Course

Name of the subject : **Computer Architecture**
Name of the Course Coordinator & Faculty Member : **K.Sudha Rani & A.Haritha**
Academic Year : **2026-27**
Year & Sem : **III & I Sem**

TKR COLLEGE OF ENGINEERING & TECHNOLOGY
TKRCET
HYDERABAD
Indian in Character ✪ International in Excellence

Department of Electronics & Communication Engineering
TKR College of Engineering & Technology
Meerpet, Balapur, Saroornagar, Hyderabad - 500097, Telangana.
Accredited by NBA & NAAC A+

.

DEPARTMENT VISION& MISSION

VISION

The department endeavors to transform itself into a centre of academic excellence in the field of Electronics & Communication Engineering thereby producing technically competent, confident, socially and ethically responsible engineers.

MISSION

- Providing an environment that nurtures creativity that would build strong foundation for higher education and career advancement.
- Conducting interactive events that would bridge the gap between academics and industry which would make the students technically competent.
- Equipping students with essential employability skills like teamwork, problem solving and time management. While imparting ethical responsibility to make them self-reliant & globally recognized.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO 1: Graduates will master Electronics and Communication Engineering with ethics, crafting innovative solutions for industry challenges.

PEO 2: Graduates will continually improve technical skills, contributing to multidisciplinary interactions and paving the way for professional development.

PEO 3: Graduates through exemplary education, will adeptly communicate, collaborate in teams to address societal needs.

PROGRAMME SPECIFIC OUTCOMES (PSOs)

PSO1

Ability to apply the acquired knowledge of Electronics and Communication engineering in design and development, in areas of VLSI and Image Processing.

PSO2

Analyze and solve the complex Electronics and Communication Engineering problems using state of art hardware and software tools.

Course Syllabus

Regulation: R22
Course Name: Computer Architecture (D5OE12)

Course Objective: Understand the detailed computer architecture and organization, hardware operation of digital computer.

Course Outcomes:

After learning the contents of this course, the student will be able to

1. Make use of the concepts of Computer Organization. L3
2. Design the hardwired and micro-programmed control units and demonstrate 8086 architecture L4
3. Analyze the computer arithmetic operations and write 8086 basic ALP programs L4
4. Analyze I/O data transfer modes and memory hierarchy. L4
5. Analyze the concurrent processing L4

Unit – I:

Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.

Basic Computer Organization and Design: Instruction codes, Computer Registers, Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt, Complete Computer Description.

Unit – II:

Central Processing Unit: Processor Organization, Register Organization, Instruction cycle, hardwired control unit, Micro program control unit. The 8086 Processor Architecture, Register organization, Physical memory organization, General Bus Operation, I/O Addressing Capability, Special Processor Activities, Minimum and Maximum mode system and timings.

Unit – III:

Computer Arithmetic

Introduction, The arithmetic logic unit, Integer representation, Integer arithmetic, Floating point representation, Floating point arithmetic, Addition and Subtraction, Multiplication Algorithms, Division Algorithms, Floating - point Arithmetic operations. Data Transfer and Manipulation.

Unit – IV:

Input-Output Organization: Peripheral Devices, Input Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt, Direct memory Access, Input–Output Processor (IOP), Intel 8089 IOP.

Memory Organization: Memory Hierarchy, Auxiliary memory, Associate Memory, Cache Memory, Virtual Memory, Memory Management Hardware.

Unit – V:

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, CISC, RISC versus CISC, Vector Processing, Array Processors.

Multi Processors: Characteristics of Multi processors, Inter connection Structures, Inter processor arbitration, Inter processor communication, and synchronization**.**

TEXT BOOKS:

1. Computer System Architecture, M.Moris Mano, Third Edition, Pearson.

2. Advanced Micro processors and Peripherals, KM Bhurchandi, A.K Ray, 3rdedition, McGraw Hill India Education Private Ltd**.**

REFERENCE BOOKS:

1. Microprocessors and Interfacing, DV Hall, SSSP Rao, 3rdedition, McGraw Hill India Education Private Ltd.
2. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill, 2002.
3. Computer Organization and Architecture, William Stallings, 9th Edition, Pearson.
4. David A. Patterson, John L. Hennessy: Computer Organization and Design–The Hardware /Software Interface ARM Edition,4th Edition

UNIT – I : DIGITAL COMPUTERS & BASIC COMPUTER ORGANIZATION AND DESIGN

Introduction to Digital Computers

1.1 What is a Digital Computer?

A digital computer is an electronic device that processes binary data (0s and 1s) to perform arithmetic, logical, and control operations. It accepts input, processes it according to stored instructions (program), and produces output.

Key Characteristics of a Digital Computer:

- Speed: Performs millions/billions of operations per second (MIPS / GFLOPS)
- Accuracy: Results are precise and reliable for arithmetic and logical operations
- Storage: Can store large volumes of data in primary and secondary memory
- Automation: Once programmed, operates without human intervention
- Versatility: Can perform diverse tasks – from word processing to scientific simulations
- Diligence: Does not fatigue; performs repeated tasks with consistent accuracy

Types of Digital Computers (by size and capacity):

- Microcomputers (PCs, laptops) – Personal use; single user
- Minicomputers – Multi-user systems; used in small organizations
- Mainframes – Large organizations (banks, airlines); handle millions of transactions
- Supercomputers – Scientific research, weather forecasting; highest performance

Block Diagram of a Digital Computer

A digital computer consists of five major functional units:

- **Input Unit**
  - Accepts data and instructions from external sources (keyboard, mouse, scanner)
  - Converts them to binary form understandable by the CPU
- **Central Processing Unit (CPU)**
  - The 'brain' of the computer – responsible for all processing
  - Consists of: ALU (Arithmetic & Logic Unit) + CU (Control Unit) + Registers
- **Memory Unit**
  - Primary Memory (RAM, ROM) – directly accessible by CPU; volatile/non-volatile
  - Secondary Memory (HDD, SSD, CD) – permanent storage; not directly accessed by CPU
- **Output Unit**
  - Converts processed binary data back to human-readable form
  - Examples: Monitor, Printer, Speaker
- **System Bus**
  - Communication pathway connecting all units
  - Three types: Data Bus, Address Bus, Control Bus

Block Diagram Description:

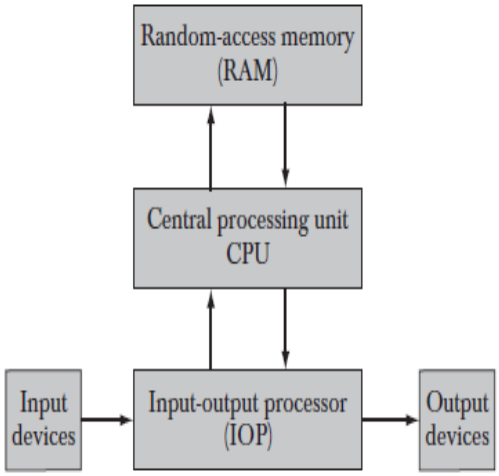


Figure 1-1 Block diagram of a digital computer.

Component	Function
Input Unit	Transfers data & instructions to CPU/Memory
ALU	Performs arithmetic (+,−,×,÷) and logical (AND, OR, NOT) operations
Control Unit	Directs operations of all units; fetches, decodes & executes instructions
Registers	High-speed temporary storage within CPU (e.g., PC, IR, AC, MAR, MDR)
Memory Unit	Stores data and instructions during processing
Output Unit	Presents results to the user
System Bus	Transfers data, addresses, and control signals among units

Operation select					Operation	Function
S_3	S_2	S_1	S_0	C_{in}		
0	0	0	0	0	$F = A + B$	Add
0	0	0	0	1	$F = A + B + 1$	Add with carry
0	0	0	1	0	$F = A + \overline{B}$	Subtract with borrow
0	0	0	1	1	$F = A + \overline{B} + 1$	Subtract
0	0	1	0	0	$F = A$	Transfer A
0	0	1	0	1	$F = A + 1$	Increment A
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F =$ shr A	Shift right A into F
1	1	×	×	×	$F =$ shl A	Shift left A into F

S1	S0	y
0	0	B
0	1	B'
1	0	0
1	1	1

D=A+Y+Cin

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

Computer Organization, Computer Design & Computer Architecture

Computer Organisation

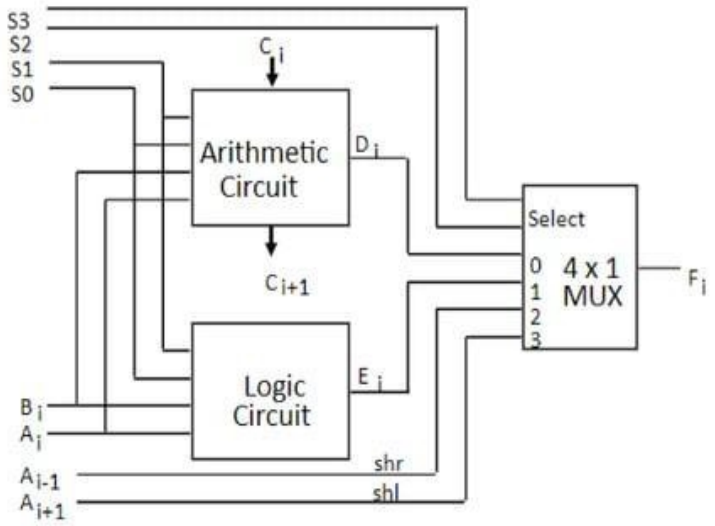
Computer Organization refers to the operational units and their interconnections that realize the architectural specifications. It deals with HOW hardware components work and interact.

Concerns of Computer Organization:

- Control signals and their timing
- Memory technology and interfaces
- Hardware mechanisms for instruction execution
- Input/Output mechanisms and data transfer
- CPU design – ALU, registers, data paths

One stage of ALSU

The arithmetic, logic, and shift circuits can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Figure.



Computer Design

Computer Design is the process of determining the hardware structure needed to implement the required architecture. It involves translating architectural specifications into actual hardware (gates, flip-flops, circuits).

Aspects of Computer Design:

- Logic design of functional units (ALU, control unit)
- Register transfer logic and data path design
- Selection of implementation technology (TTL, CMOS, VLSI)
- Timing and synchronization of hardware components

Computer Architecture

Computer Architecture refers to the programmer-visible attributes of the computer system — the conceptual structure and functional behavior. It answers WHAT the hardware is capable of doing.

Concerns of Computer Architecture:

- Instruction Set Architecture (ISA) – set of machine instructions

- Data representation – number formats, data types
- Addressing modes – how operands are specified in instructions
- Registers visible to programmer – AC, PC, SP, etc.
- I/O mechanisms – how programs interact with peripherals

Key Differences – Organisation vs Architecture:

Computer Architecture	**Computer Organization**
What the hardware does (ISA, instruction set)	How the hardware does it (implementation)
Visible to programmer / software	Transparent to programmer
Example: 'Multiply instruction exists'	Example: 'Is multiplier hardware present or software simulated?'
Defined at design specification level	Defined at hardware implementation level

Note: Two computers can have the same architecture (e.g., Intel x86 ISA) but different organizations (different pipeline depths, cache sizes, etc.).

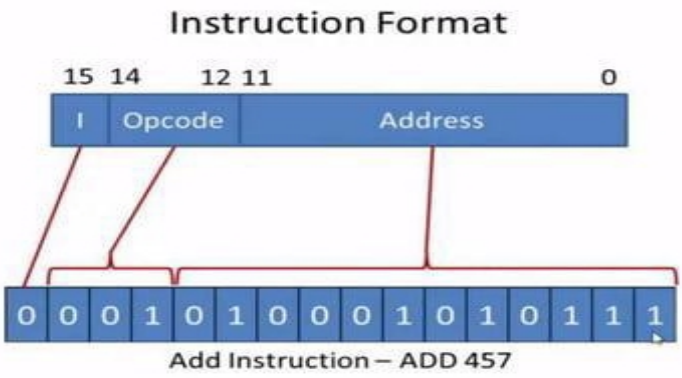
Basic Computer Organization and Design

Instruction Codes

An instruction code is a group of bits that tells the computer to perform a specific operation. Each instruction consists of:

- **Operation Code (Opcode): Specifies the operation to be performed (ADD, SUB, LOAD, STORE, etc.)**
- Address / Operand: Specifies the memory address or register containing the operand

Instruction Format – Basic Computer (16-bit):



- I (Bit 15): Mode bit – 0 = Direct Addressing, 1 = Indirect Addressing
- Opcode (bits 14–12): 3-bit operation code → supports up to 8 operations

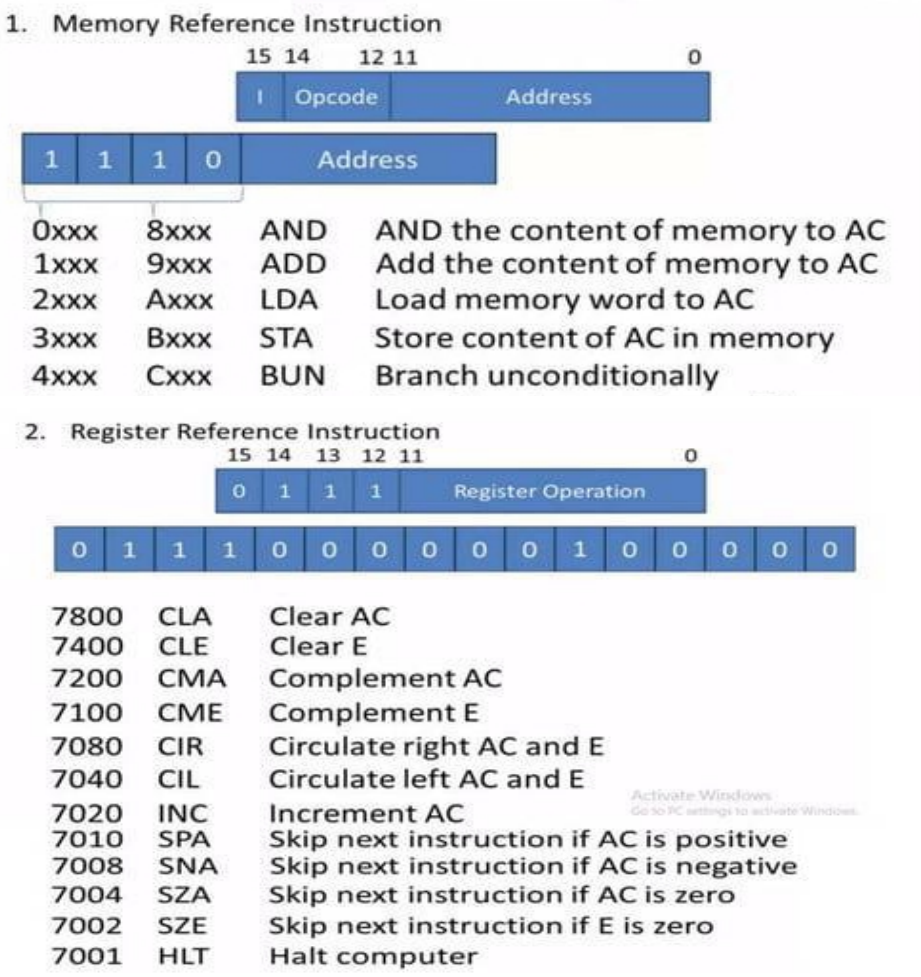
- Address field (bits 11–0): 12-bit address → can address 4096 (4K) memory locations

Instruction Type	Description & Examples
Memory Reference Instructions	Operate on memory locations – AND, ADD, LDA, STA, BUN, BSA, ISZ
Register Reference Instructions	Operate on CPU registers – CLA, CLE, INC, SPA, SNA, SZA, SZE, HLT, CME, CIR, CIL
Input/Output Instructions	Control I/O operations – INP, OUT, SKI, SKO, ION, IOF

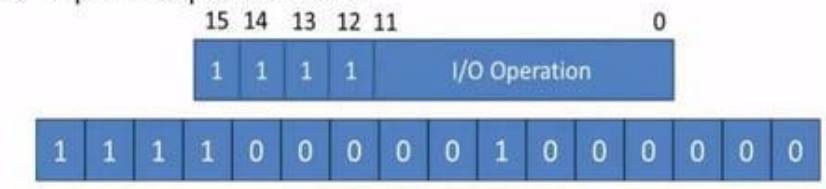
When opcode = 111 (Memory Reference):

- Remaining bits extend the instruction set further (register reference or I/O instructions)
- Bit 15 = 0 and opcode = 111: Register Reference Instructions
- Bit 15 = 1 and opcode = 111: I/O Instructions

Instruction Types in Basic Computer:

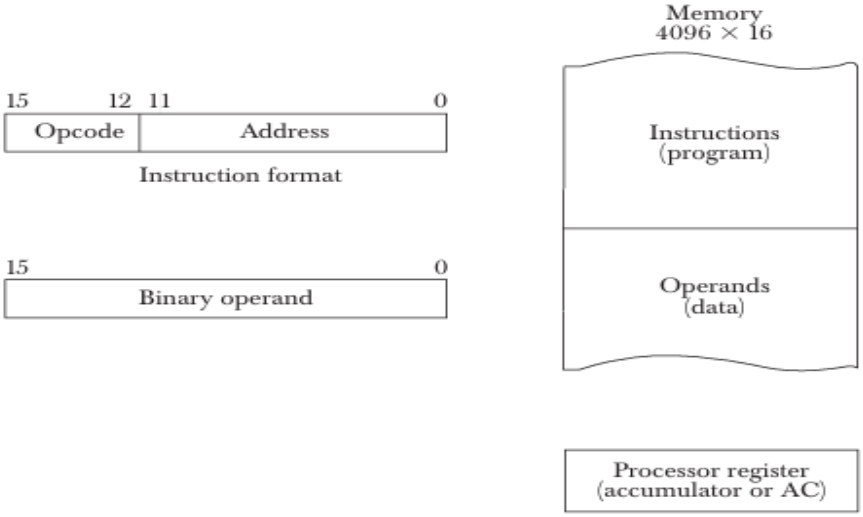


3. Input – Output Instruction



F800	INP	Input character to AC
F400	OUT	Output character from AC
F200	SKI	Skip on input flag
F100	SKO	Skip on output flag
F080	ION	Interrupt on
F040	IOF	Interrupt off

Stored Program Organization: The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

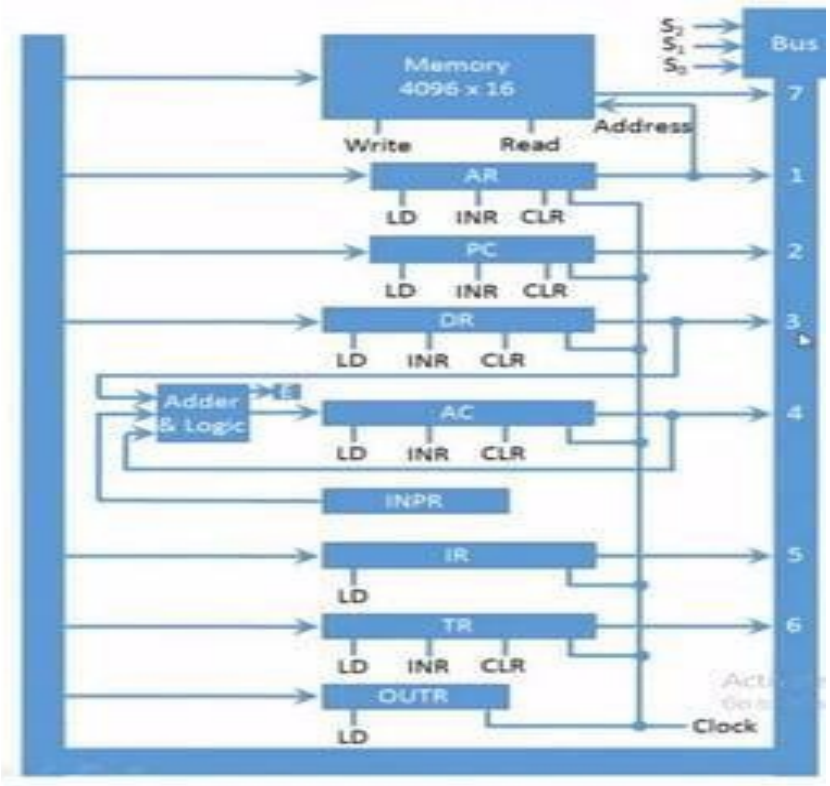


Computer Registers

Registers are high-speed storage elements within the CPU. The Basic Computer uses 8 registers:

Register	Purpose
DR – Data Register (16 bits)	Holds a memory word during read/write operations
AR – Address Register (12 bits)	Holds the memory address for read/write
AC – Accumulator (16 bits)	General-purpose register for arithmetic/logic results
IR – Instruction Register (16 bits)	Holds the current instruction being executed

PC – Program Counter (12 bits)	Holds address of the next instruction to fetch
TR – Temporary Register (16 bits)	Holds data temporarily during CPU operations
INPR – Input Register (8 bits)	Receives character from input device
OUTR – Output Register (8 bits)	Holds character to be sent to output device
SC – Sequence Counter (4 bits)	Counts timing states (T0–T15) of CPU
E – Extended AC (1 bit)	Carry/overflow bit from 16-bit AC operations



Common Bus System:

All registers in the Basic Computer are connected via a common 16-bit bus. Selection of which register drives the bus is controlled by a 3-bit selection code (S2, S1, S0):

Bus Selection (S2 S1 S0)	**Source Register**
000 (0)	None selected (bus carries no data)
001 (1)	AR placed on bus
010 (2)	PC placed on bus
011 (3)	DR placed on bus
100 (4)	AC placed on bus

101 (5)	IR placed on bus
110 (6)	TR placed on bus
111 (7)	Memory output (MBR) placed on bus

Computer Instructions

Memory Reference Instructions (Opcode ≠ 111):

Instruction	Operation
AND	AC ← AC ∧ M[EA] (Bitwise AND with memory)
ADD	AC ← AC + M[EA], E set on carry
LDA	AC ← M[EA] (Load AC from memory)
STA	M[EA] ← AC (Store AC to memory)
BUN	PC ← EA (Branch unconditionally)
BSA	M[EA] ← PC, PC ← EA+1 (Branch & Save Return address)
ISZ	M[EA] ← M[EA]+1; Skip next instr. if zero (Increment & Skip if Zero)

Register Reference Instructions (Opcode = 111, I = 0):

- CLA – Clear Accumulator: AC ← 0
- CLE – Clear E (carry) bit: E ← 0
- CMA – Complement Accumulator: AC ← $\overline{AC}$
- CME – Complement E: E ← $\bar{E}$
- CIR – Circulate Right: Shift AC and E right by 1 bit
- CIL – Circulate Left: Shift AC and E left by 1 bit
- INC – Increment Accumulator: AC ← AC + 1
- SPA – Skip if AC is Positive (AC > 0)
- SNA – Skip if AC is Negative (bit 15 = 1)
- SZA – Skip if AC is Zero
- SZE – Skip if E is Zero
- HLT – Halt the computer

Input/Output Instructions (Opcode = 111, I = 1):

- INP – Input: AC(0–7) ← INPR; FGI ← 0
- OUT – Output: OUTR ← AC(0–7); FGO ← 0
- SKI – Skip if input flag (FGI = 1)
- SKO – Skip if output flag (FGO = 1)
- ION – Enable Interrupt: IEN ← 1
- IOF – Disable Interrupt: IEN ← 0

Timing and Control

The Control Unit generates timing signals and control signals to coordinate all CPU operations. It uses a Sequence Counter (SC) that counts from T0 to T15 and resets.

