accepted.

Initiating the Interrupt Process:

- □ Load the starting address of ISR in location INTVEC (vectored interrupt).
- □ Load the address LINE in a memory location PNTR. The ISR will use this location as a pointer to store the input characters in the memory.
- □ Enable the keyboard interrupts by setting bit 2 in register CONTROL to 1.
- □ Enable interrupts in the processor by setting to 1, the IE bit in the processor status register PS.

Exception of ISR:

- □ Read the input characters from the keyboard input data register. This will cause the interface circuits to remove its interrupt requests.
- □ Store the characters in a memory location pointed to by PNTR & increment PNTR.
- □ When the end of line is reached, disable keyboard interrupt & inform program main.
- □ Return from interrupt.

Exceptions:

- An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin.
- The Exception is used to refer to any event that causes an interruption.

Kinds of exception:

- ❖ Recovery from errors
- ❖ Debugging
- ❖ Privileged Exception

Recovery From Errors:

- Computers have error-checking code in Main Memory, which allows detection of errors in the stored data.
- If an error occurs, the control hardware detects it informs the processor by raising an interrupt.
- The processor also interrupts the program, if it detects an error or an unusual condition while executing the instance (ie) it suspends the program being executed and starts an execution service routine.
- This routine takes appropriate action to recover from the error.

Debugging:

- System software has a program called debugger, which helps to find errors in a program.
- The debugger uses exceptions to provide two important facilities
- They are
  - ❖ Trace
  - ❖ Breakpoint

Trace Mode:

- When processor is in trace mode, an exception occurs after execution of every instance using the debugging program as the exception service routine.
- The debugging program examine the contents of registers, memory location etc.
- On return from the debugging program the next instance in the program being debugged is executed
- The trace exception is disabled during the execution of the debugging program.

Break point:

- Here the program being debugged is interrupted only at specific points selected by the user.

- An instance called the Trap (or) software interrupt is usually provided for this purpose.
- While debugging the user may interrupt the program execution after instance „I"
- When the program is executed and reaches that point it examine the memory and register contents.

Privileged Exception:

- To protect the OS of a computer from being corrupted by user program certain instance can be executed only when the processor is in supervisor mode. These are called privileged exceptions.
- When the processor is in user mode, it will not execute instance (ie) when theprocessor is in supervisor mode , it will execute instance.

DIRECT MEMORY ACCESS

- It is a technique used for high speed I/O device.
- Here, the device interface transfer data directly to or from the memory without continuous involvement by the processor
- A special control unit may be provided to allow the transfer of large block of data at high speed directly between the external device and main memory, without continuous intervention by the processor. This approach is called **DMA.**
- DMA transfers are performed by a control circuit called the **DMA Controller**.

  To initiate the transfer of a block of words , the processor sends,

  - Starting address
  - Number of words in the block
  - Direction of transfer.
- When a block of data is transferred , the DMA controller increment the memory address for successive words and keep track of number of words and it also informs the processor by raising an interrupt signal.
- While DMA control is taking place, the program requested the transfer cannot continue and the processor can be used to execute another program.
- After DMA transfer is completed, the processor returns to the program that requested the transfer.

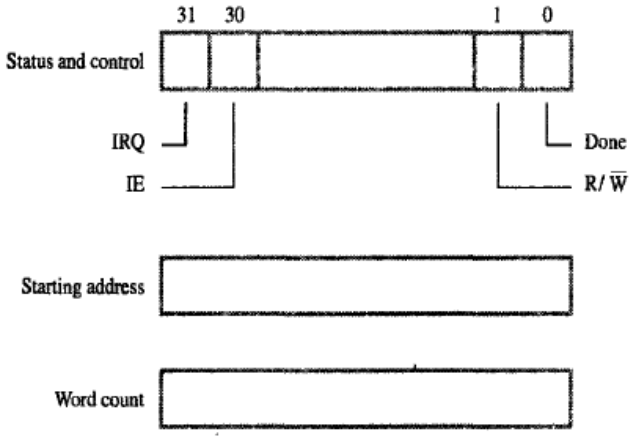


Figure 4.18 Registers in a DMA interface.

R/W - Determines the direction of transfer.
When

R/W =1, DMA controller read data from memory to I/O device.
R/W =0, DMA controller perform write operation.
Done Flag=1, the controller has completed transferring a block of data and is ready to receive another command.
IE=1, it causes the controller to raise an interrupt (interrupt Enabled) after it has completed transferring the block of data.
IRQ=1, it indicates that the controller has requested an interrupt.

Fig: Use of DMA controllers in a computer system

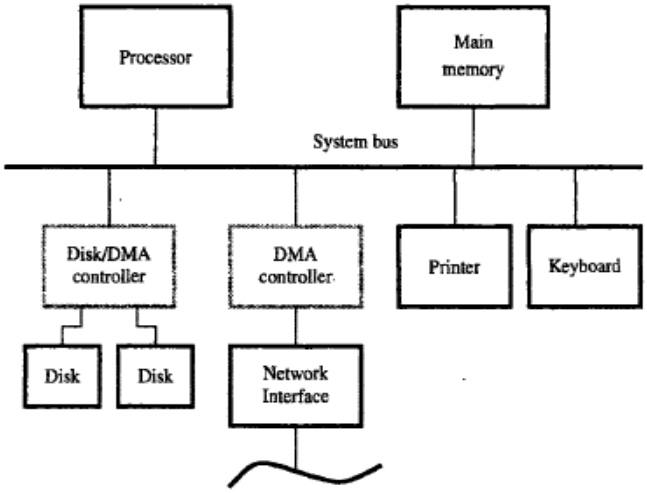


Figure 4.19 Use of DMA controllers in a computer system.

- A DMA controller connects a high speed network to the computer bus. The disk controller two disks, also has DMA capability and it provides two DMA channels.
- To start a DMA transfer of a block of data from main memory to one of the disks, the program write s the address and the word count information into the registers of the corresponding channel of the disk controller.
- When DMA transfer is completed, it will be recorded in status and controlregisters of the DMA channel (ie) **Done bit=IRQ=IE=1**.

Cycle Stealing:

- Requests by DMA devices for using the bus are having higher priority than processor requests.
- Top priority is given to high speed peripherals such as ,
  - Disk
  - High speed Network Interface and Graphics display device.

- Since the processor originates most memory access cycles, the DMA controller can be said to steal the memory cycles from the processor.
- This interviewing technique is called **Cycle stealing.**

Burst Mode:

The DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as **Burst/Block Mode**

Bus Master:

The device that is allowed to initiate data transfers on the bus at any given time is called the bus master.

Bus Arbitration:

It is the process by which the next device to become the bus master is selected and the bus mastership is transferred to it.

Types:

There are 2 approaches to bus arbitration. They are,

- Centralized arbitration (A single bus arbiter performs arbitration)
- Distributed arbitration (all devices participate in the selection of next bus master).

Centralized Arbitration:

- Here the processor is the bus master and it may grants bus mastership to one of its DMA controller.
- A DMA controller indicates that it needs to become the bus master by activating the Bus Request line (BR) which is an open drain line.
- The signal on BR is the logical OR of the bus request from all devices connected to it.
- When BR is activated the processor activates the Bus Grant Signal (BGI) and indicated the DMA controller that they may use the bus when it becomes free.
- This signal is connected to all devices using a daisy chain arrangement.
- If DMA requests the bus, it blocks the propagation of Grant Signal to other devices and it indicates to all devices that it is using the bus by activating open collector line, Bus Busy (BBSY).

Fig: A simple arrangement for bus arbitration using a daisy chain

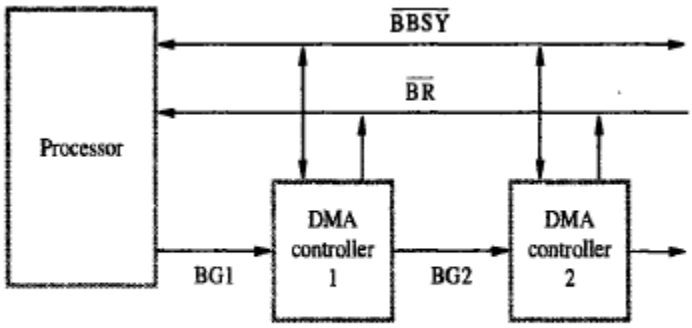


Figure 4.20 A simple arrangement for bus arbitration using a daisy chain.

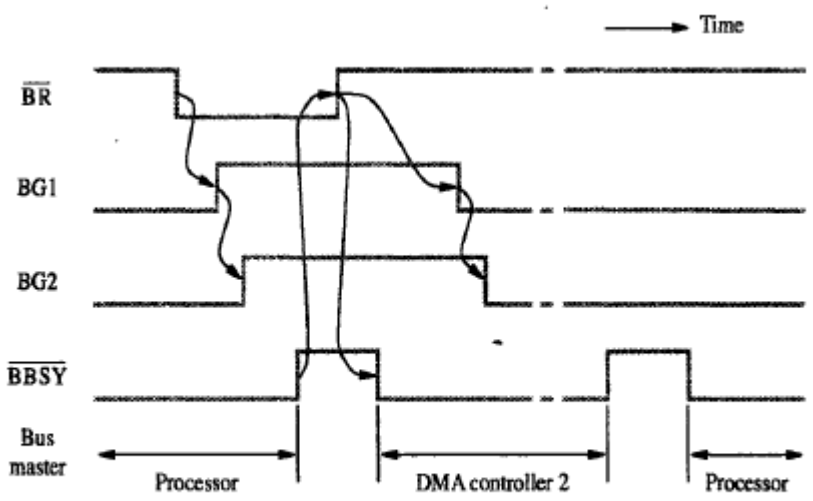


Figure 4.21 Sequence of signals during transfer of bus mastership for the devices in Figure 4.20.

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

Distributed Arbitration:

It means that all devices waiting to use the bus have equal responsibility in carrying out the arbitration process without using a central arbiter.

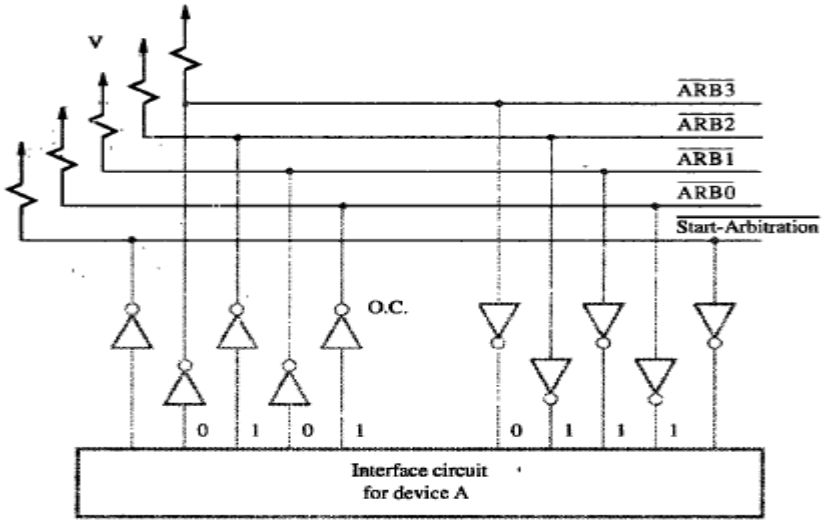


Figure 4.22 A distributed arbitration scheme.

