

Computer Organization & Architecture

Dr.M. Ashok

Dr.J. Gladson Maria Britto

V. Sreetharan

A. Mohaideen

Published by



**Centivens Institute of
Innovative Research**
Private Limited

Computer Organization & Architecture

Copyright © 2023 by CIIR

All rights reserved. Authorized reprint of the edition published by CIIR. No part of this book may be reproduced in any form without the written permission of the publisher.

Limits of Liability/Disclaimer of Warranty: The authors are solely responsible for the contents of the paper in this volume. The publishers or editors do not take any responsibility for the same in any manner. Errors, if any, are purely unintentional and readers are required to communicate such errors to the editors or publishers to avoid discrepancies in future. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Further, reader should be aware that internet website listed in this work may have changed or disappeared between when this was written and when it is read.

CIIR also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.



ISBN 978-81-964673-2-6

Price: ₹449

Authors

Dr.M. Ashok | Dr.J. Gladson Maria Britto | V. Sreetharan | A. Mohaideen

Centivens Institute of Innovative Research

307, 5th Street Extension, Gandhipuram,

Coimbatore - 641 012,

Tamil Nadu, India.

E-mail: publications@centivens.com

Website: www.centivens.com

Author Details

Dr.M. Ashok,

Professor in CSE & Principal,
Malla Reddy College of Engineering,
Maissamaguda, via Dhulapally, Hyderabad.

Dr.J. Gladson Maria Britto,

Professor in CSE-DS & Head,
Malla Reddy College of Engineering,
Maissamaguda, via Dhulapally, Hyderabad.

V. Sreetharan,

Assistant Professor,
Department of CSE, St. Martin's Engineering College,
Dhulapally, Secunderabad.

A. Mohaideen,

Assistant Professor,
Department of CSE-DS, Malla Reddy College of Engineering,
Maissamaguda, via Dhulapally, Hyderabad.

Preface

The first chapter deals with Digital Computer, Definition of Computer Organization, and Computer Architecture. Register Transfer Language and Micro operations, Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic microoperations, shift microoperations, Arithmetic logic shift unit. Basic Computer Organization and Design, Instruction codes, Computer Registers Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt.

The second chapter deals with Microprogrammed Control: Control memory, Address sequencing, micro program example, design of control unit. Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control.

The Third chapter deals with Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation. Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating – point Arithmetic operations, Decimal Arithmetic unit, Decimal Arithmetic operations.

The fourth chapter deals with Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt Direct memory Access. Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

The fourth chapter deals with Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics. Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor. Multi Processors: Characteristics of Multiprocessors, Interconnection Structures, Interprocessor arbitration, Interprocessor communication and synchronization, Cache Coherence.

Acknowledgement

First and foremost, we thank, the Almighty God for endowing his immense blessing that helped us in each step of our progress towards the successful completion of this text book **“Computer Organization & Architecture”**.

It is our privilege to thank the management and the staff of the Department of Computer Science & Engineering (DS) of Malla Reddy College of Engineering & Malla Reddy Institute of Science and Technology for their constant support and cooperation to complete the text book.

Most importantly, none of this would have been possible without the love and patience of our family. Words alone cannot describe the encouragement, affection and our parent, wife and children; we thank them for all the help, care and moral support provided throughout this work.

Dr.M. Ashok

Dr.J. Gladson Maria Britto

V. Sreetharan

A. Mohaideen



This text book is devoted to Almighty God for making it all possible, our family members and our Gurus for their loving, encouraging and everlasting support.

TABLE OF CONTENTS

UNIT NO	TITLE	PAGE NO
1.	Basics of Computer Organization and Architecture	01
	1.1. Introduction	01
	1.2. Digital Computer Block Diagram	01
	1.3. Computer Generations	04
	1.4. Types of Computers	07
	1.5. Computer Organization	10
	1.6. Temporary Register Modifies Messages and Micro Functions	10
	1.7. Types of Micro-operations	18
	1.8. Basic Computer Organization and Design	30
	1.9. Addressing of Operand	32
	1.10. Computer Registers	33
	1.11. Instruction Format with its Types	36
	1.12. Basic Computer Input and Output Configuration	44
2.	Microprogrammed Control Unit	57
	2.1. Introduction	57
	2.2. Example of Microprogram	63
	2.3. Control Mechanism Layout	67
	2.4. Sequential Micro Program	68
	2.5. Central Processing Unit	70
	2.6. Storage Organization	71
	2.7. Formats of Instructions	77
	2.8. General Registration Organization	78
	2.9. RISC Instructions	81

UNIT NO	TITLE	PAGE NO
	2.10. Addressing Modes	82
	2.11. Data Transfer and Manipulation	85
3.	Data Representation	99
	3.1. Data Representation	99
	3.2. Data Types	99
	3.3. Number Systems	99
	3.4. Character Data Representation	100
	3.5. Conversions	103
	3.6. Other Base System to Decimal Conversion	104
	3.7. Binary Addition	106
	3.8. Binary Multiplication	107
	3.9. Binary Division	107
	3.10. Fixed-point Representation	108
	3.11. Floating-Point Representation	110
	3.12. IEEE Standard 754 Floating Point Standard	110
	3.13. Computer Arithmetic	114
	3.14. Floating Point Arithmetic Operations	124
4.	Input Output Organization	134
	4.1. Input/Output System Overview	134
	4.2. I/O Bus and Interface Modules	135
	4.3. Data Transfer Starts from the Target Unit	137
	4.4. Handshaking	137
	4.5. Asynchronous Serial Transmission	139
	4.6. Modes of Transfer	140
	4.7. Interrupt-Initiated I/O	140
	4.8. Direct Memory Access (DMA)	143

UNIT NO	TITLE	PAGE NO
	4.9. What is the Memory Hierarchy?	146
	4.10. Features of Memory Hierarchy	148
	4.11. Virtual Memory	149
5.	Introduction to Processors	160
	5.1. Reducing Instruction Issues in Computers: (RISC)	160
	5.2. Characteristics of CISC Processors	162
	5.3. Pipeline and Vector Processing	164
	5.4. Memory Interleaving	175
	5.5. Multiprocessors	177
	5.6. Interconnect Structure	179
	5.7. MultiPort Memory	180
	5.8. Crossbar Switch	180
	5.9. Cache Coherence	185
	5.10. Operating System Organization	185
	5.11. Communication and Synchronization	186

UNIT I - BASICS OF COMPUTER ORGANIZATION AND ARCHITECTURE

Digital Computer: Introduction, Digital Computer Block Diagram, Description of Computer Organization, Computer Design and Computer Architecture.

Scratchpad Transfer Language and Microoperation: Scratchpad Transfer Language, Scratchpad Transfer, Bus and Memory Transfer, Arithmetic Microoperations, Logical Microoperations, Transfer Microoperations, Arithmetic Transfer.

Computer Organization and Design: Code Instructions, Computer Registers, Computer Instructions, Time and Control, Circuit Instructions, Memory Instructions, Feedback - broadcast and Effect.

1.1. Introduction

A digital computer is characterized by its ability to execute mathematical operations rapidly. It is conceptualized as a versatile tool that functions as a means for processing information, akin to a document or file. Computers encompass a diverse array of operations designed to efficiently handle and manipulate data. These functions work collaboratively to carry out tasks and manage information effectively.

1.2. Digital Computer Block Diagram

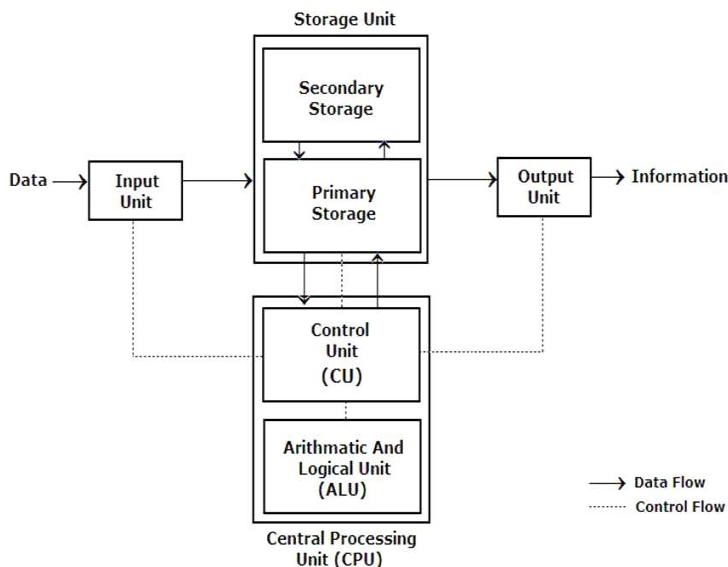


Figure 1.1: Block Diagram of Digital Computer

Input Unit

Computers necessitate the receipt of information and instructions to address problems and execute tasks. Input units, comprising various devices, facilitate this communication. Among these, the keyboard stands out as a widely utilized input device. Additionally, a range of common devices such as mice, scanners, and microphones are employed to gather information. The primary functions shared by all input devices encompass the retrieval of data and guidance from the external environment, enabling the computer to engage with and respond to input from the user or the surrounding world.

Storage Unit

The computer's storage unit retains the data and instructions provided by the input unit prior to processing. It serves as a repository for both interim and ultimate outcomes, preserving them until they are ready for output. Additionally, the storage unit functions to save data for future reference. The various elements of computer systems are categorized into two distinct groups.

Main Storage

This type of memory boasts rapid storage and retrieval processes and is primarily employed to store the actively running program, data received from the input unit, as well as the interim and ultimate outcomes of the program.

- Primary memory is of a transitory nature, implying its temporary status.
- Data becomes inaccessible or erased when the computer is powered off.
- The cost of storage holds greater significance in primary storage compared to secondary storage.

Auxiliary Storage

- Auxiliary storage serves a purpose akin to archiving, storing a diverse range of programs, files, and databases. The programs generated on the computer are transferred to the main memory before execution, and once the results are preserved, they are stored in secondary memory.
- Secondary memory exhibits a relatively slower speed and is more cost-effective compared to primary memory.
- Certain distinctive storage devices include hard drives, CDs, and similar mediums.

Memory Size

- All digital computers utilize a binary bit system, comprising 0s and 1s, with each character or number represented by an 8-bit code.

- A grouping of 8 bits is termed a byte, where one character occupies 1 byte and a number occupies 2 bytes, both serving as memory locations.
- The initial size is denoted in KB (kilobytes) or MB (megabytes), with 1 KB equivalent to 1024 bytes and 1 MB equalling 1000 KB.
- The size of primary storage in a PC typically commences at 16 MB, and PCs with memory sizes such as 32 MB, 48 MB, 128 MB, and 256 MB are commonplace.

Output unit

The computer's output unit furnishes external data and computed results, with printers and display units (VDU) frequently employed as output devices. Additionally, other commonly utilized devices for output purposes encompass speakers, headphones, projectors, and the like.

Arithmetic Logic Unit

All computations within the computer are carried out in the Arithmetic Logic Unit (ALU), which also handles comparisons and decision-making processes. The ALU is equipped to perform basic arithmetic operations such as addition, subtraction, multiplication, and division, as well as simple relational operations like $>$, $<$, $=$, and others. When a computation is required, the control unit transfers data from the storage unit to the ALU. Upon completion of the calculation, the results are transmitted from the control unit to the storage unit and subsequently to the output unit for display.

Control Unit

The central controlling entity in the computer oversees all other units. Upon receiving information from the user, this central control orchestrates the determination of where the data will be stored. It manages the data and instruction flow from the storage unit to the Arithmetic Logic Unit (ALU), as well as the transmission of values from the ALU back to the storage. Analogous to the central nervous system, the control centre is responsible for regulating and coordinating the computer's operations.

Central Processing Unit

The collaborative components of the computer, consisting of the control unit (CU) and arithmetic unit (ALU), are collectively referred to as the central processing unit (CPU). Functioning akin to a brain, the CPU carries out several key operations:

- It executes all mathematical calculations.
- It is responsible for decision-making processes.

- It exercises control over every unit within the computer.

1.3. Computer Generations

In the realm of computer terminology, a generation denotes a shift in the technology employed for computing purposes. Initially, distinctions were primarily centered around the diverse designs of hardware devices. However, the contemporary concept of a generation encompasses both hardware and software, constituting the entirety of a computer system. Currently, five distinct generations of computers are recognized, each marked by specific chronological periods and defining characteristics. The following delineates the accepted approximate dates and features associated with each generation.

First Generation

- Previous Period: 1946-1959.
- Vacuum tube technology
- Unreliable
- Supported machine language only.
- Very costly
- Generated lot of heat
- Slow input and output devices
- Huge size
- Need of A.C
- Non-portable
- Consumed lot of electricity

Some computers of this generation were:

- ENIAC
- EDVAC
- UNIVAC
- IBM-701
- IBM-650

Second Generation

- The period of second generation: 1959-1965.
- Employing transistors: Second-generation computers marked a shift from vacuum tube technology to the use of transistors.

- Enhanced reliability: In comparison to first-generation computers, second-generation machines exhibited increased reliability.
- Reduced size: These computers were smaller in size compared to their first-generation counterparts.
- Lower heat generation: Second-generation computers generated less heat than those of the first generation.
- Lower power consumption: They consumed less electricity than first-generation computers.
- Improved speed: Second-generation computers operated at a faster pace than their predecessors.
- Continued high cost: Despite advancements, computers of this generation remained relatively expensive.
- Requirement for air conditioning: Like their predecessors, second-generation computers necessitated the use of air conditioning.
- Support for machine and assembly languages: These computers could support both machine and assembly languages.
- Prominent computers of the second generation include:
 - IBM 1620
 - IBM 7094
 - CDC 1604
 - CDC 3600
 - UNIVAC 1108

Third Generation

The period of third generation: 1965-1971.

- IC used.
- More reliable in comparison to previous two generations
- Smaller size
- Generated less heat.
- Faster
- Lesser maintenance
- Still costly
- A.C needed.

- Consumed lesser electricity.
- Supported high-level language.

Some computers of this generation were:

- IBM-360 series
- Honeywell-6000 series
- PDP (Personal Data Processor)
- IBM-370/168
- TDC-316

Fourth Generation

The period of fourth generation: 1971-1980.

- VLSI technology used.
- Very cheap
- Portable and reliable
- Use of PC's
- Very small size
- Pipeline processing
- No A.C. needed.
- Concept of internet was introduced.
- Great developments in the fields of networks

Some computers of this Generation were:

- DEC 10
- STAR 1000
- PDP 11
- CRAY-1(Supercomputer)
- CRAY-X-MP (Supercomputer)

Fifth Generation

The period of fifth generation: 1980-onwards.

- ULSI technology
- Development of true artificial intelligence
- Development of Natural language processing
- Advancement in Parallel Processing

- Advancement in Superconductor technology
- More user-friendly interfaces with multimedia features
- Availability of very powerful and compact computers at cheaper rates.
- Some computer types of this generation are:
 - Desktop
 - Laptop
 - Notebook
 - Ultrabook

1.4. Types of Computers

Computers can be broadly categorized based on their speed and power consumption.

PC (Personal Computer)

A personal computer (PC) is a compact and affordable computing device primarily tailored for individual consumers. These computers leverage microprocessor technology, allowing companies to integrate the entire central processing unit (CPU) onto a single chip. Businesses extensively utilize PCs for tasks such as word processing, accounting, photo printing, and the execution of spreadsheets and information management systems.

While PCs are originally designed for single-user functionality, they are frequently interconnected to establish networks. At home, common computer uses include gaming and internet browsing. Notably, contemporary high-end Macintosh and PC models rival the computing power and graphics capabilities of lower-end workstations produced by companies like Sun Microsystems, Hewlett-Packard, and Dell.



Workstation

A workstation is employed for specialized applications such as engineering tasks (CAD/CAM), desktop publishing, and software development, among others. These computers are typically equipped with substantial features, including large, high-resolution displays, ample

RAM, network support, and advanced graphics capabilities. While most workstations include traditional storage devices like disk drives, there exists a variant known as a diskless workstation that lacks a disk drive.

Operating systems commonly used in workstations for professional settings are UNIX and Windows NT. Like personal computers (PCs), workstations are designed for single-user use, but they are frequently interconnected to establish local networks, enhancing collaborative capabilities.



Minicomputer

A minicomputer, falling between microcomputers and mainframes in terms of size and power, represents a medium-sized computing system. Possessing greater computational capabilities than microcomputers, minicomputers are frequently engineered to cater to the needs of multiple users concurrently through parallel processing. Despite being more potent, they come with a higher price tag compared to microcomputers. Illustrative examples of minicomputers include Digital Alpha and Sun Ultra.



Mainframe

A mainframe computer, distinguished by its extensive capacity and high-speed performance in contrast to smaller counterparts like microcomputers, is designed for robust computational tasks. Mainframes facilitate the concurrent usage of multiple terminals by numerous users,

making them suitable for applications like ATM transactions. Additionally, they serve as a centralized hub for the distribution of operational data.

Exemplary instances of mainframe computers include the IBM 370 and S/390 models.

Supercomputer

A supercomputer, distinguished by its exceptional capacity to store and process data at significantly accelerated rates compared to other computing systems, is gauged by its performance in tens of millions of instructions per second (mips). This level of computational prowess involves the execution of numerous instructions in rapid succession. Supercomputers find extensive application in tackling large-scale challenges in scientific and engineering domains, exemplified by tasks such as climate analysis. One notable example of a supercomputer is the IBM Deep Blue.



Computer Architecture

Computer architecture encompasses the arrangement of functionalities within a computer or processor. Physical memory components, such as Processor ISA (Instruction Set Architecture), the bit representation for data types, input and output signals, and memory addressing schemes, constitute key elements.

- It defines the interconnection of components to establish a computer system, serving as an amalgamation of both hardware and software.
- The study of computer architecture aids in comprehending the operational dynamics of the system, providing insights for programmers into its structure through instructions, address types, and data records.
- Addressing high-level design issues, computer architecture delves into logic aspects such as instruction settings, address types, file configurations, and cache optimization.

1.5. Computer Organization

Computer organization is derived from the principles outlined by computer architecture, focusing on the interconnection of functional components to fulfil the specifications set by the architectural design. This entails the arrangement and coordination of hardware elements, control symbols, and devices within the system.

For Example

Imagine you're part of a car manufacturing company. In this analogy, the design of the car and its individual components corresponds to computer architecture, offering an abstract and programmer's perspective. As these components are manufactured individually and interconnected based on the design, they contribute to the computer organization, representing the physical and visual manifestation.

- Computer organization encompasses the structure and functionality of computers from the standpoint of system users, detailing the physical connections within the system.
- It provides a comprehensive understanding of how all units in a system are structured and interconnected, guiding the utilization of organizational architecture.
- Essentially, computer organization addresses low-level design challenges, particularly those related to the physical aspects of equipment, including circuit design, connectors, signals, and peripherals.

1.6. Temporary Register Modifies Messages and Micro Functions

Temporary Register

A special and swift memory location within the CPU is known as a register. Registers play a crucial role in processing information effectively, functioning like scrapbooks where all pertinent data must be represented. For instance, when comparing two numbers for equality, both numbers need to reside in registers, and the result is likewise stored in a register. It's important to note that a register, acting as a scratchpad, may contain the address of the memory location where data is stored, rather than the actual data.

The power and speed of the CPU are influenced by the number of registers it possesses and the size, measured in bits, of each register. In practical terms, a 32-bit CPU has registers that are 32 bits wide, enabling all CPU instructions to operate on 32-bit data. In high-level programming languages, compilers play a pivotal role in translating high-level functions into low-level functions, facilitating access to temporary registers.

Register Transfer Language

Register Transfer Language (RTL) serves as an illustrative example employed for accessing temporary registers and describing the transport and manipulation of microprogram operations. Examining the internal hardware organization of a digital computer reveals key aspects:

- The term "key register" signifies the presence of logic circuits capable of executing micro-operations and transferring the resulting data to the same or another register.
- A collection of microprocesses is engaged in manipulating binary data stored within a temporary register.
- The nomenclature is borrowed from programming languages that utilize similar terminology.
- Control mechanisms are in place to initiate sequences of micro-operations, orchestrating the execution of tasks at the microprogram level.

Register Transfer

Many computer registers are denoted by uppercase letters, occasionally accompanied by numbers indicating specific register operations. Examples include the Memory Address Register (MAR), responsible for storing memory addresses, the Program Counter (PC), the Instruction Register (IR), and Processor Register (R1).

The elements within a scrapbook are typically sequentially numbered from right to left, starting from 0.

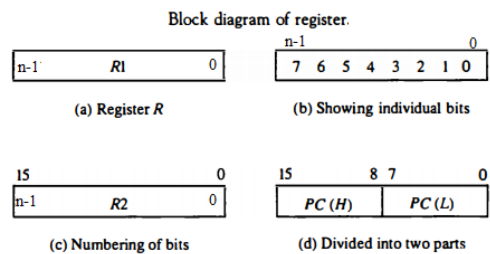


Figure 1.2: Accessing the General Registers

Figure 1.1 illustrates various representations of records.

Figure 1.1(a) depicts one approach to portraying a scratch as a rectangular box with the name of the scratch inside.

Figure 1.1(b) showcases how each record number is distinct.

In the 16-bit scratchpad, the small number is typically placed at the top of the box, as seen in Figure 1.1(c).

For a 16-bit register, the string is often divided into two parts, as demonstrated in Figure 1.1(d). Bits 0 through 7 are labelled as the Low Byte (L), while bits 8 through 15 are designated as the High Byte (H). An example is the Program Counter (PC), where PC (0-7) or PC(L) signifies the low byte, and PC (8-15) or PC(H) represents the high byte.

Some important symbols for changes in the scrapbook are shown in *Table 1.1*.

Table 1.1: Important Symbols

Symbol	Description	Example
Letters and numbers	represent	MAR and R 2
Parentheses ()	represent some of the labels	R2 (8-15) and R2 (L)
Arrows	represent the message being sent	$R2 \leftarrow R1$
Comma	separates two microtransactions	$R2 \leftarrow R1, R1 \leftarrow R2$
Colon:	Indicates the message sent under predet. ermined control conditions	P: $R2 \leftarrow R1$

Most scrapbooks are denoted by uppercase letters and may be appended with numbers, as illustrated in the initial row of the table.

- Square brackets serve to represent a segment of the register, providing additional specifications or labelling, as demonstrated in the second row of the table.
- Arrows signify data transmission and the direction of transmission, as indicated in the third, fourth, and fifth columns of the table.
- Commas are utilized to separate concurrent operations, as shown in the fourth column of the table.
- Colons are employed to delineate control aspects, as evidenced in the fifth column of the table.
- The expression $R2 \leftarrow R1$ denotes the transfer of contents from temporary register R1 to R2, with R1's contents remaining unaltered.
- The expression $R2 \leftarrow R1, R1 \leftarrow R2$ signifies the exchange functionality.
- The expression P: $R2 \leftarrow R1$ denotes the transfer of temporary register R1's contents to R2, contingent on the control function P being equal to 1. In such instances, the temporary storage device R1 remains unaffected.

Figure 1.2 presents a schematic illustrating the data transfer from register R1 to R2, with control function P set to 1. The n outputs of register R1 are linked to the n inputs of register R2. Register R2 features a load input activated by the control variable P, with the assumption that control variables are synchronized with the clock used for recordings. The time diagram illustrates that P initiates at time t, coinciding with the rising edge of the clock in the control section. At the subsequent clock tick at time t+1, the input data from R2 is loaded into the scratchpad in parallel when the load input is valid. P may revert to 0 at time t+1; otherwise, transitions occur during every active period of P.

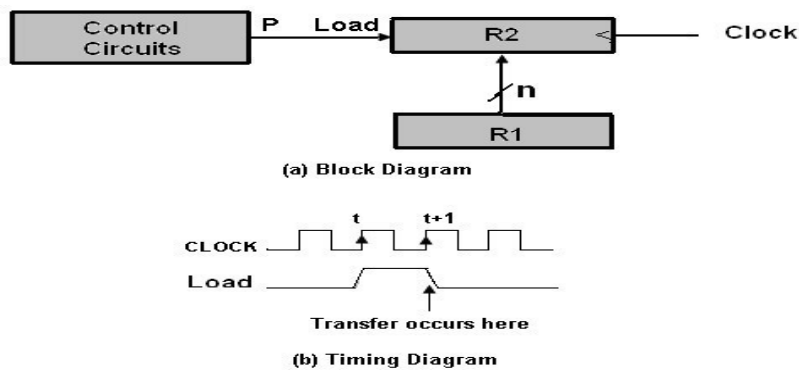


Figure 1.3: Transfer From R1 to R2 When p=1

- The transfer of information from one register to another is symbolized using a replacement operator.
- The statement "R2 $\leftarrow$ R1" signifies the transfer of the content from register R1 into register R2.
- This operation implies a replacement of the content in R2 with the content from R1.
- Importantly, the content of the source register, R1, remains unchanged after the transfer.
- To conditionally execute the transfer based on a predetermined control condition, an if-then statement can be employed: "if (P=1) then R2 $\leftarrow$ R1," where P is a control signal generated by a control section.
- Control variables can be separated from the register transfer operation by specifying a Control Function, a Boolean variable taking values 0 or 1.
- The control function is integrated into the statement, as in "P: R2 $\leftarrow$ R1," and the transfer operation is executed by the hardware only if P equals 1.
- Each statement in register transfer notation inherently implies a hardware construction for implementing the transfer.

- Figure 1.2 illustrates a block diagram depicting the transfer from R1 to R2, where the n outputs of register R1 are connected to the n inputs of register R2.
- The letter "n" denotes any number of bits for the register, to be replaced by an actual number when the register length is known.
- Register R2 features a load input activated by the control variable P, and it is assumed that the control variable is synchronized with the clock applied to the register.
- In the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t .
- The subsequent positive transition of the clock at time $t + 1$ activates the load input, leading to the parallel loading of data inputs from R2 into the register.
- P may revert to 0 at time $t + 1$; otherwise, the transfer occurs with each clock pulse transition while P remains active.
- Despite the control condition, such as P becoming active just after time t , the actual transfer does not transpire until the register is triggered by the next positive transition of the clock at time $t + 1$.

Bus and Memory Transfers

- A more effective method for transferring information across registers within a multi-register setup is through the implementation of a Common Bus System.
- This system incorporates a set of common lines, each dedicated to a specific bit within a register.
- Control signals play a pivotal role in determining which register is chosen by the bus during each specific register transfer operation.
- Various approaches exist for constructing a Common Bus System, including the utilization of Multiplexers and Tri-state Buffers.
- Employing Multiplexers is one technique for implementing a Common Bus System.
- Another approach involves the use of Tri-state Buffers to establish a Common Bus System.

Common Bus System is With Multiplexers

- Multiplexers are instrumental in the process of selecting the source register, thereby facilitating the transfer of its binary information onto the bus.
- The configuration of a bus system designed for four registers is depicted in the illustration below (Figure).

Constructing A Common Bus System Using Multiplexers

A Common Bus System can be constructed using multiplexers, in which, these. multiplexers select the source register whose information are placed on the bus.

A Common Bus System can be established using multiplexers, where these devices selectively pick the source register, and their information is subsequently transmitted onto the bus. In Figure 5, a depiction of a four-register bus system is presented. The bus comprises four 4 x 1 multiplexers, each equipped with four data inputs labelled 0 through 3, and two selection inputs denoted as S1 and S0.

For instance, the output 1 of register A is linked to input 0 of MUX 1, as this input is designated A1. The diagram illustrates the connection of bits in the same significant position from each register to the data inputs of a specific multiplexer, forming one line of the bus. MUX 0 manages the four 0 bits, MUX 1 handles the four 1 bits, and so forth.

Figure shows a four-register bus system for four registers.

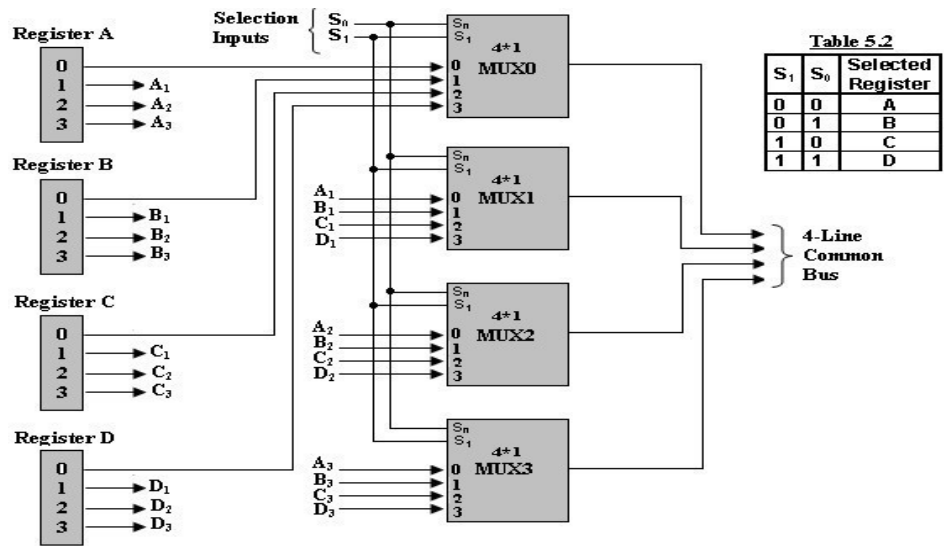


Figure 1.4: Common Bus System Using Multiplexer

The selection lines S1 and S0 are linked to the selection inputs of all four multiplexers. These lines determine the selection of four bits from one register, transferring them into the four-line common bus. When S1S0= 00, the 0 data inputs of all four multiplexers are chosen, incorporating the content of register A into the bus lines. A similar mechanism applies for registers B, C, and D with different combinations of S1S0.

Table 1.3 outlines the register selected by the bus for each of the four possible binary values of the selection lines.

It multiplexes "k" Registers, each having "n" bits, to produce an "n-line bus."

The required number of multiplexers equals n, with each multiplexer's size being k.

When the bus is incorporated into a statement, the register transfer is symbolized as follows:

$$\text{BUS} \leftarrow C, R1 \leftarrow \text{BUS}$$

This signifies that the content of register C is placed on the bus, and the content of the bus is loaded into register R1 by activating its load control input. Alternatively, if the bus is already known to exist in the system, a direct transfer can be represented as:

$$R1 \leftarrow C$$

Constructing a Common Bus System using Three-State Bus Buffers

A Three-State gate is a digital circuit that operates in three states, with two being the standard logic states, 0 and 1, and the third being a high-impedance state, which behaves akin to an open circuit. In this state, the output is cut off and lacks a defined logical meaning. The graphical symbol and logic for a three-state buffer gate are depicted in Figure 1.4.

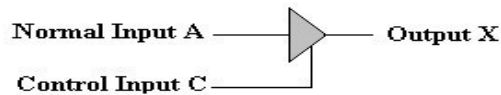


Figure 1.5: Diagram Demonstrates the Construction of a Bus System with Three-state Buffers

Table 1.2: Function Table

Control Input C	Control Input C	Gate Output X
0	0	High Impedance
0	1	High Impedance
1	0	0
1	1	1

Bus systems can be effectively constructed using three-state gates instead of multiple gates. These gates exhibit three states, with the high-impedance state providing the advantage of connecting multiple gates to form a bus without posing a loading risk.

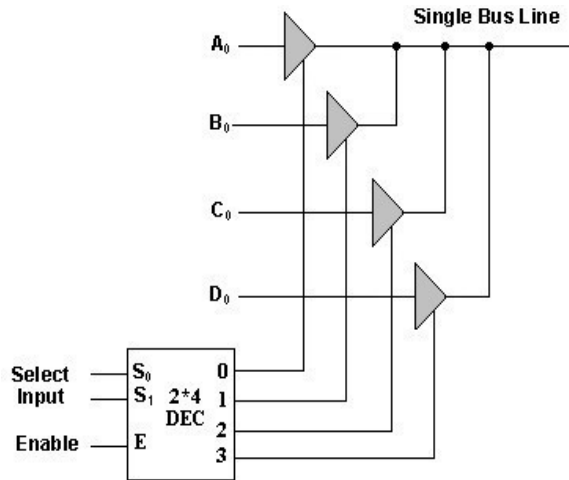


Figure 1.6: Bus Line with Three State Buffers

Unlike regular gates, three-state gates feature both an input and a control input, managing the decision-making process. When the control input is set to 1, the output is activated, and the gate functions as a buffer, transmitting the input value to the output.

Conversely, when the control input is 0, the output is disabled, and the gate assumes a high-impedance state, regardless of the input value.

The model of a three-state bus is illustrated in Figure 1.5, where the outputs of four buffers are interconnected to create a single bus line. The control inputs to these buffers determine which of the four normal inputs communicates with the bus line. It's crucial to ensure that only one three-state buffer is active at any given time to prevent conflicts. One method to achieve this is by using a decoder, as shown in the diagram.

When the enable input of the decoder is 0, all four outputs are 0, leading to a high-impedance state on the bus line. Conversely, when the enable input is active, one of the three-state buffers becomes active based on the binary value in the select inputs of the decoder. This mechanism guarantees that only one control input is active at any given time.

Memory Transfer

The process of extracting information from a memory word and sending it to the external environment is referred to as a read operation.

Conversely, when new information is intended to be stored in the memory, this action is termed a write operation.

Let the letter M symbolize a memory word, and during the transfer, a specific memory word is chosen among the available ones using the memory address.

For write memory transfer operations, it is imperative to specify the address of M. This is achieved by enclosing the address in square brackets following the letter M.

Imagine a memory unit receiving the address from a register known as the address register, denoted by AR. The data are then transferred to another register, called the data register, symbolized by DR.

The read operation can be expressed as follows: Read: $DR \leftarrow M[AR]$. This command initiates the transfer of information into DR from the memory word M selected by the address in AR.

In contrast, the write operation involves transferring the content of a data register to a specific memory word M, chosen by the address. Assuming the input data are in register R1, and the address is in AR, the write operation is articulated as follows: Write: $M[AR] \leftarrow R1$. This command facilitates the transfer of data from register R1 to the memory word M selected by the address in AR.

1.7. Types of Micro-operations

Register Transfer Micro-operations

Moving binary information from one register to another involves a Register Transfer Micro-operation. It's essential to note that this operation doesn't alter the information content during the transfer; it merely shifts the binary data from a source register to a destination register.

In contrast, three other types of micro-operations are distinct as they modify the information content during the transfer. These operations bring about changes in the data being transferred, distinguishing them from the straightforward Register Transfer Micro-operation, which maintains the integrity of the information being moved.

Arithmetic Microoperations

Conducting arithmetic operations involves manipulating numerical data stored in temporary registers. This process entails performing mathematical calculations, such as addition, subtraction, multiplication, or division, on the data residing in these registers. The temporary registers serve as a temporary storage location for numerical values, facilitating efficient computation without altering the original data.

Table 1.3: Symbolic Representation of Various Arithmetic Micro-operations

Symbolic Designation	description					
$R3 \rightarrow R1 + R2$	Contents of R1 plus R2 transferred to R3					
$R3 \rightarrow R1 - R2$	Contents of R1 minus R2 transferred to R3					
$R2 \rightarrow R2$	Complement complement)	the	contents	of	R2	(1's
$R2 \rightarrow R2 + 1$	2's complement the contents of R2 (negate)					
$R3 \rightarrow R1 + R2 + 1$	R1 plus the 2's complement of R2 transferred to R3(subtraction)					
$R1 \rightarrow R1 + 1$	Increment the contents of R1 by 1					
$R1 \rightarrow R1 - 1$	Decrement the contents of R1 by 1					

- Logical micro processing involves the manipulation of individual bits of data stored in a temporary register.
- Transfer microtransactions perform operations on data residing in a temporary list.

In logical micro processing, the focus is on processing the fundamental binary units (bits) within a temporary register, while transfer microtransactions execute operations on data contained in a temporary list.

Arithmetic Micro-operations

The basic arithmetic micro-operations are,

- Addition
- Subtraction
- Increment
- Decrement
- Shift

The arithmetic microoperation expressed as " $R3 \leftarrow R1 + R2$ " signifies the addition of the content in register R1 to the content in register R2, with the result transferred to register R3.

In practical terms, to execute addition using these instructions, the operands must be designated by three-letter codes, and the numerical values undergo addition.

For subtraction operations, a common technique involves converting it into an addition operation with an adjustment:

$$"R3 \leftarrow R1 + R2 + 1"$$

To perform subtraction, this expression signifies adding the complement of R2 (obtained by subtracting each bit from 1) to R1, effectively achieving the subtraction of R2 from R1.

Another method to express subtraction is by utilizing the 2's complement:

$$R2 \leftarrow 1's \text{ complement of } R2$$

$$R3 \leftarrow R1 + (R2 + 1)$$

In this context, subtracting 1 from 2's complement results in adding the complement of R2, achieving the subtraction of R2 from R1.

Binary Adder

The architecture of this adder comprises four full adders connected in a cascade, where the output carry of one full adder is linked to the carry input of the subsequent adder. The corresponding product of A and the complementary product of B serves as the continuous inputs for this arrangement. The initial carry bit for the binary adder is denoted as C0, and the resulting output carry bit is designated as C4. The sum bit (S) output from the full adder represents the arithmetic sum.

- A full adder is a digital circuit designed to calculate the arithmetic sum of two bits, including the carry bits.
- The term "binary adder" refers to a digital circuit capable of determining the arithmetic sum of two binary numbers, regardless of their length.

Figure 1.6 illustrates the interconnection of four full adders (FA) to create a 4-bit binary adder.

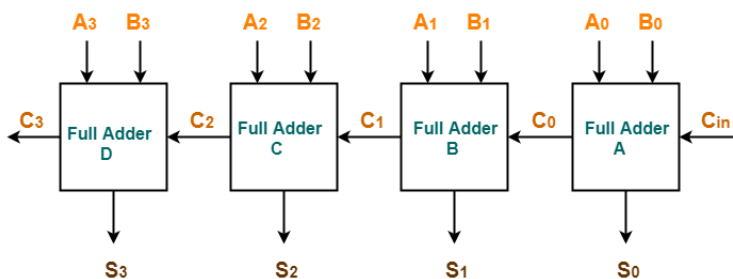


Figure 1.7: 4 Bit Binary Adder

- A's sum and B's complement are arranged in descending order from right to left, with subscript 0 indicating the least significant bit.
- The linkage of all the carry outputs forms a chain. The input carry bit for the binary adder is denoted as C₀, and the resultant output carry bit is represented as C₄. The sum bits are generated at the S output of the full adder.

- For an n-bit binary adder, the necessity is for employing n full adders to accomplish the summation operation.

Binary Adder – Subtractor

The capability to perform both addition and subtraction is achieved by incorporating XOR gates into each full adder, allowing for their integration into a unified circuit.

Figure 1.7 illustrates the configuration of a 4-bit adder-subtractor circuit.

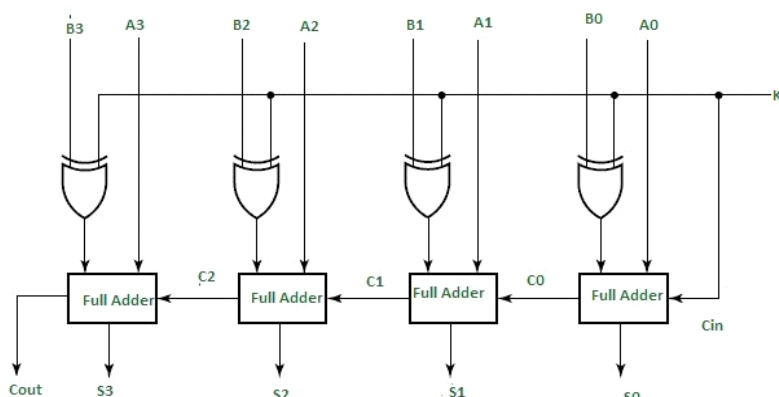


Figure 1.8: Illustrates the Configuration of a 4-bit Adder-subtractor Circuit

- The control operation, denoted as M type, dictates the behaviour of the circuit. When M equals 0, the circuit functions as an adder, and when M equals 1, it operates as a subtractor.
- Each XOR gate is influenced by the input M and one of the inputs B. In the case where M equals 0, the XOR operation with 0 results in B. Consequently, all adders take on the value of B, and the input carry bit is set to 0, leading to the circuit performing A plus B.
- Conversely, when M equals 1, the XOR operation with 1 result in the complement of B (B'). Additionally, the input carry bit is set to 1. In this scenario, the circuit executes the operation of adding A to the two's complement of B, thus performing subtraction.

