MODULE 1: BASIC STRUCTURE OF COMPUTERS

BASIC CONCEPTS

• **Computer Architecture (CA)** is concerned with the structure and behaviour of the computer.

• CA includes the information formats, the instruction set and techniques for addressing memory.

• In general covers, CA covers 3 aspects of computer-design namely: 1) Computer Hardware, 2) Instruction set Architecture and 3) Computer Organization.

1. Computer Hardware

➢ It consists of electronic circuits, displays, magnetic and optical storage media and communication facilities.

2. Instruction Set Architecture

➢ It is programmer visible machine interface such as instruction set, registers, memory organization and exception handling.

➢ Two main approaches are 1) CISC and 2) RISC.

(CISC→Complex Instruction Set Computer, RISC→Reduced Instruction Set Computer)

3. Computer Organization

➢ It includes the high level aspects of a design, such as

→ memory-system
→ bus-structure &
→ design of the internal CPU.

➢ It refers to the operational units and their interconnections that realize the architectural specifications.

➢ It describes the function of and design of the various units of digital computer that store and process information.

FUNCTIONAL UNITS

• A computer consists of 5 functionally independent main parts:

1) Input
2) Memory
3) ALU
4) Output &
5) Control units.

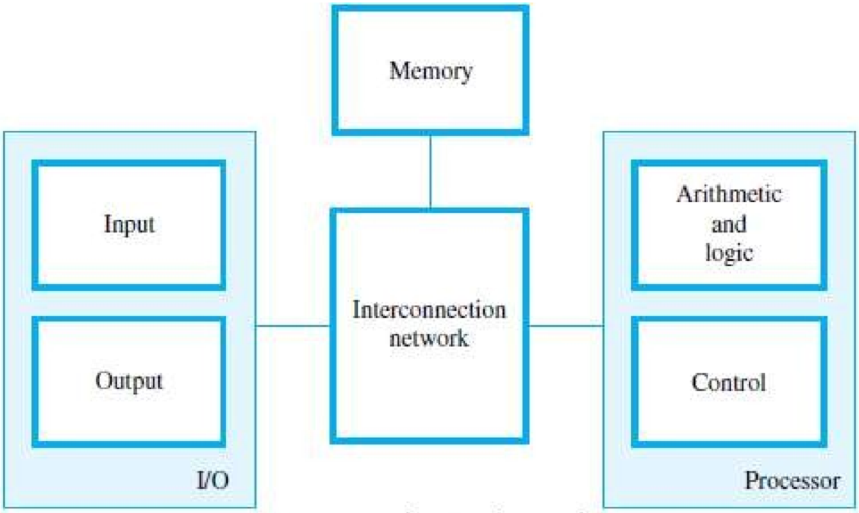


Figure 1.1 Basic functional units of a computer.

BASIC OPERATIONAL CONCEPTS

- An Instruction consists of 2 parts, 1) Operation code (Opcode) and 2) Operands.

OPCODE	*OPERANDS*

- The data/operands are stored in memory.
- The individual instruction are brought from the memory to the processor.
- Then, the processor performs the specified operation.
- Let us see a typical instruction

    ADD LOCA, R0

- This instruction is an addition operation. The following are the steps to execute the instruction:

    Step 1: Fetch the instruction from main-memory into the processor.
    Step 2: Fetch the operand at location LOCA from main-memory into the processor.
    Step 3: Add the memory operand (i.e. fetched contents of LOCA) to the contents of register R0.
    Step 4: Store the result (sum) in R0.

- The same instruction can be realized using 2 instructions as:

    Load LOCA, R1
    Add R1, R0

- The following are the steps to execute the instruction:

    Step 1: Fetch the instruction from main-memory into the processor.
    Step 2: Fetch the operand at location LOCA from main-memory into the register R1.
    Step 3: Add the content of Register R1 and the contents of register R0.
    Step 4: Store the result (sum) in R0.

MAIN PARTS OF PROCESSOR

- The **processor** contains ALU, control-circuitry and many registers.
- The processor contains „n“ general-purpose registers **R_0** through **R_{n-1}**.
- The **IR** holds the instruction that is currently being executed.
- The **control-unit** generates the timing-signals that determine when a given action is to take place.
- The **PC** contains the memory-address of the next-instruction to be fetched & executed.
- During the execution of an instruction, the contents of PC are updated to point to next instruction.
- The **MAR** holds the address of the memory-location to be accessed.
- The **MDR** contains the data to be written into or read out of the addressed location.
- MAR and MDR facilitates the communication with memory.

(IR → Instruction-Register, PC → Program Counter)
(MAR → Memory Address Register, MDR→ Memory Data Register)

STEPS TO EXECUTE AN INSTRUCTION

1) The address of first instruction (to be executed) gets loaded into PC.
2) The contents of PC (i.e. address) are transferred to the MAR & control-unit issues Read signal to memory.
3) After certain amount of elapsed time, the first instruction is read out of memory and placed into MDR.
4) Next, the contents of MDR are transferred to IR. At this point, the instruction can be decoded & executed.
5) To fetch an operand, it's address is placed into MAR & control-unit issues Read signal. As a result, the operand is transferred from memory into MDR, and then it is transferred from MDR to ALU.
6) Likewise required number of operands is fetched into processor.
7) Finally, ALU performs the desired operation.
8) If the result of this operation is to be stored in the memory, then the result is sent to the MDR.
9) The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.
10) At some point during execution, contents of PC are incremented to point to next instruction in the program.

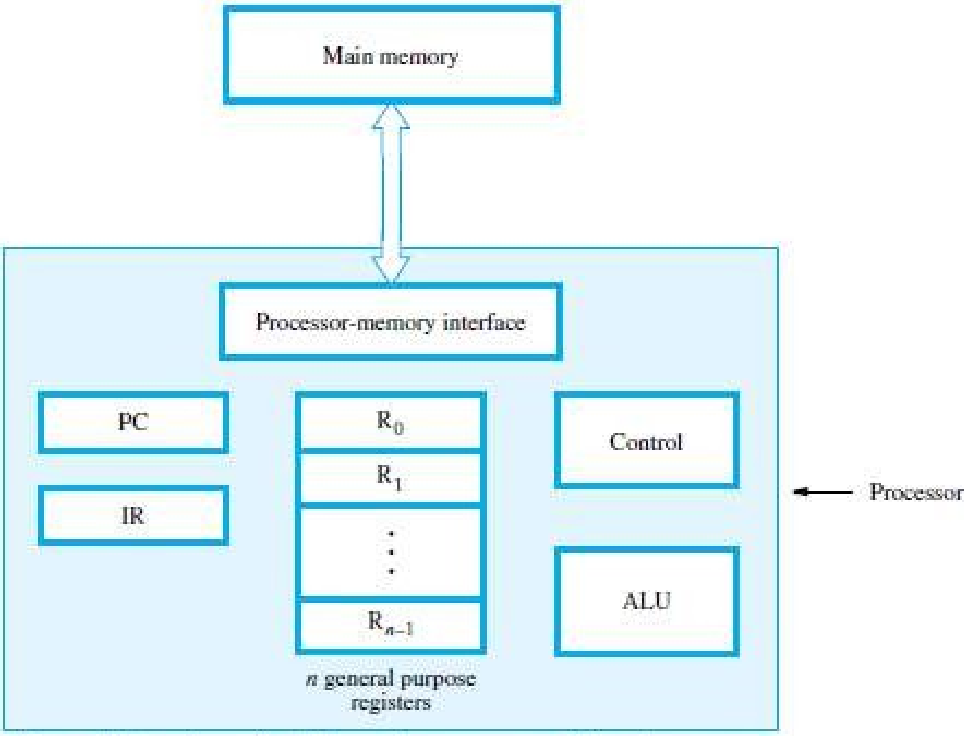


Figure 1.2 Connection between the processor and the main memory.

BUS STRUCTURE

- A bus is a group of lines that serves as a connecting path for several devices.
- A bus may be lines or wires.
- The lines carry data or address or control signal.
- There are 2 types of Bus structures: 1) Single Bus Structure and 2) Multiple Bus Structure.

1) Single Bus Structure

- ➢ Because the bus can be used for only one transfer at a time, only 2 units can actively use the bus at any given time.
- ➢ Bus control lines are used to arbitrate multiple requests for use of the bus.
- ➢ **Advantages:**
  1) Low cost &
  2) Flexibility for attaching peripheral devices.

2) Multiple Bus Structure

- ➢ Systems that contain multiple buses achieve more concurrency in operations.
- ➢ Two or more transfers can be carried out at the same time.
- ➢ **Advantage:** Better performance.
- ➢ **Disadvantage:** Increased cost.

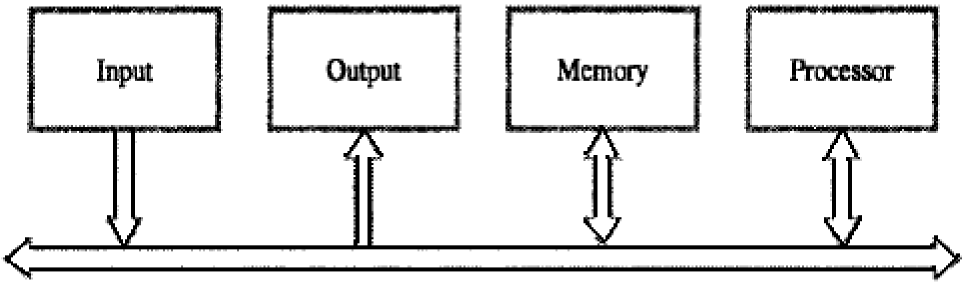


Figure 1.3 Single-bus structure.

- The devices connected to a bus vary widely in their speed of operation.
- To synchronize their operational-speed, buffer-registers can be used.
- **Buffer Registers**
  - → are included with the devices to hold the information during transfers.
  - → prevent a high-speed processor from being locked to a slow I/O device during data transfers.

PERFORMANCE

- The most important measure of performance of a computer is how quickly it can execute programs.
- The speed of a computer is affected by the design of
  1) Instruction-set.
  2) Hardware & the technology in which the hardware is implemented.
  3) Software including the operating system.
- Because programs are usually written in a HLL, performance is also affected by the compiler that translates programs into machine language. (HLL→ High Level Language).
- For best performance, it is necessary to design the compiler, machine instruction set and hardware in a co-ordinated way.

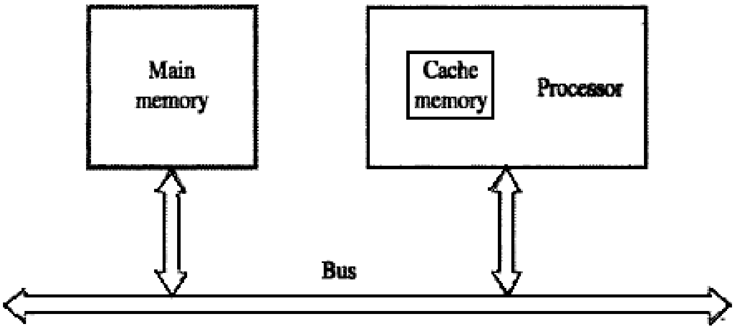


Figure 1.5 The processor cache.

- Let us examine the flow of program instructions and data between the memory & the processor.
- At the start of execution, all program instructions are stored in the main-memory.
- As execution proceeds, instructions are fetched into the processor, and a copy is placed in the cache.
- Later, if the same instruction is needed a second time, it is read directly from the cache.
- A program will be executed faster

  if movement of instruction/data between the main-memory and the processor is minimized

  which is achieved by using the cache.

PROCESSOR CLOCK

- Processor circuits are controlled by a timing signal called a **Clock**.
- The clock defines regular time intervals called **Clock Cycles**.
- To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps such that each step can be completed in one clock cycle.
- Let P = Length of one clock cycle

  R = Clock rate.
- Relation between P and R is given by

$$R = \frac{1}{P}$$

- R is measured in cycles per second.
- Cycles per second is also called Hertz (Hz)

BASIC PERFORMANCE EQUATION

- Let T = Processor time required to executed a program.

  N = Actual number of instruction executions.

  S = Average number of basic steps needed to execute one machine instruction.

  R = Clock rate in cycles per second.
- The program execution time is given by

$$T = \frac{N \times S}{R} \qquad \text{------(1)}$$

- Equ1 is referred to as the basic performance equation.
- To achieve high performance, the computer designer must reduce the value of T, which means reducing N and S, and increasing R.
  - ➢ The value of N is reduced if source program is compiled into fewer machine instructions.
  - ➢ The value of S is reduced if instructions have a smaller number of basic steps to perform.
  - ➢ The value of R can be increased by using a higher frequency clock.
- Care has to be taken while modifying values since changes in one parameter may affect the other.

CLOCK RATE

• There are 2 possibilities for increasing the clock rate R:

1) Improving the IC technology makes logic-circuits faster.

This reduces the time needed to compute a basic step. (IC → integrated circuits).

This allows the clock period P to be reduced and the clock rate R to be increased.

2) Reducing the amount of processing done in one basic step also reduces the clock period P.

• In presence of a cache, the percentage of accesses to the main-memory is small.

Hence, much of performance-gain expected from the use of faster technology can be realized.

The value of T will be reduced by same factor as R is increased „.“ S & N are not affected.

PERFORMANCE MEASUREMENT

• Benchmark refers to standard task used to measure how well a processor operates.

• The Performance Measure is the time taken by a computer to execute a given benchmark.

• SPEC selects & publishes the standard programs along with their test results for different application domains. (SPEC → System Performance Evaluation Corporation).

• SPEC Rating is given by

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

• SPEC rating = 50 → The computer under test is 50 times as fast as reference-computer.

• The test is repeated for all the programs in the SPEC suite.

Then, the geometric mean of the results is computed.

• Let SPEC_i = Rating for program „i' in the suite.

Overall SPEC rating for the computer is given by

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} \text{SPEC}_i \right)^{\frac{1}{n}}$$

where n = no. of programs in the suite.

INSTRUCTION SET: CISC AND RISC

RISC	CISC
Simple instructions taking one cycle.	Complex instructions taking multiple cycle.
Instructions are executed by hardwired control unit.	Instructions are executed by microprogrammed control unit.
Few instructions.	Many instructions.
Fixed format instructions.	Variable format instructions.
Few addressing modes, and most instructions have register to register addressing mode.	Many addressing modes.
Multiple register set.	Single register set.
Highly pipelined.	No pipelined or less pipelined.

Problem 1:

List the steps needed to execute the machine instruction:

Load R2, LOC

in terms of transfers between the components of processor and some simple control commands. Assume that the address of the memory-location containing this instruction is initially in register PC.

Solution:

1. Transfer the contents of register PC to register MAR.
2. Issue a Read command to memory.
   And, then wait until it has transferred the requested word into register MDR.
3. Transfer the instruction from MDR into IR and decode it.
4. Transfer the address LOCA from IR to MAR.
5. Issue a Read command and wait until MDR is loaded.
6. Transfer contents of MDR to the ALU.
7. Transfer contents of R0 to the ALU.
8. Perform addition of the two operands in the ALU and transfer result into R0.
9. Transfer contents of PC to ALU.
10. Add 1 to operand in ALU and transfer incremented address to PC.

Problem 2:

List the steps needed to execute the machine instruction:

Add R4, R2, R3

in terms of transfers between the components of processor and some simple control commands. Assume that the address of the memory-location containing this instruction is initially in register PC.

Solution:

1. Transfer the contents of register PC to register MAR.
2. Issue a Read command to memory.
   And, then wait until it has transferred the requested word into register MDR.
3. Transfer the instruction from MDR into IR and decode it.
4. Transfer contents of R1 and R2 to the ALU.
5. Perform addition of two operands in the ALU and transfer answer into R3.
6. Transfer contents of PC to ALU.
7. Add 1 to operand in ALU and transfer incremented address to PC.

Problem 3:

(a) Give a short sequence of machine instructions for the task "Add the contents of memory-location A to those of location B, and place the answer in location C". Instructions:

Load R*i*, LOC
and
Store R*i*, LOC

are the only instructions available to transfer data between memory and the general purpose registers. Add instructions are described in Section 1.3. Do not change contents of either location A or B.

(b) Suppose that Move and Add instructions are available with the formats:

Move Location1, Location2
and
Add Location1, Location2

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers. Is it possible to use fewer instructions of these types to accomplish the task in part (a)? If yes, give the sequence.

Solution:

(a)

Load A, R0
Load B, R1
Add R0, R1
Store R1, C

(b) Yes;

Move B, C
Add A, C

Problem 4:

A program contains 1000 instructions. Out of that 25% instructions requires 4 clock cycles,40% instructions requires 5 clock cycles and remaining require 3 clock cycles for execution. Find the total time required to execute the program running in a 1 GHz machine.

Solution:

N = 1000
25% of N= 250 instructions require 4 clock cycles.
40% of N =400 instructions require 5 clock cycles.
35% of N=350 instructions require 3 clock cycles.
T = (N*S)/R= (250*4+400*5+350*3)/1X10^9 =(1000+2000+1050)/1*10^9= 4.05 µs.

Problem 5:

For the following processor, obtain the performance.

Clock rate = 800 MHz
No. of instructions executed = 1000
Average no of steps needed / machine instruction = 20

Solution:

$$T = \frac{N \times S}{R} = (1000*20)/800 * 10^{6} = 25 \text{ micro sec or } 25*10^{-6} \text{ sec}$$

Problem 6:

(a) Program execution time T is to be examined for a certain high-level language program. The program can be run on a RISC or a CISC computer. Both computers use pipelined instruction execution, but pipelining in the RISC machine is more effective than in the CISC machine. Specifically, the effective value of S in the T expression for the RISC machine is 1.2, bit it is only 1.5 for the CISC machine. Both machines have the same clock rate R. What is the largest allowable value for N, the number of instructions executed on the CISC machine, expressed as a percentage of the N value for the RISC machine, if time for execution on the CISC machine is to be longer than on the RISC machine?

(b) Repeat Part (a) if the clock rate R for the RISC machine is 15 percent higher than that for the CISC machine.

Solution:

(a) Let $T_R = (N_R \text{ X } S_R)/R_R$ & $T_C = (N_C \text{ X } S_C)/R_C$ be execution times on RISC and CISC processors. Equating execution times and clock rates, we have

$$1.2N_R = 1.5N_C$$

Then

$$N_C/N_R = 1.2/1.5 = 0.8$$

Therefore, the largest allowable value for N_C is 80% of N_R.

(b) In this case,

$$1.2N_R/1.15 = 1.5N_C/1.00$$

Then

$$N_C/N_R = 1.2/(1.15 \text{ X } 1.5) = 0.696$$

Therefore, the largest allowable value for N_C is 69.6% of N_R.

Problem 7:

(a) Suppose that execution time for a program is proportional to instruction fetch time. Assume that fetching an instruction from the cache takes 1 time unit, but fetching it from the main-memory takes 10 time units. Also, assume that a requested instruction is found in the cache with probability 0.96. Finally, assume that if an instruction is not found in the cache it must first be fetched from the main-memory into the cache and then fetched from the cache to be executed. Compute the ratio of program execution time without the cache to program execution time with the cache. This ratio is called the speedup resulting from the presence of the cache.

(b) If the size of the cache is doubled, assume that the probability of not finding a requested instruction there is cut in half. Repeat part (a) for a doubled cache size.

Solution:

(a) Let cache access time be 1 and main-memory access time be 20. Every instruction that is executed must be fetched from the cache, and an additional fetch from the main-memory must be performed for 4% of these cache accesses.

Therefore,

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.04 \times 20)} = 11.1$$

(b)

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.02 \times 20)} = 16.7$$

MODULE 1 (CONT.): MACHINE INSTRUCTIONS & PROGRAMS

MEMORY-LOCATIONS & ADDRESSES

- **Memory** consists of many millions of storage cells (flip-flops).
- Each cell can store a bit of information i.e. 0 or 1 (Figure 2.1).
- Each group of n bits is referred to as a **word** of information, and n is called the **word length**.
- The word length can vary from 8 to 64 bits.
- A unit of 8 bits is called a **byte**.
- Accessing the memory to store or retrieve a single item of information (word/byte) requires distinct addresses for each item location. (It is customary to use numbers from 0 through 2^k-1 as the addresses of successive-locations in the memory).
- If 2^k = no. of addressable locations;

  then 2^k addresses constitute the address-space of the computer.

  For example, a 24-bit address generates an address-space of 2^{24} locations (16 MB).

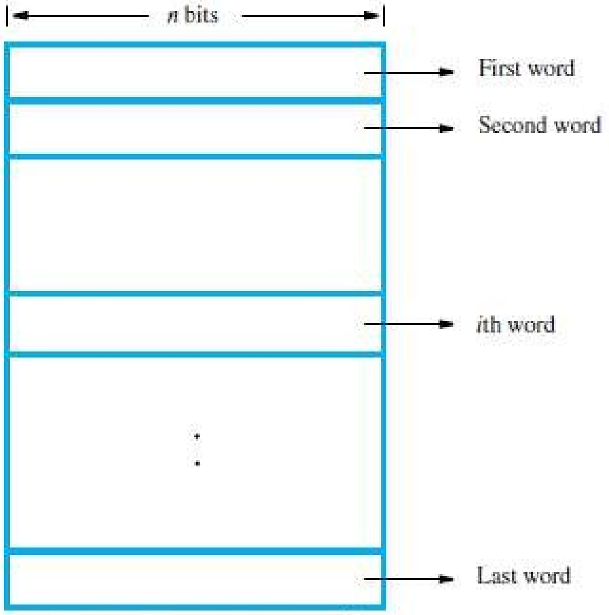
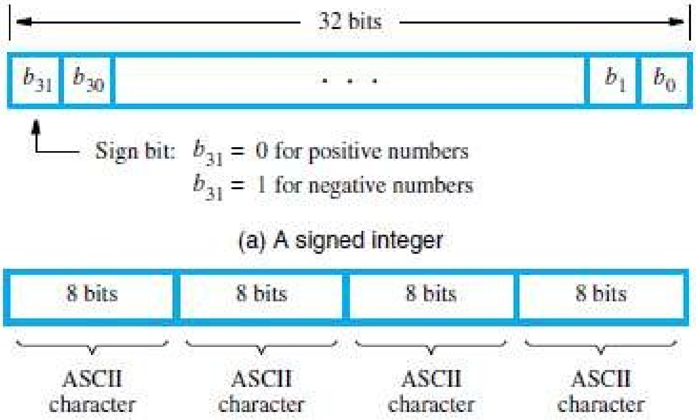


Figure 2.1 Memory words.


32 bits
b_{31} b_{30} . . . b_1 b_0
Sign bit: b_{31} = 0 for positive numbers
b_{31} = 1 for negative numbers
8 bits
8 bits
8 bits
8 bits
ASCII character
ASCII character
ASCII character
ASCII character

(a) A signed integer

(b) Four characters

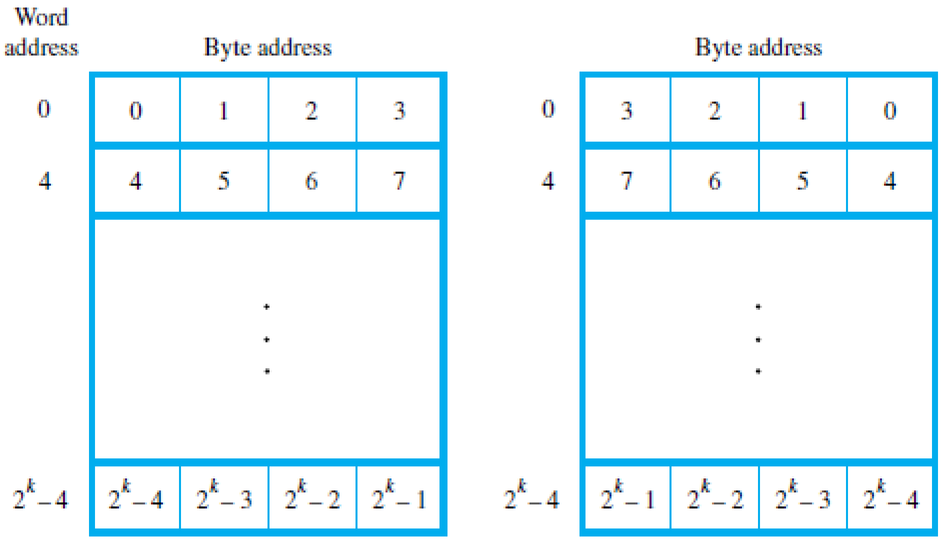
Figure 2.2 Examples of encoded information in a 32-bit word.

BYTE-ADDRESSABILITY

- In byte-addressable memory, successive addresses refer to successive byte locations in the memory.
- Byte locations have addresses 0, 1, 2. . . . .
- If the word-length is 32 bits, successive words are located at addresses 0, 4, 8. . with each word having 4 bytes.

BIG-ENDIAN & LITTLE-ENDIAN ASSIGNMENTS

- There are two ways in which byte-addresses are arranged (Figure 2.3).
  1) **Big-Endian:** Lower byte-addresses are used for the more significant bytes of the word.
  2) **Little-Endian:** Lower byte-addresses are used for the less significant bytes of the word
- In both cases, byte-addresses 0, 4, 8. . . . . are taken as the addresses of successive words in the memory.



(a) Big-endian assignment (b) Little-endian assignment

Figure 2.3 Byte and word addressing.

Consider a 32-bit integer (in hex): 0**x**12345678 which consists of 4 bytes: 12, 34, 56, and 78.

- ➢ Hence this integer will occupy 4 bytes in memory.
- ➢ Assume, we store it at memory address starting 1000.
- ➢ On little-endian, memory will look like

Address	Value
1000	78
1001	56
1002	34
1003	12

- ➢ On big-endian, memory will look like

Address	Value
1000	12
1001	34
1002	56
1003	78

WORD ALIGNMENT

- Words are said to be **Aligned** in memory if they begin at a byte-address that is a multiple of the number of bytes in a word.
- For example,
  - ➢ If the word length is 16(2 bytes), aligned words begin at byte-addresses 0, 2, 4 . . . . .
  - ➢ If the word length is 64(2 bytes), aligned words begin at byte-addresses 0, 8, 16 . . . . .
- Words are said to have **Unaligned Addresses**, if they begin at an arbitrary byte-address.

ACCESSING NUMBERS, CHARACTERS & CHARACTERS STRINGS

- A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte-address.
- There are two ways to indicate the length of the string:
  1) A special control character with the meaning "end of string" can be used as the last character in the string.
  2) A separate memory word location or register can contain a number indicating the length of the string in bytes.

MEMORY OPERATIONS

- Two memory operations are:
  1) Load (Read/Fetch) &
  2) Store (Write).
- The **Load** operation transfers a copy of the contents of a specific memory-location to the processor. The memory contents remain unchanged.
- Steps for Load operation:
  1) Processor sends the address of the desired location to the memory.
  2) Processor issues „read“ signal to memory to fetch the data.
  3) Memory reads the data stored at that address.
  4) Memory sends the read data to the processor.
- The **Store** operation transfers the information from the register to the specified memory-location. This will destroy the original contents of that memory-location.
- Steps for Store operation are:
  1) Processor sends the address of the memory-location where it wants to store data.
  2) Processor issues „write“ signal to memory to store the data.
  3) Content of register(MDR) is written into the specified memory-location.

INSTRUCTIONS & INSTRUCTION SEQUENCING

- A computer must have instructions capable of performing 4 types of operations:
  1) Data transfers between the memory and the registers (MOV, PUSH, POP, XCHG).
  2) Arithmetic and logic operations on data (ADD, SUB, MUL, DIV, AND, OR, NOT).
  3) Program sequencing and control (CALL.RET, LOOP, INT).
  4) I/0 transfers (IN, OUT).

REGISTER TRANSFER NOTATION (RTN)

- The possible locations in which transfer of information occurs are: 1) Memory-location 2) Processor register & 3) Registers in I/O device.

Location	Hardware Binary Address	Example	Description
Memory	LOC, PLACE, NUM	R1 ← [LOC]	Contents of memory-location LOC are transferred into register R1.
Processor	R0, R1 ,R2	[R3] ← [R1]+[R2]	Add the contents of register R1 &R2 and places their sum into R3.
I/O Registers	DATAIN, DATAOUT	R1 ← DATAIN	Contents of I/O register DATAIN are transferred into register R1.

ASSEMBLY LANGUAGE NOTATION

- To represent machine instructions and programs, assembly language format is used.

Assembly Language Format	Description
Move LOC, R1	Transfer data from memory-location LOC to register R1. The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.
Add R1, R2, R3	Add the contents of registers R1 and R2, and places their sum into register R3.

BASIC INSTRUCTION TYPES

Instruction Type	Syntax	Example	Description	Instructions for Operation C<-[A]+[B]
Three Address	Opcode Source1,Source2,Destination	Add A,B,C	Add the contents of memory-locations A & B. Then, place the result into location C.	
Two Address	Opcode Source, Destination	Add A,B	Add the contents of memory-locations A & B. Then, place the result into location B, replacing the original contents of this location. Operand B is both a source and a destination.	Move B, C Add A, C
One Address	Opcode Source/Destination	Load A	Copy contents of memory-location A into accumulator.	Load A Add B Store C
		Add B	Add contents of memory-location B to contents of accumulator register & place sum back into accumulator.	
		Store C	Copy the contents of the accumulator into location C.	
Zero Address	Opcode [no Source/Destination]	Push	Locations of all operands are defined implicitly. The operands are stored in a pushdown stack.	Not possible

- Access to data in the registers is much faster than to data stored in memory-locations.
- Let Ri represent a general-purpose register. The instructions:

Load A,Ri
Store Ri,A
Add A,Ri

are generalizations of the Load, Store and Add Instructions for the single-accumulator case, in which register Ri performs the function of the accumulator.

- In processors, where arithmetic operations as allowed only on operands that are in registers, the task C<-[A]+[B] can be performed by the instruction sequence:

Move A,Ri
Move B,Rj
Add Ri,Rj
Move Rj,C

INSTRUCTION EXECUTION & STRAIGHT LINE SEQUENCING

• The program is executed as follows:

1) Initially, the address of the first instruction is loaded into PC (Figure 2.8).

2) Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *Straight-Line sequencing*.

3) During the execution of each instruction, PC is incremented by 4 to point to next instruction.

• There are 2 phases for Instruction Execution:

1) Fetch Phase: The instruction is fetched from the memory-location and placed in the IR.

2) Execute Phase: The contents of IR is examined to determine which operation is to be performed. The specified-operation is then performed by the processor.

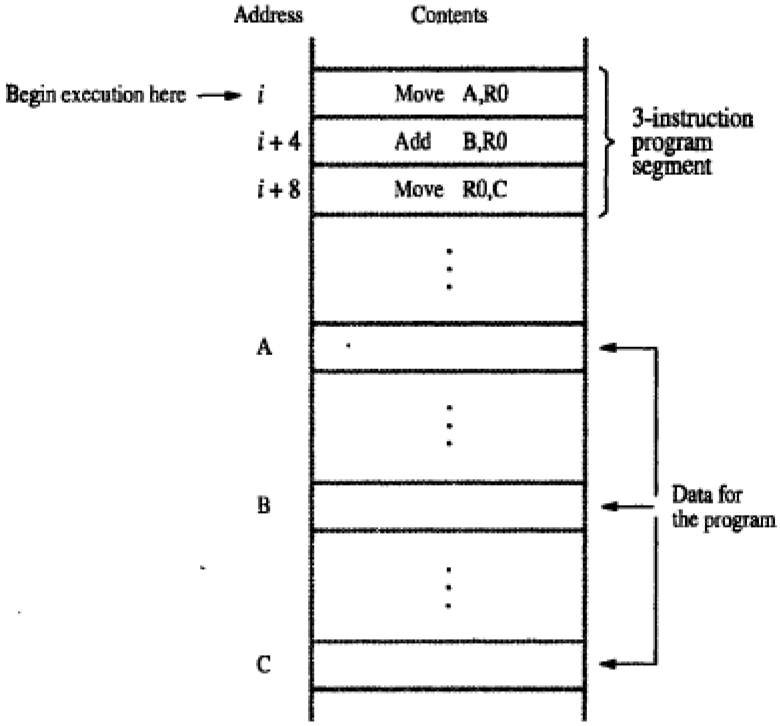


Figure 2.8 A program for C ← [A] + [B].

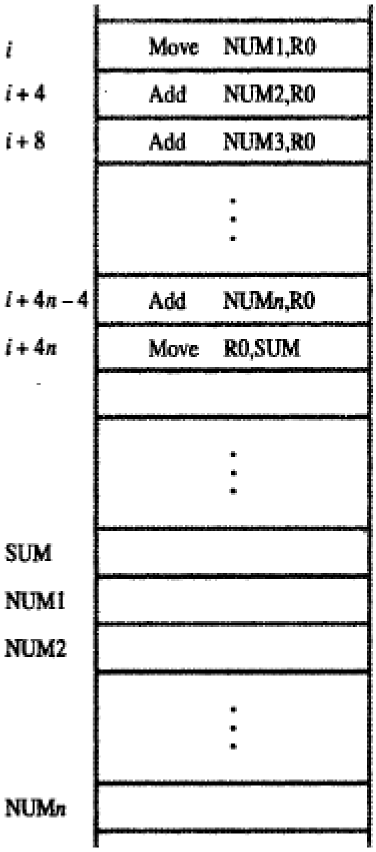


Figure 2.9 A straight-line program for adding n numbers.

Program Explanation

• Consider the program for adding a list of n numbers (Figure 2.9).

• The Address of the memory-locations containing the n numbers are symbolically given as NUM1, NUM2…..NUMn.

• Separate Add instruction is used to add each number to the contents of register R0.

• After all the numbers have been added, the result is placed in memory-location SUM.

BRANCHING

- Consider the task of adding a list of „n“ numbers (Figure 2.10).
- Number of entries in the list „n“ is stored in memory-location **N**.
- Register **R1** is used as a counter to determine the number of times the loop is executed.
- Content-location N is loaded into register R1 at the beginning of the program.
- The **Loop** is a straight line sequence of instructions executed as many times as needed.
  The loop starts at location LOOP and ends at the instruction Branch>0.
- During each pass,
  → address of the next list entry is determined and
  → that entry is fetched and added to R0.
- The instruction *Decrement R1* reduces the contents of R1 by 1 each time through the loop.
- Then **Branch Instruction** loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address called the **Branch Target**.
- A **Conditional Branch Instruction** causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

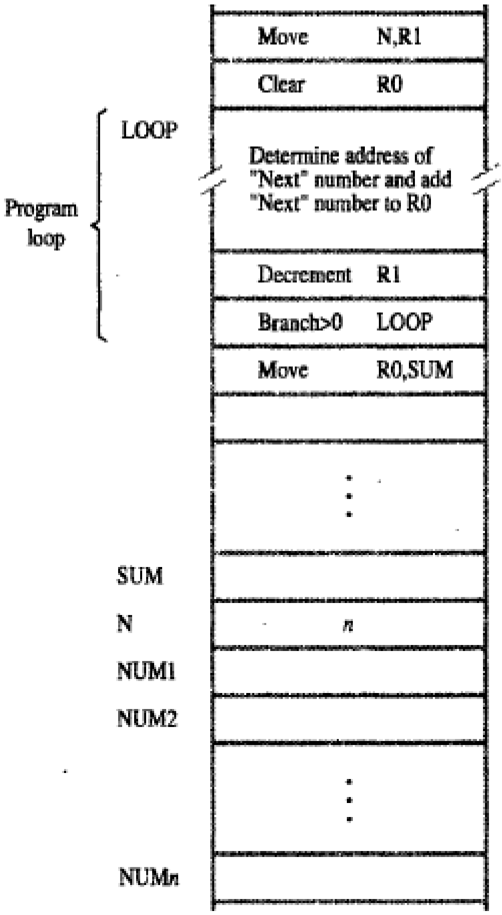


Figure 2.10 Using a loop to add n numbers.