- Each device on the bus is assigned a 4 bit id.
- When one or more devices request the bus, they assert the Start-Arbitration signal & place their 4 bit ID number on four open collector lines, ARB0 to ARB3.

- A winner is selected as a result of the interaction among the signals transmitted over these lines.
- The net outcome is that the code on the four lines represents the request that has the highest ID number.
- The drivers are of open collector type. Hence, if the input to one driver is equal to 1, the input to another driver connected to the same bus line is equal to 0, then bus will be in low-voltage state.

Example

- Assume two devices A & B have their ID 5 (0101), 6(0110). They are requesting the use of bus.
- Device A transmits the pattern 0101 and B transmits 0110. The code seen by both devices is 0111.
- Each devices compares the pattern on the arbitration line to its own ID starting from MSB.
- If it detects a difference at any bit position, it disables the drivers at that bit position and for all lower order bits.
- It does this by placing 0 at the input of these drivers.

- In our example. A detects a difference in line ARB1, hence it disables the drivers on lines ARB1 & ARB0.
- This causes the pattern on the arbitration line to change to 0110 which means that B has won the contention.
- Note that since the code on the priority line is 0111 for a shorter period, device B may be temporarily disable its driver on line ARB0. However, it will enable this driver once it sees a 0 on line ARB1 resulting from the action by device A.

Advantages:

Highly reliable – because operation of the bus is not dependent on any single device.

MEMORY SYSTEM - INTRODUCTION

Programs and data they operate on are resided in the memory of the computer. The execution speed of the program is dependent on how fast the transfer of data and instructions in-between memory and processor. There are three major types of memory available: Cache, Primary and Secondary Memories.

A good memory would be fast, large and inexpensive. Unfortunately, it is impossible to meet all three of these requirements simultaneously. Increased speed and size are achieved at increased cost.

BASIC CONCEPTS:

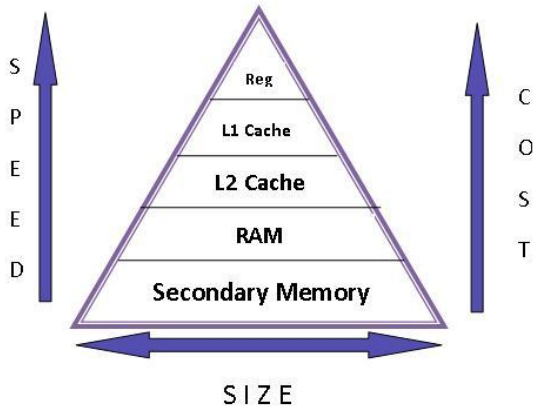
A memory unit is considered as a collection of ***cells***, in which each cell is capable of ***storing a bit*** of information. It stores information in group of bits called byte or word.The maximum size of the memory that can be used in any computer is determined by the addressing scheme.

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

Word length is the number of bits that can be transferred to or from the memory, it can be determined from the width of data bus, if the data bus has a width of n bits, it means word length of that computer system is n bits.

Memory access time: time elapses between the initiation of an operation and the completion of that operation.

Memory cycle time: minimum time delay required between the initiations of two successive memory locations.



Compared to processor, the main memory unit is very slow. So in order to transfer something between memory and processor takes a long time. The processor has to wait a lot. To avoid this speed gap between memory and processor a new memory called cache memory is placed in between main memory and processor.

In the memory hierarchy, speed will decrease and size will increase from top to bottom level.An important design issue is to provide a computer system with as large and fast a memory as possible, within a given cost target.

Random Access Memory (RAM)is a memory system in which any location can be accessed for a Read or Write operation in some fixed amount of time that is independent of the location's address.

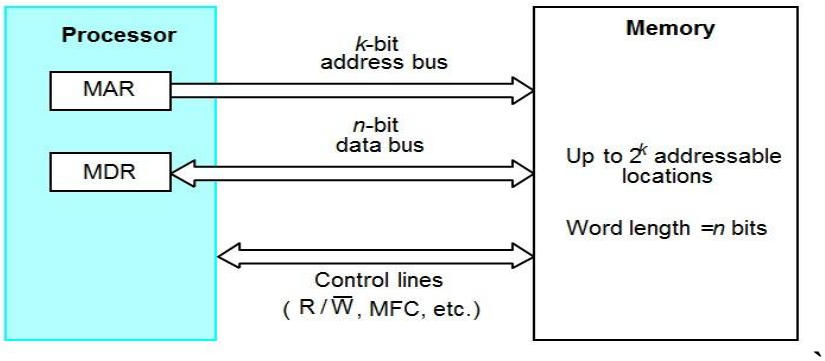
Several techniques to increase the effective size and speed of the memory: Cache memory (to increase the effective speed) &Virtual memory (to increase the effective size

Connection of a memory to a processor

The processor reads data from the memory by loading the address of the location into the MAR

register and setting the R/W line to 1.Upon receiving the MFC signal, the processor loads the data on the data lines into the MDR register.

The processor writes data into the memory location by loading the data into MDR. It indicates that a write operation is involved by setting the R/W line to 0. If MAR is k bits long and MDR is n bits long, then the memory unit may contain up to 2k addressable locations. Memory access can be synchronized by using a clock.



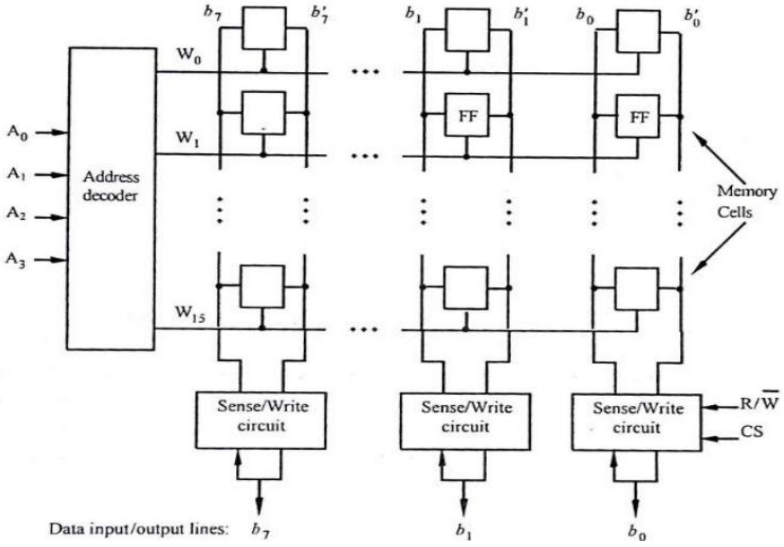
SEMICONDUCTOR RAM MEMORIES

Semiconductor memories are available in a wide range of speeds. Their cycle times range from 100ns to less than 10 ns.

When first introduced in late 1990s, they were much more expensive than the magnetic-core memories they replaced. Because of rapid advances in VLSI (Very Large Scale Integration) technology, the cost of semiconductor memories has dropped dramatically. As a result, they are now used almost exclusively in implementing memories.

Internal Organization of Memory Chips

Memory cells are usually organized in the form of array, in which each cell is capable of storing one bit of information.



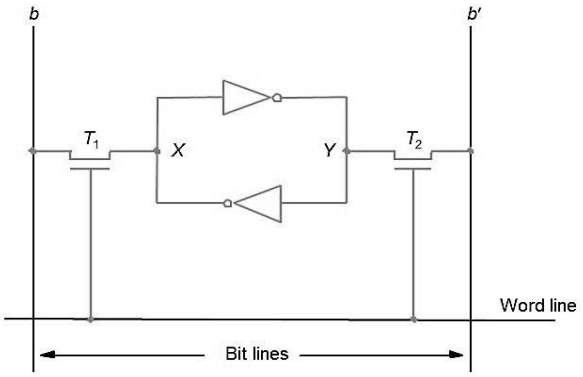
Each row of cells consists of memory word, and all cells of a row are connected to a common line referred to as the word line, which is driven by the address decoder on a chip.The cells in each column are connected to a Sense/Write circuit by two bit lines. Sense/write circuits are connected to the data input/output lines of the memory chip.

During read operation, these circuits sense or read the information stored in the cells selected by a word line and transmit this information to the output data lines. During a write operation, the sense/write circuits receive input information and store it in the cells of the selected word.

Two control lines, R/W and CS, are provided in addition to address and data lines. The Read/Write input specifics the required operation, and the CS input select a given chip in multichip memory system

Static Memories (SRAM)

Static memories are the memories that consist of circuits capable of retaining their state as long as power is applied. Two transistor inverters are cross connected to implement a basic flip-flop. The cell is connected to one word line and two bits lines by transistors T1 and T2. When word line is at ground level, the transistors are turned off and the latch retains its state.



Most of the static RAMs are built using MOS (Metal Oxide Semiconductor) technology, but some are built using bipolar technology. If the cell is in state 1/0, the signal on b is high/low and signal on bit line b' is low/high.

Read operation: In order to read state of SRAM cell, the word line is activated to close switches T1 and T2. Sense/Write circuits at the bottom monitor the state of b and b'.

Write operation: During the write operation, the state of the cell is set by placing the appropriate value on bit line b and its complement on b' and then activating the word line. This forces the cell into the corresponding state. The major advantage of SRAM is very quicly accessed by the processor. The major disadvantage is that SRAM are expensive memory and SRAM are Volatile memory. If the power is interrupted, the cell's content will be lost. Continuous power is needed for the cell to retain its state.

Dynamic Memories (DRAM)

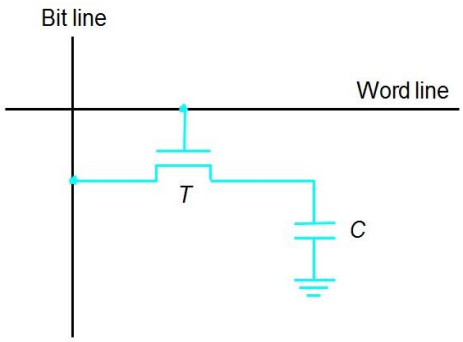
Static RAMs are fast, but the cost is too high because their cells require several transistors.Less expensive RAMs can be implemented if simpler cells are used. Such cells don't retain their state indefinitely; hence they are called dynamic RAMs

Dynamic RAMs (DRAMs) are cheap and area efficient, but they cannot retain their state indefinitely – need to be periodically refreshed. Dynamic memory cell consists of a **capacitor C**, and a

transistor T.

Information is stored is a dynamic memory cell in the form of charge on a capacitor and this charge can be maintained for only tens of milliseconds.

Since the cell is required to store information for a much longer time, its contents must be **periodically refreshed** by restoring the capacitor charge to its full value.