Binary Incrementer

The microoperation known as auto-increment involves adding one to the value stored in the scratchpad. As an illustration, consider a 4-bit register with a binary value of 0110. Following the increment operation, it transforms to 0111.

The process of auto-incrementing can be implemented through a 4-bit integrated circuit amplifier, depicted in Figure 1.8.

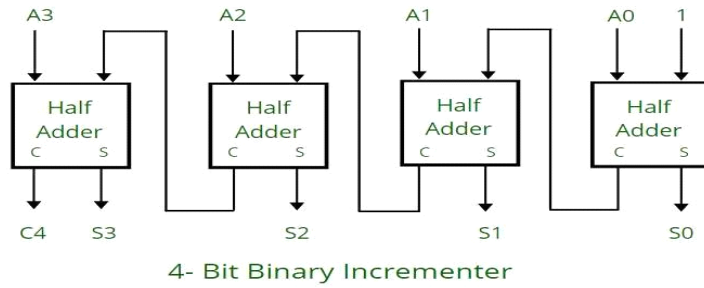


Figure 1.9: Block Diagram of 4 Bit Binary Incrementer

- One input of the half adder (HA) is linked to logic -1, while the other input is connected to the least significant bit to be incremented.
- The output generated by the half adder is then linked to one of the inputs of the higher-order half adder.
- This configuration processes four inputs from A0 to A3, increments each by one, and produces the output from S0 to S3.
- The output carry bit C4 becomes 1 only when the binary number 1111 is incremented. Consequently, the outputs from S0 to S3 become 0.
- The circuit illustrated in Figure 1.8 can be extended for n-bit binary incrementation by including additional half adders.
- An input connected to logic 1 is necessary for the least significant bit, while other inputs carry the numbers to be incremented or transferred from the preceding level.

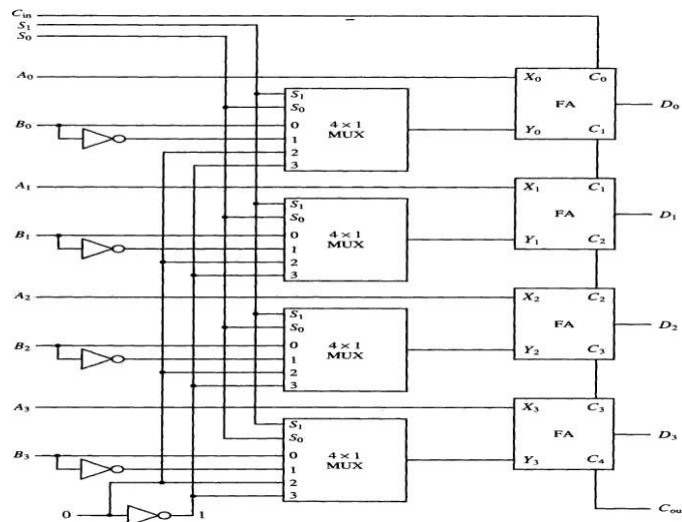


Figure 1.10: Schematic Diagram of 4 bit Arithmetic Unit

Arithmetic Circuits

- The fundamental function of the arithmetic circuit is addition.
- By manipulating the data control access connected to the adder, diverse arithmetic operations can be accomplished.
- Illustrated in Figure 1.9 is the diagram of a 4-bit arithmetic circuit, incorporating four fully integrated 4-bit adders and four multiplexers to facilitate the selection of various functions.
- The configuration involves two 4-bit inputs, A and B, with a single 4-bit output, D.
- The four inputs from A are directly routed to the X input of the binary adder.
- Each of the four inputs to B is integrated into the data input of the multiplexer and is further subjected to addition with B.
- The data input of the multiplexer is also supplied with the addition of B, and the remaining two data inputs are linked to logic 0 and logic 1.
- The operation of the four multiplexers is governed by the control of two selectable inputs, S1 and S0. These connections play a vital role in the Binary Adder. There are additional interconnections spanning across different levels.
- By manipulating the values of Y through the options S1 and S0 and setting Cin to 0 or 1, a total of 8 arithmetic microoperations can be generated.

Table 1.4: Bit Arithmetic Circuit Functional Table

<u>Select</u>		<u>In</u>	<u>Output</u>		<u>Microoperation</u>
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add w/carry
0	1	0	B	$D = A + \overline{B}$	Subtract w/borrow
0	1	1	B	$D = A + \overline{B} + 1$	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

Summation

- When S1S0 equals 00, the value of B is directed to the Y input of the adder.
- If Cin equals 0, the output D corresponds to the sum of A and B.
- In the case of Cin being 1, the output D results from the addition of A, B, and 1.
- This micro-system functions effectively, whether with or without an additional input.

Subtraction

- When S1S0 is set to 01, the addition of B is directed to the Y input of the adder.
- If Cin is equal to 1, the output D represents $A + B + 1$. This operation effectively accomplishes A plus B added twice, equivalent to $A - B$ subtracted.
- In the scenario where Cin is 0, D equals $A + B$, comparable to borrowing and subtracting, such as $A - B - 1$.

Increment

- If S1S0 is set to 10, the input from B is disregarded, and a sequence of all 0's is fed into the Y input. The resulting output is $D = A + 0 + \text{Cin}$. When Cin is 0, the outcome is $D = A$; when Cin is 1, the result is $D = A + 1$.
- In the former situation, the transition is direct from input A to output D.
- In the latter case, the value of A is incremented by 1.

Reduction

- When S1S0 = In Figure 11, all 1's are placed at the Y input of the adder to form the reduction $D = A - 1$ when Cin = 0.
- This is because the number of each is equal to 2's 1's complement (the complement of the 2's 0001 pair is 1111).
- Adding the number, A to 2's complement gives us $F = A + 1$'s 2's complement
- When Cin = 1 $D = A - 1 + 1 = A$, which leads to entry from A to D.

Logical Microoperations

- Logical microoperations are manifestations of binary operations applied to bit strings held in a temporary register.
- These operations consider each bit of the scratchpad individually, treating them as binary variables.
- For instance, the script for the XOR microoperation involving the contents of two temporary registers, R1 and R2, can be articulated as follows:
- Specifies the micro-operation to be executed on each bit of the scratchpad when the control variable P is set to 1.

List of Logical Micro-operations

- The capability to execute 16 distinct operations using two binary variables.
- These operations can be derived by considering all possible combinations of data from the two binary variables, as illustrated in the table.

Table 1.5: Truth Table for 16 Functions of Two Variables

TABLE 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 algebraic representation of the 16 Boolean functions for two variables x and y is provided in the initial row of Table 4-6.
- These 16 logical microoperations are generated by substituting the variable x with the binary contents of register A.
- The logic functions listed in the second column denote the relationships between the binary elements of B.

Table 1.6: Sixteen Logic Micro-Operations

TABLE Sixteen Logic Microoperations		
Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

Hardware Implementation

Shift Microoperation

- Shift microoperations facilitate the serial transfer of data within a register.
- The register's contents can undergo either left or right shifts.
- In a shift-left operation, a bit is serially transferred into the rightmost position.
- In a shift-right operation, a bit is serially transferred into the leftmost position.
- Three types of shifts exist: logical, circular, and arithmetic.
- The symbolic representation of shift microoperations is detailed in the provided table.

Table 1.7: Shift Micro Operations

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 involves transferring 0 through the serial input.
- The symbols "shl" and "shr" represent logical shift-left and shift-right microoperations.
- The microoperations 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 , as shown in the provided table.
- During a logical shift, the bit transferred to the end position through the serial input is assumed to be 0.

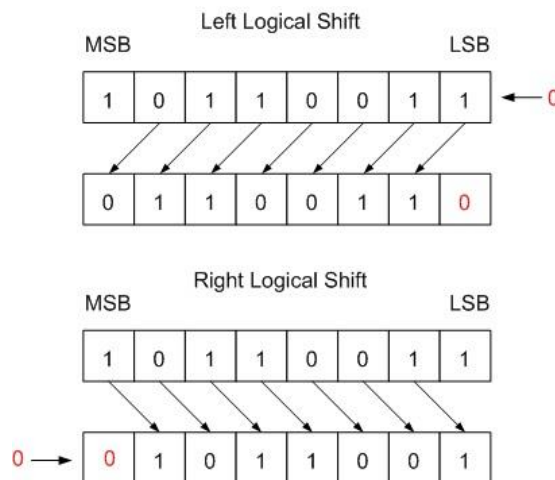


Figure 1.11: Logical Shift by One Bit

Circular Shift

- A circular shift, also known as a rotate operation, circulates the bits of the register around both ends without losing information.
- The circular shift is implemented by connecting the serial output of the shift register to its serial input.
- Symbols "cil" and "cir" represent the circular shift left and right, respectively.

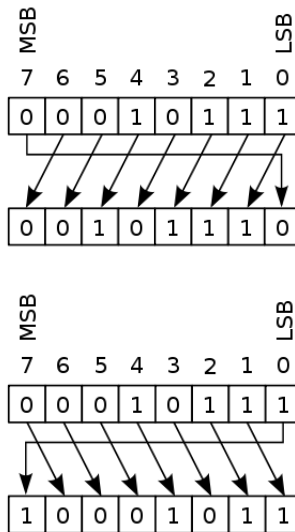


Figure 1.12: Left Circular Shift and Right Circular Shift

Arithmetic Shift

- An arithmetic shift is a microoperation designed to shift a signed binary number to the left or right.
- When performing an arithmetic shift-left, the result is equivalent to multiplying a signed binary number by 2.
- Conversely, an arithmetic shift-right mimics the effect of dividing the number by 2.
- Crucially, arithmetic shifts must preserve the sign bit, as it determines whether the number is positive or negative. This preservation ensures that the sign remains consistent, whether the operation involves multiplication or division by 2.

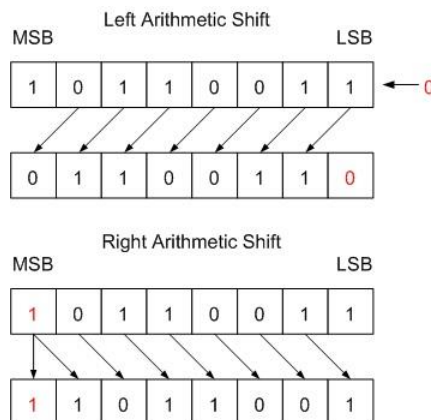


Figure 1.13: Left and Right Arithmetic Shift by One Bit

Hardware Implementation

- A combinational circuit shifter can be effectively constructed utilizing multiplexers, as illustrated in Fig.
- In the case of a 4-bit shifter, it comprises four data inputs, namely A0 through A3, and four corresponding data outputs labelled H0 through H3.
- Incorporating two serial inputs, one designated for shift left (IL) and the other for shift right (IR), this shifter allows flexible data manipulation.
- When the selection input S=0, the input data undergo a right shift (downward in the diagram).
- Conversely, when S=1, the input data undergo a left shift (upward in the diagram).
- The function table in Figure delineates which input corresponds to each output following the shift operation.
- For a shifter with n data inputs and outputs, the implementation necessitates n multiplexers.
- Additionally, the two serial inputs can be subjected to control by another multiplexer, facilitating the provision of the three possible types of shifts.

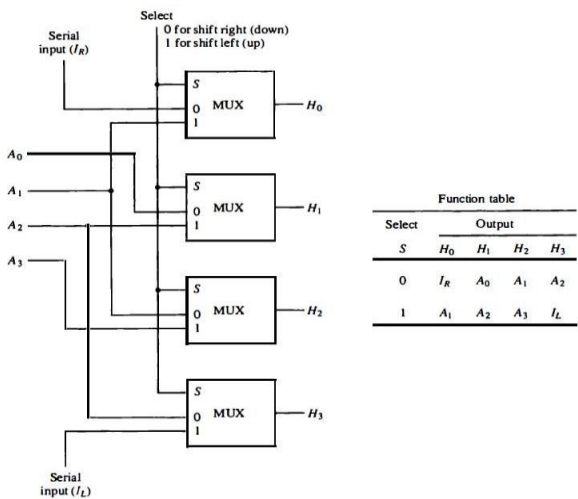


Figure 1.14: Bit Combinational Circuit Shifter

Arithmetic Logic Shift Unit

Instead of individual registers directly executing microoperations, computer systems typically integrate multiple storage registers with a common operational unit known as an Arithmetic Logic Unit (ALU).

The ALU, being a combinational circuit, enables the entire register transfer operation—starting from the source registers, passing through the ALU, and concluding with the destination register—to be executed within a single clock pulse period.

While shift microoperations are frequently handled by a distinct unit, there are instances where the shift unit is incorporated into the overarching ALU.

The amalgamation of arithmetic, logic, and shift circuits into a unified ALU with shared selection variables is conceivable. Figure illustrates a single stage of an Arithmetic Logic Shift Unit.

ARITHMETIC LOGIC SHIFT UNIT

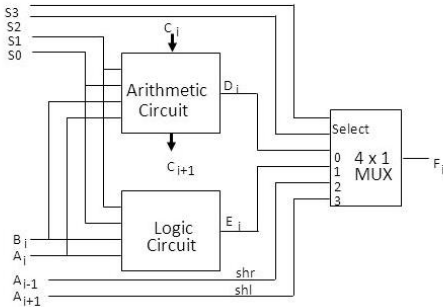


Figure 1.15: One Stage of Arithmetic Logic Shift Unit

Table 1.8: Functional Table for Arithmetic Logic Shift Unit

S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + B'$	Subtract with borrow
0	0	1	0	1	$F = A + B' + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	X	$F = A \wedge B$	AND
0	1	0	1	X	$F = A \vee B$	OR
0	1	1	0	X	$F = A \oplus B$	XOR
0	1	1	1	X	$F = A'$	Complement A
1	0	X	X	X	$F = \text{shr } A$	Shift right A into F
1	1	X	X	X	$F = \text{shl } A$	Shift left A into F

Specific microoperations are chosen using inputs S1 and S0. A 4 x 1 multiplexer at the output facilitates the selection between an arithmetic output in Di and a logic output in Ei. Inputs S3 and S2 dictate the data chosen within the multiplexer. The two additional data inputs to the multiplexer—receiving inputs Ai-1 for the shift-right operation and Ai+1 for the shift-left operation—further enhance versatility. The circuit, as delineated in Figure, encompasses eight

arithmetic operations, four logic operations, and two shift operations within a single stage. Each operation can be selected using the five variables: S3, S2, S1, S0, and Cin. It's noteworthy that the input carry Cin is exclusively employed for selecting arithmetic operations.

Table 4-8 lists the 14 operations of the ALU.

The first eight are arithmetic operations and are selected with S3S2 = 00.

- The subsequent four operations pertain to logic and are activated by. S3S2 = 01.
- It's important to note that the input carry holds no significance in logic operations, and its value is denoted by don't-care x's.
- The ultimate two operations are shift operations, designated by S3S2 = 10 and 11.
- The remaining three selection inputs exert no influence on the shift operations.

1.8. Basic Computer Organization and Design

Instruction Code

An instruction code is a group of bits that instruct the computer to perform a operation.



Figure 1.16: Basic Computer Instruction Formats

Instruction code is usually divided into two parts: Opcode and address (operand)

Operation Code (opcode)

A set of bits that specifies an operation.

For instance, operations like addition, subtraction, multiplication, shifting, or complementing are represented by opcodes.

The number of bits needed for the opcode is determined by the quantity of operands within the computer.

If there are n operands, then the n-bit opcode should accommodate at least 2^n (or fewer) operations.

Address (operand)

Designates the location of the operator, whether it's a register or a word in memory. In the case of a memory word, its address is used for specification.

For a scratchpad, a k-bit binary code is employed for identification.

To ensure coverage for scratchpad locations, the k-bit address should satisfy the condition $k\text{-bit address} \geq 2^k$ for the scratchpad.

Accumulator

An accumulator within a computer's processor functions as a form of temporary storage in the system. Specifically, it acts as a provisional dwelling, storing interim outcomes during arithmetic and logical operations.

Service

A set of instructions that specifies the task, its details, and the sequence in which the task is executed.

Instruction

A binary sequence encoding that represents the series of micro-operations performed by the computer.

Organization of Storage Services

The pivotal capability of retaining and executing instructions defines a fundamental attribute of a versatile computer. This storage arrangement, designed for comprehensive functionality, is commonly referred to as an enterprise storage program.

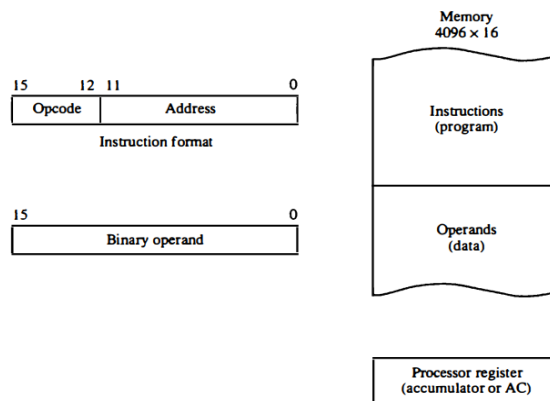


Figure 1.17: Stored Program Organization

- A straightforward method to structure a computer involves a single processor register and an instruction code format comprising two components.

- The initial segment designates the intended operation, while the subsequent part designates an address.
- Instructions are segregated, with one section for instruction storage and another for data storage.
- To address a memory unit with 4096 words, a 12-bit address is necessary, as 2^{12} equals 4096.
- If each instruction code is accommodated within a 16-bit memory word, four bits are available for the operation code (opcode), allowing specification of one of 16 potential operations. Simultaneously, the remaining 12 bits are utilized for specifying the operand's address.

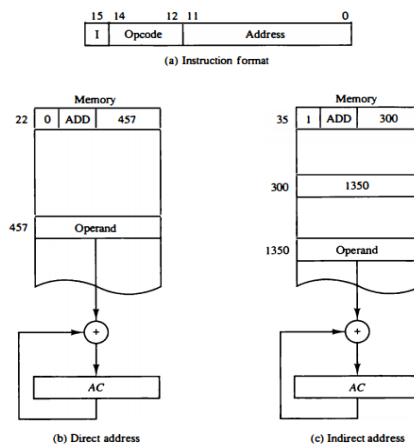


Figure 1.18: Demonstration of Direct and Indirect Address

1.9. Addressing of Operand

- Direct Address.
- Indirect Address.
- Immediate Operand.

Direct Address

When the instruction's second part designates the address of an operand, it is described as having a direct address.

Indirect Address

When the second part of the instruction designates the address of a memory word containing the operand's address, such instructions are said to have an indirect address.

Immediate Operand

When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand.

It comprises a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I. The mode bit is 0 for a direct address and 1 for an indirect address. In Figure 2.2, a direct address instruction is illustrated, residing in memory address 22.

With the I bit set to 0, the instruction is identified as a direct address instruction. The opcode indicates an ADD instruction, and the address part represents the binary equivalent of 457. The control locates the operand in memory at address 457 and performs the addition to the content of the accumulator (AC).

In Figure 2.3, the instruction located at address 35 features a mode bit I = 1, signifying it as an indirect address instruction. The address part is the binary equivalent of 300. The control navigates to address 300 to retrieve the address of the operand, which, in this case, is 1350. The operand found at address 1350 is then added to the content of the accumulator (AC).

1.10. Computer Registers

It is essential to incorporate a register within the control unit to store the instruction code once it is retrieved from memory. In a computer system, there is a requirement for processor registers dedicated to data manipulation, along with a register designated for holding memory addresses.

Table 1.9: List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
<i>DR</i>	16	Data register	Holds memory operand
<i>AR</i>	12	Address register	Holds address for memory
<i>AC</i>	16	Accumulator	Processor register
<i>IR</i>	16	Instruction register	Holds instruction code
<i>PC</i>	12	Program counter	Holds address of instruction
<i>TR</i>	16	Temporary register	Holds temporary data
<i>INPR</i>	8	Input register	Holds input character
<i>OUTR</i>	8	Output register	Holds output character

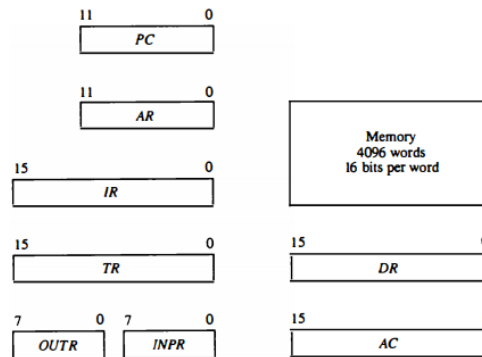


Figure Basic computer registers and memory.

Figure 1.19: Basic Computer Registers and Memory

- The operand fetched from memory is stored in the data register (DR).
- The accumulator (AC) serves as a versatile processing register.
- The instruction register (IR) holds the instruction retrieved from memory.
- The temporary register (TR) is utilized for temporary data storage during processing.
- The memory address register (AR) consists of 12 bits.
- The program counter (PC) also has 12 bits and retains the address of the subsequent instruction to be fetched from memory once the current instruction is executed.
- Instructions are sequentially read and executed, unless a branch instruction is encountered, prompting a transfer to a non-consecutive instruction in the program.
- For input and output operations, two registers are employed. The input register (INPR) receives an 8-bit character from an input device, while the output register (OUTR) stores the 8-bit character for output.

Common Bus System for Basic Computer Register

A pair of communication lines designed to efficiently transmit substantial volumes of data between two locations.

In a basic computer system, consisting of eight temporary registers, a memory unit, and a control unit, efficient data transfer mechanisms are crucial. To address this, a bus architecture is employed, offering a streamlined approach to interconnecting registers and facilitating communication between memory and registers.

The schematic above illustrates the connection between the memory, registers, and the common bus. Seven registers, denoted by names and outputs, are linked to the bus. The specific output selected for the bus is determined by the binary values of selector switches S2, S1, and

S0. The LD (load) input allows a designated register to receive data from the bus during the next transfer. When memory write input is enabled, the memory retrieves data from the bus. For reading, when S2S1S0 = 111, the memory places its 16-bit output on the data bus.

The registers, such as DR, AC, IR, and TR, are 16 bits each, while AR and PC, holding memory addresses, are 12 bits each. When data from AR or PC is applied to the 16-bit data bus, the four most significant bits are set to 0. INPR and OUTR registers are 8 bits each, communicating with the 8 significant bits on the data bus. Notably, only the INPR register provides data to the bus, while OUTR can only receive data from it.

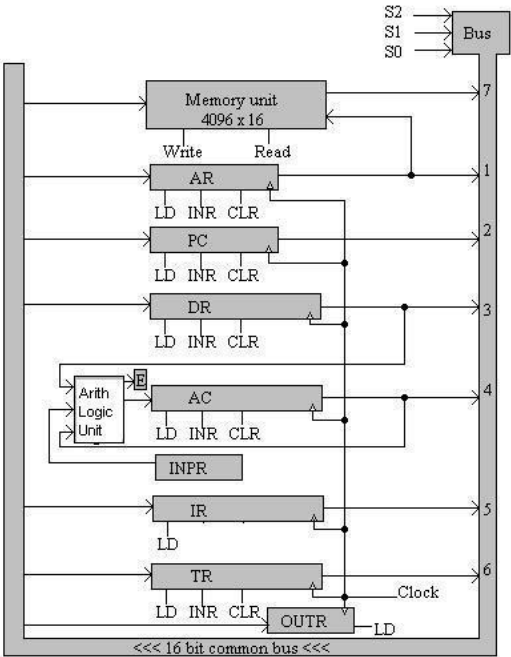


Figure 1.20: Basic Computer Registers Connected to a Common Bus

In a basic computer system, consisting of eight temporary registers, a memory unit, and a control unit, efficient data transfer mechanisms are crucial. To address this, a bus architecture is employed, offering a streamlined approach to interconnecting registers and facilitating communication between memory and registers.

The schematic above illustrates the connection between the memory, registers, and the common bus. Seven registers, denoted by names and outputs, are linked to the bus. The specific output selected for the bus is determined by the binary values of selector switches S2, S1, and S0. The LD (load) input allows a designated register to receive data from the bus during the next

transfer. When memory write input is enabled, the memory retrieves data from the bus. For reading, when $S2S1S0 = 111$, the memory places its 16-bit output on the data bus.

The registers, such as DR, AC, IR, and TR, are 16 bits each, while AR and PC, holding memory addresses, are 12 bits each. When data from AR or PC is applied to the 16-bit data bus, the four most significant bits are set to 0. INPR and OUTR registers are 8 bits each, communicating with the 8 significant bits on the data bus. Notably, only the INPR register provides data to the bus, while OUTR can only receive data from it.

Five registers exhibit three control operations: LD (load), INR (increment), and CLR (clear). AR is consistently used to represent memory addresses, tightly associated with memory operations. The AC register, with multiple sets of 16-bit inputs, receives data from diverse sources, including the DR file record, 8-bit inputs from the INPR input register, and the result of addition from adders and logic circuits. The clock switch orchestrates the transfer of bus contents to the designated register, concluding the processing cycle.

1.11. Instruction Format with its Types

- Basic computers have three types of instruction codes as shown in the figure.
- 16-bit format.
- The instruction format in this system comprises a 16-bit structure, with the opcode segment occupying 3 bits. The interpretation of the remaining 13 bits is contingent upon the specific opcode encountered during program execution.
- Instructions related to memory operations utilize 12 bits to delineate a memory address, accompanied by 1 bit designating the address type (Type I). In this context, I equal 0 for direct addressing and 1 for indirect addressing.

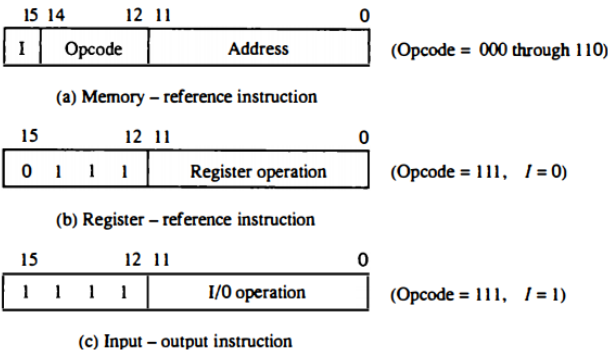


Figure 1.21: Basic Computer Instruction Formats

Scratchpad instructions, denoted by opcode 111, utilize the leftmost bit (bit 15) to signify instruction type 0. These instructions are tailored for operations or tests involving the AC register. As operands are not necessarily stored in memory, the remaining 12 bits are leveraged to convey operational details, serving as a versatile platform for experimentation.

Input and output instructions, exempt from requiring data memory access, are identified by opcode 111. The leftmost bit is set to indicate instruction type 1. Within this category, the remaining 12 components are dedicated to specifying the nature of input/output operations or the execution of tests, Time control with diagram.

The block diagram of the control room is shown in figure 2.7

The Control Unit consists of several essential components, including:

- Two Decoders
- Row Counter
- Control Logic Gate

In the control unit, instructions fetched from memory are stored in the Instruction Register (IR). The IR is then divided into three parts within the control unit: the I bit, the opcode bits (12-14), and bits 0 to 11. The 3 X 8 decoder is employed to determine the opcode from bits 12 to 14. The 15th bit of the instruction is directed to the flip-flop labeled I.

Table 1.10: Basic Computer Instructions

Symbol	Hexadecimal Code	Description
AND	0xxx 8xxx	And memory word to AC
ADD	1xxx 9xxx	Add memory word to AC
LDA	2xxx Axxx	Load memory word to AC
STA	3xxx Bxxx	Store AC content in memory
BUN	4xxx Cxxx	Branch Unconditionally
BSA	5xxx Dxxx	Branch and Save Return Address
ISZ	6xxx Exxx	Increment and skip if 0
CLA	7800	Clear AC
CLE	7400	Clear E(overflow bit)
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right AC and E
CIL	7040	Circulate left AC and E
INC	7020	Increment AC
SPA	7010	Skip next instruction if AC > 0
SNA	7008	Skip next instruction if AC < 0
SZA	7004	Skip next instruction if AC = 0
SZE	7002	Skip next instruction if E = 0
HLT	7001	Halt computer
INP	F800	Input character to AC
OUT	F400	Output character from AC
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag

The outputs of the decoder, denoted as D0 through D7, are utilized to control logic gates associated with bits 0 to 11. A 4-bit counter array facilitates counting in binary from 0 to 15. The counter's output is effectively set 16 times, progressing from T0 to T15.

The SC (Set Counter) meter can be concurrently loaded or unloaded. Typically, the counter undergoes incrementation, providing a predefined time sequence for the 4 X 16 decoder. Once the counter reaches a specific point, it is reset, and the continuous time resumes from T0.

For instance, if SC is incremented to establish set times T0, T1, T2, T3, and T4, at time T4, SC is cleared to 0 if the decoder output D3 is valid. This operation can be expressed as follows:

$$D3T4: SC \leftarrow 0$$

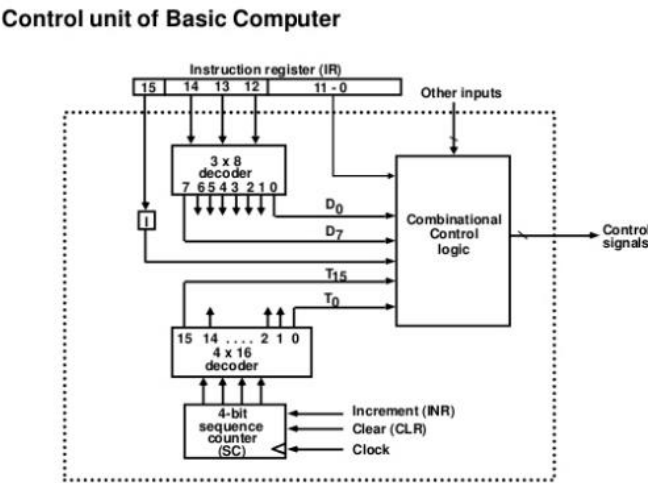


Figure 1.22: Control Unit of Basic Computer

Timing Diagram

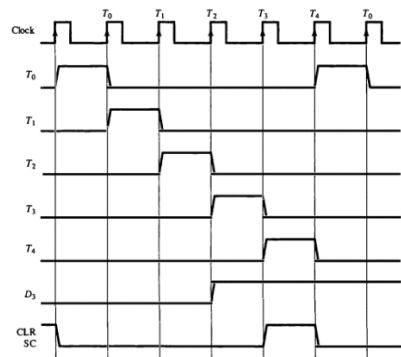


Figure 1.23: Example of Control Timing Signals

The timing diagram illustrates the temporal relationships among various control signals within the system.

Initially, the SC counter responds to positive clock transitions, with the CLR input being active. The first positive clock transition resets SC to 0, initiating the T0 time sequence. T0 remains valid for a single clock cycle, and the positive time switch labelled T0 enables the corresponding control input during this period.

Subsequently, SC increments on all positive clock transitions, excluding the CLR input function. The timing signals T0, T1, T2, T3, and T4 follow a sequential order during this process. If SC is not reset, the timing sequence continues until T5, T6, up to T15, and then loops back to T0.

The final waveforms depict how SC resets when $D3T4 = 1$. The output of the decoding circuit D3 becomes active towards the end of the T2 signal period. When the T4 signal becomes valid, the output of the AND gate, utilizing the D3T4 control function, is activated. This signal is employed as the CLR input for SC, causing the counter to reset during the subsequent clock cycle. As a result, the T0 time sequence resumes instead of progressing to T5. If SC were incremented instead of being reset, T5 would be initiated.

Instruction Cycle

- Execution of programs on a computer follows an instruction cycle.
- The instruction cycle is further subdivided into distinct phases.
- In a basic computer, the instruction cycle comprises the following stages:
 1. Fetching the instruction from memory.
 2. Decoding the instruction.
 3. If the instruction involves an indirect address, retrieving the correct address from memory.
 4. Executing the instruction.

Following step 4, the control loops back to step 1 to fetch, decode, and execute the next instruction.

This iterative process continues unless a HALT instruction is encountered.

- The flowchart illustrates the initial configuration of the instruction loop and outlines how the controller subsequently determines the type of instruction.

- If $D7 = 1$, the instruction is intended for use in a scratchpad or input/output mode. If $D7 = 0$, the opcode must fall into one of the seven other values (110), indicating memory usage. The control then examines the value of the first bit of the instruction, now stored in flip-flop I.
- If $D7 = 0$ and $I = 1$, the instruction involves memory usage with an indirect address. In this case, the correct address needs to be read from memory.
- The four types of instructions are categorized into four actions, initiated by the timing signal with T3 timing. This can be represented as:

D'7 I T3: $AR \leftarrow M[AR]$

D'7 I' T3: Nothing

D7 I' T3: Execute instruction using scratchpad.

D7 I T3: Execute instruction input and output

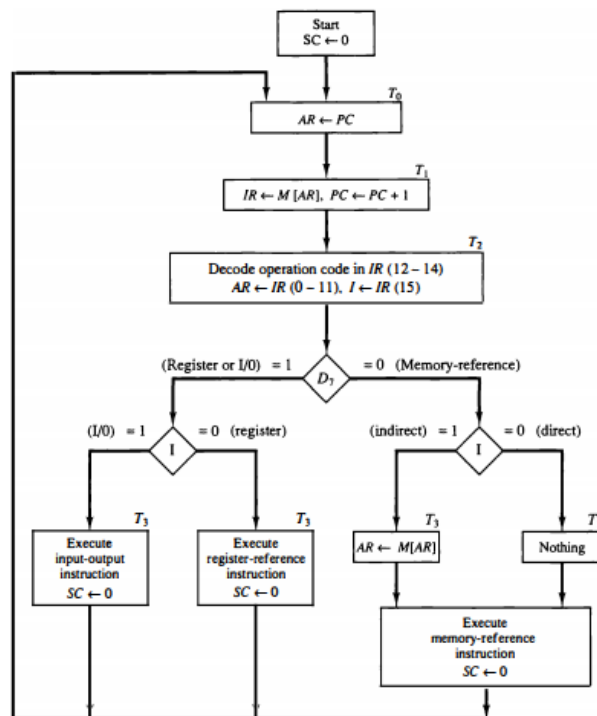


Figure 1.24: Flow Chart for Instruction Cycle

- Encountering a memory instruction with $I = 0$ requires no further action, as the address is already present in AR. However, when $D'7 I T3 = 1$, the SC counter's value must be incremented to allow the memory instruction to proceed using the T4 timing signal.

- Register or entry/exit instructions can be executed by triggering the corresponding timing signal T3. Once the instruction is completed, SC is cleared, and control returns to the instruction cycle with T0 = 1. SC is incremented or cleared at each positive clock cycle.

Register Reference Instruction

- When the register reference instruction is decoded, bit D7 is set to 1.
- All control functions must have a Boolean relationship D7 I T3



Figure 1.25: Register Reference Instructions

The 12 scratchpad instructions are: listed below:

Table 1.11: Register Reference Instructions Table

	r:	SC ← 0	Clear SC
CLA	rB ₁₁ :	AC ← 0	Clear AC
CLE	rB ₁₀ :	E ← 0	Clear E
CMA	rB ₉ :	AC ← AC'	Complement AC
CME	rB ₈ :	E ← E'	Complement E
CIR	rB ₇ :	AC ← shr AC, AC(15) ← E, E ← AC(0)	Circular Right
CIL	rB ₆ :	AC ← shl AC, AC(0) ← E, E ← AC(15)	Circular Left
INC	rB ₅ :	AC ← AC + 1	Increment AC
SPA	rB ₄ :	if (AC(15) = 0) then (PC ← PC+1)	Skip if positive
SNA	rB ₃ :	if (AC(15) = 1) then (PC ← PC+1)	Skip if negative
SZA	rB ₂ :	if (AC = 0) then (PC ← PC+1)	Skip if AC is zero
SZE	rB ₁ :	if (E = 0) then (PC ← PC+1)	Skip if E is zero
HLT	rB ₀ :	S ← 0 (S is a start-stop flip-flop)	Halt computer

- These 12 bits are housed in the Instruction Register (IR) from positions 0 to 11. They are also referred to as Address Register (AR) during T2.
- Instruction execution is completed at time T3.
- The first 7 register instructions operate on either the AC or the E register.
- Under specific conditions, the following four instructions sequentially lead to the skipping of the subsequent instruction. This skipping is accomplished by incrementing the Program Counter (PC).
- Conditional statements should be viewed as part of the control flow. The AC is considered positive when the sign bit in AC (15) = 0 and negative when AC (15) = 1. If all flip-flops in the scratchpad are set to zero, the content of AC is zero (AC = 0).

- The HLT instruction triggers the clearing of the start-stop flip-flop S, halting parallel computation. To resume operation, the start-stop trigger must be manually set.

Memory Reference Instructions

- When the memory reference instruction is specified, bit D7 is set to 0.

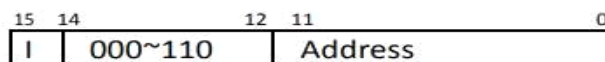


Figure 1.26: Memory Reference Instructions

The following table lists 7 memory reference instructions.

Table 1.12: Memory Reference Instructions

Symbol	Operation Decoder	Symbolic Description
AND	D ₀	$AC \leftarrow AC \wedge M[AR]$
ADD	D ₁	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D ₂	$AC \leftarrow M[AR]$
STA	D ₃	$M[AR] \leftarrow AC$
BUN	D ₄	$PC \leftarrow AR$
BSA	D ₅	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D ₆	$M[AR] \leftarrow M[AR] + 1, \text{ if } M[AR] + 1 = 0 \text{ then } PC \leftarrow PC + 1$

The accurate address of the instruction resides in the Address Register (AR) and is loaded during the T2 signal (when I = 0) or the T3 signal (when I = 1).

Execution of the memory-related command commences with the T4 time signal.

AND to AC

This instruction executes the AND logic operation between the Accumulator (AC) and the bit pair in AC. The memory word specified by the address is accessed, and the resultant outcomes of the operation are then stored back into the Accumulator (AC).

D₀T₄: DR ← M[AR]
D₀T₅: AC ← AC ∧ DR, SC ← 0

Add to AC

This operation performs a logical AND between the contents of the Accumulator (AC) and the bit pair within the AC. The instruction involves accessing the memory word indicated by the specified address. Subsequently, the results of this logical AND operation are stored back into the Accumulator (AC).

D₁T₄: DR ← M[AR]
D₁T₅: AC ← AC + DR, E ← C_{out}, SC ← 0

LDA: Load to AC

This instruction involves the transfer of the memory word indicated by the effective address to the Accumulator (AC).

D₂T₄: DR ← M[AR]
D₂T₅: AC ← DR, SC ← 0

STA: Store AC

This instruction saves the content of the Accumulator (AC) into the memory word identified by the effective address.

D₃T₄: M[AR] ← AC, SC ← 0

BUN: Branch Unconditionally

This instruction directs the program to the instruction indicated by the effective address. The BUN instruction enables the programmer to specify an instruction out of sequence, resulting in an unconditional branching or jump in the program flow.

D₄T₄: PC ← AR, SC ← 0

BSA: Branch and Save Return Address

This instruction facilitates branching to a specific section of the program referred to as a subroutine or procedure. Upon execution, the BSA instruction stores the address of the next sequential instruction (present in the Program Counter, PC) into a designated memory location identified by the effective address.