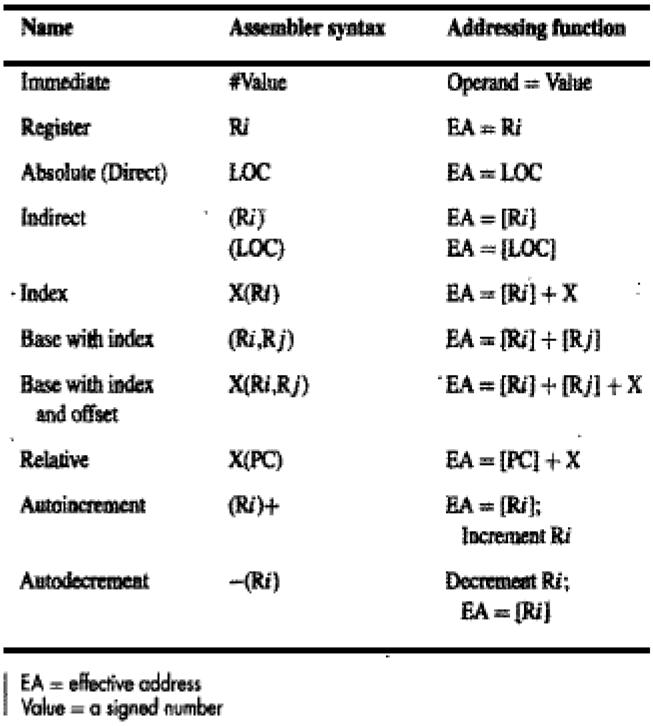
CONDITION CODES

- The processor keeps track of information about the results of various operations. This is accomplished by recording the required information in individual bits, called **Condition Code Flags**.
- These flags are grouped together in a special processor-register called the condition code register (or statue register).
- Four commonly used flags are:
  1) N (negative) set to 1 if the result is negative, otherwise cleared to 0.
  2) Z (zero) set to 1 if the result is 0; otherwise, cleared to 0.
  3) V (overflow) set to 1 if arithmetic overflow occurs; otherwise, cleared to 0.
  4) C (carry) set to 1 if a carry-out results from the operation; otherwise cleared to 0.

ADDRESSING MODES

- The different ways in which the location of an operand is specified in an instruction are referred to as **Addressing Modes** (Table 2.1).

Table 2.1 Generic addressing modes



Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

IMPLEMENTATION OF VARIABLE AND CONSTANTS

- **Variable** is represented by allocating a memory-location to hold its value.
- Thus, the value can be changed as needed using appropriate instructions.
- There are 2 accessing modes to access the variables:
  1) Register Mode
  2) Absolute Mode

Register Mode

- The operand is the contents of a register.
- The name (or address) of the register is given in the instruction.
- Registers are used as temporary storage locations where the data in a register are accessed.
- For example, the instruction

Move R1, R2 ;Copy content of register R1 into register R2.

Absolute (Direct) Mode

- The operand is in a memory-location.
- The address of memory-location is given explicitly in the instruction.
- The absolute mode can represent global variables in the program.
- For example, the instruction

Move LOC, R2 ;Copy content of memory-location LOC into register R2.

Immediate Mode

- The operand is given explicitly in the instruction.
- For example, the instruction

Move #200, R0 ;Place the value 200 in register R0.

- Clearly, the immediate mode is only used to specify the value of a source-operand.

INDIRECTION AND POINTERS

- Instruction does not give the operand or its address explicitly.
- Instead, the instruction provides information from which the new address of the operand can be determined.
- This address is called **Effective Address (EA)** of the operand.

Indirect Mode

- The EA of the operand is the contents of a register(or memory-location).
- The register (or memory-location) that contains the address of an operand is called a **Pointer**.
- We denote the indirection by

  → name of the register or

  → new address given in the instruction.

  E.g: *Add (R1),R0* ;The operand is in memory. Register R1 gives the effective-address (B) of the operand. The data is read from location B and added to contents of register R0.

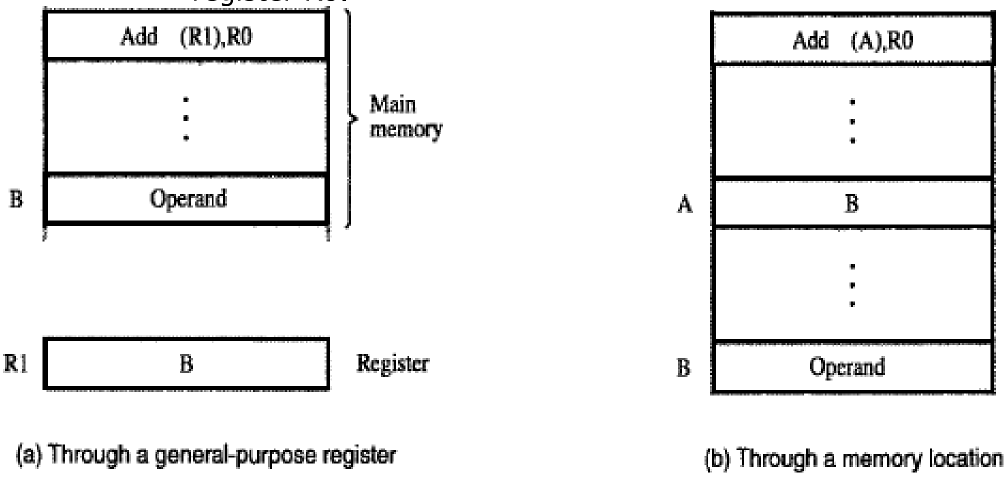
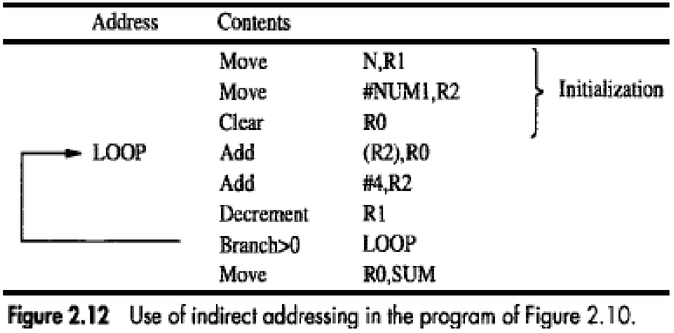

Add (R1),R0
Main memory
B
Operand
R1
B
Register
Add (A),R0
A
B
B
Operand
(a) Through a general-purpose register
(b) Through a memory location

Figure 2.11 Indirect addressing.

- To execute the Add instruction in fig 2.11 (a), the processor uses the value which is in register R1, as the EA of the operand.
- It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0.
- Indirect addressing through a memory-location is also possible as shown in fig 2.11(b). In this case, the processor first reads the contents of memory-location A, then requests a second read operation using the value B as an address to obtain the operand.



```
Address     Contents

            Move        N,R1       }
            Move        #NUM1,R2   } Initialization
            Clear       R0         }
LOOP        Add         (R2),R0
            Add         #4,R2
            Decrement   R1
            Branch>0    LOOP
            Move        R0,SUM
```

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

Program Explanation

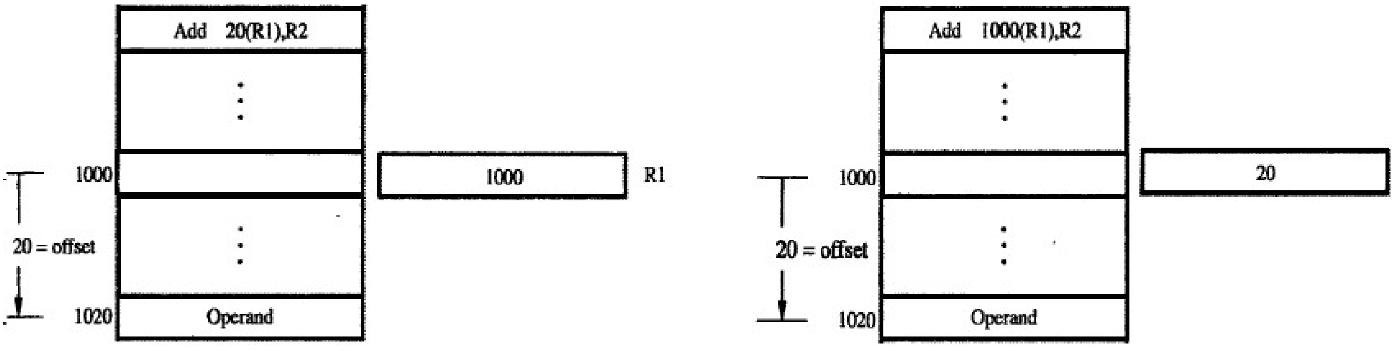
- In above program, Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2.
- The initialization-section of the program loads the counter-value n from memory-location N into R1 and uses the immediate addressing-mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0.
- The first two instructions in the loop implement the unspecified instruction block starting at LOOP.
- The first time through the loop, the instruction Add (R2), R0 fetches the operand at location NUM1 and adds it to R0.
- The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

INDEXING AND ARRAYS

- A different kind of flexibility for accessing operands is useful in dealing with lists and arrays.

Index mode

- The operation is indicated as X(Ri)
  where X=the constant value which defines an offset(also called a displacement).
  Ri=the name of the index register which contains address of a new location.
- The effective-address of the operand is given by EA=X+[Ri]
- The contents of the index-register are not changed in the process of generating the effective-address.
- The constant X may be given either
  → as an explicit number or
  → as a symbolic-name representing a numerical value.



(a) Offset is given as a constant

(b) Offset is in the index register

Figure 2.13 Indexed addressing.

- Fig(a) illustrates two ways of using the Index mode. In fig(a), the index register, R1, contains the address of a memory-location, and the value X defines an offset(also called a displacement) from this address to the location where the operand is found.
- To find EA of operand:
  Eg: Add 20(R1), R2
  EA=>1000+20=1020
- An alternative use is illustrated in fig(b). Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective-address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.

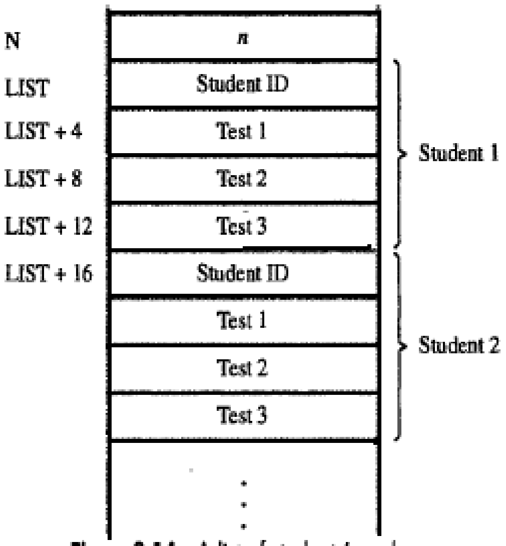
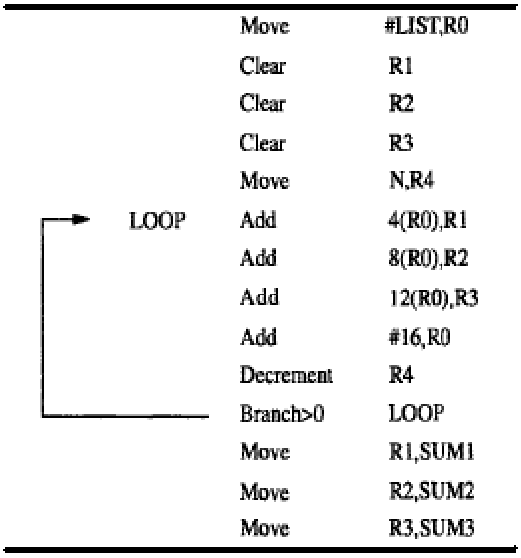


Figure 2.14 A list of students' marks.



```
        Move        #LIST,R0
        Clear       R1
        Clear       R2
        Clear       R3
        Move        N,R4
LOOP    Add         4(R0),R1
        Add         8(R0),R2
        Add         12(R0),R3
        Add         #16,R0
        Decrement   R4
        Branch>0    LOOP
        Move        R1,SUM1
        Move        R2,SUM2
        Move        R3,SUM3
```

Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

Base with Index Mode

- Another version of the Index mode uses 2 registers which can be denoted as
  (Ri, Rj)
- Here, a second register may be used to contain the offset X.
- The second register is usually called the *base register*.
- The effective-address of the operand is given by EA=[Ri]+[Rj]
- This form of indexed addressing provides more flexibility in accessing operands because both components of the effective-address can be changed.

Base with Index & Offset Mode

- Another version of the Index mode uses 2 registers plus a constant, which can be denoted as
  X(Ri, Rj)
- The effective-address of the operand is given by EA=X+[Ri]+[Rj]
- This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing-mode. In other words, this mode implements a 3-dimensional array.

RELATIVE MODE

- This is similar to index-mode with one difference:
  The effective-address is determined using the PC in place of the general purpose register Ri.
- The operation is indicated as X(PC).
- X(PC) denotes an effective-address of the operand which is X locations above or below the current contents of PC.
- Since the addressed-location is identified "relative" to the PC, the name Relative mode is associated with this type of addressing.
- This mode is used commonly in conditional branch instructions.
- An instruction such as

  Branch > 0 LOOP ;Causes program execution to go to the branch target location identified by name LOOP if branch condition is satisfied.

ADDITIONAL ADDRESSING MODES

1) Auto Increment Mode

➢ Effective-address of operand is contents of a register specified in the instruction (Fig: 2.16).

➢ After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

➢ Implicitly, the increment amount is 1.

➢ This mode is denoted as

(Ri)+ ;where Ri=pointer-register.

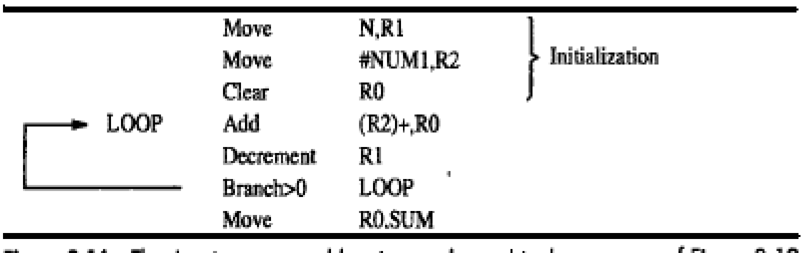
2) Auto Decrement Mode

➢ The contents of a register specified in the instruction are first automatically decremented and are then used as the effective-address of the operand.

➢ This mode is denoted as

-(Ri) ;where Ri=pointer-register.

➢ These 2 modes can be used together to implement an important data structure called a stack.



```
             Move        N,R1      }
             Move        #NUM1,R2  } Initialization
             Clear       R0        }
LOOP         Add         (R2)+,R0
             Decrement   R1
             Branch>0    LOOP
             Move        R0,SUM
```

Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

ASSEMBLY LANGUAGE

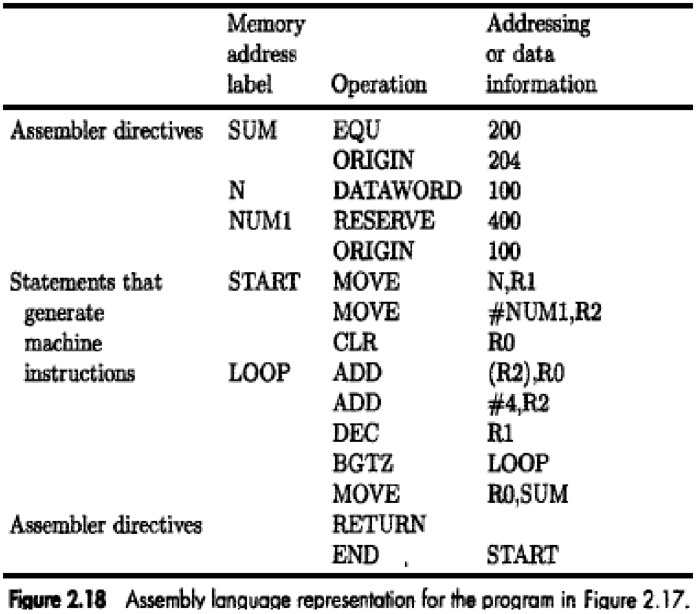
- We generally use symbolic-names to write a program.
- A complete set of symbolic-names and rules for their use constitute an **Assembly Language**.
- The set of rules for using the mnemonics in the specification of complete instructions and programs is called the **Syntax** of the language.
- Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an **Assembler.**
- The user program in its original alphanumeric text formal is called a **Source Program**, and the assembled machine language program is called an **Object Program**.

For example:

MOVE R0,SUM ;The term MOVE represents OP code for operation performed by instruction.
ADD #5,R3 ;Adds number 5 to contents of register R3 & puts the result back into registerR3.

ASSEMBLER DIRECTIVES

- **Directives** are the assembler commands to the assembler concerning the program being assembled.
- These commands are not translated into machine opcode in the object-program.



	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

Figure 2.18 Assembly language representation for the program in Figure 2.17.

- **EQU** informs the assembler about the value of an identifier (Figure: 2.18).
  Ex*: SUM EQU 200* ;Informs assembler that the name SUM should be replaced by the value 200.
- **ORIGIN** tells the assembler about the starting-address of memory-area to place the data block.
  Ex*:* ORIGIN 204 ;Instructs assembler to initiate data-block at memory-locations starting from 204.
- **DATAWORD** directive tells the assembler to load a value into the location.
  Ex: *N DATAWORD 100 ;*Informs the assembler to load data 100 into the memory-location N(204).
- **RESERVE** directive is used to reserve a block of memory.
  Ex: *NUM1 RESERVE 400 ;*declares a memory-block of 400 bytes is to be reserved for data.
- **END** directive tells the assembler that this is the end of the source-program text.
- **RETURN** directive identifies the point at which execution of the program should be terminated.
- Any statement that makes instructions or data being placed in a memory-location may be given a **label**. The label(say N or NUM1) is assigned a value equal to the address of that location.

GENERAL FORMAT OF A STATEMENT

- Most assembly languages require statements in a source program to be written in the form:

Label	*Operation*	*Operands*	*Comment*

1) **Label** is an optional name associated with the memory-address where the machine language instruction produced from the statement will be loaded.
2) **Operation Field** contains the OP-code mnemonic of the desired instruction or assembler.
3) **Operand Field** contains addressing information for accessing one or more operands, depending on the type of instruction.
4) **Comment Field** is used for documentation purposes to make program easier to understand.

ASSEMBLY AND EXECUTION OF PRGRAMS

• Programs written in an assembly language are automatically translated into a sequence of machine instructions by the **Assembler**.

• **Assembler Program**

→ replaces all symbols denoting operations & addressing-modes with binary-codes used in machine instructions.

→ replaces all names and labels with their actual values.

→ assigns addresses to instructions & data blocks, starting at address given in ORIGIN directive

→ inserts constants that may be given in DATAWORD directives.

→ reserves memory-space as requested by RESERVE directives.

• **Two Pass Assembler** has 2 passes:

1) First Pass: Work out all the addresses of labels.

➢ As the assembler scans through a source-program, it keeps track of all names of numerical-values that correspond to them in a symbol-table.

2) Second Pass: Generate machine code, substituting values for the labels.

➢ When a name appears a second time in the source-program, it is replaced with its value from the table.

• The assembler stores the object-program on a magnetic-disk. The object-program must be loaded into the memory of the computer before it is executed. For this, a **Loader Program** is used.

• **Debugger Program** is used to help the user find the programming errors.

• Debugger program enables the user

→ to stop execution of the object-program at some points of interest &

→ to examine the contents of various processor-registers and memory-location.

BASIC INPUT/OUTPUT OPERATIONS

- Consider the problem of moving a character-code from the keyboard to the processor (Figure: 2.19). For this transfer, buffer-register DATAIN & a status control flags(SIN) are used.
- When a key is pressed, the corresponding ASCII code is stored in a **DATAIN** register associated with the keyboard.
  - **SIN=1** → When a character is typed in the keyboard. This informs the processor that a valid character is in DATAIN.
  - **SIN=0** → When the character is transferred to the processor.
- An analogous process takes place when characters are transferred from the processor to the display. For this transfer, buffer-register DATAOUT & a status control flag SOUT are used.
  - **SOUT=1** → When the display is ready to receive a character.
  - **SOUT=0** → When the character is being transferred to DATAOUT.
- The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a **device interface**.

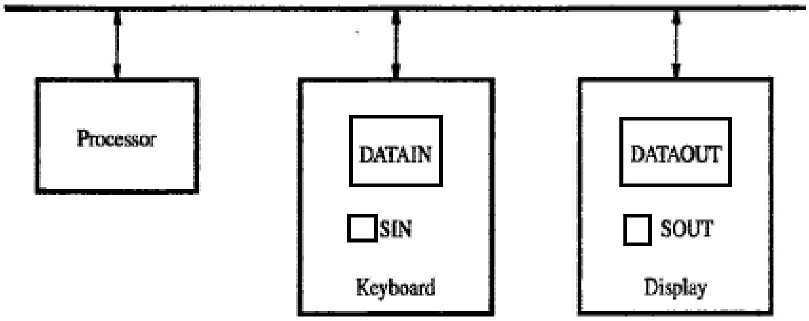
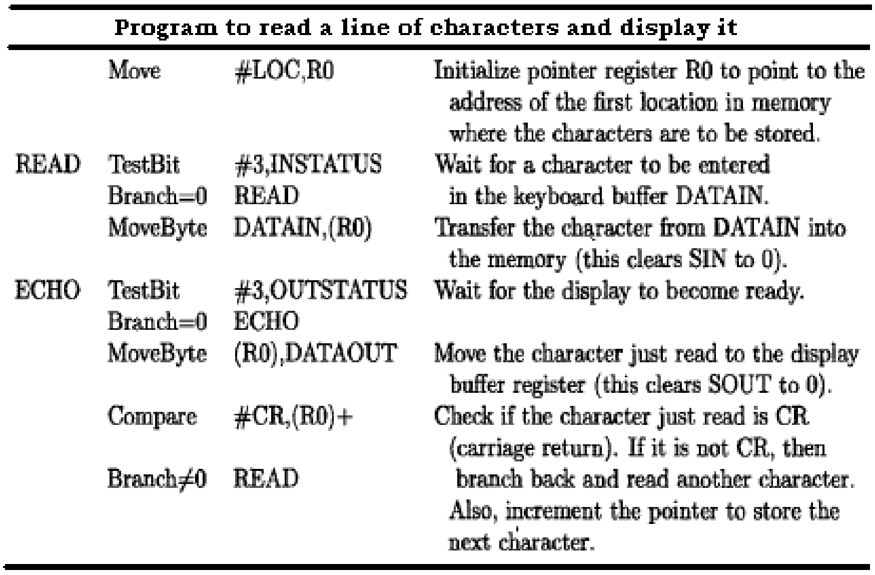


Figure 2.19 Bus connection for processor, keyboard, and display.

MEMORY-MAPPED I/O



Program to read a line of characters and display it

```
       Move       #LOC,R0          Initialize pointer register R0 to point to the
                                     address of the first location in memory
                                     where the characters are to be stored.
READ   TestBit    #3,INSTATUS      Wait for a character to be entered
       Branch=0   READ               in the keyboard buffer DATAIN.
       MoveByte   DATAIN,(R0)      Transfer the character from DATAIN into
                                     the memory (this clears SIN to 0).
ECHO   TestBit    #3,OUTSTATUS     Wait for the display to become ready.
       Branch=0   ECHO
       MoveByte   (R0),DATAOUT     Move the character just read to the display
                                     buffer register (this clears SOUT to 0).
       Compare    #CR,(R0)+        Check if the character just read is CR
                                     (carriage return). If it is not CR, then
       Branch≠0   READ               branch back and read another character.
                                   Also, increment the pointer to store the
                                     next character.
```

Figure 2.20 A program that reads a line of characters and displays it.

- Some address values are used to refer to peripheral device buffer-registers such as DATAIN & DATAOUT.
- No special instructions are needed to access the contents of the registers; data can be transferred between these registers and the processor using instructions such as Move, Load or Store.
- For example, contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

  MoveByte DATAIN,R1
- The MoveByte operation code signifies that the operand size is a byte.
- The **Testbit** instruction tests the state of one bit in the destination, where the bit position to be tested is indicated by the first operand.

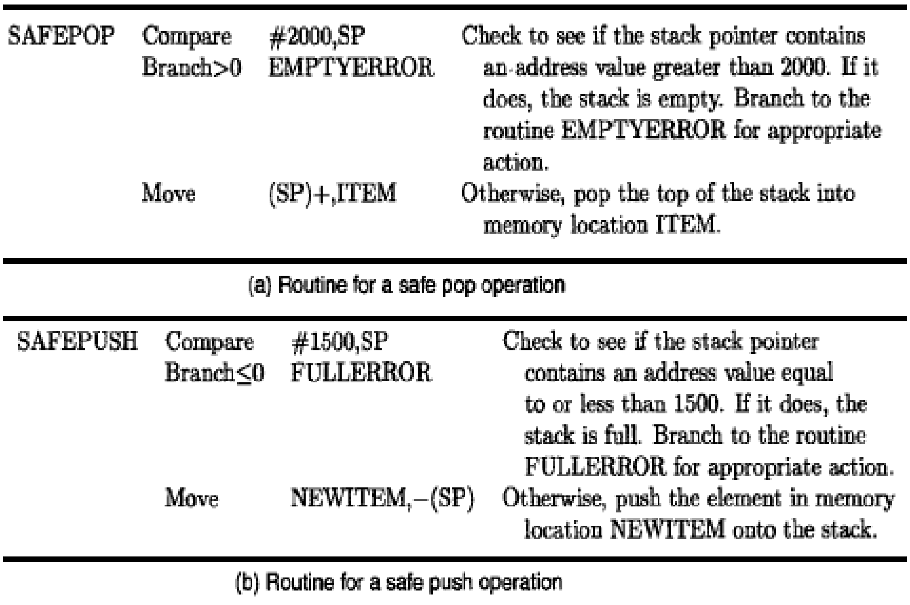
STACKS

- A **stack** is a special type of data structure where elements are inserted from one end and elements are deleted from the same end. This end is called the **top** of the stack (Figure: 2.14).
- The various operations performed on stack:
  1) Insert: An element is inserted from top end. Insertion operation is called **push** operation.
  2) Delete: An element is deleted from top end. Deletion operation is called **pop** operation.
- A processor-register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the **Stack Pointer (SP)**.
- If we assume a byte-addressable memory with a 32-bit word length,
  1) The push operation can be implemented as

     Subtract #4, SP
     Move NEWITEM, (SP)

  2) The pop operation can be implemented as

     Move (SP), ITEM
     Add #4, SP
- Routine for a safe pop and push operation as follows:



SAFEPOP	Compare Branch>0	#2000,SP EMPTYERROR	Check to see if the stack pointer contains an address value greater than 2000. If it does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action.
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

(a) Routine for a safe pop operation

SAFEPUSH	Compare Branch≤0	#1500,SP FULLERROR	Check to see if the stack pointer contains an address value equal to or less than 1500. If it does, the stack is full. Branch to the routine FULLERROR for appropriate action.
	Move	NEWITEM,–(SP)	Otherwise, push the element in memory location NEWITEM onto the stack.

(b) Routine for a safe push operation

Figure 2.23 Checking for empty and full errors in pop and push operations.

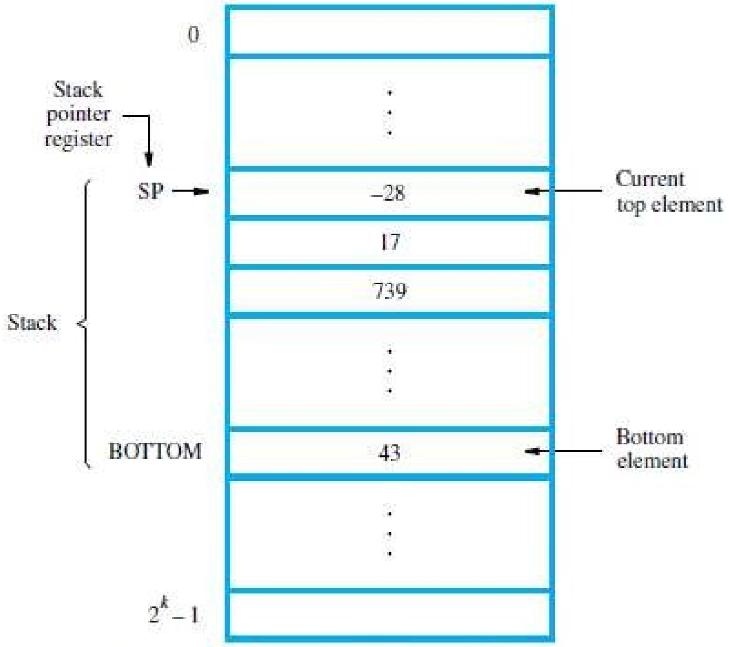


Figure 2.14 A stack of words in the memory.

QUEUE

- Data are stored in and retrieved from a queue on a FIFO basis.
- Difference between stack and queue?

  1) One end of the stack is fixed while the other end rises and falls as data are pushed and popped.

  2) In stack, a single pointer is needed to keep track of top of the stack at any given time.

     In queue, two pointers are needed to keep track of both the front and end for removal and insertion respectively.

  3) Without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a circular buffer.

SUBROUTINES

- A subtask consisting of a set of instructions which is executed many times is called a **Subroutine**.
- A Call instruction causes a branch to the subroutine (Figure: 2.16).
- At the end of the subroutine, a return instruction is executed
- Program resumes execution at the instruction immediately following the subroutine call
- The way in which a computer makes it possible to call and return from subroutines is referred to as its **Subroutine Linkage** method.
- The simplest subroutine linkage method is to save the return-address in a specific location, which may be a register dedicated to this function. Such a register is called the **Link Register**.
- When the subroutine completes its task, the Return instruction returns to the calling-program by branching indirectly through the link-register.
- The **Call Instruction** is a special branch instruction that performs the following operations:

  → Store the contents of PC into link-register.

  → Branch to the target-address specified by the instruction.
- The **Return Instruction** is a special branch instruction that performs the operation:

  → Branch to the address contained in the link-register.

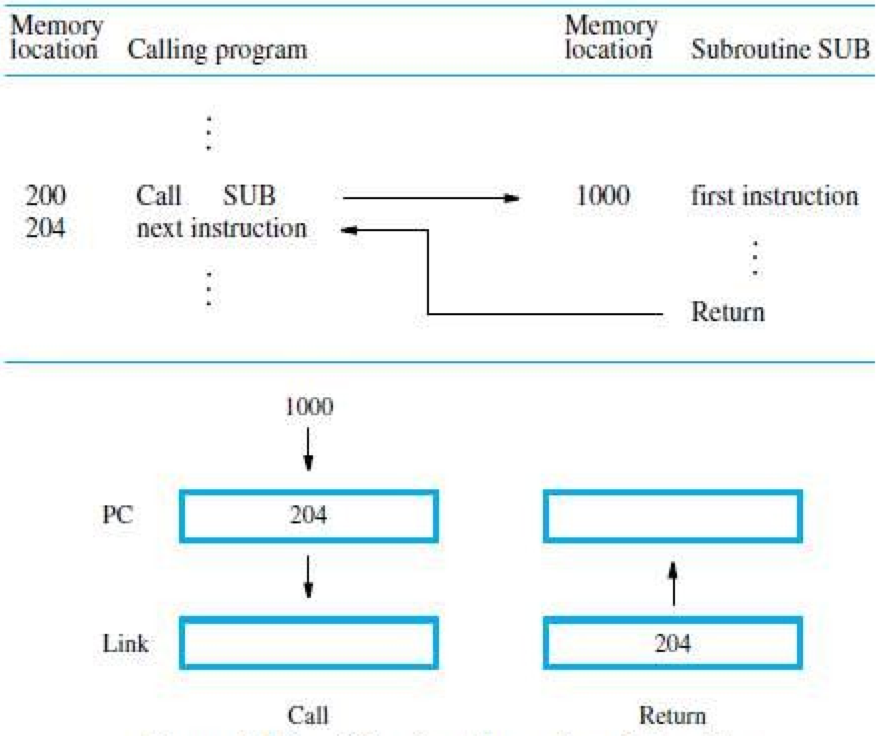


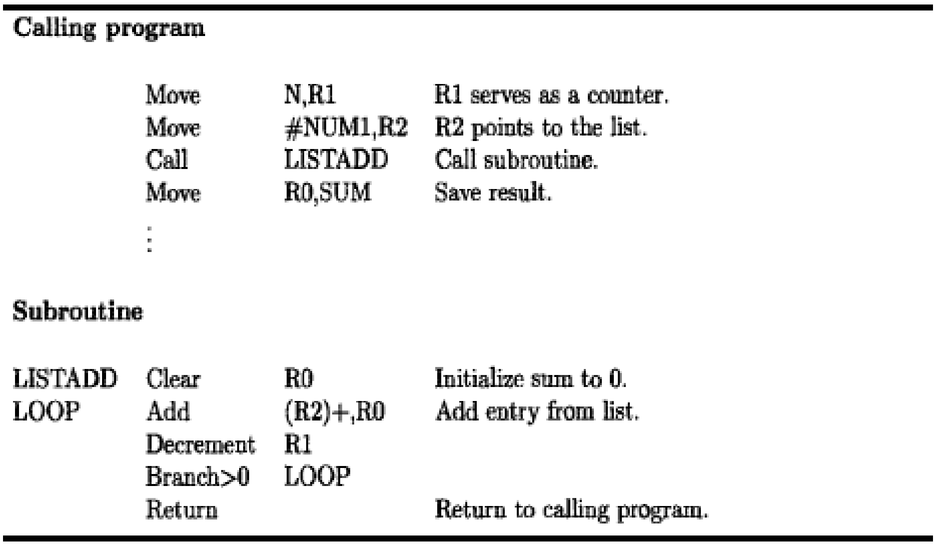
Figure 2.16 Subroutine linkage using a link register.

SUBROUTINE NESTING AND THE PROCESSOR STACK

- **Subroutine Nesting** means one subroutine calls another subroutine.
- In this case, the return-address of the second call is also stored in the link-register, destroying its previous contents.
- Hence, it is essential to save the contents of the link-register in some other location before calling another subroutine. Otherwise, the return-address of the first subroutine will be lost.
- Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it.
- The return-address needed for this first return is the last one generated in the nested call sequence. That is, return-addresses are generated and used in a LIFO order.
- This suggests that the return-addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the SP(Stack Pointer) to be used in this operation.
- SP is used to point to the processor-stack.
- Call instruction pushes the contents of the PC onto the processor-stack.
  Return instruction pops the return-address from the processor-stack into the PC.

PARAMETER PASSING

- The exchange of information between a calling-program and a subroutine is referred to as **Parameter Passing** (Figure: 2.25).
- The parameters may be placed in registers or in memory-location, where they can be accessed by the subroutine.
- Alternatively, parameters may be placed on the processor-stack used for saving the return-address.
- Following is a program for adding a list of numbers using subroutine with the parameters passed through registers.