Two Types of Control Unit Design:

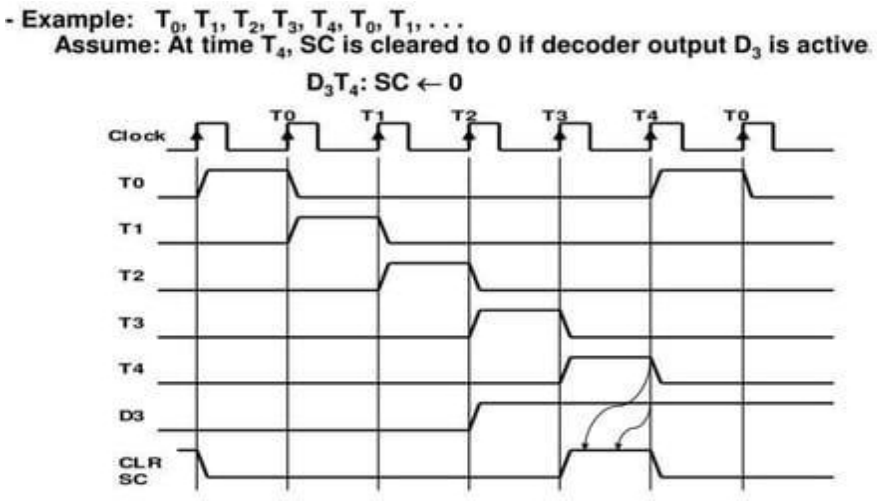
- Hardwired Control: Logic is implemented using combinational circuits. Fast but inflexible. Used in RISC processors.
- Microprogrammed Control: Control signals stored in a control memory (ROM). Easier to modify; used in CISC processors (e.g., Intel x86).

Timing Signal Generation:

- Clock pulses drive a 4-bit Sequence Counter (SC)
- SC generates timing signals T0, T1, T2, T3, T4, … T15
- Instruction bits (I, IR[14:12]) decoded by a 3×8 decoder
- Timing × Decoded signals → specific micro-operations

Control Unit Functions:

- Fetch the instruction from memory (PC → AR → Memory → IR)
- Decode the instruction (Opcode → Control signals)
- Determine the effective address (direct or indirect)
- Execute the instruction (activate ALU, memory, I/O)
- Handle interrupts (check IEN and R flags)



Note: In the Basic Computer, each clock cycle corresponds to one micro-operation (register transfer). Multiple clock cycles are required to complete one machine instruction.

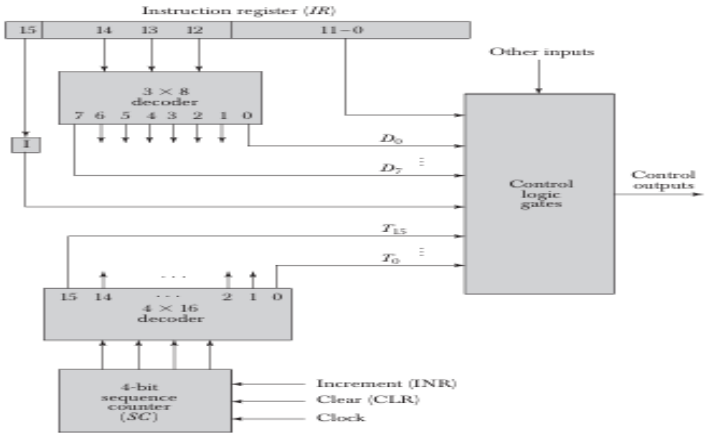


Figure 5-6 Control unit of basic computer.

Instruction Cycle

The instruction cycle (also called machine cycle or fetch-decode-execute cycle) is the basic operation cycle of the CPU. Each instruction goes through the following phases:

Phase 1 – Fetch Cycle (T0, T1):

T0: AR ← PC (Copy program counter to address register)

T1: IR ← M[AR], PC ← PC + 1 (Fetch instruction from memory; increment PC)

Phase 2 – Decode Cycle (T2):

- T2: Decode IR(14–12); AR ← IR(11–0), I ← IR(15)
  The decoder identifies the instruction type: Memory Ref / Register Ref / I/O

Phase 3 – Indirect Cycle (if I = 1):

- T3: AR ← M[AR] (Replace AR with effective address from memory)
  This adds one extra memory read cycle for indirect addressing

Phase 4 – Execute Cycle (T3 or T4 onwards):

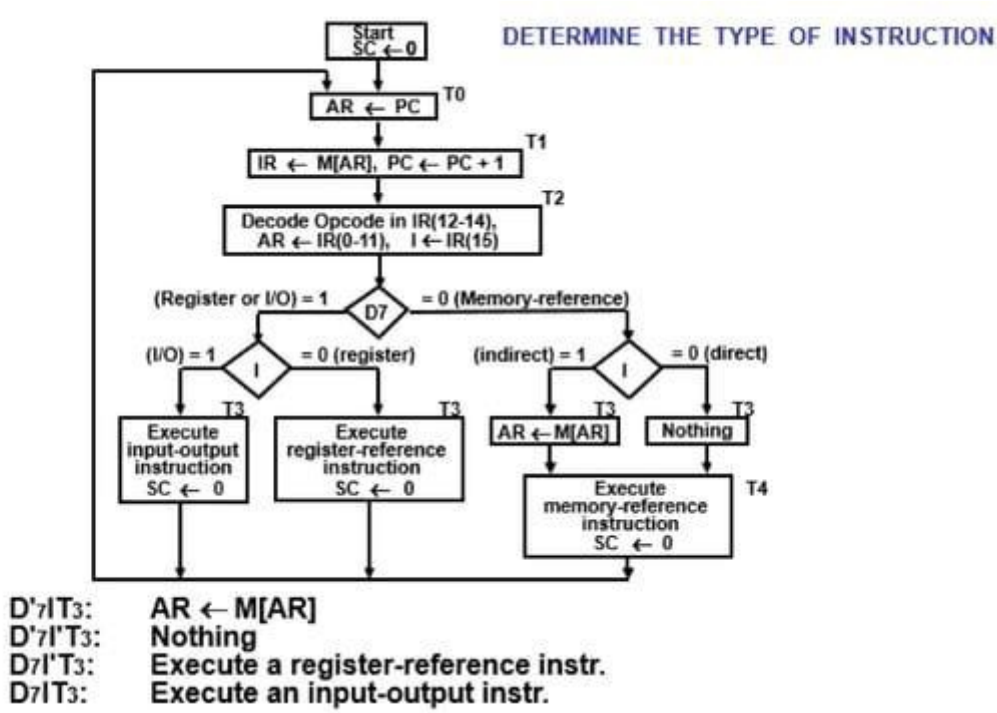
- Perform the actual operation – ADD, LDA, STA, AND, etc.
  Different instructions require different numbers of clock cycles

Phase 5 – Interrupt Cycle (if IEN = 1 and R = 1):

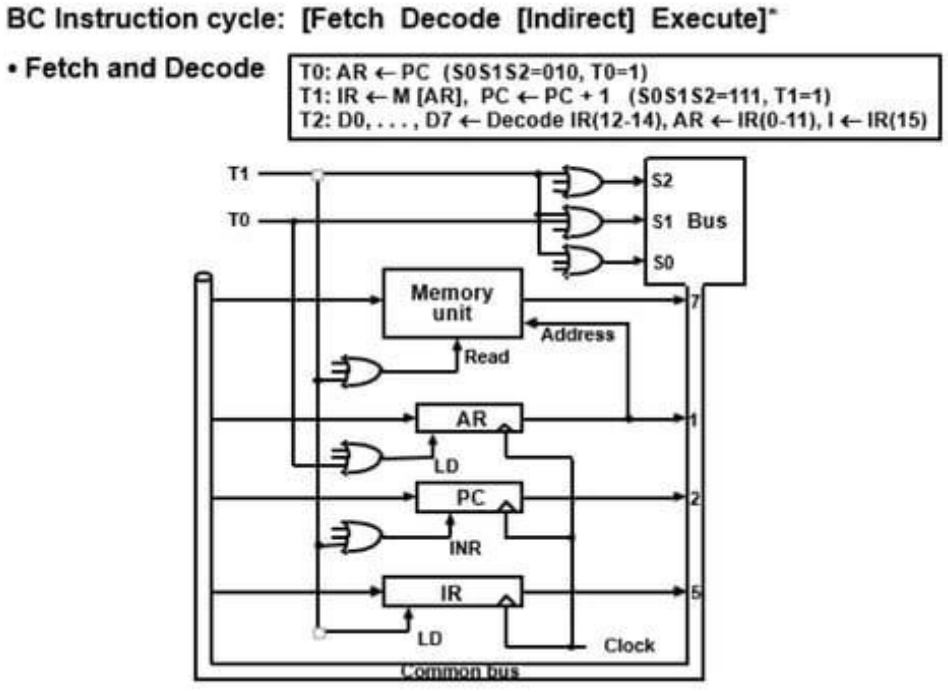
- Save current PC to M[0], branch to interrupt service routine
  R ← 0, IEN ← 0 (Disable further interrupts during service)

Instruction Cycle Flow Chart:

-



Determine the Type of Instruction: The timing signal after decoding is T_3. During the time T_3 the control signal determine the type of instruction that was just read from memory



Memory Reference Instructions – Execution Details

Each memory reference instruction executes in specific timing steps. Key examples:

AND Instruction (AC ← AC ∧ M[EA]):

- D0 T4: DR ← M[AR] (Fetch operand from effective address)
- D0 T5: AC ← AC ∧ DR, SC ← 0 (AND with AC, reset sequence counter)

ADD Instruction (AC ← AC + M[EA]):

- D1 T4: DR ← M[AR] (Fetch operand)
- D1 T5: AC ← AC + DR, E ← Carry, SC ← 0 (Add and store result; save carry in E)

LDA Instruction (AC ← M[EA]):

- D2 T4: DR ← M[AR] (Load operand from memory)
- D2 T5: AC ← DR, SC ← 0 (Transfer to AC)

STA Instruction (M[EA] ← AC):

- D3 T4: M[AR] ← AC, SC ← 0 (Store AC content to memory)

BUN Instruction (PC ← EA):

- D4 T4: PC ← AR, SC ← 0 (Branch to effective address)

BSA Instruction – Branch & Save Return (Subroutine Call):

- D5 T4: M[AR] ← PC, AR ← AR + 1 (Save return address, advance AR)
- D5 T5: PC ← AR, SC ← 0 (Jump to subroutine start)

ISZ Instruction – Increment and Skip if Zero:

- D6 T4: DR ← M[AR] (Fetch the memory operand)
- D6 T5: DR ← DR + 1 (Increment operand)
- D6 T6: M[AR] ← DR; If DR = 0, PC ← PC + 1; SC ← 0 (Write back; skip next if zero)

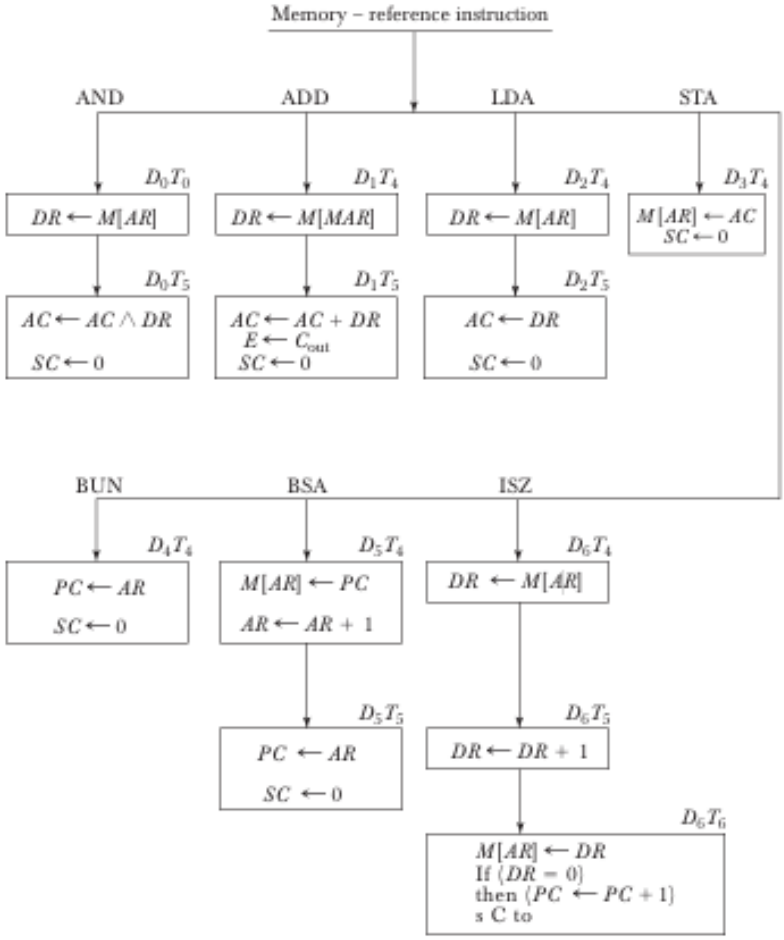


Figure 5-11 Flowchart for memory-reference instructions.

Input / Output and Interrupt

Input/Output Organization

The Basic Computer interacts with I/O devices through two 8-bit registers and two flag bits:

Register/Flag	Description
INPR (Input Register)	8-bit register receiving serial data from input device (e.g., keyboard)
OUTR (Output Register)	8-bit register transmitting data to output device (e.g., terminal)
FGI (Input Flag)	1-bit flag; set to 1 when INPR has new data; cleared after CPU reads it
FGO (Output Flag)	1-bit flag; set to 1 when device is ready; cleared when CPU writes OUTR

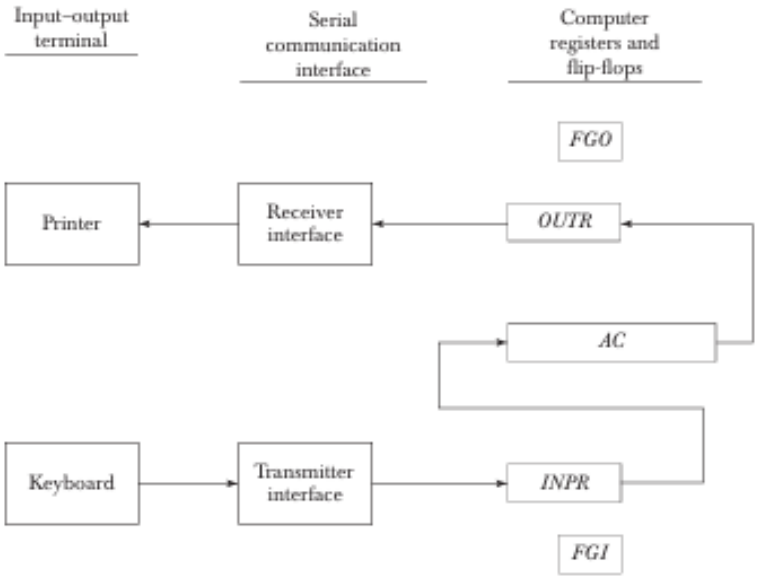
Programmed I/O (Busy-Wait / Polling):

- CPU continuously checks FGI / FGO flags in a loop
- Wasteful of CPU time – CPU cannot do other work while waiting
- Simple to implement; used when I/O is rare or fast

I/O Loop Example (Polling):

- CLA – Clear AC
- SKI – Skip if FGI = 1 (input ready)
- BUN SKI loop – Branch back if not ready
- INP – Read from INPR to AC
- 88OUT – Write AC to OUTR

Figure 5-12 Input–output configuration.



Interrupt-Driven I/O

To eliminate busy-waiting, the Basic Computer supports interrupt-driven I/O via an Interrupt Enable (IEN) flag and an Interrupt Request (R) flip-flop.

Interrupt Mechanism Steps:

- Step 1 – ION instruction sets IEN = 1 (enable interrupts)
- Step 2 – CPU executes main program
- Step 3 – When I/O device is ready, it sets FGI or FGO = 1
- Step 4 – At end of instruction cycle, CPU checks: if (IEN AND (FGI OR FGO)) → R ← 1
- Step 5 – Interrupt cycle: Save PC to M[0]; branch to ISR at address 1; IEN ← 0
- Step 6 – ISR services the I/O device, then returns with BUN M[0]

Interrupt Cycle Micro-operations:

- R·T0: AR ← 0, TR ← PC (Prepare to save return address)
- R·T1: M[AR] ← TR, PC ← 0 (Save PC to M[0], jump to address 0)
- R·T2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0 (PC points to ISR; disable further interrupts)

Note: IEN = 0 during ISR execution prevents nested interrupts. ION at the end of ISR re-enables interrupts.

Advantages of Interrupt-Driven I/O over Polling:

- CPU is free to execute main program while I/O device operates
- Better CPU utilization and system throughput
- Allows overlap of CPU computation and I/O operations

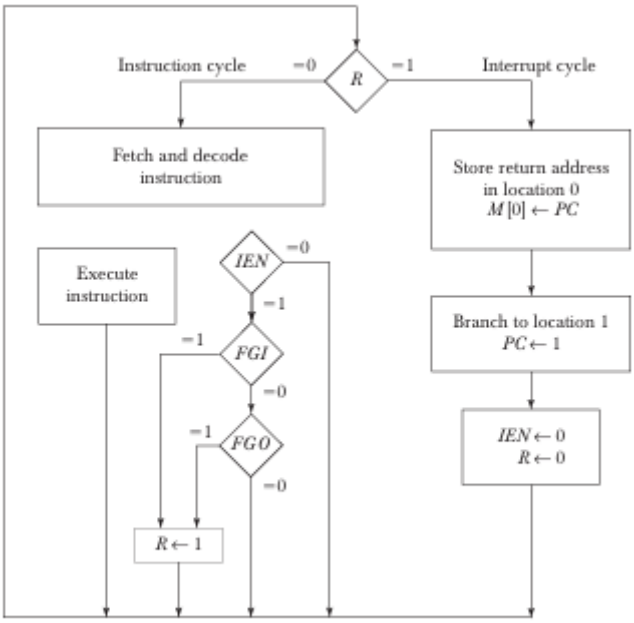


Figure 5-13 Flowchart for interrupt cycle.

Complete Computer Description

A complete description of the Basic Computer integrates all hardware components and operational behaviors described above. The complete system is defined by:

Hardware Components Summary:

Component	Specification
Memory Unit	4096 × 16 bit RAM – stores instructions and data
Registers	DR, AR, AC, IR, PC, TR, INPR, OUTR, SC (9 registers total)
Flip-Flops	I, S, E, R, IEN, FGI, FGO (7 flip-flops for status/control)
Decoder	2 decoders: 3×8 (opcode decode) + 4×16 (timing decode)
Common Bus	16-bit internal bus with 3-bit MUX selection
Control Logic Gates	Combinational logic generating all micro-operation control signals
ALU	Performs addition, AND, complement, shift operations on AC

Operational States of the Basic Computer:

- State 1 – Fetch State (T0, T1): Fetch instruction from memory pointed by PC
- State 2 – Decode State (T2): Decode instruction and set I bit
- State 3 – Indirect State (T3, if I=1): Resolve indirect address
- State 4 – Execute State (T3/T4+): Perform the operation
- State 5 – Interrupt State (R=1): Service interrupt request

Complete Instruction Execution Flow for AND M (Direct):

- T0: AR ← PC
- T1: IR ← M[AR], PC ← PC + 1
- T2: AR ← IR(11–0), I ← IR(15), decode IR(14–12)
- T3: (I=0, skip indirect) → Execute begins
- T4: DR ← M[AR]
- T5: AC ← AC ∧ DR, SC ← 0

Interconnection of All Units:

- All register inputs/outputs are routed through the common 16-bit bus
- ALU output connects to AC input; carry output connects to E flip-flop
- Memory unit connected via AR (address) and DR/AC (data)
- I/O connected via INPR, OUTR, FGI, FGO flags
- Control unit receives: IR (current instruction), SC (timing), flags (I, R, IEN, E, FGI, FGO)
- Control unit generates: Bus select signals, memory R/W, ALU function, register load enables

📝 **Note:** The complete functional behavior of the Basic Computer can be expressed as a Register Transfer Language (RTL) table mapping every combination of (SC × decoded instruction × flags) to a set of micro-operations.

UNIT-II: CENTRAL PROCESSING UNIT

Processor Organization:

The Central Processing Unit (CPU) acts as the brain of a computer system. Its main objective is to fetch instructions from memory, decode them, and coordinate the activities of all other system components to execute those instructions. Processor organization describes how internal sub-components—such as registers, Arithmetic Logic Units (ALUs), and control units—are structured and linked together to perform operations efficiently.

1. Core Structural Components of a Processor

A standard CPU architecture is generally partitioned into three foundational subsystems:

- **Arithmetic and Logic Unit (ALU):** The computational core that carries out basic binary calculations (addition, subtraction, logical AND, OR, shifting) on data operands.

- **Control Unit (CU):** The supervisory system that orchestrates internal traffic, reads operation codes (opcodes), and issues control lines to memory, the ALU, and peripherals.
- **Register File / Registers:** High-speed internal storage elements used to temporarily hold immediate data, address offsets, and processing status flags.

2. Internal Register Structure

Registers within the CPU can be broadly split into user-visible registers and control/status registers:

2.1 User-Visible Registers
These registers are exposed directly to machine-level programmers and assembly languages to minimize slow external main-memory queries:

- **General-Purpose Registers:** Can be assigned various temporary functions by the programmer (e.g., holding counters, loop parameters, or arithmetic variables).
- **Address/Pointer Registers:** Dedicated to holding memory addresses for various addressing modes (e.g., segment base addresses, stack pointers, index registers).
- **Data Registers:** Explicitly allocated to hold immediate data values or mathematical operands.

Control and Status Registers
These are critical hardware registers used exclusively by the Control Unit to govern system execution states:

- **Program Counter (PC) / Instruction Pointer (IP):** Tracks the memory location of the next sequential instruction to be fetched.
- **Instruction Register (IR):** Holds the current binary instruction word while it is decoded and executed by the CU.
- **Memory Address Register (MAR):** Directly connected to the system address bus; it holds the address from which data will be read or to which data will be written.
- **Memory Buffer Register (MBR) / Memory Data Register (MDR):** Directly tied to the data bus; it holds data read from memory or waiting to be written to a target peripheral or memory cell.
- **Program Status Word (PSW) / Flag Register:** Contains individual bit flags (e.g., Carry, Zero, Sign, Overflow, Interrupt Enable) that store status updates from recent calculations.

Register Organization: The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed.

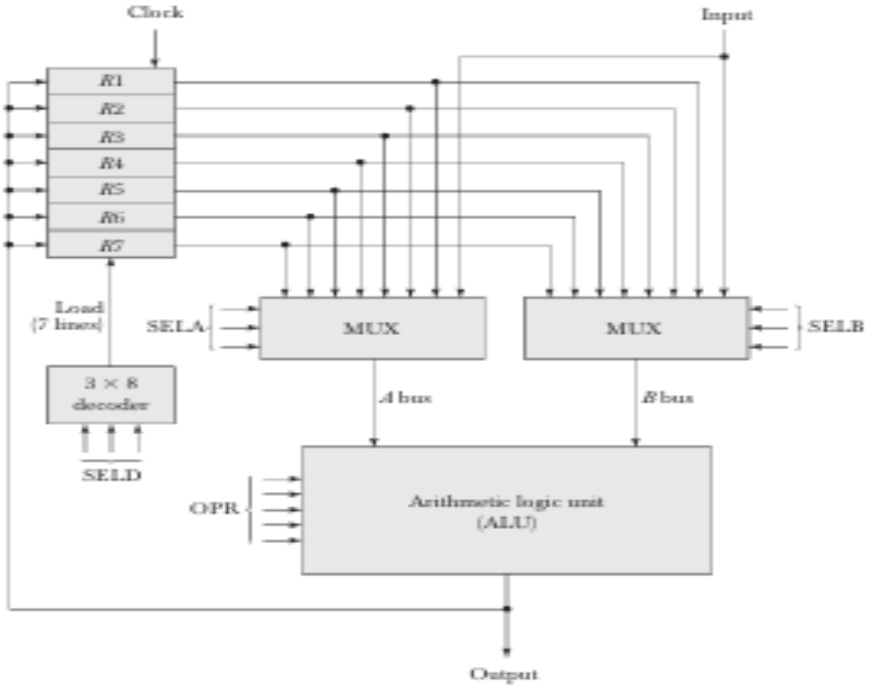
For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.