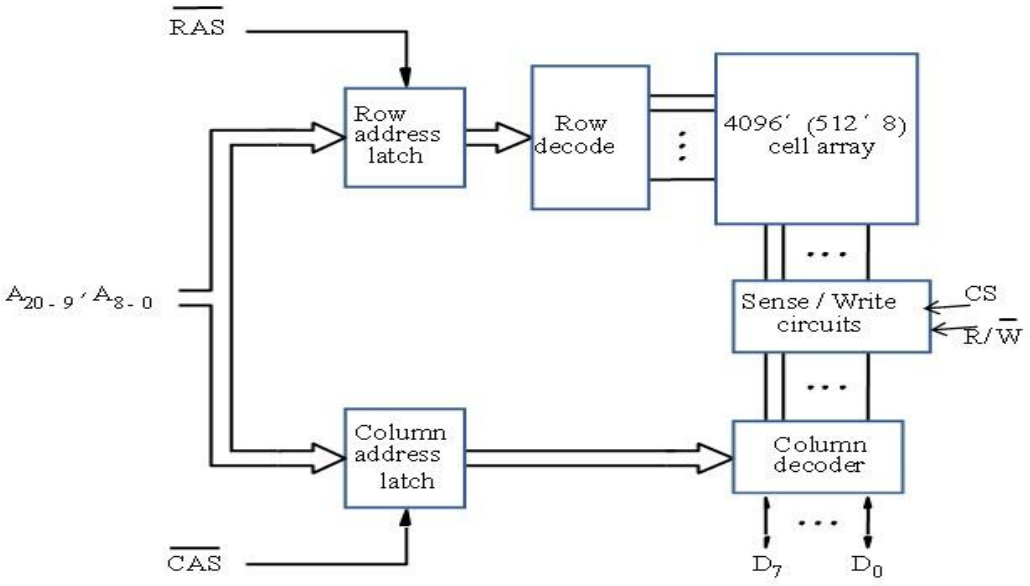


Read Operation: Transistor turned on, Sensor check voltage of capacitor. If voltage is less than Threshold value, Capacitor discharged and it represents logical '0' else if voltage is above Threshold value, Capacitor charged to full voltage and it represents Logical '1'

Write Operation - Transistor is turned on and a voltage is applied/removed to the bit line.

Asynchronous Dynamic RAM:

In Asynchronous dynamic RAM, the timing of the memory device is controlled asynchronously.A specialized memory controller circuit provides the necessary control signals, RAS and CAS, which govern the timing. The processor must take into account the delay in the response of the memory.



In the diagram above, we can see that there are two extra elements with two extra lines attached to them: the Row Address Latch is controlled by the RAS (or Row Address Strobe) pin, and the Column Address Latch is controlled by the CAS (or Column Address Strobe) pin.

Read Operation:

1. The row address is placed on the address pins via the address bus.
2. The RAS pin is activated, which places the row address onto the Row Address Latch.
3. The Row Address Decoder selects the proper row to be sent to the sense amps.

4. The Write Enable (not pictured) is deactivated, so the DRAM knows that it's not being written to.
5. The column address is placed on the address pins via the address bus.
6. The CAS pin is activated, which places the column address on the Column Address Latch.
7. The CAS pin also serves as the Output Enable, so once the CAS signal has stabilized the sense amps, it place the data from the selected row and column on the Data Out pin so that it can travel the data bus back out into the system.
8. RAS and CAS are both deactivated so that the cycle can begin again.

Write Operation:

1. In the write operation, the information on the data lines is transferred to the selected circuits. For this write enable is activated

Fast Page Mode

Suppose if we want to access the consecutive bytes in the selected row. This can be done without having to reselect the row. Add a latch at the output of the sense circuits in each row.
All the latches are loaded when the row is selected.Different column addresses can be applied to select and place different bytes on the data lines. Consecutive sequence of column addresses can be applied under the control signal CAS, without reselecting the row.

This methodology allows a block of data to be transferred at a much faster rate than random accesses.A small collection/group of bytes is usually referred to as a block.This transfer capability is referred to as the fast page mode feature. This mode of operation is useful when there is requirement for fast transfer of data *(Eg: Graphical Terminals)*

Synchronous DRAM's

Operation is directly synchronized with processor clock signal. The outputs of the sense circuits are connected to a latch. During a Read operation, the contents of the cells in a row are loaded onto the latches. During a refresh operation, the contents of the cells are refreshed without changing the contents of the latches.

Data held in the latches correspond to the selected columns are transferred to the output.

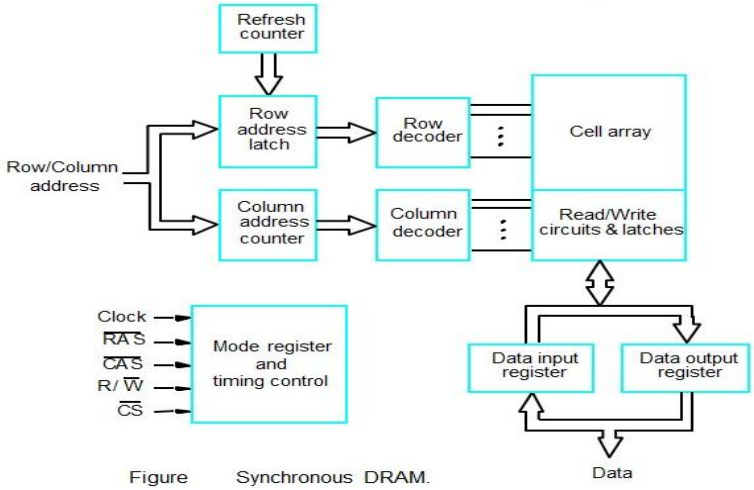


Figure Synchronous DRAM.

For a burst mode of operation, successive columns are selected using column address counter and clock. CAS signal need not be generated externally. A new data is placed duringraising edge of the clock

Memory latency is the time it takes to transfer a word of data to or from memory.

Memory bandwidth is the number of bits or bytes that can be transferred in one second.

Double Data Rate SDRAM

DDR-SDRAM is a faster version of SDRAM. The standard SDRAM perform all actions on the rising edge of the clock signal.DDR SDRAM is also access the cell array in the same way but transfers data on both edges of the clock. So bandwidth is essentially doubled for long burst transfers.

To make it possible to access the data at a high enough rate, the cell array is organized in two banks. Each bank can access separately.Consecutive words of a given block arc stored in different banks. Such interleaving of words allows simultaneous access to two words that are transferred on successive edges of the clock.

Static RAM	Dynamic RAM
More expensive	Less expensive
No refresh	Deleted & refreshed
High power	Less power
Less storage capacity	Higher storage capacity
MOS transistors	Transistor & capacitor
Faster	Slow
More reliable	Less reliable

Structure of Larger Memories

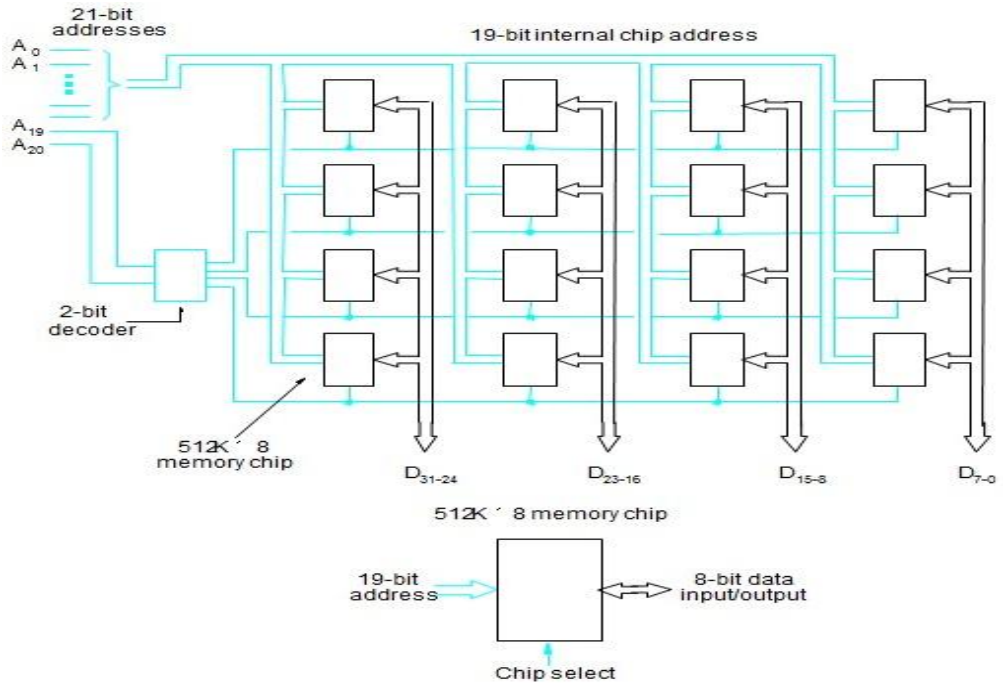
Let discuss about how memory chips may be connected to form a much larger memory.

Static Memory Systems

Implement a memory unit of 2M words of 32 bits each. Use 512x8 static memory chips. Each column consists of 4 chips. Each chip implements one byte position. A chip is selected by setting its chip select control line to 1. Selected chip places its data on the data output line, outputs of other chips are in high impedance state. 21 bits to address a 32-bit word and high order 2 bits are needed to select the row, by activating the four Chip Select signals. 19 bits are used to access specific byte locations inside the selected chip.

Dynamic Memory Systems

Large dynamic memory systems can be implemented using DRAM chips in a similar way to static memory systems. Placing large memory systems directly on the motherboard will occupy a large amount of space. Also, this arrangement is inflexible since the memory system cannot be expanded easily.



Packaging considerations have led to the development of larger memory units known as SIMMs (Single In-line Memory Modules) and DIMMs (Dual In-line Memory Modules). Memory modules are an assembly of memory chips on a small board that plugs vertically onto a single socket on the motherboard in order to occupy less space on the motherboard. And also allows for easy expansion by replacement.

MEMORY SYSTEM CONSIDERATIONS

The choice of a RAM chip for a given application depends on several factors: Cost, speed, power, size etc. SRAMs are faster, more expensive and smaller. In the case of DRAMs are slower, cheaper and larger.

If speed is the primary requirement static RAMS are the most appropriate one. Static RAMs are mostly used in cache memories. If cost is the prioritized factor then we are going for dynamic RAMs. Dynamic RAMs are used for implementing computer main memories.

Refresh overhead:

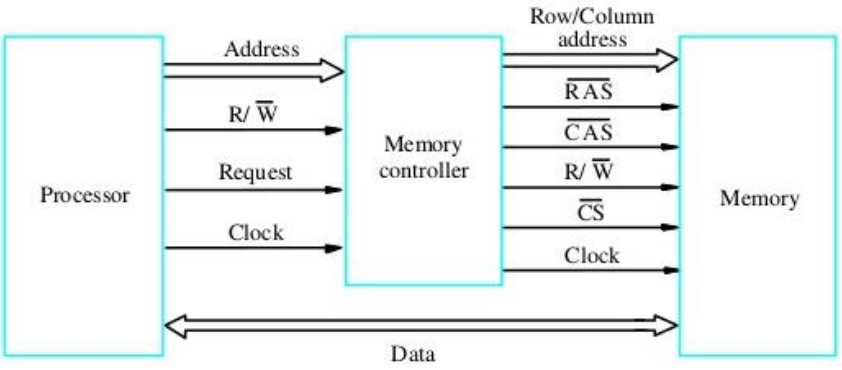
All dynamic memories have to be refreshed. In DRAM, the period for refreshing all rows is 16ms whereas 64ms in SDRAM.