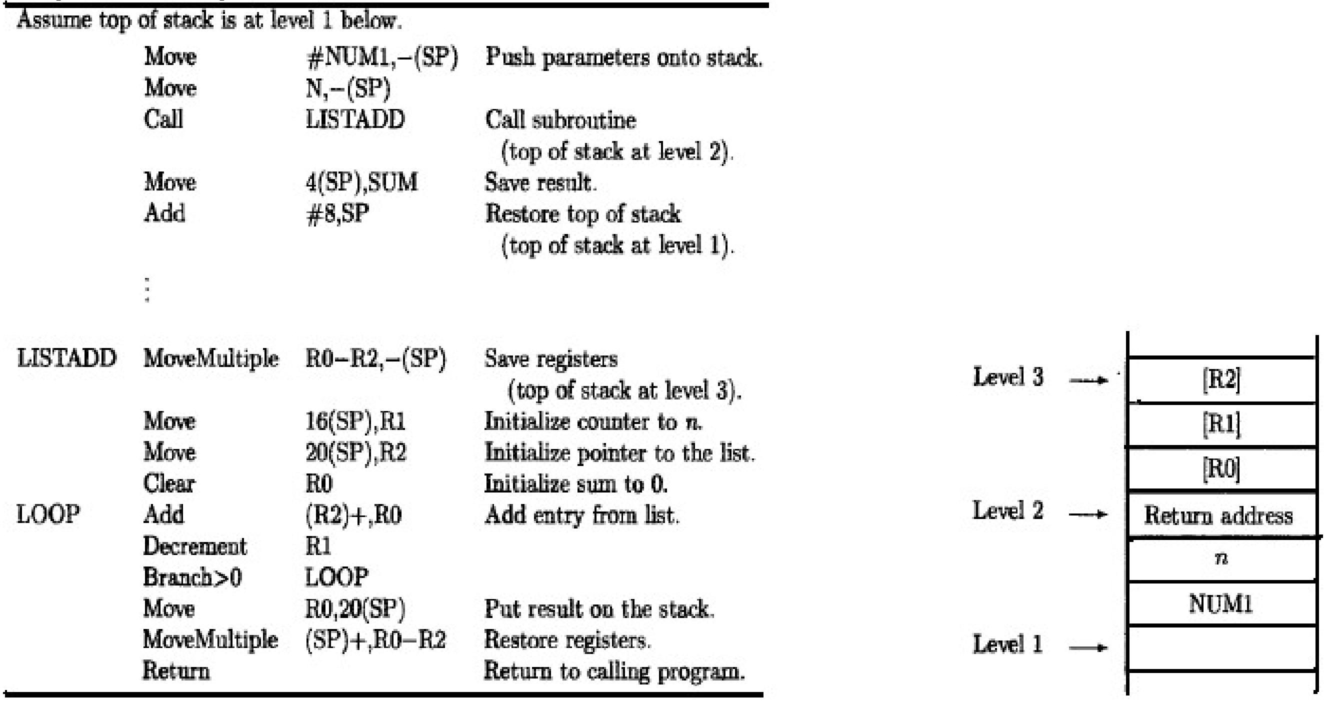
```
Calling program

            Move        N,R1         R1 serves as a counter.
            Move        #NUM1,R2     R2 points to the list.
            Call        LISTADD      Call subroutine.
            Move        R0,SUM       Save result.
            .
            .
            .

Subroutine

LISTADD     Clear       R0           Initialize sum to 0.
LOOP        Add         (R2)+,R0     Add entry from list.
            Decrement   R1
            Branch>0    LOOP
            Return                   Return to calling program.
```

Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.

STACK FRAME

- **Stack Frame** refers to locations that constitute a private work-space for the subroutine.
- The work-space is
  → created at the time the subroutine is entered &
  → freed up when the subroutine returns control to the calling-program (Figure: 2.26).

Program for adding a list of numbers using subroutine with the parameters passed to stack.



```
Assume top of stack is at level 1 below.
          Move          #NUM1,–(SP)   Push parameters onto stack.
          Move          N,–(SP)
          Call          LISTADD       Call subroutine
                                        (top of stack at level 2).
          Move          4(SP),SUM     Save result.
          Add           #8,SP         Restore top of stack
                                        (top of stack at level 1).
          ⋮

LISTADD   MoveMultiple  R0–R2,–(SP)   Save registers
                                        (top of stack at level 3).
          Move          16(SP),R1     Initialize counter to n.
          Move          20(SP),R2     Initialize pointer to the list.
          Clear         R0            Initialize sum to 0.
LOOP      Add           (R2)+,R0      Add entry from list.
          Decrement     R1
          Branch>0      LOOP
          Move          R0,20(SP)     Put result on the stack.
          MoveMultiple  (SP)+,R0–R2   Restore registers.
          Return                      Return to calling program.
```

(a) Calling program and subroutine

(b) Top of stack at various times

Figure 2.26 Program of Figure 2.16 written as a subroutine; parameters passed on the stack.

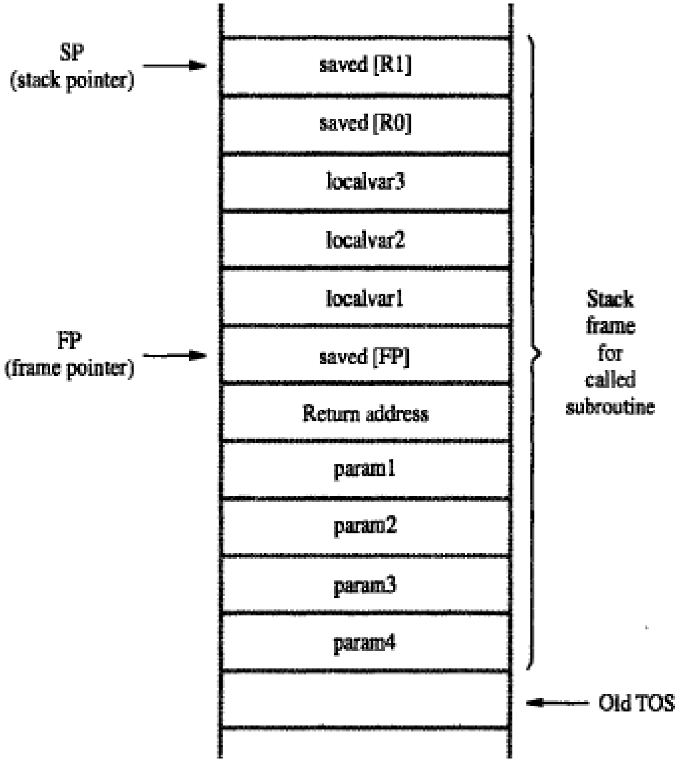


Figure 2.27 A subroutine stack frame example.

- Fig: 2.27 show an example of a commonly used layout for information in a stack-frame.

- **Frame Pointer (FP)** is used to access the parameters passed
  → to the subroutine &
  → to the local memory-variables.
- The contents of FP remains fixed throughout the execution of the subroutine, unlike stack-pointer SP, which must always point to the current top element in the stack.

Operation on Stack Frame

- Initially SP is pointing to the address of oldTOS.
- The calling-program saves 4 parameters on the stack (Figure 2.27).
- The Call instruction is now executed, pushing the return-address onto the stack.
- Now, SP points to this return-address, and the first instruction of the subroutine is executed.
- Now, FP is to be initialized and its old contents have to be stored. Hence, the first 2 instructions in the subroutine are:

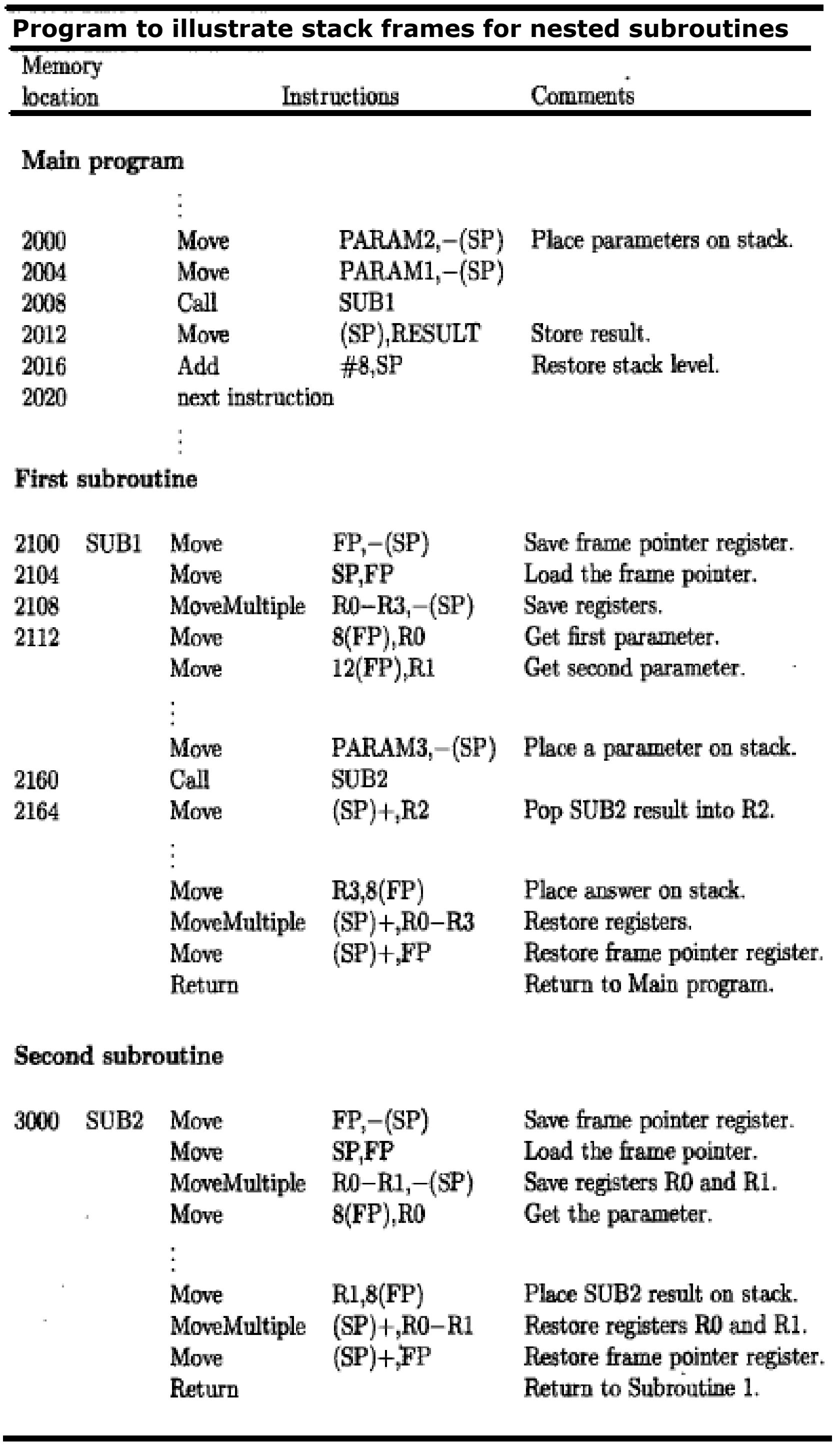
  Move FP,-(SP)
  Move SP,FP
- The FP is initialized to the value of SP i.e. both FP and SP point to the saved FP address.
- The 3 local variables may now be pushed onto the stack. Space for local variables is allocated by executing the instruction

  Subtract #12,SP
- Finally, the contents of processor-registers R0 and R1 are saved in the stack. At this point, the stack-frame has been set up as shown in the fig 2.27.
- The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction.

  Add #12, SP
- And subroutine pops saved old value of FP back into FP. At this point, SP points to return-address, so the Return instruction can be executed, transferring control back to the calling-program.

STACK FRAMES FOR NESTED SUBROUTINES

- Stack is very useful data structure for holding return-addresses when subroutines are nested.
- When nested subroutines are used; the stack-frames are built up in the processor-stack.



Program to illustrate stack frames for nested subroutines

```
Memory
location            Instructions                    Comments

Main program
                   :
2000             Move          PARAM2,-(SP)    Place parameters on stack.
2004             Move          PARAM1,-(SP)
2008             Call          SUB1
2012             Move          (SP),RESULT     Store result.
2016             Add           #8,SP           Restore stack level.
2020             next instruction
                   :
First subroutine

2100   SUB1      Move          FP,-(SP)        Save frame pointer register.
2104             Move          SP,FP           Load the frame pointer.
2108             MoveMultiple  R0-R3,-(SP)     Save registers.
2112             Move          8(FP),R0        Get first parameter.
                 Move          12(FP),R1       Get second parameter.
                   :
                 Move          PARAM3,-(SP)    Place a parameter on stack.
2160             Call          SUB2
2164             Move          (SP)+,R2        Pop SUB2 result into R2.
                   :
                 Move          R3,8(FP)        Place answer on stack.
                 MoveMultiple  (SP)+,R0-R3     Restore registers.
                 Move          (SP)+,FP        Restore frame pointer register.
                 Return                        Return to Main program.

Second subroutine

3000   SUB2      Move          FP,-(SP)        Save frame pointer register.
                 Move          SP,FP           Load the frame pointer.
                 MoveMultiple  R0-R1,-(SP)     Save registers R0 and R1.
                 Move          8(FP),R0        Get the parameter.
                   :
                 Move          R1,8(FP)        Place SUB2 result on stack.
                 MoveMultiple  (SP)+,R0-R1     Restore registers R0 and R1.
                 Move          (SP)+,FP        Restore frame pointer register.
                 Return                        Return to Subroutine 1.
```

Figure 2.28 Nested subroutines.

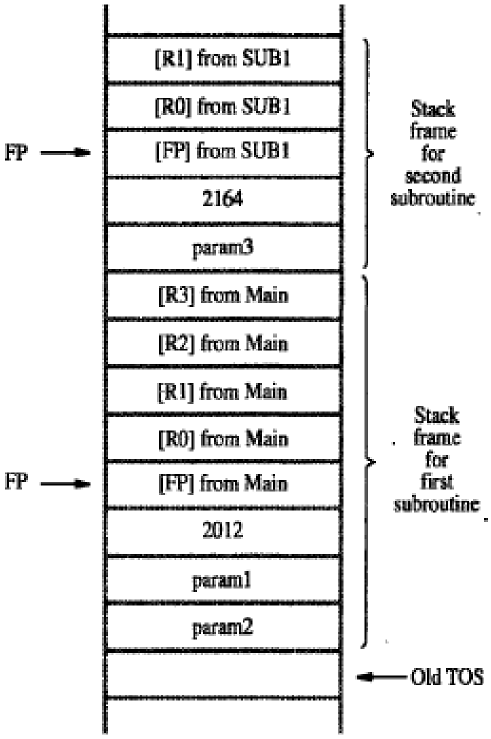


Figure 2.29 Stack frames for Figure 2.28.

The Flow of Execution is as follows:

- Main program pushes the 2 parameters param2 and param1 onto the stack and then calls SUB1.
- SUB1 has to perform an operation & send result to the main-program on the stack (Fig:2.28 & 29).
- During the process, SUB1 calls the second subroutine SUB2 (in order to perform some subtask).
- After SUB2 executes its Return instruction; the result is stored in register R2 by SUB1.
- SUB1 then continues its computations & eventually passes required answer back to main-program on the stack.
- When SUB1 executes return statement, the main-program stores this answers in memory-location RESULT and continues its execution.

LOGIC INSTRUCTIONS

- Logic operations such as AND, OR, and NOT applied to individual bits.
- These are the basic building blocks of digital-circuits.
- This is also useful to be able to perform logic operations is software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel.
- For example, the instruction

  Not dst

SHIFT AND ROTATE INSTRUCTIONS

- There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions.
- The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information.
- For general operands, we use a logical shift.

  For a number, we use an arithmetic shift, which preserves the sign of the number.

LOGICAL SHIFTS

- Two logical shift instructions are

  1) Shifting left (LShiftL) &

  2) Shifting right (LShiftR).
- These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.

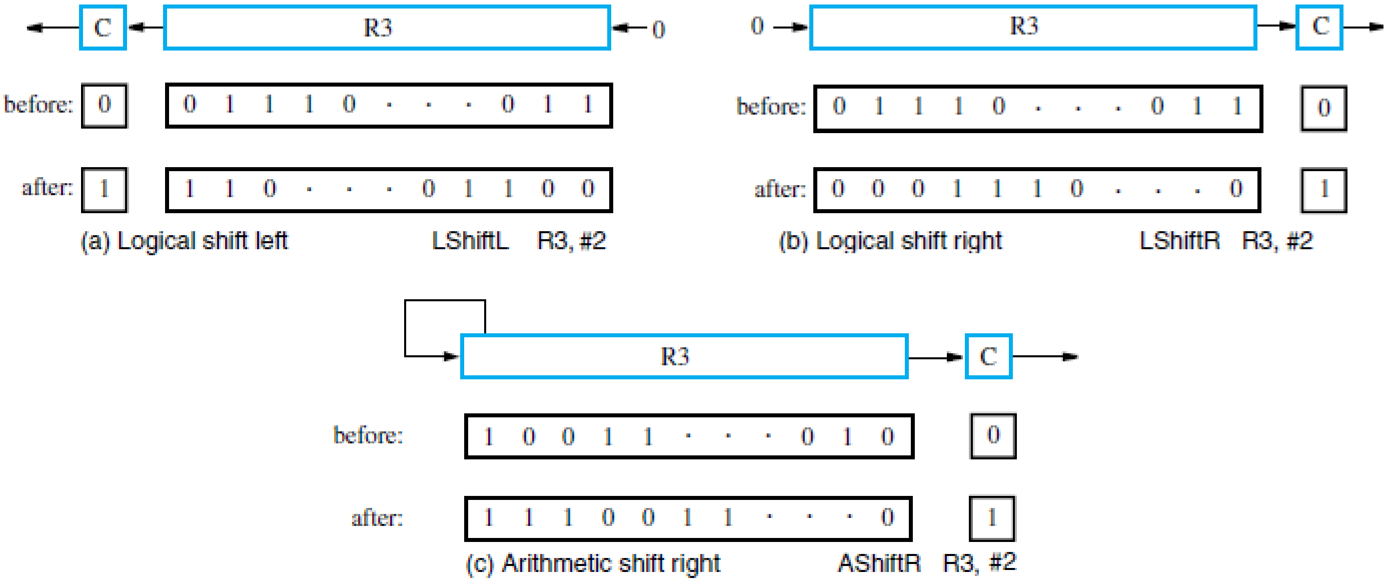
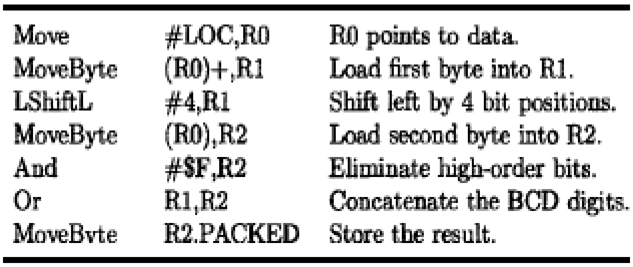


Figure 2.23 Logical and arithmetic shift instructions.



```
Move       #LOC,R0     R0 points to data.
MoveByte   (R0)+,R1    Load first byte into R1.
LShiftL    #4,R1       Shift left by 4 bit positions.
MoveByte   (R0),R2     Load second byte into R2.
And        #$F,R2      Eliminate high-order bits.
Or         R1,R2       Concatenate the BCD digits.
MoveBvte   R2.PACKED   Store the result.
```

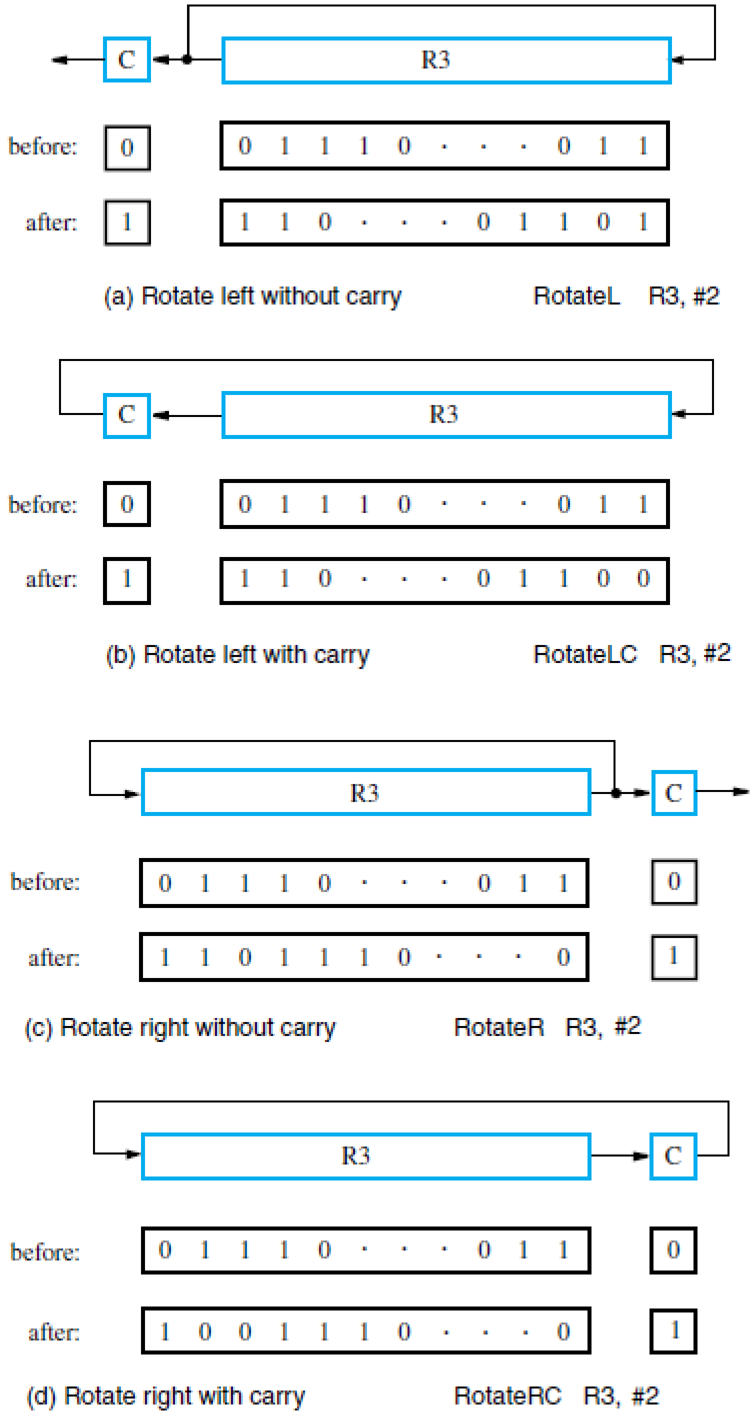
Figure 2.31 A routine that packs two BCD digits.

ROTATE OPERATIONS

- In shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry-flag C.
- To preserve all bits, a set of rotate instructions can be used.
- They move the bits that are shifted out of one end of the operand back into the other end.
- Two versions of both the left and right rotate instructions are usually provided.

  In one version, the bits of the operand is simply rotated.

  In the other version, the rotation includes the C flag.



(a) Rotate left without carry RotateL R3, #2

(b) Rotate left with carry RotateLC R3, #2

(c) Rotate right without carry RotateR R3, #2

(d) Rotate right with carry RotateRC R3, #2

Figure 2.25 Rotate instructions.

ENCODING OF MACHINE INSTRUCTIONS

- To be executed in a processor, an instruction must be encoded in a binary-pattern. Such encoded instructions are referred to as **Machine Instructions**.
- The instructions that use symbolic-names and acronyms are called *assembly language instructions*.
- We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers.
- Let us examine some typical cases.

The instruction

Add R1, R2 ;Has to specify the registers R1 and R2, in addition to the OP code. If the processor has 16 registers, then four bits are needed to identify each register. Additional bits are needed to indicate that the Register addressing-mode is used for each operand.

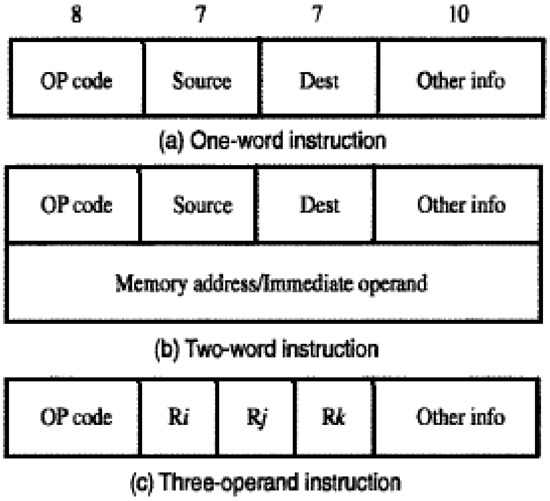
The instruction

Move 24(R0), R5 ;Requires 16 bits to denote the OP code and the two registers, and some bits to express that the source operand uses the Index addressing mode and that the index value is 24.

- In all these examples, the instructions can be encoded in a 32-bit word (Fig 2.39).
- The OP code for given instruction refers to type of operation that is to be performed.
- Source and destination field refers to source and destination operand respectively.
- The "Other info" field allows us to specify the additional information that may be needed such as an index value or an immediate operand.
- Using multiple words, we can implement complex instructions, closely resembling operations in high-level programming languages. The term complex instruction set computers (CISC) refers to processors that use
- CISC approach results in instructions of variable length, dependent on the number of operands and the type of addressing modes used.
- In RISC (reduced instruction set computers), any instruction occupies only one word.
- The RISC approach introduced other restrictions such as that all manipulation of data must be done on operands that are already in registers.

Ex: Add R1,R2,R3

- In RISC type machine, the memory references are limited to only Load/Store operations.



(a) One-word instruction

(b) Two-word instruction

(c) Three-operand instruction

Figure 2.39 Encoding instructions into 32-bit words.

Problem 1:

Write a program that can evaluate the expression A*B+C*D In a single-accumulator processor. Assume that the processor has Load, Store, Multiply, and Add instructions and that all values fit in the accumulator

Solution:

A program for the expression is:

```
Load A
Multiply B
Store RESULT
Load C
Multiply D
Add RESULT
Store RESULT
```

Problem 2:

Registers R1 and R2 of a computer contains the decimal values 1200 and 4600. What is the effective-address of the memory operand in each of the following instructions?

(a) Load 20(R1), R5
(b) Move #3000,R5
(c) Store R5,30(R1,R2)
(d) Add -(R2),R5
(e) Subtract (R1)+,R5

Solution:

(a) EA = [R1]+Offset=1200+20 = 1220
(b) EA = 3000
(c) EA = [R1]+[R2]+Offset = 1200+4600+30=5830
(d) EA = [R2]-1 = 4599
(e) EA = [R1] = 1200

Problem 3:

Registers R1 and R2 of a computer contains the decimal values 2900 and 3300. What is the effective-address of the memory operand in each of the following instructions?

(a) Load R1,55(R2)
(b) Move #2000,R7
(c) Store 95(R1,R2),R5
(d) Add (R1)+,R5
(e) Subtract-(R2),R5

Solution:

a) Load R1,55(R2) → This is indexed addressing mode. So EA = 55+R2=55+3300=3355.

b) Move #2000,R7 → This is an immediate addressing mode. So, EA = 2000

c) Store 95(R1,R2),R5 → This is a variation of indexed addressing mode, in which contents of 2 registers are added with the offset or index to generate EA. So, 95+R1+R2=95+2900+3300=6255.

d) Add (R1)+,R5 → This is Autoincrement mode. Contents of R1 are the EA so, 2900 is the EA.

e) Subtract -(R2),R5 → This is Auto decrement mode. Here, R2 is subtracted by 4 bytes (assuming 32-bt processor) to generate the EA, so, EA= 3300-4=3296.

Problem 4:

Given a binary pattern in some memory-location, is it possible to tell whether this pattern represents a machine instruction or a number?

Solution:

No; any binary pattern can be interpreted as a number or as an instruction.

Problem 5:

Both of the following statements cause the value 300 to be stored in location 1000, but at different times.

```
ORIGIN 1000
DATAWORD 300
```

And

```
Move #300,1000
```

Explain the difference.

Solution:

The assembler directives ORIGIN and DATAWORD cause the object program memory image constructed by the assembler to indicate that 300 is to be placed at memory word location 1000 at the time the program is loaded into memory prior to execution.

The Move instruction places 300 into memory word location 1000 when the instruction is executed as part of a program.

Problem 6:

Register R5 is used in a program to point to the top of a stack. Write a sequence of instructions using the Index, Autoincrement, and Autodecrement addressing modes to perform each of the following tasks:

(a) Pop the top two items off the stack, and them, and then push the result onto the stack.

(b) Copy the fifth item from the top into register R3.

(c) Remove the top ten items from the stack.

Solution:

```
(a)     Move (R5)+,R0
        Add (R5)+,R0
        Move R0,-(R5)
(b)     Move 16(R5),R3
(c)     Add #40,R5
```

Problem 7:

Consider the following possibilities for saving the return address of a subroutine:

(a) In the processor register.

(b) In a memory-location associated with the call, so that a different location is used when the subroutine is called from different places

(c) On a stack.

Which of these possibilities supports subroutine nesting and which supports subroutine recursion(that is, a subroutine that calls itself)?

Solution:

(a) Neither nesting nor recursion is supported.

(b) Nesting is supported, because different Call instructions will save the return address at different memory-locations. Recursion is not supported.

(c) Both nesting and recursion are supported.

MODULE 2: INPUT/OUTPUT ORGANIZATION

ACCESSING I/O-DEVICES

- A **single bus-structure** can be used for connecting I/O-devices to a computer (Figure 7.1).
- Each I/O device is assigned a unique set of address.
- Bus consists of 3 sets of lines to carry address, data & control signals.
- When processor places an address on address-lines, the intended-device responds to the command.
- The processor requests either a read or write-operation.
- The requested-data are transferred over the data-lines.

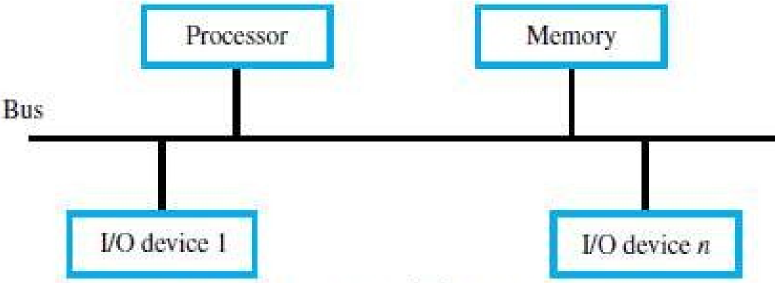


Figure 7.1 A single-bus structure.

- There are 2 ways to deal with I/O-devices: 1) Memory-mapped I/O & 2) I/O-mapped I/O.

1) Memory-Mapped I/O

- ➢ Memory and I/O-devices share a common address-space.
- ➢ Any data-transfer instruction (like Move, Load) can be used to exchange information.
- ➢ For example,

Move DATAIN, R0; This instruction sends the contents of location DATAIN to register R0.

Here, DATAIN → address of the input-buffer of the keyboard.

2) I/O-Mapped I/O

- ➢ Memory and I/0 address-spaces are different.
- ➢ A special instructions named **IN** and **OUT** are used for data-transfer.
- ➢ Advantage of separate I/O space: I/O-devices deal with fewer address-lines.

I/O Interface for an Input Device

1) Address Decoder: enables the device to recognize its address when this address appears on the address-lines (Figure 7.2).

2) Status Register: contains information relevant to operation of I/O-device.

3) Data Register: holds data being transferred to or from processor. There are 2 types:

i) DATAIN → Input-buffer associated with keyboard.

ii) DATAOUT → Output data buffer of a display/printer.

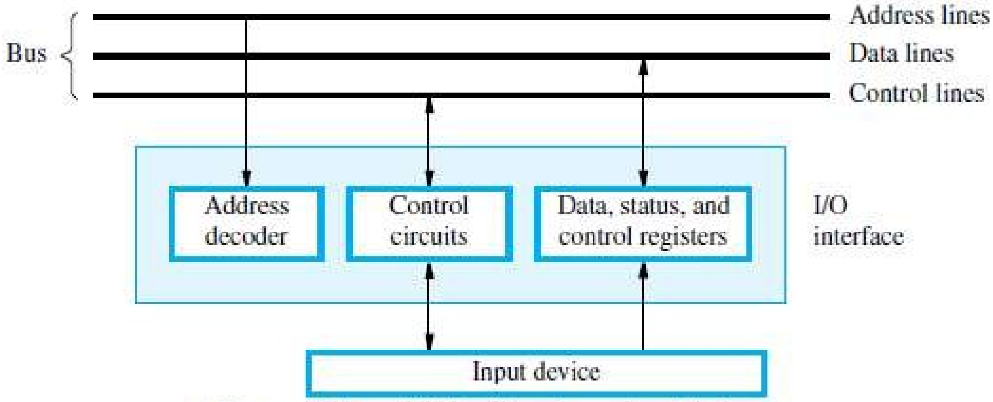


Figure 7.2 I/O interface for an input device.

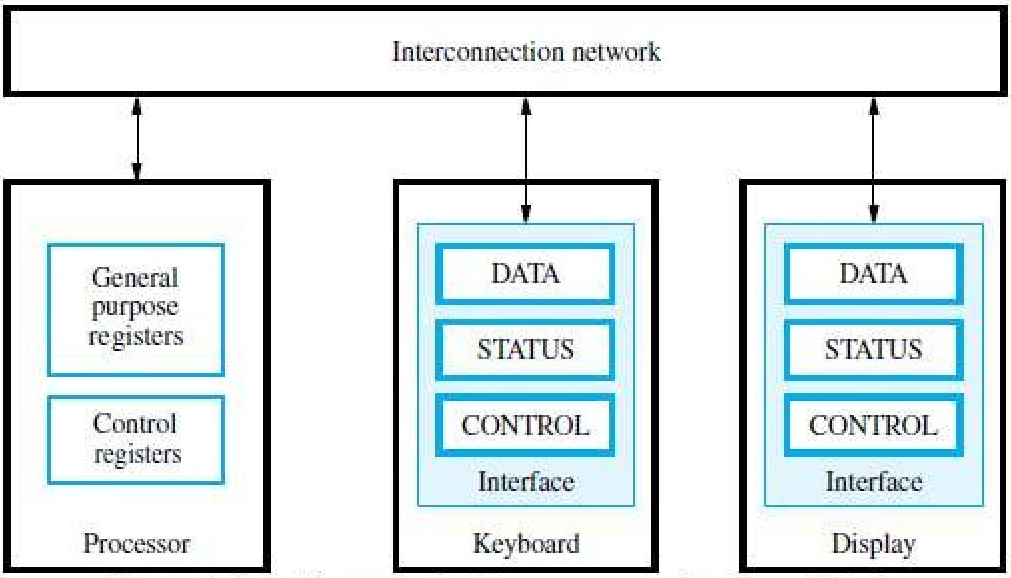


Figure 3.2 The connection for processor, keyboard, and display.

MECHANISMS USED FOR INTERFACING I/O-DEVICES

1) Program Controlled I/O

- Processor repeatedly checks status-flag to achieve required synchronization b/w processor & I/O device. (We say that the processor polls the device).
- Main drawback:

  The processor wastes time in checking status of device before actual data-transfer takes place.

2) Interrupt I/O

- I/O-device initiates the action instead of the processor.
- I/O-device sends an INTR signal over bus whenever it is ready for a data-transfer operation.
- Like this, required synchronization is done between processor & I/O device.

3) Direct Memory Access (DMA)

- Device-interface transfer data directly to/from the memory w/o continuous involvement by the processor.
- DMA is a technique used for high speed I/O-device.

INTERRUPTS

- There are many situations where other tasks can be performed while waiting for an I/O device to become ready.
- A hardware signal called an Interrupt will alert the processor when an I/O device becomes ready.
- Interrupt-signal is sent on the interrupt-request line.
- The processor can be performing its own task without the need to continuously check the I/O-device.
- The routine executed in response to an interrupt-request is called ISR.
- The processor must inform the device that its request has been recognized by sending INTA signal.
  (INTR → Interrupt Request, INTA → Interrupt Acknowledge, ISR → Interrupt Service Routine)
- For example, consider COMPUTE and PRINT routines (Figure 3.6).

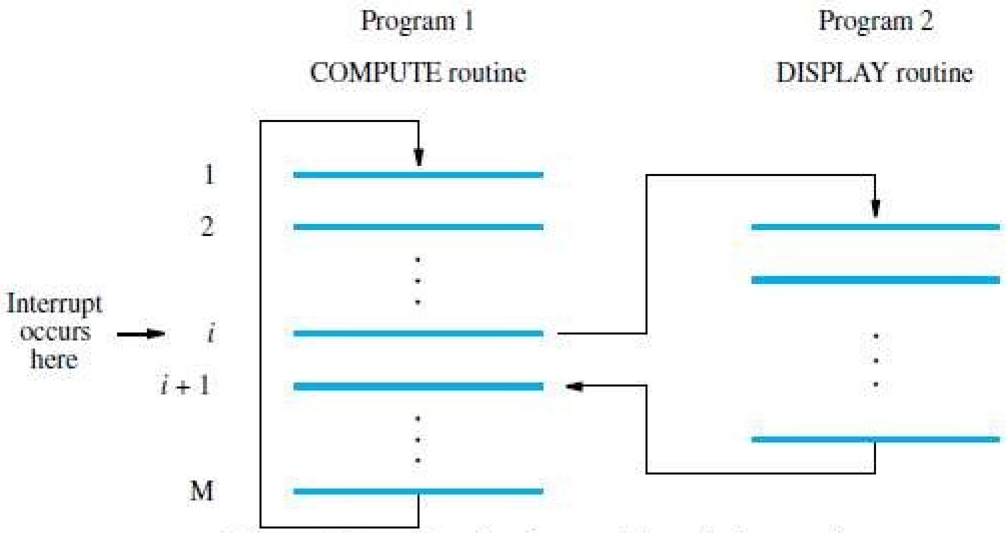


Figure 3.6 Transfer of control through the use of interrupts.