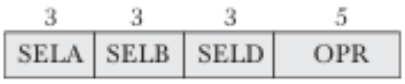
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.



(a) Block diagram

Control Word:

There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in Fig. 8-2(b). It consists of four fields. Three fields contain three bits each, and one field has five bits.



(b) Control word

Figure 8-2 Register set with common ALU.

TABLE 8-2 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add $A + B$	ADD
00101	Subtract $A - B$	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Examples of Micro-operations:

A control word of 14 bits is needed to specify a micro-peration in the CPU. The control word for a given micro-operation can be derived from the selection variables.
For example, the subtract micro-operation given by the statement

R1 ← R2-R3

specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A B. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 8-1 and 8-2. The binary control word for the subtract micro-operation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

TABLE 8-3 Examples of Microoperations for the CPU

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	–	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	–	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	–	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	–	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	–	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

The Instruction Cycle:

The processor executes programs through a repeated sequence known as the instruction cycle, which can be broken down into individual sub-cycles:

Phase Step	Primary Action	Internal Hardware Events

1. Fetch	Retrieve instruction from memory	Copy PC to MAR -> Read memory line into MBR -> Move MBR data to IR -> Increment PC.
2. Decode	Interpret opcode inside IR	The Control Unit analyzes the instruction bits to determine the operation type and operand addresses.
3. Indirect	Fetch effective address (if needed)	If indirect addressing is active, the operand address is placed in MAR to look up the true data pointer.
4. Execute	Perform command and store result	ALU processes operands under the command of CU control signals. Result is written to registers or memory.
5. Interrupt	Check for hardware/software flags	If interrupt lines are triggered, save the current PC and flags to the stack, then jump to the Interrupt Service Routine (ISR).

8086 Processor Architecture

The Intel 8086 is a seminal 16-bit microprocessor introduced in 1978. It features a 20-bit address bus, enabling it to access up to 1 MB of physical memory, and a 16-bit data bus. The internal architecture of the 8086 is split into two distinct, independently operating functional units: the Bus Interface Unit (BIU) and the Execution Unit (EU). This architectural division introduces internal pipelining, allowing instruction fetching and execution to occur concurrently.

Bus Interface Unit (BIU)

The BIU handles all data and address transfers on the external buses for the execution unit. Its primary responsibilities include fetching instructions from memory, reading data from memory/ports, and writing data to memory/ports. The BIU contains:

- **Segment Registers:** CS, DS, SS, and ES, which maintain base addresses for different memory segments.
- **Instruction Pointer (IP):** Contains the offset address of the next instruction to be fetched.
- **Instruction Queue:** A 6-byte first-in, first-out (FIFO) buffer used to pre-fetch up to 6 bytes of instruction code from memory during times when the EU is busy executing internal operations.

Execution Unit (EU)

The EU is responsible for decoding and executing instructions. It does not connect directly to the system buses; instead, it receives instructions from the BIU's instruction queue. The components of the EU include:

- **Arithmetic Logic Unit (ALU):** A 16-bit wide component that performs arithmetic operations (+, -, *, /) and logical operations (AND, OR, XOR, NOT).
- **Control Systems:** Decodes instructions and generates internal control signals.
- **General-Purpose & Flag Registers:** Holds temporary operands and status information.

Register Organization

The 8086 contains a collection of fourteen 16-bit registers, categorized based on their roles and functionality into General Purpose, Pointer & Index, Segment, and Flag registers.

General Purpose Registers

These four 16-bit registers (AX, BX, CX, DX) can be split and accessed as separate 8-bit registers (High byte: AH, BH, CH, DH; Low byte: AL, BL, CL, DL):

- **AX (Accumulator):** Optimized for arithmetic, logical, and I/O operations. Mandatory for multiplication and division operations.
- **BX (Base Register):** Used as a base pointer to hold the base memory address for indirect addressing modes.
- **CX (Count Register):** Acts as an automatic loop counter during string operations and repeat loops.
- **DX (Data Register):** Holds the 16-bit I/O port address during indirect I/O instructions, or high 16 bits of a 32-bit product/dividend.

Pointer and Index Registers

These 16-bit registers generally hold offset addresses for memory access:

- **SP (Stack Pointer):** Points to the current top of the stack area within the Stack Segment (SS).
- **BP (Base Pointer):** Used primarily to access data parameters stored on the stack during subroutine calls.
- **SI (Source Index):** Points to source string operands during string manipulation commands.
- **DI (Destination Index):** Points to the destination target for string manipulations.

Segment Registers

To manage 1 MB of memory using 16-bit addresses, the 8086 uses memory segmentation, splitting memory into overlapping logical sections of 64 KB each. The four segment registers specify the base addresses:

- **CS (Code Segment):** Points to the segment containing the program instructions currently being executed.
- **DS (Data Segment):** Points to the primary segment where program variables and global data reside.
- **SS (Stack Segment):** Points to the segment reserved for stack operations (PUSH/POP, return addresses).
- **ES (Extra Segment):** An additional data segment typically utilized for string destination targets.

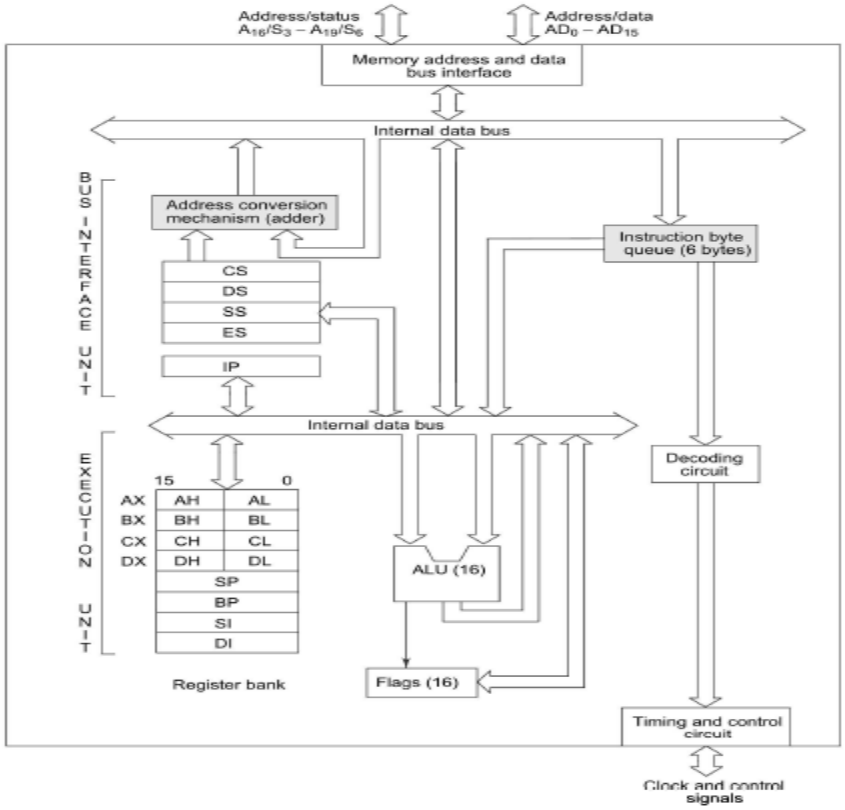


Fig. 1.2 *8086 Architecture*

PHYSICAL MEMORY ORGANISATION:

In an 8086 based system, the 1Mbytes memory is physically organised as an odd bank and an even bank, each of 512 Kbytes, addressed in parallel by the processor. Byte data with an even address is transferred on D_7-D_0, while the byte data with an odd address is transferred on D_{15}-D_8 buss lines. The processor provides two enable signals, BHE and $A_0$for selection of either even or odd or both the banks. The instruction stream is fetched from memory as words and is addressed internally by the processor as necessary. In other words, if the processor fetches a word (consecutive two bytes) from memory, there are different possibilities, like:

1. Both the bytes may be data operands
2. Both the bytes may contain opcode bits
3. One of the bytes may be opcode while the other may be data

All the above possibilities are taken care of by the internal decoder circuit of the microprocessor.

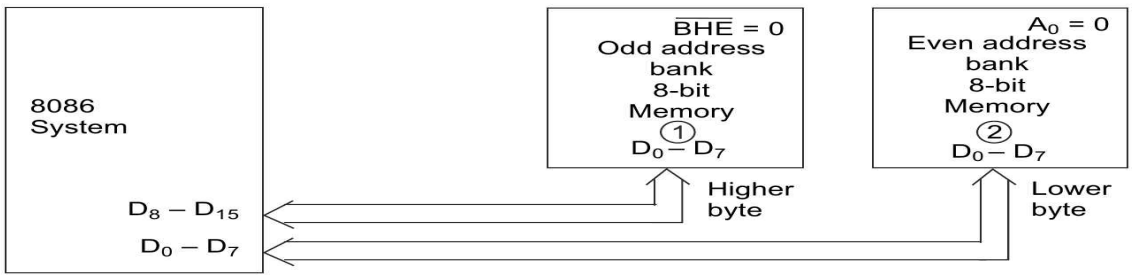


Fig. Physical Memory Organisation

GENERAL BUS OPERATION:

The 8086 has a combined address and data bus commonly referred to as a time multiplexed address and databus. The main reason behind multiplexing address and data over the same pins is the maximum utilisation of processor pins and it facilitates the use of 40 pin standard DIP package. The bus can be demultiplexed usinga few latches and trans receivers, whenever required.

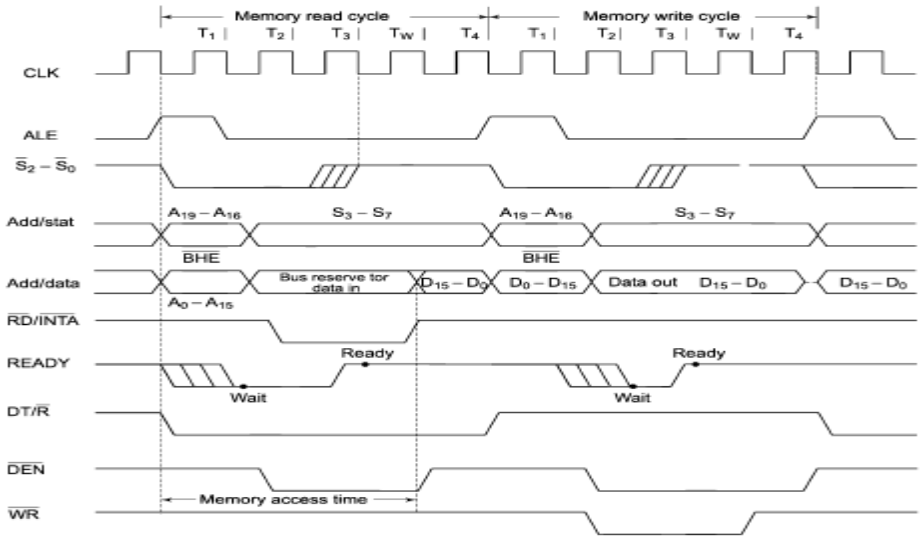


Fig. 1.8 *General Bus Operation Cycle of 8086*

I/O ADDRESSING CAPABILITY:

The 8086/8088 processor can address up to 64K I/O byte registers or 32K word registers. The limitation that the address of an I/O device must not be greater than 16 bits in size, this means that a maximum number of 216, i.e. 64Kbyte I/O devices may be accessed by the CPU. The I/O address appears on the address lines A_0 to A_{15} for one clock cycle (T_1). It may then be latched using the ALE signal. The upper address lines (A_{16}-A_{19}) are at logic 0 level during the I/O operations.

The 16-bit register DX is used as 16-bit I/O address pointer, with full capability to address up to 64K devices.

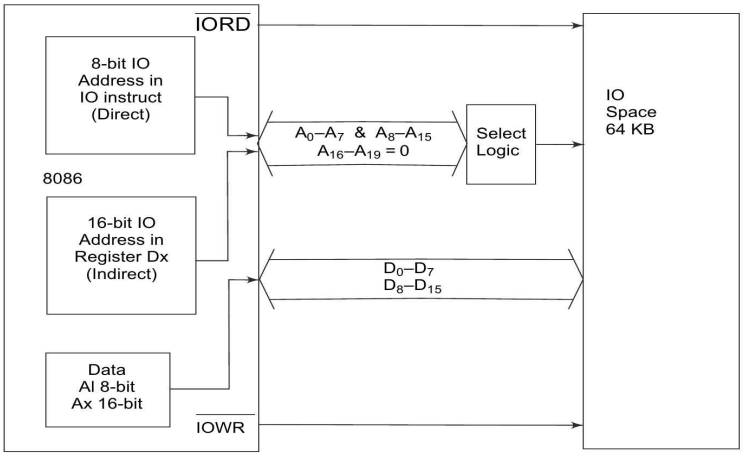


Fig. 8086 IO Addressing

SPECIAL PROCESSOR ACTIVITIES:

1. Processor Reset and Initialisation

2. HALT

3. TEST and Synchronisation with External Signals

4.Deriving System Bus

MINIMUM MODE 8086 SYSTEM AND TIMINGS

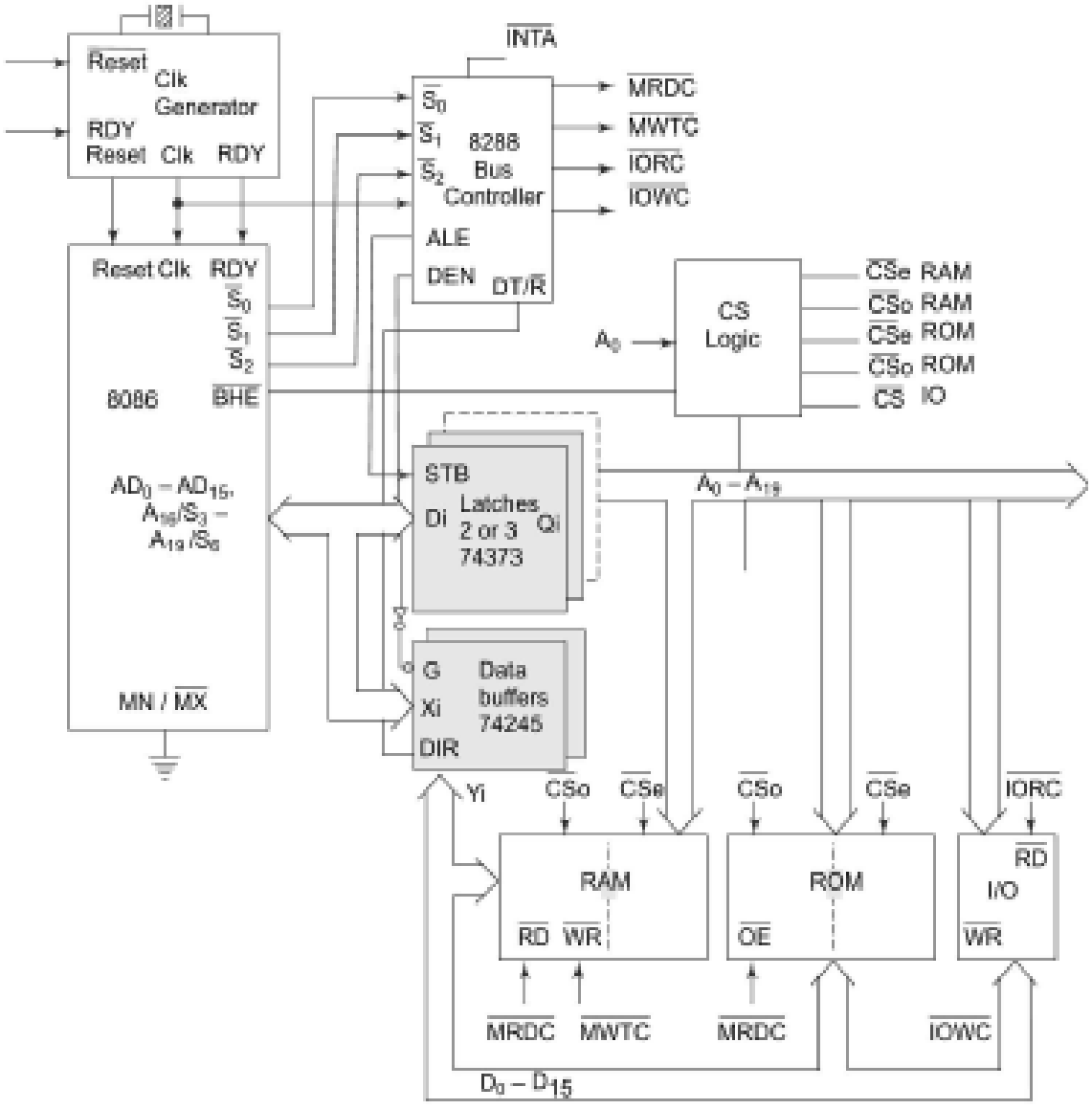


Fig. 1.15 *Maximum Mode 8086 System*

MAXIMUM MODE 8086 SYSTEM AND TIMINGS

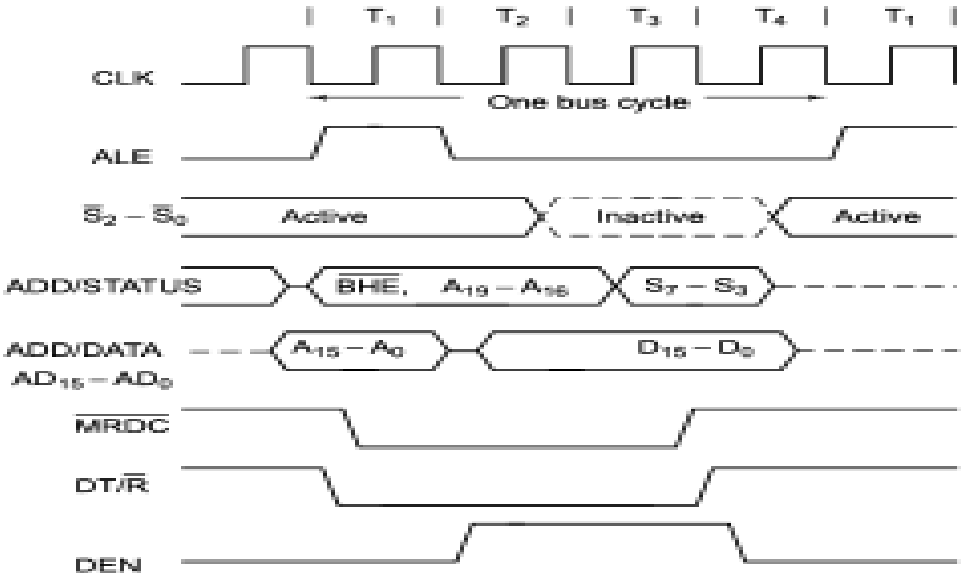


Fig. 1.16 (a) *Memory Read Timing in Maximum Mode*

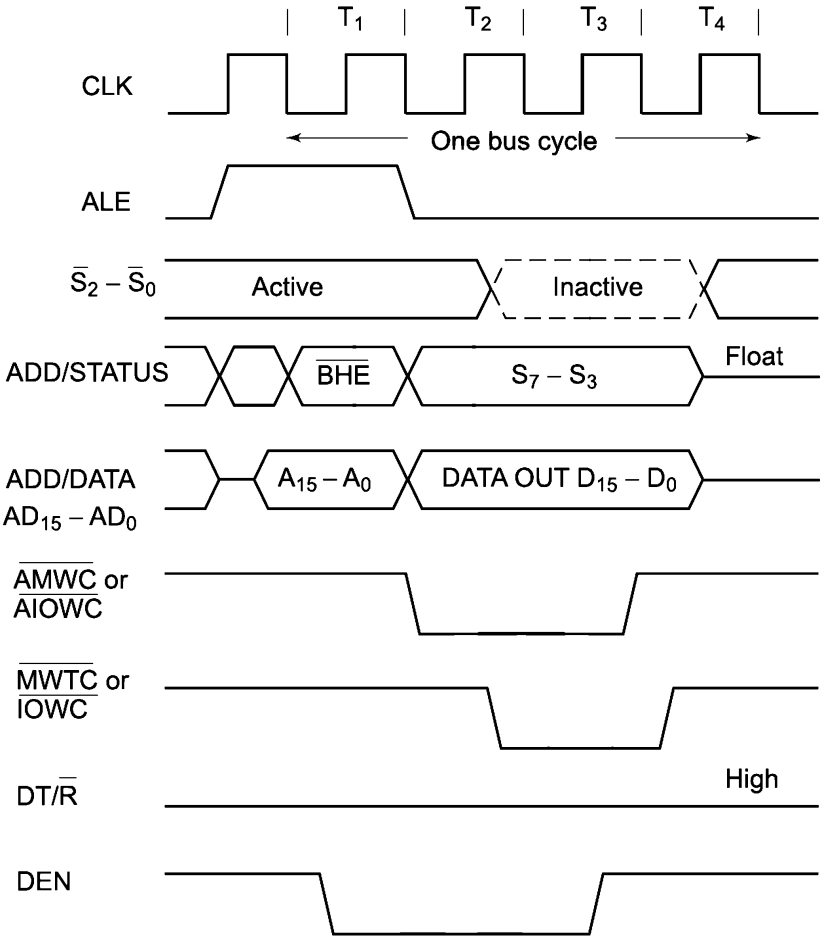


Fig. 1.16(b) *Memory Write Timing in Maximum Mode*

Table 1.5

$M/\overline{IO}$	$\overline{RD}$	$\overline{DEN}$	*Transfer Type*
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write