Eg: Suppose a SDRAM whose cells are in 8K (8192) rows; 4 clock cycles are needed to access (read) each rows; then it takes 8192×4=32,768 cycles to refresh all rows; if the clock rate is 133 MHz, then it takes $32,768/(133\times10^{-6})=246\times10^{-6}$ seconds; suppose the typical refreshing period is 64ms, then the refresh overhead is 0.246/64=0.0038<0.4% of the total time available for accessing the memory.

Memory Controller

Dynamic memory chips use multiplexed address inputs so that we can reduce the number of pins. The address is divided into two parts and they are the High order address bits and Low order address bits. The high order selects a row in the cell array and the low order address bits selects a column in the

cell array. The address selection is done under the control of RAS and CAS signal respectively for high order and low order address bits.

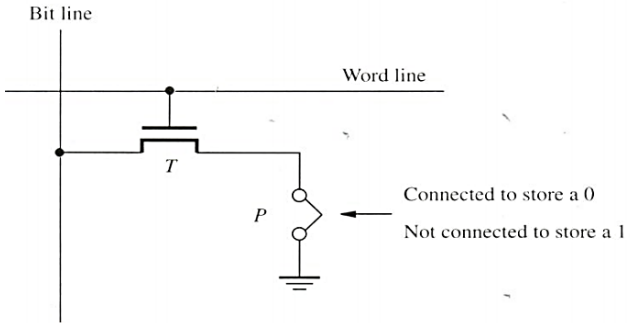


READ ONLY MEMORY

SRAM and SDRAM chips are volatile: Lose the contents when the power is turned off.Many applications need memory devices to retain contents after the power is turned off.

For example, computer is turned on; the operating system must be loaded from the diskinto the memory. For this we need to store instructions which would load the OS from the disk that they will not be lost after the power is turned off. So we need to store the instructions into a non-volatile memory.

Non-volatile memory is read in the same manner as volatile memory. The normal operation involves only reading of data, this type of memory is called Read-Only memory (ROM). The data are written into a ROM when it is manufactured and is permanent memory**.**



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

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

Types of ROM

Different types of non-volatile memory are

- PROM
- EPROM
- EEPROM
- Flash Memory

Programmable Read-Only Memory (PROM):

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

The PROMs provides flexibility and faster data access. It is less expensive because they can be programmed directly by the user.

Erasable Reprogrammable Read-Only Memory (EPROM):

EPROM allows the stored data to be erased and new data to be loaded. In an EPROM cell, a connection to ground is always made at 'P' and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned 'off'.

During programming, an electrical charge is trapped in an insulated gate region. The charge is retained for more than 10 years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use, the quartz lid is sealed with a sticker.

EPROM can be erased by exposing it to ultra-violet light for duration of up to 40 minutes. Usually, an EPROM eraser achieves this function.

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

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

Electrically Erasable Programmable Read-Only Memory (EEPROM):

EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow.

Merits:It can be both programmed and erased electrically. It allows the erasing of all cell contents selectively. **Demerits:**It requires different voltage for erasing ,writing and reading the stored data.

Flash memory:

Flash memory is a non-volatile memory chip used for storage and for transferring data between a personal computer (PC) and digital devices. It has the ability to be electronically reprogrammed and erased. It is often found in USB flash drives, MP3 players, digital cameras and solid-state drives.

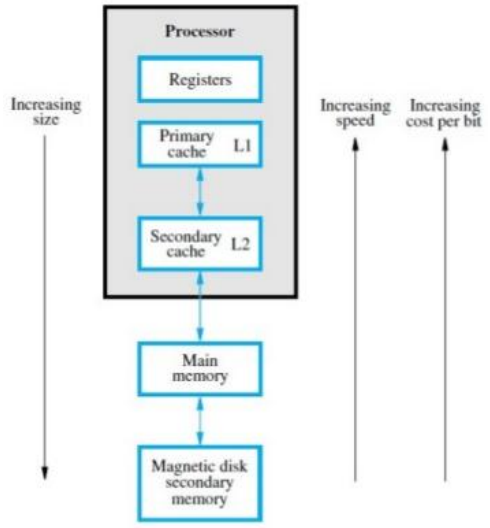
Flash memory is a type of electronically erasable programmable read only memory (EEPROM), but may also be a standalone memory storage device such as a USB drives. EEPROM is a type of data memory device using an electronic device to erase or write digital data. Flash memory is a distinct type of EEPROM, which is programmed and erased in large blocks.

Flash memory incorporates the use of floating-gate transistors to store data. Floating-gate transistors, or floating gate MOSFET (FGMOS), is similar to MOSFET, which is a transistor used for

amplifying or switching electronic signals. Floating-gate transistors are electrically isolated and use a floating node in direct current (DC). Flash memory is similar to the standard MOFSET, except the transistor has two gates instead of one.

SPEED, SIZE AND COST

A big challenge in the design of a computer system is to provide a sufficiently large memory, with a reasonable speed at an affordable cost.



Static RAM: Very fast, but expensive, because a basic SRAM cell has a complex circuit making it impossible to pack a large number of cells onto a single chip.

Dynamic RAM: Simpler basic cell circuit, hence are much less expensive, but significantly slower than SRAMs.

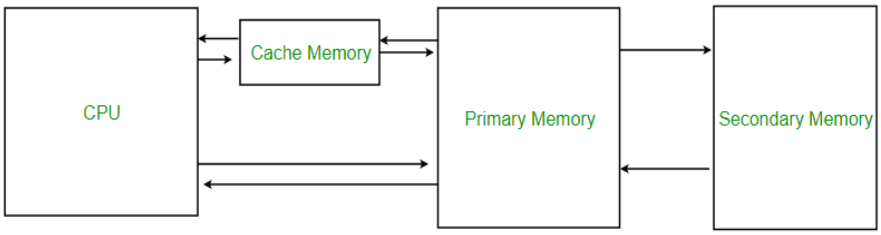
Magnetic disks: Storage provided by DRAMs is higher than SRAMs, but is still less than what is necessary. Secondary storage such as magnetic disks provides a large amount of storage, but is much slower than DRAMs.

CACHE MEMORIES

Processor is much faster than the main memory. As a result, the processor has to spend much of its time waiting while instructions and data are being fetched from the main memory. These create a major obstacle towards achieving good performance. Speed of the main memory cannot be increased beyond a certain point.

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.



Cache memory is based on the property of computer programs known as "locality of reference". Prefetching the data into cache before the processor needs it. It needs to predict processor future access requirement [Locality of Reference].

Locality of Reference

Analysis of programs indicates that many instructions in localized areas of a program areexecuted repeatedly during some period of time, while the others are accessed relatively less frequently. These instructions may be the ones in a loop, nested loop or few procedures calling each other repeatedly. This is called "locality of reference".

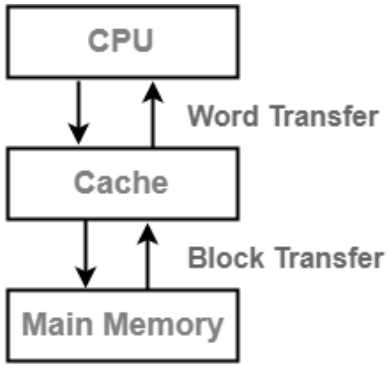
Temporal locality of reference:

Recently executed instruction is likely to be executed again very soon.

Spatial locality of reference:

Instructions with addresses close to a recently instruction are likely to be executed soon.

Basic Cache Operations



Processor issues a Read request; a block of words is transferred from the main memory to the cache, one word at a time. Subsequent references to the data in this block of words are found in the cache.

At any given time, only some blocks in the main memory are held in the cache, which blocks in the main memory in the cache is determined by a "**mapping function**".

When the cache is full, and a block of words needs to be transferred from the main memory, some block of words in the cache must be replaced. This is determined by a "**replacement algorithm**".

Cache hit

Existence of a cache is transparent to the processor. The processor issues Read and Write requests in the same manner. If the data is in the cache, it is called a Read or Write hit.

Read hit: The data is obtained from the cache.

Write hit: Cache has a replicaof the contents of the main memory. Contents of the cache and the main memory may be updated simultaneously.This is the **write-through protocol**. Update the contents of the cache, and mark it as updated by setting a bit knownas the **dirty bit** or modified bit. The contents of the main memory are updatedwhen this block is replaced. This is write-back or copy-back protocol.

Cache miss

If the data is not present in the cache, then a Read miss or Write miss occurs.

Read miss: Block of words containing this requested word is transferred from the memory. After the block is transferred, the desired word is forwarded to the processor.The desired word may also be forwarded to the processor as soon as it is transferred without waiting for the entire block to be transferred. This is called load-through or earlyrestart.

Write-miss: Write-through protocol is used, and then the contents of the main memory are

updated directly. If write-back protocol is used, the block containing the addressed word is first brought into the cache. The desired word is overwritten with new information.

MAPPING FUNCTIONS

The mapping functions are used to map a particular block of main memory to a particular block of cache. This mapping function is used to transfer the block from main memory to cache memory. Mapping functions determine how memory blocks are placed in the cache.

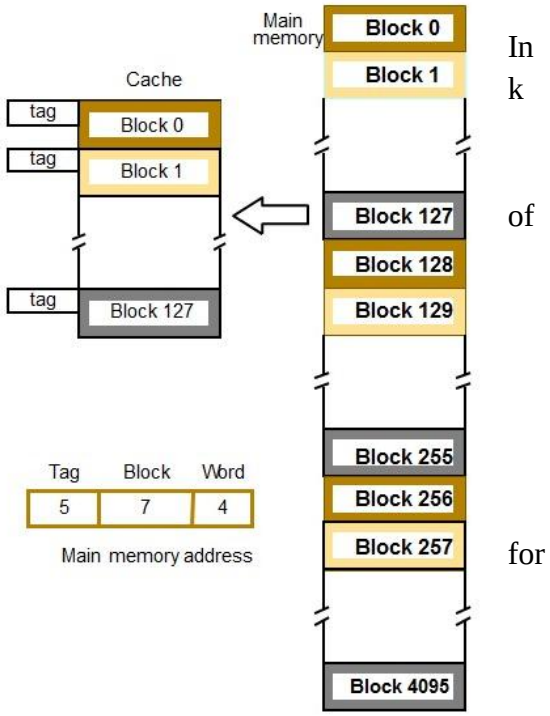
Three mapping functions:

- Direct mapping.
- Associative mapping.
- Set-associative mapping.

Direct Mapping

A particular block of main memory can be brought to a particular block of cache memory. So, it is not flexible.

The simplest way of associating main memory blocks with cache block is the direct mapping technique. In this technique, block k of main memory maps into block modulo m of the cache, where m is the total number of blocks in cache. In this example, the value of m is 128.



In direct mapping technique, one particular block of main memory can be transferred to a particular block of cache which is derived by modulo function.

Example: Block j of the main memory maps to j modulo 128 of the cache. (ie) Block 0, 128, 256 of main memory is maps to block 0 of cache memory.1,129,257 maps to 1, & so on.

More than one memory block is mapped onto the same position in the cache. This may lead to contention for cache blocks even if the cache is not full. Resolve the contention by allowing new block to replace the old block, leading to a trivial replacement algorithm.