- The processor first completes the execution of instruction i.
- Then, processor loads the PC with the address of the first instruction of the ISR.
- After the execution of ISR, the processor has to come back to instruction i+1.
- Therefore, when an interrupt occurs, the current content of PC is put in temporary storage location.
- A return at the end of ISR reloads the PC from that temporary storage location.
- This causes the execution to resume at instruction i+1.
- When processor is handling interrupts, it must inform device that its request has been recognized.
- This may be accomplished by INTA signal.
- The task of saving and restoring the information can be done automatically by the processor.
- The processor saves only the contents of **PC & Status register.**
- Saving registers also increases the Interrupt Latency.
- **Interrupt Latency** is a delay between
  → time an interrupt-request is received and
  → start of the execution of the ISR.
- Generally, the long interrupt latency in unacceptable.

Difference between Subroutine & ISR

Subroutine	ISR
A subroutine performs a function required by the program from which it is called.	ISR may not have anything in common with program being executed at time INTR is received
Subroutine is just a linkage of 2 or more function related to each other.	Interrupt is a mechanism for coordinating I/O transfers.

INTERRUPT HARDWARE

- Most computers have several I/O devices that can request an interrupt.
- A single interrupt-request (IR) line may be used to serve n devices (Figure 4.6).
- All devices are connected to IR line via switches to ground.
- To request an interrupt, a device closes its associated switch.
- Thus, if all IR signals are inactive, the voltage on the IR line will be equal to V_{dd}.
- When a device requests an interrupt, the voltage on the line drops to 0.
- This causes the INTR received by the processor to go to 1.
- The value of INTR is the logical OR of the requests from individual devices.

$$\mathbf{INTR=INTR_1+ INTR_{2+} . . . . . +INTR_n}$$

- A special gates known as open-collector or open-drain are used to drive the INTR line.
- The Output of the open collector control is equal to a switch to the ground that is

  → open when gates input is in ″0‟ state and

  → closed when the gates input is in “1‟ state.
- Resistor R is called a **Pull-up Resistor** because

  it pulls the line voltage up to the high-voltage state when the switches are open.

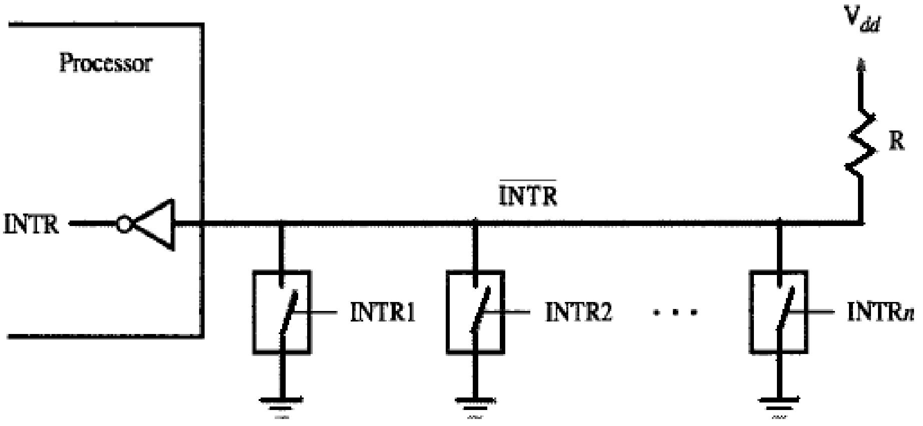


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

ENABLING & DISABLING INTERRUPTS

- All computers fundamentally should be able to enable and disable interruptions as desired.
- The problem of infinite loop occurs due to successive interruptions of active INTR signals.
- There are 3 mechanisms to solve problem of infinite loop:
  1) Processor should ignore the interrupts until execution of first instruction of the ISR.
  2) Processor should automatically disable interrupts before starting the execution of the ISR.
  3) Processor has a special INTR line for which the interrupt-handling circuit.

     Interrupt-circuit responds only to leading edge of signal. Such line is called edge-triggered.
- Sequence of events involved in handling an interrupt-request:
  1) The device raises an interrupt-request.
  2) The processor interrupts the program currently being executed.
  3) Interrupts are disabled by changing the control bits in the processor status register (PS).
  4) The device is informed that its request has been recognized.

     In response, the device deactivates the interrupt-request signal.
  5) The action requested by the interrupt is performed by the interrupt-service routine.
  6) Interrupts are enabled and execution of the interrupted program is resumed.

HANDLING MULTIPLE DEVICES

- While handling multiple devices, the issues concerned are:
  1) How can the processor recognize the device requesting an interrupt?
  2) How can the processor obtain the starting address of the appropriate ISR?
  3) Should a device be allowed to interrupt the processor while another interrupt is being serviced?
  4) How should 2 or more simultaneous interrupt-requests be handled?

POLLING

- Information needed to determine whether device is requesting interrupt is available in status-register
- Following condition-codes are used:
  - ➢ DIRQ → Interrupt-request for display.
  - ➢ KIRQ → Interrupt-request for keyboard.
  - ➢ KEN → keyboard enable.
  - ➢ DEN → Display Enable.
  - ➢ SIN, SOUT → status flags.
- For an input device, SIN status flag in used.

SIN = 1 → when a character is entered at the keyboard.
SIN = 0 → when the character is read by processor.
IRQ=1 → when a device raises an interrupt-requests (Figure 4.3).

- Simplest way to identify interrupting-device is to have ISR poll all devices connected to bus.
- The first device encountered with its IRQ bit set is serviced.
- After servicing first device, next requests may be serviced.
- **Advantage:** Simple & easy to implement.

Disadvantage: More time spent polling IRQ bits of all devices.

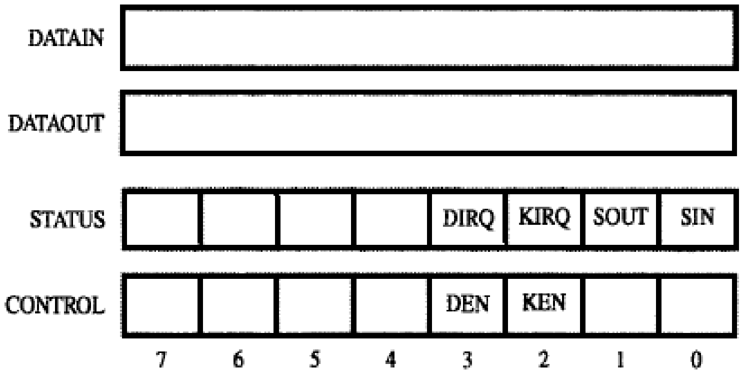
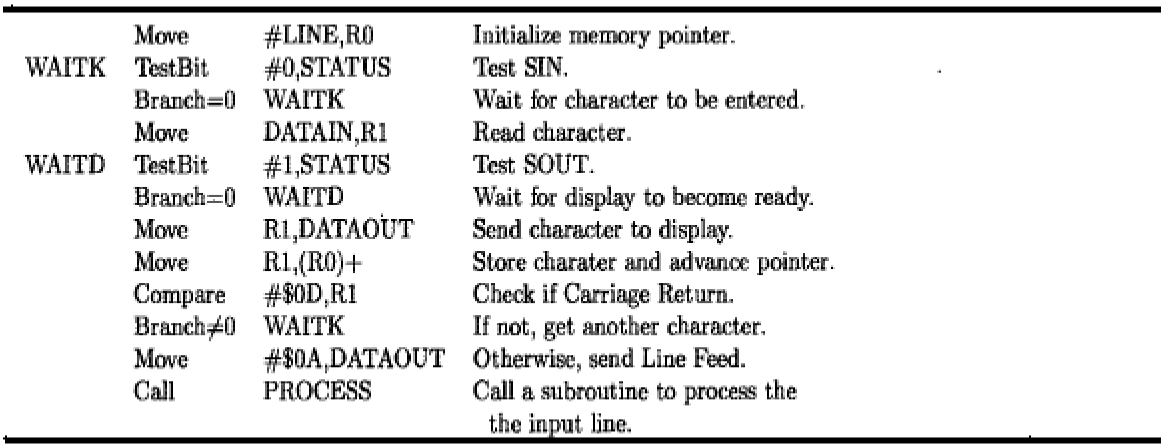


Figure 4.3 Registers in keyboard and display interfaces.



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

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

VECTORED INTERRUPTS

- A device requesting an interrupt identifies itself by sending a special-code to processor over bus.
- Then, the processor starts executing the ISR.
- The special-code indicates starting-address of ISR.
- The special-code length ranges from 4 to 8 bits.
- The location pointed to by the interrupting-device is used to store the staring address to ISR.
- The staring address to ISR is called the **interrupt vector**.
- Processor
  → loads interrupt-vector into PC &
  → executes appropriate ISR.
- When processor is ready to receive interrupt-vector code, it activates INTA line.
- Then, I/O-device responds by sending its interrupt-vector code & turning off the INTR signal.
- The interrupt vector also includes a new value for the Processor Status Register.

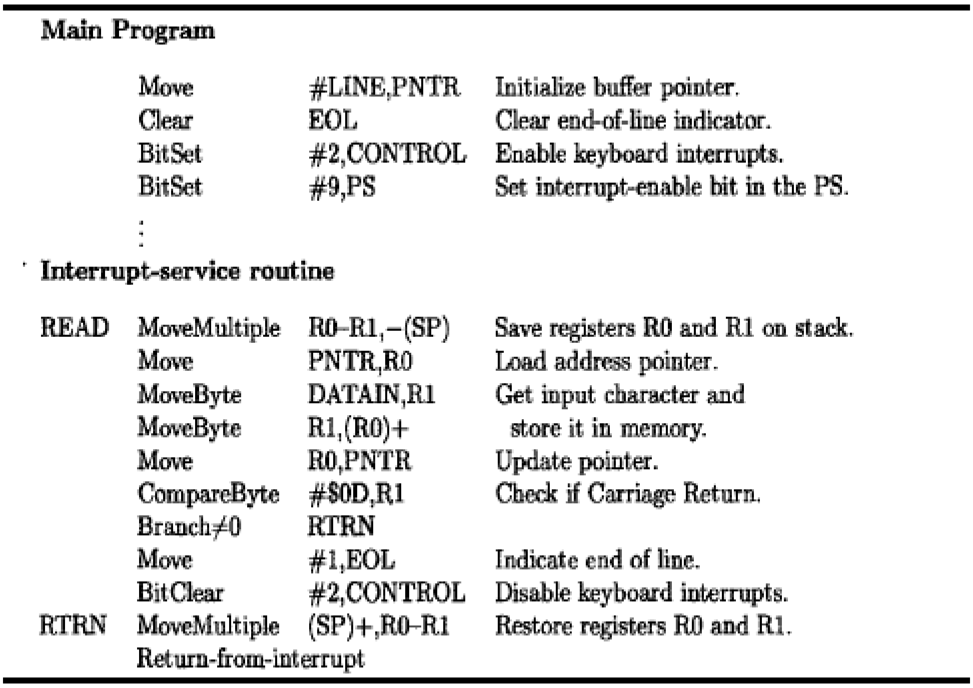
CONTROLLING DEVICE REQUESTS

- Following condition-codes are used:
  - KEN → Keyboard Interrupt Enable.
  - DEN → Display Interrupt Enable.
  - KIRQ/DIRQ → Keyboard/Display unit requesting an interrupt.
- There are 2 independent methods for controlling interrupt-requests. (IE → interrupt-enable).

1) At Device-end

IE bit in a control-register determines whether device is allowed to generate an interrupt-request.

2) At Processor-end, interrupt-request is determined by
→ IE bit in the PS register or
→ Priority structure



```
Main Program

        Move            #LINE,PNTR      Initialize buffer pointer.
        Clear           EOL             Clear end-of-line indicator.
        BitSet          #2,CONTROL      Enable keyboard interrupts.
        BitSet          #9,PS           Set interrupt-enable bit in the PS.
          :
          :
Interrupt-service routine

READ    MoveMultiple    R0–R1,–(SP)     Save registers R0 and R1 on stack.
        Move            PNTR,R0         Load address pointer.
        MoveByte        DATAIN,R1       Get input character and
        MoveByte        R1,(R0)+          store it in memory.
        Move            R0,PNTR         Update pointer.
        CompareByte     #$0D,R1         Check if Carriage Return.
        Branch≠0        RTRN
        Move            #1,EOL          Indicate end of line.
        BitClear        #2,CONTROL      Disable keyboard interrupts.
RTRN    MoveMultiple    (SP)+,R0–R1     Restore registers R0 and R1.
        Return-from-interrupt
```

Figure 4.9 Using interrupts to read a line of characters from a keyboard via the registers in Figure 4.3.

INTERRUPT NESTING

- A multiple-priority scheme is implemented by using separate INTR & INTA lines for each device
- Each INTR line is assigned a different priority-level (Figure 4.7).
- Priority-level of processor is the priority of program that is currently being executed.
- Processor accepts interrupts only from devices that have higher-priority than its own.
- At the time of execution of ISR for some device, priority of processor is raised to that of the device.
- Thus, interrupts from devices at the same level of priority or lower are disabled.

Privileged Instruction

- Processor's priority is encoded in a few bits of PS word. (PS → Processor-Status).
- Encoded-bits can be changed by **Privileged Instructions** that write into PS.
- Privileged-instructions can be executed only while processor is running in **Supervisor Mode**.
- Processor is in supervisor-mode only when executing operating-system routines.

Privileged Exception

- User program cannot
  - → accidently or intentionally change the priority of the processor &
  - → disrupt the system-operation.
- An attempt to execute a privileged-instruction while in user-mode leads to a **Privileged Exception**.

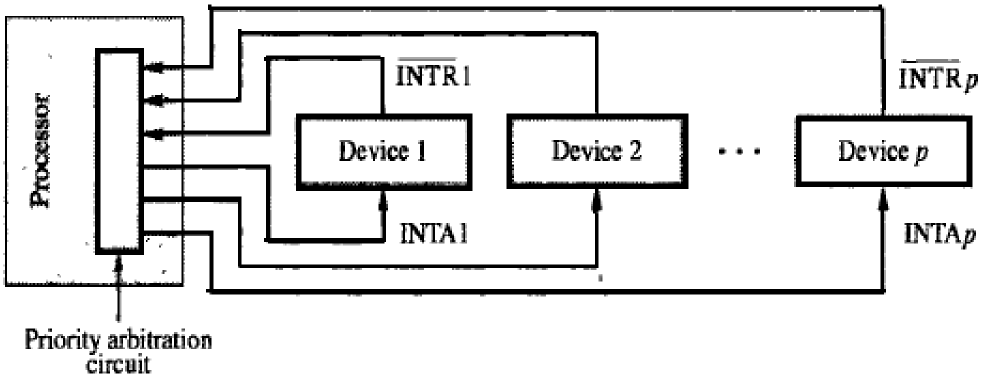

Processor
$\overline{\text{INTR}}1$
$\overline{\text{INTR}}p$
Device 1
Device 2
Device p
INTA1
INTAp
Priority arbitration circuit

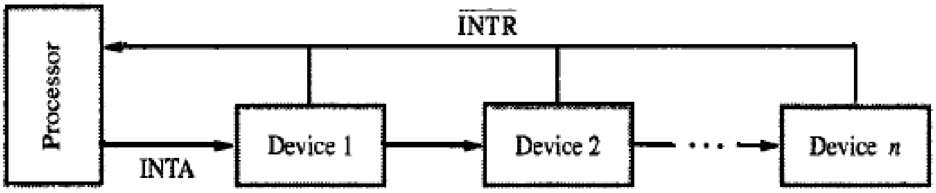
Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

SIMULTANEOUS REQUESTS

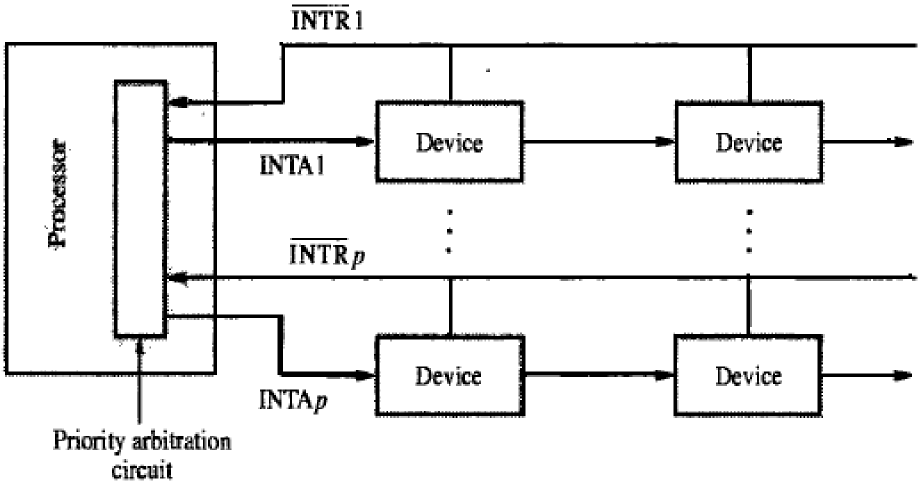
- The processor must have some mechanisms to decide which request to service when simultaneous requests arrive.
- INTR line is common to all devices (Figure 4.8a).
- INTA line is connected in a daisy-chain fashion.
- INTA signal propagates serially through devices.
- When several devices raise an interrupt-request, INTR line is activated.
- Processor responds by setting INTA line to 1. This signal is received by device 1.
- Device-1 passes signal on to device 2 only if it does not require any service.
- If device-1 has a pending-request for interrupt, the device-1
  → blocks INTA signal &
  → proceeds to put its identifying-code on data-lines.
- Device that is electrically closest to processor has highest priority.
- **Advantage:** It requires fewer wires than the individual connections.

Arrangement of Priority Groups

- Here, the devices are organized in groups & each group is connected at a different priority level.
- Within a group, devices are connected in a daisy chain. (Figure 4.8b).



(a) Daisy chain


$\overline{\text{INTR}}$1
Processor
Device
Device
INTA1
$\overline{\text{INTR}}p$
Device
Device
INTAp
Priority arbitration circuit

(b) Arrangement of priority groups

Figure 4.8 Interrupt priority schemes.

EXCEPTIONS

- An **interrupt** is an event that causes
  → execution of one program to be suspended &
  → execution of another program to begin.
- **Exception** refers to any event that causes an interruption. For ex: I/O interrupts.

1. Recovery from Errors

- These are techniques to ensure that all hardware components are operating properly.
- For ex: Many computers include an ECC in memory which allows detection of errors in stored-data.
  (ECC → Error Checking Code, ESR → Exception Service Routine).
- If an error occurs, control-hardware
  → detects the errors &
  → informs processor by raising an interrupt.
- When exception processing is initiated (as a result of errors), processor.
  → suspends program being executed &
  → starts an ESR. This routine takes appropriate action to recover from the error.

2. Debugging

- Debugger
  → is used to find errors in a program and
  → uses exceptions to provide 2 important facilities: i) Trace & ii) Breakpoints

i) Trace

- When a processor is operating in trace-mode, an exception occurs after execution of every instruction (using debugging-program as ESR).
- Debugging-program enables user to examine contents of registers, memory-locations and so on.
- On return from debugging-program,
  next instruction in program being debugged is executed,
  then debugging-program is activated again.
- The trace exception is disabled during the execution of the debugging-program.

ii) Breakpoints

- Here, the program being debugged is interrupted only at specific points selected by user.
- An instruction called Trap (or Software interrupt) is usually provided for this purpose.
- When program is executed & reaches breakpoint, the user can examine memory & register contents.

3. Privilege Exception

- To protect OS from being corrupted by user-programs, **Privileged Instructions** are executed only while processor is in supervisor-mode.
- For e.g.
  When processor runs in user-mode, it will not execute instruction that change priority of processor.
- An attempt to execute privileged-instruction will produce a **Privilege Exception**.
- As a result, processor switches to supervisor-mode & begins to execute an appropriate routine in OS.

DIRECT MEMORY ACCESS (DMA)

- The transfer of a block of data directly b/w an external device & main-memory w/o continuous involvement by processor is called DMA.
- DMA controller
  - → is a control circuit that performs DMA transfers (Figure 8.13).
  - → is a part of the I/O device interface.
  - → performs the functions that would normally be carried out by processor.
- While a DMA transfer is taking place, the processor can be used to execute another program.

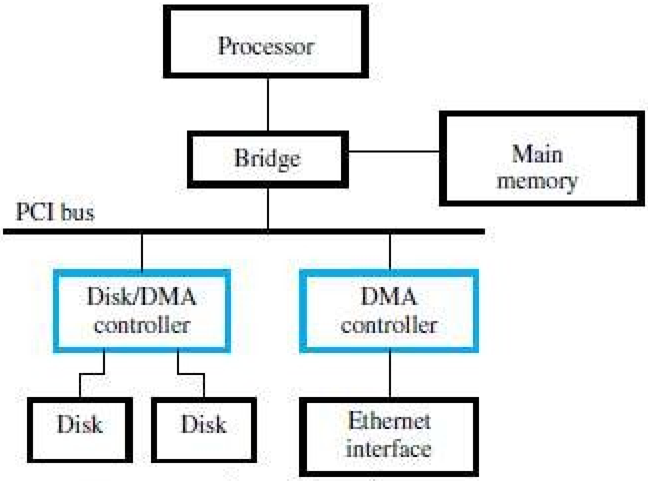


Figure 8.13 Use of DMA controllers in a computer system.

- DMA interface has three registers (Figure 8.12):
  1) First register is used for storing starting-address.
  2) Second register is used for storing word-count.
  3) Third register contains status- & control-flags.

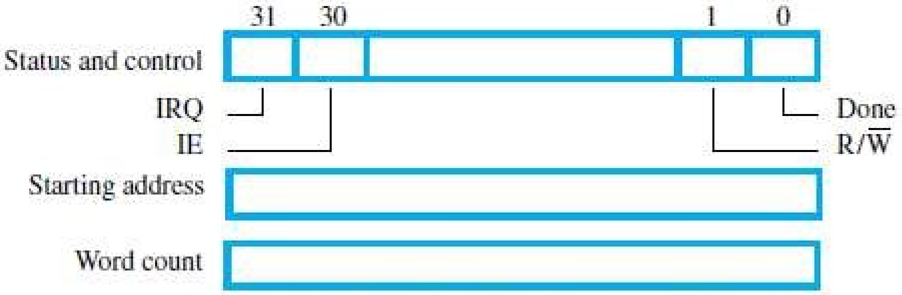


Figure 8.12 Typical registers in a DMA controller.

- The R/W bit determines direction of transfer.
  If R/W=1, controller performs a read-operation (i.e. it transfers data from memory to I/O),
  Otherwise, controller performs a write-operation (i.e. it transfers data from I/O to memory).
- If Done=1, the controller
  - → has completed transferring a block of data and
  - → is ready to receive another command. (IE → Interrupt Enable).
- If IE=1, controller raises an interrupt after it has completed transferring a block of data.
- If IRQ=1, controller requests an interrupt.
- Requests by DMA devices for using the bus are always given higher priority than processor requests.
- There are 2 ways in which the DMA operation can be carried out:
  1) Processor originates most memory-access cycles.
     - ➢ DMA controller is said to "steal" memory cycles from processor.
     - ➢ Hence, this technique is usually called **Cycle Stealing**.
  2) DMA controller is given exclusive access to main-memory to transfer a block of data without any interruption. This is known as **Block Mode** (or burst mode).

BUS ARBITRATION

- The device that is allowed to initiate data-transfers on bus at any given time is called **bus-master**.
- There can be only one bus-master at any given time.
- **Bus Arbitration** is the process by which
  - → next device to become the bus-master is selected &
  - → bus-mastership is transferred to that device.
- The two approaches are:
  - **1) Centralized Arbitration:** A single bus-arbiter performs the required arbitration.
  - **2) Distributed Arbitration:** All devices participate in selection of next bus-master.
- A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus at the same time to access the main-memory.
- To resolve this, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers.
- The bus arbiter may be the processor or a separate unit connected to the bus.

CENTRALIZED ARBITRATION

- A single bus-arbiter performs the required arbitration (Figure: 4.20).
- Normally, processor is the bus-master.
- Processor may grant bus-mastership to one of the DMA controllers.
- A DMA controller indicates that it needs to become bus-master by activating BR line.
- The signal on the BR line is the logical OR of bus-requests from all devices connected to it.
- Then, processor activates BG1 signal indicating to DMA controllers to use bus when it becomes free.
- BG1 signal is connected to all DMA controllers using a daisy-chain arrangement.
- If DMA controller-1 is requesting the bus,

  Then, DMA controller-1 blocks propagation of grant-signal to other devices.

  Otherwise, DMA controller-1 passes the grant downstream by asserting BG2.
- Current bus-master indicates to all devices that it is using bus by activating BBSY line.
- The bus-arbiter is used to coordinate the activities of all devices requesting memory transfers.
- Arbiter ensures that only 1 request is granted at any given time according to a priority scheme.

  (BR → Bus-Request, BG → Bus-Grant, BBSY → Bus Busy).

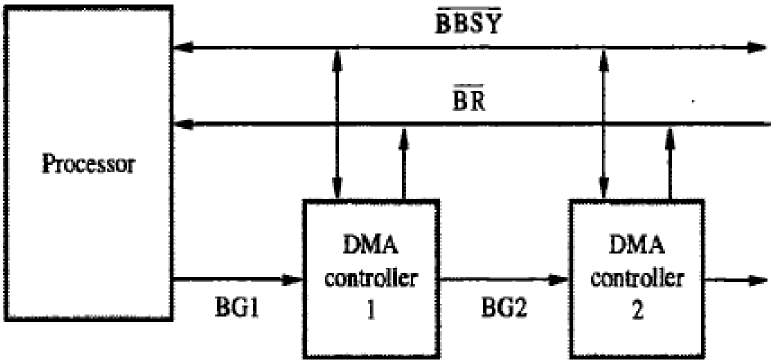


Figure 4.20 A simple arrangement for bus arbitration using a dai:y chain.

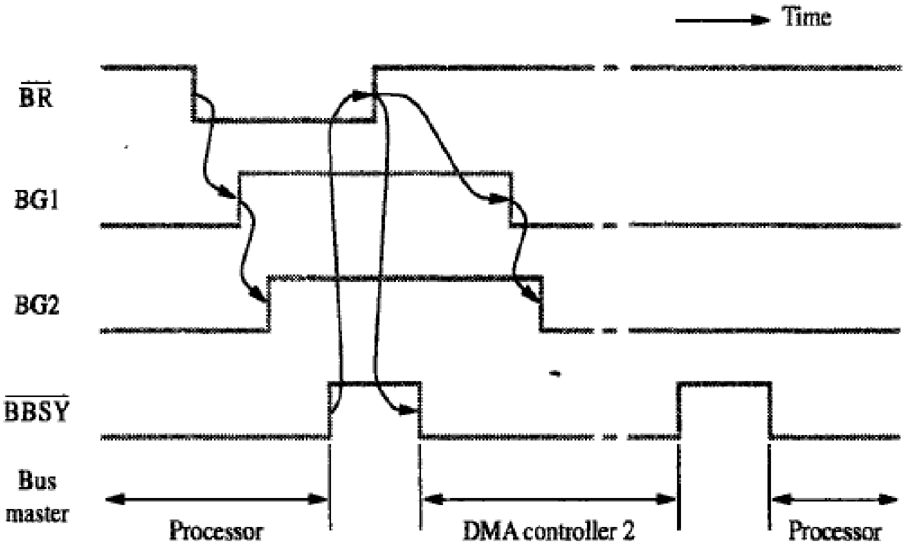


Figure 4.21 Sequence of signals during transfer of bus mastership for the devices in Figure 4.20.

- The timing diagram shows the sequence of events for the devices connected to the processor.
- DMA controller-2

  → requests and acquires bus-mastership and

  → later releases the bus. (Figure: 4.21).
- After DMA controller-2 releases the bus, the processor resources bus-mastership.

DISTRIBUTED ARBITRATION

- All device participate in the selection of next bus-master (Figure 4.22).
- Each device on bus is assigned a 4-bit identification number (ID).
- When 1 or more devices request bus, they
  - → assert Start-Arbitration signal &
  - → place their 4-bit ID numbers on four open-collector lines $\overline{ARB0}$ through $ARB3$.
- A winner is selected as a result of interaction among signals transmitted over these lines.
- Net-outcome is that the code on 4 lines represents request that has the highest ID number.
- **Advantage:**

  This approach offers higher reliability since operation of bus is not dependent on any single device.

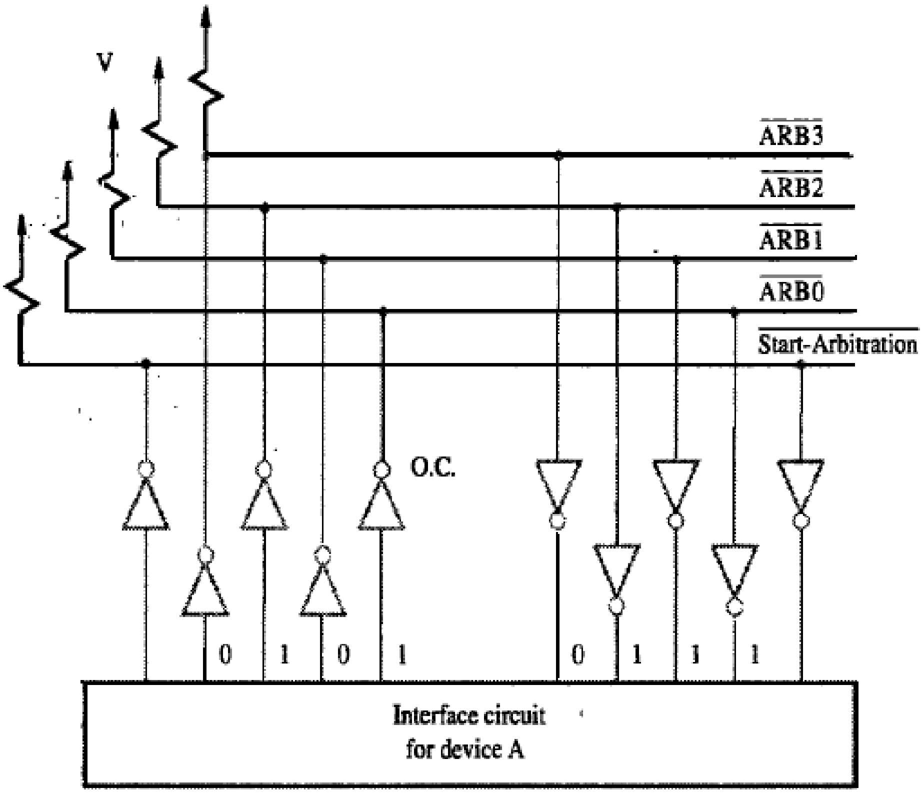


Figure 4.22 A distributed arbitration scheme.

For example:

- Assume 2 devices A & B have their ID 5 (0101), 6 (0110) and their code is 0111.
- Each device compares the pattern on the arbitration line to its own ID starting from MSB.
- If the device detects a difference at any bit position, it disables the drivers at that bit position.
- Driver is disabled by placing ″0″ at the input of the driver.
- In e.g. ‶A″ detects a difference in line ARB1, hence it disables the drivers on lines ARB1 & ARB0.
- This causes pattern on arbitration-line to change to 0110. This means that ‶B″ has won contention.

BUS

- Bus → is used to inter-connect main-memory, processor & I/O-devices
  → includes lines needed to support interrupts & arbitration.

- Primary function: To provide a communication-path for transfer of data.
- **Bus protocol** is set of rules that govern the behavior of various devices connected to the buses.
- Bus-protocol specifies parameters such as:
  → asserting control-signals
  → timing of placing information on bus
  → rate of data-transfer.
- A typical bus consists of 3 sets of lines:
  1) Address,
  2) Data &
  3) Control lines.
- Control-signals
  → specify whether a read or a write-operation is to be performed.
  → carry timing information i.e. they specify time at which I/O-devices place data on the bus.
- R/W line specifies
  → read-operation when R/W=1.
  → write-operation when R/W=0.
- During data-transfer operation,
  ➢ One device plays the role of a bus-master.
  ➢ Master-device initiates the data-transfer by issuing read/write command on the bus.
  ➢ The device addressed by the master is called as Slave.
- Two types of Buses: 1) Synchronous and 2) Asynchronous.

SYNCHRONOUS BUS

- All devices derive timing-information from a common clock-line.
- Equally spaced pulses on this line define equal time intervals.
- During a ″bus cycle‟, one data-transfer can take place.

A sequence of events during a read-operation

- At time t_0, the master (processor)
  - → places the device-address on address-lines &
  - → sends an appropriate command on control-lines (Figure 7.3).
- The command will
  - → indicate an input operation &
  - → specify the length of the operand to be read.
- Information travels over bus at a speed determined by physical & electrical characteristics.
- Clock pulse width(t_1-t_0) must be longer than max. propagation-delay b/w devices connected to bus.
- The clock pulse width should be long to allow the devices to decode the address & control signals.
- The slaves take no action or place any data on the bus before t_1.
- Information on bus is unreliable during the period $t_0$to t_1 because signals are changing state.
- Slave places requested input-data on data-lines at time t_1.
- At end of clock cycle (at time t_2), master strobes (captures) data on data-lines into its input-buffer
- For data to be loaded correctly into a storage device,

  data must be available at input of that device for a period greater than setup-time of device.

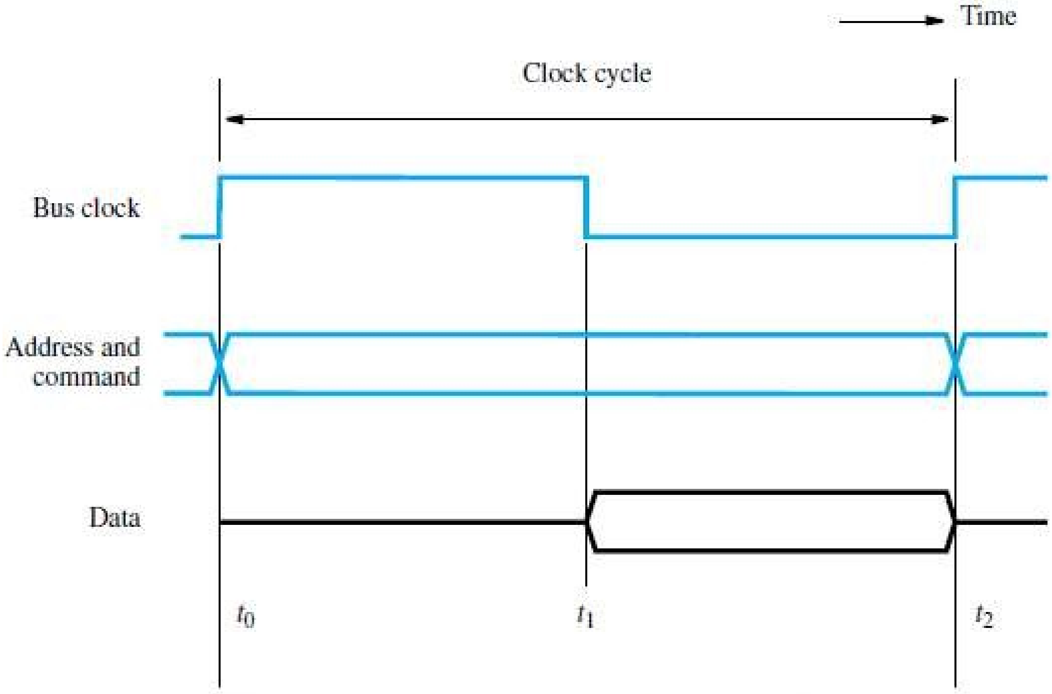


Figure 7.3 Timing of an input transfer on a synchronous bus.