M[AR] ← PC, PC ← AR + 1
M[135] ← 21, PC ← 135 + 1 = 136

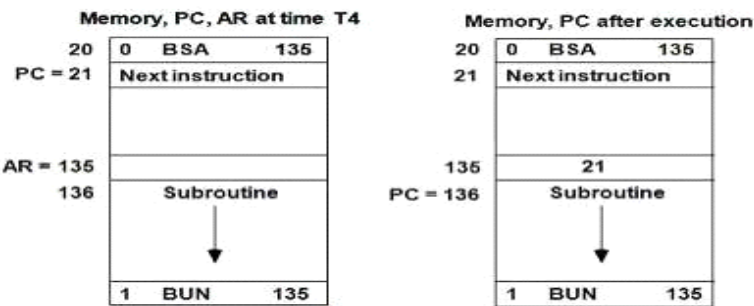


Figure 1.27: Diagram of Branch and Save Return Address

Executing the BSA instruction in a single clock cycle is impractical when employing the bus system of the basic computer. To ensure efficient utilization of memory and the bus, the BSA instruction necessitates a sequential execution involving two distinct micro-operations.

$$\begin{aligned} D_5T_4: & \quad M[AR] \leftarrow PC, AR \leftarrow AR + 1 \\ D_5T_5: & \quad PC \leftarrow AR, SC \leftarrow 0 \end{aligned}$$

ISZ: Increment and Skip if Zero

The execution of these instructions involves incrementing the memory word specified by the effective address. If the incremented value equals 0, the Program Counter (PC) is further incremented by 1. Due to the inability to directly increment a word within the memory, the process requires reading the word into the Data Register (DR), incrementing the DR, and subsequently storing the updated word back into memory.

$$\begin{aligned} D_6T_4: & \quad DR \leftarrow M[AR] \\ D_6T_5: & \quad DR \leftarrow DR + 1 \\ D_6T_6: & \quad M[AR] \leftarrow DR, \text{ if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), SC \leftarrow 0 \end{aligned}$$

1.12. Basic Computer Input and Output Configuration

- To facilitate the interaction between the computer and its user, instructions and data stored in memory must be sourced from input devices.
- Conversely, the computed results must be directed to the user via output devices.
- For illustration, let's consider a basic setup involving a terminal equipped with a keyboard for input and a printer for output. This example highlights the fundamental requirements for input and communication.

Input and Output Configuration

- The terminal functions as both a sender and receiver of messages, with each message comprising eight alphanumeric characters.
- Serial data received from the keyboard is transferred to the INPR register.
- On the other hand, serial data destined for the printer is stored in the OUTR output register.

Control Flowchart

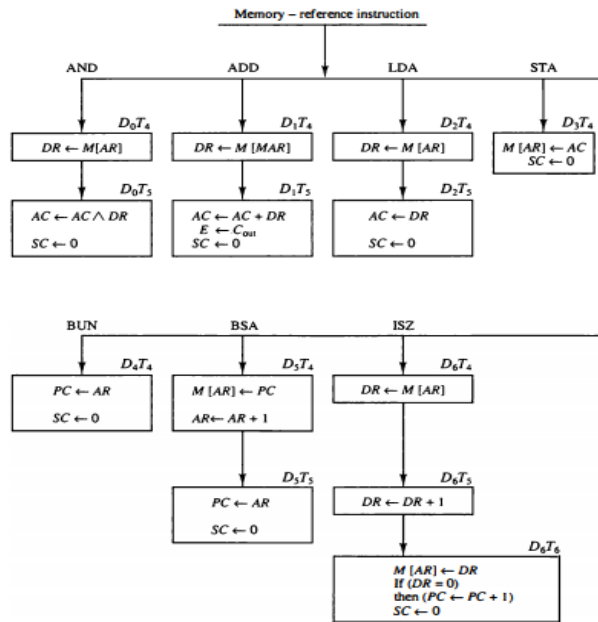


Figure 1.28: Flow Chart for Memory Reference Instructions

- Serial data received from the keyboard is transferred to the INPR register.
- On the other hand, serial data destined for the printer is stored in the OUTER output register.
- Communication between these two registers is facilitated through the intermediary use of the AC register.

Input - Output Configuration:

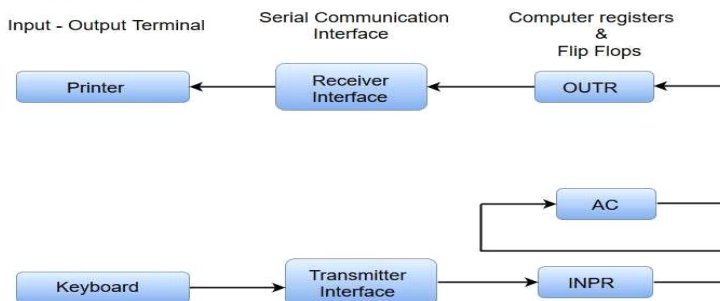


Figure 1.29: Input-Output Configuration

- The input register INPR, comprising eight bits, is responsible for holding alphanumeric input data.

- Controlled by the 1-bit input flag FGI, this flag is toggled to 1 when new information becomes available from the input device. Subsequently, it resets to 0 upon the acceptance of the information by the computer.
- In a mirrored fashion, the output register OUTF operates. Initially, the output flag FGO is set to 1. The computer checks this flag; if it is 1, the information present in AC is transferred in parallel to OUTF, simultaneously resetting FGO to 0.
- The output device processes the received coded information, prints the corresponding character, and upon completion of the operation, sets FGO back to 1.

Input-Output Instructions

To facilitate the transfer of information to and from the AC register, and to manage flag bits and control the interrupt facility, input and output instructions become essential.

- These instructions bear an operation code of 1111 and are distinguished by the control unit when $D_7 = 1$ and $I = 1$.
- The specific operation is dictated by the remaining bits of the instruction.
- Outlined below are the control functions and micro-operations associated with input-output instructions.

Table 1.13: Input- Output Instructions

List of input-output instructions

$D_7I_7 = p$ (common to all input-output instructions)		
$IR(i) = B_i$ [bit in $IR(6-11)$ that specifies the instruction]		
	$p:$	$SC \leftarrow 0$
INP	$pB_{11}:$	$AC(0-7) \leftarrow INPR, FGI \leftarrow 0$
OUT	$pB_{10}:$	$OUTR \leftarrow AC(0-7), FGO \leftarrow 0$
SKI	$pB_9:$	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	$pB_8:$	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	$pB_7:$	$IEN \leftarrow 1$
IOF	$pB_6:$	$IEN \leftarrow 0$
		Clear SC
		Input character
		Output character
		Skip on input flag
		Skip on output flag
		Interrupt enable on
		Interrupt enable off

- The INP instruction facilitates the transfer of input data from INPR to the eight low-order bits of AC while simultaneously resetting the input flag to 0.
- Conversely, the OUT instruction transfers the eight least significant bits of AC into the output register OUTF, resetting the output flag to 0.
- The subsequent two instructions in Table 2.2 assess the flag statuses, leading to the skipping of the next instruction if the flag is 1. Normally, this skipped instruction would be a branch instruction, looping back to check the flag once again.

- The branch instruction remains unskipped if the flag is 0. In the case of a flag being 1, the branch instruction is bypassed, and either an input or output instruction is executed.
- The final two instructions involve the setting and clearing of an interrupt enable flip-flop, denoted as IEN. The significance of IEN is elucidated in the context of interrupt operations.

Program Interrupt

- The computer continuously monitors the flag bit, and upon detecting it set, it initiates an information transfer.
- However, the disparity in information flow rates between the computer and the input-output device renders this type of transfer less efficient.
- An alternative approach, known as interrupt-driven transfer, involves letting the external device notify the computer when it is ready for the transfer.
- During this time, the computer can engage in other tasks. This approach employs the interrupt facility. While the computer is executing a program, it refrains from checking the flags.
- Upon the flag being set, the computer experiences a momentary interruption from its ongoing program.
- The computer briefly diverts from its current task to execute the input or output transfer prompted by the interrupt.
- It subsequently returns to the original program to resume its previous operations prior to the interrupt.
- The interrupt enable flip-flop IEN can be manipulated with two instructions.
- When IEN is cleared to 0 (via the IOF instruction), the flags are incapable of interrupting the computer.
- Contrarily, when IEN is set to 1 (using the ION instruction), the computer becomes susceptible to interruptions.
- The handling of interrupts by the computer can be elucidated through the flowchart depicted in Figure 5-13.
- An interrupt flip-flop R is integrated into the computer. When $R = 0$, the computer operates within a single instruction cycle.

At the conclusion of the instruction cycle, the controller assesses IEN. If IEN is 0, signifying the programmer's preference to avoid interrupts, the control proceeds to the next instruction cycle.

However, if IEN is 1, the control examines the flag bit. If both flags are 0, indicating that neither the input nor the output device is ready to transmit data, the control proceeds to the next instruction cycle.

If the flag is set to 1 when IEN = 1, flip-flop R is set to 1. At the conclusion of the execution cycle, R is checked, and if it equals 1, the computer enters the interrupt loop, interrupting the ongoing instruction cycle.

Interrupt Loop

- The interrupt loop serves as the hardware realization of the branch and return address preservation function.
- The addresses residing in the Program Counter (PC) are preserved in a designated location.
- This storage location may take the form of a register, a memory segment, or a specialized memory location.

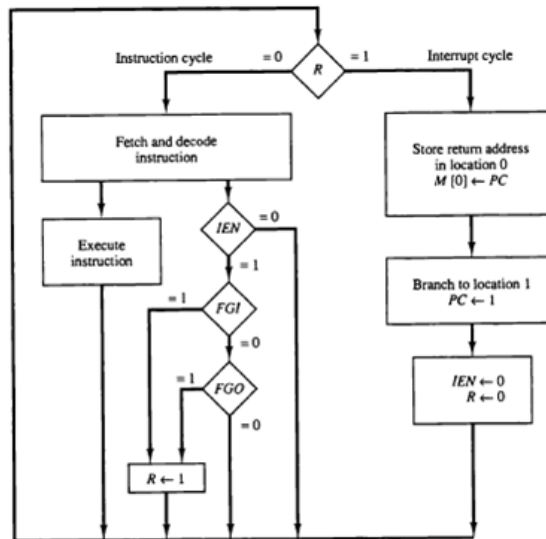


Figure 1.30: Flow Chart for Interrupt Cycle

- As the controller executes the instruction at address 255, an interrupt is triggered, setting R to 1.
- Simultaneously, the return address, which is the value in PC at 256, is stored.
- Prior to this, the programmer has established an input/output program in memory, commencing at address 1120, and the BUN 1120 instruction at address 1 is executed as illustrated in Figure 5.14(a).

- When control reaches time signal T0 with R equal to 1, the interrupted execution resumes.

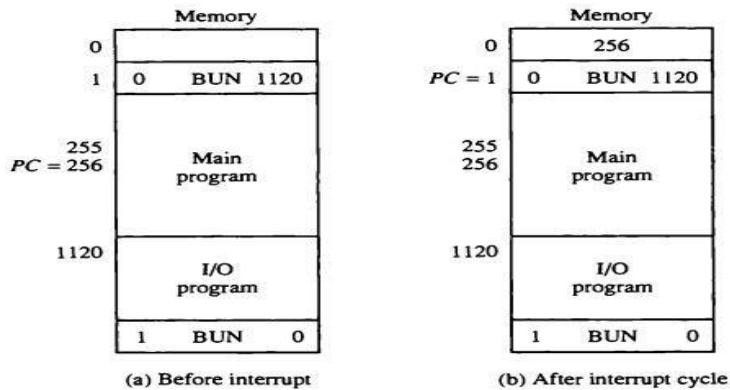


Figure 1.31: Demonstration of Interrupt Cycle

- The content of PC, which was stored in memory location 0, is restored to PC at 1, and R is reset to 0.
- The branch instruction at address 1 redirects the program flow to the input program starting at address 1120.
- The program examines the flags to determine which ones are set and proceeds to send or receive the required message.
- Upon completion, the ION command is received to set IEN to 1 (enabling further interrupts), and the program returns to its interrupted state.

Short Questions

1. What is language change?
2. What does it mean to manage memory?
3. What is the role of the address management record?
4. What is a letter?
5. What is the transaction code?
6. What is the address?
7. What is the use of incrementing and skipping if it is zero?
8. Explain non-event branch?
9. Declare the branch and save the address back?
10. Can you write about memory manipulation?
11. Explain control room and drawing time?
12. Do you write about computer classes?

13. What is computer organization?

Long Question

1. Explain word change in scrapbook?
2. List three event bus buffers?
3. Do you discuss micro mathematics studies?
4. Explain binary adder subtractor and binary incrementer?
5. What is a logical microtransaction? Explain the use of logical micro's operation?
6. Explain with diagrams the different types of computers registration and public buses?
7. Explain Shift micro-operation?
8. What is memory? Explain in order of address?
9. Clarify the activities involved in completing the instruction
10. Explain the cycle of action?

Multiple Choice Questions

1. RTL stands for
 - a) Random transmission
 - b) Recorded transmission.
 - c) Radar transmission
 - d) Radio transmission
2. The method that represents the calculation process with symbols is
 - a) Programming language
 - b) Random transfer language
 - c) Register transfer language
 - d) Arithmetic transfer language
3. How many types of micro transactions are there
 - a) 2
 - b) 4
 - c) 6
 - d) 8
4. What operations are used to add, subtract, increase, decrease, and add?
 - a) Datapath
 - b) Shift
 - c) Arithmetic
 - d) All of the above

5. Where a micro function converts a message from one register to a not
 - a) Register transfer
 - b) Arithmetic
 - c) Logic
 - d) All of the above
6. What operation does the computer perform on the data stored in the temporary register
 - a) Register variable
 - b) Arithmetic
 - c) Logic
 - d) All of the above
7. In which transfers the computer register are indicated in capital letter s for depicting its function
 - a) Memory transfer
 - b) Register transfer
 - c) Bus transfer
 - d) None of the above
8. In memory transfer location address is supplied by that puts this on address bus
 - a) ALU
 - b) CPU
 - c) MAR
 - d) MDR
9. IN memory read the operation puts memory address on to a register known as
 - a) PC
 - b) ALU
 - c) MAR
 - d) All of the above
10. Operations of memory transfers are
 - a) Read
 - b) Write
 - c) Both read and write
 - d) None
11. Which operation puts memory address in memory address register and data in DR
 - a) Memory read
 - b) Memory writes

- c) Both
 - d) None
12. The register that includes the address of memory unit is termed as the
- a) MAR
 - b) PC
 - c) IR
 - d) None of the above
13. N register transfer which system is a sequential logic system in which flip-flops and gates are constructed.
- a) Digital system
 - b) Register
 - c) Data
 - d) None
14. The memory bus is also referred as
- a) Data bus
 - b) Address bus
 - c) Memory bus
 - d) All of the above
15. High level language C support register transfer technique for application
- a) Executing
 - b) Compiling
 - c) Both
 - d) None
16. A counter is incremented by one and memory unit is considered as collection of
- a) Transfer register
 - b) storage register
 - c) RTL
 - d) All of the above
17. How many parts of the memory bus
- a) 2
 - b) 3
 - c) 5
 - d) 6

18. Which is the straightforward register transfer the data from register to another register temporarily
- a) Digital system
 - b) Register
 - c) Data
 - d) Register transfer operations
19. A three-state gate is defined as:
- a) Analog circuit
 - b) Analog fundamental wave
 - c) a & b
 - d) Digital circuit
20. In each case, the selected bus name is
- a) Control signal
 - b) No signal
 - c) All signal
 - d) Label all the above
21. In each bit of the scrapbook
- a) 2 types of lines
 - b) 3 types of lines
 - c) 1 line of lines
 - d) None of these
22. What conversion is used for signed binary numbers?
- a) Logic
 - b) Arithmetic
 - c) a and b
 - d) None of these
23. What are the operations of addition, subtraction, increase, decrease and addition?
- a) Data bus
 - b) Memory transfer
 - c) Arithmetic operations
 - d) All of these
24. A system in which symbols are used to represent the computational process.
- a) Programming language
 - b) Random transfer language

- c) Register transfer language
 - d) Arithmetic transfer language
25. Memory read function in it, the function that places the memory in the scratchpad is called
- a) PC
 - b) ALU
 - c) MR
 - d) All these
26. Which of the following names is used to indicate the next instruction to be executed in the CPU's control unit?
- a) Accumulator
 - b) Index register
 - c) Instruction decoder
 - d) Program Counter
27. Asynchronous transmission is also known as
- a) switched transmission
 - b) variable transmission
 - c) batch transmission
 - d) handshake transmission
28. The bus that allows I/O memory and process integration is
- a) Bus
 - b) Workload bus
 - c) Return bus behind the bus
 - d) None
29. The CPU can execute a set of machine instructions called
- a) Instruction set
 - b) Register
 - c) Procedure
 - d) Instructions
30. The length of the scrapbook is called
- a) word limit
 - b) word size
 - c) scrapbook limit
 - d) scrapbook size

31. Which of the below? record not found
- a) General purpose record
 - b) Address record
 - c) Status record
 - d) MAR
32. Which of the following is a data transfer instruction
- a) STA 16-bit address
 - b) Add A, B
 - c) MUL C, D
 - d) RET
33. Between CPU and Cache transfer is as follows
- a) Block transfer
 - b) Word transfer
 - c) Transfer
 - d) Shared transfer
34. CISC stands for
- a) Complex data-oriented CPU
 - b) Simple computer instructions
 - c) Intelligent-oriented CPU
 - d) Complex instructions CPU
35. MAR stands for.
- a) Memory address register
 - b) Main address register
 - c) Main accessible register
 - d) Memory accessible register
36. The circuit that responds to and processes the basic instructions needed to run a computer is.
- a) Memory
 - b) ALU
 - c) CU
 - d) Processor
37. Any electronic storage where information can be stored and can be retrieved later when needed.
- a) Memory

- b) Disk drive
- c) Magnetic disk
- d) Circuit

38. Which of the following systems has long instructions

- a) CISC
- b) RISC
- c) EPIC
- d) Multi-core

UNIT II - MICROPROGRAMMED CONTROL UNIT

Microprogram Control: Control memory, address sequencing, microprogram example, control unit design. **Central processing unit:** Organization of general registers, instruction formats, addressing modes, data transfer and manipulation, program control.

2.1. Introduction

The primary role of the controller in a digital computer is to initiate a series of micro-operations. This control unit can be realized through two distinct approaches:

- Fixed control
- Microprogrammed control

Fixed Control Unit

When the control signals are produced through traditional logic design methods within the hardware, the control unit is described as being hard-wired.

Microprogrammed Control Unit

A microprogrammed controller is characterized by storing its binary control variables in memory.

Control Memory

- Control memory is the designated storage within the microprogrammed control unit dedicated to storing the microprogram.

Control Word

- The representation of control variables at any given moment can be expressed as a binary sequence known as a control word, consisting of ones and zeros.

Micro Operations

- In the central processing units (CPUs) of computers, microoperations are intricate low-level instructions employed in certain designs to execute intricate machine instructions.

Micro Instructions

- The translation of a symbolic microprogram into its binary counterpart is facilitated through an assembler. In assembler language, every line of the microprogram delineates a symbolic microinstruction, which is further segmented into five fields: label, microoperation, CD, BR, and AD.

Microprogram

- A microprogram is composed of a series of microinstructions. Employing a microprogram entail storing all control variables within ROM words, which are accessed by the controller through sequential read operations. The microinstruction is determined by the content of the ROM word at the specified address.

Microcode

- Microinstructions can be organized and stored using subroutines, which leverage shared segments of microcode. As an illustration, the set of micro-operations necessary to generate the effective operand address for an instruction is universal across all memory instructions. This approach of utilizing subroutines enhances efficiency and facilitates the management of common micro-operations in the microcode.

Organization of the Microprogrammed Control Unit

The general configuration of the microprogrammed control unit is shown in the block diagram.

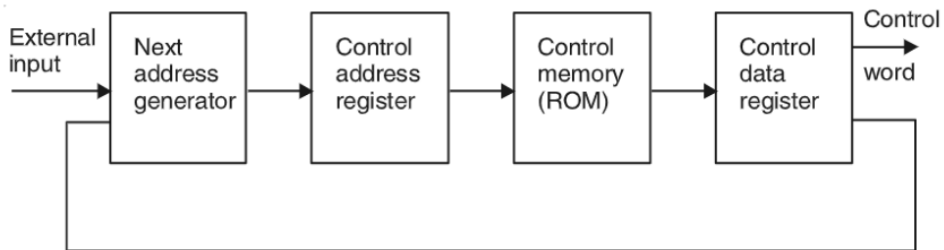


Figure 2.1: Organization of Microprogrammed Control Unit

Control memory is envisioned as a Read-Only Memory (ROM) where all essential control information is permanently stored. The control memory address register indicates the address of the microinstruction, and the control data register holds the microinstruction retrieved from the memory.

A microinstruction encompasses a control word specifying one or more microoperations for the data processor. Following the execution of these operations, the control mechanism must ascertain the subsequent address. This location could either be the next in sequence or situated elsewhere in the control memory.

Simultaneously with the performance of micro-operations, the next address is computed in the microprogram sequencer and then transferred to the control address register to access the

subsequent micro-instruction. Consequently, a microinstruction comprises components for initiating microoperations in the data processor and determining the address sequence for the control memory.

The microprogram sequencer, or another address generator, carries out functions such as incrementing the control address register by one, loading an address from control memory into the control address register, transferring an external address, or loading a start address for initiating control operations.

The control data register retains the ongoing microinstruction while the next address from memory is computed and retrieved. Sometimes referred to as a pipeline register, this data register enables the execution of microoperations specified by a control word concurrently with the generation of another microinstruction.

In this configuration, a two-phase clock system is employed, with one phase applied to the address register and the other to the data register. A key advantage of microprogrammed control lies in its adaptability—once the hardware configuration is established, introducing a different control sequence only requires entering a distinct set of microinstructions into the control memory without necessitating additional hardware or wiring modifications.

Order of Addresses

Microinstructions are organized into control memory in clusters, with each group delineating a specific routine. Each computer instruction is associated with its unique microprogram, orchestrating the generation of microoperations.

The hardware responsible for managing the sequence of control memory addresses needs the capability to navigate microinstructions within a given routine and seamlessly transition from one routine to another.

To comprehend the intricacies of address ordering in a microcontroller, let's outline the essential steps that the control unit must undertake when executing a solitary computer instruction.

Step 1

- Upon powering on the computer, the initial step involves loading the start address into the control address register.
- Typically, this start address corresponds to the location of the first microinstruction responsible for initiating the instruction fetch routine.

- The fetch routine progresses by incrementing the control address register sequentially through its subsequent microinstructions.
- As the fetch routine concludes, the computer's instruction register holds the fetched instruction, signifying the completion of this phase. register.

Step 2

- The control memory proceeds to execute a routine aimed at determining the effective address of the operand.
- Machine instructions often include bits indicating various addressing modes, such as indirect addresses and index registers.
- An optimized address calculation routine within the control memory can be implemented using a branch microinstruction, with its execution contingent on the status of the instruction mode bits.
- Following the completion of the effective address calculation routine, the memory address register contains the address of the operand.

Step 3

- The subsequent phase involves generating micro-operations to execute the instructions retrieved from memory.
- The specific micro-operation steps to be generated in the processor registers are contingent on the opcode segment of the instruction.
- Every instruction is associated with its unique microprogram routine, stored at a designated location in the control memory.
- The process of converting the instruction code bits into the address within the control memory where the routine is stored is termed the mapping process.
- A mapping procedure is essentially a set of rules defining how an instruction code is transformed into a control memory address.

Step 4

- Upon reaching the targeted routine, the microinstructions responsible for executing the instruction can be ordered by incrementing the control address register.
- Microprograms employing subroutines necessitate an external register to hold the return address.
- Storing return addresses in ROM is impractical due to the lack of write capability in read-only memory.

- Following the completion of instruction execution, control needs to revert to the fetch routine.
- This transition is facilitated by executing an unconditional branch microinstruction, directing control to the initial address of the fetch routine.

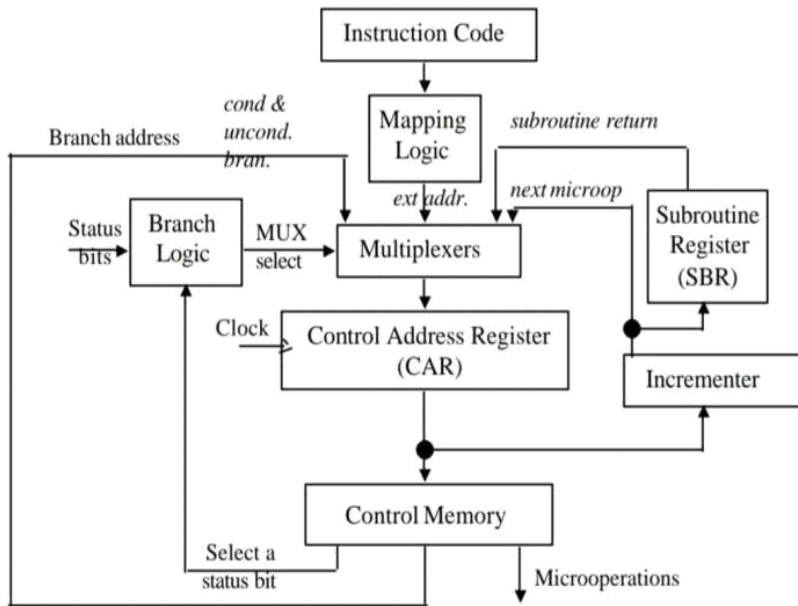


Figure 2.2: Selection of Address for Control Memory

Briefly, the address ordering options required in control memory are:

- The Control Address Register (CAR) can be incremented to select the next microinstruction in the sequence.
- Branching is achieved by specifying the branch address within one of the fields of the microinstruction.
- Conditional branching involves using a part of the microinstruction to select a specific status bit, determining its state for conditional decisions.
- The external address is conveyed to the control memory through the mapping logic circuit.
- A special register stores the return address for subroutine calls, utilized when the microprogram needs to return from the subroutine.
- The branch logic in the controller provides decision-making capabilities, relying on status conditions, which are special bits indicating parameters such as adder execution, sign bit status, instruction mode bits, and input/output conditions.

- Multiplexer output 1 generates a control signal for transferring the branch address from the microinstruction to the control address register.
- Multiplexer output 0 causes the address register to be incremented.
- Microprogrammed control involves storing control information in the control memory, programmed to execute the desired sequence of microoperations.
- Key features of microprogrammed control include slower operation compared to hardwired solutions, lower complexity, cost-effectiveness, and the flexibility to add new instructions.
- **Examples** of CPUs with microprogrammed control include the Intel 8080, Motorola 68000, and various Complex Instruction Set Computing (CISC) CPUs. Conditional branching:
 - Conditional branching is obtained by using part of a microinstruction to select a particular status bit to determine its state.
 - Status conditions are special bits in the system that provide information about parameters such as address execution, the sign bit of a number, instruction mode bits, and input/output status conditions.
 - The status bits, along with a field in the microinstruction that specifies the branch address, control the branch logic.
 - The branch logic tests the condition, if it is met, then it branches, otherwise it increments the CAR.
 - If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide selection variables for the multiplexer.
 - For unconditional branching, fix the value of one status bit so that there is one read of the branch address from control memory to CAR.

Instruction Mapping

- A specific form of branching occurs when a microinstruction indicates a branch to the initial word in control memory housing the microprogram routine.
- The status bits for this branch type are determined by the bits in the opcode.
- Consider a scenario with a four-bit opcode and a control memory featuring 128 locations.
- The mapping process is responsible for transforming the 4-bit opcode into a 7-bit address for the control memory, as illustrated in the accompanying figure.

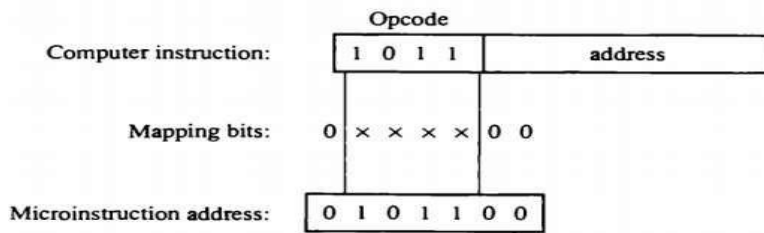


Figure 2.3: Mapping from Instruction Code to Microinstruction Address

- Mapping involves inserting a 0 in the most significant bit of the address, then transferring the four bits of the operation code.
- Additionally, the two least significant bits of the control address register are cleared during this process.
- This configuration enables a microprogram routine to accommodate four microinstructions for each computer instruction.

Subroutines

- Subroutines are specialized programs that utilize other routines to execute specific tasks, and they can be invoked from any location within the main body of the microprogram.
- In numerous microprograms, identical segments of code are often present.
- For efficiency and code reuse, microinstructions can be organized using subroutines that leverage common sections of microcode.
- Subroutines in microprograms must include mechanisms for preserving the return address when calling the subroutine and restoring it during the subroutine's return.
- A dedicated subroutine register is employed as both the source and destination for managing addresses in the context of subroutine operations.

2.2. Example of Microprogram

A microprogram is the result of generating code for the control memory, a crucial step in computer configuration. The diagram illustrates a configuration consisting of two key storage units: main memory, where instructions and data reside, and control memory, housing the microprogram. Various registers, including the Program Counter (PC), Address Register (AR), four processor registers, Data Register (DR), and Accumulator Register (AC), along with two control unit registers the Control Register of Addresses (CAR) and the Register of Subprograms (SBR) compose the system. Information transfer between processor registers occurs through the MUX instead of the bus.

Computer Configuration

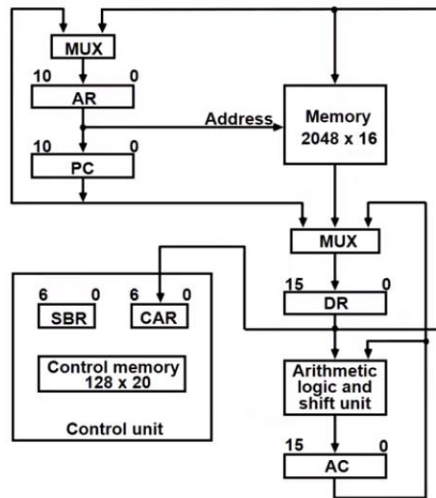
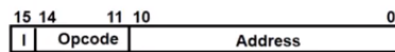


Figure 2.4: Computer Hardware Configuration

The structure of computer instructions is shown in below figure.

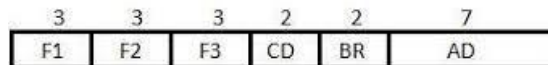
Machine instruction format



Sample machine instructions

Symbol	OP-code	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	if $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address



F1, F2, F3: Microoperation fields
 CD: Condition for branching
 BR: Branch field
 AD: Address field

Figure 2.5: Machine Instruction Format

The microinstruction format comprises 20 bits divided into four distinct sections:

- Three fields—F1, F2, and F3—each consisting of 3 bits, designating specific micro-operations for the computer.
- The CD field, spanning 2 bits, serves to select status bit conditions.

- The BR field, encompassing 2 bits, specifies the branch type to be utilized.
- The AD field holds the branch address within 7 bits.
- Each of the three micro-operation fields has the capacity to specify one of seven available options.
- A maximum of three micro-operations can be chosen for a single micro-instruction.
- Should fewer than three operations be necessary, the code 000 represents a NOP (No Operation).
- The encoding of the three bits in each field facilitates the specification of seven distinct micro-operations, as outlined in the accompanying table.

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

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0-10) \leftarrow PC$	PCTDR

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

Figure 2.6: Microinstruction Field Descriptions

- A five-letter code specifies the type of transfer microoperation.
- The initial two letters designate the source register, while the third letter is "T".
- The final two letters dictate the destination register, such as
- $AC \leftarrow DR$, where F1 = 100 corresponds to DRTAC.
- Additionally, the condition field (CD) employs two bits to denote four different status bit conditions.

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

Figure 2.7: Microinstruction Field Descriptions-BR, CD

- The branch field (BR) is a two-bit segment employed alongside the address field to determine the subsequent microinstruction's address.

BR	Symbol	Function
00	JMP	CAR $\leftarrow$ AD if condition = 1 CAR $\leftarrow$ CAR + 1 if condition = 0
01	CALL	CAR $\leftarrow$ AD, SBR $\leftarrow$ CAR + 1 if condition = 1 CAR $\leftarrow$ CAR + 1 if condition = 0
10	RET	CAR $\leftarrow$ SBR (Return from subroutine)
11	MAP	CAR(2-5) $\leftarrow$ DR(11-14), CAR(0,1,6) $\leftarrow$ 0

Figure 2.8: Branch Microinstruction Address

- Each line within an assembler microprogram defines a symbolic microinstruction, structured into five parts.
- The label field at the beginning of a line might be vacant or may hold a symbolic address, terminating with a colon.
- Micro operation field consists of 1-3 symbols separated by commas. Only one symbol from each field.

If NOP, then translated to 9 zeros

- The condition field specifies one of four conditions
- The branch field has one of the four branch symbols
- The address field has three formats
- Symbolic address – must also be a label
- NEXT symbol to indicate the next address in the sequence
- Empty if branch field is RET or MAP and converted to 7 zeros
- Every line within an assembler microprogram delineates a symbolic microinstruction, typically segmented into five distinct parts.
- The symbolic representation 'ORG' designates the initial address of a microprogram routine.
- For instance, 'ORG 64' positions the initial microinstruction within control memory at address 1000000.

Loading Procedure

- The control memory consists of 128 locations, each comprising 20 bits.
- The initial 64 locations are designated for routines associated with 16 instructions, covering addresses 0 through 63.
- Initiating the fetch routine is possible by commencing at address 64.

- To execute the loading routine, three specific microinstructions are necessary, housed within locations 64 to 66.

Sequence of microoperations in the fetch cycle:

```

AR ← PC
DR ← M[AR], PC ← PC + 1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

```

Symbolic microprogram for the fetch cycle:

```

      ORG 64
FETCH: PCTAR      U JMP NEXT
      READ, INCPC U JMP NEXT
      DRTAR      U MAP

```

Binary equivalents translated by an assembler

Binary address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

Figure 2.9: Symbolic Microprogram

2.3. Control Mechanism Layout

- Distinct microoperations are derived by decoding the control memory in each subarray.
- The outputs from these decoders are linked to corresponding inputs within the processor unit.
- Illustrated in the figure below are three decoders and their interconnections, representing some of the required connections originating from their outputs.

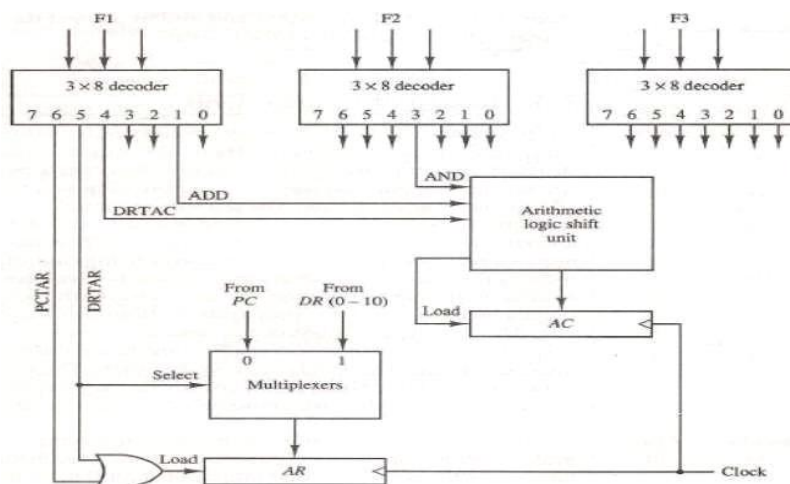


Figure 2.10: Decoding of Microoperation Fields

The microinstruction's three fields, derived from the control memory, are decoded through a 3x8 decoder, delivering eight distinct outputs. Each output is meticulously linked to the respective circuit, thereby initiating the designated micro-operation as detailed earlier.

When F1 holds the value 101 (equivalent to binary 5), the subsequent pulse transition orchestrates the transfer of data from DR (0-10) to AR. Similarly, in the case of F1 being 110 (binary 6), the transfer occurs from PC to AR, denoted as PCTAR. Outputs 5 and 6 from the F1 decoder, as depicted in the figure, interface with the AR load input. Consequently, the activation of either output prompts the transfer of information from the multiplexers to the AR.

These multiplexers are designed to select data from DR when output 5 is active and from PC when output 5 is inactive. Notably, the conversion to AR exclusively takes place during a clock transition when either output 5 or output 6 from the decoder is active.

2.4. Sequential Micro Program

The core elements of a microprogrammed control unit comprise the control memory and the mechanisms responsible for selecting subsequent addresses. This address selection component is termed the microprogram sequencer, pivotal in presenting addresses to the control memory for the execution of microinstructions.

The microprogram sequencer holds the crucial role of providing an address to the control memory, facilitating the reading and execution of microinstructions. Its next address logic defines the precise source of the address loaded into the control address register (CAR).

Illustrated in the figure is the block diagram of the microprogram sequencer, incorporating the control memory to exhibit the interaction between the sequencer and its connected memory unit. The circuit comprises two multiplexers, each serving distinct functions.

The first multiplexer selects an address from four different sources and channels it towards the CAR address control register. Meanwhile, the second multiplexer assesses the selected status bit's value, employing the test outcome in the input logic circuit.

The address for the control memory is sourced from the CAR output, which is simultaneously incremented and fed into one of the multiplexer inputs, also directed to the SBR subroutine register. The other three inputs to this multiplexer originate from:

- The address field within the ongoing microinstruction
- An external source external to the SBR that maps the instruction

- The CD (status) field of the microinstruction, dictating the selection of a status bit in the second multiplexer. If the chosen bit equals 1, the variable T equals 1; otherwise, it equals 0.

The resulting T value, combined with two bits from the BR (branch) field, influences the input logic circuit. This inputs logic, tailored to each sequencer, defines the available operations within the unit.

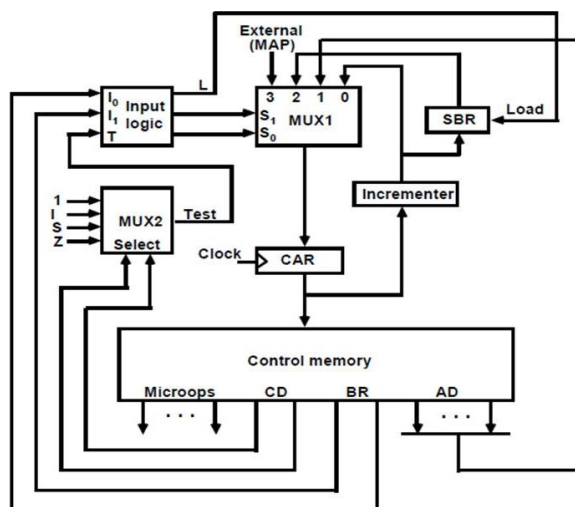


Figure 2.11: Microprogram Sequencer

The input logic circuit depicted in the diagram features three inputs labelled as I_0 , I_1 , and T , and yields three distinct outputs: S_0 , S_1 , and L . S_0 and S_1 serve to designate one of the source addresses for CAR , while the variable L enables the loading input in the SBR .

The binary values assigned to these selection variables determine the routing within the multiplexer. For instance, when S_1, S_0 equals 10, it triggers the selection of multiplexer input number 2, establishing a path from SBR to CAR .

Outlined below is the truth table delineating the behaviour of the input logic circuit,

BR Field	Input			MUX 1		Load <i>SBR</i>
	I_1	I_0	T	S_1	S_0	L
0 0	0	0	0	0	0	0
0 0	0	0	1	0	1	0
0 1	0	1	0	0	0	0
0 1	0	1	1	0	1	1
1 0	1	0	×	1	0	0
1 1	1	1	×	1	1	0

Figure 2.12: Input Logic Truth Table for Microprogram Sequencer

- I1 and I0 inputs mirror the bit values found in the BR field.
- S1 and S0 bit values are established based on the defined function and the pathway within the multiplexer that generates the intended transmission.
- In the call microinstruction (BR = 01), the subroutine register receives the incremented CAR value, contingent upon the fulfilment of the status bit condition (T = 1).
- Utilizing a truth table allows the derivation of simplified Boolean functions for an input logic circuit: specifically, S1 equals I1.