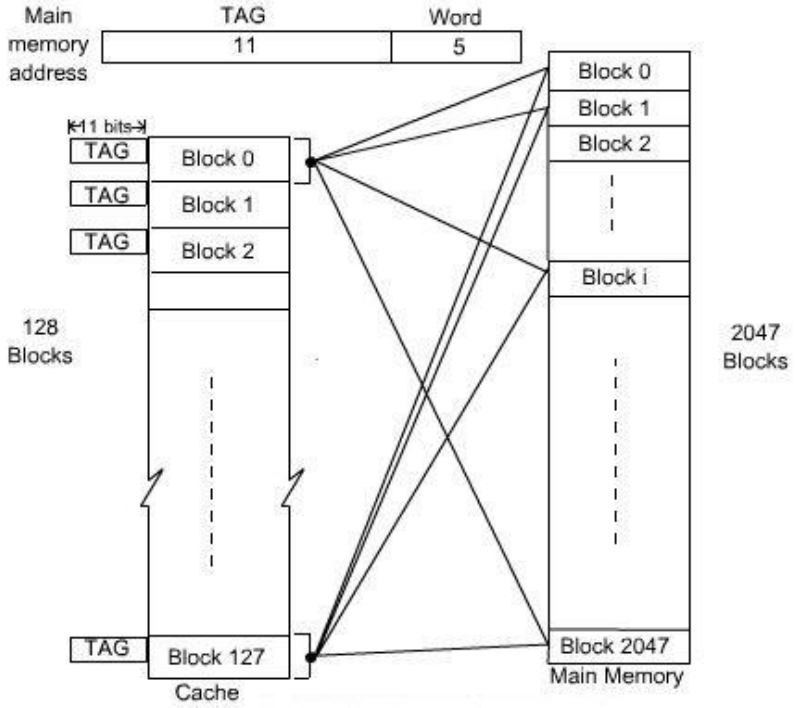
Memory address is divided into three fields: Low order 4 bits determine one of the 16 ***words*** in a block. When a new block is brought into the cache, the next 7 bits determine whichcache ***block*** this new block is placed in. High order 5 bits determine which of the possible32 blocks is currently presentin the cache. These are ***tag*** bits.

This mapping methodology is simple to implement but not very flexible.

Associative mapping

In the associative mapping technique, a main memory block can potentially reside in any cache block position. In this case, the main memory address is divided into two groups, a low-order bit identifies the location of a word within a block and a high-order bit identifies the block.

In the example here, 11 bits are required to identify a main memory block when it is resident in the cache , high-order 11 bits are used as TAG bits and low-order 5 bits are used to identify a word within a block. The TAG bits of an address received from the CPU must be compared to the TAG bits of each block of the cache to see if the desired block is present.



In the associative mapping, any block of main memory can go to any block of cache, so it has got the complete flexibility and we have to use proper replacement policy to replace a block from cache if the currently accessed block of main memory is not present in cache.

It might not be practical to use this complete flexibility of associative mapping technique due to searching overhead, because the TAG field of main memory address has to be compared with the TAG field of the entire cache block.

In this example, there are 128 blocks in cache and the size of TAG is 11 bits. The whole arrangement of Associative Mapping Technique is shown in the figure below.

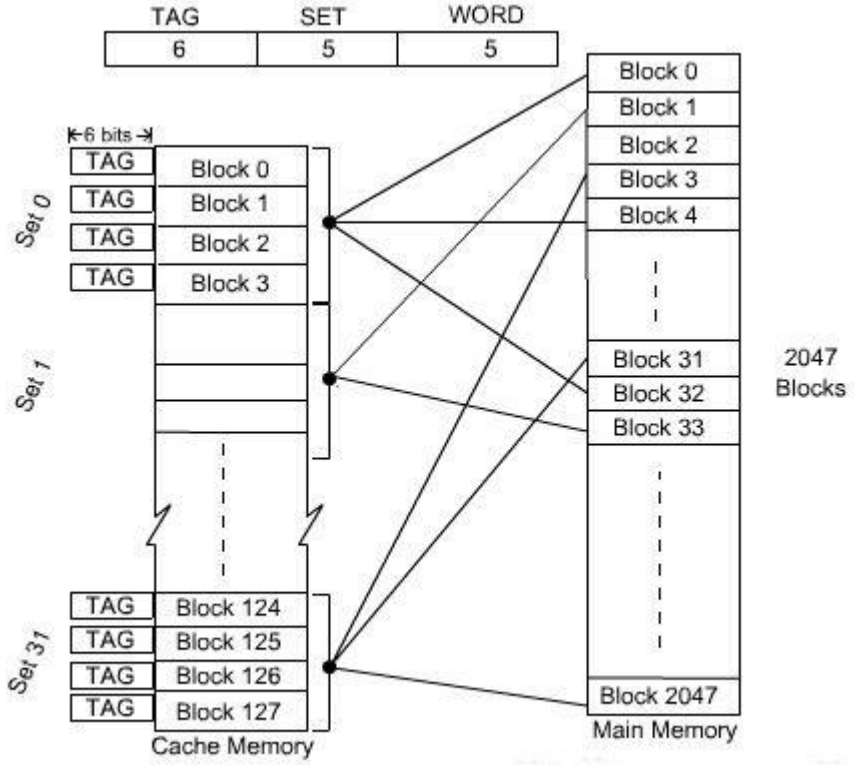
Set-Associative mapping

This mapping technique is intermediate to the previous two techniques. Blocks of the cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. Therefore, the flexibility of associative mapping is reduced from full freedom to a set of specific blocks.

This also reduces the searching overhead, because the search is restricted to number of sets, instead of number of blocks. Also the contention problem of the direct mapping is eased by having a few choices for block replacement.

Consider the same cache memory and main memory organization of the previous example. Organize the cache with 4 blocks in each set. The TAG field of associative mapping technique is divided into two groups, one is termed as SET bit and the second one is termed as TAG bit. Each set

contains 4 blocks, total number of set is 32. The main memory address is grouped into three parts: low-order 5 bits are used to identifies a word within a block. Since there are total 32 sets present, next 5 bits are used to identify the set. High-order 6 bits are used as TAG bits.



Replacement Algorithms

When the cache is full, there is a need for replacement algorithm for replacing the cache block with a new block. For achieving the high-speed such types of the algorithm is implemented in hardware.

In the cache memory, there are three types of replacement algorithm are used that are:

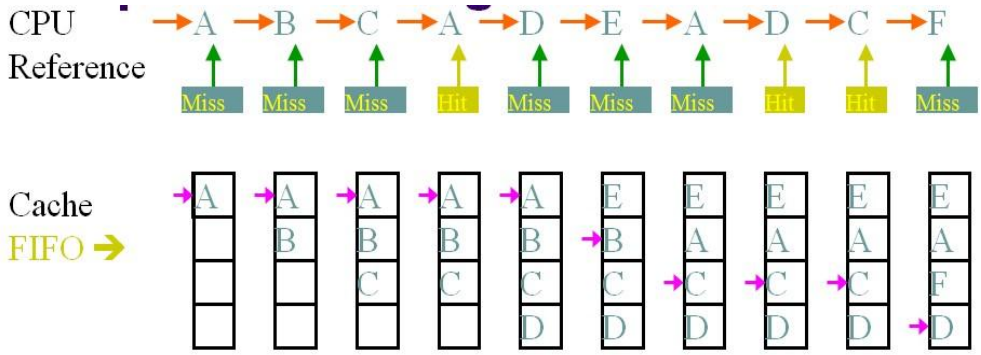
- Random replacement policy.
- First in first Out (FIFO) replacement policy
- Least recently used (LRU) replacement policy.

Random replacement policy

This is a very simple algorithm which used to choose the block to be overwritten at random. In this algorithm replace any cache line by using random selection. It is an algorithm which is simple and has been found to be very effective in practice.

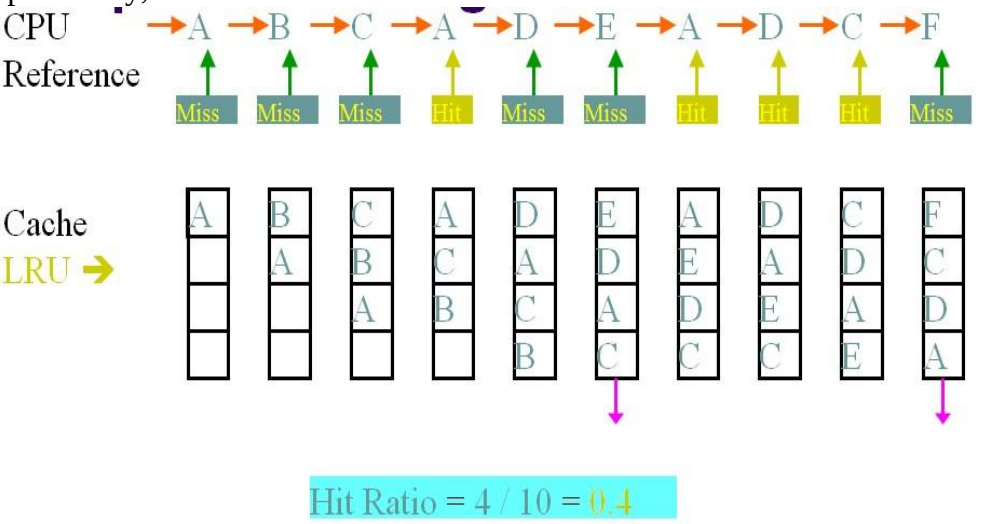
First in first out (FIFO)

In this algorithm replace the cache block which is having the longest time stamp. While using this technique there is no need of updating when a hit occurs but when there is a miss occur then the block is put into an empty block and the counter values are incremented by one.



Least recently used (LRU)

In the LRU, replace the cache block which is having the less reference with the longest time stamp. In this case also when a hit occurs when the counter value will be set to 0 but when the miss occurs there will be arising of two possibilities in which one possibility is that counter value is set as 0 and in another possibility, the counter value can be incremented as 1.



CONTENT ADDRESSABLE MEMORY (CAM)/ ASSOCIATIVE MEMORY

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. An account number may be searched in a file to determine the holder's name and account status.

The established way to search a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the

item and the efficiency of the search algorithm.

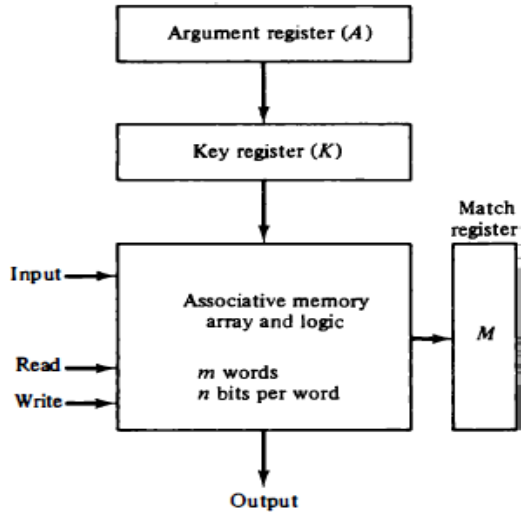
The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called **an associative memory or Content Addressable Memory (CAM).**

This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified.