A Detailed Timing Diagram for the Read-operation

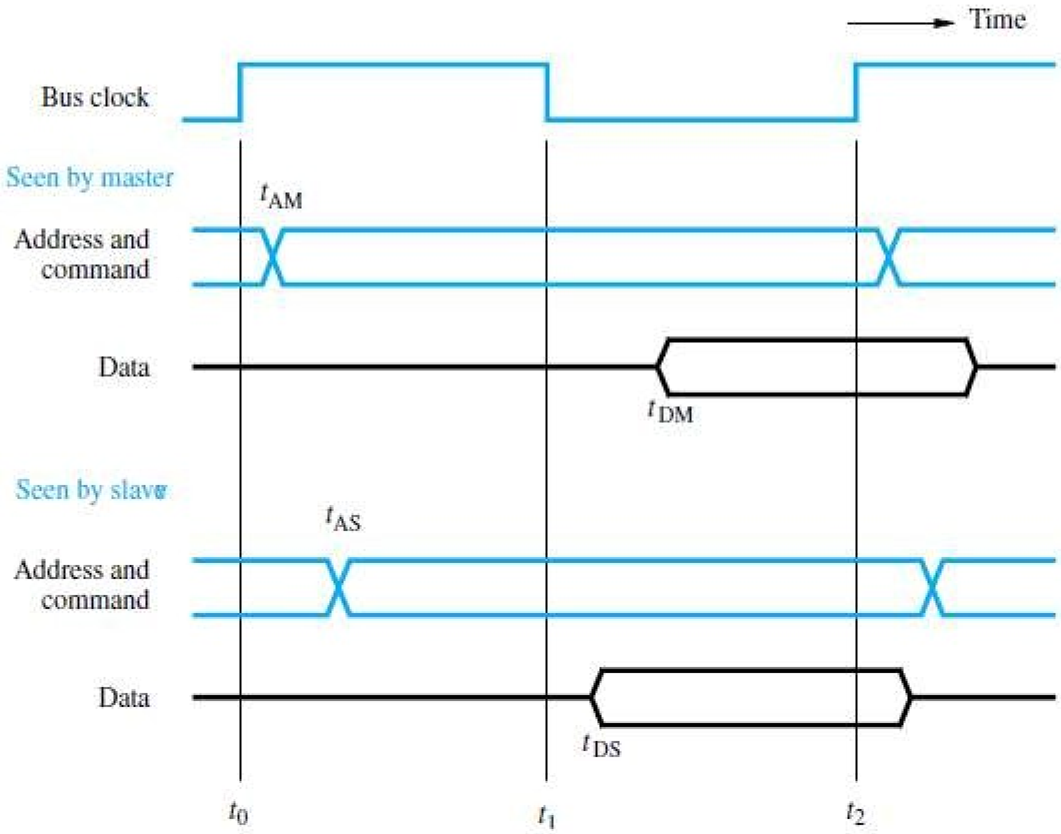


Figure 7.4 A detailed timing diagram for the input transfer of Figure 7.3.

- The picture shows two views of the signal except the clock (Figure 7.4).
- One view shows the signal seen by the master & the other is seen by the salve.
- Master sends the address & command signals on the rising edge at the beginning of clock period (t_0).
- These signals do not actually appear on the bus until t_{am}.
- Sometimes later, at t_{AS} the signals reach the slave.
- The slave decodes the address.
- At t_1, the slave sends the requested-data.
- At t_2, the master loads the data into its input-buffer.
- Hence the period t_2, t_{DM} is the setup time for the master"s input-buffer.
- The data must be continued to be valid after t_2, for a period equal to the hold time of that buffers.

Disadvantages

- The device does not respond.
- The error will not be detected.

Multiple Cycle Transfer for Read-operation

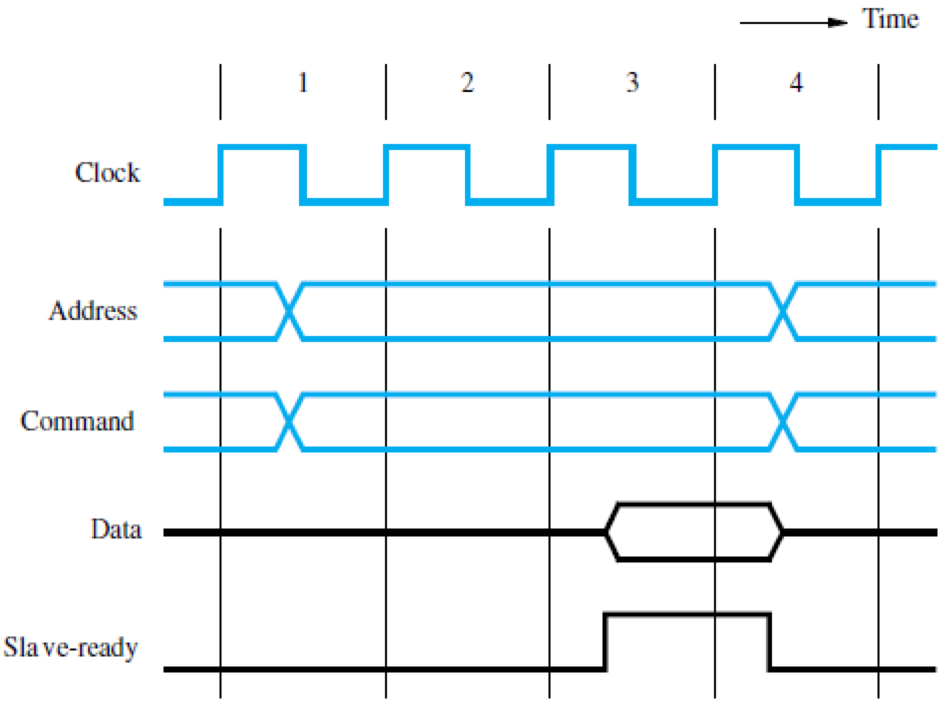


Figure 7.5 An input transfer using multiple clock cycles.

- During, clock cycle-1, master sends address/command info the bus requesting a "read" operation.
- The slave receives & decodes address/command information (Figure 7.5).
- At the active edge of the clock i.e. the beginning of clock cycle-2, it makes accession to respond immediately.
- The data become ready & are placed in the bus at clock cycle-3.
- At the same times, the slave asserts a control signal called **slave-ready**.
- The master strobes the data to its input-buffer at the end of clock cycle-3.
- The bus transfer operation is now complete.
- And the master sends a new address to start a new transfer in clock cycle4.
- The slave-ready signal is an acknowledgement from the slave to the master.

ASYNCHRONOUS BUS

- This method uses handshake-signals between master and slave for coordinating data-transfers.
- There are 2 control-lines:
  1) **Master-Ready (MR)** is used to indicate that master is ready for a transaction.
  2) **Slave-Ready (SR)** is used to indicate that slave is ready for a transaction.

The Read Operation proceeds as follows:

- At t_0, master places address/command information on bus.
- At t_1, master sets MR-signal to 1 to inform all devices that the address/command-info is ready.
  - ➢ MR-signal =1 → causes all devices on the bus to decode the address.
  - ➢ The delay $t_1 - t_0$ is intended to allow for any skew that may occurs on the bus.
  - ➢ Skew occurs when 2 signals transmitted from 1 source arrive at destination at different time
  - ➢ Therefore, the delay $t_1 - t_0$ should be larger than the maximum possible bus skew.
- At t_2, slave
  - → performs required input-operation &
  - → sets SR signal to 1 to inform all devices that it is ready (Figure 7.6).
- At t_3, SR signal arrives at master indicating that the input-data are available on bus.
- At t_4, master removes address/command information from bus.
- At t_5, when the device-interface receives the 1-to-0 transition of MR signal, it removes data and SR signal from the bus. This completes the input transfer.

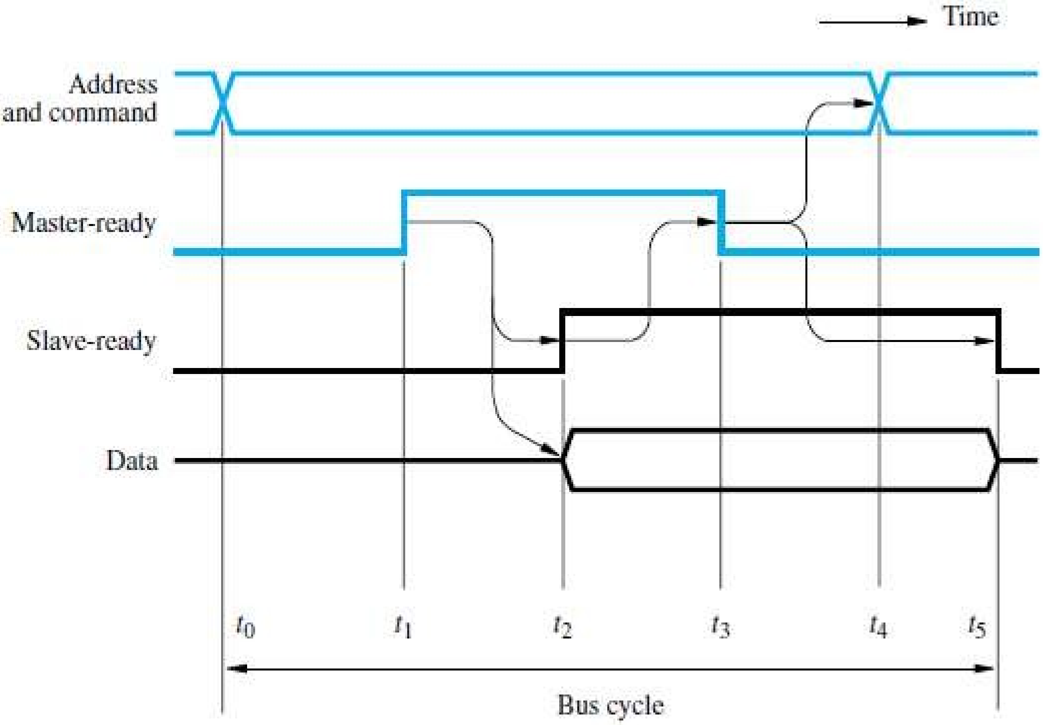


Figure 7.6 Handshake control of data transfer during an input operation.

- A change of state is one signal is followed by a change is the other signal. Hence this scheme is called as **Full Handshake**.
- **Advantage:** It provides the higher degree of flexibility and reliability.

MODULE 2: INPUT/OUTPUT ORGANIZATION (CONT.)

INTERFACE-CIRCUITS

- An **I/O Interface** consists of the circuitry required to connect an I/O device to a computer-bus.
- On one side of the interface, we have bus signals.

  On the other side, we have a data path with its associated controls to transfer data between the interface and the I/O device known as **port**.
- Two types are:

  1. Parallel Port transfers data in the form of a number of bits (8 or 16) simultaneously to or from the device.

  2. Serial Port transmits and receives data one bit at a time.
- Communication with the bus is the same for both formats.
- The conversion from the parallel to the serial format, and vice versa, takes place inside the interface-circuit.
- In parallel-port, the connection between the device and the computer uses

  → a multiple-pin connector and

  → a cable with as many wires.
- This arrangement is suitable for devices that are physically close to the computer.
- In serial port, it is much more convenient and cost-effective where longer cables are needed.

Functions of I/O Interface

1) Provides a storage buffer for at least one word of data.

2) Contains status-flags that can be accessed by the processor to determine whether the buffer is full or empty.

3) Contains address-decoding circuitry to determine when it is being addressed by the processor.

4) Generates the appropriate timing signals required by the bus control scheme.

5) Performs any format conversion that may be necessary to transfer data between the bus and the I/O device (such as parallel-serial conversion in the case of a serial port).

PARALLEL-PORT
KEYBOARD INTERFACED TO PROCESSOR

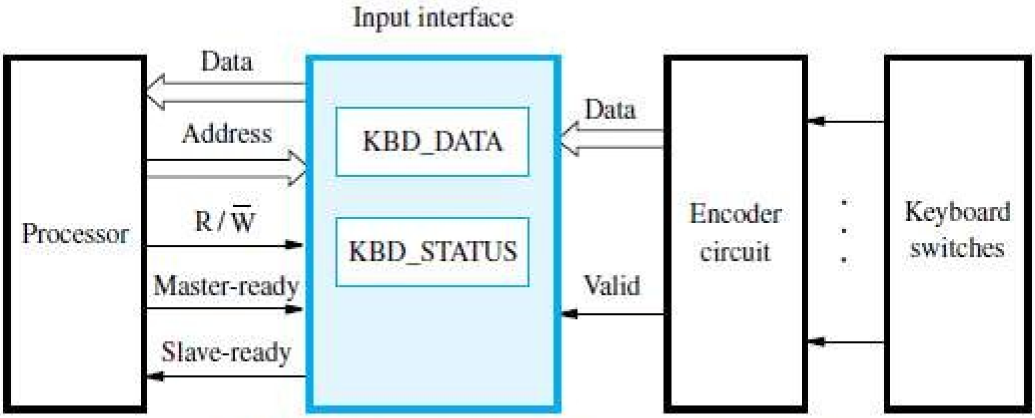


Figure 7.10 Keyboard to processor connection.

- The output of the encoder consists of
  → bits representing the encoded character and
  → one signal called **valid**, which indicates the key is pressed.
- The information is sent to the interface-circuits (Figure 7.10).
- Interface-circuits contain
  1) Data register DATAIN &
  2) Status-flag SIN.
- When a key is pressed, the Valid signal changes from 0 to1.
  Then, SIN=1 → when ASCII code is loaded into DATAIN.
  SIN = 0 → when processor reads the contents of the DATAIN.
- The interface-circuit is connected to the asynchronous bus.
- Data transfers on the bus are controlled using the handshake signals:
  1) Master ready &
  2) Slave ready.

INPUT-INTERFACE-CIRCUIT

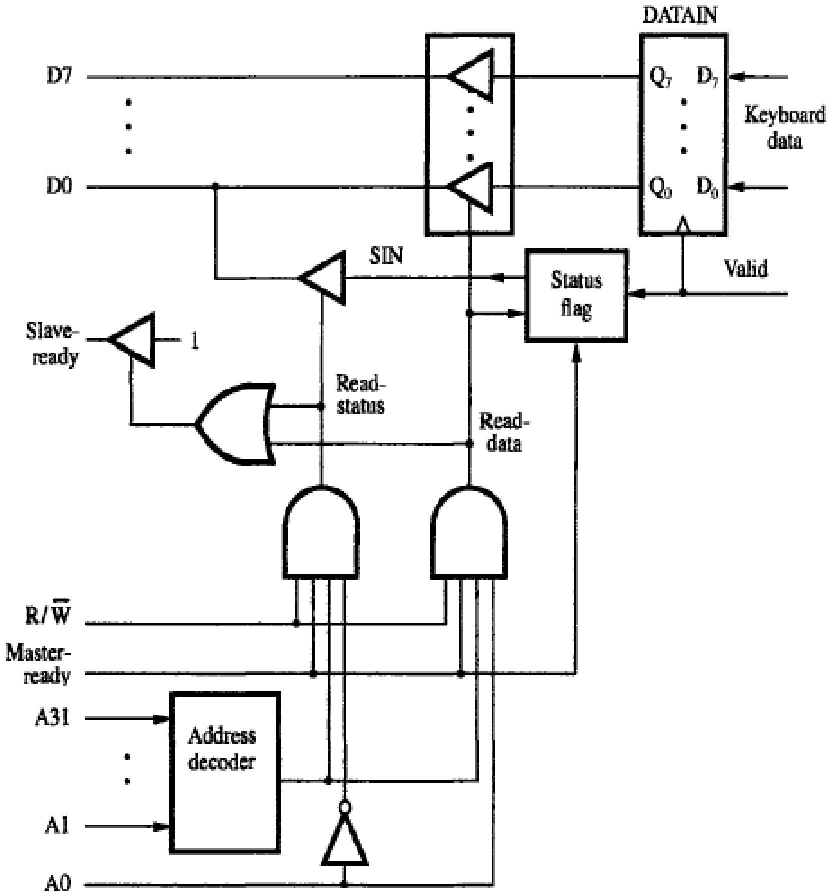


Figure 4.29: Input-interface-circuit

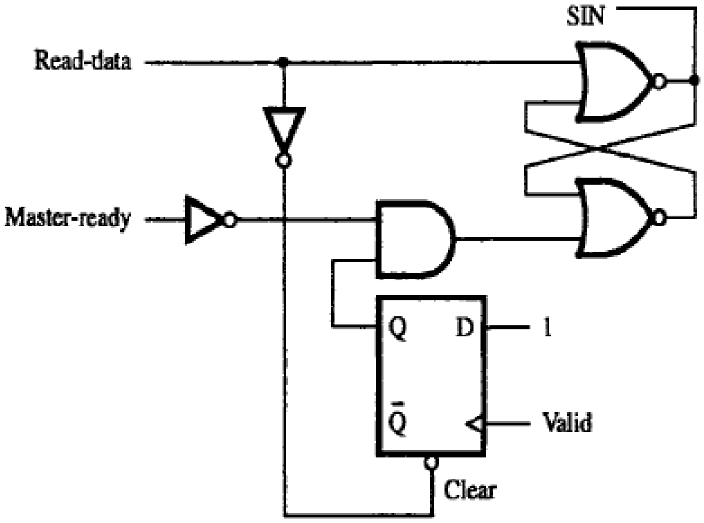


Figure 4.30 Circuit for the status flag block in Figure 4.29.

- Output-lines of DATAIN are connected to the data-lines of bus by means of 3-state drivers (Fig 4.29).
- Drivers are turned on when
  - → processor issues a read signal and
  - → address selects DATAIN.
- SIN signal is generated using a status-flag circuit (Figure 4.30).
  SIN signal is connected to line D_0 of the processor-bus using a 3-state driver.
- Address-decoder selects the input-interface based on bits A_1 through A_{31}.
- Bit A_0 determines whether the status or data register is to be read, when Master-ready is active.
- Processor activates the Slave-ready signal, when either the Read-status or Read-data is equal to 1.

PRINTER INTERFACED TO PROCESSOR

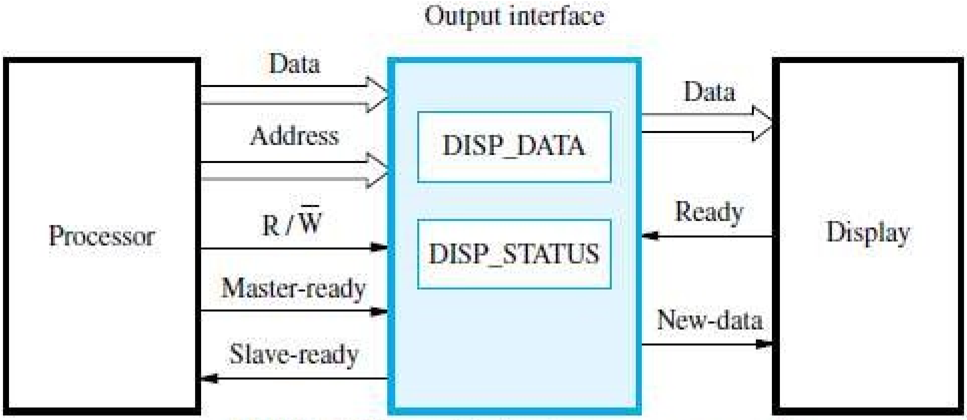


Figure 7.13 Display to processor connection.

- Keyboard is connected to a processor using a parallel-port.
- Processor uses
  - → memory-mapped I/O and
  - → asynchronous bus protocol.
- On the processor-side of the interface, we have:
  - → Data-lines
  - → Address-lines
  - → Control or R/W line
  - → Master-Ready signal and
  - → Slave-Ready signal.
- On the keyboard-side of the interface, we have:
  - → Encoder-circuit which generates a code for the key pressed.
  - → Debouncing-circuit which eliminates the effect of a key.
  - → Data-lines which contain the code for the key.
  - → Valid line changes from 0 to 1 when the key is pressed. This causes the code to be loaded into DATAIN and SIN to be set to 1.

GENERAL 8 BIT PARALLEL PROCESSING

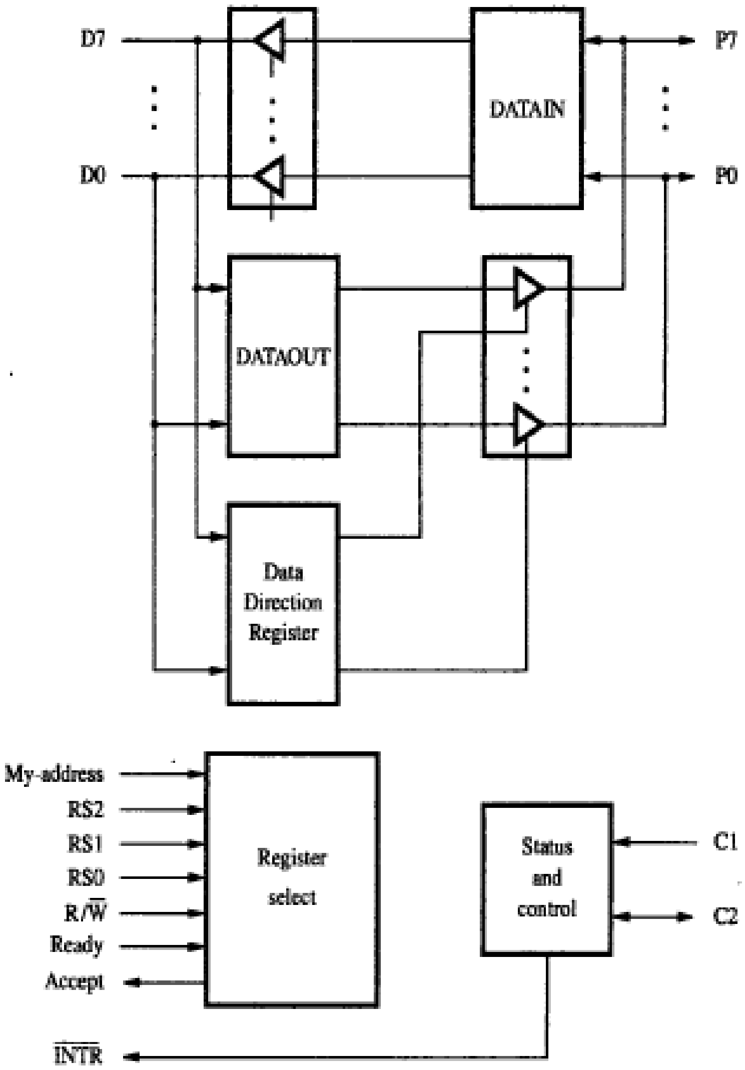


Figure 4.34: General 8 bit parallel interface

- Data-lines **P_7** through **P_0** can be used for either input or output purposes (Figure 4.34).
- For increased flexibility,
  - → some lines can be used as inputs and
  - → some lines can be used as outputs.
- The **DATAOUT** register is connected to data-lines via 3-state drivers that are controlled by a **DDR**.
- The processor can write any 8-bit pattern into DDR. (DDR → Data Direction Register).
- If DDR=1,

  Then, data-line acts as an output-line;

  Otherwise, data-line acts as an input-line.
- Two lines, **C_1** and **C_2** are used to control the interaction between interface-circuit and I/0 device.

  Two lines, C_1 and C_2 are also programmable.
- Line C_2 is bidirectional to provide different modes of signaling, including the handshake.
- The **Ready** and **Accept** lines are the handshake control lines on the processor-bus side.

  Hence, the Ready and Accept lines can be connected to Master-ready and Slave-ready.
- The input signal **My-address** should be connected to the output of an address-decoder.

  The address-decoder recognizes the address assigned to the interface.
- There are 3 register select lines: **RS_0-RS_2**.

  Three register select lines allows up to eight registers in the interface.
- An interrupt-request **INTR** is also provided.

  INTR should be connected to the interrupt-request line on the computer-bus.

STANDARD I/O INTERFACE

- Consider a computer system using different interface standards.
- Let us look in to Processor bus and Peripheral Component Interconnect (PCI) bus (Figure 4.38).
- These two buses are interconnected by a circuit called **Bridge**.
- The bridge translates the signals and protocols of one bus into another.
- The bridge-circuit introduces a small delay in data transfer between processor and the devices.

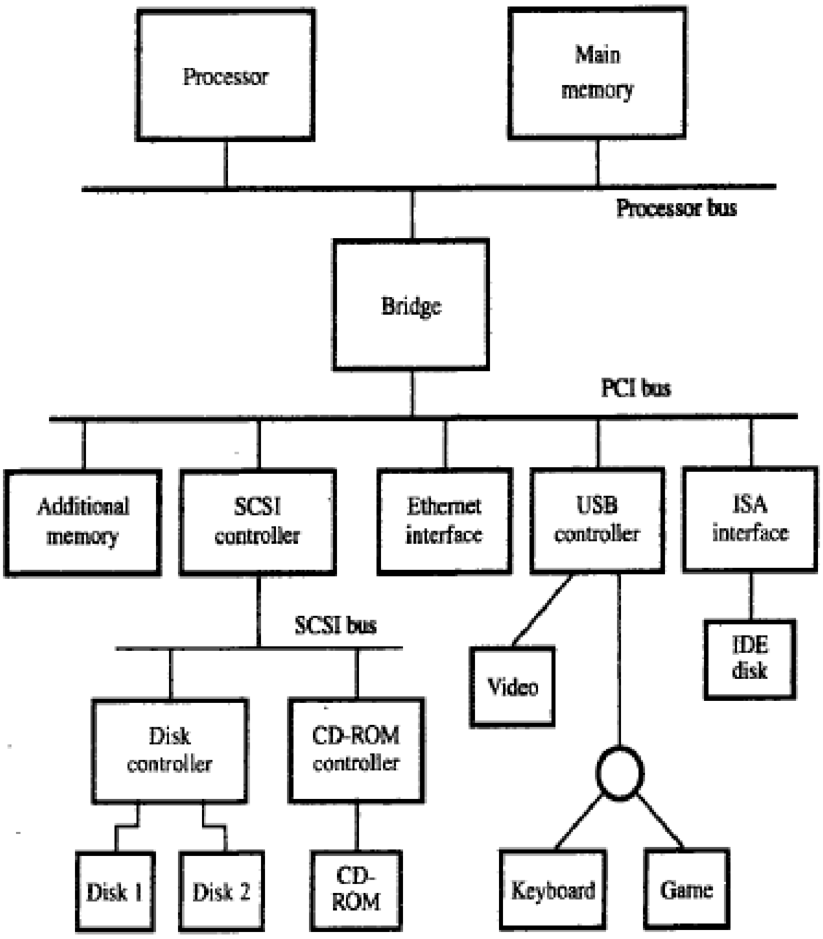


Figure 4.38 An example of a computer system using different interface standards.

- The 3 major standard I/O interfaces are:
  1) PCI (Peripheral Component Interconnect)
  2) SCSI (Small Computer System Interface)
  3) USB (Universal Serial Bus)
- PCI defines an expansion bus on the motherboard.
- SCSI and USB are used for connecting additional devices both inside and outside the computer-box.
- SCSI bus is a high speed parallel bus intended for devices such as disk and video display.
- USB uses a serial transmission to suit the needs of equipment ranging from keyboard to game control to internal connection.
- IDE (Integrated Device Electronics) disk is compatible with ISA which shows the connection to an Ethernet.

PCI

- PCI is developed as a low cost bus that is truly processor independent.
- PCI supports high speed disk, graphics and video devices.
- PCI has plug and play capability for connecting I/O devices.
- To connect new devices, the user simply connects the device interface board to the bus.

DATA TRANSFER IN PCI

- The data are transferred between cache and main-memory.
- The data is a sequence of words which are stored in successive memory-locations.
- During **read-operation**,
  - ➢ When the processor specifies an address, the memory responds by sending a sequence of data-words from successive memory-locations.
- During **write-operation**,
  - ➢ When the processor sends an address, a sequence of data-words is written into successive memory-locations.
- PCI supports read and write-operation.
- A read/write-operation involving a single word is treated as a burst of length one.
- PCI has 3 address-spaces. They are
  1) Memory address-space
  2) I/O address-space &
  3) Configuration address-space.
- I/O Address-space → Intended for use with processor.
  Configuration space → Intended to give PCI, its plug and play capability.
- **PCI Bridge** provides a separate physical connection to main-memory.
- The master maintains the address information on the bus until data-transfer is completed.
- At any time, only one device acts as **Bus-Master**.
- A master is called "initiator" which is either processor or DMA.
- The addressed-device that responds to read and write commands is called a **Target.**
- A complete transfer operation on the bus, involving an address and burst of data is called a **transaction.**

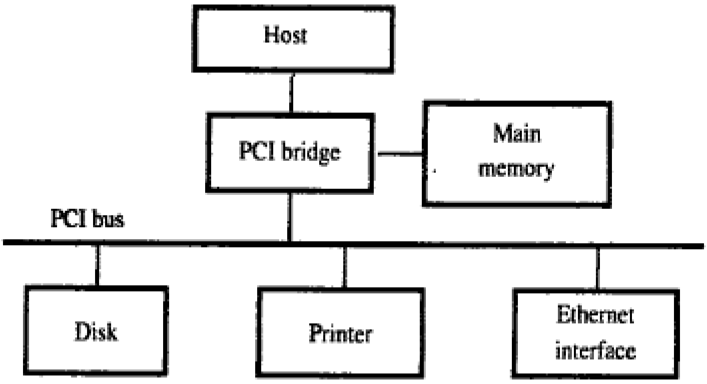


Figure 4.39 Use of a PCI bus in a computer system.

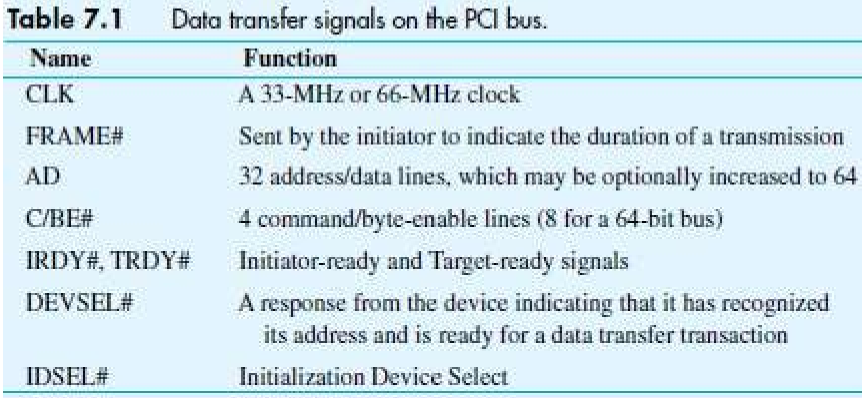


Table 7.1 Data transfer signals on the PCI bus.

Name	Function
CLK	A 33-MHz or 66-MHz clock
FRAME#	Sent by the initiator to indicate the duration of a transmission
AD	32 address/data lines, which may be optionally increased to 64
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus)
IRDY#, TRDY#	Initiator-ready and Target-ready signals
DEVSEL#	A response from the device indicating that it has recognized its address and is ready for a data transfer transaction
IDSEL#	Initialization Device Select

- Individual word transfers are called "**phases'**.

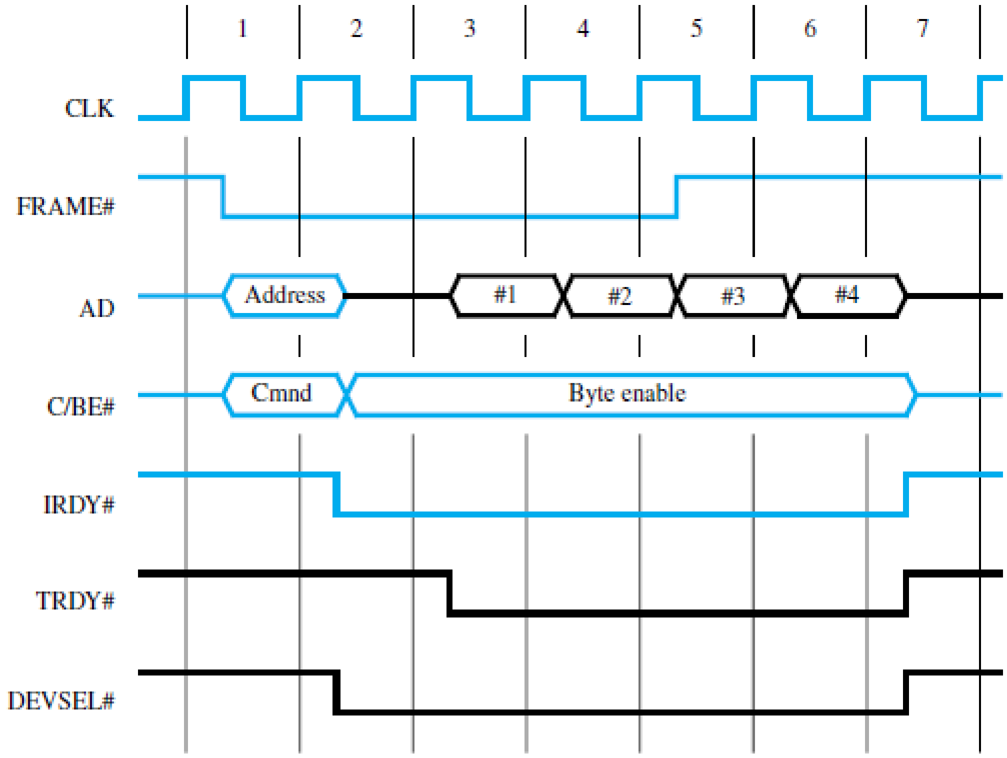


Figure 7.19 A Read operation on the PCI bus.

- During Clock cycle-1,
  - ➢ The processor a
    - → asserts FRAME# to indicate the beginning of a transaction;
    - → sends the address on AD lines and command on C/BE# Lines.
- During Clock cycle-2,
  - ➢ The processor removes the address and disconnects its drives from AD lines.
  - ➢ Selected target
    - → enables its drivers on AD lines and
    - → fetches the requested-data to be placed on bus.
  - ➢ Selected target
    - → asserts DEVSEL# and
    - → maintains it in asserted state until the end of the transaction.
  - ➢ C/BE# is
    - → used to send a bus command and it is
    - → used for different purpose during the rest of the transaction.
- During Clock cycle-3,
  - ➢ The initiator asserts IRDY# to indicate that it is ready to receive data.
  - ➢ If the target has data ready to send then it asserts TRDY#. In our eg, the target sends 3 more words of data in clock cycle 4 to 6.
- During Clock cycle-5
  - ➢ The indicator uses FRAME# to indicate the duration of the burst, since it read 4 words, the initiator negates FRAME# during clock cycle 5.
- During Clock cycle-7,
  - ➢ After sending 4th word, the target
    - → disconnects its drivers and
    - → negates DEVSEL# during clock cycle 7.

DEVICE CONFIGURATION OF PCI

- The PCI has a configuration ROM that stores information about that device.
- The configuration ROM's of all devices are accessible in the configuration address-space.
- The initialization software read these ROM's whenever the system is powered up or reset.
- In each case, it determines whether the device is a printer, keyboard or disk controller.
- Devices are assigned address during initialization process.
- Each device has an input signal called IDSEL# (Initialization device select) which has 21 address-lines (AD_{11} to AD_{31}).
- During configuration operation,
  - ➢ The address is applied to AD input of the device and
  - ➢ The corresponding AD line is set to 1 and all other lines are set to 0.

    AD_{11} - AD_{31} → **Upper** address-line

    A_0 - A_{10} → **Lower** address-line: Specify the type of the operation and to access the content of device configuration ROM.
- The configuration software scans all 21 locations. PCI bus has interrupt-request lines.
- Each device may requests an address in the I/O space or memory space

SCSI Bus

- SCSI stands for Small Computer System Interface.
- SCSI refers to the standard bus which is defined by ANSI (American National Standard Institute).
- SCSI bus the several options. It may be,

Narrow bus	It has 8 data-lines & transfers 1 byte at a time.
Wide bus	It has 16 data-lines & transfer 2 byte at a time.
Single-Ended Transmission	Each signal uses separate wire.
HVD (High Voltage Differential)	It was 5v (TTL cells)
LVD (Low Voltage Differential)	It uses 3.3v

- Because of these various options, SCSI connector may have 50, 68 or 80 pins. The data transfer rate ranges from 5MB/s to 160MB/s 320Mb/s, 640MB/s. The transfer rate depends on,

  1) Length of the cable
  2) Number of devices connected.
- To achieve high transfer rate, the bus length should be 1.6m for SE signaling and 12m for LVD signaling.
- The SCSI bus us connected to the processor-bus through the SCSI controller. The data are stored on a disk in blocks called sectors.