2.5. Central Processing Unit

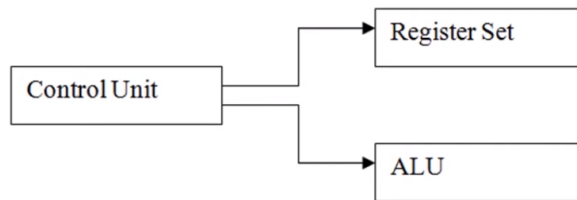


Figure 2.13: Central Processing Unit

The Central Processing Unit (CPU) serves as the computer's core, handling data processing operations.

Within the CPU, the register set stores interim data crucial for executing instructions.

The Arithmetic Logic Unit (ALU) handles the micro-operations necessary for instruction execution.

Meanwhile, the control unit oversees information transfer between registers and directs the ALU on which operations to execute.

To facilitate communication among numerous CPU registers, a shared bus system is employed. This system not only enables direct data transfer between registers but also facilitates diverse micro-operations.

A bus structure, as depicted in Figure 3.2, interconnects CPU registers through two multiplexers (MUX), forming buses A and B. The selected lines within each multiplexer designate a specific register for input data onto the respective bus.

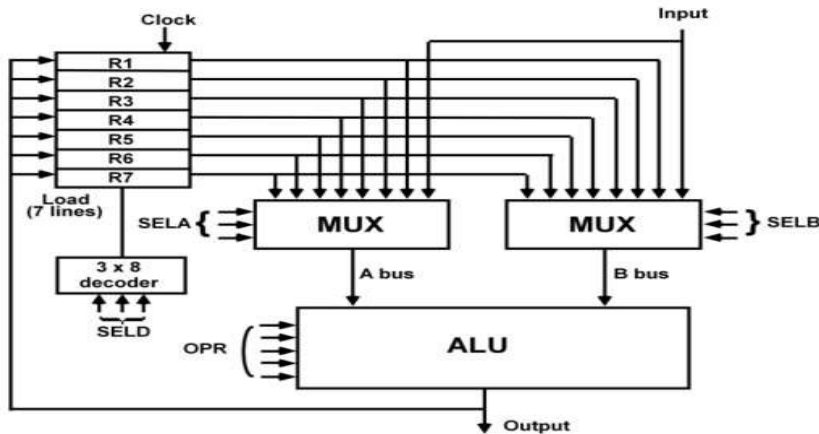


Figure 2.13: A General Organization of Seven CPU Registers

2.6. Storage Organization

The CPU in most computers incorporates a stack, also known as Last-In-First-Out (LIFO), offering valuable functionality.

Reservoir

A stack serves as a storage mechanism where the last stored item becomes the first retrieved one.

The stack's operation can be likened to a pile of trays, where the topmost tray is accessed first.

Within the computer's stack lies a memory unit accompanied by an address register capable of solely counting addresses.

The register designated for holding the stack's address is termed the Stack Pointer (SP), consistently pointing to the topmost item within the stack.

There are two primary operations executed on the stack: insertion and deletion.

The insertion operation is known as PUSH, while the deletion operation is termed POP.

These operations are emulated through the manipulation of the stack pointer (SP) register, involving its incrementation for PUSH and decrementation for POP.

Register Stack

A stack can be situated within a dedicated portion of extensive memory or structured as an assembly of a limited number of memory words or registers.

Illustrated in the figure below is the arrangement of a 64-word register stack.

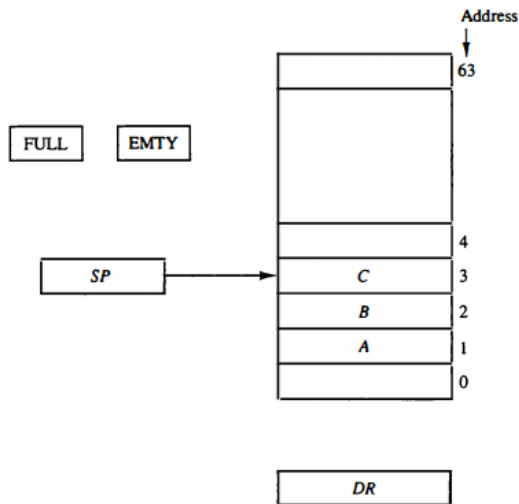


Figure 2.14: 64-word Register Stack

- The Stack Pointer register (SP) retains a binary value representing the address of the stack's topmost word. In a sequence of A, B, and C placed onto the stack, C currently resides at the stack's pinnacle, setting the SP content to 3
- Removing the top entry involves a 'pop' operation, necessitating the reading of the memory word at address 3 while concurrently decrementing the SP content. Consequently, B ascends to the stack's summit, adjusting the SP content to 2.
- For stack modification resembling an addition of a new item, the stack shifts by incrementing the SP and writing the word to the succeeding position in the stack.
- Within a 64-word stack, the SP comprises 6 bits, correlating with $2^6 = 64$.
- Due to the 6-bit capacity of SP, it cannot represent values greater than 63 (111111 in binary).
- Incrementing 63 by 1 results in 0 ($111111 + 1 = 1000000$ in binary), surpassing the SP's six-bit limit.
- Consequently, a one-bit register named FULL is triggered to 1, signaling the stack's full status.
- Conversely, when 000000 is decremented by 1, the outcome is 111111. At this point, a one-bit register named EMPTY is set to 1, indicating an empty stack condition.
- The Data Register (DR) retains binary data designated for writing to or reading from the stack.

Push

- Initially, the Stack Pointer (SP) is reset to 0, setting EMPTY to 1 and CLEARING FULL to 0. Consequently, SP points to the word at address 0, indicating an empty stack that isn't full.
- A new item insertion occurs via the push operation, contingent upon the stack not being full (if FULL = 0).
- The push operation unfolds through the following micro-operations:
- Incrementing the stack pointer to denote the address of the subsequent higher word.
- A memory write operation transposes a word from DR onto the stack's top.
- The first item enters the stack at address 1, while the last item occupies address 0.
- SP reaching 0 signifies a full stack, leading to the setting of FULL to 1.
- This condition arises when, before the last shift, the top item rests at position 63, and after SP incrementation, the last item occupies position 0.
- Once an item settles at position 0, signaling no vacant slots, EMPTY is reset to 0.

SP $\leftarrow$ SP + 1	Increment stack pointer
M[SP] $\leftarrow$ DR	Write item on top of the stack
If (SP = 0) then (FULL $\leftarrow$ 1)	Check if stack is full
EMPTY $\leftarrow$ 0	Mark the stack not empty

Pop

- When the stack isn't empty (EMPTY = 0), a new item can be removed from the stack. The 'pop' operation involves a sequence of specific steps:
- The top item from the stack is loaded into the Data Register (DR).
- Subsequently, the stack pointer is decremented. Upon reaching zero, indicating an empty stack, EMPTY is set to 1.
- This condition arises if the item being read was at position 1. When read, SP decrements and eventually reaches 0, which represents the stack's initial value.
- If the 'pop' operation reads an item from location 0 and is then decremented further, SP transforms into 11111, equivalent to decimal 63 in this setup. At this point, the word at address 0 receives the last item present on the stack.

DR $\leftarrow$ M[SP]	Read item from the top of stack
SP $\leftarrow$ SP - 1	Decrement stack pointer
If (SP = 0) then (EMPTY $\leftarrow$ 1)	Check if stack is empty
FULL $\leftarrow$ 0	Mark the stack not Full

Memory Storage

The CPU in practical implementations manages the stack by dedicating a section of memory specifically for stack operations, utilizing a processor register as the stack pointer.

The visual representation below depicts a fragment of computer memory partitioned into three distinct segments: program, data, and stack.

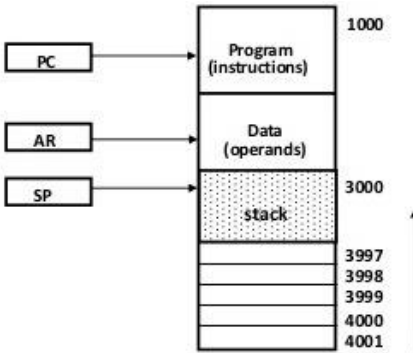


Figure 2.15: Memory Stack Organization

The Program Counter (PC) directs to the subsequent instruction address in the program flow, while the Address Register (AR) specifically points to the data field. Simultaneously, the Stack Pointer (SP) designates the topmost stack position.

These three registers link to a shared address bus, enabling each to furnish memory addresses. During the loading phase, the PC is utilized to retrieve instructions, while the AR comes into play during the execution phase to retrieve operands. On the other hand, the SP facilitates pushing or popping items onto or off the stack.

In the provided Figure , the initial SP value is 4001, and the stack expands with decreasing addresses. Consequently, the first item resides at address 4000, followed by subsequent items at decreasing addresses, culminating at 3000 as the last address available for stack use.

Regrettably, there exist no mechanisms for stack limit checks within this system. Items within the stack interact via the DR data register. Adding a new item involves a push operation, decrementing the SP to signify the subsequent word's address. Subsequently, a write operation from DR to memory inserts a word into the stack's top position.

To remove an item, a pop operation decrements the stack pointer, facilitating the retrieval of the topmost item stored at the address indicated by SP.

The stack pointer decrements to indicate the address of the subsequent word in line.

To add a new item to the stack, a memory write operation transfers a word from the DR into the stack's top position. Conversely, for the removal of an item, the pop operation begins by loading the top item from the stack into the DR. Subsequently, the stack pointer decreases, pointing to the subsequent item within the stack.

Most computer systems lack hardware mechanisms to check for stack overflow (full stack) or underflow (empty stack). However, stack limits can be verified using CPU registers. One register maintains the upper limit (3000 in this instance), while another keeps track of the lower limit (4001 in this case). Following a push operation, the SP is compared against the upper limit register, and after a pop operation, the SP is compared with the lower limit register.

Two micro-operations are necessary for pushing or popping:

- Accessing memory via SP.
- Updating the SP.

The advantage of a memory stack lies in its CPU referencing capability without requiring an address specification. This convenience arises because the stack pointer continually maintains and automatically updates the stack's address.

Reverse Polish Notation

Stack organization proves highly efficient for the evaluation of arithmetic expressions. Typically, these expressions are depicted in infix notation, where each operator is placed between operands.

Consider the expression $A*B+C*D$. Evaluating this necessitates calculating $A*B$, retaining this result while computing $C*D$, and then summing up the two products. The nature of infix notation requires scanning back and forth across the expression to determine the next operation to execute.

Polish mathematician Lukasiewicz introduced an alternative representation for arithmetic expressions known as prefix notation. This scheme, also called Polish notation, positions the operator before the operands. Conversely, postfix notation, or Reverse Polish Notation (RPN), places the operator after the operands.

In the context of representations:

- $A+B$ is Infix notation.
- $+AB$ is Prefix or Polish notation.
- $AB+$ is Post or Reverse Polish notation.

Reverse Polish notation, in particular, is well-suited for stack manipulation. An expression like $A*B+C*D$ is written in RPN as $AB*CD*+$ and can be evaluated methodically:

- Scan the expression left to right.
- Upon reaching an operator, execute the operation on the two operands preceding it.
- Replace the operands and operator with the result.
- Continue this process until no operators remain.

For instance, in the expression $A*B+C*D$:

- The $*$ operator after A and B initiates the $A*B$ operation and substitutes A, B, and $*$ with the result.
- Subsequently, encountering the $*$ operator and the preceding operands C and D triggers the $C*D$ operation, updating the C, D, and $*$ locations with the outcome.
- Finally, the $+$ operator combines the two obtained products to produce the final result.

When converting from infix to Reverse Polish notation, it's crucial to consider the operational hierarchy delineated for infix notation. This hierarchy dictates performing operations within inner parentheses first, then outer parentheses, and prioritizing multiplication/division before addition/subtraction operations.

Evaluation of Arithmetic Expressions

Reverse Polish notation (RPN), coupled with a stacked register setup, stands as one of the most efficient methods recognized for assessing arithmetic expressions.

This method finds application in various electronic calculators and computer systems due to its effectiveness.

To illustrate its efficacy, let's take an arithmetic expression like $(3*4) + (5*6)$.

In RPN, this expression is represented as $34 * 56 +$

The exploration of this method, the operations performed on the stack are depicted in Figure 8-5.

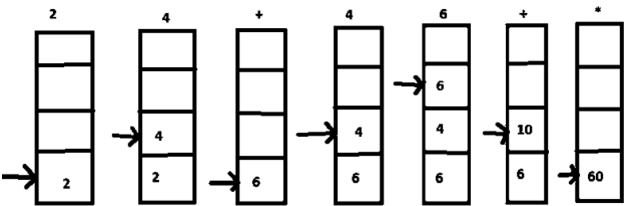


Figure 2.16: Stack Operation to Evaluate $(2+4)*(4+6)$

Each box in the sequence represents a single stack operation, with the arrow indicating the top of the stack at each step.

While scanning the expression left to right, two operands are encountered: 3 is placed into the stack first, followed by 4.

The multiplication operator (*) is encountered, prompting the multiplication of the top two items in the stack. The resulting product replaces these two original operands atop the stack.

Continuing, the subsequent operands, 5 and 6, are added to the stack. The "*" operation multiplies these two numbers, and the resulting product replaces them in the stack.

Finally, the addition operation takes place, summing the two highest numbers within the stack and yielding a result of 42.

2.7. Formats of Instructions

The instruction format, typically depicted within a rectangular box, represents the instruction bits as they exist in memory words or control registers.

These bits are segmented into distinct groups referred to as fields, and among the most prevalent fields are:

- Operation Code (opcode) field, specifying the intended operation.
- Address field, designating either a memory address or a processor register.
- Mode field, determining the method for establishing the operand or effective address.

Instructions within computers can exhibit varied lengths, accommodating different quantities of addresses. The count of address fields present in a computer's instruction format is contingent upon the internal configuration of its registers.

Computers often align with one of three principal CPU organizations:

- Single accumulator organization.
- General register organization.
- Stack organization.

Organization with One Accumulator

In an architecture following an accumulator type, all computations occur using a dedicated accumulator register.

The instruction format within this computer model utilizes a single address field. For instance, an arithmetic addition instruction is represented symbolically as "ADD X," where X

refers to the operand's address. The execution of this ADD instruction implies an operation where AC (the accumulator register) undergoes $AC + M[X]$, with $M[X]$ denoting the memory word located at address X.

2.8. General Registration Organization

This computer model's instruction format typically necessitates three register address fields. For instance, the instruction "ADD R1, R2, R3" signifies the operation $R1 \leftarrow R2 + R3$.

To condense the number of address fields from three to two, the instruction can be simplified if the destination register matches one of the source registers. For example, "ADD R1, R2" implies the operation $R1 \leftarrow R1 + R2$. Here, only the register addresses for R1 and R2 are explicitly mentioned.

In general register computers, the instruction format utilizes either two or three address fields. Each of these fields has the capability to specify either a processor register or a memory word.

Consider the instruction "ADD R1, X," indicating the operation $R1 \leftarrow R1 + M[X]$. This instruction features two address fields: one for the R1 register and another for the memory address X.

Stack Organization

- In a stacked CPU, PUSH and POP instructions incorporate an address field, where PUSH X relocates the word found at address X to the topmost position in the stack.
- The stack pointer undergoes automatic updates during these operations.
- Operational instructions in stacked computers don't necessitate an address field. These instructions operate on the top two items within the stack.
- For instance, the ADD instruction in a stacked computer solely consists of an opcode, lacking an address field. It functions by extracting the two highest numbers from the stack, performing addition, and depositing the sum back onto the stack.
- There's no explicit requirement to specify operands via an address field as all operands inherently reside within the stack.
- Most computer systems align with three distinct organizational structures, although some amalgamate features from multiple structures.
- Evaluating the impact of address numbers on computer programs, consider an arithmetic statement: $X = (A + B) * (C + D)$.

- Analyze the effect of zero, one, two, or three address instructions employing symbols like ADD, SUB, MUL, and DIV for arithmetic operations, MOV for transfer operations, and LOAD/STORE to facilitate memory-to-AC register transfers.
- Given operands in memory at addresses A, B, C, and D, and the requirement to store the result at memory address X, suppose the CPU features general registers R1, R2, R3, and R4.

Three Directions to the Address

Three-address instruction formats offer the flexibility to designate a processor register or memory operand in each address field.

An assembly language program is presented below, accompanied by comments that delineate the register transfer operation for each instruction.

The notation $M[A]$ denotes the operand located at the memory address represented by A.

The merit of the three-address format lies in the concise nature of constructing short programs, especially when evaluating arithmetic expressions.

Conversely, the downside emerges from the extensive binary-coded instructions necessary to specify three addresses, demanding an excessive number of bits.

Using three address instructions

ADD R1, A, B	; $R1 \leftarrow M[A] + M[B]$
ADD R2, C, D	; $R2 \leftarrow M[C] + M[D]$
MUL Y, R1, R2	; $M[Y] \leftarrow R1 * R2$

Two Address Instructions

Dual-address instruction formats allow each address field to specify either a processor register or a memory word.

The program designed to evaluate $X = (A+B) * (C+D)$ follows a specific sequence of instructions:

The MOV instruction serves the purpose of transferring operands between processor memory and registers.

In these instructions, the initial symbol listed is considered both the source and the destination, facilitating the transfer of the operation's result.

Instructions for a Single Address

Single-address instructions rely solely on the implied accumulator (AC) register for data manipulation purposes.

Although multiplication and division typically necessitate a second register, for simplicity's sake, let's disregard this additional register and assume that AC encompasses the outcome of all operations.

Consider a program designed to compute $X = (A+B) * (C+D)$.

All operations are executed between the AC register and the memory operand.

The address T represents the location in memory designated for temporarily storing intermediate results.

LOAD	A	; AC $\leftarrow$ M[A]
ADD	B	; AC $\leftarrow$ AC + M[B]
STORET		; M[T] $\leftarrow$ AC
LOAD	C	; AC $\leftarrow$ M[C]
ADD	D	; AC $\leftarrow$ AC + M[D]
MUL	T	; AC $\leftarrow$ AC * M[T]
STOREX		; M[X] $\leftarrow$ AC

Instructions for Null Address

- In a stack-oriented computer, ADD and MUL instructions operate without an address field, utilizing the stack arrangement.
- However, for PUSH and POP instructions, an address field becomes essential to specify the operand interacting with the stack.
- The provided program outlines how the expression $X = (A+B) * (C+D)$ would be expressed within a stack-based computer.
- Converting arithmetic expressions for stack computation requires transforming the expression into Reverse Polish Notation (RPN).
- This type of computer is labeled as "zero address" due to the absence of an address field in the calculation instructions, aligning with its stack-based nature.

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

2.9. RISC Instructions

In a typical RISC processor's instruction set, the primary communication between the memory and CPU occurs solely through load and store instructions. These instructions facilitate the transfer of data between memory and CPU registers. However, all other instructions execute exclusively within CPU registers and don't reference memory.

The LOAD and STORE instructions operate with one memory address and one register address, enabling the transfer of data between memory and registers. Conversely, compute-type instructions involve three addresses, all of which specify processor registers.

To illustrate, consider the program designed to evaluate the expression $X = (A + B) * (C + D)$:

- LOAD instructions initiate the transfer of operands from memory into CPU registers.
- Subsequent addition and multiplication operations are executed solely within the registers, without involving memory access.
- Finally, the outcomes of these calculations are stored back into memory through storage instructions.

This process showcases how load and store instructions facilitate data transfer between memory and registers, allowing arithmetic operations to occur within the CPU registers, leading to the storage of the final results back into memory.

Table 2.1: RISC VLSI Set

Instruction	Operands	Comments
ADD	S1,S2,Rd	$Rd \leftarrow S1 + S2$ integer add
ADDC	S1,S2,Rd	$Rd \leftarrow S1 + S2 + \text{carry}$ add with carry
SUB	S1,S2,Rd	$Rd \leftarrow S1 - S2$ integer subtract
SUBC	S1,S2,Rd	$Rd \leftarrow S1 - S2 - \text{carry}$ subtract with carry
AND	S1,S2,Rd	$Rd \leftarrow S1 \& S2$ logical AND
OR	S1,S2,Rd	$Rd \leftarrow S1 S2$ logical OR
XOR	S1,S2,Rd	$Rd \leftarrow S1 \text{ xor } S2$ logical EXCLUSIVE OR
SLA	S1,S2,Rd	$Rd \leftarrow S1 \text{ shifted by } S2$ shift left arithmetic
SRA	S1,S2,Rd	$Rd \leftarrow S1 \text{ shifted by } S2$ shift right arithmetic
SLL	S1,S2,Rd	$Rd \leftarrow S1 \text{ shifted by } S2$ shift left logical
SRL	S1,S2,Rd	$Rd \leftarrow S1 \text{ shifted by } S2$ shift right logical
LDL	(Rx)X,Rd	$Rd \leftarrow M[Rx+X]$ load long
LDSU	(Rx)X,Rd	$Rd \leftarrow M[Rx+X]$ load short unsigned
LDSS	(Rx)X,Rd	$Rd \leftarrow M[Rx+X]$ load short signed
LDBU	(Rx)X,Rd	$Rd \leftarrow M[Rx+X]$ load byte unsigned
LDBS	(Rx)X,Rd	$Rd \leftarrow M[Rx+X]$ load byte signed
STL	Rm,(Rx)X	$M[Rx+X] \leftarrow Rm$ store long
STS	Rm,(Rx)X	$M[Rx+X] \leftarrow Rm$ store short
STB	Rm,(Rx)X	$M[Rx+X] \leftarrow Rm$ store byte
JMP	COND,X(Rm)	$pc \leftarrow X+Rm$ conditional jump
JMPR	COND,Y	$pc \leftarrow pc + Y$ conditional relative
CALL	Rm,X(Rn)	$Rm \leftarrow pc, \text{ next}$ $pc \leftarrow X+Rn, CWP--$
CALLR	Rm,Y	$Rm \leftarrow pc, \text{ next}$ $pc \leftarrow pc + Y, CWP--$
RET	Rm,X	$pc \leftarrow Rm+X, CWP++$
GTLPc	Rm	$Rm \leftarrow \text{last pc}$ get last pc
GTIN	Rm	$Rm \leftarrow \text{INR}$ get interrupt number

2.10. Addressing Modes

During program execution, the method of selecting operands hinges on the instruction's addressing mode. Computer systems utilize addressing mode techniques to serve two primary purposes:

- Enhancing programming flexibility for users by offering functionalities such as memory pointers, loop control counters, data indexing, and program relocation.
- Minimizing the number of bits required in the instruction's address field.

Implied Mode

- This mode implies operands within the instruction itself.
- For instance, the instruction "accessory accumulator" is in implied mode as it implies the operand is in the accumulator register.
- All reference register instructions utilizing the accumulator fall under implicit mode.
- In a zero-address stack-based computer, the implicit mode instruction is employed.

Immediate Mode

- In this mode, the instruction specifies the operand within itself.
- Immediate mode instructions contain an operand rather than an address field.
- Useful for initializing registers to constant values.
- The instruction's address field can denote a memory word or a processor register.
- When the address denotes a processor register, it's in register mode.

Register Mode

- Operands reside within CPU registers.
- The specific register is chosen from the instruction's register field.

Register Indirect Mode

- This mode specifies a CPU register whose contents indicate the operand's memory address.
- The selected register holds the address of the operand, not the operand itself.
- This mode benefits from using a few more bits to select a register within the instruction's address field than directly specifying the memory address.

Auto-Increment/Decrement Mode

- Similar to indirect register mode but involves incrementing or decrementing the register after/before using its value to access memory.
- The controller retrieves the operand from memory using the instruction's address field.

- The address field might either be the operand's address or the base for computing the operand's address.
- Primary CPU addressing modes encompass direct and indirect addressing.

Direct Addressing Mode

- In this mode, the effective address equals the instruction's address part.
- The operand exists in memory, with its address directly provided by the instruction's address field.
- In branch instructions, the address field denotes the actual branch address.

Indirect Addressing Mode

- The instruction's address field specifies the address where the effective address resides in memory.
- The control fetches the instruction from memory and utilizes its address part to access memory again, reading the effective address.
- Various addressing modes mandate adding the instruction's address field to CPU register contents.
- The effective address is obtained by: $\text{Effective address} = \text{address part of instruction} + \text{CPU register contents}$.
- This CPU register can be a program counter, index register, or base register.

Relative Addressing Mode

- Here, the program counter's contents are added to the instruction's address portion to derive the effective address.

Indexed Addressing Mode

- Contents of the index register are added to the instruction's address part to determine the effective address.
- The index register holds a specific index value.

Base Register Addressing Mode

- Similar to indexed addressing mode but involves adding the base register contents to the instruction's address portion for the effective address.
- Unlike indexed addressing, this mode uses a base register instead of an index register.

Numerical Example

To illustrate these different modes' effects on instructions, we'll showcase how addressing modes impact instructions in a specific scenario.

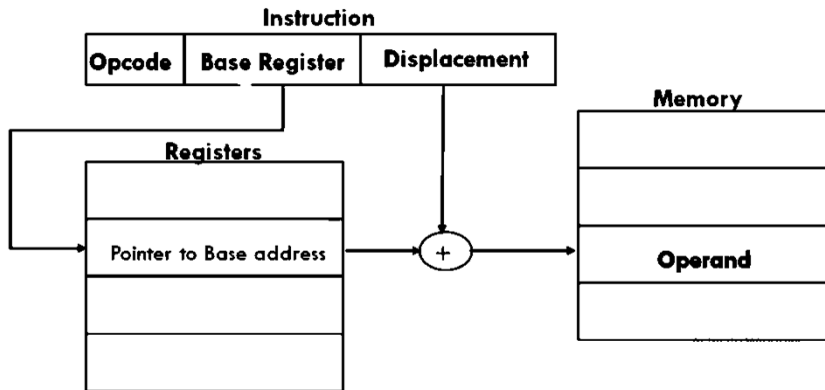


Figure 2.17: Base Register Addressing Mode

Implicit Direct Address Mode

- The instruction at addresses 200 and 201 is a "load to AC" instruction with an address field pointing to 500.
- In this mode, the effective address and operand to be loaded into AC are both 500.

Immediate Addressing Mode

- The second word of the instruction (500) serves as the operand, directly loading 500 into AC.

Indirect Addressing Mode

- The address stored at memory address 500 is used as the effective address.
- Consequently, the effective address becomes 800, and the operand loaded into AC is 300.

Relative Addressing Mode

- The effective address in relative mode is $500 + 202$ (PC value during execution) = 702.
- The resulting operand loaded into AC is 325.

Indexed Addressing Mode

- With XR content (index register) set to 100, the effective address becomes $\text{XR} + \text{address part (500)} = 600$.
- The operand loaded into AC from this address is 900.

Register Addressing Mode

- The operand is sourced from R1, containing 400, and is loaded into AC.

Indirect Register Addressing Mode

- The contents of R1 (400) serve as the effective address.
- Resultantly, the operand loaded into AC from this address is 700.

Auto-Increment Addressing Mode

- Similar to indirect register mode, but R1 is incremented to 401 after executing the instruction.
- The operand loaded into AC from the incremented address (400) is now 700.

Auto-Decrement Addressing Mode

- Preceding instruction execution, R1 is decremented to 399.
- Consequently, the operand loaded into AC from this decremented address (400) is 450.

2.11. Data Transfer and Manipulation

Computing instructions typically fall into three primary categories:

- Instructions involving the transfer of data
- Commands designed for data manipulation
- Directives aimed at program operation

Data Transfer Instructions

- These instructions facilitate moving data within a computer without altering its content.
- Common transfers occur between memory and processor registers, between processor registers and input/output, and among processor registers themselves.

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

Figure 2.18: Data Transfer Instruction

Data Transfer Instructions Table

- Load: Transfers data from memory to a processor register, typically an accumulator.
- Store: Moves data from a processor register to memory.
- Move: Used in systems with multiple CPU registers for transferring data between registers, as well as between CPU registers and memory or two memory words.
- Swap: Exchanges information between two registers or between a register and a memory word.

- Input/Output: Transfers data between processor registers and input/output terminals.
- Push/Pop: Shifts data between processor registers and the memory stack.

Addressing Mode Convention

- Different computers employ unique mnemonics to differentiate addressing modes.
- For example, the accumulator load instruction showcases various addressing modes in a symbolic instruction language, illustrating the actual transfer performed in each scenario.

Addressing Mode Conventions Table (Excerpt)

- ADR: Represents an address.
- NBA: Denotes a number or operand.
- X: Indicates the index register.
- @: Symbolizes indirect addressing.
- R1: Refers to the processor register.
- AC: Represents the accumulator register.
- \$: Makes the address relative to the PC program.
- #: Precedes an operand in immediate mode.

Register Mode: Specifies the processor register name directly.

Indirect Register Mode: Encloses the register name containing the memory address in parentheses.

Auto-Increment Mode: Differentiated from indirect register mode by adding a plus sign after the register in parentheses. Conversely, auto-decrement mode would use a minus sign.

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

Figure 2.19: 8-Addressing Mode for load instruction

Data Handling Instructions

Data manipulation instructions are pivotal in performing operations on data, enhancing the computing capabilities of a computer system.

These instructions commonly fall into three fundamental categories:

- **Arithmetic Instructions:** These operations involve arithmetic computations, including addition, subtraction, multiplication, and division, facilitating mathematical processing on data.
- **Logical and Bit Manipulation Instructions:** This category involves logical operations such as AND, OR, NOT, as well as bit manipulation, enabling the alteration and evaluation of individual bits within data.
- **Sorting Instructions:** These instructions focus on organizing and arranging data in specific orders, facilitating the sorting of datasets for various computational purposes.

Arithmetic Instructions

- The fundamental arithmetic operations encompass addition, subtraction, multiplication, and division.
- Most computer systems include instructions for all four operations.
- However, some smaller systems may only offer instructions for addition and sometimes subtraction, requiring intermediate software subroutines to generate multiplication and division functionalities.

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

Figure 2.20: Typical Arithmetic Instructions

Typical Arithmetic Instructions List

- A series of typical arithmetic instructions is provided in Table 8-7.

Increment and Decrement Instructions

- An increment instruction adds 1 to the value stored in a register or memory word.
- Incrementing a number represented by all ones results in a number represented by all zeroes.
- Conversely, a decrement instruction subtracts 1 from the value stored, turning a number represented by all zeroes into a number represented by all ones.

Arithmetic Instruction Data Types

- Arithmetic instructions can handle various data types, dictated by the opcode.
- These instructions may specify fixed or floating-point data, binary or decimal data, and single or double precision data.

Examples of Arithmetic Mnemonics

- The mnemonics for three addition instructions denoting different data types are:
ADDI: Adds two binary integers
ADDF: Adds two floating-point numbers
ADDD: Adds two decimal numbers in Binary Coded Decimal (BCD) format.

Special Arithmetic Instructions

- A special flip-flop is utilized for storing the carry from an operation.
- Instructions like "add carry" execute addition on two operands along with the carry value from the previous calculation.
- Similarly, "subtract with borrow" subtracts two words considering any borrow from a prior subtraction.
- The negation instruction creates a 2's complement number, effectively altering the sign of an integer in 2's complement representation.

Logical and Bit Manipulation Instructions

Logical instructions are designed to perform binary operations on sets of storage bits or registers, offering valuable tools for manipulating binary-coded information.

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Figure 2.21: Logical and Bit Manipulation Instructions

Key Aspects of Logical Instructions

- These instructions are instrumental in manipulating individual bits or groups of bits that represent binary-coded information.

- They handle each bit of the operand individually, treating it as a Boolean variable.

Capabilities and Usage of Logical Instructions

- Proper application of logical instructions allows for altering bit values, removing bit groups, or inserting new bit values into operands stored within memory words or registers.
- Typical logic and bit manipulation instructions are outlined in the accompanying table.

Specific Logical Instructions

- "Clear" instruction replaces the specified operand with 0.
- "Complement" instruction generates a 1's complement by inverting all bits within the operand.
- Instructions like AND, OR, and XOR perform their corresponding logical operations on individual bits within operands.
- Logical instructions can also execute bit manipulation operations.

Bit Manipulation Operations

- Three primary bit manipulation operations are viable: clearing a selected bit to 0, setting it to 1, or inverting it.
- For instance, the AND instruction clears a bit or selected bit group within an operand.
- Conversely, the OR instruction sets a bit or a chosen bit group within an operand.
- Similarly, the XOR instruction selectively complements bits within the operand.

Bit Manipulation Instructions

- Additional bit manipulation instructions, as presented in the table, handle operations on individual bits, including clearing, setting, or appending a carry.
- For instance, a flip-flop governing an interrupt function can be enabled or disabled using these bit manipulation instructions.

Sorting Instructions

Shift operations involve moving bits within a word either to the left or right, with the shifted end bit influencing the type of shift performed.

Key Aspects of Shift Instructions

Shift instructions encompass logical shifts, arithmetic shifts, and rotation-type operations, executed either to the right or left.

Sort Commands Table

The provided table outlines four distinct types of sort commands.

SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate carry left
RCR	Rotate carry right

Insights into Shift Operations

- A logical shift alters the bit at the end position.
- The position differs based on whether it's a right or left shift.

Arithmetic Shifts and their Characteristics

- Arithmetic shifts adhere to the principles of signed-2's complement numbers.
- In an arithmetic shift right, the sign bit, typically on the leftmost side, must be preserved.
- The sign bit shifts to the right along with the number, maintaining its value.

Differentiating Logical and Arithmetic Shifts

- The arithmetic shift-left operation is akin to the logical shift-left, inserting a 0 at the end position.

Role of Rotate Instructions

- A rotate instruction executes a circular shift, ensuring no bits are lost; instead, they circulate from one end to the other.
- Rotate via carry treats the carry bit as an extension of the register being rotated.

Functionality of Rotate Via Carry

For instance, a rotate-left instruction via carry shifts the register to the left, transferring the carry bit to the rightmost position and moving the leftmost bit to the carry position simultaneously.

Program Control

Instructions governing program control dictate the circumstances for altering the program counter's contents, potentially interrupting the sequence of instruction execution. This control mechanism not only influences program flow but also enables branching into various program segments, as exemplified in the accompanying table.

SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate carry left
RCR	Rotate carry right

Distinct Characteristics of Program Control Instructions

- Branch and jump instructions come in conditional or unconditional forms. Unconditional branches directly lead to a specified address, whereas conditional branches require specific conditions, like positive or zero, to execute the branch.
- Skip instructions, being zero-address instructions, don't necessitate an address field. Conditional skip instructions skip the subsequent instruction by incrementing the program counter based on specific conditions being met.
- Call and return instructions are integral to managing subroutines.
- Comparison instructions conduct a subtraction between two operands, yielding certain status bit conditions without preserving the operation's outcome. Similarly, test instructions perform logical AND operations on operands, updating status bits without altering the operands or retaining the result.

Interrupt Handling in Computer Systems

Interrupt handling involves transferring program control from a running program to another service program triggered by internal or external signals. This process, akin to calling a subroutine, holds three key distinctions:

Signal Initiation and Address Determination

Interrupts stem from internal or external signals and, unlike regular subroutines, their service program address is hardware-determined.

Types of Interrupts

External Interrupts

These originate from various external sources like I/O devices, timing devices, or power monitoring circuits. Examples include I/O device requests, completed data transfers, timer events, or power failures.

Internal Interrupts (Traps)

These arise due to improper use of instructions or data, often associated with internal errors like register overflow, division by zero attempts, invalid opcodes, stack overflows, or protection violations. They're triggered by signals within CPU hardware.

Software Interrupts

These interrupts are initiated by specific instructions, behaving more like interrupts than regular subroutine calls. They execute special call instructions in the software.

Reduced Instruction Set Computers (RISC) vs. Complex Instruction Set Computers (CISC):

CISC Characteristics

- Large instruction set (100-250 instructions) including specialized but less commonly used tasks.
- Offers a wide range (5-20) of addressing modes.
- Variable-length instructions that manipulate operands in memory.

RISC Attributes

- Relatively fewer instructions and addressing modes.
- Focuses on memory access through loading and storing instructions.
- Operates most operations within CPU registers.
- Employs fixed-length, easily decoded instructions with one-cycle execution.
- Uses hard-wired instead of micro-programmed control.
- Houses a relatively larger number of registers in the processor unit, optimizing an efficient instruction pipeline.

Short Answer Questions

1. What are buffers?
2. What is the function of ALU?
3. What is the function of CU?
4. What are the basic operations of a computer?
5. What are the registers generally contained in a processor?
6. What are the steps in starting the program?
7. Define interrupt?
8. Define bus?
9. What is the purpose of the buffer register?

10. Compare single bus structure and multi bus structure?
11. What is system software? Give an example?
12. What is a pipeline?
13. What is byte addressable memory?
14. What are status code symptoms?
15. Define the addressing mode.
16. Define different addressing modes.
17. Explain MDR and MAR.
18. Name two special registers.
19. Describe the two control signals used for register transfer.
20. What does subject teaching mean?
21. Define the registry file.
22. What factors determine control signals?
23. Explain wire control.
24. What are the characteristics of wire control?
25. What is a control word?
26. Define microroutine and microinstructions.
27. What is a control store?
28. List the four steps in pipelining.
29. What is a buffer used for?
30. What is data risk?
31. What are the dangers of teaching?
32. When is a memory unit called RAM?
33. What is MMU?
34. What is the word?
35. Define static memory?
36. Why are SRAMs said to be volatile?
37. What are the properties of DRAM?
38. Define memory latency?
39. What is a memory controller?
40. What are the mapping techniques?
41. What is a DMA controller?

Long Answer Questions

1. Describe stack organization.
2. What are the operations that take place in the "interrupt cycle"? Discuss.
3. Explain in detail "The control memory is part of the control unit".
4. Elaborate "The function of the controller is to control the flow of data and coordinate with the components involved in the process".
5. Microprogrammed control is simpler for the design of the control unit. do you agree Justify.
6. Discuss in detail about "A computer system consists of two memory units namely main memory and control memory."
7. Explain instruction execution of RISC processors.
8. How would you optimize RISC pipelining?
9. How concatenation optimization is achieved in the RISC architecture. Explain.
10. Discusses the motivation for CISC.
11. List the characteristics of CISC architecture.
12. Compare CISC and RISC processors.
13. Explain the instruction formats for the Pentium processor.
14. Explain arithmetic instructions and data transfer instructions.