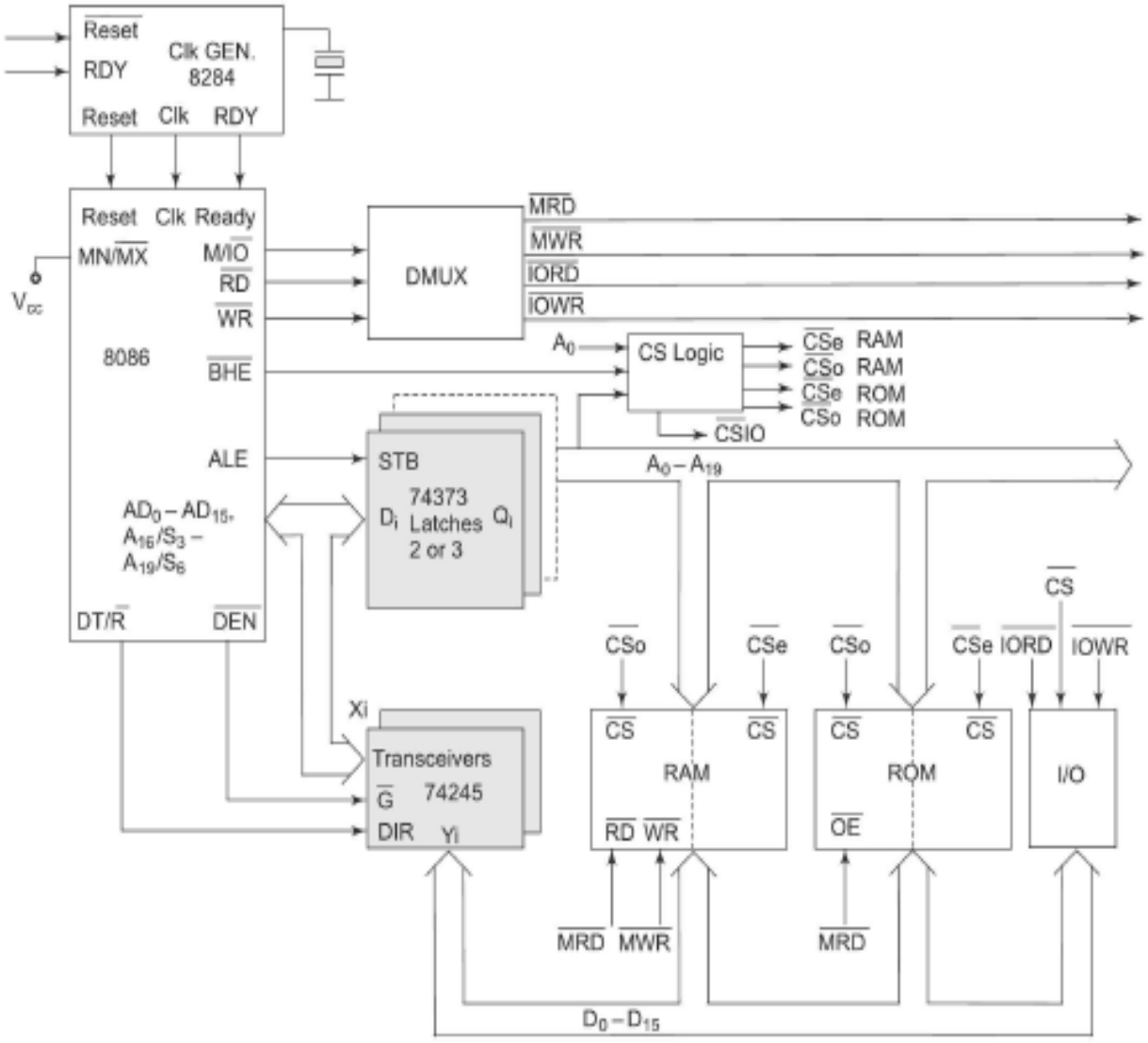


Fig. 1.13 *Minimum Mode 8086 System*

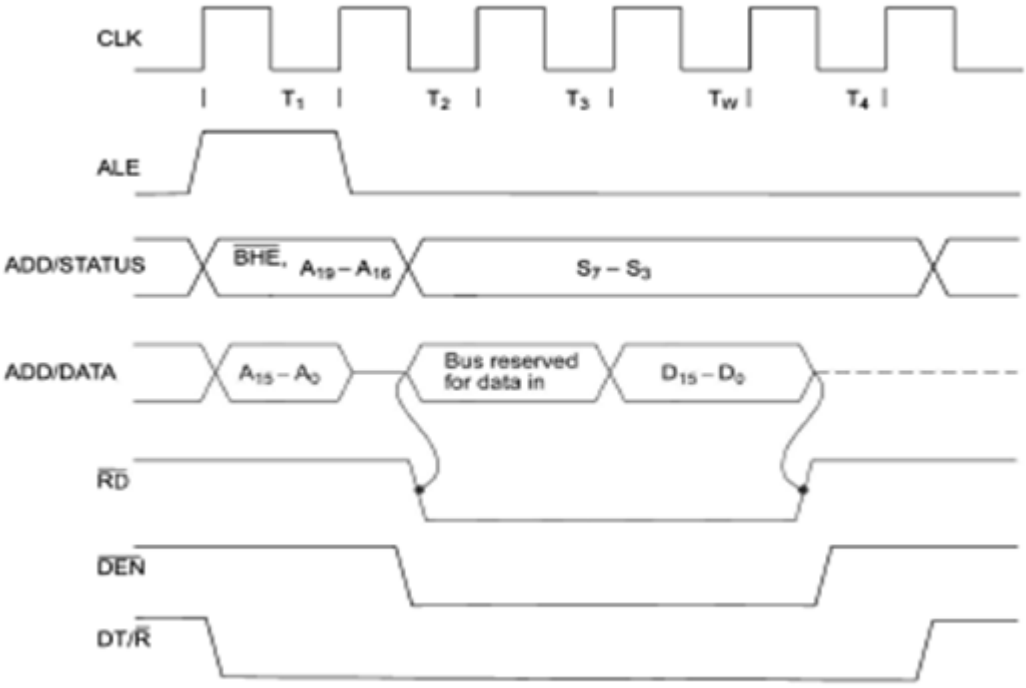


Fig. 1.14(a) *Read Cycle Timing Diagram for Minimum Mode*

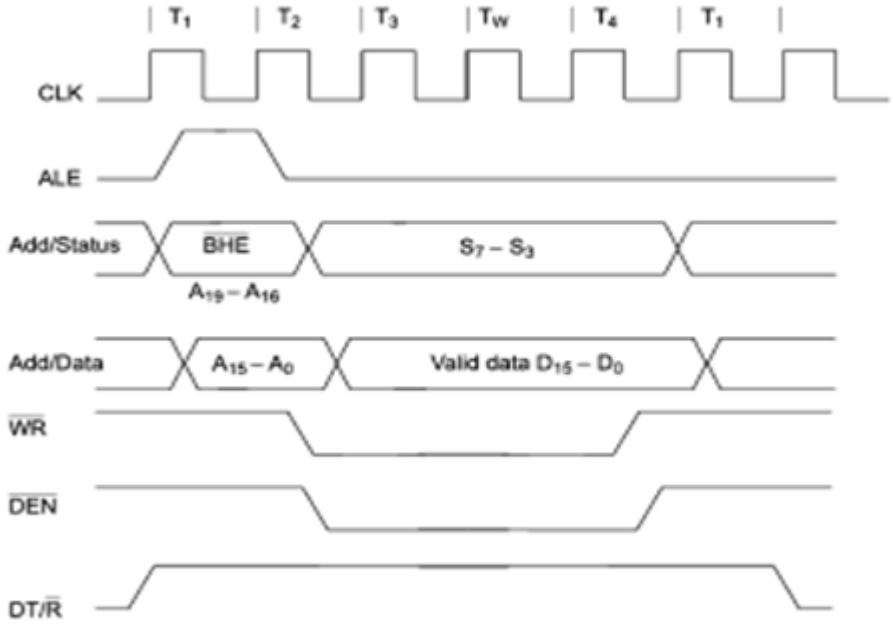


Fig. 1.14(b) *Write Cycle Timing Diagram for Minimum Mode Operation*

Timings for RQ/GT Signals

The request/grant response sequence contains a series of three pulses as shown in the timing diagram Fig.1.16 (c). The request/grant pins are checked at each rising pulse of clock input. When a request is detected and if the conditions discussed in pin diagram section of this chapter for valid HOLD request are satisfied, the processor issues a grant pulse over the RQ/GTo pin immediately during the T4 (current) or T_1 (next) state. When the requesting master receives this pulse, it accepts the control of the bus. The requesting master uses the bus till it requires. When it is ready to relinquish the bus, it sends a release pulse to the processor (host) using the RQ/GT pin.

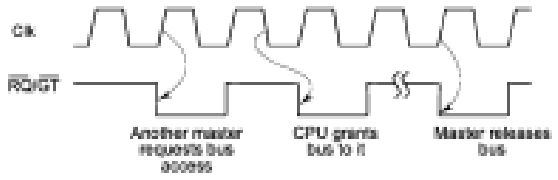


Fig. 1.16 (c) *$\overline{RQ}/\overline{GT}$ Timings in Maximum Mode*

Introduction, The arithmetic logic unit, Integer representation, Integer arithmetic, Floating point representation, Floating point arithmetic, Addition and Subtraction, Multiplication Algorithms, Division Algorithms, Floating - point Arithmetic operations. Data Transfer and Manipulation

ARITHMETIC AND LOGIC UNIT:

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out. We have, in a sense, reached the core or essence of a computer when we consider the ALU.

An ALU and indeed, all electronic components in the computer, are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

Figure 3.1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU .The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.

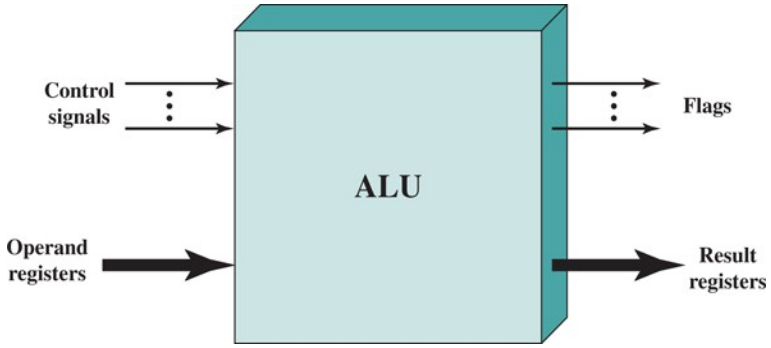


Figure 3.1 ALU Inputs and Outputs

The flag values are also stored in registers within the processor. The processor provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

INTEGER REPRESENTATION:

In the binary number system, arbitrary numbers can be represented with just the digits zero and one, the minus sign (for negative numbers), and the period, or **radix point** (for numbers with a fractional component).for a basic refresher on number systems (decimal, binary, hexadecimal).

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of special symbols for the minus sign and radix point. Only binary digits (0 and 1) may be used to represent

numbers. If we are limited to nonnegative integers, the representation is straightforward. An 8-bit word can represent the numbers from 0 to 255, such as

$$\begin{aligned}00000000 &= 0\\00000001 &= 1\\00101001 &= 41\\10000000 &= 128\\11111111 &= 255\end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ A, its value is

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

SIGN-MAGNITUDE REPRESENTATION

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a **sign bit**. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost $n-1$ bits hold the magnitude of the integer

$$+18 = 00010010$$

$$-18 = 10010010 \text{ (sign magnitude)}$$

The general case can be expressed as follows:

Sign Magnitude

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. Another drawback is that there are two representations of 0:

$$+0_{10} = 00000000$$

$$-0_{10} = 10000000 \text{ (sign magnitude)}$$

This is inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation.

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.

In the literature, the terms *two's complement* or *2's complement* are often used. Here we follow the practice used in standards documents and omit the apostrophe (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

TWO'S COMPLEMENT REPRESENTATION

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted. **Table 3.1** highlights key characteristics of twos complement representation and arithmetic, which are elaborated in this section and the next.

Table 3.1 Characteristics of Twos Complement Representation and Arithmetic

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract *B* from *A*, take the twos complement of *B* and add it to *A*.

INTEGER ARITHMETIC

This section examines common arithmetic functions on numbers in twos complement representation.

Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.
2. Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the **twos complement operation**, or the taking of the twos complement of an integer.

ADDITION AND SUBTRACTION

Addition in twos complement is illustrated Addition proceeds as if the two numbers were unsigned integers. The first four examples illustrate successful operations. If the result of the operation is positive, we get a positive number in twos complement form, which is the same as in unsigned-integer form. If the result of the operation is negative, we get a negative number in

twos complement form. Note that, in some instances, there is a carry bit beyond the end of the word (indicated by shading), which is ignored.

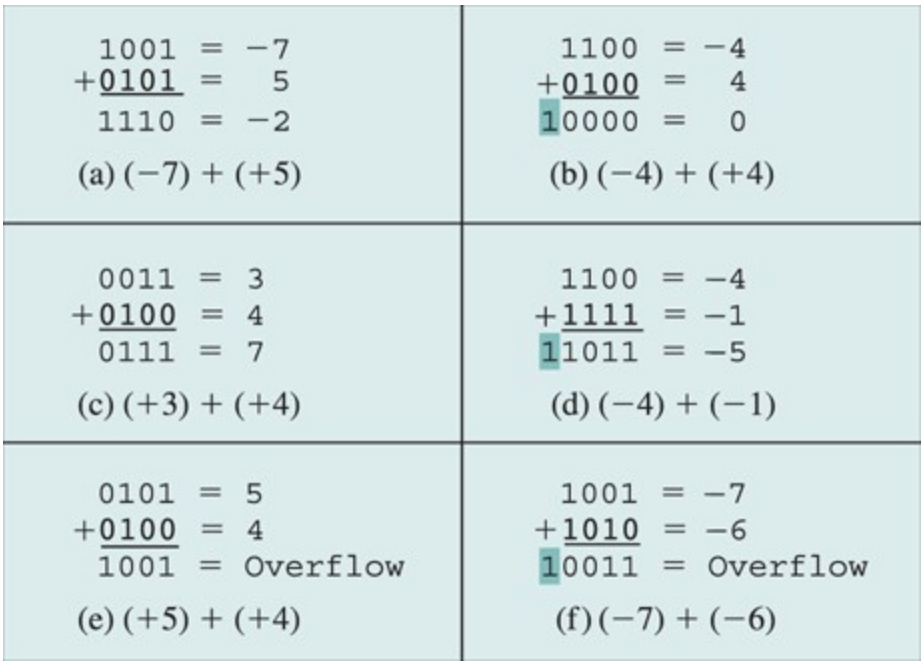


Figure 3.3 Addition of Numbers in Twos Complement Representation

On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**. When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result. To detect overflow, the following rule is observed:

OVERFLOW RULE:
If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign. **Figures 3.3e** and **f** show examples of overflow. Note that overflow can occur whether or not there is a carry.

Subtraction is easily handled with the following rule:

SUBTRACTION RULE:
To subtract one number (**subtrahend**) from another (**minuend**), take the twos complement (negation) of the subtrahend and add it to the minuend.
Thus, subtraction is achieved using addition, as illustrated in **Figure 3.4**. The last two examples demonstrate that the overflow rule still applies.

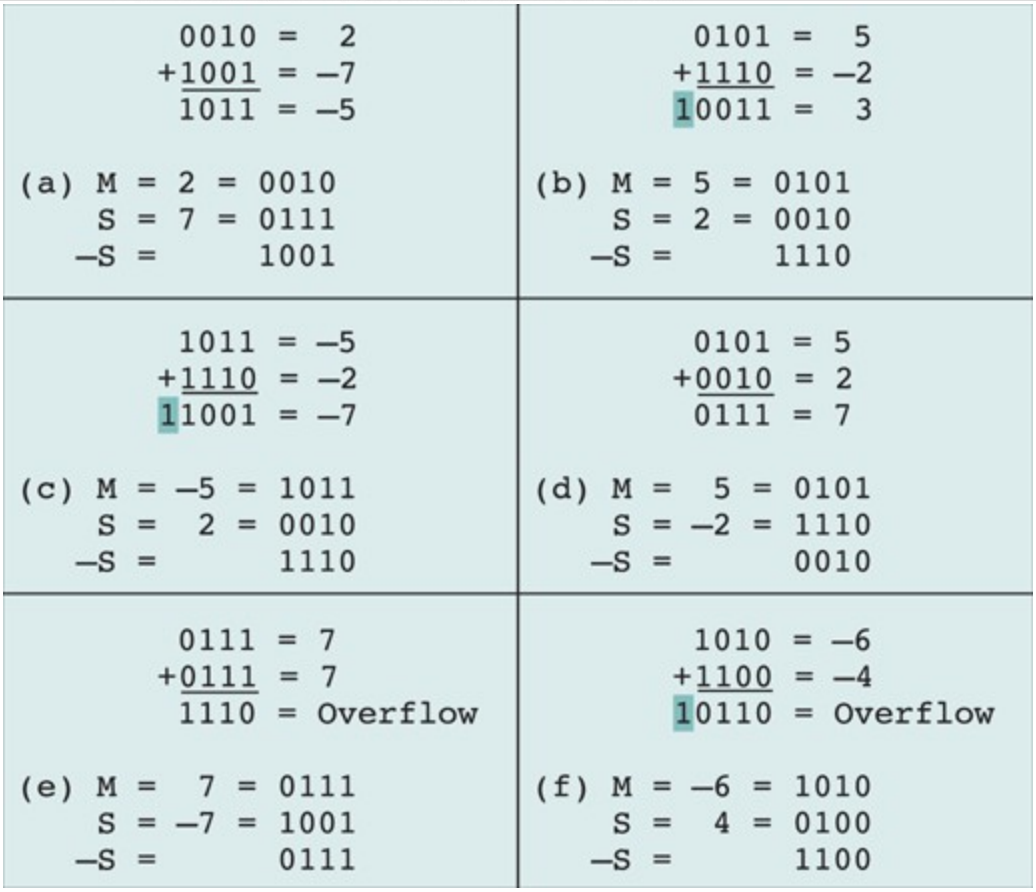


Figure 3.4 Subtraction of Numbers in Twos Complement Representation ($M - S$)

Some insight into twos complement addition and subtraction can be gained by looking at a geometric depiction [BENH92], as shown in **Figure 3.5**. The circle in the upper half of each part of the figure is formed by selecting the appropriate segment of the number line and joining the endpoints. Note that when the numbers are laid out on a circle, the twos complement of any number is horizontally opposite that number (indicated by dashed horizontal lines). Starting at any number on the circle, we can add positive k (or subtract negative k) to that number by moving k positions clockwise, and we can subtract positive k (or add negative k) from that number by moving k positions counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given (overflow).

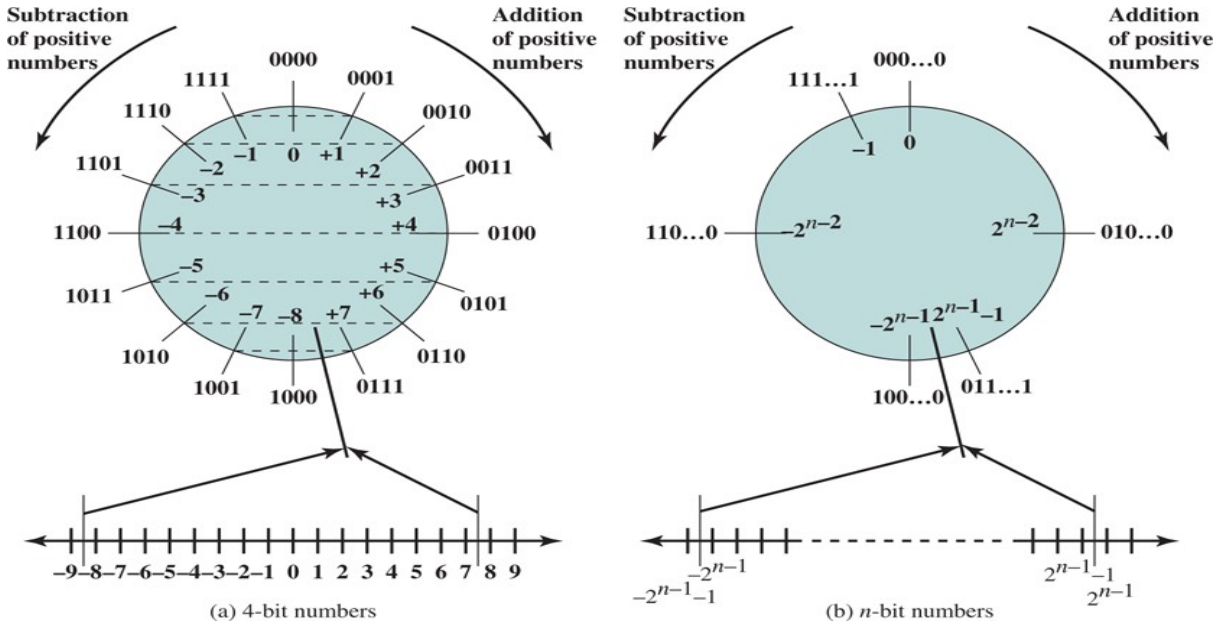


Figure 3.5 Geometric Depiction of Twos Complement Integers

ALL OF the examples of **Figures 3.3** and **3.4** are easily traced in the circle of **Figure 3.5**.,**Figure 3.6** suggests the data paths and hardware elements needed to accomplish addition and subtraction. The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. (A logic implementation of an adder is given in Chapter 12.) For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag (0 = no overflow ; 1 = overflow) . For subtraction, the subtrahend (B register) is passed through a twos complement so that its twos complement is presented to the adder. Note that **Figure3.6** only shows the data paths. Control signals are needed to control whether or not the complement is used, depending on whether the operation is addition or subtraction

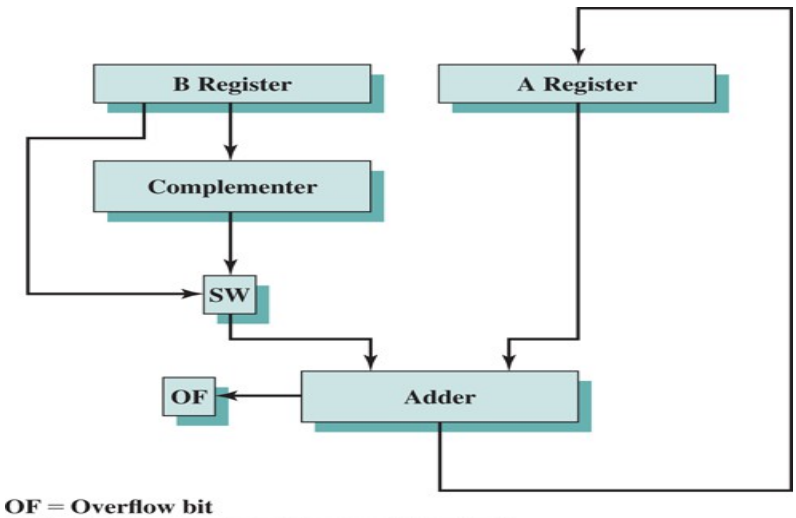


Figure 3.6 Block Diagram of Hardware for Addition and Subtraction

MULTIPLICATION:

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software. A wide variety of algorithms have been used in various computers. The purpose of this subsection is to give the reader some feel for the type of approach typically taken. We begin with the simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation.

UNSIGNED INTEGERS

Figure 3.7 illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Several important observations can be made

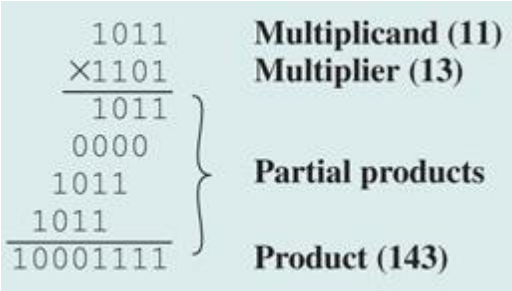


Figure 3.7

Multiplication involves the generation of **partial products**, one for each digit in the **multiplier** These partial products are then summed to produce the final product.

1. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the **multiplicand**.
2. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
3. The multiplication of two *n*-bit binary integers results in a **product** of up to 2*n* bits in length (e.g.,11 × 11 = 1001).

Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient. First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Figure 3.8a shows a possible implementation employing these measures. The multiplier and multiplicand are loaded into two registers (Q and M). A third register, the A register, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition.