Each sector contains several hundreds of bytes. These data will not be stored in contiguous memory-location.

- SCSI protocol is designed to retrieve the data in the first sector or any other selected sectors.
- Using SCSI protocol, the burst of data are transferred at high speed.
- The controller connected to SCSI bus is of 2 types. They are1) Initiator * 2) Target

  1) Initiator

  ➢ It has the ability to select a particular target & to send commands specifying the operation to be performed.

  ➢ They are the controllers on the processor side.

  2) Target

  ➢ The disk controller operates as a target.

  ➢ It carries out the commands it receive from the initiator.

  ➢ The initiator establishes a logical connection with the intended target.

Steps for Read-operation

1) The SCSI controller contends for control of the bus (initiator).

2) When the initiator wins the arbitration-process, the initiator

→ selects the target controller and

→ hands over control of the bus to it.

3) The target starts an output operation. The initiator sends a command specifying the required read-operation.

4) The target

→ sends a message to initiator indicating that it will temporarily suspend connection b/w them.

→ then releases the bus.

5) The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read-operation.

6. The target

→ transfers the contents of the data buffer to the initiator and

→ then suspends the connection again.

7) The target controller sends a command to the disk drive to perform another seek operation.

8) As the initiator controller receives the data, it stores them into the main-memory using the DMA approach.

9) The SCSI controller sends an interrupt to the processor indicating that the data are now available.

BUS SIGNALS OF SCSI

• The bus has no address-lines. Instead, it has data-lines to identify the bus-controllers involved in the selection/reselection/arbitration-process.

• For narrow bus, there are 8 possible controllers numbered from 0 to 7. For a wide bus, there are 16 controllers.

• Once a connection is established b/w two controllers, there is no further need for addressing & the data-lines are used to carry the data.

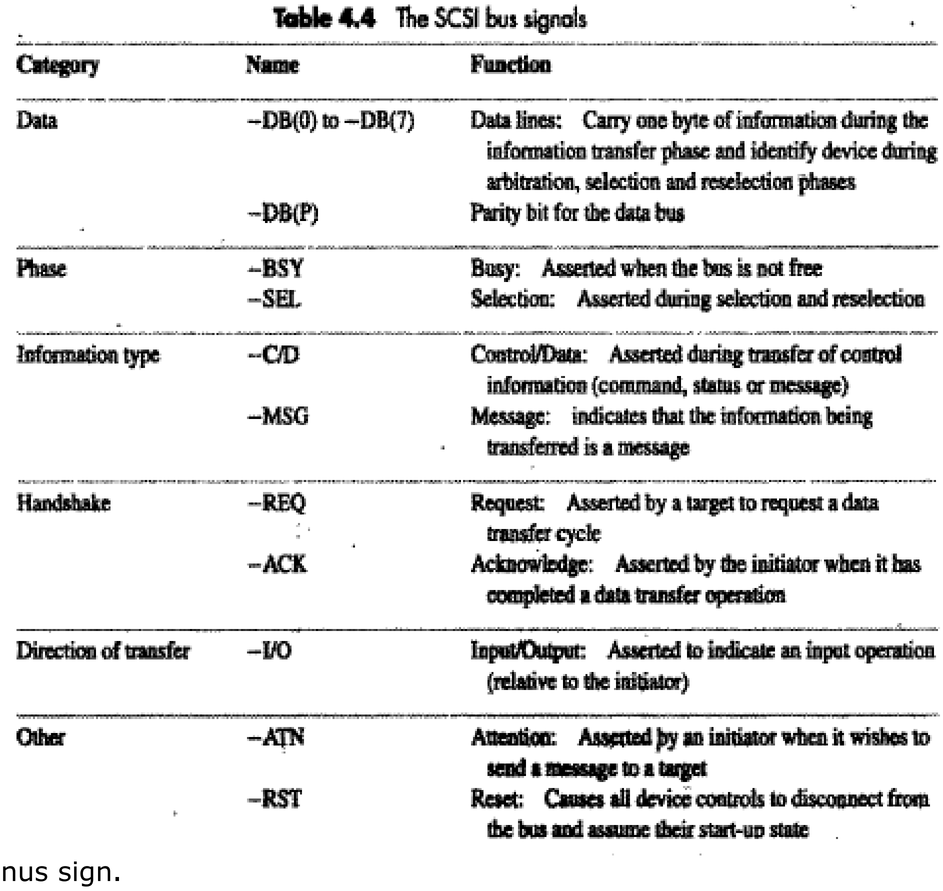


Table 4.4 The SCSI bus signals

Category	Name	Function
Data	–DB(0) to –DB(7)	Data lines: Carry one byte of information during the information transfer phase and identify device during arbitration, selection and reselection phases
	–DB(P)	Parity bit for the data bus
Phase	–BSY	Busy: Asserted when the bus is not free
	–SEL	Selection: Asserted during selection and reselection
Information type	–C/D	Control/Data: Asserted during transfer of control information (command, status or message)
	–MSG	Message: indicates that the information being transferred is a message
Handshake	–REQ	Request: Asserted by a target to request a data transfer cycle
	–ACK	Acknowledge: Asserted by the initiator when it has completed a data transfer operation
Direction of transfer	–I/O	Input/Output: Asserted to indicate an input operation (relative to the initiator)
Other	–ATN	Attention: Asserted by an initiator when it wishes to send a message to a target
	–RST	Reset: Causes all device controls to disconnect from the bus and assume their start-up state

• All signal names are proceeded by minus sign.

• This indicates that the signals are active or that the data-line is equal to 1, when they are in the low voltage state.

PHASES IN SCSI BUS

- The phases in SCSI bus operation are:

    1) Arbitration
    2) Selection
    3) Information transfer
    4) Reselection

1) Arbitration

- When the –BSY signal is in inactive state,
    → the bus will be free &
    → any controller can request the use of bus.
- SCSI uses distributed arbitration scheme because
    each controller may generate requests at the same time.
- Each controller on the bus is assigned a fixed priority.
- When –BSY becomes active, all controllers that are requesting the bus
    → examines the data-lines &
    → determine whether highest priority device is requesting bus at the same time.
- The controller using the highest numbered line realizes that it has won the arbitration-process.
- At that time, all other controllers disconnect from the bus & wait for –BSY to become inactive again.

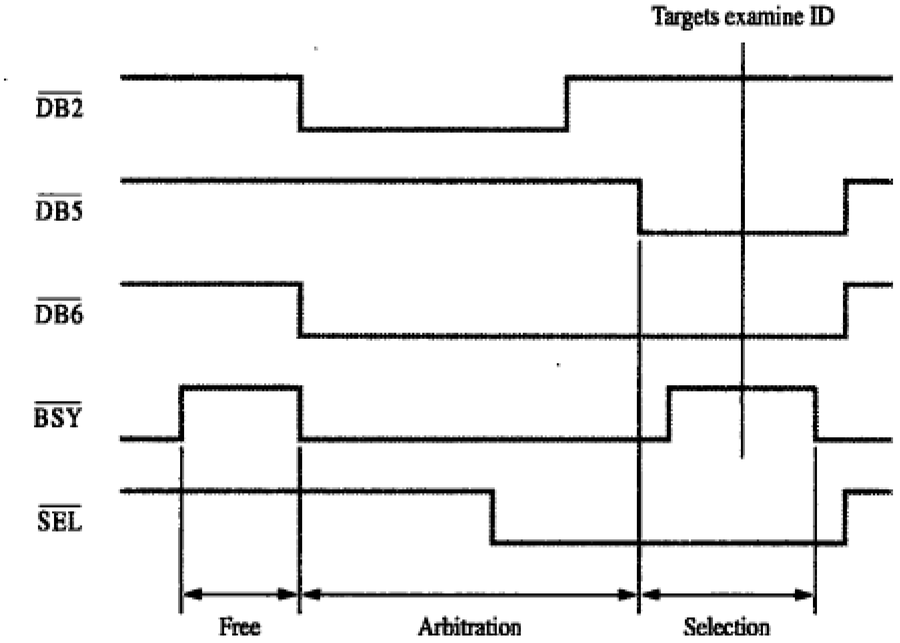


Figure 4.42 Arbitration and selection on the SCSI bus. Device 6 wins arbitration and selects device 2.

2) Information Transfer

- The information transferred between two controllers may consist of
    → commands from the initiator to the target
    → status responses from the target to the initiator or
    → data-transferred to/from the I/0 device.
- Handshake signaling is used to control information transfers, with the target controller taking the role of the bus-master.

3) Selection

- Here, Device
    → wins arbitration and
    → asserts –BSY and –DB6 signals.
- The Select Target Controller responds by asserting –BSY.
- This informs that the connection that it requested is established.

4) Reselection

- The connection between the two controllers has been reestablished, with the target in control of the bus as required for data transfer to proceed.

USB

- USB stands for Universal Serial Bus.
- USB supports 3 speed of operation. They are,
    1) Low speed (1.5 Mbps)
    2) Full speed (12 mbps) &
    3) High speed (480 mbps).
- The USB has been designed to meet the key objectives. They are,
    1) Provide a simple, low-cost and easy to use interconnection system.
       This overcomes difficulties due to the limited number of I/O ports available on a computer.
    2) Accommodate a wide range of data transfer characteristics for I/O devices.
       For e.g. telephone and Internet connections
    3) Enhance user convenience through a "plug-and-play" mode of operation.
- **Advantage:** USB helps to add many devices to a computer system at any time without opening the computer-box.

Port Limitation

➢ Normally, the system has a few limited ports.

➢ To add new ports, the user must open the computer-box to gain access to the internal expansion bus & install a new interface card.

➢ The user may also need to know to configure the device & the s/w.

Plug & Play

➢ The main objective: USB provides a plug & play capability.

➢ The plug & play feature enhances the connection of new device at any time, while the system is operation.

➢ The system should

→ Detect the existence of the new device automatically.
→ Identify the appropriate device driver s/w.
→ Establish the appropriate addresses.
→ Establish the logical connection for communication.

DEVICE CHARACTERISTICS OF USB

- The kinds of devices that may be connected to a computer cover a wide range of functionality.
- The speed, volume & timing constrains associated with data transfer to & from devices varies significantly.

Eg: 1 Keyboard

➢ Since the event of pressing a key is not synchronized to any other event in a computer system, the data generated by keyboard are called asynchronous.

The data generated from keyboard depends upon the speed of the human operator which is about 100 bytes/sec.

Eg: 2 Microphone attached in a computer system internally/externally

➢ The sound picked up by the microphone produces an analog electric signal, which must be converted into digital form before it can be handled by the computer.

➢ This is accomplished by sampling the analog signal periodically.

➢ The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a stream is called isochronous (i.e.) successive events are separated by equal period of time.

➢ If the sampling rate in „S" samples/sec then the maximum frequency captured by sampling process is s/2.

➢ A standard rate for digital sound is 44.1 KHz.

USB ARCHITECTURE

- To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure as shown in the figure 7.17.
- Each node of the tree has a device called a **Hub**.
- A hub acts as an intermediate control point between the host and the I/O devices.
- At the root of the tree, a **Root Hub** connects the entire tree to the host computer.
- The leaves of the tree are the I/O devices being served (for example, keyboard or speaker).
- A hub copies a message that it receives from its upstream connection to all its downstream ports.
- As a result, a message sent by the host computer is broadcast to all I/O devices, but only the addressed-device will respond to that message.

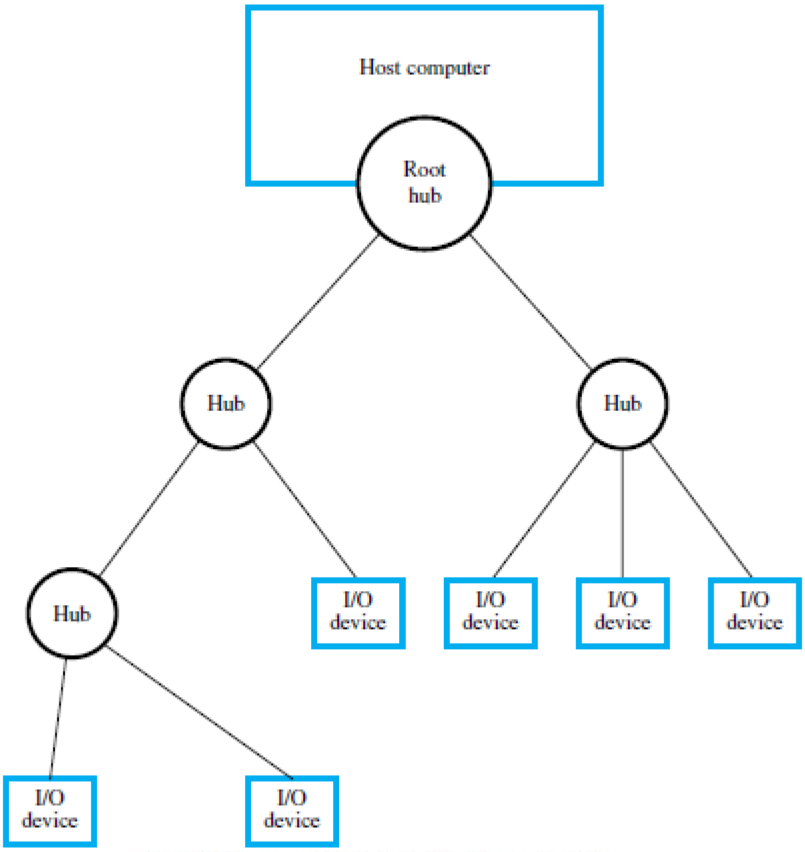


Figure 7.17 Universal Serial Bus tree structure.

USB ADDRESSING

- Each device may be a hub or an I/O device.
- Each device on the USB is assigned a 7-bit address.
- This address
  → is local to the USB tree and
  → is not related in any way to the addresses used on the processor-bus.
- A hub may have any number of devices or other hubs connected to it, and addresses are assigned arbitrarily.
- When a device is first connected to a hub, or when it is powered-on, it has the address 0.
- The hardware of the hub detects the device that has been connected, and it records this fact as part of its own status information.
- Periodically, the host polls each hub to
  → collect status information and
  → learn about new devices that may have been added or disconnected.
- When the host is informed that a new device has been connected, it uses sequence of commands to
  → send a reset signal on the corresponding hub port.
  → read information from the device about its capabilities.
  → send configuration information to the device, and
  → assign the device a unique USB address.
- Once this sequence is completed, the device
  → begins normal operation and
  → responds only to the new address.

USB PROTOCOLS

- All information transferred over the USB is organized in packets.
- A packet consists of one or more bytes of information.
- There are many types of packets that perform a variety of control functions.
- The information transferred on USB is divided into 2 broad categories: 1) Control and 2) Data.
- Control packets perform tasks such as
  → addressing a device to initiate data transfer.
  → acknowledging that data have been received correctly or
  → indicating an error.
- Data-packets carry information that is delivered to a device.
- A packet consists of one or more fields containing different kinds of information.
- The first field of any packet is called the **Packet Identifier (PID)** which identifies type of that packet.
- They are transmitted twice.
  1) The first time they are sent with their true values and
  2) The second time with each bit complemented.
- The four PID bits identify one of 16 different packet types.
- Some control packets, such as ACK (Acknowledge), consist only of the PID byte.
- Control packets used for controlling data transfer operations are called **Token Packets.**

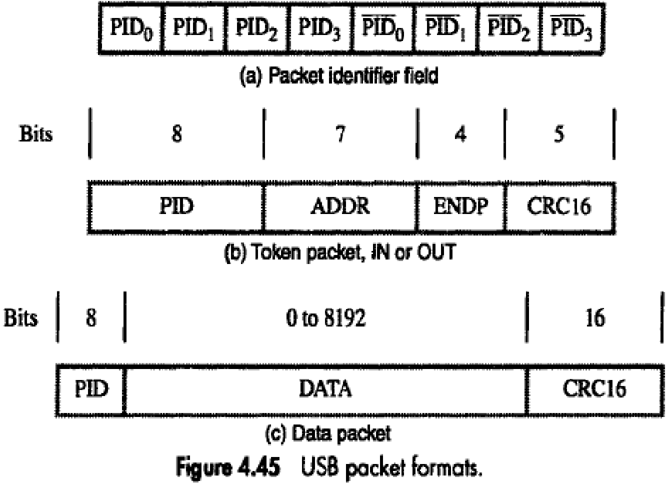

PID_0 PID_1 PID_2 PID_3 $\overline{PID_0}$ $\overline{PID_1}$ $\overline{PID_2}$ $\overline{PID_3}$
(a) Packet identifier field
Bits 8 7 4 5
PID ADDR ENDP CRC16
(b) Token packet, IN or OUT
Bits 8 0 to 8192 16
PID DATA CRC16
(c) Data packet

Figure 4.45 USB packet formats.

Problem 1:
The input status bit in an interface-circuit is cleared as soon as the input data register is read. Why is this important?

Solution:

After reading the input data, it is necessary to clear the input status flag before the program begins a new read-operation. Otherwise, the same input data would be read a second time.

Problem 2:
What is the difference between a subroutine and an interrupt-service routine?

Solution:

A subroutine is called by a program instruction to perform a function needed by the calling program.

An interrupt-service routine is initiated by an event such as an input operation or a hardware error. The function it performs may not be at all related to the program being executed at the time of interruption. Hence, it must not affect any of the data or status information relating to that program.

Problem 3:
Three devices A, B, & C are connected to the bus of a computer. I/O transfers for all 3 devices use interrupt control. Interrupt nesting for devices A & B is not allowed, but interrupt-requests from C may be accepted while either A or B is being serviced. Suggest different ways in which this can be accomplished in each of the following cases:

(a) The computer has one interrupt-request line.

(b) Two interrupt-request lines INTR1 & INTR2 are available, with INTR1 having higher priority.

Specify when and how interrupts are enabled and disabled in each case.

Solution:

(a) Interrupts should be enabled, except when C is being serviced. The nesting rules can be enforced by manipulating the interrupt-enable flags in the interfaces of A and B.

(b) A and B should be connected to INTR , and C to INTR. When an interrupt-request is received from either A or B, interrupts from the other device will be automatically disabled until the request has been serviced. However, interrupt-requests from C will always be accepted.

Problem 4:
Consider a computer in which several devices are connected to a common interrupt-request line. Explain how you would arrange for interrupts from device j to be accepted before the execution of the interrupt service routine for device i is completed. Comment in particular on the times at which interrupts must be enabled and disabled at various points in the system.

Solution:

Interrupts are disabled before the interrupt-service routine is entered. Once device i turns off its interrupt-request, interrupts may be safely enabled in the processor. If the interface-circuit of device i turns off its interrupt-request when it receives the interrupt acknowledge signal, interrupts may be enabled at the beginning of the interrupt-service routine of device i. Otherwise, interrupts may be enabled only after the instruction that causes device i to turn off its interrupt-request has been executed.

Problem 5:
Consider the daisy chain arrangement. Assume that after a device generates an interrupt-request, it turns off that request as soon as it receives the interrupt acknowledge signal. Is it still necessary to disable interrupts in the processor before entering the interrupt service routine? Why?

Solution:

Yes, because other devices may keep the interrupt-request line asserted.

MODULE 3: MEMORY SYSTEM

BASIC CONCEPTS

- Maximum size of memory that can be used in any computer is determined by addressing mode.

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

- If MAR is k-bits long then

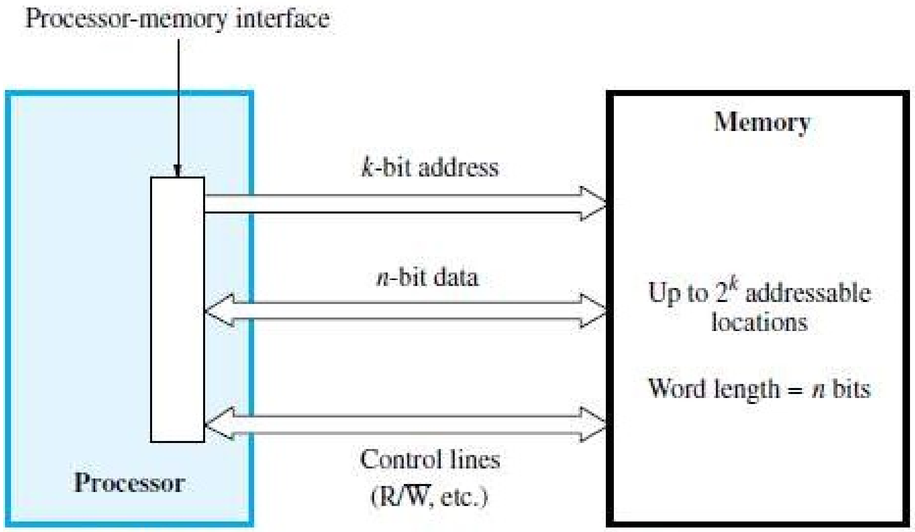


Figure 8.1 Connection of the memory to the processor.

→ memory may contain upto 2^K addressable-locations

- If MDR is n-bits long, then
  → n-bits of data are transferred between the memory and processor.
- The data-transfer takes place over the processor-bus (Figure 8.1).
- The processor-bus has
  1) Address-Line
  2) Data-line &
  3) Control-Line (R/W“, MFC – Memory Function Completed).
- The Control-Line is used for coordinating data-transfer.
- The processor reads the data from the memory by
  → loading the address of the required memory-location into MAR and
  → setting the R/W“ line to 1.
- The memory responds by
  → placing the data from the addressed-location onto the data-lines and
  → confirms this action by asserting MFC signal.
- Upon receipt of MFC signal, the processor loads the data from the data-lines into MDR.
- The processor writes the data into the memory-location by
  → loading the address of this location into MAR &
  → setting the R/W“ line to 0.
- **Memory Access Time:** It is the time that elapses between
  → initiation of an operation &
  → completion of that operation.
- **Memory Cycle Time:** It is the minimum time delay that required between the initiation of the two successive memory-operations.

RAM (Random Access Memory)

- In RAM, any location can be accessed for a Read/Write-operation in fixed amount of time,

Cache Memory

- ➢ It is a small, fast memory that is inserted between
  - → larger slower main-memory and
  - → processor.
- ➢ It holds the currently active segments of a program and their data.

Virtual Memory

- ➢ The address generated by the processor is referred to as a **virtual/logical address**.
- ➢ The virtual-address-space is mapped onto the physical-memory where data are actually stored.
- ➢ The mapping-function is implemented by MMU. (MMU = memory management unit).
- ➢ Only the active portion of the address-space is mapped into locations in the physical-memory.
- ➢ The remaining virtual-addresses are mapped onto the bulk storage devices such as magnetic disk.
- ➢ As the active portion of the virtual-address-space changes during program execution, the MMU
  - → changes the mapping-function &
  - → transfers the data between disk and memory.
- ➢ During every memory-cycle, MMU determines whether the addressed-page is in the memory.

  If the page is in the memory.

  Then, the proper word is accessed and execution proceeds.

  Otherwise, a page containing desired word is transferred from disk to memory.

- Memory can be classified as follows:

  1) RAM which can be further classified as follows:

     i) Static RAM

     ii) Dynamic RAM (DRAM) which can be further classified as synchronous & asynchronous DRAM.

  2) ROM which can be further classified as follows:

     i) PROM

     ii) EPROM

     iii) EEPROM &

     iv) Flash Memory which can be further classified as Flash Cards & Flash Drives.

SEMI CONDUCTOR RAM MEMORIES

INTERNAL ORGANIZATION OF MEMORY-CHIPS

- Memory-cells are organized in the form of array (Figure 8.2).
- Each cell is capable of storing 1-bit of information.
- Each row of cells forms a memory-word.
- All cells of a row are connected to a common line called as **Word-Line**.
- The cells in each column are connected to **Sense/Write** circuit by 2-bit-lines.
- The Sense/Write circuits are connected to data-input or output lines of the chip.
- During a write-operation, the sense/write circuit

  → receive input information &

  → store input info in the cells of the selected word.

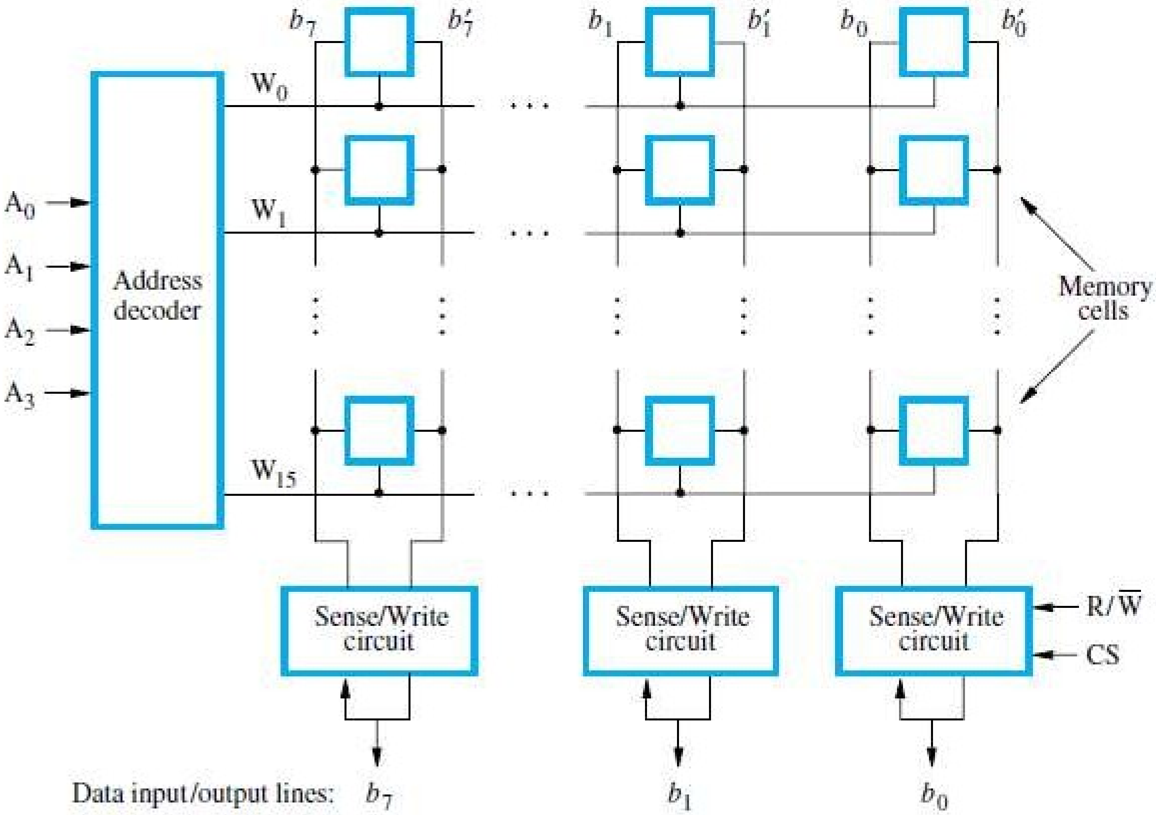


Figure 8.2 Organization of bit cells in a memory chip.

- The data-input and data-output of each Sense/Write circuit are connected to a single bidirectional data-line.
- Data-line can be connected to a data-bus of the computer.
- Following 2 control lines are also used:

  1) R/W' → Specifies the required operation.

  2) CS' → Chip Select input selects a given chip in the multi-chip memory-system.

Bit Organization	Requirement of external connection for address, data and control lines
128 (16x8)	14
(1024) 128x8(1k)	19

STATIC RAM (OR MEMORY)

- Memories consist of circuits capable of retaining their state as long as power is applied are known.

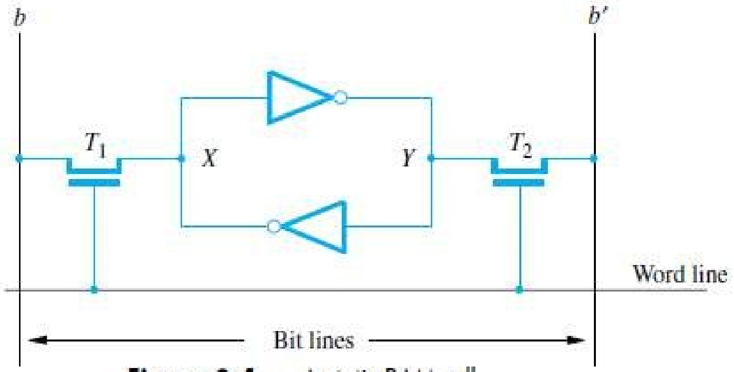


Figure 8.4 A static RAM cell.

- Two inverters are cross connected to form a latch (Figure 8.4).
- The latch is connected to 2-bit-lines by transistors T_1 and T_2.
- The transistors act as switches that can be opened/closed under the control of the word-line.
- When the word-line is at ground level, the transistors are turned off and the latch retain its state.

Read Operation

- To read the state of the cell, the word-line is activated to close switches T_1 and T_2.
- If the cell is in state 1, the signal on bit-line b is high and the signal on the bit-line b“ is low.
- Thus, b and b“ are complement of each other.
- Sense/Write circuit
  → monitors the state of b & b“ and
  → sets the output accordingly.

Write Operation

- The state of the cell is set by
  → placing the appropriate value on bit-line b and its complement on b“ and
  → then activating the word-line. This forces the cell into the corresponding state.
- The required signal on the bit-lines is generated by Sense/Write circuit.

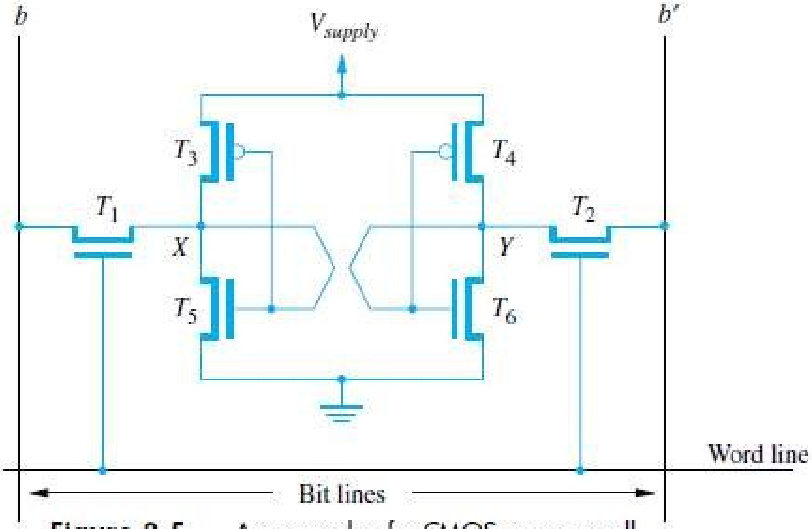


Figure 8.5 An example of a CMOS memory cell.

CMOS Cell

- Transistor pairs (T_3, T_5) and (T_4, T_6) form the inverters in the latch (Figure 8.5).
- In state 1, the voltage at point X is high by having T_5, T_6 ON and T_4, T_5 are OFF.
- Thus, T_1 and T_2 returned ON (Closed), bit-line b and b“ will have high and low signals respectively.
- **Advantages:**
  1) It has low power consumption „.“ the current flows in the cell only when the cell is active.
  2) Static RAM“s can be accessed quickly. It access time is few nanoseconds.
- **Disadvantage:** SRAMs are said to be volatile memories „.“ their contents are lost when power is interrupted.

ASYNCHRONOUS DRAM

- Less expensive RAMs can be implemented if simple cells are used.
- Such cells cannot retain their state indefinitely. Hence they are called **Dynamic RAM (DRAM).**
- The information stored in a dynamic memory-cell in the form of a charge on a capacitor.
- This charge can be maintained only for tens of milliseconds.
- The contents must be periodically refreshed by restoring this capacitor charge to its full value.

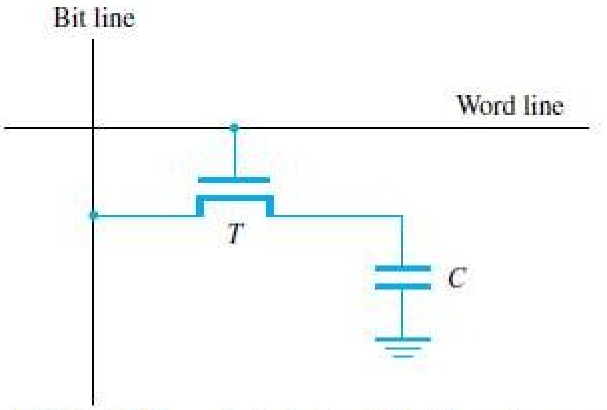


Figure 8.6 A single-transistor dynamic memory cell.

- In order to store information in the cell, the transistor T is turned „ON" (Figure 8.6).
- The appropriate voltage is applied to the bit-line which charges the capacitor.
- After the transistor is turned off, the capacitor begins to discharge.
- Hence, info. stored in cell can be retrieved correctly before threshold value of capacitor drops down.
- During a read-operation,

  → transistor is turned „ON"

  → a sense amplifier detects whether the charge on the capacitor is above the threshold value.

  ➢ If (charge on capacitor) > (threshold value) → Bit-line will have logic value „1".

  ➢ If (charge on capacitor) < (threshold value) → Bit-line will set to logic value „0".

ASYNCHRONOUS DRAM DESCRIPTION

- The 4 bit cells in each row are divided into 512 groups of 8 (Figure 5.7).
- 21 bit address is needed to access a byte in the memory. 21 bit is divided as follows:
  1) 12 address bits are needed to select a row.
     i.e. A_{8-0} → specifies row-address of a byte.
  2) 9 bits are needed to specify a group of 8 bits in the selected row.
     i.e. A_{20-9} → specifies column-address of a byte.

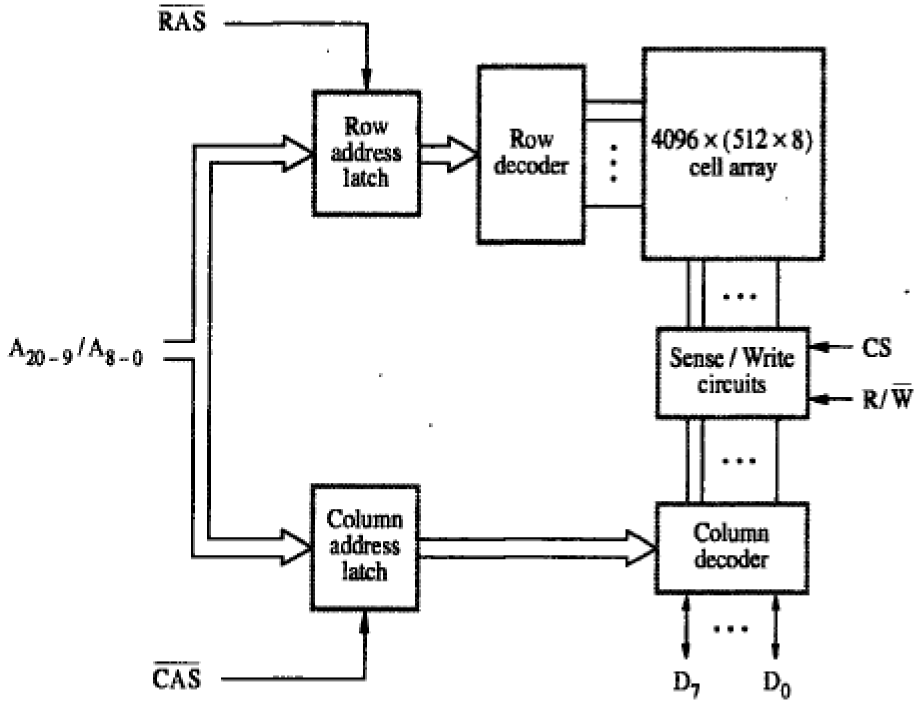


Figure 5.7 Internal organization of a 2M × 8 dynamic memory chip.