Multiple Answer Questions

1. How information is stored in floating point representation:
 - a) Boolean values
 - b) Integers
 - c) Real integers
 - d) Integers
2. How subtraction is typically executed in computers:
 - a) Supplement 9
 - b) Supplement 10
 - c) Supplement 1
 - d) Supplement 2
3. The component utilized to store a single bit of data:
 - a) Register
 - b) Encoder
 - c) Decoder

- d) Flip Flop
- 4. Identifying a non-weighted code among the options:
 - a) Decimal number system
 - b) Exceeding 3 codes
 - c) Binary number system
 - d) None of these
- 5. Characteristics of symbolic instructions:
 - a) Alphabetic codes replace binary numbers in machine language
 - b) Does it provide the simplest programming language?
 - c) No translation required into machine language
 - d) None of these
- 6. The product of the multiplier register and the multiplier register in Booth's algorithm is (11101) & (1100). What will be the result?
 - a) (812) 10
 - b) (-12) 10
 - c) (12) 10
 - d) (-812)
- 7. What characteristic of RAM makes it unsuitable for permanent storage?
 - a) (a) Too slow
 - b) (b) Unreliable
 - c) (c) Volatility
 - d) (d) Bulky
- 8. (2FA0C)16 is equivalent to:
 - a) (195,084)10
 - b) (0010111110100000 1100)2
 - c) Both (A) and (B)
 - d) None of these
- 9. What term describes the average time required to reach a storage location in memory and retrieve its contents?
 - a) Search time
 - b) Turnaround time
 - c) Access time
 - d) Transmission time

10. In signed binary division, with a dividend of $(11100)_2$ and a divisor of $(10011)_2$, what's the result?
- a) $(00100)_2$
 - b) $(10100)_2$
 - c) $(11001)_2$
 - d) $(01100)_2$
11. How is the expression $A*B+C*D$ written in reverse Polish notation?
- a) $AB*CD*+$
 - b) $A*BCD*+$
 - c) $AB*CD+*$
 - d) $A*B*CD+$
12. In computer systems, what is typically used for subtraction?
- a) Supplement 9
 - b) Supplement 10
 - c) Supplement 1
 - d) Supplement 2
13. What characteristic of RAM makes it unsuitable for permanent storage?
- a) Too slow
 - b) Unreliable
 - c) Volatility
 - d) Bulky
14. Which storage element holds a single bit of data?
- a) Register
 - b) Encoder
 - c) Decoder
 - d) Flip Flop
15. In a memory-mapped I/O system, which instruction won't be present?
- a) LDA
 - b) IN
 - c) ADD
 - d) OUT
16. What type of architecture is Von Neumann architecture?
- a) SISD
 - b) SIMD

- c) MIMD
 - d) MISD
17. What technique doesn't update data in memory?
- a) Virtual memory
 - b) Main memory
 - c) Auxiliary memory
 - d) Buffer memory
18. What's the term for the memory unit accessed by the content?
- a) Read only memory
 - b) Programmable memory
 - c) Virtual memory
 - d) Associative memory
19. What does the 'ORG 0' instruction refer to?
- a) Machine instructions
 - b) Pseudo instructions
 - c) High-level teaching
 - d) Memory instruction
20. In the translation from symbolic program to binary code, how many passes are involved?
- a) Two passes
 - b) Direct
 - c) Three passes
 - d) Four passes
21. What's indicated by a floating-point number with a '0' in the Most Significant Bit (MSB) of the mantissa?
- a) Overflow
 - b) Underflow
 - c) Significant number
 - d) Undefined
22. What does the BSA instruction stand for?
- a) Branch and store the accumulator
 - b) Branch and store the return address
 - c) Branch and offset address
 - d) Branch and display accumulator

23. What's the term for an arrangement of logic gates with inputs and outputs?
- a) Combinational circuit
 - b) Logic circuit
 - c) Draft circuits
 - d) Register
24. What does MIMD stand for?
- a) Multiple data of multiple instructions
 - b) Memory data of multiple instructions
 - c) Multiple data memory instruction
 - d) Multiple information memory data
25. How many registers can be specified by a k-bit array?
- a) 3k registers
 - b) 2k registers
 - c) registers K2
 - d) registers K3
26. What's the term for the time interval between adjacent bits?
- a) Word time
 - b) Bit time
 - c) Turn time
 - d) Cut time

UNIT III - DATA REPRESENTATION

Data Representation: Data Types, Addition, Fixed Representation, Floating Point Representation. Computer Arithmetic: Addition, Subtraction, Multiplication, Division, Floating Point Arithmetic. Decimal Arithmetic Unit, Arithmetic Operations.

3.1. Data Representation

Data representation is a method used to represent information in computers.

- The 0s and 1s used to represent digital data are called binary numbers.
- A bit is 0 or 1 in the numerical representation of data.
- In digital computers, user input is first converted into electrical pulses and sent; these can be represented by two special states, ON and OFF.
- The ON state can be represented by "1" and the OFF state by "0". The ON and OFF sequences create electrical signals that the computer can understand.

3.2. Data Types

- Holds either record data or control data.
- Control data involves a bit or a collection of elements specifying the necessary system commands for the operational file.
- In digital computer records, data types fall into distinct categories:
- Numerical values utilized in computations.
- Alphabetic characters for data processing.
- Special characters tailored for specific purposes.
- All non-binary data types are encoded in binary form.

3.3. Number Systems

1. Number systems serve as the foundation for expressing values in computer memory, offering distinct representations for each stored or retrieved value.
2. They provide diverse methods of representing numbers.
3. Each number system possesses a distinct base associated with its symbols, termed as the base of the number system, defining the count of symbols employed.
4. Computer architecture accommodates various codes:
 - Binary
 - Octal
 - Decimal

- Hexadecimal

Binary Number System

In binary, there exist two digits: 0 and 1, representing values within the system. Its base, being just two, derives from these two digits.

Octal Number System

Octal employs eight digits, ranging from 0 to 7, where each digit represents values from 0 to 7. The base of the octal system, being 8, is due to its use of these eight digits exclusively.

Decimal Number System

Decimal encompasses ten digits, from 0 to 9. Each digit signifies values ranging from 0 to 9. With its base as 10, the decimal system relies on these ten digits for representation.

Hexadecimal Number System

Hexadecimal uses sixteen characters, combining numerals 0-9 and letters A-F, signifying values from 0 to 15. Its base, 16, reflects the presence of these 16 characters in its mathematical representation.

3.4. Character Data Representation

There are many ways to represent characters. Some of these are discussed below:

- ASCII
- EBCDIC
- ISCH
- Unicode

ASCII

- Represents characters using 7 bits in computer memory.
- Assigns a unique number to each character, converting it into a 7-bit binary code.
- Covers standard keyboard characters adequately.

EBCDIC

- An 8-bit code similar to ASCII, used in IBM computers.
- Enables the encoding of 256 characters.

ISCH

- Stands for Indian Standard Code for Information Interchange.

- Utilizes 8 bits for representing various Indian scripts.

Unicode

- Overcomes limitations of 8-bit ASCII, which can only represent 256 characters.
- Aims to encompass symbols and characters from all world languages.
- Initially used 16 bits, allowing representation of up to 65,536 characters.

Number system	Base (Radix)	Used digits	Example
Binary	2	0,1	(11110000)2
Octal	8	0 to 7	(360)8
Decimal	10	0 to 9	(240)10
Hexadecimal	16	0 to 9 and A to F	(F0)16

Every octal digit represents three binary digits, while each hexadecimal digit corresponds to four binary digits.

Rather than specifying numbers in binary form, refer to them in octal or hexadecimal and reduce the number of digits by 1/3 or 1/4, respectively.

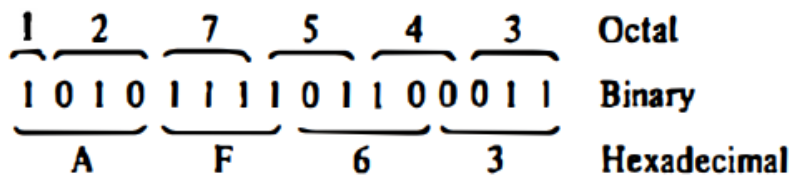


Figure 3.1: Binary Octal, Hexadecimal Conversion

Table 3.1: Binary Coded Octal Number

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	
1	001	1	
2	010	2	
3	011	3	
4	100	4	
5	101	5	
6	110	6	
7	111	7	
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

Table 3.2: Binary Coded Hexa Decimal Number

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	↑ Code for one hexadecimal digit ↓
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	
7	0111	7	
8	1000	8	
9	1001	9	
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

A binary code comprises n bits, allowing for up to 2^n unique combinations. To represent the ten decimal digits, a four-bit code is needed, leaving six combinations unused. One widely used decimal code is Binary-Coded Decimal (BCD). BCD differs from converting a decimal number directly to binary. For instance, while 99 converts to binary as 1100011, in BCD, it's represented as 1001 1001.

Table 3.3: Binary Coded Decimal Number

Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑ Code for one decimal digit ↓
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

The ASCII code, a standard alphanumeric binary code, employs seven bits to represent 128 characters. Binary codes are essential as registers can exclusively contain binary data.

Table 3.4: American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	,	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	\$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	—	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

3.5. Conversions

Decimal Number System to Other Base

Converting a number from the Decimal Number System to any other base involves two simple steps:

1. Divide the Decimal Number by the target base of the system you're converting to—Binary (2), Octal (8), or Hexadecimal (16).
2. Arrange the remainders obtained from the division, starting from the Least Significant Bit (LSB) to the Most Significant Bit (MSB).

Table 3.5: Decimal Number System to Other Base conversion

Base 10 to Base 2 Conversion	Result																																												
BASE 10 is: (12345)₁₀ <table><tr><td>2</td><td>12345</td><td>1</td><td rowspan="12">LSB</td></tr><tr><td>2</td><td>6172</td><td>0</td></tr><tr><td>2</td><td>3086</td><td>0</td></tr><tr><td>2</td><td>1543</td><td>1</td></tr><tr><td>2</td><td>771</td><td>1</td></tr><tr><td>2</td><td>385</td><td>1</td></tr><tr><td>2</td><td>192</td><td>0</td></tr><tr><td>2</td><td>96</td><td>0</td></tr><tr><td>2</td><td>48</td><td>0</td></tr><tr><td>2</td><td>24</td><td>0</td></tr><tr><td>2</td><td>12</td><td>0</td></tr><tr><td>2</td><td>6</td><td>0</td></tr><tr><td>2</td><td>3</td><td>1</td></tr><tr><td></td><td>1</td><td>1</td><td>MSB</td></tr></table>	2	12345	1	LSB	2	6172	0	2	3086	0	2	1543	1	2	771	1	2	385	1	2	192	0	2	96	0	2	48	0	2	24	0	2	12	0	2	6	0	2	3	1		1	1	MSB	BASE is (11000000111001) ₂
2	12345	1	LSB																																										
2	6172	0																																											
2	3086	0																																											
2	1543	1																																											
2	771	1																																											
2	385	1																																											
2	192	0																																											
2	96	0																																											
2	48	0																																											
2	24	0																																											
2	12	0																																											
2	6	0																																											
2	3	1																																											
	1	1	MSB																																										

Base 10 to Base 8

Table 3.6: Decimal to Octal Conversion

BASE 10 TO BASE 8	Result																	
BASE 10: (12345)₁₀ <table><tr><td>8</td><td>12345</td><td>1</td><td rowspan="4">LSB</td></tr><tr><td>8</td><td>1543</td><td>7</td></tr><tr><td>8</td><td>192</td><td>0</td></tr><tr><td>8</td><td>24</td><td>0</td></tr><tr><td></td><td>3</td><td>3</td><td>MSB</td></tr></table>	8	12345	1	LSB	8	1543	7	8	192	0	8	24	0		3	3	MSB	BASE 8 is(30071) ₈
8	12345	1	LSB															
8	1543	7																
8	192	0																
8	24	0																
	3	3	MSB															

Base 10 to Base 16

Table 3.7: Decimal to Hexadecimal

Example 1 BASE 10: (12345)₁₀ <table><tr><td>16</td><td>12345</td><td>9</td><td rowspan="3">LSB</td></tr><tr><td>16</td><td>771</td><td>3</td></tr><tr><td>16</td><td>48</td><td>0</td></tr><tr><td>8</td><td>3</td><td>3</td><td>MSB</td></tr></table>	16	12345	9	LSB	16	771	3	16	48	0	8	3	3	MSB	BASE 16 :(3039)₁₆
16	12345	9	LSB												
16	771	3													
16	48	0													
8	3	3	MSB												
Example 2 BASE 10: (725)₁₀ <table><tr><td>16</td><td>725</td><td>5</td><td>5</td><td rowspan="2">LSB</td></tr><tr><td>16</td><td>45</td><td>13</td><td>D</td></tr><tr><td></td><td>2</td><td>2</td><td>2</td><td>MSB</td></tr></table>	16	725	5	5	LSB	16	45	13	D		2	2	2	MSB	BASE 16 (2D5)₁₆ Convert 10, 11, 12, 13, 14, 15 to its equivalent... A, B, C, D, E, F,
16	725	5	5	LSB											
16	45	13	D												
	2	2	2	MSB											

3.6. Other Base System to Decimal Conversion

Binary to Decimal

In this conversion, binary number to a decimal number, we use multiplication method, in such a way that, if a number with base n has to be converted into a number with base 10, then each digit of the given number is multiplied from M SB to LSB with reducing the power of the base. Let us understand this conversion with the help of an example.

Example 1. Convert (1101)₂ into a decimal number.

Solution: Given a binary number (1101)₂.

Now, multiplying each digit from MSB to LSB with reducing the power of the base number 2.

$$\begin{aligned} &1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 8 + 4 + 0 + 1 \\ &= 13 \end{aligned}$$

Therefore, (1101)₂ = (13)₁₀

Octal to Decimal

To convert from octal to decimal, each digit of the octal number is multiplied by descending powers of the base number 8. This multiplication starts from the Most Significant Bit (MSB) and proceeds to the Least Significant Bit (LSB). The resulting products are then summed together to obtain the decimal equivalent.

Example: Convert 228 to its decimal equivalent.

Solution: The octal number given is 228.

Performing the multiplication using decreasing powers of 8:

$$2 \times 8^1 + 2 \times 8^0$$

$$= 16 + 2$$

$$= 18$$

Hence, 228 in octal is equal to 18 in decimal.

Hexadecimal to Decimal

Example 3: Converting 12116 to its decimal equivalent.

Solution: The given number is 12116.

Using the octal to decimal conversion:

$$1 \times 162 + 2 \times 161 + 1 \times 160$$

$$= 16 \times 16 + 2 \times 16 + 1 \times 1$$

$$= 289$$

Therefore, 12116 equals 28910 in decimal.

Regarding the explanation of binary system complements:

1's Complement

In the binary system, characterized by its base value $r = 2$, the 1's complement is derived by flipping all the 1s to 0s and vice versa. This process is known as the one's complement or 1's complement. For instance, the conversion of 1's complement appears as follows.

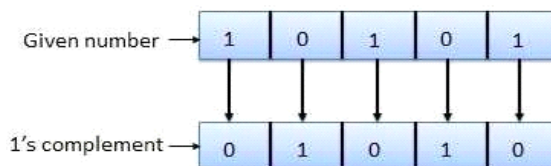


Figure 3.2: 1's Complement Representation

2's Complement

The 2's complement of a binary number is achieved by adding 1 to the least significant bit (LSB) of the 1's complement.

2's complement = 1 + 1's complement.

The sum of Example 2 is as follows.

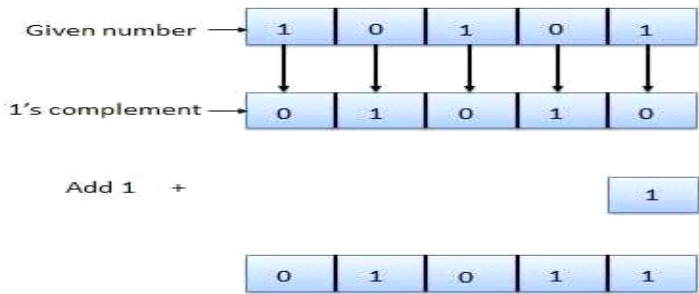


Figure 3.3: 2's Complement Representation

3.7. Binary Addition

Binary Addition Rules

Addition in binary follows specific rules:

- 0 plus 0 equals 0
- 0 plus 1 equal 1
- 1 plus 0 equals 1
- 1 plus 1 result in a carry of 1 and a bit of 0

Table 3.8: Binary Addition Rules

1	1								Carries
0	0	0	1	1	0	1	0	=	26(base 10)
+ 0	0	0	0	1	1	0	0	=	12(base 10)
0	0	1	0	0	1	1	0	=	38(base 10)
00011010 + 00001100 = 00100110									
1	1	1	1	1					carries
0	0	0	1	0	0	1	1	=	19(base 10)
+ 0	0	1	1	1	1	1	0	=	62(base 10)
0	1	0	1	0	0	0	1	=	81(base 10)
00010011 + 00111110 = 0101000									

3.8. Binary Multiplication

Binary Multiplication Rules

Binary multiplication rules are relatively straightforward:

- 0 multiplied by 0 equals 0.
- 0 multiplied by 1 equal 0.
- 1 multiplied by 0 equals 0.
- 1 multiplied by 1 equal 1

00101001 × 00000110 = 11110110	0 0	1 0 1 0 0 1	=	41(base 10)
	× 0 0	0 0 0 1 1 0	=	6(base 10)
	0 0	0 0 0 0 0 0		

0 0 1 0 1 0 0 1
0 0 1 0 1 0 0 1
0 0 1 1 1 1 0 1 1 0
=246(base10)

3.9. Binary Division

Binary division is a repetition of the subtraction process, just like arithmetic division.

For example,

00101010 ÷ 00000110 = 1 1 1 = 7(base 10)

What is 213/5?

Dividend (213) = 11010101

Divisor (5) = 101

101010 (Quotient =42)

101) 11010101
 101

 11
 110
 101

 11
 110
 101

 11 (Reminder =3)

$$101010 / 000110 = 000111$$

$$\begin{array}{r}
 \begin{array}{c} 111 \\ \hline 000110 \overline{) 101010} \\ \underline{-110} \\ 1001 \\ \underline{-110} \\ 110 \\ \underline{-110} \\ 0 \end{array}
 \end{array}
 \begin{array}{l}
 = 7_{10} \\
 = 42_{10} \\
 = 6_{10}
 \end{array}$$

3.10. Fixed-point Representation

- Unsigned numbers represent positive and zero values.
- Negative numbers are represented using signed numbers since there are no explicit symbols like + or - in binary, only 0s and 1s.
- Two primary methods indicate the placement of binary numbers:
- Two Ways to Specify Binary Point Position in Scratchpad:
 - Fixed Point Position
 - Floating Point Representation

Fixed positions typically occupy one of two locations:

- The leftmost binary decimal place in the scratchpad indicates a whole number.
- The rightmost binary decimal place in the scratchpad indicates a fractional number.
- In both cases, the binary point's location is vital but not explicitly shown.
- Floating point representation uses one segment to indicate the binary point's position in another segment.
- A positive number's Most Significant Bit (MSB) is 0, and the remaining bits represent its magnitude.

For negative numbers, the most significant or signed bit is 1, and the remaining bits have various representations:

- Signed Number Representation
- Complementary Representation of Sign 1
- Complementary Representation of Sign 2

Consider an 8-bit scratchpad and the number +14:

- The representation for this number is 00001110.
- For a -14 in an 8-bit scratchpad:

- Signed Value Number: 10001110
- Signed Addition 1: 11110001
- Signed Addition 2: 11110010
- Two's complement is usually used for negative numbers.

When adding two signed numbers, the rules depend on the signs:

- If signs are the same, add the two numbers and use the sign of the larger value. If signs differ, subtract the smaller value from the larger value and use the larger value's sign.
- Addition of two numbers with two signs requires only addition and considering the final sign bit.
- All negative numbers must be in 2's complement form, meaning there are 2 more papers for negative numbers.

+6	00000110	-6	11111010
$\oplus$ +13	00001101	+13	00001101
+19	00010011	+7	00000111
+6	00000110	-6	11111010
-13	11110011	-13	11110011 $\square$
-7	11111001	-19	11101101
+6	00000110	-6	11111010
$\oplus$ +13	00001101	+13	00001101
+19	00010011	+7	00000111
+6	00000110	-6	11111010
-13	11110011	-13	11110011 $\square$
-7	11111001	-19	11101101

- The sum of two signs 2 is subtracted by
- Use the addition form of 2 for minus (e.g. little sign)
- Add to minus (e.g. sign bit)
- One position of the signed bit position: is thrown away
- Overflow occurs when two n-bit numbers are added and take up a total of n + 1 bits
- So if this happens, the flag is set and controlled by the user can be
- Overflow detection depends on whether the number is signed or unsigned
- For unsigned numbers, carry from the end of the msb • To add signed, there is no overflow if one of the numbers is positive and the other is negative - both must have the same sign.
- If it is moved, it voluntarily detects that the lifter of the paper is slightly out of place. and sign bit position is not equal.

0 375	(0000 0011 0111 1010) _{BCD}
+9 760	(1001 0111 0110 0000) _{BCD}
0 135	(0000 0001 0011 0101) _{BCD}

- The representation of a digit in a scratchpad is a function of the binary code used to represent the digit
- A 4-digit digit requires four flip-flops per digit
- This is faster than arithmetic. binary equivalent The representation and looping required to perform arithmetic operations are more complex.
- The representation of signed numbers in BCD is similar to the representation of signed numbers in binary.
- Capitalization of the sign or addition of the sign together
- Sign is expressed by four parts of the number
0000 to + 0 1001 to -
- Add 9 digits to get 10 more digits of the BCD number and then add 1 to the least significant number

Example: $(+375) + (-240) = +135$

3.11. Floating-Point Representation

Floating-point representation of a number comprises two main components:

- The first part, known as the mantissa, represents the number with a fixed sign.
- The second part, known as the exponent, indicates the function of the binary number's decimal point.
- The mantissa can be either a fraction or an integer. For example, in the decimal number +6132.789, the fraction is +0.6123789, and the exponent is +04, resulting in +0.6123789 multiplied by 10^4 .
- A floating-point number is always represented as m multiplied by 2 raised to the power of exponent. For instance, $+.1001110_2 \times 2^4$.
- In a floating-point number, the most significant digit is the non-zero mantissa. Non-standard representations like adding leading zeros (e.g., 00350) or having trailing zeros (e.g., 11010000) are not considered standard, the fraction and exponent can be separated, such as in the case of the fractional representation 11010000 with an exponent of -3.

3.12. IEEE Standard 754 Floating Point Standard

The IEEE 754 standard, established by the Institute of Electrical and Electronics Engineers (IEEE) in 1985, governs the representation of floating-point numbers, a common format for real numbers in modern computing across various platforms like Intel-based PCs, Macs, and Unix systems. Among several methods for representing such numbers, IEEE 754 is the most

widespread and considered the most reliable in numerous cases. This standard is characterized by three main elements:

Mantissa Sign

The sign of the mantissa is straightforward: 0 indicates a positive number, while 1 denotes a negative number.

Biased Exponent

The exponent field covers both positive and negative indices. A bias is added to the actual value to determine the measured value.

Normalized Mantissa

The mantissa, representing the significant digits in scientific notation, consists solely of 0s and 1s. A normalized mantissa holds a single 1 to the left of the decimal point.

The IEEE 754 code encompasses two divisions based on these aspects: single precision and double precision.

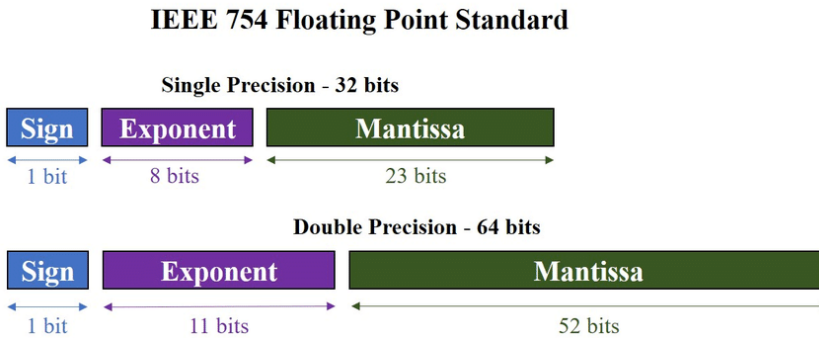


Figure 3.4: IEEE 754 Floating Point Representation

$$0.1_{10} = 0.000110011001100110011001100_2$$

$$0.000110011001100110011001100 = 1.10011001100110011001100 \times 2^{-4}$$

Exponent = $-4 + 127 = 123$
Sign bit = 0
Mantissa = 10011001100110011001100

S (1 bit)	Exponent (8 bits)	Mantissa (23 bits)
0	01111011	10011001100110011001100

Alternate Binary Codes

- Digital systems exclusively handle discrete data.

- Continuous or analog information undergoes conversion to digital form through an analog-to-digital converter.
- The Gray code, also known as reflected binary code, is occasionally employed for the converted digital data.
- Unlike traditional binary codes, the Gray code alters only one bit as it progresses from one number to the subsequent one.
- Gray code counters find applications in generating timing sequences essential for overseeing operations within a digital system.

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

Table 3.9: 4 -Bit Gray Code

At least four bits are necessary for binary codes representing decimal digits.

Beyond Binary Coded Decimal (BCD), various alternative codes are available to represent decimal digits.

TABLE 3-6 Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
Unused bit combinations	1010 1011 1100 1101 1110 1111	0101 0110 0111 1000 1001 1010	0000 0001 0010 1101 1110 1111	0000 0001 0011 1000 1001 1011

Table 3.10: Four Different Binary Codes for the Decimal Digit

2421 and excess-3 Codes

- Both the 2421 code and the excess-3 code possess self-complementing proper ties.
- Obtaining the 9's complement of each digit involves complementing each bit in the code.

- The 2421 code is a form of weighted code, where each bit holds a specific weight contributing to the final decimal digit.
- Excess-3 code derivation involves adding 3 to the corresponding BCD code.

Error Detection Codes

- Binary data transmitted is prone to noise that might invert bits (1 to 0 and vice versa).
- Error detection codes serve to identify digital errors occurring during data transmission.
- While these detected errors cannot be rectified, they signal the need for retransmitting the data.
- The widely used error detection method involves implementing a parity bit, an additional bit incorporated into a binary message to ensure the total count of 1's is either odd or even.

Table 3.11: Parity Bit Generator (Even)

3-bit message			Even parity bit generator (P)
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table 3.12: Parity Bit Generator(odd)

3-bit message			Odd parity bit generator (P)
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Parity Bit and Error Detection

- The choice of the P(odd) bit aims to ensure that the total count of 1's across all four bits becomes odd.
- The even-parity scheme has a downside, primarily that it leads to an all-zero bit combination.
- During transmission, at the sender's end, the message undergoes parity generation.
- The transmitted message includes the generated parity bit.
- Upon reception, the incoming bits, including the parity bit, are processed by a parity checker.
- Detection of any odd number of errors is feasible.
- Parity generators and checkers are structured using XOR gates, implementing an odd function that generates a '1' only if the input variables are in an odd state.

3.13. Computer Arithmetic

- Arithmetic instructions execute operations like addition, subtraction, multiplication, and division to solve computational problems.
- These operations serve as the foundation for addressing various numerical problems through numerical analysis techniques.
- Negative fixed-point binary numbers can be represented using signed magnitude, signed 1's complement, or signed 2's complement. Among these, signed 2's complement is the most prevalent, although signed magnitude representation is encountered on occasion.

Addition and Subtraction of Signed Magnitude Data

When performing addition or subtraction between two signed numbers (A and B), various conditions must be considered, leading to eight different scenarios described in the following table.

Algorithm for Addition and Subtraction

- When A and B have the same signs (different signs), the algorithm adds (subtracts) their magnitudes and assigns the sign of A to the resultant value.
- In the case of different signs between A and B (identical signs), the algorithm compares the magnitudes, subtracting the smaller from the larger.

Hardware Implementation

To execute these arithmetic operations via hardware, the numbers are stored i n two registers—A and B. Two flip-flops, As and Bs, store the corresponding si gns. The resulting values are then transferred to A and As.

Table 3.13: Addition and Subtraction Methods

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

Operation	Add Magnitude	Subtract Magnitude		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$	$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) + (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (+B)$				
$(-A) + (-B)$	$-(A + B)$	$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (+B)$				
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

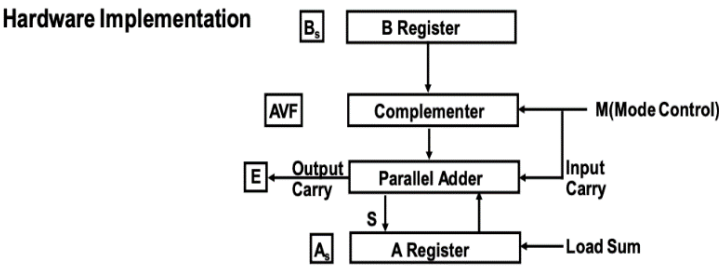


Figure 3.5: Signed Magnitude Addition and Subtraction and Hardware Implementation

- The hardware setup includes two registers labeled A and B, along with sign flip-flops denoted as As and Bs.
- A magnitude comparator assesses whether A is greater than B.
- A parallel adder computes the sum of A and B.
- Dual parallel subtraction modules enable computations for A-B and B-A.
- The sign relationships are determined using an XOR gate, taking As and Bs as inputs.
- This hardware configuration comprises registers A and B, in addition to the sign flip-flops As and Bs.
- A complemer generates an output of B or B' based on the input mode M.
 - When $M=0$, the output of B is channeled into the adder, initializing the input carry as 0, thereby producing the output of $A+B$. The carry output passes to flip-flop E, allowing controlled transitions. E permits passage if $E=1$; otherwise, it doesn't.

- If $M=1$, adding 1 to B results in the input becoming 1, generating an output $S = A + B' + 1$ (A-B). The carry output proceeds to flip-flop E, verifying equality between the two numbers.

Hardware Algorithm

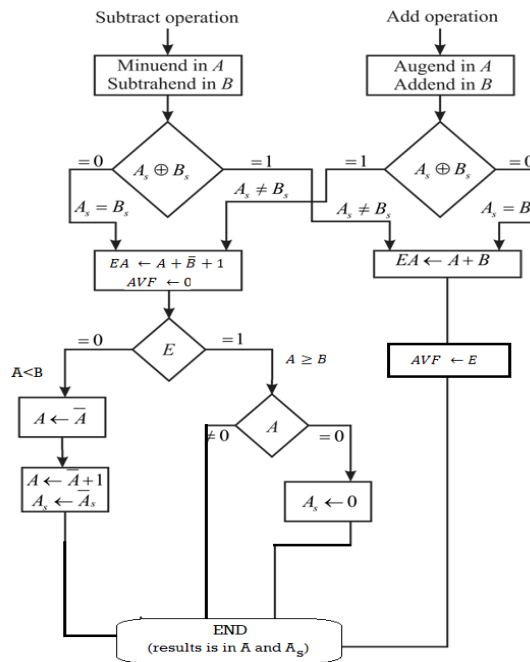


Figure 3.6: Flow Chart

The Hardware Algorithm Flowchart Follows this Sequence

- A_s and B_s 's comparison with an XOR gate: Output of 0 indicates the same signs, while 1 suggests different signs.
- Similar signs trigger addition, while different signs signal magnitude addition for subtraction.
- Microfunction E specifies $A \leftarrow A + B$ (EA is the result of concatenating E and A).
- If $E=1$, overflow signals and directs to AVF.
- For different signs in addition but the same in subtraction, it involves subtraction of two values.
- Micro-operation E triggers $A \leftarrow A + B' + 1$ for subtraction from a larger value.
- AVF is reset to 0 if no flow is detected.
- When $E = 1$, $A \geq B$ implies a true result in A. If A is 0, A_s must remain positive ($A_s = 0$) to prevent negative zeros.

- If $E = 0$, indicating $A < B$, A is handled as 2's complement. It operates as $A \leftarrow A' + 1$, ensuring the sign of A_s is the same as B_s (i.e., $A_s \leftarrow A'_s$).

For Signed 2's Complement Representation

- The sign bit represents the remainder (0 for positive, 1 for negative). For integers represented in 2's complement, the sign bit is 1.
- In addition, the sign bit is treated as any other code part, maintaining and discarding portions.
- Subtraction entails adding 2 from the negative and then performing the subtraction.
- Overflow detection occurs in XOR gates during the addition process, signaling overflow when the gate output is 1.

Hardware Implementation

- Registers remain similar to signed number representation, but the signed bit is not separate. The AC and BR's remaining bits reflect the sign. The significant distinction lies in how the sign bit integrates with the complement and carry elements. If an overflow occurs, the overflow flop V is set to 1, leading to the dismissal of the operation.

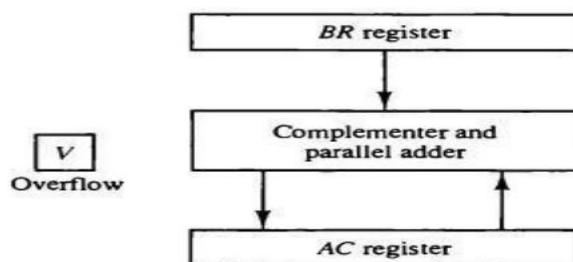


Figure 3.7: Hardware for Signed -2's Complement Addition and Subtraction

Hardware Algorithm

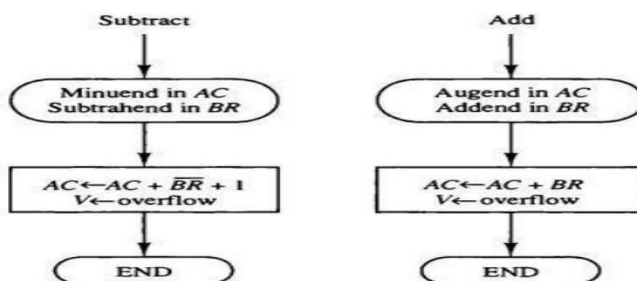


Figure 3.8: Algorithm for Adding and Subtracting Numbers in Signed 2's Complement Representation

Multiplication Algorithm

Multiplication of Signed-Magnitude Data

The multiplication algorithm for Signed-Magnitude Data:

- At the start, A holds the multiplier, B stores the multiplicand, and Q maintains the accumulator. Corresponding sign bits are in A and Q. Comparisons between signs A and Q determine the sign of the result as the operands are stored in these registers. A and E records are cleared, and the SC (Sign Control) register is set to hold the count of elements. The operand is stored with its sign, with one bit reserved for the sign and the remaining n-1 bits for its size.
- The algorithm begins with the examination of the least significant bit of Qn. If it's 1, equation (B) is added to the ongoing multiplication (A); otherwise, it's 0. The EAQ record is then modified accordingly to form a new partial product. The equation is reduced by 1, and its new value is assessed. If it isn't zero, the process is reiterated, creating another partial product. This cycle continues until SC equals zero, signaling the end of the process.

Hardware Usage

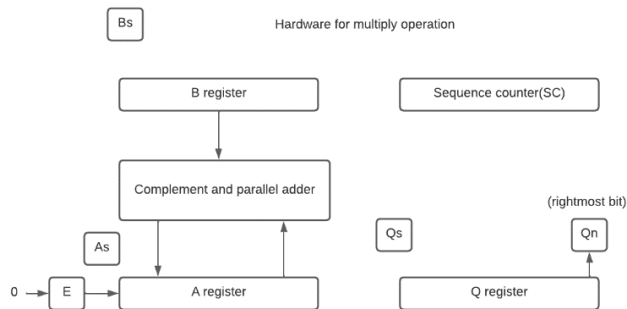


Figure 3.9: Hardware for Multiply Operation

- B ← multiplication, Bs ← sign
- Q ← multiplier, Qs ← sign
- Continue writing partial products and working rights methods
- SC ← None. Number of items in balance (larger only)
- SC is reduced after each item is created. When SC is 0, the process stops and the final product is created.
- The numbers A and B form part of the product.

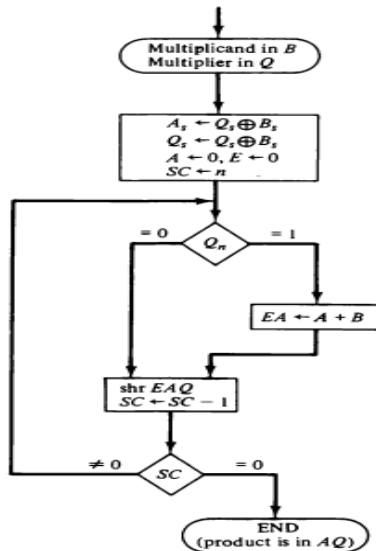


Figure 3.10: Flow Chart of Multiplication

Table 3.14: Multiplication Table

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
Qn = 1; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
Qn = 1; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
Qn = 0; shift right EAQ	0	01000	10110	010
Qn = 0; shift right EAQ	0	00100	01011	001
Qn = 1; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000

Final product in AQ
0110110101

Multiplication of Signed 2's Complement Data (Booth Multiplication Algorithm)

Addition Table algorithm is used to combine multiple binary numbers into signed 2's complement form.

Hardware for Booth Algorithm

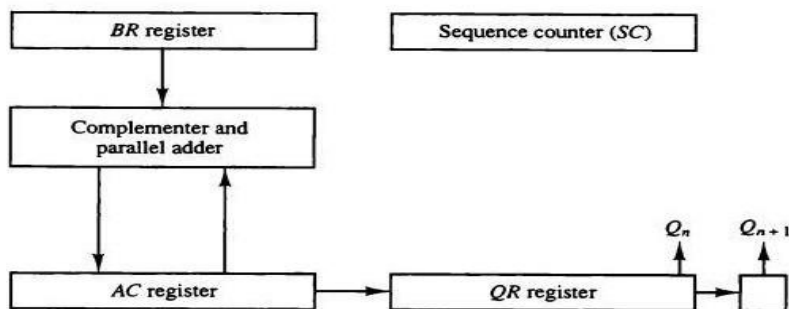


Figure 3.11: Booth Multiplication Algorithm

- The bit symbol is not reserved here.
- The scratchpads A, B, and Q are now referred to as AC, BR, and QR, respectively.
- Special additional flip-flop $Q_n + 1$ connected to QR is required to recover the almost lost equal (compared to that provided by the Now Q_n combination) about order a little more information).
- Matching $Q_n Q_{n+1}$ Check the number of elements as twice the integer.

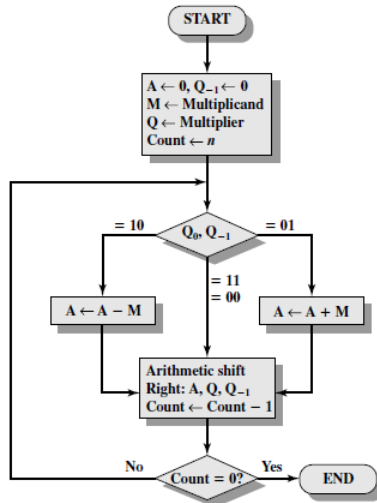


Figure 3.12: Flow Chart for Booth Multiplication Algorithm

Example: Let us multiply (-6) and (2) using Booth's algorithm.

Solution: $(6)_{10} = (0110)_2$

As it is given multiplicand, $M = (-6)_{10} = 2$ complement of $0110 = 1010$

Multiplier, $Q = (2)_{10} = 0010$

Table 3.15: Booth Algorithm

AC	Q	Q ₋₁		
0000	0010	0	Initial	
0000	0001	0	shift	Q ₀ , Q ₋₁ = 00
0110	0001	0	AC=AC-M	Q ₀ , Q ₋₁ = 10
0011	0000	1	shift	
1101	0000	1	AC=AC+M	Q ₀ , Q ₋₁ = 01
1110	1000	1	Shift	
1111	0100	0	Shift	Q ₀ , Q ₋₁ = 00

Product by Booth's algorithm= 1111 0100

- Both's algorithm facilitates division of binary integers in signed 2's complement representation.
- Like all division methods, Booth's Algorithm handles partial products, checking for duplicates, and cycling through them. Before conversion, the product equivalent can be added to the partial product, subtracted from it, or remain unchanged based on specific rules:
- Subtract half the number of 1s from the product while ensuring the first bit is 1; subtract this from the product.
- When a series of 0s in the multiplier reaches the first encountered 0 after a sequence of 1s, add the multiplier to half of the product.
- If the balance is similar to the previous one, the partial stock remains unchanged.
- Q_n represents the least significant bit in QR. An additional Q_{n+1} flip-flop in QR helps cross-verify even numbers.
- Initially, AC and the additional bit Q_{n+1} are set to 0, and the sequence counter (SC) matches the number of bits, 'n', in the digital multiplier.
- Check two products using Q_n and Q_{n+1}.
- both equal '10', it indicates the first 1 in the string, requiring subtraction of the sum of partial components from AC.
- If both equal '01', it denotes the first 0 in a series of 0s, demanding addition of the product and partial product of AC.
- Under equal circumstances, the partial product remains unchanged. Continuous addition and subtraction avoid overflow issues.
- The next step involves right shifting the partial product and the even number (including Q_{n+1}) using the arithmetic right shift (ashr) function, which shifts AC and QR to the right while maintaining the bit in AC.
- Reduce the sequence counter and repeat the loop 'n' times.