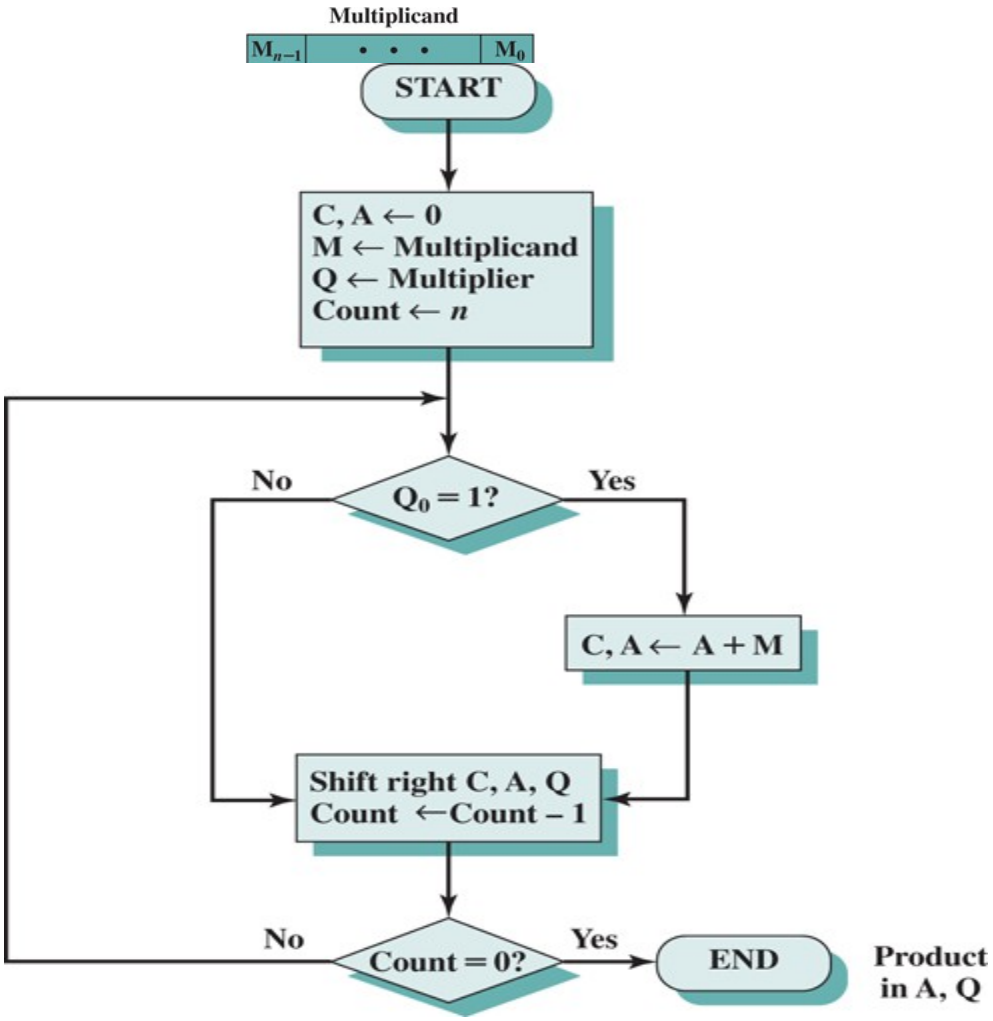


Figure 3.8 Hardware Implementation of Unsigned Binary Multiplication

The operation of the multiplier is as follows. Control logic reads the bits of the multiplier one at a time. If Q_0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost. If Q_0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier. The resulting 2*n*-bit product is contained in the A and Q registers. A flowchart of the operation is shown in **Figure 3.9**, and an example is given in **Figure 3.8b**. Note that on the second cycle, when the multiplier bit is 0, there is no add operation

TWO'S COMPLEMENT MULTIPLICATION

We have seen that addition and subtraction can be performed on numbers in twos complement

notation by treating them as unsigned integers. Consider

$$\begin{array}{r} 1001 \\ +0011 \\ \hline 1100 \end{array}$$

If these numbers are considered to be unsigned integers, then we are adding 9 (1001) plus 3 (0011) to get 12 (1100). As twos complement integers, we are adding -7 (1001) to 3 (0011) to get -4 (1100).

Unfortunately, this simple scheme will not work for multiplication. To see this, consider again **Figure 3.7**. We multiplied 11 (1011) by 13 (1101) to get 143 (10001111). If we interpret these as twos complement numbers, we have -5 (1011) times -3 (1101) equals -113 (10001111). This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative. In fact, it will not work if either the multiplicand or the multiplier is negative. To justify this statement, we need to go back to **Figure 3.7** and explain what is being done in terms of operations with powers of 2. Recall that any unsigned binary number can be expressed as a sum of powers of 2. Thus, 1101 = $1 \times 2^3 + 1 \times 2^2 + 0 \times$

1001	(9)		1001	(−7)	
× 0011	(3)		× 0011	(3)	
00001001	1001 × 2^0		11111001	(−7) × 2^0 = (−7)	
00010010	1001 × 2^1		11110010	(−7) × 2^1 = (−14)	
00011011	(27)		11101011	(−21)	

(a) Unsigned integers (b) Twos complement integers

× 1101	
00001011	1011 × 1 × 2^0
00000000	1011 × 0 × 2^1
00101100	1011 × 1 × 2^2
01011000	1011 × 1 × 2^3
10001111	

$$2^1 + 1 \times 2^0 = 2^3 + 2^2 + 2^0$$

Figure 3.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011. Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative. The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a 2*n*-bit field; the sign bits of the partial products must line up. This is demonstrated in **Figure 3.11**, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of $9 \times 3 = 27$ proceeds simply. However, if 1001 is interpreted as the twos complement value -7, then each partial product must be a negative twos complement number of 2*n* (8) bits, as shown in **Figure 3.11b**. Note that this is accomplished by padding out each partial product to the left with binary 1s.

1001 (9)	1001 (−7)
× 0011 (3)	× 0011 (3)
00001001 1001 × 2^0	11111001 (−7) × 2^0 = (−7)
00010010 1001 × 2^1	11110010 (−7) × 2^1 = (−14)
00011011 (27)	11101011 (−21)
(a) Unsigned integers	(b) Twos complement integers

Figure 11.11 Comparison of Multiplication of Unsigned and Twos Complement Integers

If the multiplier is negative, straightforward multiplication also will not work. The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place. For example, the 4-bit decimal number −3 is written 1101 in twos complement. If we simply took partial products

based on each bit position, we would have the following correspondence:

$$1101 \leftrightarrow -(1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0) = -(2^3 + 2^2 + 2^0)$$

1 0

Division

Division is somewhat more complex than multiplication, but is based on the same general principles. As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction.

Figure 3.15 shows an example of the long division of unsigned binary integers. It is instructive to describe the process in detail. First, the bits of the **dividend** are examined from left to right, until the set of bits examined represents a number greater than or equal to the **divisor**; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the **quotient** from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder.*

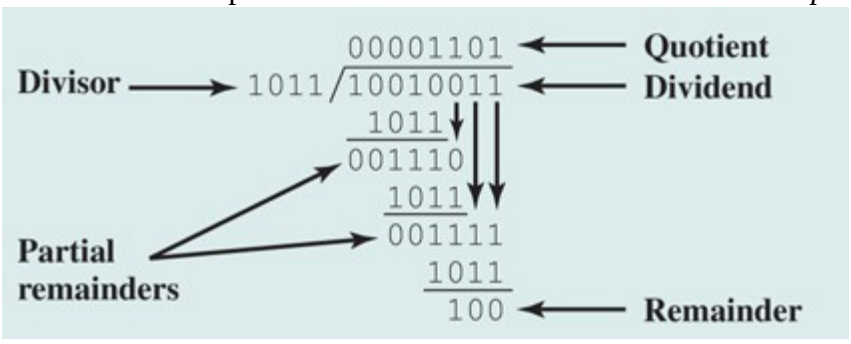


Figure 11.15 Example of Division of Unsigned Binary Integers

From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the

divisor. As before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

Figure 3.16 shows a machine algorithm that corresponds to the long division process. The divisor is placed in the M register, the dividend in the Q register. At each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder. If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A

to restore the previous value. The count is then decremented, and the process continues for *n* steps. At the end, the quotient is in the Q register and the **remainder** is in the A register.

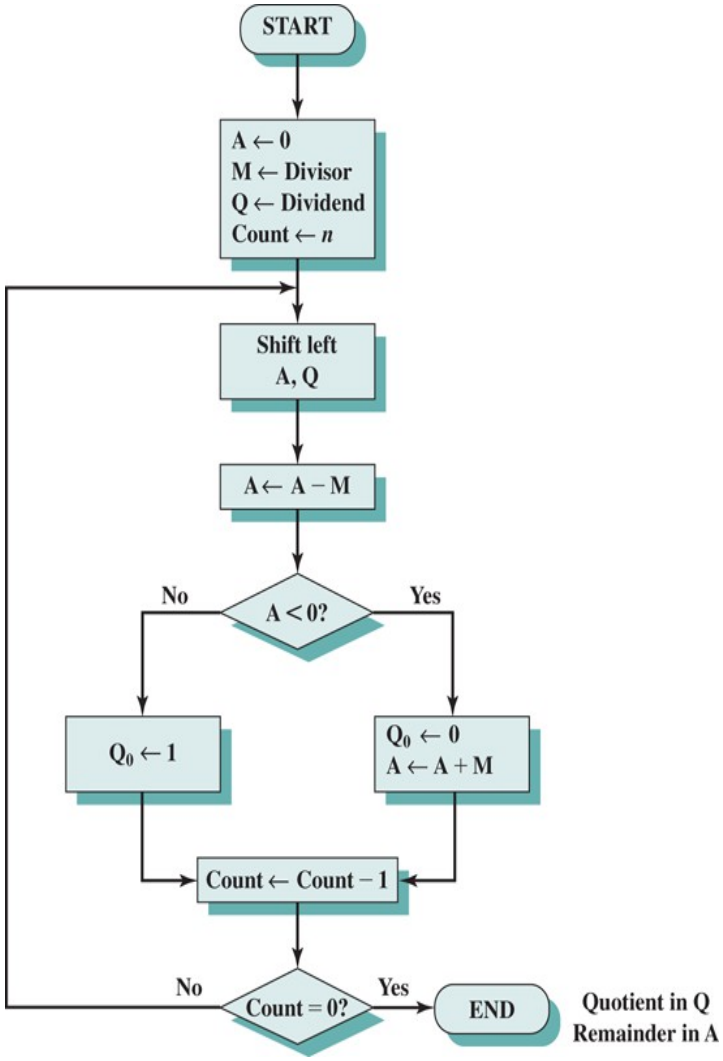


Figure 3.16 Flowchart for Unsigned Binary Division

This process can, with some difficulty, be extended to negative numbers. We give here one approach for twos complement numbers. An example of this approach is shown in **Figure 3.17**.

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 3.17 Example of Restoring Twos Complement Division (7/3)

The algorithm assumes that the divisor V and the dividend D are positive and that $|V| < |D|$. If $|V| = |D|$, then the quotient $Q = 1$ and the remainder $R = 0$. If $|V| > |D|$, then $Q = 0$ and $R = D$. The algorithm can be summarized as follows:

1. Load the twos complement of the divisor into the M register; that is, the M register contains the negative of the divisor. Load the dividend into the A, Q registers. The dividend must be expressed as a 2*n*-bit positive number. Thus, for example, the 4-bit 0111 becomes 00000111.
2. Shift A, Q left 1 bit position.
3. Perform $A \leftarrow A - M$. This operation subtracts the divisor from the contents of A.
   a. If the result is nonnegative (most significant bit of $A = 0$), then set $Q_0 \leftarrow 1$.
   b. If the result is negative (most significant bit of $A = 1$), then set $Q_0 \leftarrow 0$ and restore the previous value of A.
4. Repeat steps 2 through 4 as many times as there are bit positions in Q.
5. The remainder is in A and the quotient is in Q.

To deal with negative numbers, we recognize that the remainder is defined by

$$D = Q \times V + R$$

That is, the remainder is the value of R needed for the preceding equation to be valid Consider the following examples of integer division with all possible combinations of signs of D and V:

$$D = 7 \; V = \; 3 \quad \Rightarrow Q = 2 \qquad R = 1$$

$$D = 7 \qquad V = -3 \Rightarrow Q = -2 \; R = 1$$

$$D = -7\ V = 3 \Rightarrow Q = -2\ R = -1$$

$$D = -7\ V = -3 \Rightarrow Q = 2\ R = -1$$

FLOATING-POINT REPRESENTATION

Principles

With a fixed-point notation (e.g., twos complement) it is possible to represent a range of positive and negative integers centered on or near 0. By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well.

This approach has limitations. Very large numbers cannot be represented, nor can very small fractions. Furthermore, the fractional part of the quotient in a division of two large numbers could be lost.For decimal numbers, we get around this limitation by using scientific notation. Thus,976,000,000,000,000 can be represented as 9.76×10^{14} and 0.0000000000000976 can be represented as 9.76×10^{-14} What we have done, in effect, is dynamically to slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point. This allows a range of very large and very small numbers to be represented with only a few digits.

This same approach can be taken with binary numbers. We can represent a number in the form

$$\pm S \times B^{\pm E}$$

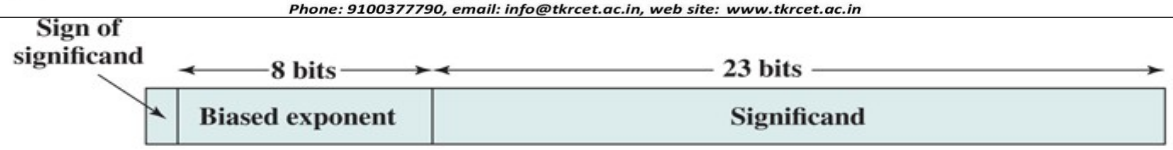
This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand S
- Exponent E

The **base** B is implicit and need not be stored because it is the same for all numbers. Typically, it is assumed that the radix point is to the right of the leftmost, or most significant, bit of the significand.

That is, there is one bit to the left of the radix point.

The principles used in representing binary floating-point numbers are best explained with an example. **Figure 3.18a** shows a typical 32-bit floating-point format. The leftmost bit stores the **sign** of the number (0 = positive,1 = negative) . The **exponent** value is stored in the next 8 bits. The representation used is known as a **biased representation**. A fixed value, called the bias, is subtracted from the field $k - 1$ to get the true exponent value. Typically, the bias equals $(2^{k-1} - 1)$, where k is the number of bits in the binary exponent. In this case, the 8-bit field yields the numbers 0 through 255. With a bias of 127 $(2^7 - 1)$, the true exponent values are in the range -127 to $+128$. In this example, the base is assumed to be 2.



(a) Format

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$
$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$
$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$
$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$

(b) Examples

Figure 3.18 Typical 32-Bit Floating-Point Format

Table 3.2 shows the biased representation for 4-bit integers. Note that when the bits of a biased representation are treated as unsigned integers, the relative magnitudes of the numbers do not change. For example, in both biased and unsigned representations, the largest number is 1111 and the smallest number is 0000. This is not true of sign-magnitude or twos complement representation. An advantage of biased representation is that nonnegative floating-point numbers can be treated as integers for comparison purposes. The final portion of the word (23 bits in this case) is the **significand**.The term **mantissa**, sometimes used instead of *significand*, is considered obsolete. *Mantissa* also means "the fractional part of a logarithm," so is best avoided in this context.

Data Transfer and Manipulation

Computer systems consist of a set of instructions that help the users to easily carry out their computational tasks. The instruction set differs from one computer system to another. The operations are represented by binary codes and the binary code assignment in the operation field of the instruction can be different in different computers. However, the actual operations in the instruction set are not very different from one computer system to another. The symbolic names of the instruction (assembly language notation) may also differ from one computer to another. An instruction usually contains opcode, addressing field, and operand field. There are different types of opcodes and based on the type of opcode, the instructions can be classified as follows:

1. Data Transfer Instructions
2. Data Manipulation Instructions
3. Program Control Instructions

The instructions can be described as follows:

1. **Load:** The load instruction is used to transfer data from the memory to a processor register, which is usually an accumulator.
2. **Store:** The store instruction transfers data from processor registers to memory.
3. **Move:** The move instruction transfers data from processor register to memory or memory to processor register or between processor registers itself.
4. **Exchange:** The exchange instruction swaps information either between two registers or between
a register and a memory word.

5. **Input:** The input instruction transfers data between processor register and input terminal.

6. **Output:** The output instruction transfers data between processor register and output terminal.
7. **Push and Pop:** The push and pop instructions transfer data between a processor register and memory stack.

All these instructions are associated with a variety of addressing modes. Some assembly language instructions use different mnemonic symbols just to differentiate between the different addressing modes.

Example: The mnemonic symbols for load immediate is LDI

Thus, it is necessary to be familiar with various addressing modes and different types of instructions to write efficient assembly language programs for a computer

Data Manipulation Instructions

Data manipulation instructions have computational capabilities. They perform arithmetic, logic, and shift operations on data. There are three basic types of data manipulation instructions:

1. Arithmetic Instructions
2. Logical and Bit Manipulation Instructions
3. Shift Instructions

During execution of the instruction, each instruction goes through the fetch phase, where it reads the binary code of the instruction from the memory. According to the rules of the instruction addressing mode, the operands are brought in processor registers. Finally, the instruction in the processor is executed.

Arithmetic Instructions

Arithmetic operations include addition, subtraction, multiplication and division. Some computers provide instructions only for addition and subtraction operations, and generate multiplication and division operations from these two operations. Each instruction is represented by a mnemonic
symbol.

The description of these instructions is as follows:

1. **Increment:** The increment instruction adds 1 to the value stored in register or memory word.
2. **Decrement:** The decrement instruction subtracts 1 from the contents stored in register or memory
word.
3. **Arithmetic Instructions:** The arithmetic instructions are available for different types of data such as floating point, binary, single precision, or double precision data.

During execution of arithmetic instructions, the processor status flags or conditional codes are set to designate the outcome of the operation.

Logical and Bit Manipulation Instructions

Logical instructions carry out binary operations on the bits stored in the registers. In logical operations, each bit of the operand is treated as a Boolean variable. Logical instructions can change bit value, clear a group of bits, or can even insert new bit value into operands that are stored in registers or memory words. Each logical instruction is represented by mnemonic symbols.

The clear instruction replaces the specific operand by 0's. The complement instruction inverts all the bits of the operand and produces 1's complement. The AND, OR, and XOR instructions perform
logical operations on each bit or group of bits of the operand.

Logical instructions can also manipulate individual bits or group of bits. The bit manipulation operation can clear a bit to 0, can set a bit to 1, or can complement a bit.

The **AND** instruction can clear a bit or group of bits of an operand. For Boolean variable a, the relationship 'ab0 = 0' and 'ab1 = a' indicates that the binary variable when ANDed with 0 changes
the value to 0. However, the variable when ANDed with 1 does not change the value. Thus, bits of an operand can be cleared by ANDing the operand with another operand that has to clear all 0 bits in its position. It is also known as mask because it masks 0s in selected bit positions of an operand.

The **OR** instruction can set a bit or group of bits of an operand. For Boolean variable a, the relationship 'a + 1 =1' and 'a + 0 = a' indicates that the binary variable when ORed with 1, changes
the value to 1. However, the variable when OR ed with 0 does not change the value. Thus, OR instruction is used to set the bits to 1 by OR ing the bits of an operand with another operand that has 1s in its bit positions.

The **XOR** instruction can complement bits of an operand. For Boolean variable a, the relationship
'a + 1 = a' and 'a + 0 = a' indicates that the binary variable is complemented when XOR ed with 1.
However, the variable does not change value when XOR ed with 0.

The carry bits can be cleared, set, or complemented with appropriate instructions. The bit manipulation instructions can also enable or disable the interrupt facility, which is controlled by the flip-flops.

Shift Instructions

Shift instruction helps to shift the bits of an operand to the right or to the left. The direction of shift is based on specific instructions. The operand is first loaded into the accumulator and then the shift operation is performed bit by bit.

The shift-left operation shifts the zero into low-order vacated position.

Example: Operand in Accumulator

1111 0101 1000 1011

After a shift-left operation

1 1110 1011 0001 0110 zero is shifted in

High order bit is shifted out

In shift-right operations, zeros are shifted into high-order vacated position. The bits that are shifted can also be the original value of the sign bit as in case of arithmetic right shift or can be the bits that are shifted out of low-order position of the accumulator-extension as in case of **Rotate Right Accumulator and Extension (RRAE).** The main purpose of this RRAE is to fetch the bits from the accumulator-extension position 15 and shift the bits back to the accumulator position 0. This ensures that no bits are lost during the shift.

The shift operation can be ended either by decrementing the shift count to zero or by shifting the bit value of 1 into the high-order position (bit 0) of the accumulator.

Unit – IV:

Input-Output Organization: *Peripheral Devices, Input Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt, Direct memory Access, Input–Output Processor (IOP), Intel 8089 IOP.*

Introduction

We are aware that computer organization refers to operational units and their interconnections that conform to the architecture specifications. A computer is a complex system that contains millions of elementary electronic components. A computer serves no purpose unless it communicates with the external environment. A user provides instructions to the computer through input devices, and the input data from the user after processing is stored in the memory. The computational results are given to the user through an output device.

11.1 Peripheral Devices

Peripheral devices are devices that are connected either internally or externally to a computer. These devices are commonly used to transfer data. The most common processes that are carried out in a computer are entering data and displaying processed data. Several devices can be used to receive data and display processed data. The devices used to perform these functions are called as peripherals or I/O devices.

Peripherals read information from or write in the memory unit on receiving a command from the CPU. They are considered to be a part of the total computer system. As they require a conversion of signal values, these devices can be referred to as electromechanical and electromagnetic devices. The most common peripherals are printer, scanner, keyboard, mouse, tape device, microphone and external modem that are externally connected to the computer.

The following are some of the commonly used peripherals:

Keyboard

Keyboard is the most commonly used input device. It is used to provide commands to the computer. The commands are usually in the form of text. The keyboard consists of many keys such as function keys, numeric keypad, character keys, and various symbols.

Monitor

The most commonly used output device is the monitor. A cable connects the monitor to the video adapters in the computer's motherboard. These video adapters convert the electrical signals to the text and images that are displayed. The images on the monitor are made of thousands of pixels.

The cursor is the characteristic feature of display devices. It marks the position on the screen where the next character will be inserted.

Printer

Printers provide a permanent record of computer data or text on paper. We can classify printers as impact and non-impact printers. Impact printers print characters due to the physical contact of the print head with the paper. In non-impact printers, there is no physical contact. Some examples of printers are:

1. Daisywheel
2. Dot Matrix
3. Laser Printer

Let us understand in detail about these printers:

1. **Daisywheel:** The daisywheel consists of a wheel with characters placed along its circumference. To print a character, the wheel rotates to the required position, and an energized magnet presses the letter against the ribbon.
2. **Dot Matrix:** The dot matrix printer has a set of dots along the printing mechanism. The decision to print a dot depends on the specific characters that are printed on the line.

Magnetic Tape

Magnetic tapes are used in most companies to store data files.

Magnetic tapes use a read-write mechanism. The read-write mechanism refers to writing data on or reading data from a magnetic tape. The tapes store the data in a sequential manner. In this sequential processing, the computer must begin searching at the beginning and check each record until the desired data is available. As the tape moves along the stationary read-write mechanism, the access is sequential and the records are accessed one after another. The magnetic tape is the cheapest medium for storage because it can store large number of binary digits, bytes, or frames on every inch of the tape. The advantages of using magnetic tape include unlimited storage, low cost, high data density, rapid transfer rate, portability and ease of use.

Magnetic Disk

Another medium for storing data is magnetic disks. Magnetic disks have high speed rotational surfaces coated with magnetic material. A read-write mechanism is used to achieve the access to write on or read from the magnetic disk. Magnetic disks are mostly used for bulk storage of programs and data.

There are many other peripheral devices found in computer systems such as digital incremental plotters, optical and magnetic readers, analog to digital converters, and various data acquisition equipment.

The following section deals with the method used to transfer information from the computer to the peripherals.

ASCII Alphanumeric Characters

Alphanumeric characters are used for the transfer of information to and from the I/O devices and the computer. American Standard Code for Information Interchange (ASCII) is the standard binary code used to represent alphanumeric characters. This standard uses seven bits to code 128 characters. However, there is an additional bit on the left that is always assigned 0. Therefore, there are 8 bits in total.

The control characters are used to route the data and arrange the printed text into a prescribed format.

Table 11.1: Control Character and Description

Control Character	Description
NUL	Null
SOH	Start of Heading
STX	Start of text
EOT	End of transmission
ENQ	Enquiry
ACK	Acknowledge
DLE	Data Link Escape
ETB	End of transmission block
EM	End of medium

Table 11.1: Control Character and Description

There are three types of control characters. They are:

1. Format Effectors
2. Information Separators
3. Communication Control Characters

The functions of these control characters are:

1. **Format Effectors:** They control the layout of printing. They include familiar typewrite controls such as Back Space (BS), Horizontal Tabulation (HT), and Carriage Return (CR).
2. **Information Separators:** They separate the data into divisions such as paragraphs and pages. They include Record Separator (RS) and File Separator (FS).
3. **Communication Control:** They are used during the transmission of text between remote terminals.