The memory locaters all words which match the specified content and marks them for reading. Because of its organization, the associative memory isuniquely suited to do parallel searches by data association.An associative memory is more expensive then a random access memorybecause each cell must have storage capability as well as logic circuits for matching its content with an external argument.For this reason, associative memories are used in applications where the search time is very critical and must be very short.

HARDWARE ORGANIZATION

The block diagram of an associative memory consists of a memory array and logic from words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word.



Block Diagram of Associative Memory

The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register.

The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

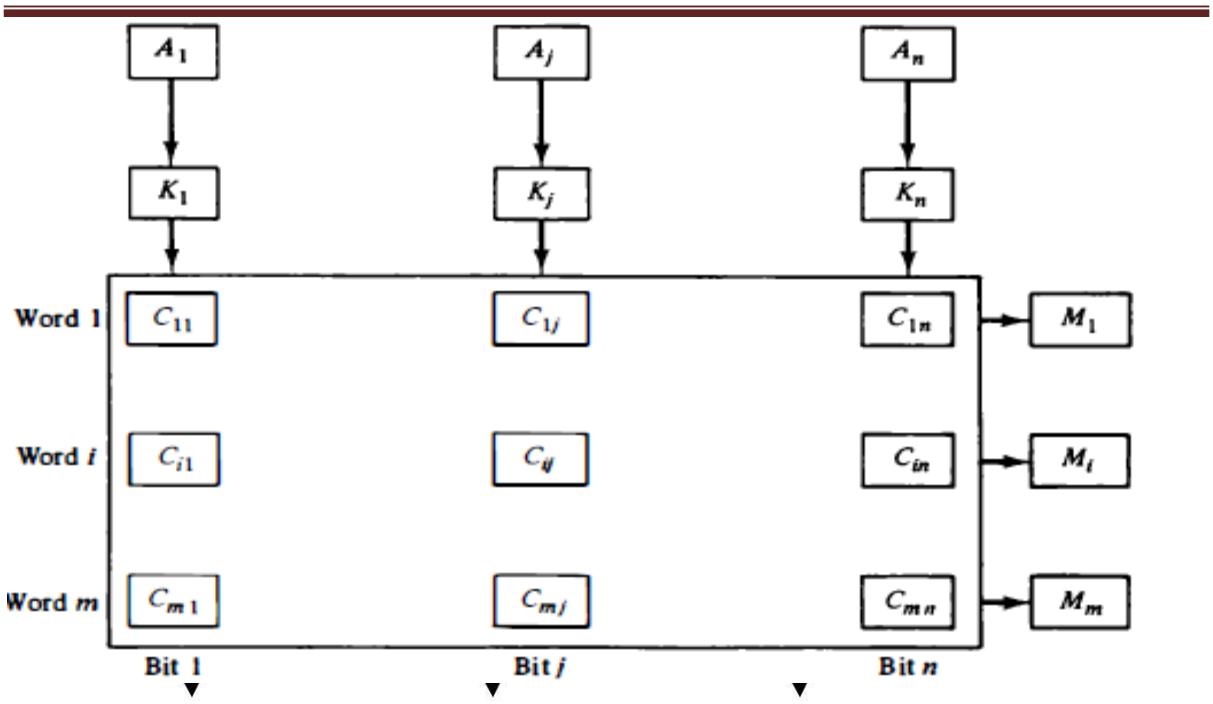
The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared.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

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

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in below figure.



The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell Cij is the cell for bit j in word i. A bit A j in the argument register is compared with all the bits in column j of the array provided that K j =1. This is done for all columns j = 1, 2,…,n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit Mi in the match register is set to 1.

If one or more unmasked bits of the argument and the word do not match, Mi is cleared to 0.

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

READ OPERATION

The matched words are read in sequence by applying a read signal to each word line whose corresponding Mi bit is a 1. In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output Mi directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines

and no special read command signal is needed. Furthermore, if we exclude words having a zero content, an all-zero output will indicate that no match occurred and that the searched item is not available inmemory.

WRITE OPERATION

If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of addresslines can be reduced by the decoder to d lines, where $m = 2^d$.

If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a tag register, would have as many bits as there are words in the memory. For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the newword is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this valueis used to specify an empty location.

CONTENT BEYOND SYLLABUS

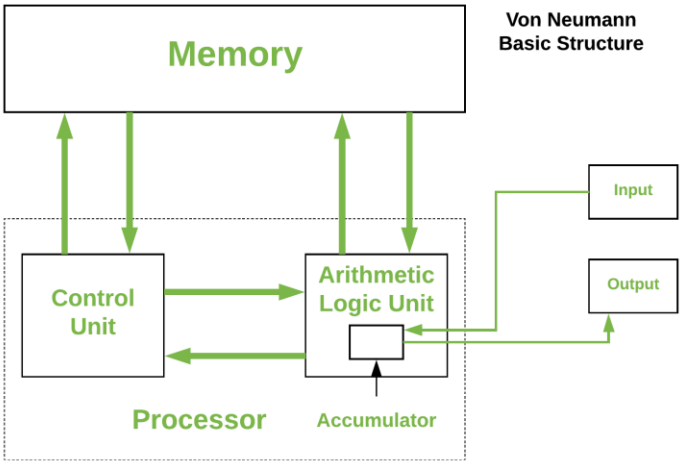
Von Neumann architecture

Historically there have been 2 types of Computers:

1. **Fixed Program Computers –** Their function is very specific and they couldn't be programmed, e.g. Calculators.
2. **Stored Program Computers –** These can be programmed to carry out many different tasks, applications are stored on them, hence the name.

The modern computers are based on a stored-program concept introduced by John Von Neumann. In this stored-program concept, programs and data are stored in a separate storage unit called memories and are treated the same. This novel idea meant that a computer built with this architecture would be much easier to reprogram.

The basic structure is like,



It is also known as **IAS** computer and is having three basic units:

1. The Central Processing Unit (CPU)
2. The Main Memory Unit
3. The Input/Output Device

Let's consider them in details.

- **Control Unit –** A control unit (CU) handles all processor control signals. It directs all input and output flow, fetches code for instructions, and controls how data moves around the system.
- **Arithmetic and Logic Unit (ALU) –** The arithmetic logic unit is that part of the CPU that handles all the calculations the CPU may need, e.g. Addition, Subtraction, Comparisons. It performs Logical Operations, Bit Shifting Operations, and Arithmetic operations.

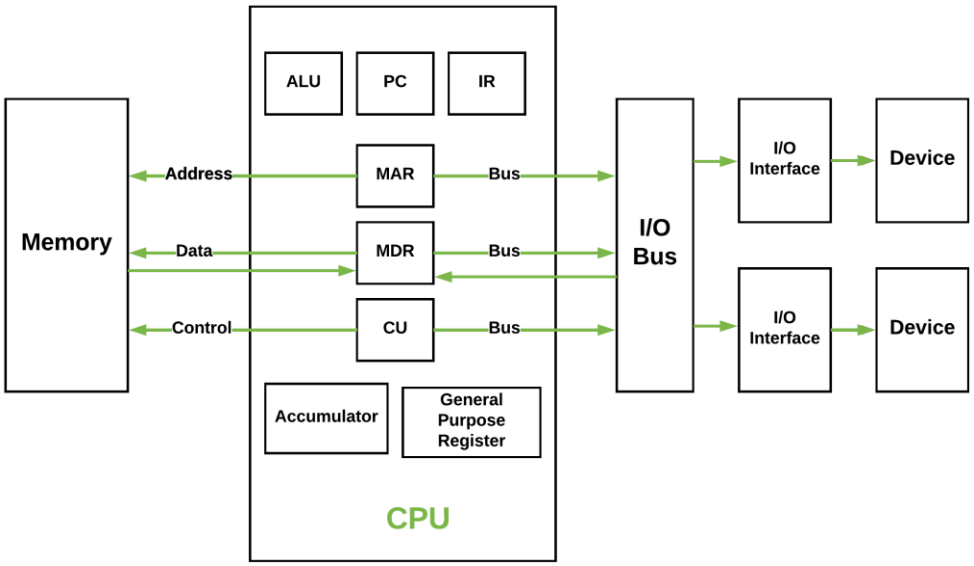


Figure – Basic CPU structure, illustrating ALU

- **Main Memory Unit (Registers) –**
  1. **Accumulator:** Stores the results of calculations made by ALU.
  2. **Program Counter (PC):** Keeps track of the memory location of the next instructions to be dealt with. The PC then passes this next address to Memory Address Register (MAR).
  3. **Memory Address Register (MAR):** It stores the memory locations of instructions that need to be fetched from memory or stored into memory.
  4. **Memory Data Register (MDR):** It stores instructions fetched from memory or any data that is to be transferred to, and stored in, memory.
  5. **Current Instruction Register (CIR):** It stores the most recently fetched instructions while it is waiting to be coded and executed.
  6. **Instruction Buffer Register (IBR):** The instruction that is not to be executed immediately is placed in the instruction buffer register IBR.

- **Input/Output Devices –** Program or data is read into main memory from the *input device* or secondary storage under the control of CPU input instruction. *Output devices* are used to output the information from a computer. If some results are evaluated by computer and it is stored in the computer, then with the help of output devices, we can present them to the user.
- **Buses –** Data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory, by the means of Buses. Types:
  1. **Data Bus:** It carries data among the memory unit, the I/O devices, and the processor.
  2. **Address Bus:** It carries the address of data (not the actual data) between memory and processor.
  3. **Control Bus:** It carries control commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer.

Von Neumann bottleneck – Whatever we do to enhance performance, we cannot get away from the fact that instructions

can only be done one at a time and can only be carried out sequentially. Both of these factors hold back the competence of the CPU. This is commonly referred to as the ‘Von Neumann bottleneck’. We can provide a Von Neumann processor with more cache, more RAM, or faster components but if original gains are to be made in CPU performance then an influential inspection needs to take place of CPU configuration.
This architecture is very important and is used in our PCs and even in Super Computers.

Computer Organization and Architecture | Pipelining

To improve the performance of a CPU we have two options:
1) Improve the hardware by introducing faster circuits.
2) Arrange the hardware such that more than one operation can be performed at the same time.

Since, there is a limit on the speed of hardware and the cost of faster circuits is quite high, we have to adopt the 2^{nd} option.

Pipelining : Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor.

Let us see a real life example that works on the concept of pipelined operation. Consider a water bottle packaging plant. Let there be 3 stages that a bottle should pass through, Inserting the bottle(**I**), Filling water in the bottle(**F**), and Sealing the bottle(**S**). Let us consider these stages as stage 1, stage 2 and stage 3 respectively. Let each stage take 1 minute to complete its operation.

Now, in a non pipelined operation, a bottle is first inserted in the plant, after 1 minute it is moved to stage 2 where water is filled. Now, in stage 1 nothing is happening. Similarly, when the bottle moves to stage 3, both stage 1 and stage 2 are idle. But in pipelined operation, when the bottle is in stage 2, another bottle can be loaded at stage 1. Similarly, when the bottle is in stage 3, there can be one bottle each in stage 1 and stage 2. So, after each minute, we get a new bottle at the end of stage 3. Hence, the average time taken to manufacture 1 bottle is :

Without pipelining = 9/3 minutes = 3m
I F S | | | | | |

| | | I F S | | |

| | | | | | I F S (9 minutes)

With pipelining = 5/3 minutes = 1.67m
I F S | |

| I F S |

| | I F S (5 minutes)

Thus, pipelined operation increases the efficiency of a system.