Division Algorithm

- Binary division involves constant comparison, manipulation, and subtraction of two binary elements represented by their signed values. In contrast to general division, binary division is straightforward with the quotient being either 0 or 1, eliminating the need to calculate multiple divisors or the remainder.

$$\begin{array}{r}
 \text{Divisor } 1101 \overline{) 100010010} \\
 \underline{-1101} \\
 10000 \\
 \underline{-1101} \\
 1110 \\
 \underline{-1101} \\
 1 \text{ Remainder}
 \end{array}$$

Compare the divisor against the most significant digits of the dividend, starting with five digits. If the number 5 is smaller than the divisor, continue the process. Upon reaching the sixth digit, which is greater than the divisor, denote 1 in the quotient above the divisor. Now, perform subtraction with the divisor, resulting in the remainder. At this point, division can cease, having obtained a quotient of 1 and a corresponding remainder.

Proceed by comparing this remainder with the divisor. If the remainder is greater than or equal to the divisor, assign a quotient of 1. Then, shift the divisor to the right and subtract it from the remainder. Conversely, if the remainder is less than the divisor, assign a quotient of 0 without performing subtraction. However, shift the divisor to the right once. This process provides both the quotient and the remainder.

Hardware Implementation of Signed Numeric Data

The hardware setup for division is similar to that required for multiplication. Key aspects include:

- Register B stores the divisor, while registers A and Q hold the two-word dividend.
- Relative size information is stored in E.
- An array named SC is utilized to store the numbers involved in the partition.

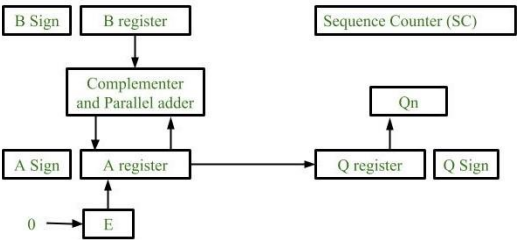


Figure 3.13: Hardware Implementation of Multiplication Algorithm

Algorithm

Initially, allocate the divisor to registers A and Q, and place the dividend in register B. Capture the sign of the outcome in Q as part of the quotient. Fix a constant value in SC to determine the digit count in the quotient. As the operand must include its sign, allocate one bit for the sign, making the word size $n-1$ bits.

Conduct a previous division check by deducting half of the divisor in B from the one in A. If A is greater than or equal to B, adjust the Division VF (DVF) and conclude the process early. However, if A is less than B, proceed without any changes, returning the result obtained by adding B to A.

The division process starts by performing a shifting operation from AQ to the left, taking $n-1$ elements (while B has only $n-1$ elements). During this step, subtract B from EA and assign 1 to Q as part of the quotient.

If the left operation (shl) adds 0 to E, subtract the sum of the divisor and propagate the carry to E by adding 2. A resulting $E=1$ signifies $A \geq B$, hence Q is set to 1. Conversely, $E=0$ indicates $A < B$, and the initial number is restored by adding B to A.

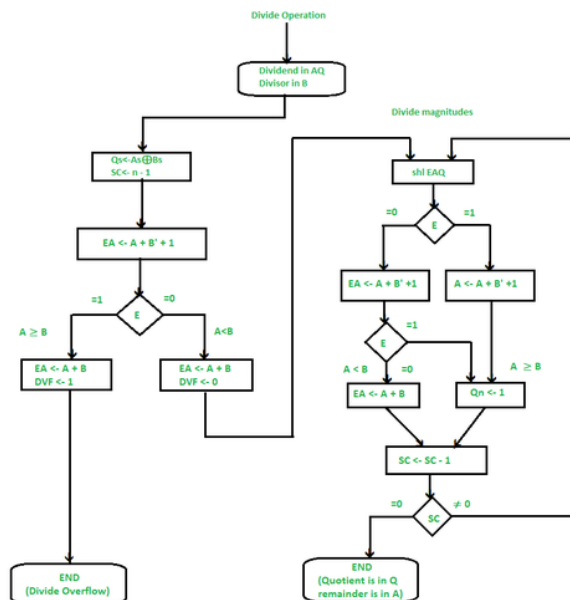


Figure 3.14: Flow Chart

Example

Divisor B = 10001, Dividend A = 0111000000

Table 3.16: Example of a Binary Division Using Digital Hardware

Divisor B = 10001	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add B + 1		01111		
E = 1	1	01011		
Set Q _n = 1	1	01011	00001	4
shl EAQ	0	10110	00010	
Add B + 1		01111		
E = 1	1	00101		
Set Q _n = 1	1	00101	00011	3
shl EAQ	0	01010	00110	
Add B + 1		01111		
E = 0; leave Q _n = 0	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add B + 1		01111		
E = 1	1	00011		
Set Q _n = 1	1	00011	01101	1
shl EAQ	0	00110	11010	
Add 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	

Last part: 00110

Last part: 11010

If the divisor is greater than or equal to the dividend, an overflow happens in this process. EA holds the value of A + B when Q is not utilized. For instance, when the dividend equals the divisor, Q becomes 1, and the remainder becomes 0. However, in all other scenarios, the quotient is 1, and the remainder equals the dividend.

3.14. Floating Point Arithmetic Operations

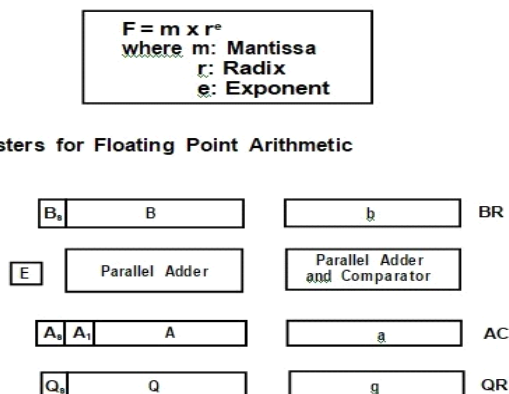


Figure 3.15: Floating Point Arithmetic Operations Hardware Implementation

Addition and Subtraction of Floating-Point Numbers

Performing addition and subtraction of floating-point numbers involves processing two operands stored in AC and BR. The sum or difference is generated in AC, following a four-step algorithm:

Zero Check

- Verify if the operands are zero.
- If either operand is zero, terminate the process as the result will be zero.

Mantissa Setting

- Align the mantissas before the operation.
- For proper normalization, set the mantissas accordingly.

Addition or Subtraction of Mantissas

- Perform the addition or subtraction of mantissas.
- The outcome will differ after these operations.

Static Values

- Normalize the values before storing them in memory.
- If subtracting a large number results in a zero or smaller number, clear all floats in AC to zero.
- If the mantissa is zero, indicating underflow, shift the mantissa left, and reduce the exponent until the most significant bit (A1) is 1
- Repeat this process until A1 equals 1, ensuring the normalization of the mantissa is completed.

This algorithm ensures proper handling of floating-point numbers during addition and subtraction, considering zero checks, mantissa alignment, and normalization procedures.

Multiplication

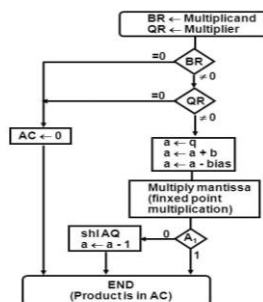


Figure 3.16: Floating Point Division

Division

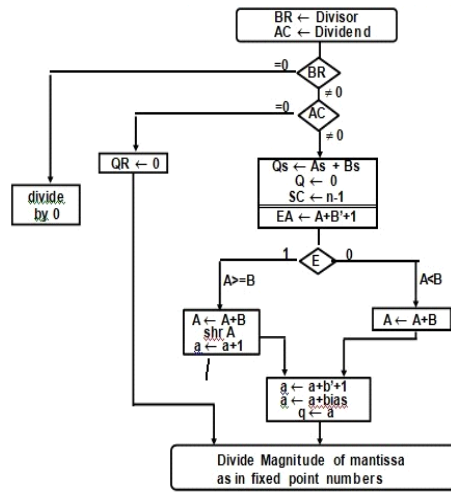


Figure 3.17: Floating Point Division

Decimal Arithmetic Unit Overview

A Decimal Arithmetic Unit (DAU) is a specialized component, whether in hardware or software, tailored for performing decimal arithmetic operations. Differing from Binary Arithmetic Units (BAU), which handle binary numbers (0 and 1), the DAU is specifically designed to manage decimal numbers ranging from 0 to 9.

Arithmetic operations hold significance across various fields such as finance, accounting, and computing, where precision and accuracy in numbers often demand a higher level of detail than binary arithmetic can provide.

Key Features and Functions

- **Decimal Notation:** DAU commonly utilizes binary-coded decimal (BCD) or another decimal format for representing decimal numbers. BCD assigns a 4-digit binary code to each digit (0-9).
- **Decimal Arithmetic:** Similar to binary arithmetic for binary numbers, DAU supports a range of arithmetic operations including addition, subtraction, multiplication, and division.
- **Rounding and Precision:** Decimal arithmetic units incorporate rounding techniques to handle numbers with precision, especially crucial in fields like accounting where accuracy is paramount.

- **Decimal Point Handling:** DAU effectively manages decimal points, facilitating correction and accommodating numbers with varying decimal positions.
- **Error Management:** Mathematical systems integrated into DAU may include error detection and correction procedures, ensuring the reliability and accuracy of calculations.
- **Performance:** Depending on the application, DAU can be optimized for high-speed arithmetic operations, making it well-suited for scenarios requiring swift calculations.
- **Compatibility:** Certain computer architectures and programming languages provide support for decimals, streamlining the manipulation of decimal numbers for software developers.

Applications

Decimal arithmetic finds extensive utility in financial institutions, retail, scientific research, and other domains where precise arithmetic is critical. It aids in averting balance errors and furnishes a more precise representation of numbers in mathematical contexts.

Short Answer

1. Add +5 and -9 using the 2-addition method.
2. Write 2 additions of 1011011.
3. Represent (70)10 signatures on a large poster and additional representation.
4. Do 2 additions, subtract the smaller number (101011) from the larger number (111001).
5. When can a number tell a pattern?
6. Define the terms computer organization and computer design
7. Define r's complement. 8. Write rules for balancing floating-point numbers.
8. Define Gray code.
9. Draw simple tasks on the computer.
10. What are the key features of Booth's algorithm?

Long Answer Questions

1. Define constant representation.
2. Draw the connection between the processor and memory and show the function of each connection. What is the best way to represent a negative number? example.
3. Distinguish between fixed-point and floating-point representations of given numbers. 4. Replace the following numbers with base six numbers.
(i) (12121)3 (ii) (4310)5 (iii) (50)7

4. Draw a diagram and explain the Bootstrap algorithm
5. Use the Boots algorithm to multiply 100111 by 11011.
6. Give an example of how to multiply two unsigned numbers. 7. Give and explain algorithms for addition and subtraction of two floating point binary numbers.
7. Explain the algorithm for dividing numbers using suitable examples.
8. Give an example of the addition of r and $(r-1)$. 10. Define the following terms
9. Computer organization II. Computer design III. Computer Architecture

Multiple Choice Questions

1. How many options are there in a binary option? ()
 - a) One
 - b) Two
 - c) Three
 - d) Depending on the amount of computer memory
2. A sequence of eight 0s and 1s is called ()
 - a) Megabyte
 - b) Kilobyte
 - c) Gigabyte
 - d) None of these
3. The smallest data that a computer can understand and process is called
 - a) Number
 - b) Kilobyte
 - c) Bit
 - d) Byte
4. _____ indicates how much data a single storage location can hold.
 - a) Access
 - b) Capacity
 - c) Memory
 - d) Storage
5. Computer works on _____ digital system ()
 - a) Binary
 - b) Octal
 - c) Decimal
 - d) Hexadecimal

6. Which of the following is the largest storage facility? ()
- a) GB
 - b) KB
 - c) MB
 - d) TB
7. Data stored in the computer ()
- a) Analog data
 - b) Digital data
 - c) Routine data
 - d) Watt data
8. A kilobyte is ()
- a) Kilobyte
 - b) Megabyte
 - c) Gigabyte
 - d) Terabyte
9. The smallest unit in the number system is _____()
- a) Byte
 - b) Kilobyte
 - c) Word
 - d) Bit
10. To calculate perform calculation and store data computer user__number system
- a) Decimal
 - b) Hexadecimal
 - c) Octal
 - d) Binary
11. In the binary language, each letter of the alphabet, each number and each character is made up of a unique combination of _____()
- a) Eight byte
 - b) Eight kilobytes
 - c) Eight characters
 - d) Eight bits
12. Numbers that are written with base 10 are classified as ()
- a) Decimal number
 - b) Whole number

- c) Hexadecimal number
 - d) Exponential integers
13. To convert binary number to decimal, multiply the all-binary digits by power of ()
- a) 0
 - b) 2
 - c) 4
 - d) 6
14. Converting decimal number (15) to 10 hexadecimal numbers. ()
- a) (14)₁₆
 - b) (13)₁₆
 - c) (F)₁₆
 - d) (7F)₁₆
15. The maximum possible character size in EBCDIC code is ()
- a) 356
 - b) 756
 - c) 556
 - d) 256
16. To calculate stored data, computers use the___ number system ()
- a) decimal
 - b) hexadecimal
 - c) octal
 - d) binary
17. Convert the decimal number (61) 10 to its binary equivalent as ()
- a) (110011)₂
 - b) (11001110)₂
 - c) (111101)₂
 - d) (11111)₂
18. Which of the following is used in double multiplication? ()
- a) Restorative multiplication
 - b) Booth's algorithm
 - c) Pascal's rule
 - d) Bitwise multiplication
19. Booth's algorithm is valid for_____ ()
- a) Decimal digit

- b) Binary digit
 - c) Hexadecimal digit
 - d) Octal digit
20. The result of binary multiplication of $1010 * 1100$ is ()
- a) 000111
 - b) 001111
 - c) 111110
 - d) 111100
21. The probability of getting less than the smallest number is called
- a) NaN
 - b) Underflow
 - c) Minimum
 - d) Mantissa
22. $1101 + 10$ binary addition is done as follows:_____ ()
- a) 1110
 - b) 1111
 - c) 0111
 - d) 1.1101
23. The signed amplitude of -9 is expressed as
- a) 00001001
 - b) 11111001
 - c) 10001001
 - d) 11001
24. If a word of size n is given, the range of the binary 2's complement is
- a) $-2n + 1$ to $+2n + 1$
 - b) $-2n - 1$ to $+2n - 1$
 - c) $2n - 1$ to $+2n + 1$
 - d) $-2n - 1$ to $+2n - 1$
25. In the checked case and addition of 2, positive and negative numbers are separated using _____
- a) LSB
 - b) MSB
 - c) 0
 - d) 1

26. The single format contains_____elements. ()
- a) 4
 - b) 8
 - c) 16
 - d) 32
27. If m is 2's complement and n is a binary number, then
- a) $m = n'$
 - b) $m = n' + 1$
 - c) $m = n' - 1$
 - d) $m =$
28. Number of possible patterns for 8 bits ()
- a) 128
 - b) 8
 - c) 24
 - d) 256
29. What is the default value of the accumulator when adding two 4-bit binary numbers?
- a) 0
 - b) 1
 - c) 0000
 - d) 00000
30. What is the value obtained when multiplying $(-2) * (-3)$ using Booth's algorithm?
- a) 6
 - b) -6
 - c) -2
 - d) -3
31. The sum of 10 of 562 is _____ ()
- a) 4
 - b) 3
 - c) 7
 - d) 8
32. The sum of 9 is 6578:_____ ()
- a) 1234
 - b) 3421
 - c) 3124

- d) 3420
33. Subtraction using the sum of 1 of 110-100 will give:_____
- a) -011
b) 011
c) 010
d) -010
34. The 1's complement of 1111111110.101 is _____()
- a) 1.010
b) 0000000000.010
c) 1111111110.101
d) 1.101
35. The sum of 7 of 432 is_____ ()
- a) 432
b) 543
c) 345
d) 777
36. 16 Addition of 74E_____ ()
- a) 8B2
b) F8B2
c) 2B8
d) 8C2

Points

1. (b) 2 (d) 3. (c) 4. (b) 5. (a) 6. (d) 7. (b) 8. (a) 9. (d) 10. (d) 11. (d) 12. (a) 13. (b) 14. (c) 15. (d)
16. (d) 17. (c) 18. (b) 19. (b) 20. (d) 21. (b) 22. (b) 23. (C) 24. (d) 25. (b) 26. (d) 27. (b) 28. (d)
29. (d) 30. (a) 31. (b) 32. (d) 33. (c) 34. (a) 35. (c) 36. (b)

UNIT IV - INPUT OUTPUT ORGANIZATION

Input-Output Organization: Input-Output Interface, Asynchronous Data Transfer, Modes of Transfer, Priority Interrupt Direct memory Access. Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

4.1. Input/Output System Overview

The computer's Input/Output (I/O) system configuration is contingent on its size and the connected peripherals. This subsystem ensures efficient communication between the central system and the external environment, managing all input and output functions within the computer system.

Peripheral Devices

Peripheral devices refer to input or output devices connected to the computer, tasked with reading or transmitting data as directed by the CPU. They are integral parts of the computer system and encompass a wide array of devices. Common peripherals include keyboards, monitors, and printers, and they are categorized into three types:

- **Input Peripherals:** Devices facilitating input from the external world into the computer, such as keyboards and mice.
- **Output Peripherals:** Instruments enabling data output from the computer to the external world, like printers and monitors.
- **Input-Output Peripheral:** Devices allowing both input (from external sources to the computer) and output (from the computer to the external world), such as touchscreens.

Interface

The interface acts as a shared boundary between independent computer systems, facilitating their connection for communication. It comprises two primary types:

- **CPU Interface:** Provides a programming interface, handling interrupt request signals from the CPU.
- **Input-Output Interface:** Facilitates data transfer between internal memory and external I/O devices.

Peripherals connected to the computer necessitate a specialized communication link with the CPU to bridge the gap between the central computer and these devices. Key differences include:

- Hardware in peripherals involves electrical/electronic devices, necessitating signal value conversion, unlike the electronic devices in the CPU/memory.
- Data transfers within peripheral devices generally occur at slower rates compared to the CPU, demanding synchronization mechanisms.
- Data codes and formats in peripherals often differ from those used in CPU and memory languages.
- Each peripheral device operates in distinct modes and requires control to avoid interference with other connected peripherals. Special CPUs and hardware, known as interface units, manage these variations.

Computer systems incorporate specialized CPUs and hardware, namely interface units, to monitor, synchronize, and reconcile the differences between CPU operations and peripheral interactions. These units address diverse modes of operation, data format distinctions, and varying speed of data transfers within the system.

4.2. I/O Bus and Interface Modules

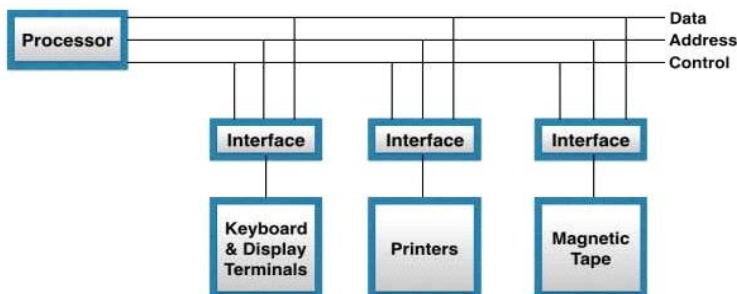


Figure 4.1: I/O Bus and Interface

Processor-Peripheral Communication Interface

The diagram illustrates the interface linking the processor to various peripherals. Within this setup, the I/O bus comprises data lines, address lines, and control lines, crucial for communication between the processor and external devices.

Interface Units for Peripherals

Each peripheral device is associated with an interface unit. These units perform several essential functions:

- Decode addresses and controls received via the I/O bus.
- Manage the location and handling of data on peripheral devices.

- Transmit signals to the controller.
- Synchronize data flow and monitor changes between processors and peripheral devices.

Interface Functions and I/O Device Access

The processor's I/O bus connects to each interface. To engage a specific device, the processor assigns the device's address to the address line and a corresponding function code to the control line. Each interface bears a unique address number aligned with the bus address. Once an interface detects its address, it establishes the connection between the bus line and its controlled device. Peripherals whose addresses don't match the bus addresses are unaffected by their respective interfaces.

IO Instructions and Data Transfer

IO instructions, in the form of operation codes, allow the interface to execute various commands:

- Control Commands: Initialize the peripheral and dictate its actions.
- Events Commands: Assess different interface and environmental events.
- Data Out: Transmit data from the bus to a register within the peripheral.
- Data In: Receive data from the peripheral and place it into the buffer register.

Asynchronous Data Transfer

- Synchronous Transfer: Components share time information from the same clock line.
- Asynchronous Transfer: Occurs without synchronization via a common clock.
- In digital systems, an internal pulse generator sets the synchronization timing. Synchronous transfer happens when the interface register and the CPU share a clock.
- Asynchronous transfer involves units using their clocks, requiring communication control signals to indicate data transmission.

Strobe Control for Asynchronous Data Transmission

- In this method, a single control line manages each state, triggering a strobe pulse to initiate or halt transmission.
- The source or target unit initiates the strobe pulse, signaling the start of data transfer.
- Once data is placed on the bus, the target unit acknowledges it, leading to subsequent actions and data removal from the bus after a specified duration.

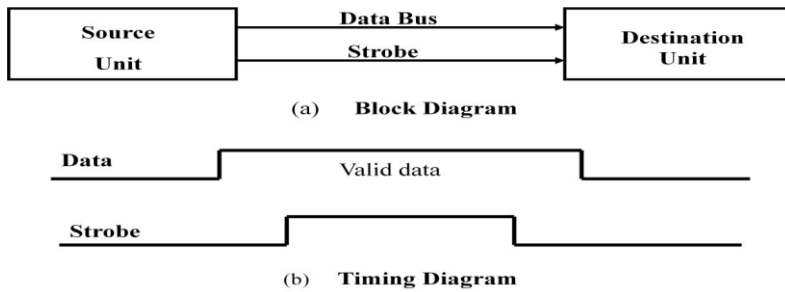


Figure 4.2: Source Initiated Strobe for Data Transfer

4.3. Data Transfer Starts from the Target Unit

The data transfer begins when the target unit triggers a flash pulse, signalling the source unit to transmit the requested binary data onto the data bus. It's crucial for this data to be valid and persist on the bus until the target unit successfully receives it. Upon reception, the source unit deactivates the flash and removes the data from the bus volume.

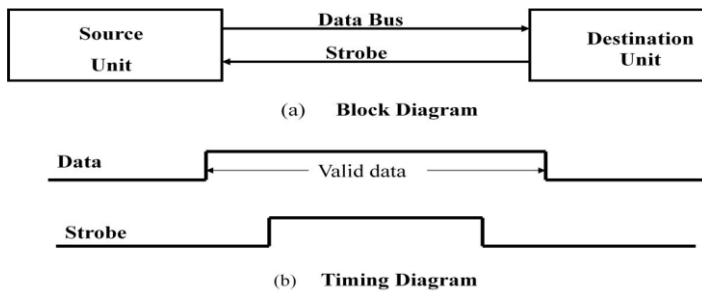


Figure 4.3: Destination Initiated Strobe for Data Transfer

4.4. Handshaking

This method is commonly employed for all transmitted data and control signals indicating the presence of data on the bus. The recipient unit acknowledges the received data through another control signal, creating a mutual confirmation known as a handshake.

In two-wire data transmission, both the control line and data path follow the same direction as data flows from the source to the destination. The master unit utilizes a control line to inform the target unit of the existence of valid data on the data path. Additionally, another control line, going from ground to ground, is employed by the target unit to signal whether the data can be accepted.

Handshaking is utilized to initiate resource transitions in the following sequence of events, specifying four states the system can be in at any given time. The central unit commences the

transfer by placing data on the bus and confirming the presence of valid data. Subsequently, the target unit triggers the data acceptance signal upon receiving data from the data bus. The central unit then deactivates the data confirmation signal, returning the system to its initial state.

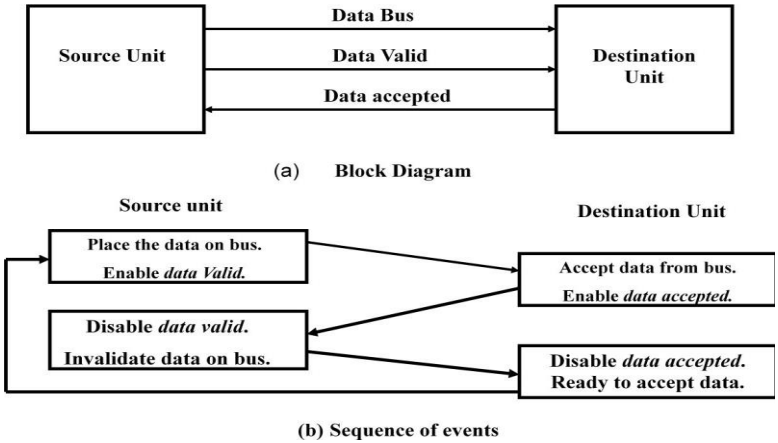


Figure 4.4: Source Initiated Handshaking Signal

Target Changes are Initiated using the Handshaking

The signal produced by the target undergoes a name change to signify the readiness of the data in a modified context. Consequently, the central unit refrains from transmitting data onto the bus until it receives the data-ready signal from the destination. Following this cue, the handshake process proceeds akin to the initial data exchange.

The distinction between a starting variable and a target starting variable lies in their respective starting states of choice.

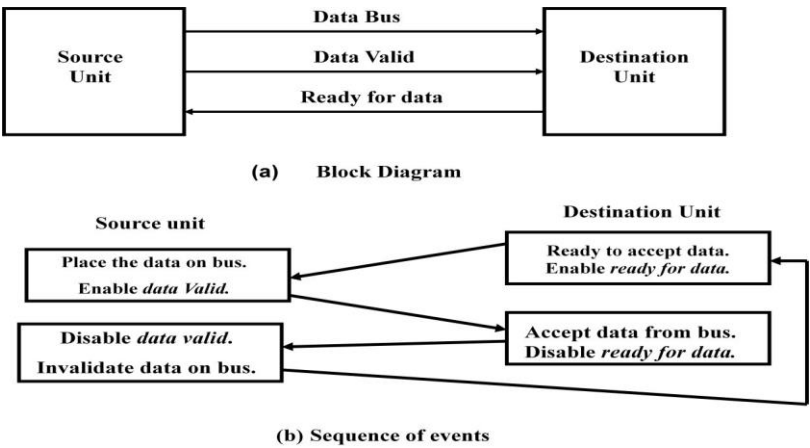


Figure 4.5: Destination Initiated Handshaking Signal

4.5. Asynchronous Serial Transmission

Data transmission between devices can occur through two primary methods: serial or parallel. In parallel transmission, n bits of the message are sent simultaneously through n separate conductor paths. On the other hand, serial transmission involves sending each bit of the message sequentially, one after another.

Parallel transmission is faster but demands more cables. It's typically employed in situations where distance is short and high speed is crucial. In contrast, serial transmission is slower but more cost-effective.

In asynchronous serial transmission, bits of the message are sent individually and only when available. The line remains silent when there's no ongoing message. Each character consists of three parts:

- Start bit: This initial bit, always set to zero, marks the beginning of the character.
- Character bits: These bits, positioned between the start and stop bits, constitute the core of the transmitted information. The character follows the start bit.
- Stop bit: The last bit, always set to one, signifies the end of the character. It remains in a state of 1, signifying an idle or waiting status, marking the frame's conclusion.

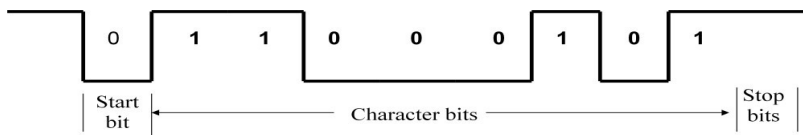


Figure 4.6: Asynchronous Serial Transmission

There are two primary methods for achieving asynchronous serial transmission:

Asynchronous Communication Interface

- This interface operates as a receiver or transmitter, initiated by the CPU via a control register.
- The register receives byte data from the CPU through the bus and transmits it to the appropriate register for transmission.
- Data can be selected and read by the CPU from the receive register, with status register information utilized for input and output flags.

First-In First-Out (FIFO) Buffer

- A FIFO buffer is a storage unit designed for data retention, featuring independent input and output terminals.
- Its key attribute is its ability to accept and output data at differing speeds.

- Placing a FIFO between two units allows it to accept data at one speed and transfer it to another unit at a different speed, beneficial for asynchronous data transfer in applications where data bursts occur.

4.6. Modes of Transfer

Data transfer methods within a computer system can be categorized as synchronous or asynchronous:

- Synchronous: Devices using the same clock for data transfer.
- Asynchronous: Devices operating independently without synchronized timing.

Data transfer between units or devices within a computer can occur in several modes:

- Programmed I/O: Data exchanges driven by I/O instructions in the initial program, usually involving transfers between the CPU and peripherals.
- Interrupt-Initiated I/O: Initiation of data transfer through an interrupt request from peripherals.
- Direct Memory Access (DMA): A method allowing peripherals to access the system's memory directly for data exchange.

In Programmed I/O, data transfer is instigated by specific I/O instructions within the computer's program. Changes in data are executed based on commands in the program, commonly involving transfers between the CPU's scratchpad and peripherals. The CPU monitors the interface and I/O devices, directing data exchange as outlined by the program's instructions.

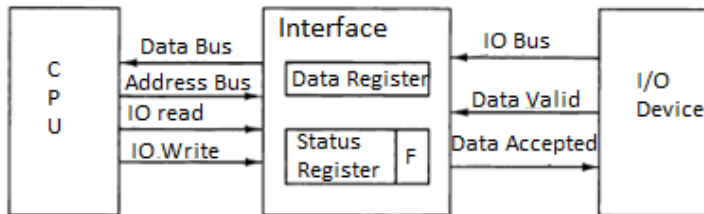


Figure 4.7: Programmed I/O Data Transmission

4.7. Interrupt-Initiated I/O

In the Interrupt-Initiated I/O mode, the interrupt center utilizes interrupt commands to signal the start and end of data transfer, allowing the CPU to proceed with other tasks simultaneously. When the interface detects readiness for data transfer, it generates an interrupt request to the computer. Upon receiving this signal, the CPU temporarily halts its ongoing program execution and shifts to manage the I/O transfer. Upon completion, it returns to its prior task.

Interrupt signals are sent by devices requiring attention, prompting the CPU to suspend its current operation, retrieve the address sent by the device, and proceed accordingly. There are two methods for selecting addresses: Vector Interrupt and Non-Vector Interrupt.

- **Vector Interrupt:** CPU interrupts include data branches, known as interrupt vectors. These vectors direct the CPU to the appropriate handling routine.
- **Non-Vector Interrupt:** Fixed memory locations hold branch addresses assigned for specific interrupts.
- **Priority Interrupt:** Priority interrupts manage multiple interrupt requests based on their urgency. When several requests vie for attention simultaneously, priority levels decide which request takes precedence. For instance, high-speed devices like disks receive higher priority than slower peripherals like keyboards. This scheme ensures critical processes are not delayed by less vital ones. Interrupts can be software or hardware-generated, with hardware managing the overall interrupt environment, determining priorities, and signaling the CPU based on these priorities.

Polling Procedure

Each interrupt holds a branch address indicating the department it affects. The CPU examines these addresses in a predetermined order to prioritize and manage the interrupts' impact. However, polling several devices takes longer compared to servicing multiple I/O devices simultaneously.

Hardware-Based Priority System

This approach bypasses voting and delegates all decisions to relevant hardware, allowing quicker operation as each interrupt has direct access to its routine. The hardware unit receives interrupt requests, prioritizes them, and directs the CPU accordingly, eliminating the need for importance-based decisions in software.

Parallel Priority Interrupt

This method involves a register activated by each device's interrupt signal, alongside mask registers for status checks. It permits higher-priority interrupts while excluding lower-priority ones. Devices are arranged in order of priority, and their interrupt signals are encoded to generate address vectors for the CPU. This setup allows high-speed devices like disks to receive priority attention over slower peripherals like keyboards, ensuring efficient handling of multiple interrupt requests.

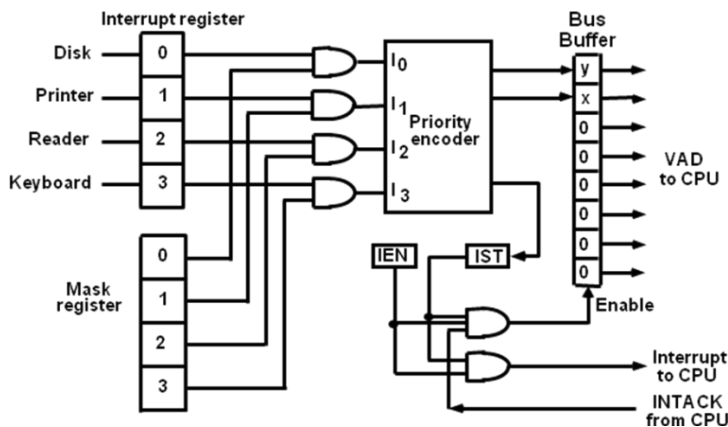


Figure 4.8: Parallel Priority Interrupt

Inputs				Outputs			Boolean functions
I_0	I_1	I_2	I_3	x	y	IST	
1	×	×	×	0	0	1	$x = I'_0 I'_1$ $y = I'_0 I_1 + I'_0 I'_2$ $(IST) = I_0 + I_1 + I_2 + I_3$
0	1	×	×	0	1	1	
0	0	1	×	1	0	1	
0	0	0	1	1	1	1	
0	0	0	0	×	×	0	

Table 4.1: Priority Encoder Truth Table

Serial or Daisy Chaining Priority

In the daisy chaining priority setup, devices requiring attention are interconnected in a sequential manner. Each device connects to the next in line, with the most critical device positioned at the start, and others following suit in order of importance, forming what's known as a daisy chain. Interrupt requests are shared among these devices, creating a network link for these requests.

When a device needs to signal an interrupt, its interrupt line goes low, indicating to the CPU that an interrupt is required. In the absence of any pending interrupts, the interrupt line remains high, and the CPU remains unaware until an interrupt request is generated by activating the interrupt acknowledge line.

This acknowledgment signal originates from the Priority Input (PI) of Device 1. It's passed to the subsequent device in the daisy chain (the Post Priority or PO version) only when Device 1 isn't requesting an interrupt.

If Device 1 triggers a significant interrupt, preventing the next device from acknowledging, it sets its PO output to 0. It then places its interrupt vector address (VAD) on the data bus for the CPU to process during the interrupt cycle.

A device receiving a 0 on its PI input will relay a 0 on its PO output, signalling to the following device that acknowledgment is blocked. Conversely, a device seeking an interrupt and having a 1 on its PI input inhibits the acknowledgment signal by setting its PO output to 0. If no significant interference occurs, the device sends the acknowledgment signal to the next device by setting its PO output to 1.

Consequently, the device with PI=1 and PO=0 is the most crucial device requiring an intervention. This device places its VAD on the bus. The daisy chain structure prioritizes devices based on CPU-approved interrupts, with devices positioned farther down in the chain holding lower importance.

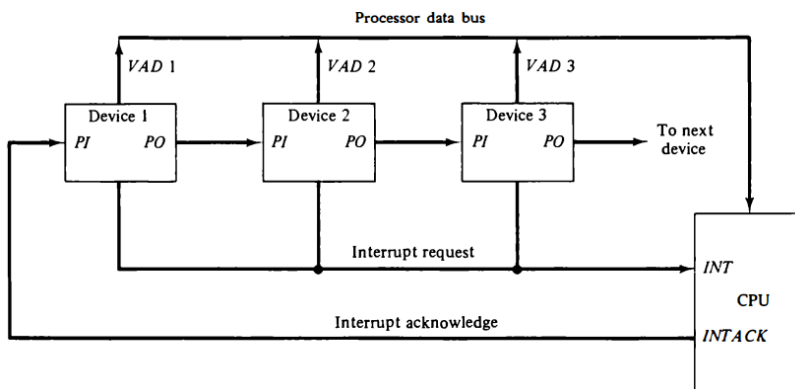


Figure 4.9: Serial or Daisy Chaining Priority

4.8. Direct Memory Access (DMA)

DMA, or Direct Memory Access, is a mechanism facilitating data transfer between memory and input/output devices, independent of CPU intervention. In this process, an external component known as the DMA controller orchestrates the data flow.

DMA operations involve two critical pins: HOLD and HLDA. Initially, the DMA controller initiates a request by raising the bus request (BR) control line. Upon receiving a high signal from the HOLD pin, the processor concludes its ongoing machine cycle and issues the HLDA signal to the DMA controller.

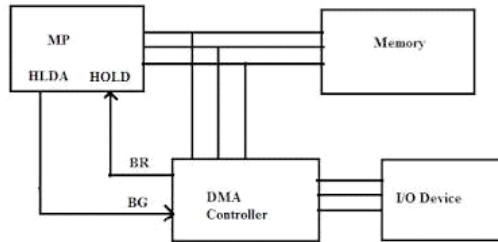


Figure 4.10: Serial or Daisy Chaining Priority

When the DMA controller receives HLDA via the Bus Grant (BG) pin, it takes over the bus, enabling direct data transfer between memory and I/O, bypassing CPU involvement. Throughout this DMA operation, the CPU is unoccupied and can execute other non-data path-dependent tasks.

Upon completing the data transfer, the DMA controller terminates the request by sending a low signal to the HOLD pin. This action returns control of the bus to the processor, maintaining HLDA at a low level.

In the context of direct memory access, data flows to and from the memory unit through the memory bus. For high-speed devices like disks and memory, data transfer speed can be constrained by the CPU's capabilities. To optimize this, DMA bypasses CPU control, allowing hardware to directly manage the memory bus, enhancing data transfer rates.

During DMA transfers, the CPU remains inactive and relinquishes control of the memory bus. This can be achieved through specific controls, such as Bus Request (BR) and Bus Grant (BG), which serve as signals for DMA requests. BR, for instance, prompts the CPU to halt ongoing instructions, placing the address, data bus, and read/write lines in a high impedance state, effectively cutting off outputs.

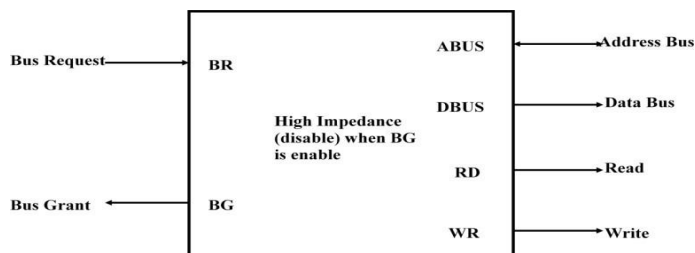


Figure 4.11: CPU Bus Signals for DMA Transfer

The CPU signals the external DMA by activating the Bus Grant (BG) output, granting the DMA control over the bus for transfers without CPU involvement.

When the DMA completes a transfer, it deactivates the bus request (BR) line, enabling the CPU to regain control over the bus and resume its normal operations.

This transition is managed through two key methods:

- **DMA Burst:** In this mode, DMA executes burst transfers by moving multiple memory words sequentially in changing blocks. The DMA controller orchestrates these transfers, maintaining control over the memory bus throughout the process.
- **Cycle Stealing:** Cycle stealing allows the DMA controller to transfer data one word at a time. After each word transfer, control of the bus must be returned to the CPU for its use.

The DMA controller plays a crucial role in this process and typically comprises three key registers:

- **Address Register:** Stores the memory address where data needs to be transferred.
- **Word Count Register:** Keeps track of the number of words to be transferred. It increments/decrements after each word transfer and can be reset.
- **Control Register:** Dictates the mode of transfer.

Communication between the DMA unit and the CPU occurs via bus and control lines. The CPU enables DS (DMA Select) and RS (Register Selection) to designate the scratchpad in DMA entries. RD (Read) and WR (Write) inputs are bidirectional.

When BG (Bus Grant) is set to 0, the CPU can access the DMA register via the bus, enabling read or write operations on the DMA register. However, when

BG = 1, the DMA can directly communicate with memory by determining the address on the address bus and enabling RD or WR control.