I/O Interface

The concept of I/O interface helps us understand how devices intercommunicate. I/O interface provides a method by which information is transferred between internal storage and external I/ O devices. All the peripherals connected to a computer require special communication connections for interfacing them with the CPU. The importance of communication connections is to resolve the differences that occur between the computer and each peripheral.

Some of the major differences are:

1. The operation of electromagnetic devices and peripherals are different from the operation of CPU and memory, which are electronic devices. There is a need for conversion of signal values to resolve this difference.
2. The transfer rate of CPU is faster than the transfer rate of the peripherals. Additionally, the peripherals require a synchronization mechanism to transfer the data.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. Each operating mode is different and each mode must be controlled to prevent disturbing the operation of the peripheral devices connected to the CPU.

Therefore, computer systems include special hardware components between the CPU and peripherals to resolve these differences. This special hardware supervises and synchronizes all input and output transfers. They are named interface units as they interface the processor bus and peripheral bus. Each device also has its own controller that supervises the operations of the

The I/O bus is the pathway used for peripheral devices to communicate with the computer processor. A typical connection of I/O bus to I/O devices is shown in figure 11.1.

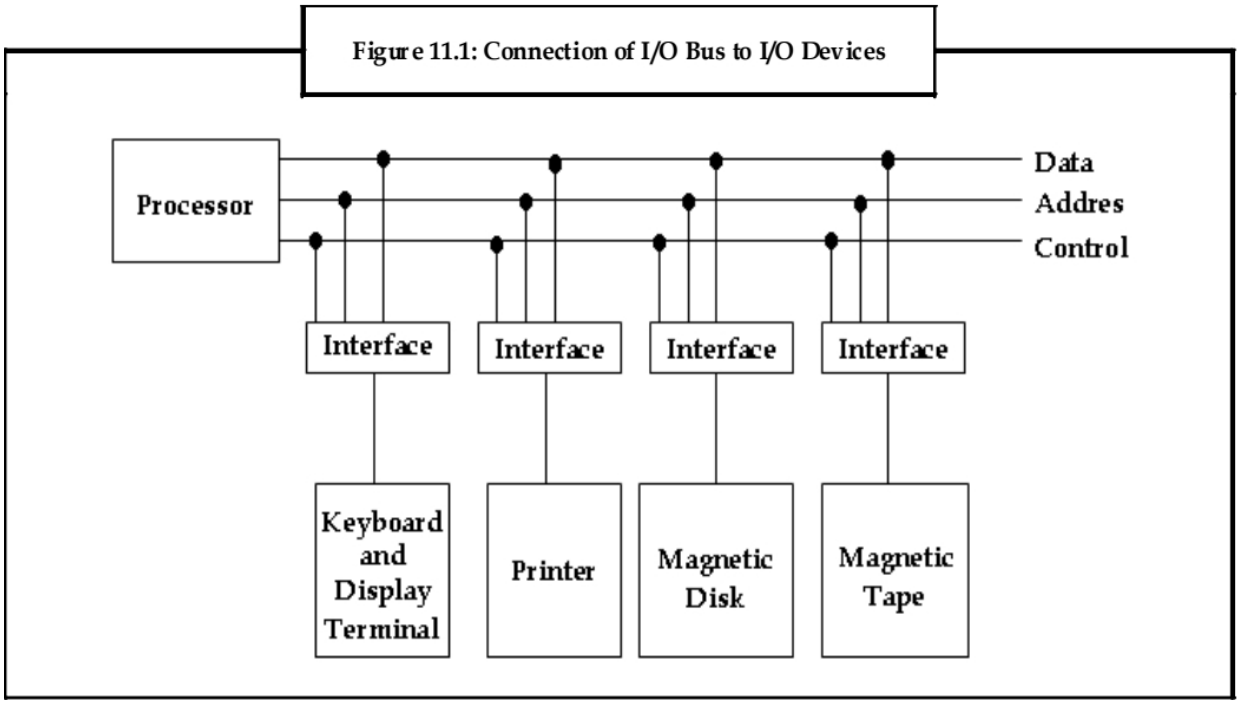


Figure 11.1: Connection of I/O Bus to I/O Devices

The I/O bus consists of data lines, address lines, and control lines. In any general purpose computer, the magnetic disk, printer, and keyboard and display terminal are commonly employed. Each peripheral unit has an interface unit associated with it. Each interface decodes the control and address received from the I/O bus. It interprets the address and control received from the peripheral, and provides signals for the peripheral controller. It also supervises the transfer of data between peripheral and processor and also synchronizes the data flow.

The I/O bus is connected to all peripheral interfaces from the processor. The processor places a device address on the address line to communicate with a particular device. Each interface contains an address decoder attached to the I/O bus that monitors the address lines. When the address is detected by the interface, it activates the path between the bus lines and the device that it controls. The interface disables the peripherals whose address does not correspond to the address in the bus.
At the same time, when the address is made available in the address lines, the processor provides a function code in the control lines. The interface that is selected replies to the function code and executes it. This function code is referred to as an I/O command. This also corresponds to an

instruction that is executed in the interface and the peripheral unit connected to that interface. An interface receives any of the following four commands:

1. Control
2. Status
3. Data Output
4. Data Input

These interface commands are used for the following purposes:

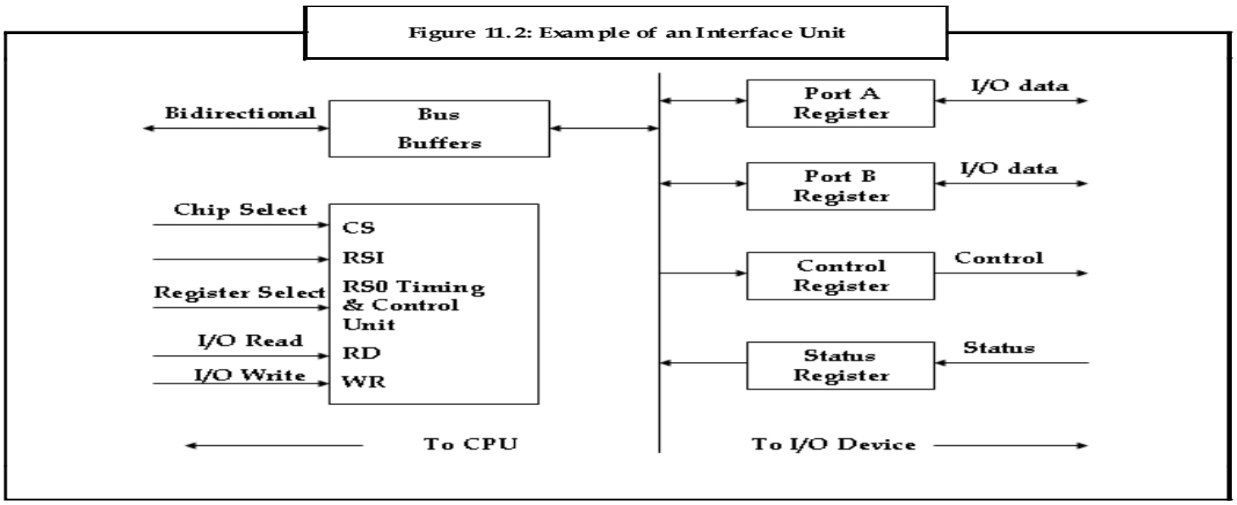
1. **Control:** A command control is given to activate the peripheral and to inform its next task. This control command depends on the peripheral, and each peripheral receives its own sequence of control commands, depending on its mode of operation.
2. **Status:** A status command is used to test different test conditions in the interface and the peripheral.
3. **Data Output:** A data output command makes the interface respond to the command by transferring data from the bus to one of its registers.
4. **Data Input:** The data input command is opposite to the data output command. In data input, the interface receives an item of data from the peripheral and places it in its buffer register. Later, the processor checks if the data is available by means of a status command and then issues a data input command.

I/O Versus Memory Bus

In addition to communicating with I/O devices, the processor must also communicate with the memory unit. Similar to the I/O bus, the memory bus contains independent sets of data, address, and control lines. There are three ways for buses to communicate with memory and I/O. They are:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O, but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

Figure 11.2: Example of an Interface Unit



In figure 11.2, the I/O interface consists of two data registers, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The Chip Select (CS) and the Register Select (RS) inputs determine the address assigned to the interface. The I/O read and I/O write are control lines that specify input and output respectively. There are four registers that communicate with the I/O device attached to the interface.

The I/O data from the device can be transferred to either port A or port B. There is a magnetic disk unit that transfers data in both directions but not at the same time. In this kind of system, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data is transferred between ports A and B.

The control register receives control information from the CPU. The interface and the I/O device attached to it can be placed in various operating modes by loading the corresponding bits into the control register.

Table 11.2 shows a list of control characters.

Table 11.2: List of Control Characters

CS	RS_1	RS_0	Register Selected
0	X	X	None
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

The interface registers communicate with the CPU through the bidirectional data bus. The address bus selects the interface unit through the Chip Select (CS) and the two Register Select (RS0 and RS1) inputs. A circuit is provided externally to detect the address assigned to the interface registers. This enables the CS input when the interface is selected. RS1 and RS0 are usually connected to the least significant lines of the address bus. These two bits select any of the two inputs listed in table 7.2. The content from the selected register is transferred to the CPU through the data bus when I/ O read signal is enabled.

Data Transfer Schemes

It is important to know the data transfer schemes because they provide an efficient means of transmitting data between the processing unit and the I/O devices. In a computer, the data transfer happens between any of these combinations — CPU and memory, CPU and I/O devices, and memory and I/O devices. A computer is interfaced with many devices of different speeds. Therefore, I/O devices may not be ready to transfer data as soon as the microprocessor issues the instruction for this purpose. Many data transfer schemes have been developed to solve this problem. The data transfer schemes have been broadly classified into two categories:

1. Programmed data transfer schemes
2. Direct Memory Access (DMA) data transfer scheme

Let us now discuss these data transfer schemes.

Programmed Data Transfer Schemes

In programmed data transfer scheme, data transfer takes place between the CPU and I/O device under the control of a program that resides in the memory. In this scheme, the program is executed by the CPU. This scheme is used when a small amount of data is to be transferred. The three important types of programmed data transfer schemes are:

1. **Synchronous Data Transfer Scheme:** This type of programmed data transfer scheme is used when the processor and the I/O devices match in speed. Some suitable instructions such as IN and OUT are used for 'to and from' data transfer of I/O devices.
2. **Asynchronous Data Transfer Scheme:** This type of programmed data transfer scheme is used when the speeds of I/O devices and the microprocessor do not match and also when the timing characteristics of the I/O devices are not predictable.
3. **Interrupt Driven Data Transfer Scheme:** In this programmed data transfer scheme, the processor enables the I/O devices and then continues to execute its original program instead of wasting time on checking the status of the I/O devices. When the I/O devices are ready to send and receive data, then the processor is informed through a specific control line called the 'Interrupt line'.

DMA Data Transfer Scheme

In DMA data transfer, data is directly transferred from the memory to the I/O device or vice versa without going through the microprocessor. This scheme is used when there is a need to transfer bulk data. Transferring bulk data using a microprocessor consumes more time. Therefore, the microprocessor performs the data transfer between an I/O device and memory using this DMA technique. For a DMA transfer, I/O devices must also contain an electronic circuitry to solve this problem, manufacturers have developed a single chip programmable DMA controller to interface I/O devices with the microprocessor for DMA transfer.

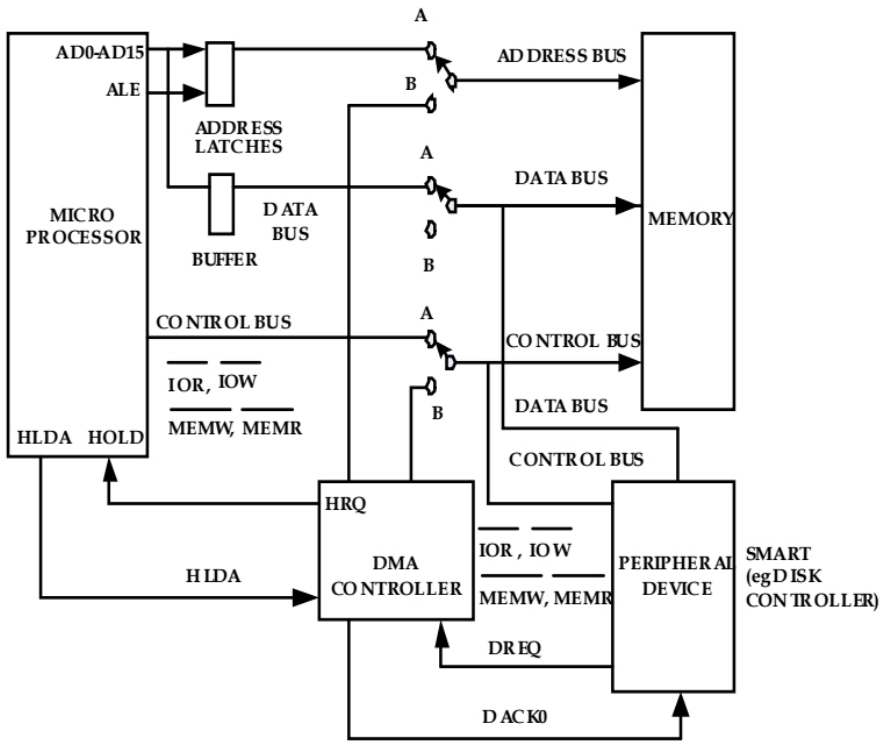
Direct Memory Access

Direct Memory Access (DMA) is a hardware controlled data transfer technique. An external device is used to control data transfer. External device generates address and control signals that are required to control data transfer. External devices also allow peripheral devices to directly access memory. The external device which controls the data transfer is called the DMA controller.

DMA Idle Cycle

When the system is turned on, the switches are in position A. The processor starts executing the program until it needs to read a block of data from the disk. The disk processor sends a series of commands to the disk controller to search and read the desired block of data from the disk. When the disk controller is ready to send the data from the disk, it sends DMA request (DRQ) signal to the DMA controller. Then the DMA controller sends a HOLD signal to the processor HOLD input. The processor responds to this signal by suspending the buses and sending an HLDA acknowledgement signal. When the DMA controller receives the HLDA signal, it sends a control signal to change the switch position from A to B.

Figure 11.8: DMA Controller Operating in Microprocessor System



DMA Active Cycle

When the DMA controller gets control of the buses, it sends the memory address where the first byte of data from the disk is to be written. It also sends a DMA acknowledge (DACK) signal to the disk controller device signaling it to get ready to send the output byte. Then it asserts both the IOR and MEMW signals on the control bus. IOR enables the disk controller to yield the byte of data from the disk and MEMW signal enables the addressed memory to accept data from the data bus.

Cycle Stealing Mode

In this data transfer mode, the device can make only one transfer (byte or word). After each transfer, DMAC gives the control of all buses to the processor. This is a single transfer mode with the process as follows:

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the processor.
3. The processor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts DACK to the requesting I/O device and executes DMA bus cycle, resulting in data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts DACK line.
7. The word/byte transfer count is decremented and the memory address is incremented.

8. The HOLD line is deasserted to give control of all buses back to the processor.
9. HOLD signal is reasserted to request the use of buses when I/O device is ready to transfer another byte or word. The same process is then repeated until the last transfer.

I/O Processor

Sometimes it is necessary for a processor to have some special features like controlling the I/O operations along with the ability to execute I/O instructions according to the requirements. Therefore, it is necessary to study the features of I/O processors (IOPs). An I/O processor is an I/ O channel that has a special-purpose processor. This processor has the ability to execute I/O instructions and it can also have control over I/O operation. The I/O instructions are stored in the main memory. The CPU initiates an I/O transfer by instructing the I/O channel to execute an I/O program when I/O transfer is required. The I/O program specifies the area of memory storage priority and action that needs to be taken for some conditions.

Features

1. An IOP can fetch and execute its own instructions.
2. Instructions are specially designed for I/O processing.
3. In addition to data transfer, an IOP can perform arithmetic and logic operations, branches, searches, and translations.
4. IOP performs all work involved in I/O transfer including device setup, programmed I/O, and DMA operation.
5. IOP can transfer data from an 8-bit source to a 16-bit destination and vice versa.
6. Memory-based control blocks are used for communication between IOP and CPU.
7. IOP supports multiprocessing environment. Both IOP and CPU can process simultaneously.

Memory Organization: *Memory Hierarchy, Auxiliary memory, Associate Memory, Cache Memory, Virtual Memory, Memory Management Hardware.*

Memory Hierarchy

You may be aware that the data required to operate a computer is stored in the hard drive. However, other storage methods are also required. This requirement is for a number of reasons, but mostly it is because the hard drive is slow and executing programs using it could be impractical. When the processor requires data or functions, it first fetches the required data from the hard drive and then loads them into the main memory (RAM). This increases the operation speed and programs are executed faster.

Main memory and the hard drive form two levels of the computer's memory hierarchy. In memory hierarchy, the storage devices are arranged in such a way that they take advantage of the characteristics of different storage technologies in order to improve the overall performance of a computer system.

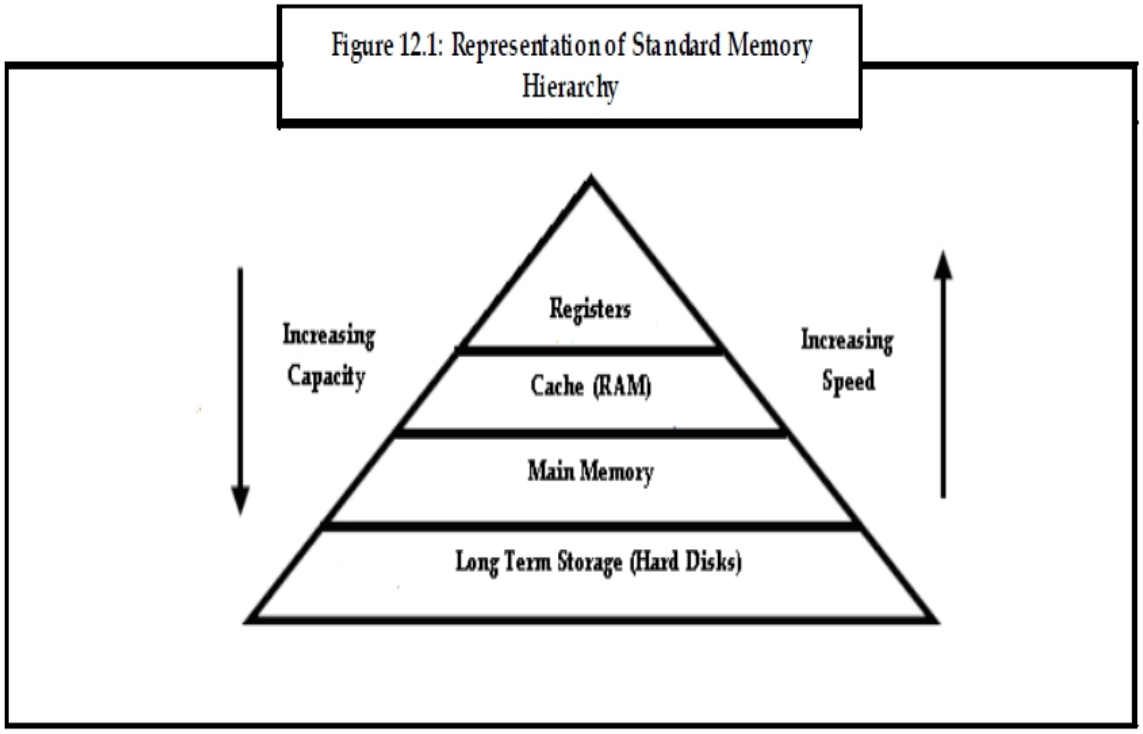


Figure shows the standard memory hierarchy of a computer.

The memory unit is an essential component of any digital computer, because all the programs and data are stored here. A very small computer with a limited purpose may be able to accomplish its intended task without the need for additional storage capacity. Nearly all conventional computers would run more efficiently if they were provided with additional storage space, excluding the capacity of the main memory. It is not possible for one memory unit to accommodate all the programs used in a conventional computer due to lack of storage space. Additionally, almost all computer users collect and continue to amass large amount of data processing software. Not all information that is gathered is needed by the processor at the same time. Thus, it is more appropriate to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU.

The memory unit that exchanges information directly with the CPU is called the main memory. Devices that support backup storage are referred to as auxiliary memory. The most common devices that support storage in typical computer systems are magnetic disks and magnetic tapes. They are mainly used to store system programs, large data files, and other backup information. Only programs and data that are currently required by the processor reside in the main memory. All other information is stored in auxiliary/secondary memory and moved to the main memory when needed.

The overall memory capacity of a computer can be pictured as being a hierarchy of components. The memory hierarchy system includes all storage devices used in a computer system. They range from the slow but large capacity auxiliary memory to a comparatively faster main memory, to an even smaller but faster cache memory accessible to the high-speed processing logic.

Figure: Memory Hierarchy in a Computer Sys

Placed at the bottom of the hierarchy is the comparatively slow magnetic tape used to store removable files. Next is the magnetic disk that is used as backup storage.

The main memory takes up a central position because of its ability to communicate directly with the CPU and with auxiliary memory devices, through an Input/Output (I/O) processor. When the CPU needs programs that are not present in the main memory, they are brought in from the auxiliary memory. Programs that are not currently required in the main memory are moved into the auxiliary memory to provide space for currently used programs and data.

To increase the speed of processing, a very-high-speed memory, known as cache, is used. It helps in making the current data and programs available to the CPU at high speed. The cache memory is used in computer systems to compensate for the difference between the main memory access time and processor logic speed. CPU logic is quicker than main memory access time. Yet, the processing speed of the CPU is limited by the main memory's speed. The difference in these operating speeds can be balanced by using an extremely fast, small cache between the CPU and main memory. The cache access time is nearly equal to the processor logic clock cycle time. The cache is used to store the following:

1. Fragments of program that is presently being executed in the CPU
2. Temporary data that is repeatedly needed in the current operation

When programs and data are available at a high rate, it is possible to increase the performance rate of the computer.

Although the I/O processor manages data transfers between main memory and auxiliary memory, the cache organization deals with the transfer of information between the main memory and the CPU. Thus, the cache and the CPU interact at different levels in the memory hierarchy system. The main reason for having two or three levels of memory hierarchy is to achieve cost-effectiveness. As the storage capacity of the memory increases, the cost-per-bit of storing binary information decreases and the memory access time increases. When compared to main memory, the auxiliary memory has a large storage capacity and is relatively inexpensive. However, the auxiliary memory has low access speed. The cache memory is very small, quite expensive, and has very high access speed. Consequently, as the memory access speed increases, so does its relative cost. The main purpose of

employing a memory hierarchy is to achieve the optimum average access speed while minimizing the total cost of the entire memory system.

Main Memory

The main memory is the fundamental storage unit in a computer system. It is a relatively large and fast memory, and stores programs and data during computer operations. The technology that makes the main memory work is based on semiconductor integrated circuits.

As mentioned earlier, RAM is the main memory. Integrated circuit Random Access Memory (RAM) chips are available in two possible operating modes. They are:

1. **Static:** It basically consists of internal flip-flops, which store the binary information. The stored information remains valid as long as power is supplied to the unit. The static RAM is simple to use and has shorter read and write cycles.

2.**Dynamic:** It stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are made available inside the chip by Metal Oxide Semiconductor (MOS) transistors. The stored charge on the capacitors tends to discharge with time and thus, the capacitors must be regularly recharged by refreshing the dynamic memory. Refreshing is done by progressively supplying the capacitor with words every few milliseconds to restore the decaying charge. The dynamic RAM uses minimum power and provides ample storage capacity in a single memory chip.

Most of the main memory in a computer is typically made up of RAM integrated circuit chips, but a part of the memory may be built with Read Only Memory (ROM) chips. Formerly, RAM was used to refer to a random-access memory. But now it is used to address a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the volumes of programs and data that are subject to change. ROM is used for storing programs that permanently reside in the computer. It is also used for storing tables of constants whose value does not change once the computer is constructed.