- During Read/Write-operation,
  → row-address is applied first.
  → row-address is loaded into row-latch in response to a signal pulse on **RAS'** input of chip.
  (RAS = Row-address Strobe CAS = Column-address Strobe)
- When a Read-operation is initiated, all cells on the selected row are read and refreshed.
- Shortly after the row-address is loaded, the column-address is
  → applied to the address pins &
  → loaded into **CAS'.**
- The information in the latch is decoded.
- The appropriate group of 8 Sense/Write circuits is selected.
  R/W'=1(read-operation) → Output values of selected circuits are transferred to data-lines D_0-D_7.
  R/W'=0(write-operation) → Information on D_0-D_7 are transferred to the selected circuits.
- RAS" & CAS" are active-low so that they cause latching of address when they change from high to low.
- To ensure that the contents of DRAMs are maintained, each row of cells is accessed periodically.
- A special memory-circuit provides the necessary control signals RAS" & CAS" that govern the timing.
- The processor must take into account the delay in the response of the memory.

Fast Page Mode

➢ Transferring the bytes in sequential order is achieved by applying the consecutive sequence of column-address under the control of successive CAS" signals.
➢ This scheme allows transferring a block of data at a faster rate.
➢ The block of transfer capability is called as *fast page mode***.**

SYNCHRONOUS DRAM

- The operations are directly synchronized with clock signal (Figure 8.8).
- The address and data connections are buffered by means of registers.
- The output of each sense amplifier is connected to a latch.
- A Read-operation causes the contents of all cells in the selected row to be loaded in these latches.
- Data held in latches that correspond to selected columns are transferred into data-output register.
- Thus, data becoming available on the data-output pins.

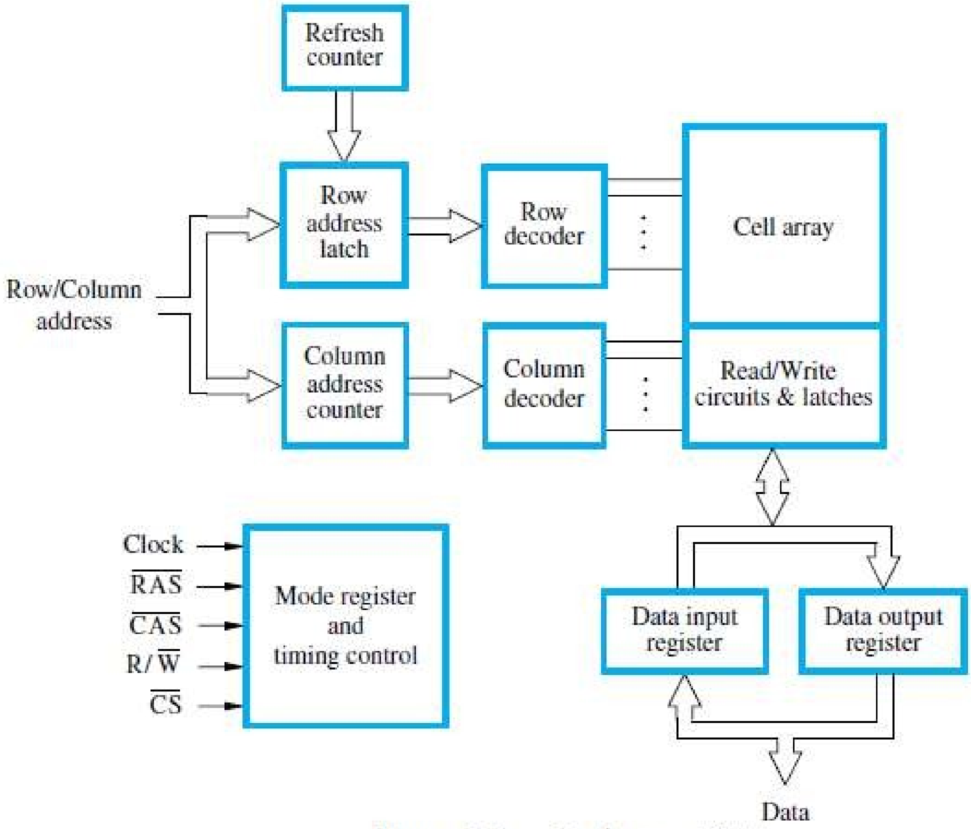


Figure 8.8 Synchronous DRAM.

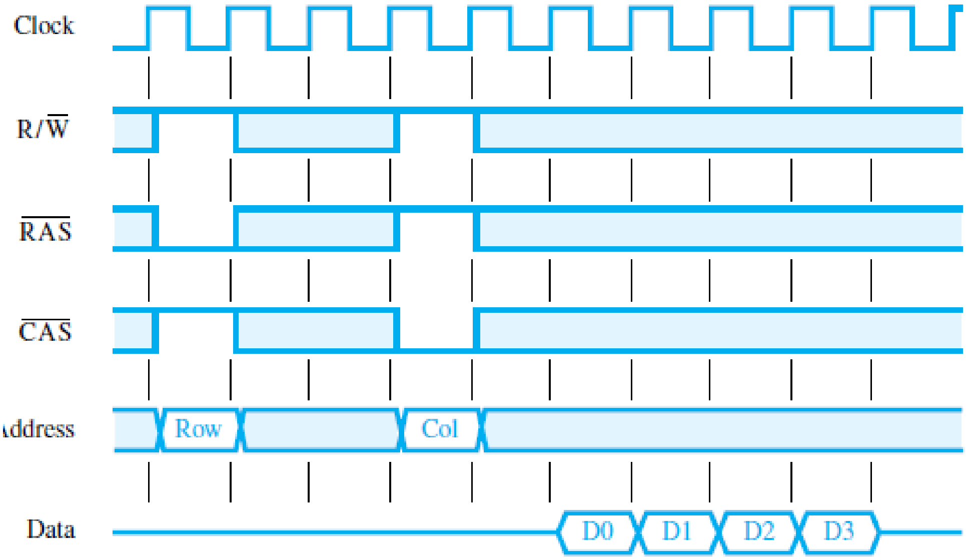


Figure 8.9 A burst read of length 4 in an SDRAM.

- First, the row-address is latched under control of RAS“ signal (Figure 8.9).
- The memory typically takes 2 or 3 clock cycles to activate the selected row.
- Then, the column-address is latched under the control of CAS“ signal.
- After a delay of one clock cycle, the first set of data bits is placed on the data-lines.
- SDRAM automatically increments column-address to access next 3 sets of bits in the selected row.

LATENCY & BANDWIDTH

- A good indication of performance is given by 2 parameters: 1) Latency 2) Bandwidth.

Latency

- It refers to the amount of time it takes to transfer a word of data to or from the memory.
- For a transfer of single word, the latency provides the complete indication of memory performance.
- For a block transfer, the latency denotes the time it takes to transfer the first word of data.

Bandwidth

- It is defined as the number of bits or bytes that can be transferred in one second.
- Bandwidth mainly depends on
  1) The speed of access to the stored data &
  2) The number of bits that can be accessed in parallel.

DOUBLE DATA RATE SDRAM (DDR-SDRAM)

- The standard SDRAM performs all actions on the rising edge of the clock signal.
- The DDR-SDRAM transfer data on both the edges (loading edge, trailing edge).
- The Bandwidth of DDR-SDRAM is doubled for long burst transfer.
- To make it possible to access the data at high rate, the cell array is organized into two banks.
- Each bank can be accessed separately.
- Consecutive words of a given block are stored in different banks.
- Such interleaving of words allows simultaneous access to two words.
- The two words are transferred on successive edge of the clock.

STRUCTURE OF LARGER MEMORIES

Dynamic Memory System

- The physical implementation is done in the form of memory-modules.
- If a large memory is built by placing DRAM chips directly on the Motherboard, then it will occupy large amount of space on the board.
- These packaging consideration have led to the development of larger memory units known as SIMM"s & DIMM"s.
  1) SIMM → Single Inline memory-module
  2) DIMM → Dual Inline memory-module
- SIMM/DIMM consists of many memory-chips on small board that plugs into a socket on motherboard.

MEMORY-SYSTEM CONSIDERATION

MEMORY CONTROLLER

- To reduce the number of pins, the dynamic memory-chips use multiplexed-address inputs.
- The address is divided into 2 parts:

1) High Order Address Bit

➢ Select a row in cell array.

➢ It is provided first and latched into memory-chips under the control of RAS" signal.

2) Low Order Address Bit

➢ Selects a column.

➢ They are provided on same address pins and latched using CAS" signals.

- The Multiplexing of address bit is usually done by **Memory Controller Circuit** (Figure 5.11).

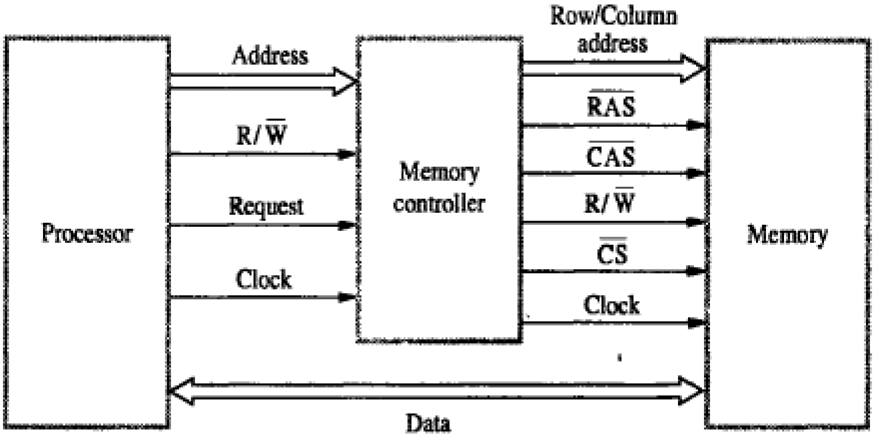


Figure 5.11 Use of a memory controller.

- The Controller accepts a complete address & R/W" signal from the processor.
- A Request signal indicates a memory access operation is needed.
- Then, the Controller

→ forwards the row & column portions of the address to the memory.

→ generates RAS" & CAS" signals &

→ sends R/W" & CS" signals to the memory.

RAMBUS MEMORY

- The usage of wide bus is expensive.
- Rambus developed the implementation of narrow bus.
- Rambus technology is a fast signaling method used to transfer information between chips.
- The signals consist of much smaller voltage swings around a reference voltage V_{ref}.
- The reference voltage is about 2V.
- The two logical values are represented by 0.3V swings above and below V_{ref}.
- This type of signaling is generally is known as **Differential Signalling**.
- Rambus provides a complete specification for design of communication called as **Rambus Channel**.
- Rambus memory has a clock frequency of 400 MHz.
- The data are transmitted on both the edges of clock so that effective data-transfer rate is 800MHZ.
- Circuitry needed to interface to Rambus channel is included on chip. Such chips are called **RDRAM.**

(RDRAM = Rambus DRAMs).

- Rambus channel has:

1) 9 Data-lines (1st-8th line ->Transfer the data, 9th line->Parity checking).

2) Control-Line &

3) Power line.

- A two channel rambus has 18 data-lines which has no separate Address-Lines.
- Communication between processor and RDRAM modules is carried out by means of packets transmitted on the data-lines.
- There are 3 types of packets:

1) Request

2) Acknowledge &

3) Data.

READ ONLY MEMORY (ROM)

- Both SRAM and DRAM chips are volatile, i.e. They lose the stored information if power is turned off.
- Many application requires non-volatile memory which retains the stored information if power is turned off.
- For ex:

  OS software has to be loaded from disk to memory i.e. it requires non-volatile memory.
- Non-volatile memory is used in embedded system.
- Since the normal operation involves only reading of stored data, a memory of this type is called ROM.
  - **At Logic value '0'** → Transistor(T) is connected to the ground point (P).

    Transistor switch is closed & voltage on bit-line nearly drops to zero (Figure 8.11).
  - **At Logic value '1'** → Transistor switch is open.

    The bit-line remains at high voltage.

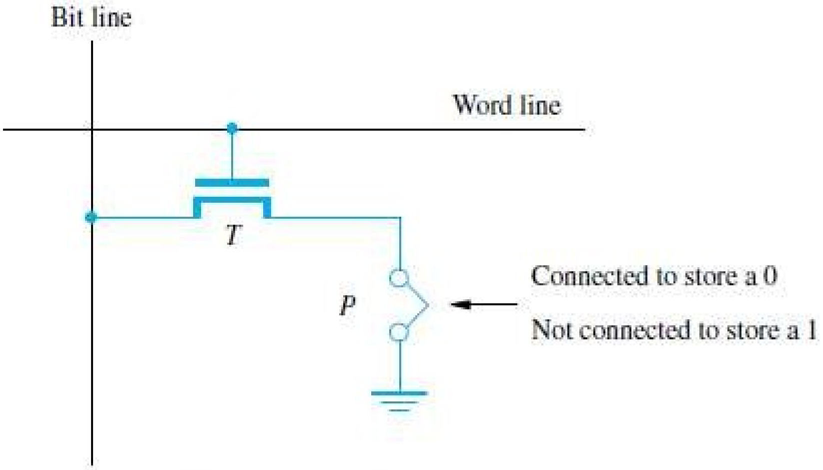


Figure 8.11 A ROM cell.

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

TYPES OF ROM

- Different types of non-volatile memory are
  1) PROM
  2) EPROM
  3) EEPROM &
  4) Flash Memory (Flash Cards & Flash Drives)

PROM (PROGRAMMABLE ROM)

- PROM allows the data to be loaded by the user.
- Programmability is achieved by inserting a „fuse" at point P in a ROM cell.
- Before PROM is programmed, the memory contains all 0"s.
- User can insert 1"s at required location by burning-out fuse using high current-pulse.
- This process is irreversible.
- **Advantages:**
  1) It provides flexibility.
  2) It is faster.
  3) It is less expensive because they can be programmed directly by the user.

EPROM (ERASABLE REPROGRAMMABLE ROM)

- EPROM allows
  → stored data to be erased and
  → new data to be loaded.
- In cell, a connection to ground is always made at „P" and a special transistor is used.
- The transistor has the ability to function as
  → a normal transistor or
  → a disabled transistor that is always turned „off".
- Transistor can be programmed to behave as a permanently open switch, by injecting charge into it.
- Erasure requires dissipating the charges trapped in the transistor of memory-cells.
  This can be done by exposing the chip to ultra-violet light.
- **Advantages:**
  1) It provides flexibility during the development-phase of digital-system.
  2) It is capable of retaining the stored information for a long time.
- **Disadvantages:**
  1) The chip must be physically removed from the circuit for reprogramming.
  2) The entire contents need to be erased by UV light.

EEPROM (ELECTRICALLY ERASABLE ROM)

- **Advantages:**
  1) It can be both programmed and erased electrically.
  2) It allows the erasing of all cell contents selectively.
- **Disadvantage:** It requires different voltage for erasing, writing and reading the stored data.

FLASH MEMORY

- In EEPROM, it is possible to read & write the contents of a single cell.
- In Flash device, it is possible to read contents of a single cell & write entire contents of a block.
- Prior to writing, the previous contents of the block are erased.
  Eg. In MP3 player, the flash memory stores the data that represents sound.
- Single flash chips cannot provide sufficient storage capacity for embedded-system.
- **Advantages:**
  1) Flash drives have greater density which leads to higher capacity & low cost per bit.
  2) It requires single power supply voltage & consumes less power.
- There are 2 methods for implementing larger memory: 1) Flash Cards & 2) Flash Drives

  1) Flash Cards
  ➢ One way of constructing larger module is to mount flash-chips on a small card.
  ➢ Such flash-card have standard interface.
  ➢ The card is simply plugged into a conveniently accessible slot.
  ➢ Memory-size of the card can be 8, 32 or 64MB.
  ➢ Eg: A minute of music can be stored in 1MB of memory. Hence 64MB flash cards can store an hour of music.

  2) Flash Drives
  ➢ Larger flash memory can be developed by replacing the hard disk-drive.
  ➢ The flash drives are designed to fully emulate the hard disk.
  ➢ The flash drives are solid state electronic devices that have no movable parts.

  Advantages:
  1) They have shorter seek & access time which results in faster response.
  2) They have low power consumption. .". they are attractive for battery driven application.
  3) They are insensitive to vibration.

  Disadvantages:
  1) The capacity of flash drive (<1GB) is less than hard disk (>1GB).
  2) It leads to higher cost per bit.
  3) Flash memory will weaken after it has been written a number of times (typically at least 1 million times).

SPEED, SIZE COST

Characteristics	SRAM	DRAM	Magnetis Disk
Speed	Very Fast	Slower	Much slower than DRAM
Size	Large	Small	Small
Cost	Expensive	Less Expensive	Low price

Memory	Speed	Size	Cost
Registers	Very high	Lower	Very Lower
Primary cache	High	Lower	Low
Secondary cache	Low	Low	Low
Main memory	Lower than Seconadry cache	High	High
Secondary Memory	Very low	Very High	Very High

- The main-memory can be built with DRAM (Figure 8.14)
- Thus, SRAM"s are used in smaller units where speed is of essence.
- The Cache-memory is of 2 types:
  - **1) Primary/Processor Cache** (Level1 or L1 cache)
    - ➢ It is always located on the processor-chip.
  - **2) Secondary Cache** (Level2 or L2 cache)
    - ➢ It is placed between the primary-cache and the rest of the memory.
- The memory is implemented using the dynamic components (SIMM, RIMM, DIMM).
- The access time for main-memory is about 10 times longer than the access time for L1 cache.

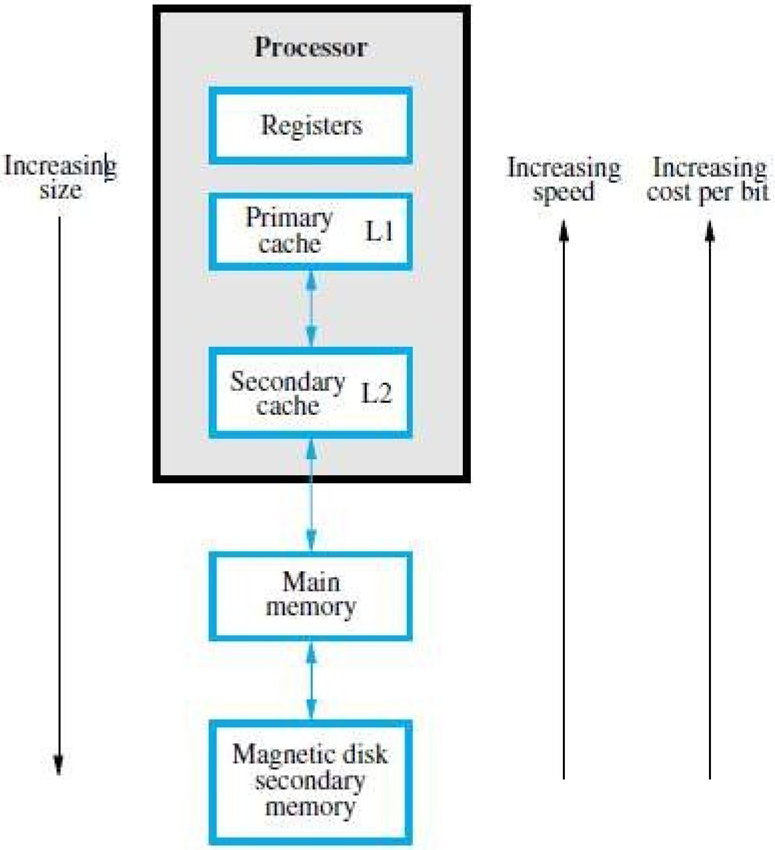


Figure 8.14 Memory hierarchy.

CACHE MEMORIES

- The effectiveness of cache mechanism is based on the property of „**Locality of Reference'.**

Locality of Reference

- Many instructions in the localized areas of program are executed repeatedly during some time period
- Remainder of the program is accessed relatively infrequently (Figure 8.15).
- There are 2 types:

  1) Temporal

  ➢ The recently executed instructions are likely to be executed again very soon.

  2) Spatial

  ➢ Instructions in close proximity to recently executed instruction are also likely to be executed soon.
- If active segment of program is placed in cache-memory, then total execution time can be reduced.
- **Block** refers to the set of contiguous address locations of some size.
- The cache-line is used to refer to the cache-block.

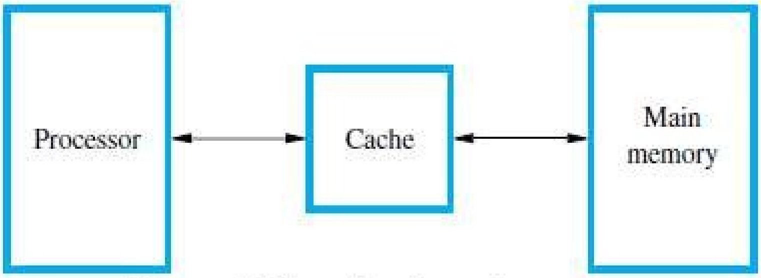


Figure 8.15 Use of a cache memory.

- The Cache-memory stores a reasonable number of blocks at a given time.
- This number of blocks is small compared to the total number of blocks available in main-memory.
- Correspondence b/w main-memory-block & cache-memory-block is specified by mapping-function.
- Cache control hardware decides which block should be removed to create space for the new block.
- The collection of rule for making this decision is called the **Replacement Algorithm.**
- The cache control-circuit determines whether the requested-word currently exists in the cache.
- The write-operation is done in 2 ways: 1) Write-through protocol & 2) Write-back protocol.

  Write-Through Protocol

  ➢ Here the cache-location and the main-memory-locations are updated simultaneously.

  Write-Back Protocol

  ➢ This technique is to

  → update only the cache-location &

  → mark the cache-location with associated flag bit called **Dirty/Modified Bit.**

  ➢ The word in memory will be updated later, when the marked-block is removed from cache.

During Read-operation

- If the requested-word currently not exists in the cache, then **read-miss** will occur.
- To overcome the read miss, *Load–through/Early restart protocol* is used.

  Load–Through Protocol

  ➢ The block of words that contains the requested-word is copied from the memory into cache.

  ➢ After entire block is loaded into cache, the requested-word is forwarded to processor.

During Write-operation

- If the requested-word not exists in the cache, then **write-miss** will occur.

  1) If **Write Through Protocol** is used, the information is written directly into main-memory.

  2) If **Write Back Protocol** is used,

  → then block containing the addressed word is first brought into the cache &

  → then the desired word in the cache is over-written with the new information.

MAPPING-FUNCTION

- Here we discuss about 3 different mapping-function:

  1) Direct Mapping

  2) Associative Mapping

  3) Set-Associative Mapping

DIRECT MAPPING

- The block-j of the main-memory maps onto block-j modulo-128 of the cache (Figure 8.16).
- When the memory-blocks 0, 128, & 256 are loaded into cache, the block is stored in cache-block 0.
  Similarly, memory-blocks 1, 129, 257 are stored in cache-block 1.
- The contention may arise when
  1) When the cache is full.
  2) When more than one memory-block is mapped onto a given cache-block position.
- The contention is resolved by
  allowing the new blocks to overwrite the currently resident-block.
- Memory-address determines placement of block in the cache.

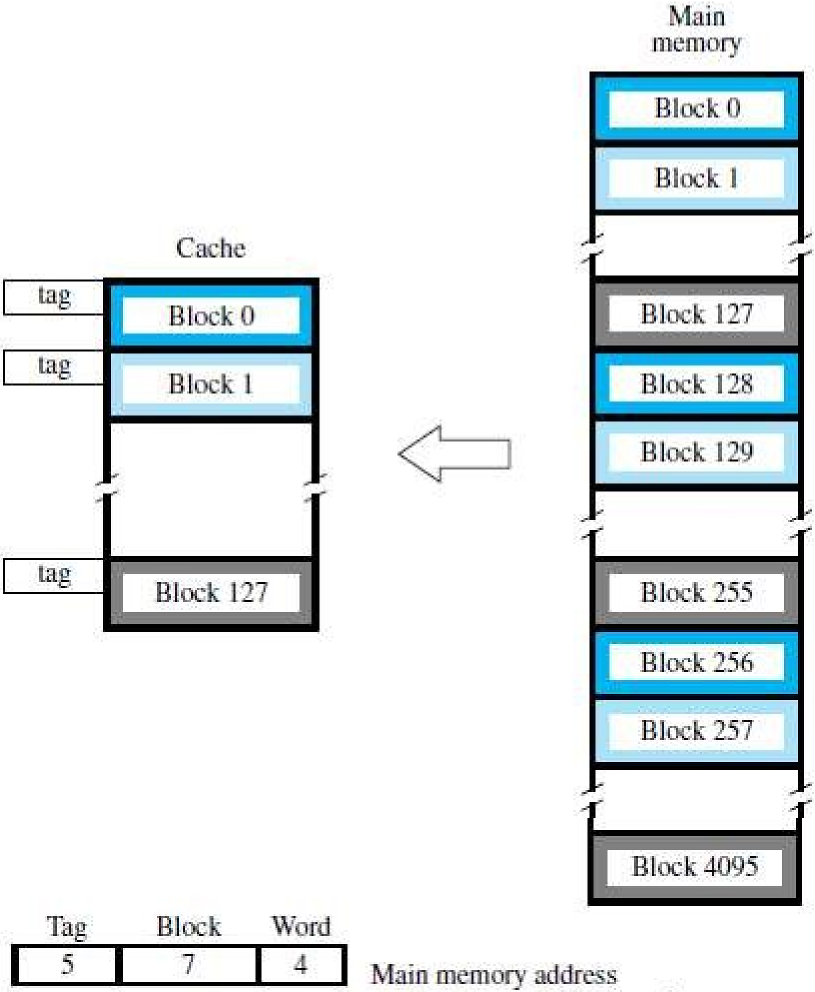


Figure 8.16 Direct-mapped cache.

- The memory-address is divided into 3 fields:

  1) Low Order 4 bit field
  ➢ Selects one of 16 words in a block.

  2) 7 bit cache-block field
  ➢ 7-bits determine the cache-position in which new block must be stored.

  3) 5 bit Tag field
  ➢ 5-bits memory-address of block is stored in 5 tag-bits associated with cache-location.
- As execution proceeds,
  5-bit tag field of memory-address is compared with tag-bits associated with cache-location.
  If they match, then the desired word is in that block of the cache.
  Otherwise, the block containing required word must be first read from the memory.
  And then the word must be loaded into the cache.

ASSOCIATIVE MAPPING

- The memory-block can be placed into any cache-block position. (Figure 8.17).
- 12 tag-bits will identify a memory-block when it is resolved in the cache.
- Tag-bits of an address received from processor are compared to the tag-bits of each block of cache.
- This comparison is done to see if the desired block is present.

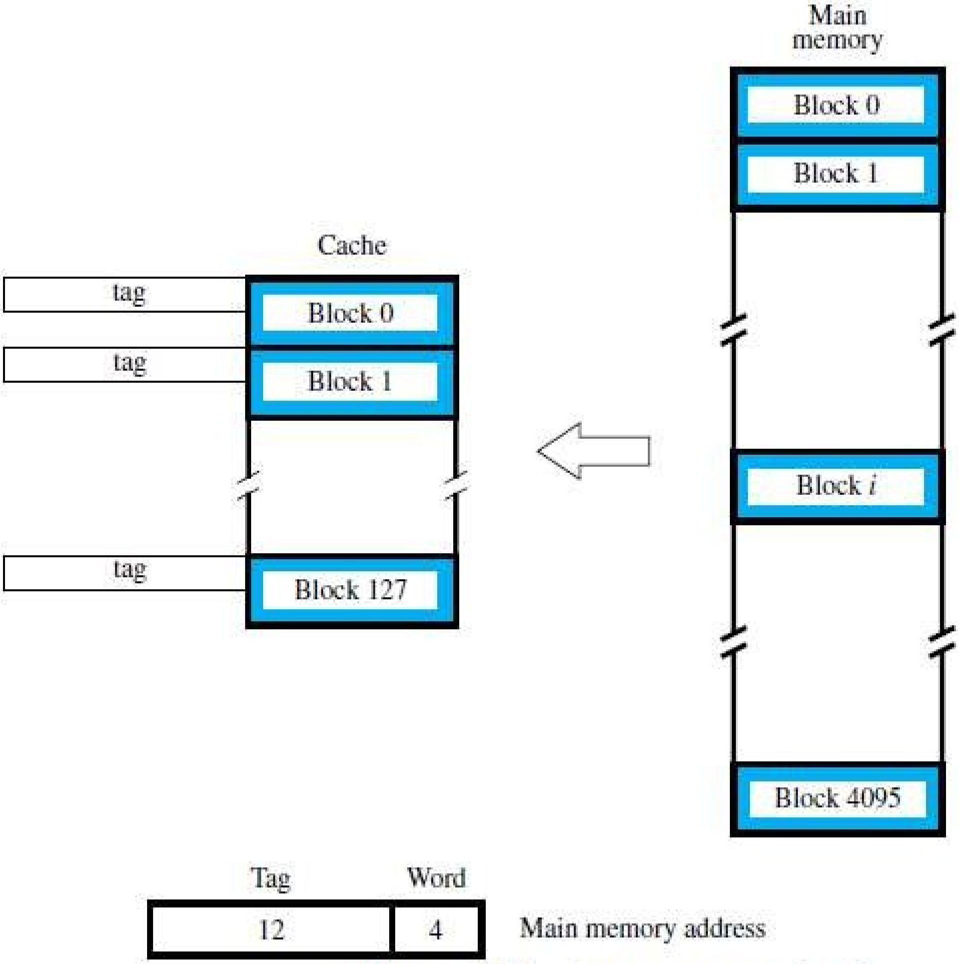


Figure 8.17 Associative-mapped cache.

- It gives complete freedom in choosing the cache-location.
- A new block that has to be brought into the cache has to replace an existing block if the cache is full.
- The memory has to determine whether a given block is in the cache.
- **Advantage:** It is more flexible than direct mapping technique.
- **Disadvantage:** Its cost is high.

SET-ASSOCIATIVE MAPPING

- It is the combination of direct and associative mapping. (Figure 8.18).
- The blocks of the cache are grouped into sets.
- The mapping allows a block of the main-memory to reside in any block of the specified set.
- The cache has 2 blocks per set, so the memory-blocks 0, 64, 128…….. 4032 maps into cache set „0".
- The cache can occupy either of the two block position within the set.

6 bit set field

➢ Determines which set of cache contains the desired block.

6 bit tag field

➢ The tag field of the address is compared to the tags of the two blocks of the set.

➢ This comparison is done to check if the desired block is present.

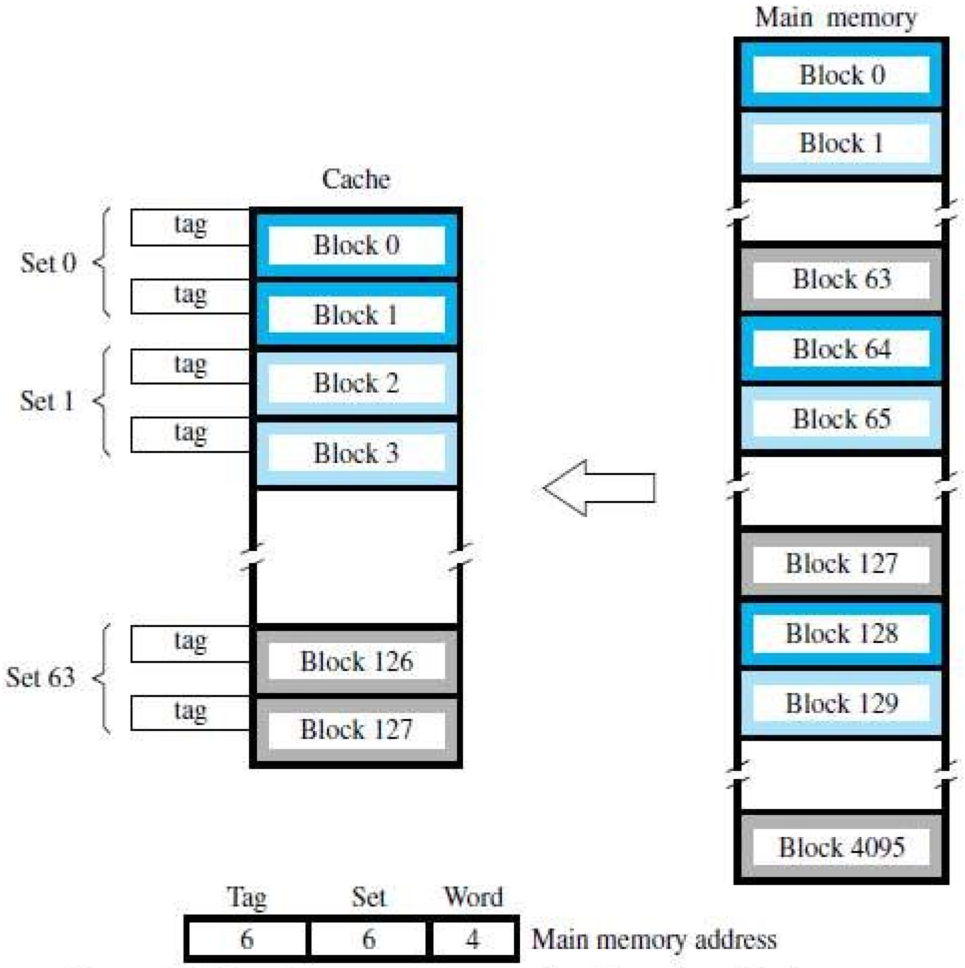


Figure 8.18 Set-associative-mapped cache with two blocks per set.

- The cache which contains 1 block per set is called **direct mapping.**
- A cache that has „k" blocks per set is called as"**k-way set associative cache**".
- Each block contains a control-bit called a **valid-bit**.
- The Valid-bit indicates that whether the block contains valid-data.
- The dirty bit indicates that whether the block has been modified during its cache residency.

Valid-bit=0 → When power is initially applied to system.

Valid-bit=1 → When the block is loaded from main-memory at first time.

- If the main-memory-block is updated by a source & if the block in the source is already exists in the cache, then the valid-bit will be cleared to "0".
- If Processor & DMA uses the same copies of data then it is called as **Cache Coherence Problem**.
- **Advantages:**

1) Contention problem of direct mapping is solved by having few choices for block placement.

2) The hardware cost is decreased by reducing the size of associative search.

REPLACEMENT ALGORITHM

- In direct mapping method,

  the position of each block is pre-determined and there is no need of replacement strategy.
- In associative & set associative method,

  The block position is not pre-determined.

  If the cache is full and if new blocks are brought into the cache,

  then the cache-controller must decide which of the old blocks has to be replaced.
- When a block is to be overwritten, the block with longest time w/o being referenced is over-written.
- This block is called **Least recently Used (LRU) block** & the technique is called **LRU algorithm**.
- The cache-controller tracks the references to all blocks with the help of block-counter.
- **Advantage:** Performance of LRU is improved by randomness in deciding which block is to be over-written.

Eg:

Consider 4 blocks/set in set associative cache.