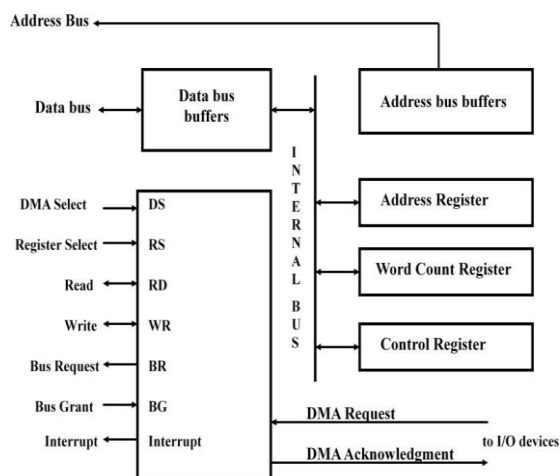


Figure 4.12: Block Diagram of DMA Controller

DMA (Direct Memory Access) facilitates communication between the CPU and DMA via address and data buses. The DMA uses its designated address lines, DS and RS, to initiate communication. The CPU launches DMA through the data bus, allowing DMA to toggle between external devices and memory upon receiving the control command.

When BG (Bus Grant) is set to 0, RD and WR act as input lines enabling the CPU to interact with the internal DMA scratchpad. Conversely, when BG = 1, RD and WR become output lines from the DMA controller to access memory, facilitating read or write operations on specified data.

The interface serves as the connection point between distinct parts of a system. A more reliable method, known as handshake, addresses issues found in gating by providing feedback to the initiating unit during data transfer.

I/O data transfer scheduling follows the instructions embedded in the computer program. In the interrupt initiated I/O method, interrupt settings and commands signal the start and end of the transfer to the device.

Regarding memory organization, it encompasses storage devices that store binary data as objects. Memory falls broadly into two categories:

- Volatile memory: Loses information when power is disconnected.
- Non-volatile memory: Retains data even when power is turned off, providing a permanent storage solution.

4.9. What is the Memory Hierarchy?

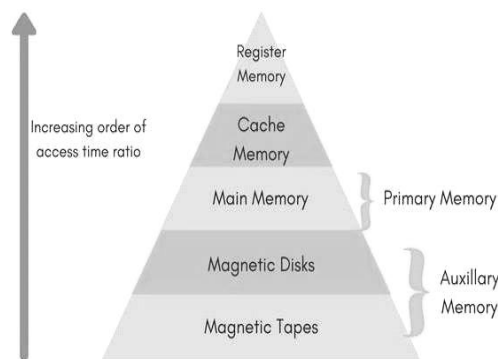


Figure 4.13: Memory Hierarchy

Memory Hierarchy in Computer Systems

Computer memory is organized into five hierarchical levels, each differing in speed and permanence. The processor can access these levels based on its needs. The five levels include

Scratchpad, Cache, Main Memory, Disk, and Tape. The first three levels are volatile memories, losing data upon power-off, while the last two are non-volatile, retaining data permanently.

Primary Memory

Primary memory, also known as internal memory, is directly accessible by the processor. It comprises main memory, cache, and CPU registers.

Secondary Memory

Secondary memory, also referred to as external memory, is accessible to the processor via input/output devices. Examples include optical discs, magnetic discs, and tapes.

Registers

Registers are static RAM within the processor used to hold data words. They typically include the program counter register, a status word register, and an accumulator, aiding decision-making and storing operational data.

Cache Memory

Cache memory stores frequently accessed data from the main memory, allowing quicker access for the CPU. It continually updates itself by replacing old data with new, relevant information.

Main Memory

Main memory serves as the central storage unit directly communicating with the CPU, auxiliary memory, and cache. It comprises RAM, ROM, and their variations.

- RAM (Random Access Memory): Includes DRAM (Dynamic RAM) and SRAM (Static RAM). DRAM is slower and cheaper, refreshed periodically, while SRAM stores data until power-off.
- NVRAM (Non-volatile RAM): Retains data even when powered off, e.g., flash memory.
- ROM (Read-Only Memory): Stores bootloaders and permanent information. Variants include PROM, EPROM, and EEPROM.

Magnetic Disks and Tapes

- Magnetic Disks: Made of plastic or metal, storing data in circular tracks and sectors. Work collectively at high speeds.
- Magnetic Tape: Long film coated with a magnetizable layer used for data backup. Slower access compared to disks.

4.10. Features of Memory Hierarchy

- **Performance:** Designed to bridge the gap in access time between various memory levels, enhancing overall system performance.
- **Capacity:** Increases when moving down the hierarchy, allowing larger data storage.
- **Access Time:** Ranges from quick access in higher levels to slower access in lower levels.
- **Cost per Bit:** Increases as one moves up the hierarchy due to faster and more expensive memory.
- **Associative Memory:** Also known as Content Addressable Memory (CAM), accesses data based on content rather than specific addresses, aiding data optimization.

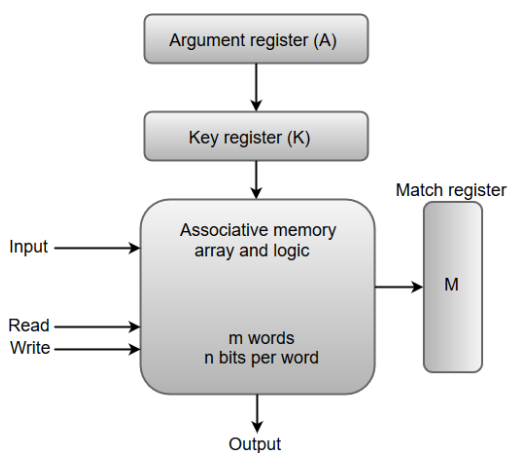


Figure 4.14: Block Diagram of Associate Memory

The hardware layout depicted above represents shared memory in a hierarchical format. It comprises a combination of 'm' words, each containing 'n' objects, alongside specific registers: input, mask, selection, and output.

The input register's primary function is to store or retrieve data within the memory. It can contain a single long word at any given time. The mask, on the other hand, serves to select specific keys or fields from the table, with its length limited by 'n' to match the word size.

The list function houses 'm' objects, corresponding to each memory word. When data in the input register is compared against keys stored in these registers, a match triggers a set bit in the selection register. This selection register acts as a flag for matched data.

The purpose of the selection register is to indicate which data comparisons have returned positive matches. It maintains the information regarding the matching data for further processing.

Regarding the associative memory's relationship with memory arrays and other scratchpads, the figure illustrates how these components interact and operate within the associative memory system.

These are rows of cells marked with the letter C in them. Subscriptions. The first subscript gives the word number, the second defines the position of the bit in the word. That is, cell C_{ij} is the cell at position j in word i . If $K_i = 1$, bit A_i in the parameter register is compared with each bit in column j of the array. This is done for all rows $j = 1, 2, N$. If a match occurs between all parameters and a bit in word I , the corresponding M_1 bit in the match will be set to 1. If one or more items not covered in the scale e do not match the message, clear M_1 to 0.

4.11. Virtual Memory

Virtual memory is a period of non-volatile memory created by a storage facility for a limited period of time.

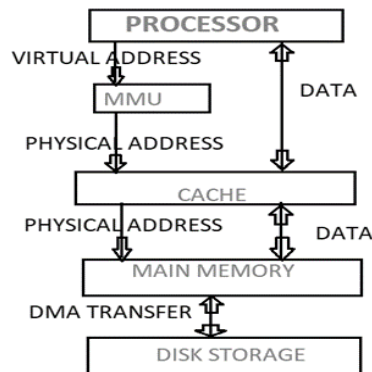


Figure 4.15: Block Diagram of Virtual Memory

Programmers often utilize a larger memory space allocated on disk, termed as virtual memory, to facilitate the development of programs or software requiring extensive space.

Within a program or processor, there's a distinction between the data location referred to by the processor and its corresponding physical memory location. The address provided by the processor is known as the virtual address.

The conversion of virtual addresses into physical addresses is accomplished through a combination of hardware and software mechanisms. Accessing a virtual address directly from the program is feasible. To optimize performance, hardware components like the Memory Management Unit (MMU) are integrated into the system.

In systems using a paging approach, a page table is employed to translate virtual addresses into their corresponding physical addresses. The MMU's core responsibility lies in this conversion process, allowing the transformation of virtual addresses into their respective physical counterparts.

Quick Questions

1. What is the role of the I/O interface?
2. How does the process work?
3. What are the processes required to initiate an I/O operation using DMA?
4. What three types of channels are commonly found in mainframes?
5. Why is DMA more important than CPU when both want memory transfer?
6. What are the advantages of using interrupts to initiate data transfer?
7. Why is DMA needed?
8. What is the difference between subroutine and interrupt service routine?
9. In what situations is an impact mask necessary?
10. How does bus booking usually work?
11. How does the machine work?
12. Difference between synchronous bus and asynchronous bus.
13. Why can't Input/Output devices connect directly to the bus? 14. What is the mainfunction of the I/O system?
14. What is an I/O interface?
15. List the details required when designing the I/O subsystem?
16. What is Direct Memory Access (DMA) and DMA Controller?
17. What is polling?
18. Define USB
19. What is the need of interrupt controller?
20. What is a priority interrupt?
21. Define bus and PCI bus.
22. State the differences between memory mapped I/O and I/O mapped I/O.
23. Define interrupt and exception.
24. What is SCSI?
25. What are the different methods used for handling the situation when multiple interrupts occurs?
26. What is a privileged instruction?

27. What is a bus order?
28. What is a port?
29. What types of ports are there?

Long Answer Questions

1. Describe the functions performed by a typical I/O interface and a typical input-output interface.
2. Discuss the DMA driver data transfer mechanism. 3. Discuss the operation of two input devices
3. Explain the interference model.
4. Detailed description of the standard I/O interface.
5. Use diagrams to explain the functions of SCSI.
6. What is the importance of I/O interface? Compare the features of SCSI and PCI interfaces.
7. Difference between PCI and ISA bus architecture
8. Write briefly:
 - a) Bus Arbitration
 - b) Printer communication process
 - c) USB
 - d) DMA
9. Explain the use of vector interrupts in the process. Why should the response manager be first? The main difference is how it works?
10. Distinguish between subroutines and interrupts. 12. Discuss the impact of I/O data transfer.
11. Describe two methods of using various materials.
12. Write a note about the bus order.
13. Explain the operation of a 2-channel DMA controller.
14. Explains the characteristics of connections and interconnections of technologies.
15. Explain the operation of synchronous and asynchronous data buses.

Multiple Choice Questions

1. The input or output device connected to the computer is called ()
 - a) input/output subsystem
 - b) peripheral device
 - c) interface

- d) Interference
- 2. How many types are there for I/O data transfer? ()
 - a) 2
 - b) 3
 - c) 4
 - d) 5
- 3. Are the keyboard and mouse there? ()
 - a) Input peripheral
 - b) Output peripheral
 - c) Input and output peripheral
 - d) None of the above
- 4. The methods given above for I/O transmission are: ()
 - a) interrupts
 - b) Memory mapping
 - c) C. Programmed I/O
 - d) DMA
- 5. Mapped I/O in memory ()
 - a) I/O devices have separate addresses
 - b) I/O devices and memory share the same address
 - c) Part of the memory is set specifically for I/O set aside for operations
 - d) Locations with memory and I/O devices
- 6. circuits are often used to extend the processor bus to connect devices.
 - a) Router
 - b) Router
 - c) Bridge
 - d) None of the above
- 7. ISA is an architecture standard developed by ()
 - a) IBM
 - b) AT&T Labs
 - c) Microsoft
 - d) Oracle
- 8. SCSI bus is used to provide:
 - a) Single bus
 - b) USB

- c) SCSI
 - d) Parallel bus to connect video equipment to the processor.
9. Which of the following statements about DMA is correct? ()
- a) DMA is a method of transferring data between memory and external devices without processor intervention.
 - b) The DMA controller acts as a processor for DMA transfers and ignores all processes.
 - c) DMA controller has 3 temporary registers.
 - d) All of the above
10. Name of controller _____()
- a) 16 bit
 - b) 32 bit
 - c) 64 bit
 - d) 128 bit
11. In memory map _____() In I/O
- a) I/O devices and memory share the same address
 - b) I/O devices have separate addresses
 - c) Memory bank and I/O device have a contiguous address
 - d) Part of memory Dedicated to I/O operation
12. Generally used to connect I/O devices Bus structure ()
- a) Star bus structure
 - b) Multiple bus
 - c) Single data path
 - d) Node to node data path structure
13. The advantages of the I/O mapping device for memory mapping are:
- a) The former provides faster data transfer
 - b) The space of devices connected using I/O mapping is not larger
 - c) The device is larger than one line process less
 - d) There is no such thing
14. Notify the system of read or write operations on ()
- a) Add additional objects to the address
 - b) Enable read or write operations on the device win
 - c) Cause appropriate interrupt signal
 - d) Bus Send a special signal throughout

15. To overcome the delay in I/O devices and processing speed, we use:
- a) No field
 - b) Status flag
 - c) Interrupt signal effect
 - d) Exemption
16. The way to access the I/O device by checking the status flag is:
- a) Check I/O
 - b) Memory mapped I/O
 - c) report I/O
 - d) above None
17. Method of synchronizing the processor with the I/O device to which the device sends a signal when ready
- a) Exception
 - b) Signal processing
 - c) interrupt
 - d) DMA
18. Providing faster I/O transfer The technique is ()
- a) Interrupt
 - b) Memory
 - c) Programmed I/O
 - d) DMA
19. The process of regularly checking the correct flag in the processor is called ()
- a) Selection
 - b) Control
 - c) Review
 - d) Echo
20. Requested line ()
- a) Text line
 - b) Control line
 - c) Address line
 - d) None of the above
21. Name of the networked input or output device computer ()
- a) Input/output subsystem
 - b) Peripheral device

- c) Interface
 - d) Interrupt
22. How many types of I/O data transfer are there? ()
- a) 2
 - b) 3
 - c) 4
 - d) 5
23. What are keyboard and mouse? ()
- a) Input peripherals
 - b) Output peripherals
 - c) Input-Output peripherals
 - d) None of the above
24. The method which offers higher speeds of I/O transfers is _____()
- a) Interrupts
 - b) Memory mapping
 - c) Program-controlled I/O
 - d) DMA
25. In memory-mapped I/O _____()
- a) The I/O devices have a separate address space
 - b) The I/O devices and the memory share the same address space
 - c) A part of the memory is specifically set aside for the I/O operation
 - d) The memory and I/O devices have an associated address space
26. The _____circuit is basically used to extend the processor BUS to connect devices.
- a) Router
 - b) Router
 - c) Bridge
 - d) None of the above
27. ISA is an architecture standard developed by ()
- a) IBM
 - b) AT&T Labs
 - c) Microsoft
 - d) Oracle
28. The SCSI bus is used to connect video devices to the processor by providing
- a) A bus

- b) USB
 - c) SCSI
 - d) Parallel bus
29. Which of the following statements about DMA is correct?
- a) DMA is a method that allows data transfer between memory and external devices without interrupting the process.
 - b) DMA controller acts as a processor for DMA transfer and ignores all operations.
 - c) DMA controller has 3 scratchpads.
 - d) (d) All of the above
30. Name of controller _____()
- a) 16 bit
 - b) 32 bit
 - c) 64 bit
 - d) 128 bit
31. Besides the ability to program operation with a specific device, which interface allows communication with most specific devices and outputs?
- a) Parallel peripheral interface
 - b) Serial communication interface
 - c) Special purpose interface
 - d) Direct memory access interface
32. Which lines should control or control the transmission between two devices in asynchronous mode using a different bus to understand the transmission status? ()
- a) Control line
 - b) Letter line
 - c) Transmission line
 - d) Handshake line
33. Which register is incremented and decremented with each byte transferred in direct memory access (DMA)? ()
- a) Address register and byte count register
 - b) Control register and byte count register
 - c) Mail register and code Group count register
 - d) status register and byte count register

34. If the Z flag is 0, which instruction tells the routine to go to the address specified by the 16-bit value?
- a) CZ address
 - b) CNZ address
 - c) CPE address
 - d) CPO address
35. What instruction is used to set an interrupt from the control output in a workflow?
- a) SIM
 - b) STC
 - c) SBI data
 - d) SUI data
36. Which command has the dual function of reading interrupt events and data entry serial objects? ()
- a) RNZ
 - b) RZ
 - c) RPO
 - d) RIM
37. Which type of instruction will jump to the new location when completed from the continuation of the previous program? ()
- a) Data transfer instructions
 - b) Logic instructions
 - c) Bit-oriented instructions
 - d) Irregular instructions
38. The bit is processed or controlled by the microcontroller and CPOL. What is the relationship between clock and data in an integrated circuit?
- a) CPHA
 - b) DORD
 - c) MSTR
 - d) SPE
39. What happens when the WCOL bit is running if the SPDR register is saved prematurely without affecting the operation of the document period? normal mode?
- a) Read conflict
 - b) Conflict
 - c) Write conflict

- d) Data conflict
40. Which can be considered as a disadvantage of microprocessor?
- a) The motherboard is larger than the microcontroller
 - b) Different devices can be used for different functions
 - c) Notification notification for each of its quirks
 - d) All of the above
41. The biggest problems of microcontrollers include
- a) Onboard peripheral performance information
 - b) High cost
 - c) Difficulty in operation
 - d) None of the above
42. Why are microcontrollers better and more popular than microprocessors?
- a) Due to simple handling features
 - b) Due to operational level robustness
 - c) Faster processing due to energy savings
 - d) All of the above
43. Primary objects What is the name given to the group it forms?
- a) Code
 - b) Obstruction
 - c) Word
 - d) All of the above
44. Which gateway configuration allows different independent nodes to be used over each other?
- a) OR gate
 - b) Non-linear hybrid gate
 - c) a and b
 - d) None of the above
45. OR gate configuration can reduce the interference of light, that's exactly what it is called
- a) Dimmer circuit
 - b) Non-linear hybrid circuit
 - c) Synchronous circuit
 - d) All of the above
46. Find the difference a
- a) Light pen

- b) USB stick
 - c) Keyboard
 - d) VDU
47. An I/O interface is required when an I/O device is operated by
- a) Processor
 - b) Memory
 - c) CU
 - d) None
48. Concurrent data transfer includes:
- a) Sorting
 - b) Stacking
 - c) Linked list
 - d) B-tree
49. The branch path of the I/O program is:
- a) Program control branch
 - b) CPU forced branch
 - c) a & b
 - d) None
50. In addition to the procedures required to initiate I/O operations, subroutines are required to
- a) transfer data messages between the I/O device and main memory
 - b) hold I/O. Exception for I/O devices
 - c) Waiting for end of I/O device operation
 - d) All of the above

Keys

1. (c) 2. (b) 3. (a) 4. (d) 5. (b) 6. (c) 7. (a) 8. (d) 9. (d) 10. (b) 11. (a) 12. (c) 13. (c) 14. (d) 15. (b)
 16. (a) 17. (c) 18. (d) 19. (a) 20. (b) 21. (a) 22. (b) 23. (a) 24. (d) 25. (a) 26. (c) 27. (a) 28. (c)
 29. (d) 30. (a) 31. (d) 32. (c) 33. (a) 34. (a) 35. (b) 36. (b) 37. (c) 38. (d) 39. (a) 40. (b) 41. (a)
 42. (b) 43. (c) 44. (c) 45. (d) 46. (b) 47. (a) 48. (a) 49. (c) 50. (d)

UNIT V - INTRODUCTION TO PROCESSORS

Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics. Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor. Multi Processors: Characteristics of Multiprocessors, Interconnection Structures, Interprocessor Arbitration, Interprocessor communication and Synchronization, Cache Coherence.

5.1. Reducing Instruction Issues in Computers: (RISC)

RISC, or Reduced Instruction Set Computing, focuses on optimizing and streamlining instruction execution in computers. The goal is to minimize instruction times by simplifying and reducing the number of instructions needed for each operation. In RISC processors, an instruction cycle typically requires just one clock cycle, consisting of fetch, execute, and writeback phases. These processors efficiently handle complex instructions by breaking them down into simpler ones. Notable RISC processors include SUN's SPARC, PowerPC, Microchip PIC processor, and RISC-V.

Key Features of RISC

- Simplified instructions for easier decoding and decision-making.
- More general-purpose registers available for operations.
- Each instruction typically completes in a single clock cycle.
- Compact instruction size, often smaller than a word.
- Enables efficient pipelining for improved performance.
- Supports various file formats.
- Utilizes simpler address formats for operations.

RISC Architecture: RISC architecture has found extensive applications in reliable devices such as the Apple iPod, mobile phones, smartphones, and gaming consoles like the Nintendo DS.

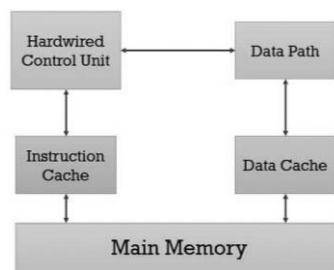


Figure 5.1: RISC Processor Architecture

Characteristics of RISC Processors

- **Single Clock Cycle:** A RISC processor typically executes instructions within a single Clock Per Cycle (CPI). Each CPI encompasses fetching, decision-making, and execution of computer instructions.
- **Pipelining Technology:** RISC systems employ pipeline technology, allowing the processing of instructions at various stages or locations simultaneously to enhance performance.
- **Abundant Registers:** RISC processors are designed with a large number of registers to store instructions, minimizing memory interference and enabling rapid computer response.
- **Use of Simple and Complex Instructions:** RISC supports both simple command forms and more elaborate expressions to efficiently complete pipeline tasks.
- **LOAD and STORE Instructions:** These processors employ LOAD and STORE instructions to access memory locations directly.
- **Simplicity and Limited Instructions:** The simplicity and constraint of instructions in RISC reduce the execution time of processes.

Advantages of RISC Processors

- **Enhanced Performance:** Simplified and limited instructions lead to improved performance.
- **Cost-Efficient Design:** More transistors reduce design costs.
- **Optimal Space Utilization:** RISC's simplicity allows for efficient utilization of microprocessor space.
- **Single Cycle Operations:** Operations are often completed in a single cycle due to the simple and fast design.

Disadvantages of RISC Processors

- **Execution Dependency:** The performance of a RISC processor can be influenced by execution dependencies between instructions.
- **Complexity for Programmers:** Programmers and compilers might resort to complex instructions.
- **Need for High Speed and Cache:** RISC processors require speed and substantial cache to swiftly handle numerous instructions.

5.2. Characteristics of CISC Processors

- **Complex Instructions:** CISC instructions are intricate, leading to complex instruction decoding.
- **Variable Execution Times:** Instructions in CISC processors may take varying durations to complete.
- **Larger Addressing Modes:** CISC instructions might possess larger addressing modes.
- **Sparse Scratchpad:** CISC focuses less on registers as most processing occurs in memory.
- **Wide Instruction Variety:** CISC processors support various instruction types.

Architecture of CISC Processors

CISC architecture aims to condense program size by encompassing numerous functions within each instruction. These processors are geared toward reducing memory costs, although larger programs necessitate more memory space, potentially increasing overall costs.

Examples of CISC Processors include VAX, AMD, Intel x86, and System/360.

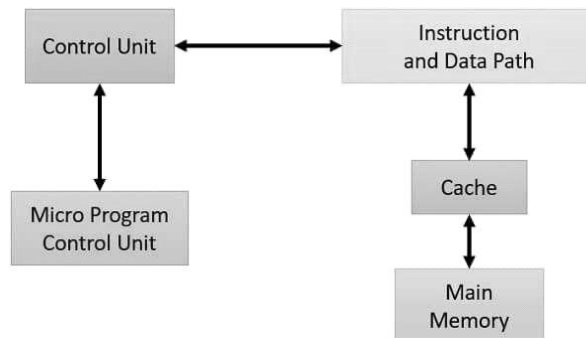


Figure 5.2: CISC Processor Architecture

Advantages of CISC Processors

- **Compiler Efficiency:** CISC processors demand some effort from the compiler to efficiently translate higher-level or complex instructions into machine language.
- **Compact Code Length:** CISC instructions tend to be concise, reducing the need for extensive memorization.
- **Low RAM Requirements:** Each CISC instruction requires minimal RAM storage.
- **Streamlined Guide Creation:** Developing guides for CISC involves minimal low-level work.
- **Power Management:** CISC incorporates power management processes that regulate clock speed and voltage, aiding in power usage optimization.

- **Instruction Set Economy:** CISC achieves similar instructions using fewer instruction sets compared to RISC.

Disadvantages of CISC Processors

- **Slower Execution:** CISC chips execute instructions slower per cycle compared to RISC processors.
- **Reduced Machine Performance:** Slower clock speed negatively impacts the overall performance of the machine.
- **Complex Pipeline Execution:** Utilizing the pipeline in CISC processors can be intricate and challenging.
- **Transistor Demands:** CISC designs require more transistors than RISC counterparts.
- **Limited Utilization:** Only approximately 20% of the available instructions in a programming event are utilized in CISC architecture.

Difference between the RISC and CISC Processors

RISC	CISC
• Emphasizes software optimization for the instruction set. • Utilizes a hardwired unit for programming. • Requires multiple register sets to store instructions. • Simplifies instruction decoding. • Uses a limited number of instructions for faster execution. • Employs LOAD and STORE instructions for register-to-register program interaction. • Contains more transistors on memory registers. • Suitable for high-end applications like telecommunications and image processing. • Features fixed-format instructions. • Programs written for RISC architecture occupy more memory space. Examples: ARM, PA-RISC, Power Architecture, Alpha, AVR, ARC, SPARC	• Emphasizes hardware optimization for the instruction set. • Uses microprogramming in the unit. • Utilizes a single register set to store instructions. • Involves complex instruction decoding. • Uses a larger number of instructions, leading to longer execution times. • Employs LOAD and STORE instructions for memory-to-memory program interaction. • Has transistors dedicated to storing complex instructions. • Suited for low-end applications like home automation and security systems. • Features variable-format instructions. • Programs written for CISC architecture occupy less memory space. Examples: VAX, Motorola 68000 series, System/360, AMD, Intel x86 CPUs

5.3. Pipeline and Vector Processing

Parallel Processing

The primary objective of parallel processing is to enhance computer power and productivity by executing multiple operations simultaneously. In systems employing parallel processing, there can be two or more Arithmetic Logic Units (ALUs), allowing the concurrent execution of two or more instructions.

The goal is to amplify the volume of work that can be accomplished within a given timeframe. The process involves dividing data and establishing numerous connections to run a large number of operations concurrently. Multifunctional organizations frequently integrate parallel processing into management systems to efficiently oversee various individual activities.

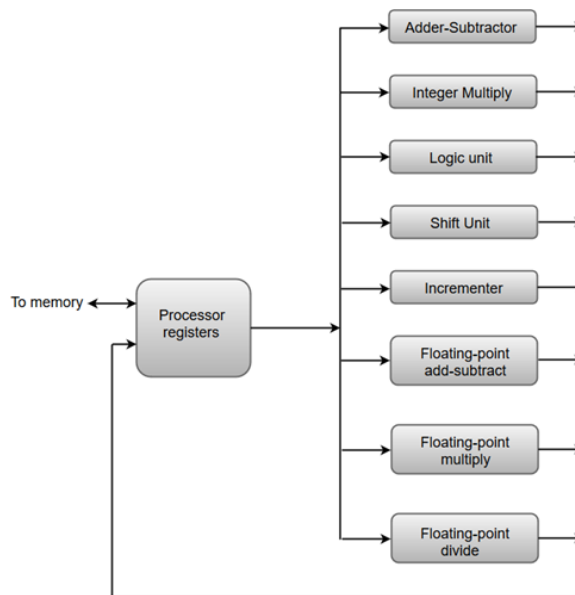


Figure 5.3: Parallel Processing Structure

- Applying an operand within the scratchpad involves assigning it to a specific position according to the operation outlined in the associated instruction.
- Each unit in the diagram performs designated functions, where adders and integer multipliers handle arithmetic operations, and floating-point operations operate via three parallel circuits.
- Simultaneous logical, transfer, and increment operations can occur on distinct files.
- Given the independence of these units, changes in one number can affect another within the system.

- Multifunctional organizations frequently integrate management systems to oversee diverse product activities.

Regarding parallel systems, definitions can vary based on the:

- Internal processor organization
- Interaction model among processes
- Data flow throughout the system

Architectural classification or Flynn's classification:

Flynn's categorization divides computers into four main categories:

- SISD - Single Instruction stream, Single Data stream
Sequential execution of instructions without parallelism.
- SIMD - Single Instruction stream, Multiple Data stream
Multiple processors execute the same instruction on different data items.
- MISD - Multiple Instruction stream, Single Data stream
Mostly theoretical; no practical implementation exists.
- MIMD - Multiple Instruction stream, Multiple Data stream

Computers capable of simultaneous multiple task execution, such as multiprocessor and multicomputer systems.

Pipeline Processing

Pipeline processing involves gathering and executing instructions through a sequence of stages within pipes. Instructions are stored and executed in a predetermined order, forming a tubular structure where instructions enter at one end and exit at the other.

This method divides a series of processes into subprocesses, akin to an industrial assembly line, allowing for simultaneous calculations in various sections. Temporary records hold information in each section, progressing through stages indicated by labels and connections.

Pipelines improve overall efficiency by enabling simultaneous computations in different sections. Each section has a log entry, next link, and uses temporary records to store and process data, ensuring progress through clocked cycles for task completion.

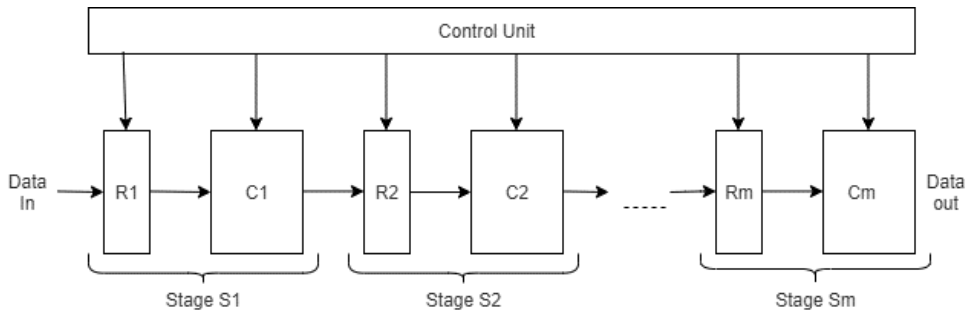


Figure 5.4: Serial or Daisy Chain Pipeline Processing

Example

Let's consider a straightforward example to demonstrate pipeline organization by combining division and addition operations with a sequence of numbers represented as $A_i * B_i + C_i$ for $i = 1, 2, 3, \dots, 7$.

The various stages within the pipeline handle each operation:

- R1 stores A_i , R2 stores B_i .
- R3 holds the result of $R1 * R2$ (multiplication), while R4 contains C_i .
- R5 stores the sum of R3 and R4.
 - The clock is used to load new data into these five registers.
 - During the first clock cycle, A_1 and B_1 are assigned to R1 and R2, respectively.
 - In the second clock cycle, R1 and R2 undergo multiplication ($R1 * R2$), while C1 is assigned to R4.
 - Simultaneously, A_2 and B_2 are loaded into R1 and R2 during this clock cycle.
 - The third clock cycle executes all these parts together.
 - In this cycle, A_3 and B_3 are processed in R1 and R2, the multiplication result ($R1 * R2$) goes to R3, C2 is assigned to R4, and the sum of R3 and R4 is stored in R5.
 - To load the pipeline and capture the initial output of R5, three clock pulses are necessary.
 - Subsequently, the pipeline generates a new output with each clock cycle, moving data through the pipeline step by step.
 - This process continues until all input data has flowed through the pipeline. If there's no more input data, the clock must persist until the final output exits the pipeline.

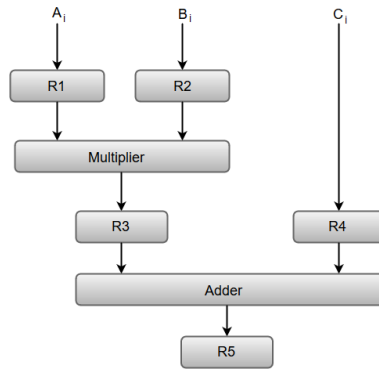


Figure 5.5: Pipeline Organization Structure

Different types of pipelines can be categorized as follows:

- Arithmetic Pipeline
- Instruction Pipeline

Arithmetic Pipeline

This pipeline is commonly found in various computer systems and is utilized for floating-point operations and fixed-point multiplication, among other functions. For instance, in a floating-point arithmetic pipeline, the following concept is employed:

$$X = A * 2^a$$

$$Y = B * 2^b$$

Where A and B represent mantissas (the value portion of floating-point numbers), and 'a' and 'b' denote exponents. Execution of floating-point addition and subtraction involves four steps:

- Comparison of indices
- Setting of mantissas
- Addition or subtraction of mantissas
- Normalization of the result

Intermediate outcomes of these operations are temporarily stored in a list. These arithmetic pipeline units are typically present in high-speed computers and are applied in scientific problems requiring floating-point arithmetic, fixed-point number multiplication, and similar operations.

The steps involved in performing minor operations within these pipelines include:

- Comparison of terms: Determining which number is greater than the other.

- Aligning mantissas: Shifting the mantissa of the smaller number according to the difference in exponents.
- Mantissa addition or subtraction: Accounting for differences in exponents and performing addition or subtraction accordingly.
- Generating/normalizing results: Adjusting the mantissa and exponent to balance the result, either by shifting or adding zeros based on overflows or underflows.

This process ensures accurate arithmetic computations in floating-point operations while managing exponent and mantissa manipulations effectively.

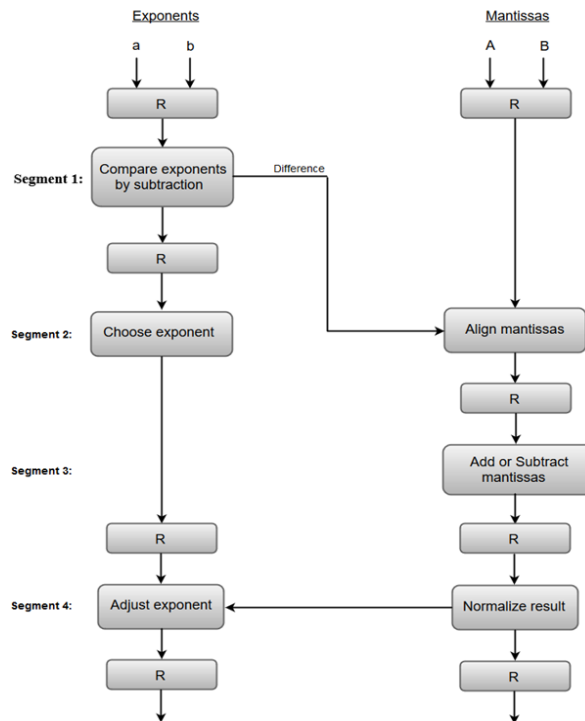


Figure 5.6: Arithmetic Pipeline

Instruction Pipelining

Instruction pipelining is a technique extensively used to enhance the efficiency of computer systems. It significantly contributes to system reliability while reducing cycle time, allowing for the simultaneous execution of multiple instructions.

This method isn't limited to data streams but extends to instruction processing. Consider a computer equipped with a two-stage pipeline, integrating first-in, first-out (FIFO) and instruction-complete (PC) systems.

In computers dealing with complex instructions, additional steps are incorporated to execute instructions effectively. The instruction pipeline operates by fetching instructions from memory while concurrently executing the previous instruction in another segment of the pipeline. This concurrent processing enables multiple tasks to proceed simultaneously, enhancing overall efficiency.

For optimal performance, dividing the instruction cycle into equal time periods is crucial. Typically, a computer performs a sequence of operations that involve retrieving instructions from memory, decision-making based on these instructions, calculating appropriate addresses, fetching operands from memory, executing instructions, making adjustments, and storing results.

For instance, a four-part instruction line can be optimized by combining instruction decision-making with address calculation and merging instruction execution with result storage. Consequently, up to four operations can overlap within an instruction cycle, allowing the simultaneous execution of multiple instructions for enhanced efficiency.

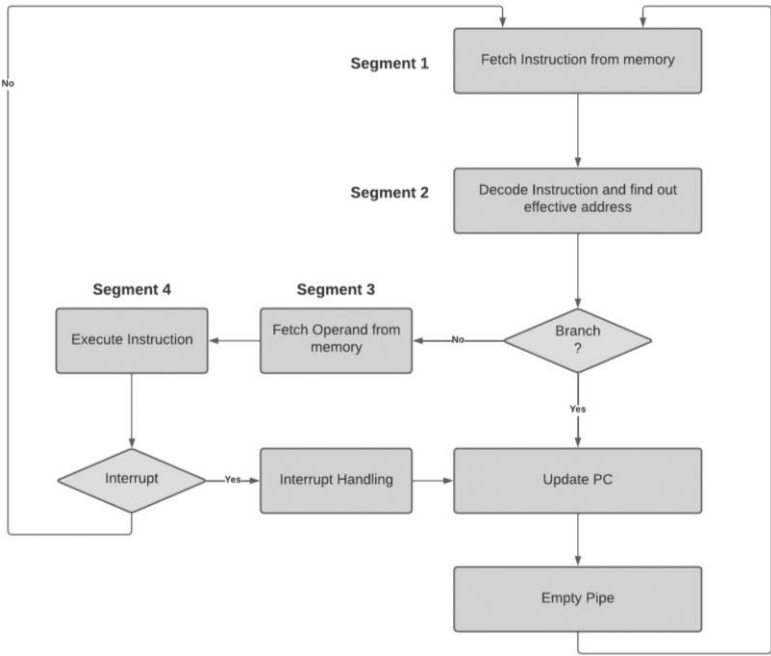


Figure 5.7: Instruction Pipeline

An instruction sequence may lead to a branch from the regular flow of instructions. In such cases, the operations in the final two segments conclude, retaining all the data stored within the command.

Likewise, an interrupted request prompts the channel to pause, preserving its data, and then resumes with a new address. Time is typically segmented into equal steps on the horizontal axis. These segments are symbolically represented in the illustration:

- F1 denotes the segment responsible for carrying out the instructions.
- DA refers to the part that determines instructions and computes the correct address.
- Fo represents the department where the task is initiated.
- EX embodies the segment responsible for executing the instructions.

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 5.8: Timing of Instruction Pipeline

Assuming that the processor has separate instruction and data memory, allowing operations in F1 and PC to occur simultaneously, each section in the pipeline operates on a different instruction without branching verbs. In step 4, the following actions occur:

- Instruction 1 is executed in the EX-section.
- The operands of instruction 2 are fetched from the FO section.
- Instruction 3 is determined in the DA section.
- Instruction 4 is retrieved from memory in the FI section.

Now, let's consider the case where instruction 3 is a branch instruction. During step 4, the transfer of other instructions from F1 to DA is blocked until the branch instruction is completed in step 6. If a branch occurs, new commands are fetched in step 7. If there is no branch, the previously fetched instructions from step 4 can be used, and the pipeline continues until a new branch is encountered.

However, if the EX-part needs to store the result in data memory and the FO part needs to receive the operand, an additional delay occurs in the pipeline. In this case, the FO section must wait for the EX-section to finish its work.

There are three main problems that can disrupt the pipeline:

- Conflicts when both sides access memory simultaneously can be resolved by using separate instruction and data memory.

- Information conflicts occur when instructions depend on the result of previous instructions, but the result is not yet available.
- Data Dependency occurs when an expression depends on the result of a previous expression, but that result does not yet exist.

For branching, it is complicated by branch and other instructions that change the PC value.

In the case of RISC pipelines, the simplicity of the instruction set allows for effective implementation of the instruction pipeline with a small number of operations, each completed in one clock cycle. Data transfer instructions in RISC are limited to load and store instructions, and conflicts are avoided by using two independent data paths and signals.

The example of a three-stage pipeline for RISC processors involves dividing the instruction cycle into three operations: Test Instructions, ALU Operation, and Execute Instructions. The processor unit contains various registers, and the instruction cycle can be completed in three stages.