Design of a basic pipeline

- In a pipelined processor, a pipeline has two ends, the input end and the output end. Between these ends, there are multiple stages/segments such that output of one stage is connected to input of next stage and each stage performs a specific operation.
- Interface registers are used to hold the intermediate output between two stages. These interface registers are also called latch or buffer.
- All the stages in the pipeline along with the interface registers are controlled by a common clock.

Execution in a pipelined processor

Execution sequence of instructions in a pipelined processor can be visualized using a space-time diagram. For example, consider a processor having 4 stages and let there be 2 instructions to be executed. We can visualize the execution sequence through the following space-time diagrams:

Non overlapped execution:

Stage / Cycle	1	2	3	4	5	6	7	8
S1	I_1				I_2			
S2		I_1				I_2		
S3			I_1				I_2	
S4				I_1				I_2

Total time = 8 Cycle

Overlapped execution:

Stage / Cycle	1	2	3	4	5
S1	I_1	I_2			
S2		I_1	I_2		
S3			I_1	I_2	
S4				I_1	I_2

Total time = 5 Cycle

Pipeline Stages

RISC processor has 5 stage instruction pipeline to execute all the instructions in the RISC instruction set. Following are the 5 stages of RISC pipeline with their respective operations:

- **Stage 1 (Instruction Fetch)**
  In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.
- **Stage 2 (Instruction Decode)**
  In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.
- **Stage 3 (Instruction Execute)**
  In this stage, ALU operations are performed.

- **Stage 4 (Memory Access)**
  In this stage, memory operands are read and written from/to the memory that is present in the instruction.
- **Stage 5 (Write Back)**
  In this stage, computed/fetched value is written back to the register present in the instructions.

Performance of a pipelined processor

Consider a 'k' segment pipeline with clock cycle time as 'Tp'. Let there be 'n' tasks to be completed in the pipelined processor. Now, the first instruction is going to take 'k' cycles to come out of the pipeline but the other 'n – 1' instructions will take only '1' cycle each, i.e, a total of 'n – 1' cycles. So, time taken to execute 'n' instructions in a pipelined processor:

$$ET_{pipeline} = k + n - 1 \text{ cycles}$$
$$= (k + n - 1)\ Tp$$

In the same case, for a non-pipelined processor, execution time of 'n' instructions will be:

$$ET_{non\text{-}pipeline} = n * k * Tp$$

So, speedup (S) of the pipelined processor over non-pipelined processor, when 'n' tasks are executed on the same processor is:

S = Performance of pipelined processor /

Performance of Non-pipelined processor

As the performance of a processor is inversely proportional to the execution time, we have,

$$S = ET_{non\text{-}pipeline} / ET_{pipeline}$$
$$\Rightarrow S = [n * k * Tp] / [(k + n - 1) * Tp]$$
$$S = [n * k] / [k + n - 1]$$

When the number of tasks 'n' are significantly larger than k, that is, n >> k

$$S = n * k / n$$

$$S = k$$

where 'k' are the number of stages in the pipeline.

Also, **Efficiency** = Given speed up / Max speed up = S / S_{max}
We know that, Smax = k
So, **Efficiency** = S / k
Throughput = Number of instructions / Total time to complete the instructions
So, **Throughput** = n / (k + n – 1) * Tp

Systems I: Computer Organization and Architecture

Lecture 10: Microprogrammed Control

Microprogramming

- The control unit is responsible for initiating the sequence of microoperations that comprise instructions.
  - When these control signals are generated by hardware, the control unit is ***hardwired***.
  - When these control signals originate in data stored in a special unit and constitute a program on the small scale, the control unit is ***microprogrammed***.

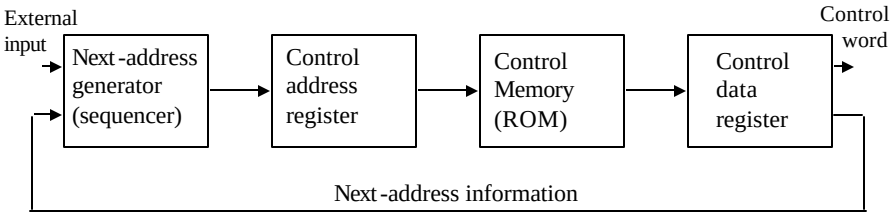
Control memory

- The control function specifying a microoperation is a binary variable whose active state could be either 1 or 0.
  - In the variable's active state, the microoperation is executed.
  - The string of control variables which control the sequence of microoperations is called a ***control word***.
- The microoperations specified in a control word is called a ***microinstruction***.
  - Each microinstruction specifies one or more microoperations that is performed.
- The control unit coordinates stores microinstruction in its own memory (usually ROM) and performed the necessary steps to execute the sequences of microinstructions (called microprograms).

The Microprogrammed Control Unit

- In a microprogrammed processor, the control unit consists of:
  - ***Control address register*** – contains the address of the next microinstruction to be executed.
  - ***Control data register*** – contains the microinstruction to be executed.
  - ***The sequencer*** – determines the next address from within control memory
  - ***Control memory*** – where microinstructions are stored.

Microprogrammed Control Organization



Sequencer

- The sequencer generates a new address by:
  - incrementing the CAR
  - loading the CAR with an address from control memory.
  - transferring an external address

or

  - loading an initial address to start the control operations.

Address Sequencing

- Microinstructions are usually stored in groups where each group specifies a routine, where each routine specifies how to carry out an instruction.
- Each routine must be able to branch to the next routine in the sequence.
- An initial address is loaded into the CAR when power is turned on; this is usually the address of the first microinstruction in the instruction fetch routine.
- Next, the control unit must determine the effective address of the instruction.

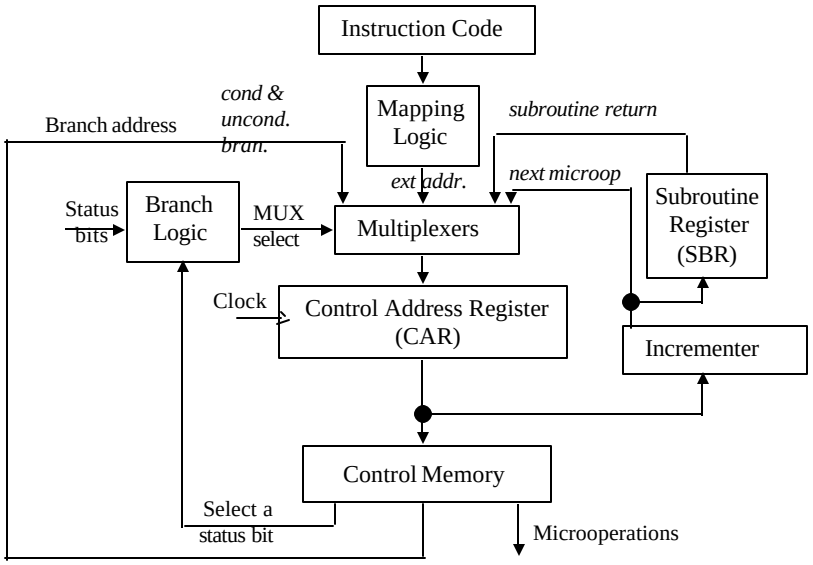
Mapping

- The next step is to generate the microoperations that executed the instruction.
  - This involves taking the instruction's opcode and transforming it into an address for the the instruction's microprogram in control memory. This process is called ***mapping***.
  - While microinstruction sequences are usually determined by incrementing the CAR, this is not always the case. If the processor's control unit can support subroutines in a microprogram, it will need an external register for storing return addresses.

Addressing Sequencing (continued)

- When instruction execution is finished, control must be return to the fetch routine. This is done using an unconditional branch.
- Addressing sequencing capabilities of control memory include:
  - Incrementing the CAR
  - Unconditional and conditional branching (depending on status bit).
  - Mapping instruction bits into control memory addresses
  - Handling subroutine calls and returns.

Selection Of Address For Control Memory



Conditional Branching

- Status bits
  - provide parameter information such as the carry-out from the adder, sign of a number, mode bits of an instruction, etc.
  - control the conditional branch decisions made by the branch logic together with the field in the microinstruction that specifies a branch address.

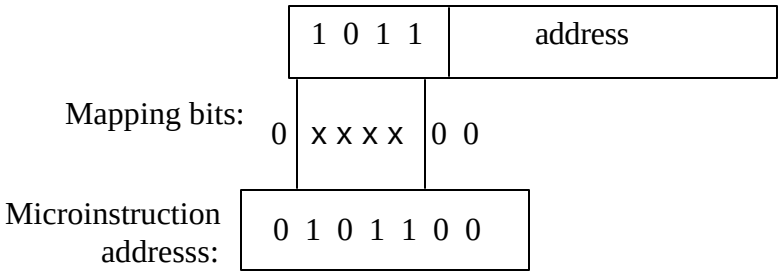
Branch Logic

- Branch Logic - may be implemented in one of several ways:
  - The simplest way is to test the specified condition and branch if the condition is true; else increment the address register.
  - This is implemented using a multiplexer:
    - If the status bit is one of eight status bits, it is indicated by a 3-bit select number.
    - If the select status bit is 1, the output is 0; else it is 0.
    - A 1 generates the control signal for the branch; a 0 generates the signal to increment the CAR.
- Unconditional branching occurs by fixing the status bit as always being 1.

Mapping of Instruction

- Branching to the first word of a microprogram is a special type of branch. The branch is indicated by the opcode of the instruction.
- The mapping scheme shown in the figure allows for four microinstruction as well as overflow space from 1000000 to 1111111.

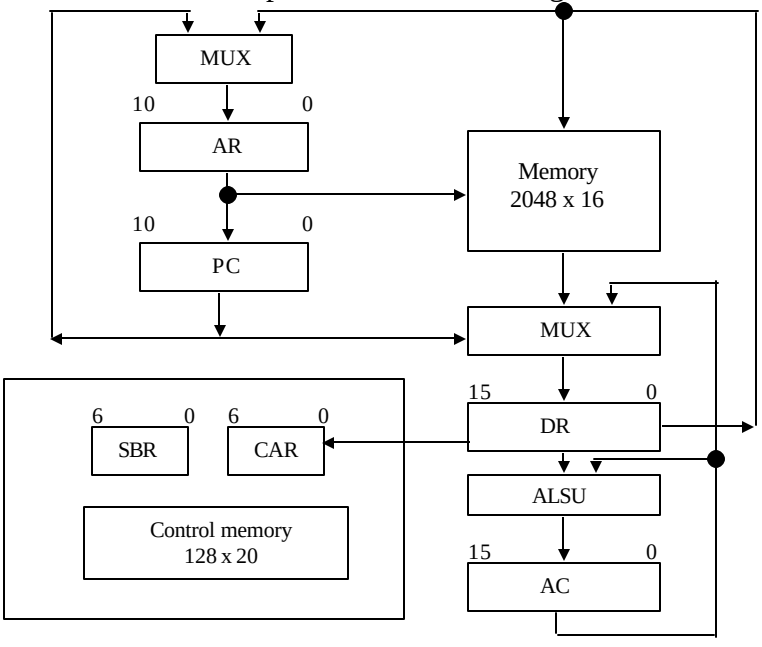
Mapping From Instruction Code To Microoperation Address



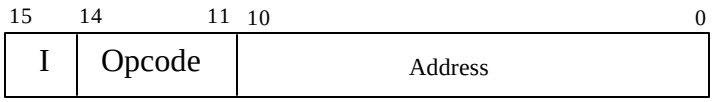
Subroutines

- Subroutine calls are a special type of branch where we return to one instruction below the calling instruction.
  - Provision must be made to save the return address, since it cannot be written into ROM.