RAM and ROM chips are available in a variety of sizes. If the memory requirement for the computer

Random Access Memory

The term Random Access Memory or RAM is typically used to refer to memory that is easily read from and written to by the microprocessor. In reality, it is not right to use this term. For a memory to be called random access, it should be possible to access any address at any time. This differentiates RAM from storage devices such as tapes or hard drives where the data is accessed sequentially.

Practically, RAM is the main memory of a computer. Its objective is to store data and applications that are currently in use. The operating system controls the usage of this memory. It gives instructions like when the items are to be loaded into RAM, where they are to be located in RAM, and when they need to be removed from RAM. RAM is intended to be very fast both for reading and writing data. RAM also tends to be volatile, that is, all the data is lost as soon as power is cut off.

Read Only Memory

In every computer system, there must be a segment of memory that is stable and unaffected by power loss. This kind of memory is called Read Only Memory or ROM. Once again, this is not the right term. If it was not possible to write to this type of memory, it would not have been possible to store the code or data that is to be contained in it. It simply indicates that without special mechanisms in place, a processor may not be able to write to this type of memory.

SRAM

RAMs that are made up of circuits and can preserve the information as long as power is supplied are referred to as Static Random Access Memories (SRAM). Flip-flops form the basic memory elements in a SRAM device. A SRAM consists of an array of flip-flops, one for each bit.

Since SRAM consists of an array of flip-flops, a large number of flip-flops are needed to provide higher capacity memory. Because of this, simpler flip-flop circuits, BJT and MOS transistors are used for SRAM. This helps to save chip area and provides memory integrated circuits at relatively reduced cost, increased speed and reduces the power dissipation as well. SRAMs have very short access times typically less than 10 ns. SRAMs with battery backup are commonly used to provide stability to data during power loss.

DRAM

SRAMs are faster but their cost is high, because their cells require many transistors. RAMs can be obtained at a lower cost if simpler cells are used. A MOS storage cell based on capacitors can be used to replace the SRAM cells. Such a storage cell cannot preserve the charge (that is, data) indefinitely and must be recharged periodically. Therefore, these cells are called as dynamic storage cells. RAMs using these cells are referred to as Dynamic RAMs or simply DRAMs.

Cache

Even with improvements in hard drive performance, it is still not practical to execute programs or access data directly from the mechanical devices like hard disk and magnetic tapes, because they are very slow. Therefore, when the processor needs to access data, it is first loaded from the hard drive into the main memory where the higher performance RAM allows fast access to the data. When the processor does not require the data anymore, it can either be discarded or used to update the hard drive.

The cost makes the capacity of a computer's main memory inadequate when compared to its hard drive. However, this would not have a major impact as all of the data on a hard drive need not be accessed all the time by the processor. Only the currently active data or programs need to be in RAM. Additional performance improvements can be achieved by taking this concept to another level.

As discussed earlier, there are two main classifications of RAM: Static RAM (SRAM) and Dynamic RAM (DRAM). SRAM is faster, but that speed comes at a high cost - it has a lower density and it is more expensive. Since main memory needs to be relatively large and inexpensive, it is implemented with DRAM.

Main memory improves the performance of the system by loading only the data that is currently required or in use from the hard drive. However, the system performance can be improved considerably by using a small, fast SRAM to store data and code that is in immediate use. The code and data that is not currently needed can be stored in the main memory.

You must be aware that in a programming language, instructions that are executed often tend to be clustered together. This is mainly due to the basic constructs of programming such as loops and subroutines. Therefore, when one instruction is executed, the chances of it or its surrounding instructions being executed again in the near future are high. Over a short interval, a cluster of instructions may execute over and over again. This is referred to as the principle of locality. Data also behaves as per this principle as related data is often stored in consecutive locations.

To benefit from this principle, a small, fast SRAM is placed between the processor/CPU and main memory to hold the most recently used instructions/programs and data under the belief that they will most likely be used again soon. This small, fast SRAM is called a RAM cache or just a cache. The location of a cache in a memory hierarchy is shown in Figure.

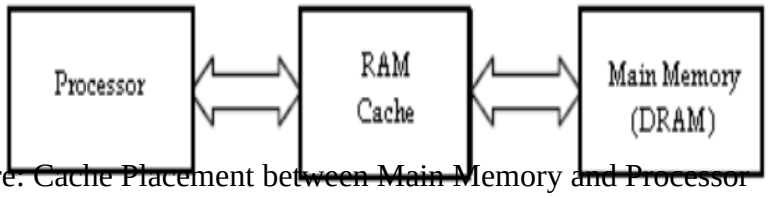


Figure: Cache Placement between Main Memory and Processor

The SRAM of the cache needs to be small, the reason being that the larger address decoder circuits are slower than small address decoder circuits. As the memory increases, the complexity of the address decoder circuit also increases. As the complexity of the address decoder circuit increases, the time taken to select a memory location based on the address it received also increases. For this reason, making a memory smaller makes it faster.

The concept of reducing the size of memory can be optimized by placing an even smaller SRAM

Notes between the cache and the processor, thereby creating two levels of cache. This new cache is usually

contained inside of the processor. As the new cache is put inside the processor, the wires connecting the two become very short, and the interface circuitry becomes more closely integrated with that of the processor. These two conditions together with the smaller decoder circuit facilitate faster data access. When two caches are present, the cache within the processor is referred to as a level 1 or L1 cache. The cache between the L1 cache and memory is referred to as a level 2 or L2 cache.
Figure 12.4 shows the placement of L1 and L2 cache in memory.

L1 and L2 Cache Placement

Figure: L1 and L2 Cache Placement

The split cache, another cache organization, is shown in figure 12.5. Split cache requires two caches. In this case, a processor uses one cache to store code/instructions and a second cache to store data. This cache organization is typically used to support an advanced type of processor architecture such as pipelining. Here, the mechanisms used by the processor to handle the code are so distinct from those used for data that it does not make sense to put both types of information into the same cache.

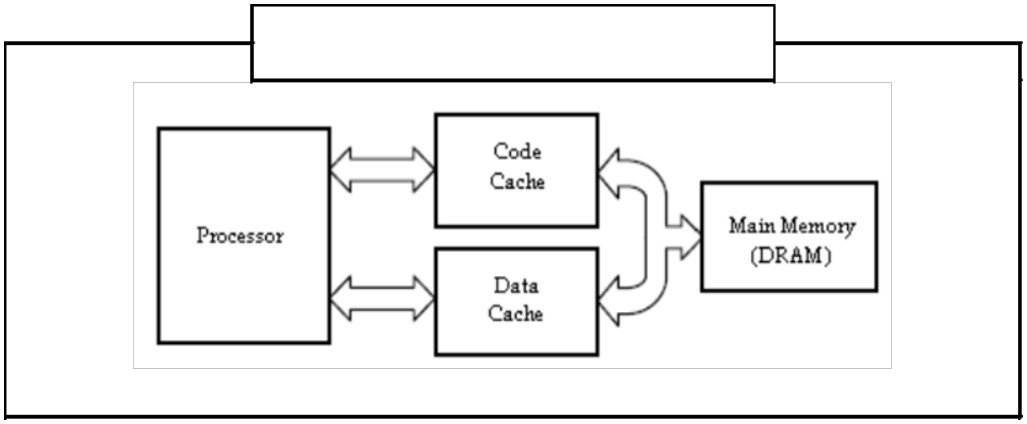


Figure: Split Cache Organization

The success of caches depends upon the principle of locality. The principle proposes that when one data item is loaded into a cache, the items close to it in memory should be loaded too.

If a program enters a loop, most of the instructions that are part of that loop are executed multiple times. Therefore, when the first instruction of a loop is being loaded into the cache, it loads its bordering instructions simultaneously to save time. In this way, the processor does not have to wait for the main memory to provide subsequent instructions. As a result of this, caches are organized in such a way that when one piece of data or code is loaded, the block of neighboring items are loaded too. Each block loaded into the cache is identified with a number known as a tag. This tag can be used to find the original addresses of the data in the main memory. Therefore, when the processor is in search of a piece of data or code (hereafter referred to as a word), it only needs to check the tags to see if the word is contained in the cache.

Each block of words and its corresponding tag is combined in the cache to form a line. The lines are structured into a table. When the main memory needs a word from within a block, the whole block is moved into one of the lines of the cache along with its tag, which is used to identify the address of the block.

The processor first checks the cache when it needs a word from the memory. If the word is not in the cache, a condition known as miss occurs. In order to ensure that no time is lost due to a miss, the process for accessing the same word from main memory is simultaneously activated. If the word is present in the cache, then the processor uses the cache's word and disregards the results from the main memory. This mechanism is referred to as a hit.

In case of a miss, the entire block containing the word is loaded into a line of the cache, and the word is sent to the processor. Depending on the design of the cache/processor interface, the word is either loaded into the cache first and then given to the processor or it is loaded into the cache and sent to the processor simultaneously. In the first case, the cache is controlled by the memory interface and lies between memory and the processor. In the second case, the cache acts like an extra memory on the same bus with the main memory.

Suppose the main memory is divided into n blocks and the cache has space to accommodate exactly m blocks. By virtue of the nature of the cache, m is much smaller than n. If we divide m into n , get an integer which illustrates the number of times that the main memory could fill the cache with different blocks from its contents.

The main memory is much larger than a cache, so each line in the cache is responsible for storing one of many blocks from main memory. In our above example, each line of the cache is responsible for storing one of 512 different blocks from main memory at a specific time.

The process of transferring data from main memory to cache memory is called as mapping. There are three methods used to map a line in the cache to an address in memory so that the processor can quickly find a word. They are:

1. Direct Mapping
2. Associative Mapping
3. Set Associative Mapping

Direct Mapping

Direct mapping is a procedure used to assign each memory block in main memory to a specific line in the cache. If a line is already filled with a memory block and a new block needs to be loaded, then the old block is discarded from the cache.

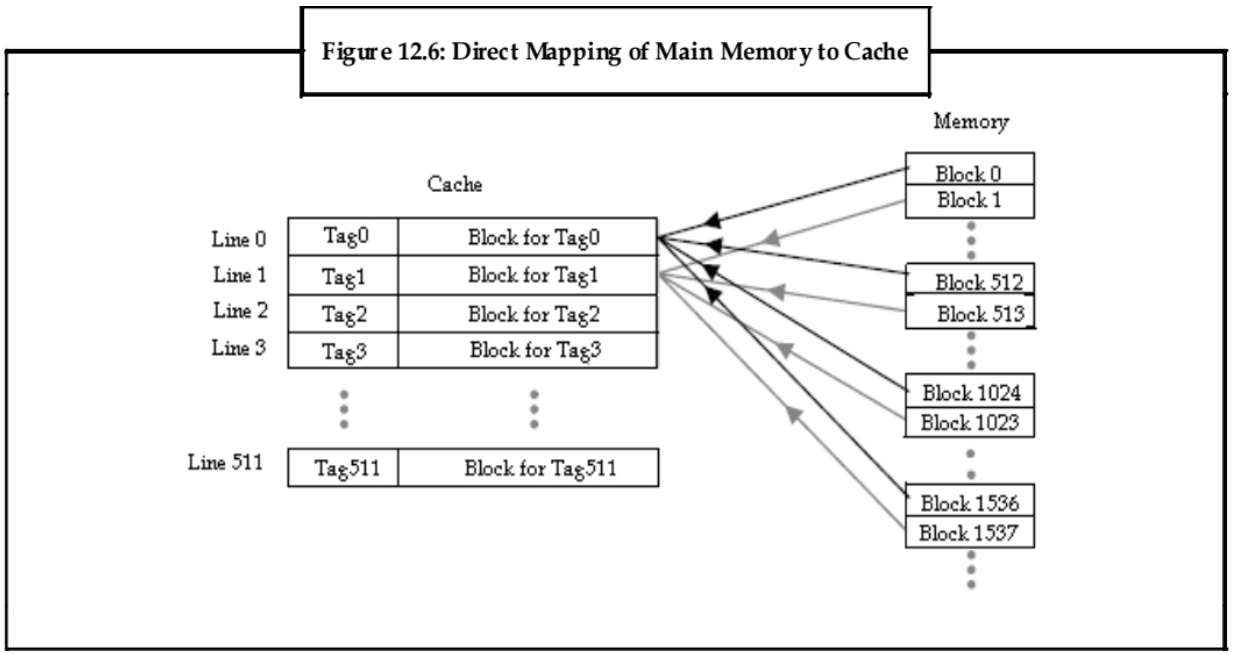


Figure: Direct Mapping of Main Memory to Cache

Just like locating a word within a block, bits are taken from the main memory address to uniquely describe the line in the cache where a block can be stored.

Example: Consider a cache with 29 = 512 lines, then a line would need 9 bits to be uniquely identified.

Therefore, the 9 bits of the address immediately to the left of the word identification bits would recognize the line in the cache where the block is to be stored. The bits of the address that were not used for the word offset or the cache line would be used for the tag. Figure 12.7 represents this partitioning of the bits.

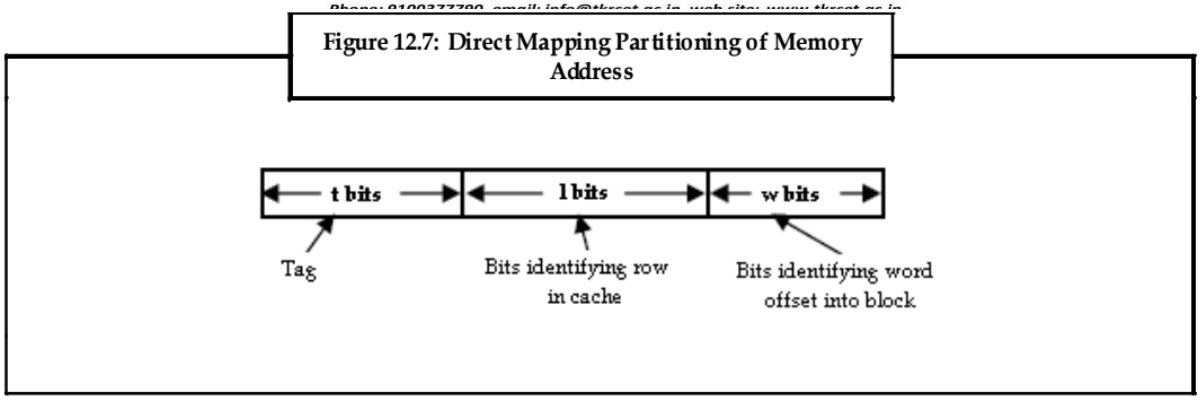


Figure Direct Mapping Partitioning of Memory Address

As soon as the block is stored in the line of the cache, the tag is copied to the tag location of the line. From the cache line number, the tag, and the word position within the block, the original address of the word can be reproduced.

In short, direct mapping divides an address into three parts: t tag bits, l line bits, and w word bits. The word bits are the least significant bits that identify the specific word within a block of memory. The line bits are the next least significant bits that identify the line of the cache in which the block is stored. The remaining bits are stored along with the block as the tag which locates the block's position in the main memory.

Associative mapping or fully associative mapping does not make use of line numbers. It breaks the main memory address into two parts - the word ID and a tag as shown in Figure 12.8. In order to check for a block stored in the memory, the tag is pulled from the memory address and a search is performed through all of the lines of the cache to see if the block is present.

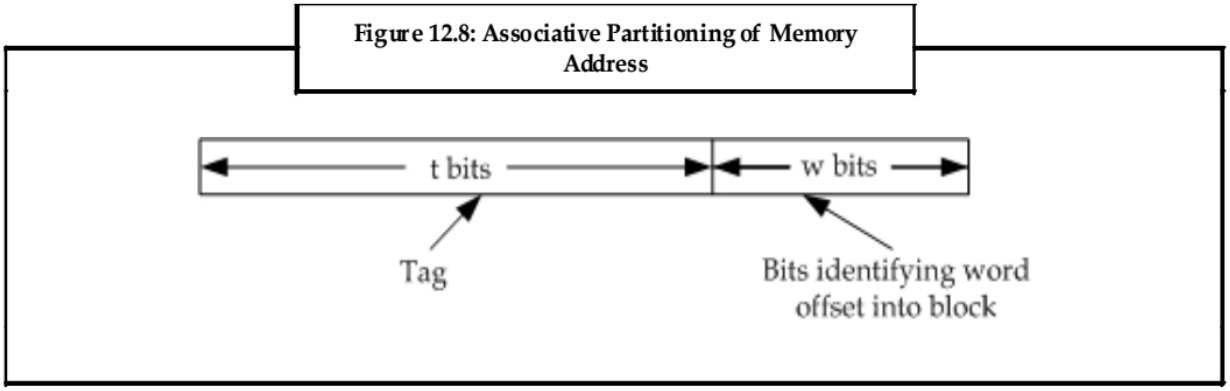


Figure: Associative Partitioning of Memory Address

This method of searching for a block within a cache appears like it might be a slow process, but it is not. Each line of the cache has its own compare circuitry, which can quickly analyze whether or not the block is contained at that line. With all of the lines performing this comparison process in parallel, the correct line is identified quickly.

This mapping technique is designed to solve a problem that exists with direct mapping where two active blocks of memory could map to the same line of the cache. When this happens, neither block of memory is allowed to stay in the cache as it is replaced quickly by the competing block. This leads

to a condition that is referred to as thrashing. In thrashing, a line in the cache goes back and forth between two or more blocks, usually replacing a block even before the processor goes through it. Thrashing can be avoided by allowing a block of memory to map to any line of the cache.

However, this advantage comes with a price. When an associative cache is full and the processor needs to load a new block from memory, a decision has to be made regarding which of the existing blocks should be discarded. The selection method, known as a replacement algorithm, should aim to replace the block least likely to be needed by the processor in the near future.

There are many replacement algorithms, none of which has any precedence over the others. In an attempt to realize the fastest operation, each of these algorithms is implemented in hardware.

1. Least Recently Used (LRU): This method replaces the block that has not been read by the processor in the longest period of time.
2. First In First Out (FIFO): This method replaces the block that has been in cache the longest.
3. Least Frequently Used (LFU): This method replaces the block which has had fewest hits since being loaded into the cache.
4. Random: This method randomly selects a block to be replaced. Its performance is slightly lower than LRU, FIFO, or LFU.

The objective of a replacement algorithm is to try to remove the page least likely to be referenced in the immediate future.

Set Associative Mapping

Set associative mapping merges direct mapping with fully associative mapping by grouping together lines of a cache into sets. The sets are determined using a direct mapping scheme. However, the lines within each set are considered as tiny fully associative cache where any block that is to be stored in the set can be stored to any line within the set.

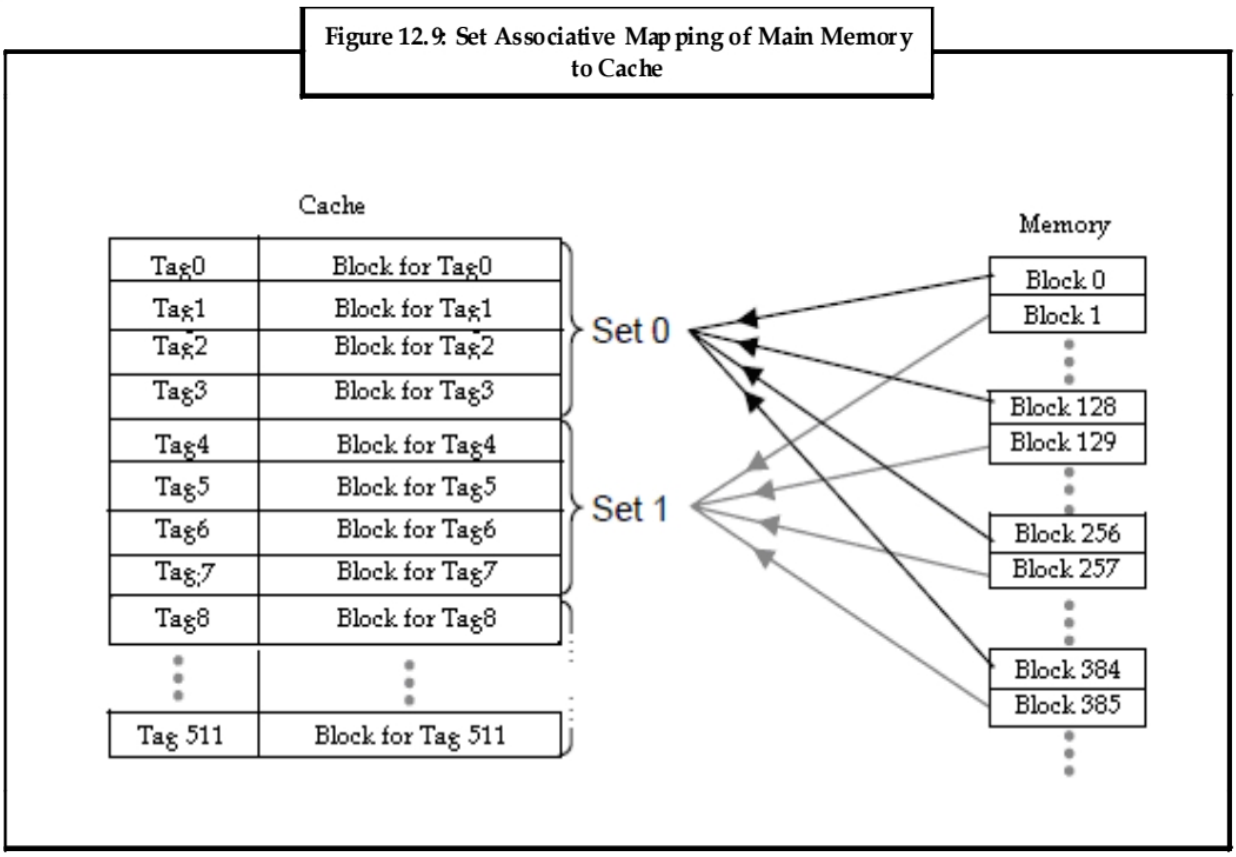


Figure: Set Associative Mapping of Main Memory to Cache

A set associative cache that contains k lines per set is called as a k way set associative cache. Since the mapping technique uses the memory address just like direct mapping does, the number of lines contained in a set must be equal to an integer power of two, for example, two, four, eight, sixteen, and so on.

By shifting from direct mapping to set associative with a set size of two lines per set, the number of sets obtained equals to half the number of lines. In the instance of the cache having 512 lines, we would obtain 256 sets of two lines each, which would need eight bits from the memory address to identify the set. This would leave 30 – 8 – 3 = 19 bits for the tag. By shifting to four lines per set, the number of sets is reduced to 128 sets requiring 7 bits to identify the set and twenty bits for the tag.