Finally, the concept of delayed load is discussed, considering the operation of four commands: LOAD, LOAD, ADD, and STORE. Each of these commands involves fetching or storing data at different stages in the pipeline.

In rewriting, these concepts can be presented in a concise and clear manner to avoid plagiarism.

Clock cycles:	1	2	3	4	5	6
1. Load R1	I	A	E			
2. Load R2		I	A	E		
3. Add R1 + R2			I	A	E	
4. Store R3				I	A	E

(a) Pipeline timing with data conflict

Clock cycle:	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1 + R2				I	A	E	
5. Store R3					I	A	E

(b) Pipeline timing with delayed load

Figure 5.9: Three Segment Pipeline Timing

In the RISC processor's pipeline, if the function in R2 is not yet defined in a specific clock cycle, it might lead to conflicting data in expression 3. For instance, in clock cycle 4, Part E places data memory in R2, but the section in clock cycle 4 utilizes data from R2, which might not have been updated from memory yet, causing incorrect values.

The responsibility falls on the compiler to ensure that instructions following a load instruction utilize data fetched from memory. If no valid instruction is found after a load operation, the compiler might introduce no-operation instructions, leading to wasted clock cycles. This concept, delaying the use of loaded data, is termed "lazy loading."

To compensate for data conflicts in the pipeline, an out-of-order instruction might be employed, adding one clock cycle to resolve the issue of using outdated data.

In the case of delayed branching, RISC processors depend on compilers to reorder branches so they can efficiently function in the pipeline. This strategy, known as "slow branching," involves examining instructions before and after branches to add essential instructions gradually.

An example illustrating delayed branching involves a program with five instructions:

- 1. Load from memory to R1
- 2. Increment R2
- 3. Add R4 R3
- 4. Subtract R6 R5
- 5. Branch to address X

The compiler may add two no-op instructions after the branch instruction, and the branch address is sent to the PC in clock cycle 7. Resetting occurs by introducing additional instructions after the branch command, updating the PC to the value X at clock cycle 5.

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

Figure 5.10: No Operation Instructions

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in X						I	A	E

Figure 5.11: Rearranging the Instructions

The compiler introduces two dummy instructions after the branch. During the 7th cycle, the branch address X is conveyed to the PC. Instruction X gets delayed by two cycles due to instruction 2. Execution of instruction X begins in the 8th clock cycle after the program counter (PC) is updated.

The insertion and deletion instructions have been rearranged after the branch instructions instead of before them. Analyzing the pipeline timing reveals that the computer updates to X by the 5th clock cycle. However, the addition and subtraction instructions are fetched and executed correctly. For instance, if the load instruction is at address 101 and X equals 350, the branch instruction moves to address 103. The add instruction is fetched from address 104 and executed in clock cycle 6, while the subtract instruction is fetched from address 105 and executed in clock cycle 7. Since the value of X is transferred to the PC at clock cycle 5 in segment E, the instruction fetched from memory at clock cycle 6 is from address 350, which aligns with the branch address.

Vector (Array) Processing involves computational problems that exceed the capabilities of conventional computers. These problems demand extensive computations on multiple data items, which would take a traditional scalar processor days or weeks to complete.

Super scalar processors, unlike scalar processors that work on single data items with simple instructions, operate on multiple instructions simultaneously. This approach enhances processing efficiency significantly.

Applications of Vector Processors span various domains like image processing, artificial intelligence, medical diagnosis, data analysis, weather forecasting, mapping the human genome, among others.

In Vector Operations, scientific problems often necessitate arithmetic operations on extensive arrays of numbers. Vectors, represented as ordered sets in a one-dimensional array, eliminate the overhead associated with fetching and executing instructions in program loops.

A vector instruction might look like $C(1:100) = A(1:100) + B(1:100)$, assuming vector operands are in memory.

Another design approach involves processors with multiple registers to store all operands in pre-assigned registers. The vector instruction specifies CPU registers' set based on address and length.

Matrix Multiplication involves numerous multiplication and addition operations. For instance, the multiplication of two $n \times n$ matrices requires n^2 elements or n^3 operations. For example, consider the equation for two 3×3 matrices, A and B.

◆ Vector Instruction Format :

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length
ADD	A	B	C	100

◆ Matrix Multiplication

- 3 x 3 matrices multiplication : $n^2 = 9$ inner product

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

$$\gg c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} :$$

- Cumulative multiply-add operation : $n^3 = 27$ multiply-add

$$c = c + a \times b$$

$$\gg c_{11} = c_{11} + a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} :$$

$$\textcircled{1} \textcircled{1} \textcircled{2} \textcircled{2} \textcircled{3} \textcircled{3}$$

$$9 \times 3 \text{ multiply-add} = 27$$

Figure 5.12: Example of Matrix Multiplication Using Vector Processing

The calculation $c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$ involves three multiplications and three additions. In a general inner product, the sum of k product terms can be expressed as $C = A_1B_1 + A_2B_2 + A_3B_3 + \dots + A_kB_k$, where k may be equal to 100 or even 1000 in typical applications.

The inner product calculation on a pipeline vector processor is depicted in the figure below:

$$C = A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \dots$$

$$+ A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \dots$$

$$+ A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \dots$$

This arrangement optimizes the computation of the inner product on a vector processor with a pipeline. Each term is calculated in parallel, taking advantage of the pipeline's ability to perform multiple operations simultaneously.

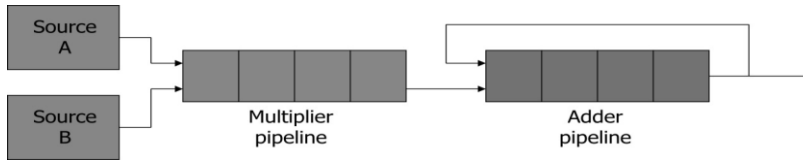


Figure 5.13: Vector Processing

5.4. Memory Interleaving

Memory interleaving is a technique commonly employed by pipeline and vector processors to facilitate simultaneous memory access from multiple sources. For instance, in an instruction pipeline, the fetching of an instruction and its corresponding operand might be required simultaneously from different memory segments. Similarly, an arithmetic pipeline often necessitates multiple operands to enter the pipeline concurrently.

Rather than utilizing two distinct memory buses for simultaneous access, memory can be divided into several modules interconnected to shared memory address and data buses. Each memory module consists of a memory array alongside its dedicated address and data registers.

The diagram below illustrates a memory unit comprising four such modules, which demonstrates how memory can be divided and interconnected for improved access and data handling in pipeline and vector processors.

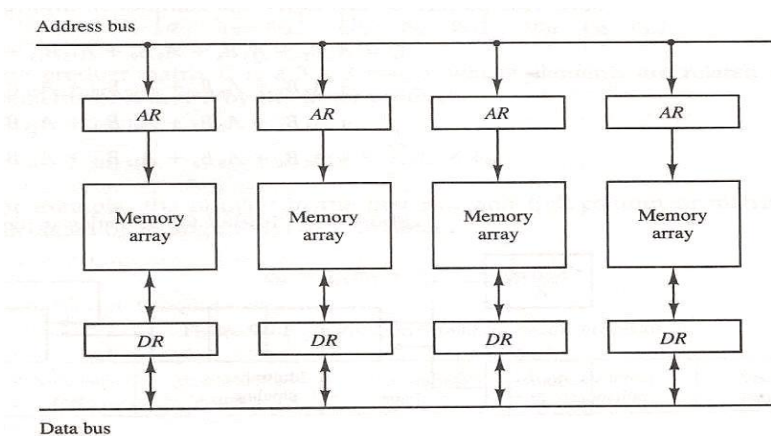


Figure 5.14: Multiple Module Memory Organization

Modular memory brings forth a technique called interleaving, where different sets of addresses are assigned to distinct memory modules. This strategy reduces effective memory cycle time by a factor nearly equivalent to the number of modules, enhancing overall memory access efficiency.

Moving on to Array Processors

There are two primary types of array processors:

Attached Array Processor

Acts as an auxiliary unit, aiming to boost the host computer's performance, especially in specific numerical computation tasks. Facilitates parallel processing through multiple functional units that connect to the host.

SIMD Array Processor (Single-Instruction Multiple-Data)

Organized to execute vector instructions via multiple functional units operating based on a common instruction.

The Attached Array Processor plays a role in augmenting computer performance by offering vector processing capabilities for complex scientific applications. It integrates with the host computer through an input and output controller, acting as an external interface. This auxiliary system receives data from the main memory and transfers it to local memory via a high-speed bus.

These additional systems cater to specialized arithmetic applications, enhancing the capabilities of general-purpose computers. The primary aim behind such an add-on array processor is to provide affordable vector processing abilities to conventional computers, delivering a fraction of the capabilities of a supercomputer at a reduced cost.

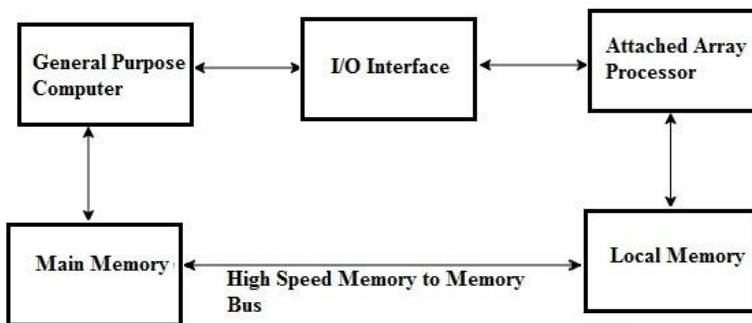


Figure 5.15: Attached Array Processor

SIMD Array Processor

An SIMD array processor constitutes a computer equipped with numerous processing units functioning in parallel. Within this architecture, a collection of identical Processing Elements (PEs) exists, each endowed with its own local memory, denoted as M.

Every PE unit encompasses crucial components such as an Arithmetic Logic Unit (ALU), a dedicated floating-point arithmetic unit, and a set of operational registers. The processor operates by broadcasting vector instructions simultaneously to all PEs.

To manage the execution of vector instructions effectively, masking schemes come into play, controlling the operational state of each PE. Each individual PE unit is equipped with a flag that is toggled to indicate active or inactive status during processing.

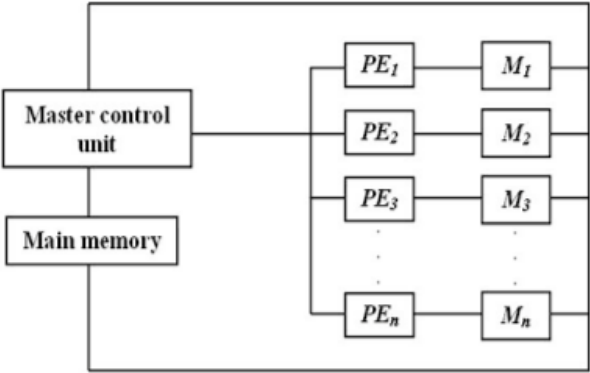


Figure 5.16: SIMD Processor

5.5. Multiprocessors

Multiprocessors refer to a collection of processors linked together via a communication network. This system comprises two or more CPUs interconnected with memory and input-output equipment.

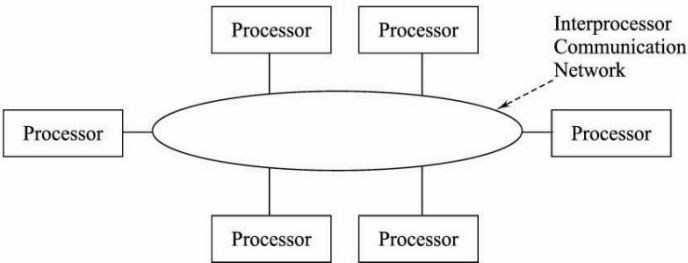


Figure 5.17: Multiprocessor

Multiprocessor systems are categorized as Multiple Instruction Stream, Multiple Data Stream (MIMD) systems. While both multiprocessors and multicomputers facilitate concurrent operations, they differ in their structure and interaction.

Multicomputers consist of several independent computers linked via a network, where communication between them might be optional. On the other hand, in a multiprocessor system, there's a single OS control governing interactions among processors and coordinating all system components to collaboratively address problems.

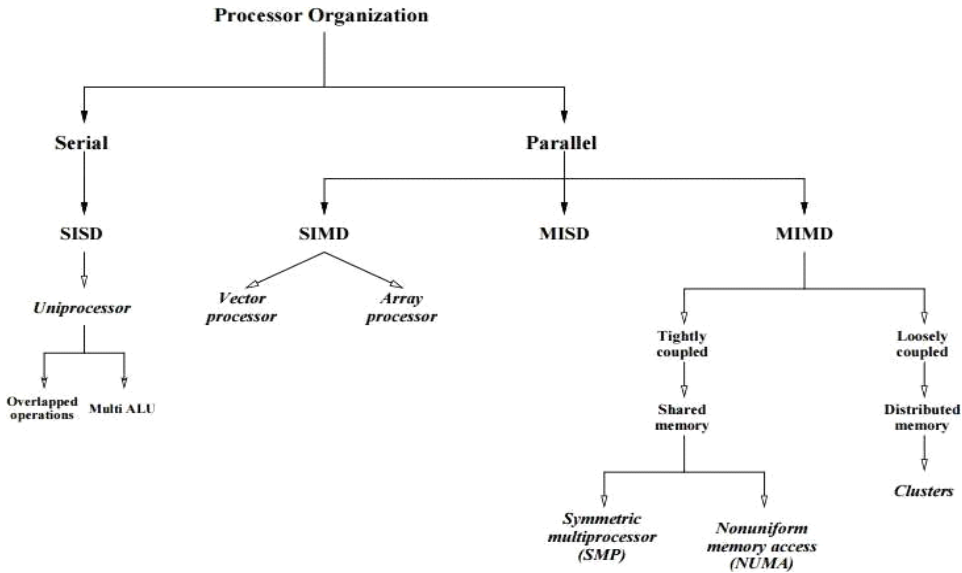


Figure 5.18: Processor Organization

Characteristics of Multiprocessors

Multiprocessing enhances system reliability by limiting the impact of a failure in one part on the entire system. Should one processor encounter an issue, another can take over its functions.

System performance improves significantly through parallel computation in two primary ways:

- Multiple independent tasks can run concurrently.
- A single task can be divided into multiple parallel sub-tasks.

This parallel execution can occur through explicit user declarations or via compilers equipped with multiprocessor software capable of detecting and leveraging parallelism by checking for data dependencies.

Coupling of Processors

1. Tightly Coupled System/Shared Memory

- Tasks and processors synchronize closely through a common shared memory.
- Shared memory systems lack protection of each processor's local memory (cache).

2. Loosely Coupled Systems/Distributed Memory

- Tasks or processors don't communicate synchronously.
- Communication occurs through messaging, including location, information content, and error checks.
- Higher data exchange overhead.
- Memory allocation varies between tightly and loosely coupled systems.

5.6. Interconnect Structure

A multitasking system comprises CPUs, I/O devices connected to inputs/outputs, and memory segments. The physical coupling between these components varies based on the number of transmission paths between processors and memory in a shared memory setup or the layers in a loosely connected system. Various physical techniques create communication links, such as:

- Time-shared common bus: Multiple processors linked to memory via a network.
- Multi-port memory.
- Vertical and horizontal switching.
- Multilayer switching networks.
- Hypercube systems.

The description explores the different options for creating communication links in multiprocessor systems without replicating previous content.

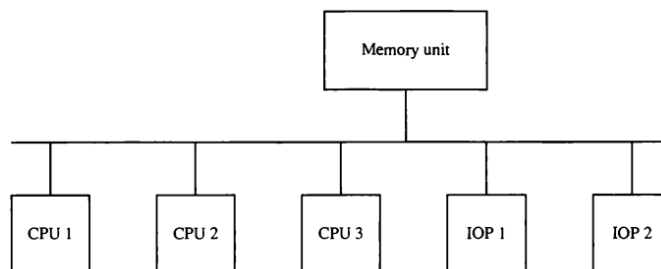


Figure 5.19: Time Shared Common Bus Organization

In a system where only one processor can communicate with memory or another processor at a time, the transfer operation relies on the controlling processor of the data path. If another processor aims to initiate a transfer, it must first confirm the data path's availability. Upon availability, the process can direct the target to commence the transfer. The receiver on the bus, upon receiving a command, determines its position and responds to the sender's signals to start transmission.

As all processes share a single data path, contention issues arise, necessitating resolution by the bus operator who establishes priority among applicants. A public bus supports only one operation at a time, leading to other processors either engaging in internal operations or remaining idle while waiting for the bus. This limitation constrains system changes to the speed of a single method. Implementing multiple independent buses allows for concurrent operations, yet escalates system cost and complexity.

5.7. MultiPort Memory

Multiport memory systems feature a distinct bus connecting each memory module and CPU, granting each processor bus access to every memory module. Each memory module contains several ports, each corresponding to a bus. These systems manage memory access by assigning fixed values to each memory location and establishing priority among processors based on their physical port connections to the memory modules.

While multiport memory offers enhanced flexibility due to multiple paths between processors and memory, it demands extensive memory management, numerous cables, and connectors. Therefore, this model is generally suitable for systems with fewer processors due to its higher cost and complexity.

This representation outlines the complexities and advantages of multiport memory systems without reproducing prior content.

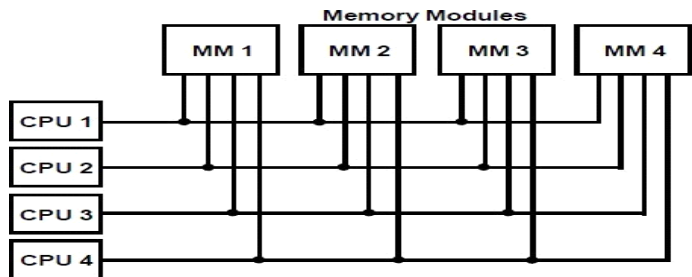


Figure 5.20: Multiport Memory

5.8. Crossbar Switch

The crossbar architecture comprises crosspoints positioned at the junction of the processor bus and memory modules. In Figure 13-4, the crossbar interconnects four CPUs with four memory modules. Each crosspoint houses a switch, represented by a small square, responsible for establishing the path between a processor and a memory module. These switch points include control logic that configures the transfer route between a processor and memory.

The control logic within each switch point analyses the address transmitted across the bus to ascertain whether its corresponding memory module is the intended destination. Additionally, it manages competing requests for access to the same memory module, often based on predefined priorities. This prioritization helps resolve multiple simultaneous access requests to a particular memory module.

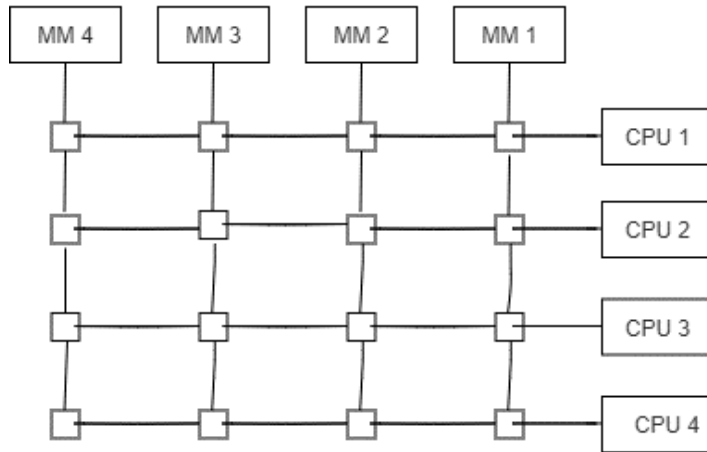


Figure 5.21: Crossbar Switch

Crossbar Switch Architecture

The Crossbar configuration facilitates concurrent transfers from memory modules by employing individual pathways for each module. However, implementing the switch hardware necessary for this setup can lead to increased complexity and a larger physical footprint.

Multistage Switching Networks

In a Multistage Network, the fundamental unit is the 2 X 2 interchange switch. Each switch possesses two inputs (labeled A and B) and two outputs (labeled 0 and 1). These switches are governed by control signals that coordinate the interaction between inputs and outputs. For instance, input A connects to output 0 while input B behaves similarly. They also resolve conflicts, ensuring that if both inputs A and B necessitate the same output terminal, only one connection is established.

By utilizing these 2 X 2 switches as foundational components, a multilayered network can be constructed to manage communications across various sources and destinations. This network configuration efficiently handles routing and communication between multiple points within the system.

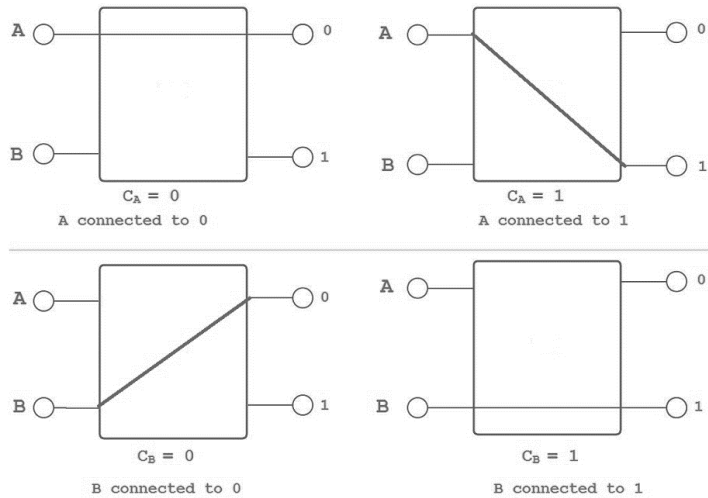


Figure 5.22: Multistage Switching Networks

Hypercube Interconnect

The hypercube or dual ncube architecture in multiprocessor systems involves an interconnected network comprising $N = 2^n$ processors arranged in an n -dimensional binary cube. Each processor constitutes a segment of this cube. Although the term "node" often refers to just a CPU, in reality, it encompasses not only the CPU but also local memory and I/O interfaces.

Each processor establishes direct communication channels with neighboring nodes, typically corresponding to the edges of the cube. With 2^n distinct n -bit binary addresses available for assignment to processors, every processor address differs from its neighboring nodes' addresses by only one bit position.

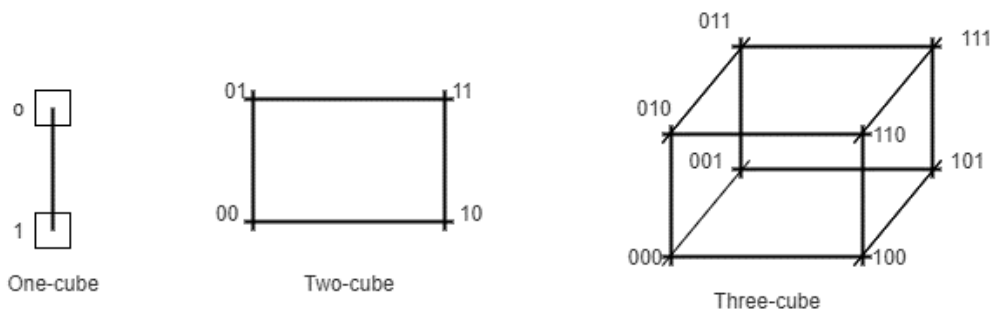


Figure 5.23: Hypercube Switch

Certainly, here's a rephrased version along with added subtopics:

Hypercube Models

Hypercube models are categorized based on their dimensions. For instance:

- For $n = 1$, a linear model involves two processors linked in a path.
 - In a 2-cube model with $n = 2$ ($2^n = 4$), four nodes interconnect forming a square.
 - The structure expands to a three-cube model, consisting of eight interconnected cubes.
- Each node in these models is allocated a binary address that differentiates it from its neighboring nodes.

Interprocessor Arbitration

Computer systems employ multiple data paths to enable data transfer among components. These paths include internal data buses within the CPU, the memory bus used for data and addressing, and the I/O bus facilitating communication between inputs and outputs.

Multi-System Bus

The system's central data path connecting major components like the CPU, IOPs, and memory is referred to as the bus. In a shared-memory multiprocessor system, processors request access to memory or other programs through this bus. However, if another processor is using the bus, the requesting process has to wait, leading to potential conflicts among processors vying for the bus.

Serial Arbitration Procedure

To address these conflicts, a serial arbitration process is implemented. This process operates by assigning priority to processor requests. Hardware solutions like bus switch technologies or serial key decoding techniques are employed. Priority control lines dictate which connected device gains access to the bus based on its position, granting the highest priority to the closest device to the main line. When multiple devices attempt bus access concurrently, the device with the highest priority is granted access.

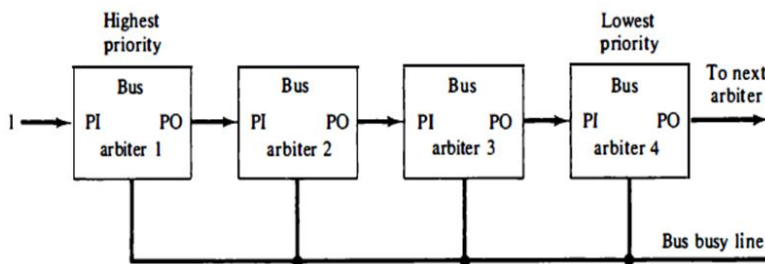


Figure 5.24: Serial or Daisy Chain Arbitration

Let's consider a scenario where each processor is equipped with its own bus arbiter logic featuring switch input and output lines. These arbiters are interlinked in a chain, where the priority output (Po) of each arbiter connects to the priority input (PI) of the subsequent arbiter. The PI signal with the highest priority remains at logical value 1, ensuring that the highest-priority processor gets access to the system bus when it requests it.

When a processor doesn't seek bus control ($PI = 1$) and the corresponding arbiter detects this, it sets its output Po to 1, passing this value to the next unit. Conversely, if a process requests bus control and the arbiter find that its input PI is 1, it sets the output Po to 0. This chain continues, ensuring that the processor whose arbiter has $PI = 1$ and $Po = 0$ gains control of the bus.

In this setup, when a higher-priority process requests the bus, it overtakes control, and a lower-priority processor must finish its bus operation before releasing control. The 'busy bus line,' which originates from each unit's open collector circuit, serves as a communication mechanism. When an arbiter intends to take control ($PI = 1$ and $Po = 0$), it checks this busy line. If inactive, indicating no other processors are using the bus, it proceeds to take control. However, if the busy line is active, it implies another processor is using the bus. The arbiter continues to monitor this busy line, allowing lower-priority processes that lose bus control to finish their operations. Once the bus returns to an idle state, the priority arbiter maintains the busy line, enabling relevant processes to make necessary bus modifications.

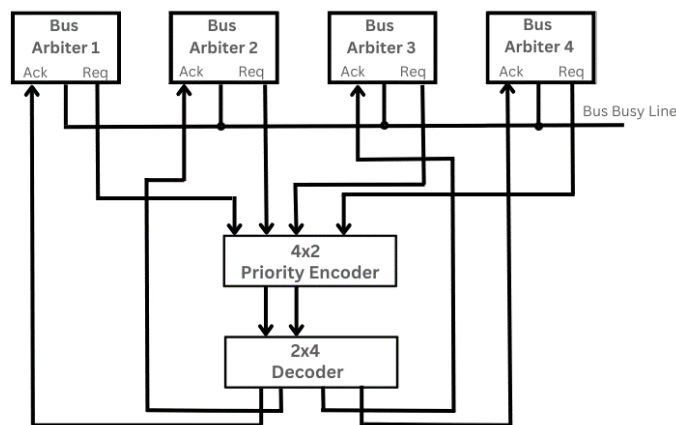


Figure 5.25: Parellel Arbitration

In the realm of parallel bus arbitration, each arbiter features both bus request and acknowledgment lines. Activating a request pipeline when a processor needs bus access, it checks for bus availability similar to the chained problem.

A 4 x 2 key encoder is employed in this parallel scheme, where the four referees' lines feed into it. This encoder generates a 2-bit code representing the most significant unit seeking bus access. The output code drives a 2-bit pipeline, thereby affecting the bus control decisions.

Let's explore various dynamic arbitration algorithms that operate using a dynamic priority system:

1. **Time Slicing:** Allocates fixed bus time to each process sequentially, irrespective of its bus location. All devices receive equal time for bus communication without any prioritization.
2. **Polling:** Utilizes a polling line to address connected devices. The bus controller polls devices based on predetermined rules. Programmable in nature, the polling system can adjust its selection criteria.
3. **Least Recently Used (LRU):** Prioritizes devices that haven't utilized the bus for the longest duration. It updates priority after several data cycles to ensure each device gets access.
4. **First-In, First-Out (FIFO):** Processes requests in the order they are received. Bus controller maintains a queue based on arrival time, allowing each processor to access the bus in turn.
5. **Rotating Daisy Chain:** A variant of the daisy chain algorithm, using a closed loop without a central control bus, linking main equipment and end equipment.

Interprocessor communication and synchronization are crucial in a multi-processor setup. It involves shared memory or separate memory channels for communication and synchronization between processors.

5.9. Cache Coherence

Ensuring consistency among shared data across multiple local caches is essential. Caches aim to reduce processor access time by storing frequently used data. Write strategies differ between updating both cache and main memory immediately (overwrite strategy) and updating the cache first with later synchronization to main memory (write-back strategy).

5.10. Operating System Organization

Operating systems in multiprocessor systems can use different structures: master-slave configuration, separate operating systems, or distributed operating systems. These structures dictate how tasks are assigned among processors.

5.11. Communication and Synchronization

Communication involves information exchange between different systems, while synchronization focuses on managing synchronized data between processors. Both are crucial for process management and shared data access.

In managing synchronization resources, multiprocessor systems employ various mechanisms. These mechanisms aim to coordinate data exchange and ensure efficient resource allocation among processes.

Short Answer Questions

1. What is pipelining?
2. What are the major characteristics of a pipeline?
3. What are the main issues in designing pipelined computers?
4. Write a short note on multiport memory system.
5. What is pipeline register delay?
6. What is synchronization?
7. What are the types of Pipeline hazards?
8. What is instruction level parallelism?
9. Draw and explain the time-shared bus.
10. What do you think are the reasons for the increasing importance of multi-processors?

Long Answer Questions

1. List the two models available for communication in multi-processing environment.
2. What are the challenges in parallel processing?
3. Explain the performance of the instruction pipeline can be improved.
4. Explain how pipelining helps to speed-up the processor. Discuss the hazards that have to be taken care of in a pipelined processor.
5. Explain in detail about types of pipeline
6. Describe in detail about Multiprocessor and its functions.
7. What do you understand by Cache coherence Problem? Give an example.
8. When can we say that the memory is coherent in a multi-processor system?
9. Explain the interprocessor communication using shared variables.
10. Explain the interprocessor communication using message passing.

Multiple Choice Questions

1. The Excess-3 decimal code is a self-complementing code because()
 - a) The binary sum of a code and its 9's complement is equal to 9
 - b) It is a weighted code
 - c) Complement can be generated by inverting each bit pattern
 - d) The binary sum of a code and its 10's complement is equal to 9
2. The equivalent hexadecimal notation for octal number 2550276 is ()
 - a) FADED
 - b) AEOBE
 - c) ADOBE
 - d) ACABE
3. A byte addressable computer has a memory capacity of 2m Kbytes and can perform 2^n operations. An instruction involving 3 operands and one operator needs a maximum of
 - a) 3m bits
 - b) $m + n$ bits
 - c) $3m + n$ bits
 - d) $3m + n + 30$ bits
4. One of the main features that distinguish microprocessor from micro-computers is
 - a) Words are usually larger in microprocessors
 - b) Words are shorter in microprocessors
 - c) Microprocessor does not contain I/O devices
 - d) None of the above
5. Consider a code with five valid code words of length ten code is 0000000000, 0000011111, 1111100000, 1110000011, 1111111111 Hamming distance of the
 - a) 5
 - b) 10
 - c) 8
 - d) 9
6. The IEEE-754 double precision format to represent floating point numbers, has a length of bits.
 - a) 16
 - b) 32
 - c) 48

- d) 64
7. The Simplified form of a Boolean equation for the following truth table is ()
- $F = yz' + y'z$
 - $F = xy' + x'y$
 - $F = x'z + xz'$
 - $F = X'z + xz' + xyz$
8. The Simplified form of a Boolean equation $(AB' + AB'C + Ac) (A'C' + B')$ is ()
- AB'
 - $AB'C$
 - $A'B$
 - ABC
9. Assume a processor takes 3 cycles to execute any instruction (fetch, decode, execute) How many cycles would a non-pipelined processor take to execute 6 instructions?
- 12 cycles
 - 18 cycles
 - 3 cycles
 - 6 cycles
10. Assume a processor takes 3 cycles to execute any instruction (fetch, decode, execute) How many cycles would a pipelined processor take to execute 6 instructions?
- 12 cycles
 - 18 cycles
 - 3 cycles
 - cycles
11. Pipeline implements
- Fetch instruction
 - Decode instruction
 - Fetch operand
 - Calculate operand
12. Where does a computer add and compare data?()
- Hard disk
 - Floppy disk
 - CPU chip
 - Memory chip

13. Pipelining strategy is called implement ()
- a) Instruction execution
 - b) Instruction prefetch
 - c) Instruction decoding
 - d) Instruction manipulation
14. In single processor systems, they only have _____processor. ()
- a) Zero
 - b) One
 - c) Two
 - d) Multiple
15. How many types of multiprocessors? ()
- a) 2
 - b) 3
 - c) 4
 - d) 5
16. Which multiprocessor system contains a master slave relationship? ()
- a) Symmetric Multiprocessors
 - b) Singleton Multiprocessors
 - c) Asymmetric Multiprocessors
 - d) Both A and B
17. Which of the following is Disadvantages of Multiprocessor Systems? ()
- a) Multiprocessor systems is quite expensive.
 - b) All the processors in the multiprocessor system share the memory. So, amuch large pool of memory is required as compared to single processor systems.
 - c) more complex and complicated operating system is required in multiprocessor systems
 - d) All of the above.
18. The feature of the multi-microprocessor architecture is ()
- a) Task dependent
 - b) Single bus provider for many processors
 - c) Design is for a specific task
 - d) All of the above
19. The main objective in building the multi-microprocessor is
- a) Greater throughput

- b) Enhanced fault tolerance
 - c) Both A and B
 - d) None of the above
20. A multiprocessor operating system must take care of ()
- a) Authorized data access and data protection
 - b) Unauthorized data access and data protection
 - c) Authorized data access
 - d) Data protection
21. A common synchronization mechanism used in multiprocessor operating systems is
- a) Complex
 - b) Lockstep
 - c) Clock
 - d) Locks
22. In a multiprocessor system, if one processor fails, the system will halt ()
- a) True
 - b) False
 - c) Can be true or false
 - d) cannot say
23. An example of the symmetric multiprocessing system is the Encore version of Unix for the Multimax Computer.
- a) True
 - b) False
 - c) Can be true or false
 - d) Can not say
24. Memory management on a multiprocessor must deal with all of the following ()
- a) Uniprocessor Computer
 - b) Computer
 - c) Processor
 - d) System
25. In a multi-processor configuration two coprocessors are connected to host 8086 processor. The instruction sets of the two coprocessors ()
- a) Must be same
 - b) May overlap
 - c) Must be disjoint

- d) Must be the same as that of host
26. The feature of multi-microprocessor architecture is()
- a) Task dependent
 - b) Single bus provider for many processors
 - c) Design is for a specific task
 - d) All of the mentioned
27. The main objective in building the multi-microprocessor is()
- a) Greater throughput
 - b) Enhanced fault tolerance
 - c) Greater throughput and enhanced fault tolerance
 - d) None of the mentioned
28. An interface between the user or an application program, and the systemresources is
- a) Microprocessor
 - b) Microcontroller
 - c) Multi-microprocessor
 - d) Operating system
29. A multiprocessor operating system should perform()
- a) A mechanism to split a task into concurrent subtasks
 - b) Optimise the system performance
 - c) Handling structural or architectural changes
 - d) All of the mentioned
30. Zero address instruction format is used for()
- a) RISC architecture
 - b) CISC architecture
 - c) Von-Neuman architecture
 - d) Stack-organized architecture
31. If no node having a copy of a cache block, this technique is known as ()
- a) Uniform memory access
 - b) Cached
 - c) Un-cached
 - d) Commit
32. A processor that continuously tries to acquire the locks, spinning around a loop till it reaches () its success, is known as
- a) Spin locks

- b) Store locks
 - c) Link locks
 - d) Store operational
33. The straight-forward model used for the memory consistency, is called ()
- a) Sequential consistency
 - b) Random consistency
 - c) Remote node
 - d) Host node
34. _____ have been developed specifically for pipelined systems. ()
- a) Utility software
 - b) Speed up utilities
 - c) Optimizing compilers
 - d) None of the mentioned
35. The pipelining process is also called as _____ ()
- a) Superscalar operation
 - b) Assembly line operation
 - c) Von Neumann cycles
 - d) None of the mentioned
36. The fetch and execution cycles are interleaved with the help of ()
- a) Modification in processor architecture
 - b) Clock
 - c) Special unit
 - d) Control unit
37. Each stage in pipelining should be completed within _____ cycle. ()
- a) 1
 - b) 2
 - c) 3
 - d) 4
38. If a unit completes its task before the allotted time period, then ()
- a) It'll perform some other task in the remaining time
 - b) Its time gets reallocated to a different task
 - c) It'll remain idle for the remaining time
 - d) None of the mentioned

39. To increase the speed of memory access in pipelining, we make use of
- a) Special memory locations
 - b) Special purpose registers
 - c) Cache
 - d) Buffers
40. The periods of time when the unit is idle is called as ()
- a) Stalls
 - b) Bubbles
 - c) Hazards
 - d) Both Stalls and Bubbles
41. The contention for the usage of a hardware device is called
- a) Structural hazard
 - b) Stalk
 - c) Deadlock
 - d) None of the mentioned
42. The situation wherein the data of operands are not available is called ()
- a) Data hazard
 - b) Stock
 - c) Deadlock
 - d) Structural hazard
43. Arithmetic Pipeline is used for?
- a) Floating point operations
 - b) Integer operations
 - c) Character operations
 - d) None of the above
44. Which of the following is not a Pipeline Conflicts? ()
- a) Timing Variations
 - b) Branching
 - c) Load Balancing
 - d) Data Dependency
45. Which of the following is an advantage of pipelining? ()
- a) Instruction throughput increases.
 - b) Faster ALU can be designed when pipelining is used.
 - c) Pipelining increases the overall performance of the CPU.

- d) All of the above
46. Memory Management on a Multiprocessor must deal with all of found on ()
- a) Uniprocessor computer
 - b) Computer
 - c) Processor
 - d) System
47. The feature of multi-microprocessor architecture is ()
- a) Task dependent
 - b) Single Bus provider for many processors
 - c) Design is for a specific task
 - d) All of the above
48. Which of the following memory unit communicates directly with the CPU? ()
- a) Auxiliary memory
 - b) Main memory
 - c) Secondary memory
 - d) None of the above
49. In which addressing mode the operand is given explicitly in the instruction ()
- a) Absolute
 - b) Immediate
 - c) Indirect
 - d) Direct
50. When necessary, the results are transferred from the CPU to the main memory by
- a) I/O devices
 - b) CPU
 - c) Shift registers
 - d) None of these

Keys

1. (c) 2. (c) 3. (d) 4. (c) 5. (a) 6. (d) 7. (c) 8. (a) 9. (c) 10. (a) 11. (d) 12. (b) 13. (b) 14. (a) 15. (a)
 16. (c) 17. (d) 18. (d) 19. (C) 20. (b) 21. (d) 22. (a) 23. (a) 24. (a) 25. (c) 26. (b) 27. (d) 28. (d)
 29. (a) 30. (a) 31. (d) 32. (c) 33. (c) 34. (b) 35. (a) 36. (d) 37. (c) 38. (b) 39. (b) 40. (a) 41. (b)
 42. (c) 43. (a) 44. (d) 45. (d) 46. (a) 47. (a) 48. (b) 49. (b) 50. (b)