Computer Hardware Configuration

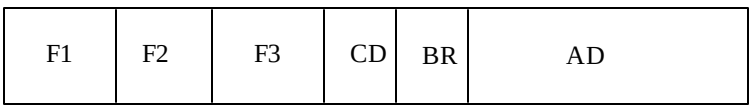


Computer Instructions



Symbol	**Opcode**	**Description**
ADD	0000	AC← AC + M[EA]
BRANCH	0001	IF (AC > 0) THEN PC ← EA
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA]← AC

Microinstruction Code Format (20 bits)



F1, F2, F3 : Microoperation Field

CD: Condition For Branching

BR: Branch Field

AD: Address Field

Symbols and Binary Code For Microinstruction Fields

F1	**Microoperation**	**Symbol**
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

Symbols and Binary Code For Microinstruction Fields (continued)

F2	**Microoperation**	**Symbol**
000	None	NOP
001	AC ← AC- DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

Symbols and Binary Code For Microinstruction Fields (continued)

F3	**Microoperation**	**Symbol**
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC'	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

Symbols and Binary Code For Microinstruction Fields (continued)

CD	**Condition**	**Symbol**	**Comments**
00	Always = 1	U	Unconditional Branch
01	DR(15)	I	Indirect Address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

Symbols and Binary Code For Microinstruction Fields (continued)

BR	**Symbol**	**Function**
00	JMP	CAR ←AR if condition = 1 CAR←CAR + 1 if condition = 0
01	CAL	CAR ←AR, SBR ← CAR + 1 if cond. = 1 CAR←CAR + 1 if condition = 0
10	RET	CAR ← SBR (return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0, 1, 6) ← 0

Symbolic Microinstructions

- It is possible to create a symbolic language for microcode that is machine-translatable to binary code.
- Each line define a symbolic microinstruction with each column defining one of five fields:
  - **Label** - Either blank or a name followed by a colon (*indicates a potential branch*)
  - **Microoperations** - One, Two, Three Symbols, separated by commas (*indicates that the microoperation being performed*)
  - **CD** - Either U, I, S or Z (*indicates condition*)
  - **BR** - One of four two-bit numbers
  - **AD** - A Symbolic Address, NEXT (address), RET, MAP (both of these last two converted to zeros by the assembler) (*indicates the address of the next microinstruction*)
- We will use the pseudoinstruction ORG to define the first instruction (or origin) of a microprogram, e.g., ORG 64 begins at 1000000.

Partial Symbolic Microprogram

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH

Partial Symbolic MicroProgram (continued)

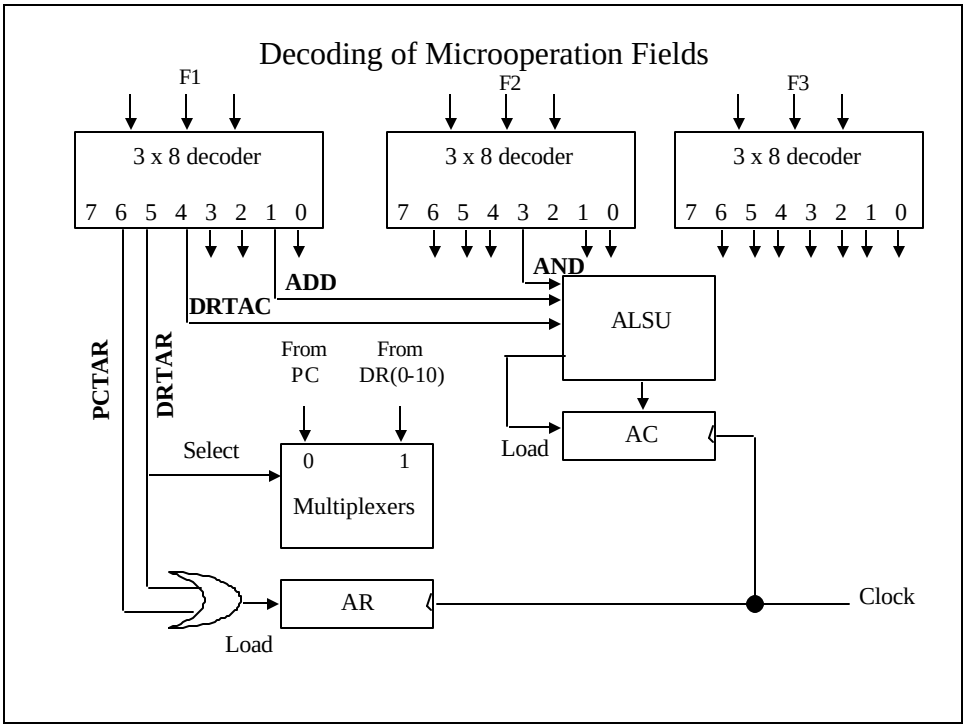
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ARTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAC	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAC	U	RET	

Partial Binary Microprogram

	Address		Binary Microinstruction					
Micro-Routine	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	000	000	000	00	00	1000001
	65	1000001	000	100	000	00	00	1000010
	66	1000010	000	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	000	000	000	00	10	0000000

Control Unit Design

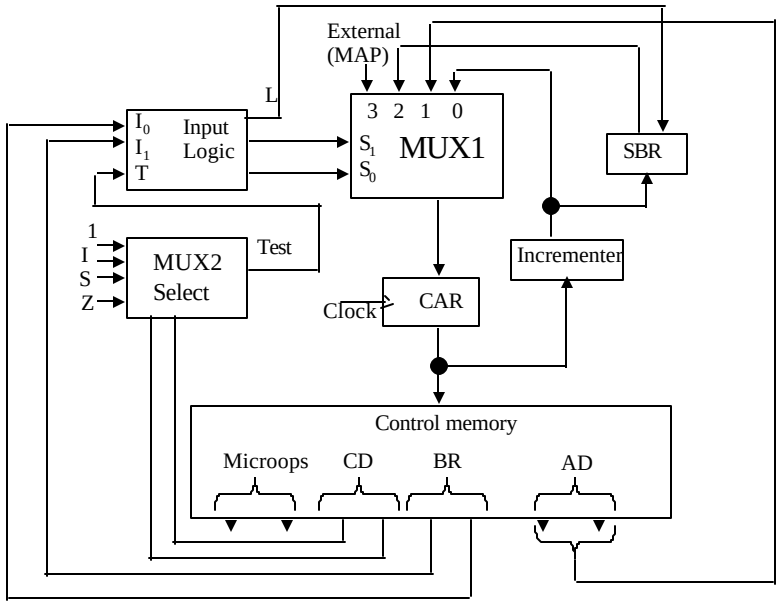
- Each field of k bits allows for 2^k microoperations.
- The number of control bits can be reduced by grouping mutually exclusive microoperations together.
- Each field requires its own decoder to produce the necessary control signals.



Microprogram Sequencer

- The microprogram sequencer selects the next address in control memory from which a microinstruction is to be fetched.
- Depending on the condition and on the branching type, it will be:
  - an external (mapped) address
  - the next microinstruction
  - a return from a subroutine
  - the address indicated in the microinstruction.

Microprogram Sequencer For A Control Memory



Input Logic Truth Table For A Microprogrammed Sequencer

BR Field		**Input**			**MUX 1**		**Load SBR**	
		I_1	**I_0**	**T**	**S_1**	**S_0**	**L**	
0	0	0	0	0	0	0	0	Next address
0	0	0	0	1	0	1	0	Specified addr.
0	1	0	1	0	0	0	0	
0	1	0	1	1	0	1	1	
1	0	1	0	x	1	0	0	Subroutine ret.
1	1	1	1	x	1	1	0	Ext. addr.

Name										
Reg. No.										

NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)

(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

SERIES TEST - 1 (2020-21)

Semester:IV	Programme: B.TECH	Max.Mark:30	Date:
Course Code & Name: CST 202 Computer Organization and Architecture		Duration:90 min	SET : 1
Knowledge Level (KL)	K1 : Remembering	K3:Applying	K5:Creating
Course Outcome(COL)	K2: Understanding	K4: Analysing	K6:Evaluation

PART-A

*Answer **ALL** Questions (3 x 3 = 9 Marks)*

1.	List out Memory Operations.	K4/CO1
2.	Define Memory Addressability? Classify different addressing modes.	K4,K6/CO1
3.	Explain Status Registers.	K6/CO2

PART- B

*Answer **ALL** Questions (3 x 7 = 21Marks)*

4 a.	Differentiate between Big endian and Little endian addressing **5 Marks**	K2/CO1
b.	List different instruction types. **2 Marks**	K4/CO1
	OR	
5	Explain single bus organization with neat diagram. **7 Marks**	K6/CO1

6	Write short notes on Register transfer logic. **7 Marks**	K4/CO2
	OR	
7	Explain Design of Logic Circuits **7 Marks**	K6/CO2

OR

8	Explain restoring method of Division **7 Marks**	K2/CO3
	OR	
9	Design 2x3 multiplier **7 Marks**	K2/CO3

Name										
Reg. No.										

Question paper quality assessment using Blooms taxonomy RUBRICS

Blooms taxonomy Definitions	Scale
Remembering	1
Understanding	2
Applying	3
Analyzing	4
Evaluating	5
Creating	6

Questions to Blooms taxonomy mapping

Qn No.	Marks	Remembering	Understanding	Applying	Analyzing	Evaluating	Creating
1	3		✓				
2	3	✓					
3	3						✓
4 a	5				✓		
4 b	2	✓					
5	7					✓	

Name										
Reg. No.										

6	7					✓	
7	7					✓	
8	7					✓	
9	7		✓				

EVALUATION OF QUALITY OF QUESTION PAPER USING BLOOMS TAXONOMY

Blooms taxonomy definitions	Scale	Marks	Rating (out of 6)
Remembering	**1**	**5**	3.90
Understanding	**2**	10	
Applying	**3**		
Analyzing	**4**	**5**	
Evaluating	**5**	**28**	
Creating	**6**	**3**	

CO MAPPING WITH QUESTIONS

Cos	T1	T2	T3	A1	A2
CST201.1	**Q(1),Q(2) Q(4),Q(5)**				
CST201.2	**Q(3),Q(6),Q(7)**				
CST201.3	**Q(8),Q(9)**				

Name										
Reg. No.										

CST201.4					
CST201.5					

APPROVED BY

MODULE CO-ORDINATOR **SCRUTINY COMMITTEE** **HOD**