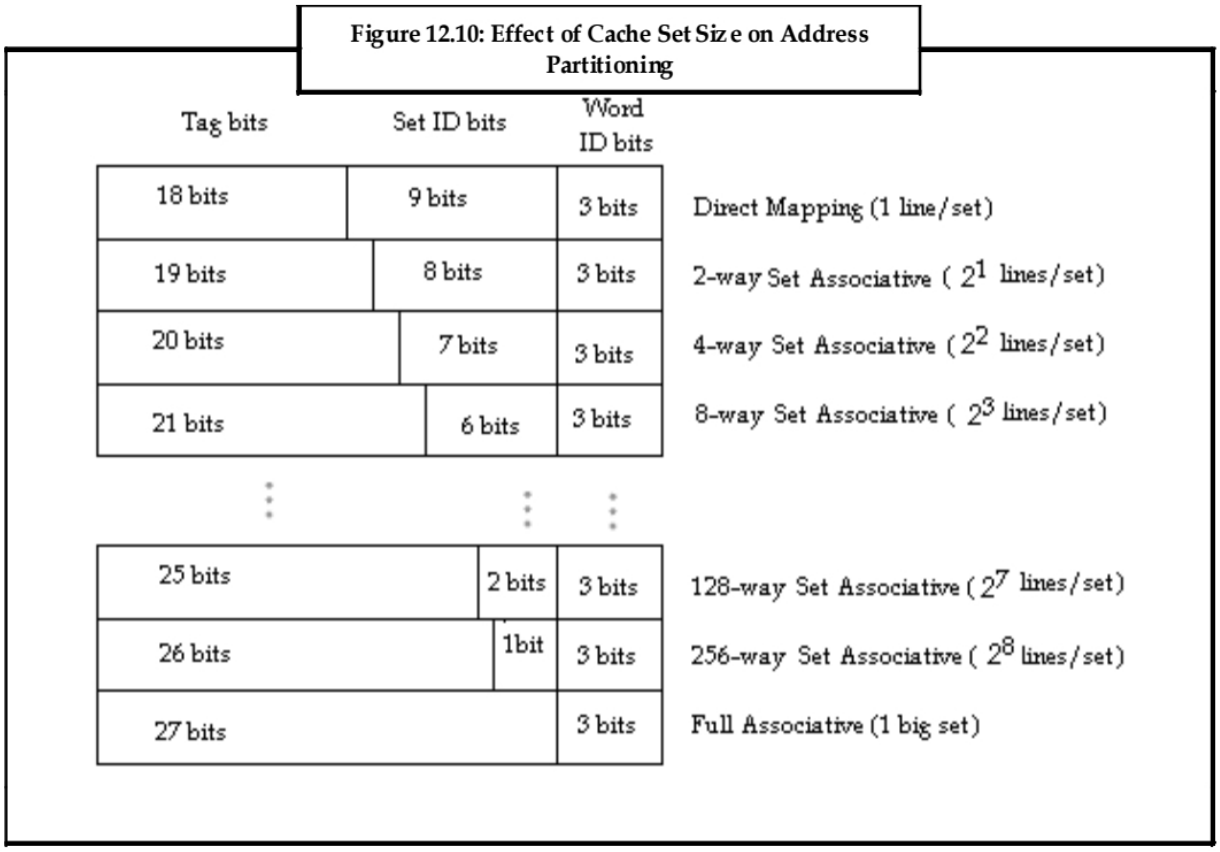


Figure: Effect of Cache Set Size on Address Partitioning

Whenever a block from memory has to be stored in a set already filled with other blocks, one of the replacement algorithms described for fully associative mapping is used.

Virtual Memory

In typical computer systems, data and programs are initially stored in auxiliary memory. Fragments of data or programs are brought into main memory as and when the CPU requires them. Virtual memory is a method or approach used in some large computer systems. It allows the user to construct programs as though a large memory space was available, which is equal to the whole of auxiliary memory. Every address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in the main memory. Virtual memory is made use of to give programmers the impression that they have a very large memory at their disposition, even though the computer actually has a relatively small main memory. A virtual memory system implements a mechanism that translates program-generated addresses into correct main memory locations. This translation happens dynamically even as programs are being executed in the CPU. The translation or mapping is automatically handled by the hardware by means of a mapping table.

Address Space and Memory Space

Addresses that are used by programmers are called virtual addresses, and the set of such addresses is called the address space. The space or spot where the address is stored in the main memory is called a location or physical address and the set of such locations is called the memory space. Therefore, the

address space is the set of addresses generated by programs as they reference instructions and data. The memory space holds the actual main memory locations that are directly addressable for processing. In most computers, the address and memory spaces are the same.

In a multiprogramming computer system, programs and data are transferred to and from auxiliary memory and main memory when required by the CPU. Suppose Program1 is currently being executed in the CPU, Program1 and a section of its associated data are transferred from auxiliary memory into main memory as shown in figure 12.11. The associated programs and data need not be in adjacent locations in the memory, since information is being moved in and out, and empty spaces may be scattered in the memory.

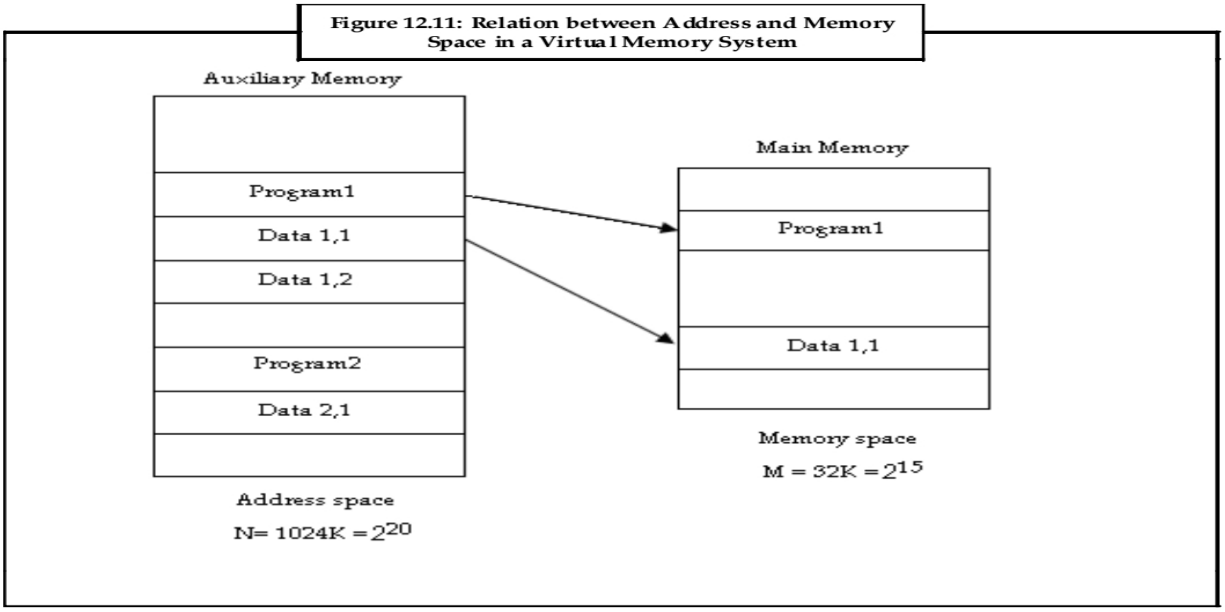


Figure: Relation between Address and Memory Space in a Virtual Memory System

In a virtual memory system, programmers are made to believe that they have the total address space for their use. Additionally, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. Suppose, the address field of an instruction code consists of 20 bits, but physical memory addresses can only be specified with 15 bits. As a result, CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words would be extremely long.

Thus, a table is needed to map a virtual address of say 20 bits to a physical address of say 15 bits. Mapping is a dynamic process, which means that every address is translated instantly as a word is referenced by CPU.

A separate memory or main memory may be used to store the mapping as shown in figure 12.12.

1. In the first case, an additional memory unit is needed along with one extra memory access time.

2. In the second case, the table takes space from main memory and two accesses to memory are required with the program running at half speed.
3. In the third case, an associative memory can be used. The associative mapping technique is discussed later.

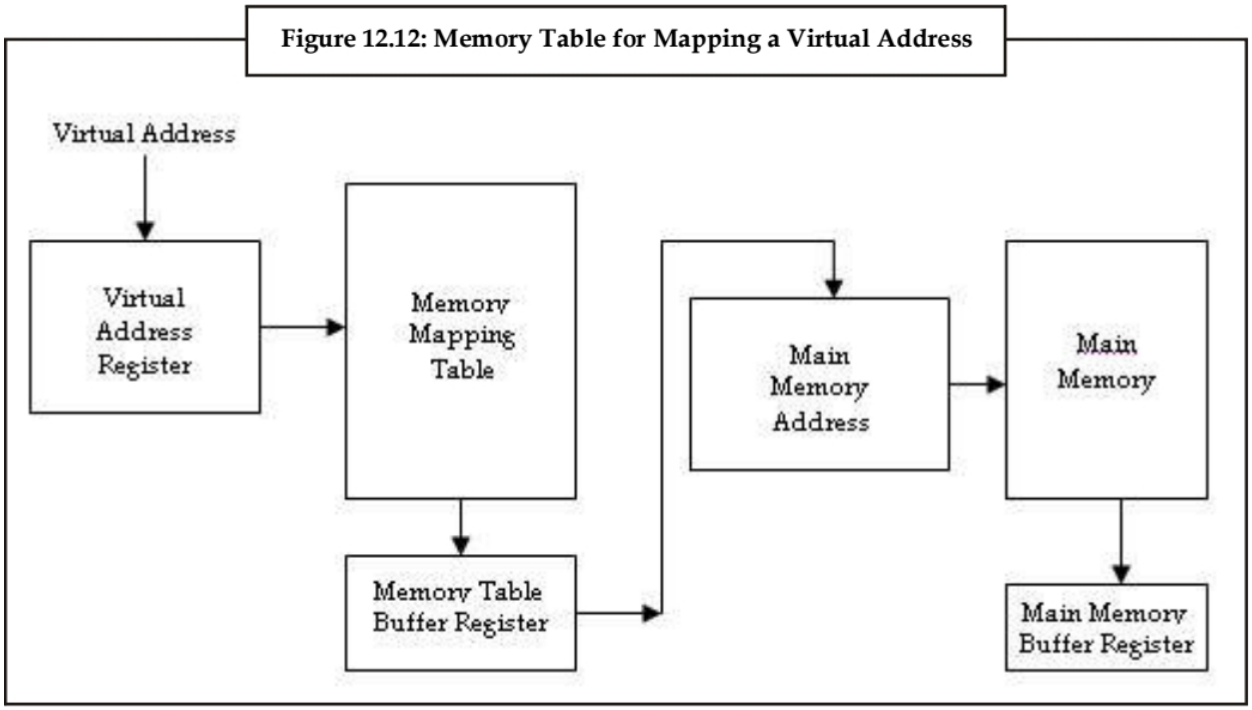


Figure: Memory Table for Mapping a Virtual Address

In figure 12.12, a virtual address of 20 bits is mapped and listed in the memory mapping table. It is then mapped to a 15 bit physical address which refers to a location in the main memory.

Address Mapping Using Pages

Presenting the address mapping in table form is simplified, if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is divided into clusters of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to clusters of address space of the same size.

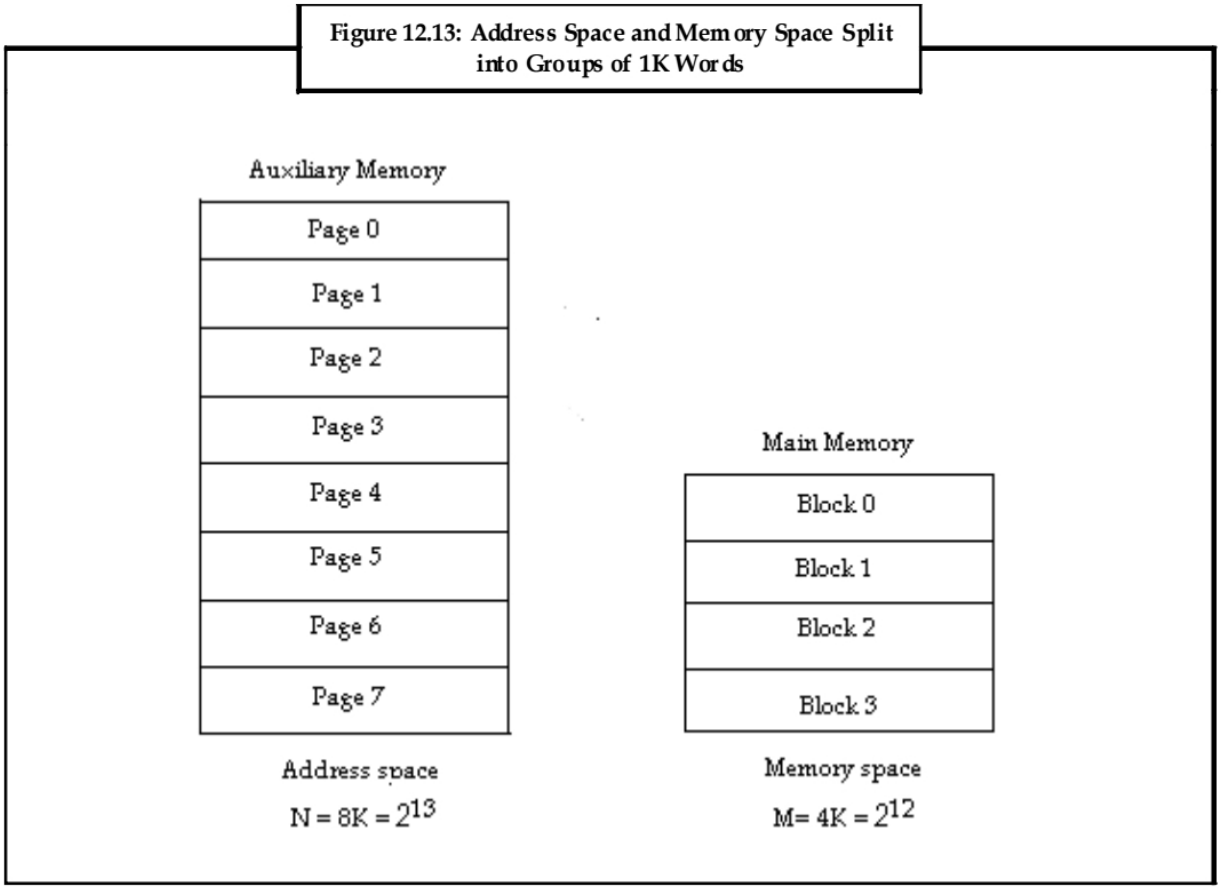


Figure: Address Space and Memory Space Split into Groups of 1K Words

Notes The line address is the same in address space as well as memory space; the only mapping needed is from a page number to a block number.

The structure of the memory mapping table in a paged system is shown in Figure 12.14. The memory-page table comprises eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5, and 6 are now present in the main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location signifies whether the page has been moved from auxiliary memory into main memory. Zero in the presence bit signifies that this page is not available in main memory. The CPU points to a word in memory with a virtual address of 13 bits. The three high-order bits of the virtual address specify a page number and also an address for the memory-page table.

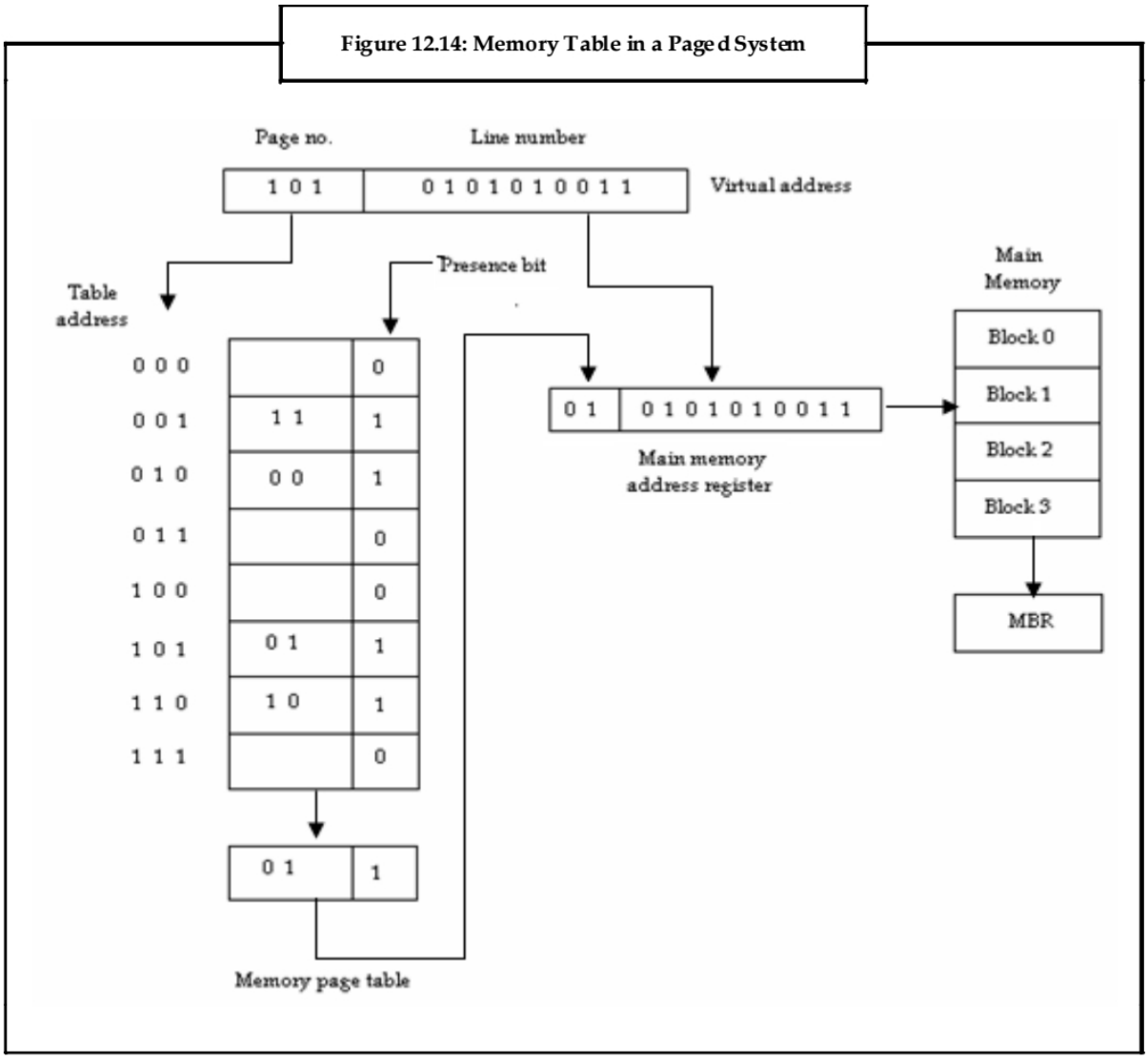


Figure: Memory Table in a Paged System

The content of the word in the memory page table at the page number address is copied into the memory table buffer register. If the presence bit is 1, the block number thus copied is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is moved into the 10 lower-order bits of the memory address register. A read signal to main memory moves the content of the word to the main memory buffer register that is ready to be used by the CPU. If the presence bit of the word copied from the page table is 0, it indicates that the content of the word referenced by the virtual address is not present in the main memory. A request to the operating system is then generated to get the required page from auxiliary memory and place it into main memory before resuming computation.

Associative Memory Page Table

A random-access memory page table is not appropriate when it comes to storage utilization. In the illustration in figure 12.14, we noticed that eight words of memory are needed, one for each page. But at least four words are always marked empty because the main memory cannot accommodate

more than four blocks. Normally, a system with m blocks and n pages would require a memory-page table of n locations of which, up to m blocks is marked with block numbers and all others are empty. A better approach towards organizing the page table would be to construct it with a number of words equal to the number of blocks in the main memory. In this way, the memory size is reduced and each location is fully utilized. This method can be implemented using an associative memory where each word in memory includes a page number together with its related block number. Each word's page field is compared with the page number present in the virtual address and if a match is found, the word is read from memory and its corresponding block number is determined.

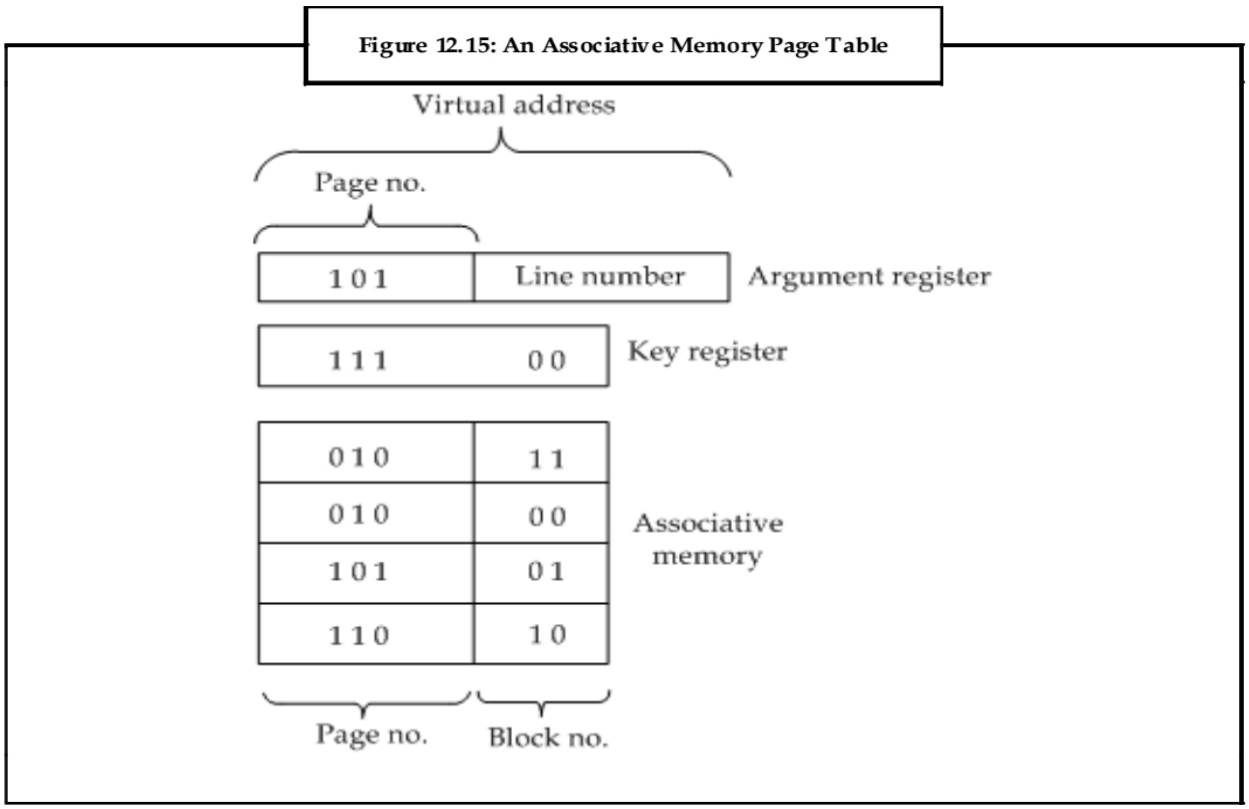


Figure: An Associative Memory Page Table

Let us consider the same case of eight pages and four blocks shown in the example of figure 12.14. If we replace the random access memory-page table with an associative memory of four words as shown in figure 12.15, then each entry in the associative memory array would consist of two fields. By observing the figure, you would find that the first three bits specify a field for storing the page number and the last two bits make up a field for storing the block number. The virtual address is stored in the argument register. The page number bits in the page field of the associative memory and the page numbers in the argument register are matched against each other. If a match is found, the 5-bit word is read out from memory and the related block number is transferred to the main memory address register. If no match occurs, a request to the operating system is generated to fetch the required page from auxiliary memory.

Page Replacement

A virtual memory organization is a combination of hardware and software systems. To make efficient utilization of memory space all the software operations are handled by the memory management software. The memory management software must decide on the following three things,

1. Which page in main memory must to be removed to create space for a new page?
2. When a new page is to be moved from auxiliary memory to main memory?
3. Where the page is to be located in main memory?

The hardware mapping system and the memory management software together form the architecture of a virtual memory.

When the program execution starts, one or more pages are moved into main memory and the page table is set to indicate their location. The program is executed from main memory until a reference is made for a page that is not in memory. This event is termed as page fault. When page fault occurs, the program that is currently in execution is stopped until the required page is moved into main memory. Since the act of loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In this interval, control is transferred to the next program in main memory that is waiting to be processed in the CPU. Soon after the memory block is assigned and then moved, the suspended program can resume execution.

If main memory is full, a new page cannot be moved in. Therefore, it would be necessary to remove a page from a memory block to accommodate the new page. The decision of removing specific pages from memory is determined by the replacement algorithm. The different replacement algorithms have been discussed briefly in the previous sections.