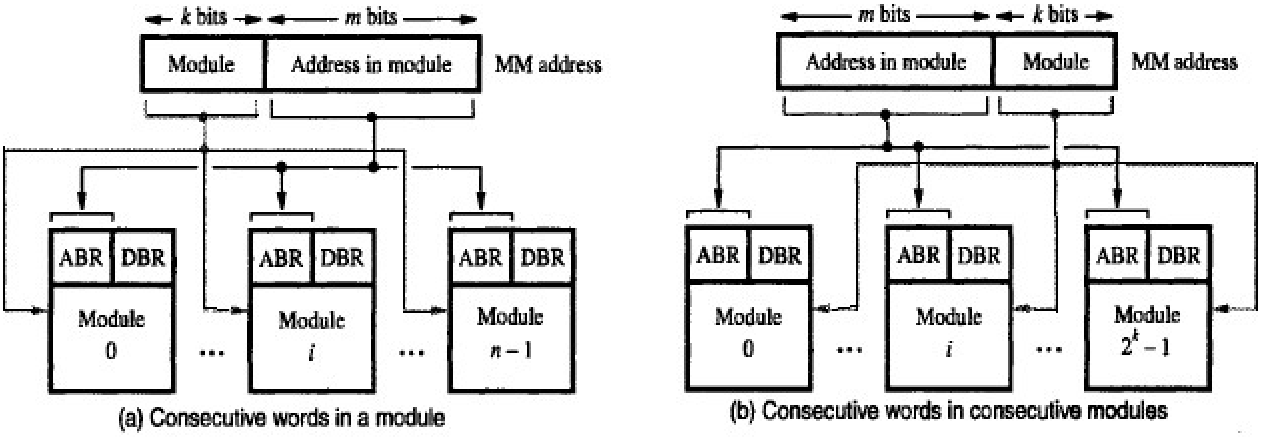
- ➢ 2 bit counter can be used for each block.
- ➢ When a **'hit'** occurs, then block counter=0; The counter with values originally lower than the referenced one are incremented by 1 & all others remain unchanged.
- ➢ When a **'miss'** occurs & if the set is full, the blocks with the counter value 3 is removed, the new block is put in its place & its counter is set to "0" and other block counters are incremented by 1.

PERFORMANCE CONSIDERATION

- Two key factors in the commercial success are 1) performance & 2) cost.
- In other words, the best possible performance at low cost.
- A common measure of success is called the **Pricel Performance ratio.**
- Performance depends on
  → how fast the machine instructions are brought to the processor &
  → how fast the machine instructions are executed.
- To achieve parallelism, *interleaving* is used.
- Parallelism means both the slow and fast units are accessed in the same manner.

INTERLEAVING

- The main-memory of a computer is structured as a collection of physically separate modules.
- Each module has its own
  1) ABR (address buffer register) &
  2) DBR (data buffer register).
- So, memory access operations may proceed in more than one module at the same time (Fig 5.25).
- Thus, the aggregate-rate of transmission of words to/from the main-memory can be increased.



(a) Consecutive words in a module

(b) Consecutive words in consecutive modules

Figure 5.25 Addressing multiple-module memory systems.

- The low-order k-bits of the memory-address select a module.
  While the high-order m-bits name a location within the module.
  In this way, consecutive addresses are located in successive modules.
- Thus, any component of the system can keep several modules busy at any one time T.
- This results in both
  → faster access to a block of data and
  → higher average utilization of the memory-system as a whole.
- To implement the interleaved-structure, there must be 2k modules;
  Otherwise, there will be gaps of non-existent locations in the address-space.

Hit Rate & Miss Penalty

- The number of hits stated as a fraction of all attempted accesses is called the **Hit Rate**.
- The extra time needed to bring the desired information into the cache is called the **Miss Penalty**.
- High hit rates well over 0.9 are essential for high-performance computers.
- Performance is adversely affected by the actions that need to be taken when a miss occurs.
- A performance penalty is incurred because
  of the extra time needed to bring a block of data from a slower unit to a faster unit.
- During that period, the processor is stalled waiting for instructions or data.
- We refer to the total access time seen by the processor when a miss occurs as the miss penalty.
- Let h be the hit rate, M the miss penalty, and C the time to access information in the cache. Thus, the average access time experienced by the processor is

$$t_{avg} = hC + (1 - h)M$$

VIRTUAL MEMORY

- It refers to a technique that automatically move program/data blocks into the main-memory when they are required for execution (Figure 8.24).
- The address generated by the processor is referred to as a **virtual/logical address**.
- The virtual-address is translated into physical-address by **MMU** (Memory Management Unit).
- During every memory-cycle, MMU determines whether the addressed-word is in the memory.

    If the word is in memory.

        Then, the word is accessed and execution proceeds.

        Otherwise, a page containing desired word is transferred from disk to memory.
- Using DMA scheme, transfer of data between disk and memory is performed.

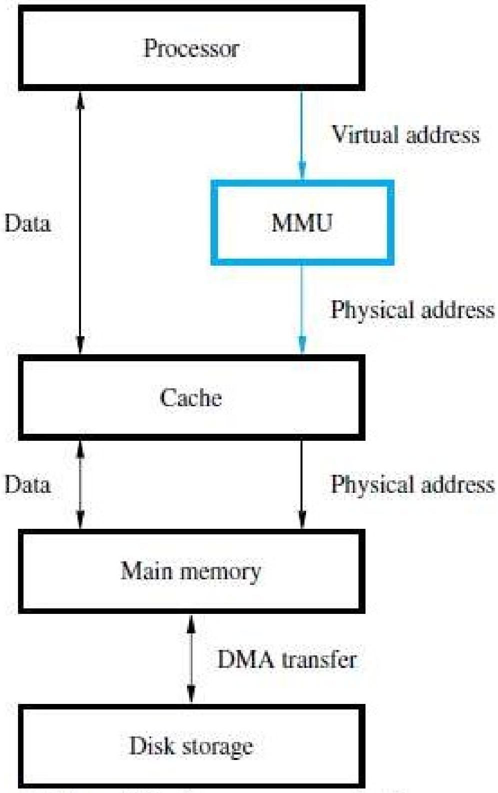


Figure 8.24 Virtual memory organization.

VIRTUAL MEMORY ADDRESS TRANSLATION

- All programs and data are composed of fixed length units called **Pages** (Figure 8.25).

  The Page consists of a block-of-words. The words occupy contiguous locations in the memory.

  The pages are commonly range from 2K to 16K bytes in length.
- **Cache Bridge** speed-up the gap between main-memory and secondary-storage.
- Each virtual-address contains

  1) Virtual Page number (Low order bit) and

  2) Offset (High order bit).

  Virtual Page number + Offset → specifies the location of a particular word within a page.
- **Page-table:** It contains the information about

  → memory-address where the page is stored &

  → current status of the page.
- **Page-frame:** An area in the main-memory that holds one page.
- **Page-table Base Register:** It contains the starting address of the page-table.
- *Virtual Page Number + Page-table Base register* → Gives the starting address of the page if that page currently resides in memory.
- **Control-bits in Page-table:** The Control-bits is used to

  1) Specify the status of the page while it is in memory.

  2) Indicate the validity of the page.

  3) Indicate whether the page has been modified during its stay in the memory.

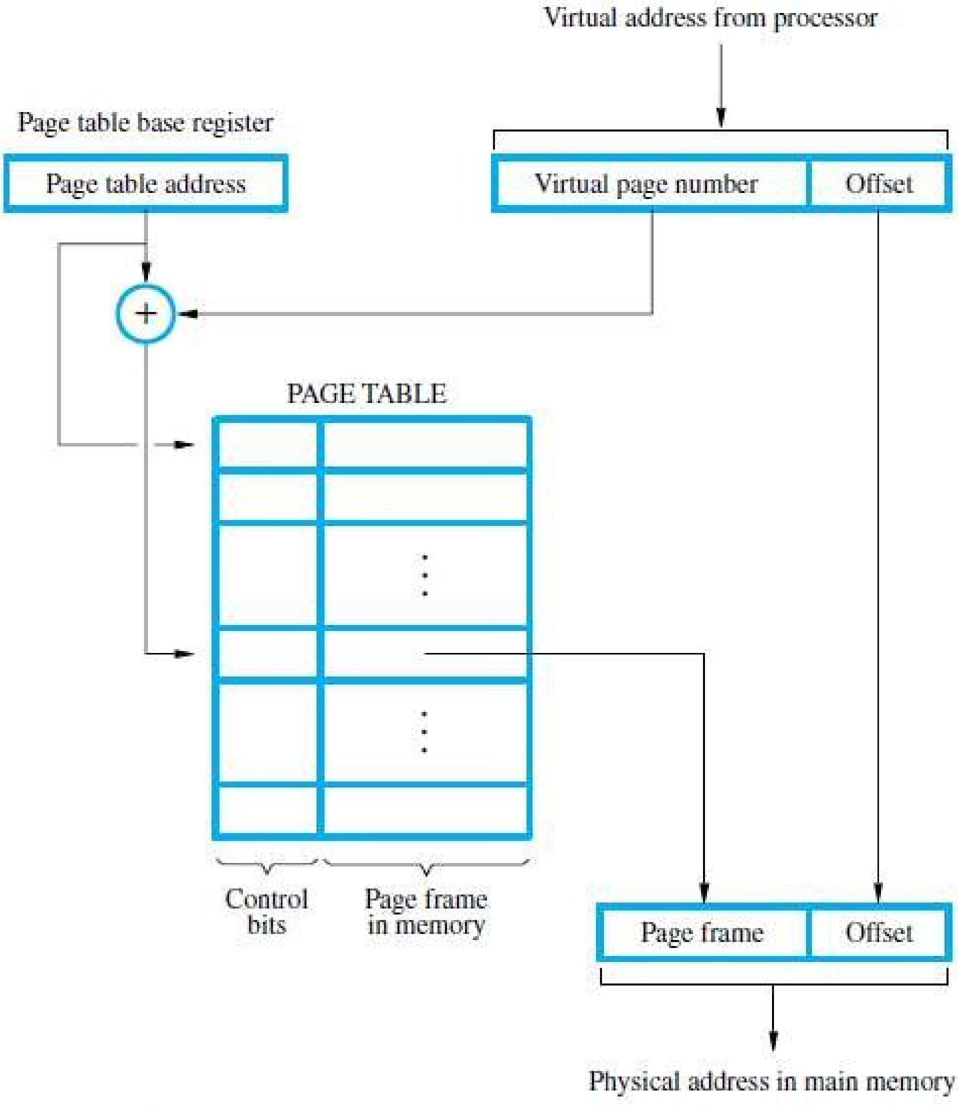


Figure 8.25 Virtual-memory address translation.

TRANSLATION LOOKASIDE BUFFER (TLB)

- The Page-table information is used by MMU for every read/write access (Figure 8.26).
- The Page-table is placed in the memory but a copy of small portion of the page-table is located within MMU. This small portion is called **TLB** (Translation LookAside Buffer).

TLB consists of the page-table entries that corresponds to the most recently accessed pages.

TLB also contains the virtual-address of the entry.

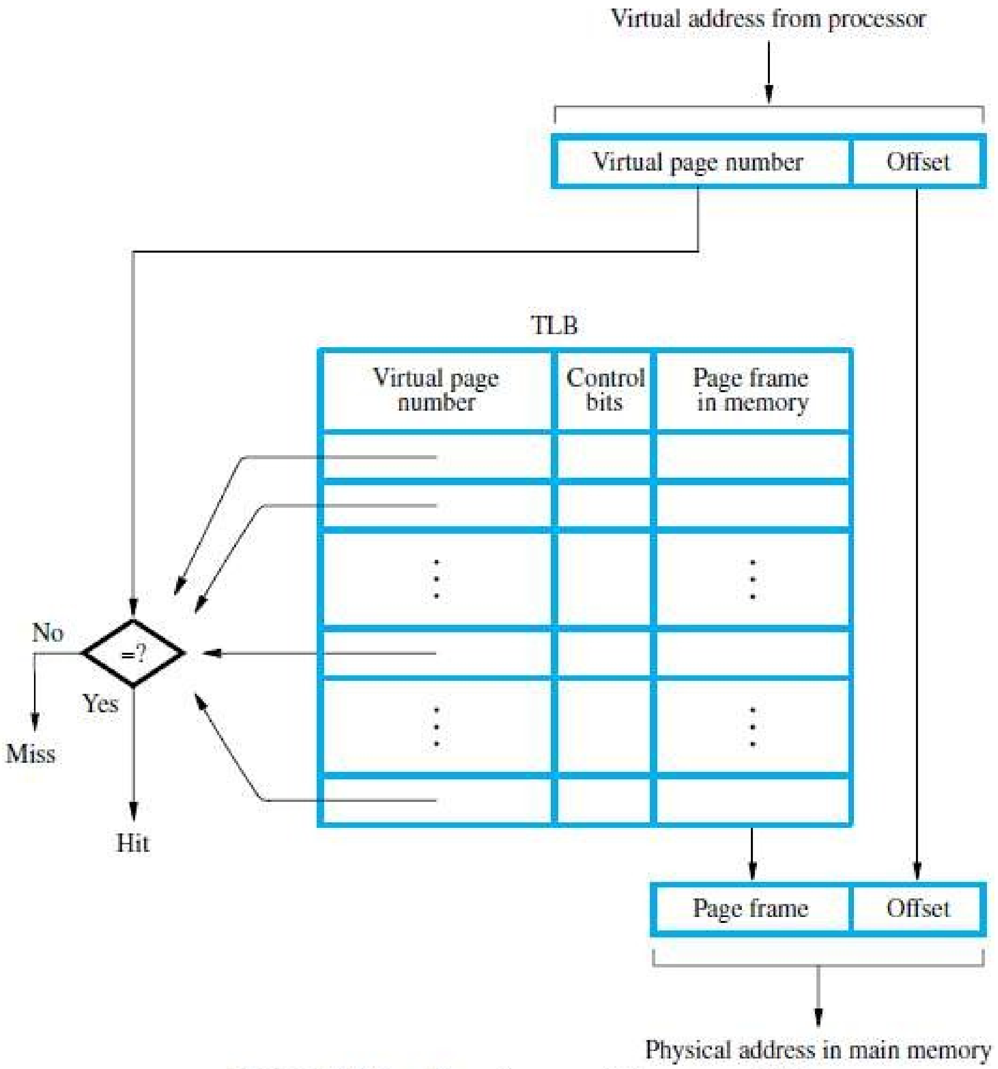


Figure 8.26 Use of an associative-mapped TLB.

- When OS changes contents of page-table, the control-bit will invalidate corresponding entry in TLB.
- Given a virtual-address, the MMU looks in TLB for the referenced-page.

If page-table entry for this page is found in TLB, the physical-address is obtained immediately.

Otherwise, the required entry is obtained from the page-table & TLB is updated.

Page Faults

- Page-fault occurs when a program generates an access request to a page that is not in memory.
- When MMU detects a page-fault, the MMU asks the OS to generate an interrupt.
- The OS

→ suspends the execution of the task that caused the page-fault and

→ begins execution of another task whose pages are in memory.

- When the task resumes the interrupted instruction must continue from the point of interruption.
- If a new page is brought from disk when memory is full, disk must replace one of the resident pages.

In this case, **LRU algorithm** is used to remove the least referenced page from memory.

- A modified page has to be written back to the disk before it is removed from the memory.

In this case, **Write–Through Protocol** is used.

MEMORY MANAGEMENT REQUIREMENTS

- Management routines are part of the Operating-system.
- Assembling the OS routine into virtual-address-space is called **System Space**.
- The virtual space in which the user application programs reside is called the **User Space**.
- Each user space has a separate page-table.
- MMU uses the page-table to determine the address of the table to be used in the translation process.
- The process has two stages:
  - **1) User State:** In this state, the processor executes the user-program.
  - **2) Supervisor State:** In this state, the processor executes the OS routines.

Privileged Instruction

- In user state, the machine instructions cannot be executed.
- Hence a user-program is prevented from accessing the page-table of system-spaces.
- The control-bits in each entry can be set to control the access privileges granted to each program.

  i.e. One program may be allowed to read/write a given page.

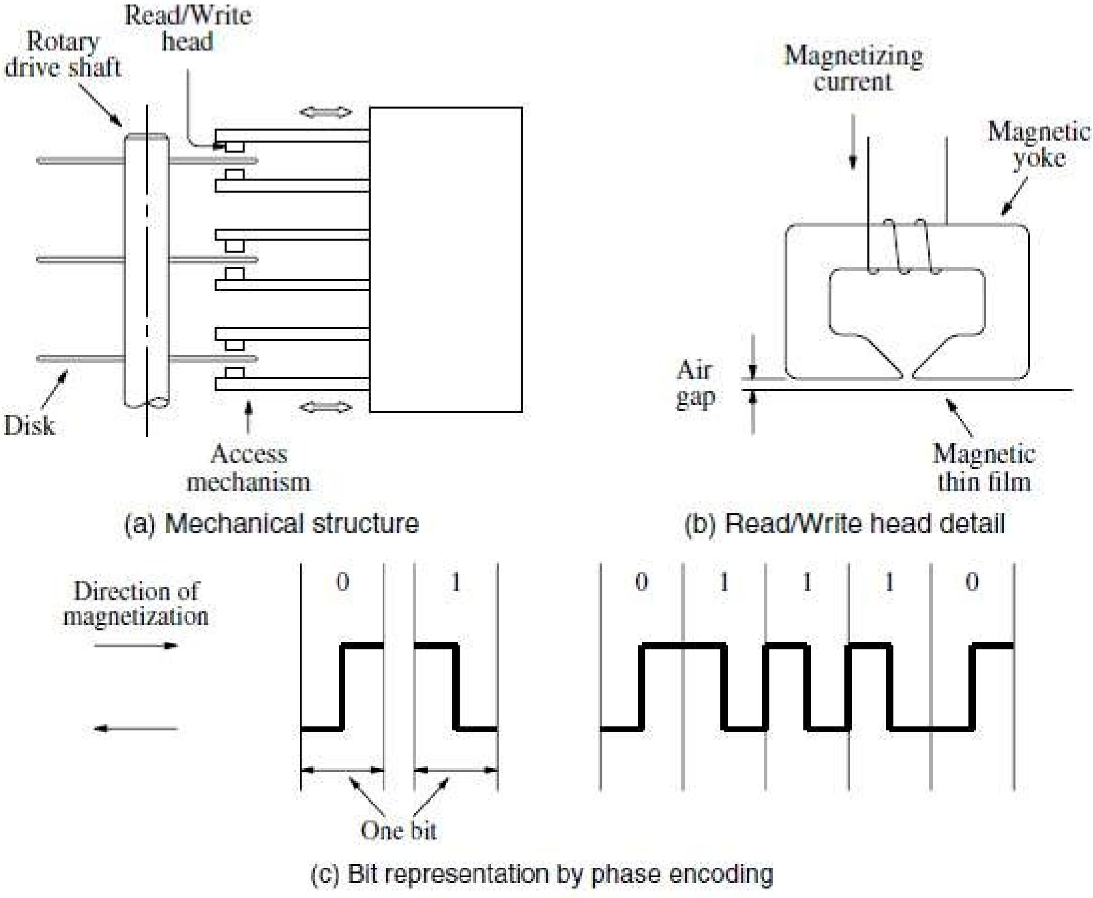
  While the other programs may be given only read access.

SECONDARY-STORAGE

- The semi-conductor memories do not provide all the storage capability.
- The secondary-storage devices provide larger storage requirements.
- Some of the secondary-storage devices are:
  1) Magnetic Disk
  2) Optical Disk &
  3) Magnetic Tapes.

MAGNETIC DISK

- Magnetic Disk system consists of one or more **disk** mounted on a common **spindle**.
- A **thin magnetic film** is deposited on each disk (Figure 8.27).
- Disk is placed in a **rotary-drive** so that magnetized surfaces move in close proximity to R/W heads.
- Each **R/W head** consists of 1) Magnetic Yoke & 2) Magnetizing-Coil.
- Digital information is stored on magnetic film by applying current pulse to the magnetizing-coil.
- Only changes in the magnetic field under the head can be sensed during the Read-operation.
- Therefore, if the binary states 0 & 1 are represented by two opposite states,
  then a voltage is induced in the head only at 0-1 and at 1-0 transition in the bit stream.
- A consecutive of 0"s & 1"s are determined by using the clock.
- **Manchester Encoding** technique is used to combine the clocking information with data.



(a) Mechanical structure

(b) Read/Write head detail

(c) Bit representation by phase encoding

Figure 8.27 Magnetic disk principles.

- R/W heads are maintained at small distance from disk-surfaces in order to achieve high bit densities.
- When disk is moving at their steady state, the air pressure develops b/w disk-surfaces & head.
  This air pressure forces the head away from the surface.
- The flexible spring connection between head and its arm mounting permits the head to fly at the desired distance away from the surface.

Winchester Technology

- Read/Write heads are placed in a sealed, air–filtered enclosure called the Winchester Technology.
- The read/write heads can operate closure to magnetic track surfaces because
  the dust particles which are a problem in unsealed assemblies are absent.

Advantages

- It has a larger capacity for a given physical size.
- The data intensity is high because
  the storage medium is not exposed to contaminating elements.
- The read/write heads of a disk system are movable.
- The disk system has 3 parts: 1) Disk Platter (Usually called Disk)
  2) Disk-drive (spins the disk & moves Read/write heads)
  3) Disk Controller (controls the operation of the system.)

ORGANIZATION & ACCESSING OF DATA ON A DISK

- Each surface is divided into concentric **Tracks** (Figure 8.28).
- Each track is divided into **Sectors.**
- The set of corresponding tracks on all surfaces of a stack of disk form a **Logical Cylinder**.
- The data are accessed by specifying *the surface number, track number and the sector number.*
- The Read/Write-operation start at sector boundaries.
- Data bits are stored serially on each track.

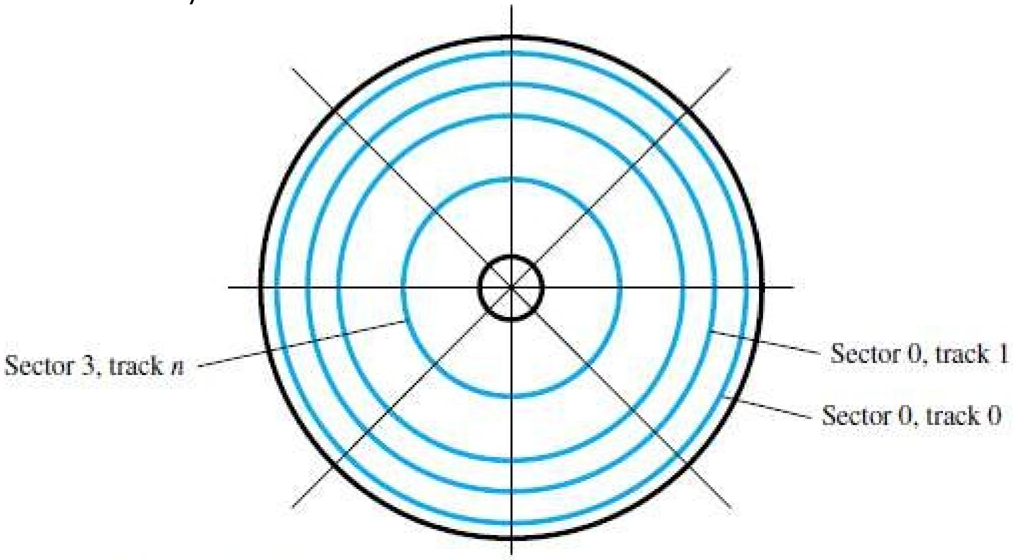


Figure 8.28 Organization of one surface of a disk.

- Each sector usually contains 512 bytes.
- **Sector Header** --> contains identification information.
  It helps to find the desired sector on the selected track.
- **ECC** (Error checking code)- is used to detect and correct errors.
- An unformatted disk has no information on its tracks.
- The formatting process divides the disk physically into tracks and sectors.
- The formatting process may discover some defective sectors on all tracks.
- **Disk Controller** keeps a record of various defects.
- The disk is divided into logical partitions:
  1) Primary partition
  2) Secondary partition
- Each track has same number of sectors. So, all tracks have same storage capacity.
- Thus, the stored information is packed more densely on inner track than on outer track.

Access Time

- There are 2 components involved in the time-delay:

  1) Seek time: Time required to move the read/write head to the proper track.

  2) Latency/Rotational Delay: The amount of time that elapses after head is positioned over the correct track until the starting position of the addressed sector passes under the R/W head.

  Seek time + Latency = Disk access time

  Typical Disk

  ➢ One inch disk-weight = 1 ounce, size -> comparable to match book
  Capacity -> 1GB

  ➢ Inch disk has the following parameter
  Recording surface=20
  Tracks=15000 tracks/surface
  Sectors=400.
  Each sector stores 512 bytes of data

  Capacity of formatted disk=20x15000x400x512=60x109 =60GB
  Seek time=3ms
  Platter rotation=10000 rev/min
  Latency=3ms
  Internet transfer rate=34MB/s

DATA BUFFER/CACHE

- A disk-drive that incorporates the required SCSI circuit is referred as **SCSI Drive**.
- The SCSI can transfer data at higher rate than the disk tracks.
- A data buffer can be used to deal with the possible difference in transfer rate b/w disk and SCSI bus
- The buffer is a semiconductor memory.
- The buffer can also provide cache mechanism for the disk.

    i.e. when a read request arrives at the disk, then controller first check if the data is available in the cache/buffer.

    If data is available in cache.

    Then, the data can be accessed & placed on SCSI bus.

    Otherwise, the data will be retrieved from the disk.

DISK CONTROLLER

- The disk controller acts as interface between disk-drive and system-bus (Figure 8.13).
- The disk controller uses DMA scheme to transfer data between disk and memory.
- When the OS initiates the transfer by issuing R/W" request, the controllers register will load the following information:

    1) Memory Address: Address of first memory-location of the block of words involved in the transfer.

    2) Disk Address: Location of the sector containing the beginning of the desired block of words.

    3) Word Count: Number of words in the block to be transferred.

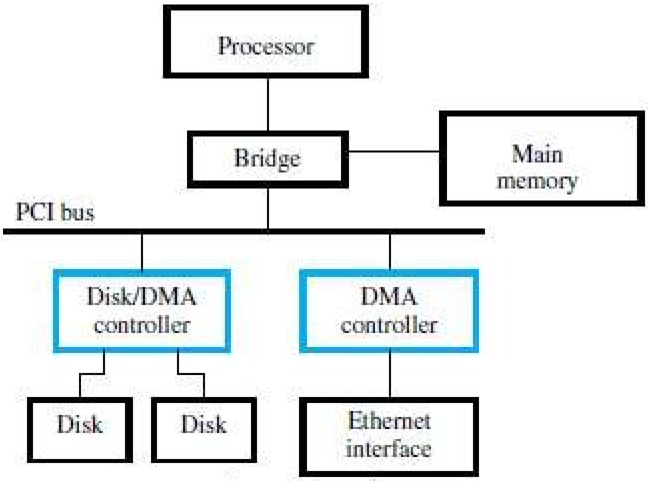


Figure 8.13 Use of DMA controllers in a computer system.

- The disk-address issued by the OS is a logical address.
- The corresponding physical-address on the disk may be different.
- The controller's major functions are:

    1) Seek - Causes disk-drive to move the R/W head from its current position to desired track.

    2) Read - Initiates a Read-operation, starting at address specified in the disk-address register.

    Data read serially from the disk are assembled into words and placed into the data buffer for transfer to the main-memory.

    3) Write - Transfers data to the disk.

    4) Error Checking - Computes the error correcting code (ECC) value for the data read from a given sector and compares it with the corresponding ECC value read from the disk.

    In case of a mismatch, it corrects the error if possible;

    Otherwise, it raises an interrupt to inform the OS that an error has occurred.

Problem 1:

Consider the dynamic memory cell. Assume that *C* = 30 femtofarads (10^{-15} F) and that leakage current through the transistor is about 0.25 picoamperes (10^{-12} A). The voltage across the capacitor when it is fully charged is 1.5 V. The cell must be refreshed before this voltage drops below 0.9 V. Estimate the minimum refresh rate.

Solution:

The minimum refresh rate is given by

$$\frac{50 \times 10^{-15} \times (4.5 - 3)}{9 \times 10^{-12}} = 8.33 \times 10^{-3} \text{ s}$$

Therefore, each row has to be refreshed every 8 ms.

Problem 2:

Consider a main-memory built with SDRAM chips. Data are transferred in bursts & the burst length is 8. Assume that 32 bits of data are transferred in parallel. If a 400-MHz clock is used, how much time does it take to transfer:

(a) 32 bytes of data

(b) 64 bytes of data

What is the latency in each case?

Solution:

(a) It takes 5 + 8 = 13 clock cycles.

$$\text{Total time} = \frac{13}{(133 \times 10^{6})} = 0.098 \times 10^{-6} \text{ s} = 98 \text{ ns}$$

$$\text{Latency} = \frac{5}{(133 \times 10^{6})} = 0.038 \times 10^{-6} \text{ s} = 38 \text{ ns}$$

(b) It takes twice as long to transfer 64 bytes, because two independent 32-byte transfers have to be made. The latency is the same, i.e. 38 ns.

Problem 3:

Give a critique of the following statement: “Using a faster processor chip results in a corresponding increase in performance of a computer even if the main-memory speed remains the same.”

Solution:

A faster processor chip will result in increased performance, but the amount of increase will not be directly proportional to the increase in processor speed, because the cache miss penalty will remain the same if the main-memory speed is not improved.

Problem 4:

A block-set-associative cache consists of a total of 64 blocks, divided into 4-block sets. The main-memory contains 4096 blocks, each consisting of 32 words. Assuming a 32-bit byte-addressable address-space,

(a) how many bits are there in main-memory address

(b) how many bits are there in each of the Tag, Set, and Word fields?

Solution:

(a) 4096 blocks of 128 words each require 12+7 = 19 bits for the main-memory address.

(b) TAG field is 8 bits. SET field is 4 bits. WORD field is 7 bits.

Problem 5:

The cache block size in many computers is in the range of 32 to 128 bytes. What would be the main advantages and disadvantages of making the size of cache blocks larger or smaller?

Solution:

Larger size

- Fewer misses if most of the data in the block are actually used
- Wasteful if much of the data are not used before the cache block is ejected from the cache

Smaller size

- More misses

Problem 5:

Consider a computer system in which the available pages in the physical memory are divided among several application programs. The operating system monitors the page transfer activity and dynamically adjusts the number of pages allocated to various programs. Suggest a suitable strategy that the operating system can use to minimize the overall rate of page transfers.

Solution:

The operating system may increase the main-memory pages allocated to a program that has a large number of page faults, using space previously allocated to a program with a few page faults

Problem 6:

In a computer with a virtual-memory system, the execution of an instruction may be interrupted by a page fault. What state information has to be saved so that this instruction can be resumed later? Note that bringing a new page into the main-memory involves a DMA transfer, which requires execution of other instructions. Is it simpler to abandon the interrupted instruction and completely re-execute it later? Can this be done?

Solution:

Continuing the execution of an instruction interrupted by a page fault requires saving the entire state of the processor, which includes saving all registers that may have been affected by the instruction as well as the control information that indicates how far the execution has progressed. The alternative of re-executing the instruction from the beginning requires a capability to reverse any changes that may have been caused by the partial execution of the instruction.

Problem 7:

When a program generates a reference to a page that does not reside in the physical main-memory, execution of the program is suspended until the requested page is loaded into the main-memory from a disk. What difficulties might arise when an instruction in one page has an operand in a different page? What capabilities must the processor have to handle this situation?

Solution:

The problem is that a page fault may occur during intermediate steps in the execution of a single instruction. The page containing the referenced location must be transferred from the disk into the main-memory before execution can proceed.

Since the time needed for the page transfer (a disk operation) is very long, as compared to instruction execution time, a context-switch will usually be made.

(A context-switch consists of preserving the state of the currently executing program, and ”switching” the processor to the execution of another program that is resident in the main-memory.) The page transfer, via DMA, takes place while this other program executes. When the page transfer is complete, the original program can be resumed.

Therefore, one of two features are needed in a system where the execution of an individual instruction may be suspended by a page fault. The first possibility is to save the state of instruction execution. This involves saving more information (temporary programmer-transparent registers, etc.) than needed when a program is interrupted between instructions. The second possibility is to ”unwind” the effects of the portion of the instruction completed when the page fault occurred, and then execute the instruction from the beginning when the program is resumed.

Problem 8:

Magnetic disks are used as the secondary storage for program and data files in most virtual-memory systems. Which disk parameter(s) should influence the choice of page size?

Solution:

The sector size should influence the choice of page size, because the sector is the smallest directly addressable block of data on the disk that is read or written as a unit. Therefore, pages should be some small integral number of sectors in size.

Problem 9:

A disk unit has 24 recording surfaces. It has a total of 14,000 cylinders. There is an average of 400 sectors per track. Each sector contains 512 bytes of data.

(a) What is the maximum number of bytes that can be stored in this unit?

(b) What is the data-transfer rate in bytes per second at a rotational speed of 7200 rpm?

(c) Using a 32-bit word, suggest a suitable scheme for specifying the disk address.

Solution:

(a) The maximum number of bytes that can be stored on this disk is 24 X 14000 X 400 X 512 = 68.8 X 10^9 bytes.

(b) The data-transfer rate is (400 X 512 X 7200)/60 = 24.58 X 10^6 bytes/s.

(c) Need 9 bits to identify a sector, 14 bits for a track, and 5 bits for a surface.

Thus, a possible scheme is to use address bits A_{8-0} for sector, A_{22-9} for track, and A_{27-23} for surface identification. Bits A_{31-28} are not used.

MODULE 4: ARITHMETIC

NUMBERS, ARITHMETIC OPERATIONS AND CHARACTERS

NUMBER REPRESENTATION

- Numbers can be represented in 3 formats:
  1) Sign and magnitude
  2) 1's complement
  3) 2's complement
- In all three formats, MSB=0 for +ve numbers & MSB=1 for -ve numbers.
- In **sign-and-magnitude system**,

  negative value is obtained by changing the MSB from 0 to 1 of the corresponding positive value.

  For ex, +5 is represented by 0101 &

  -5 is represented by 1101.
- In **1's complement system**,

  negative values are obtained by complementing each bit of the corresponding positive number.

  For ex, -5 is obtained by complementing each bit in 0101 to yield 1010.

(In other words, the operation of forming the 1's complement of a given number is equivalent to subtracting that number from 2^n-1).

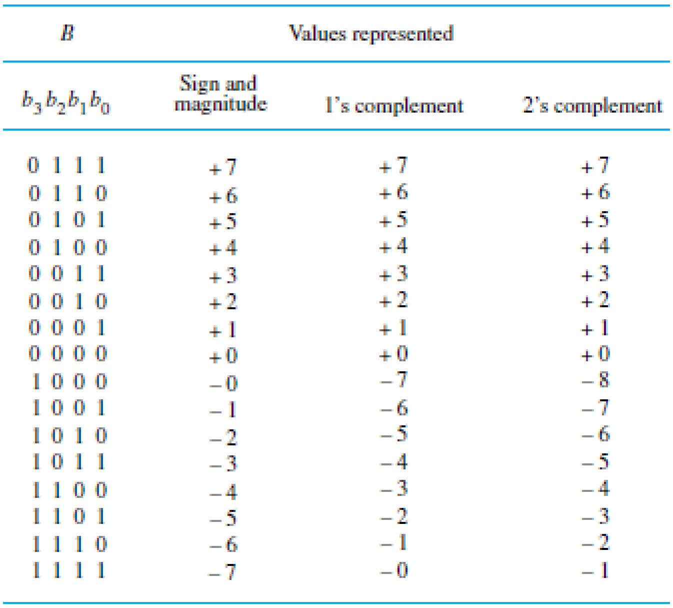
- In **2's complement system**,

  forming the 2's complement of a number is done by subtracting that number from 2^n.

  For ex, -5 is obtained by complementing each bit in 0101 & then adding 1 to yield 1011.

(In other words, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number).

- 2's complement system yields the most efficient way to carry out addition/subtraction operations.



B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Figure 1.3 Binary, signed-integer representations.

ADDITION OF POSITIVE NUMBERS

- Consider adding two 1-bit numbers.
- The sum of 1 & 1 requires the 2-bit vector 10 to represent the value 2. We say that sum is 0 and the carry-out is 1.

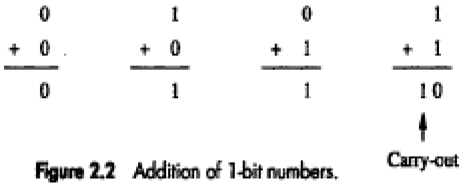


Figure 2.2 Addition of 1-bit numbers.

ADDITION & SUBTRACTION OF SIGNED NUMBERS

- Following are the two rules for addition and subtraction of n-bit signed numbers using the 2's complement representation system (Figure 1.6).

Rule 1:

➢ **To Add** two numbers, add their n-bits and ignore the carry-out signal from the MSB position.

➢ Result will be algebraically correct, if it lies in the range -2^{n-1} to $+2^{n-1}-1$.

Rule 2:

➢ **To Subtract** two numbers X and Y (that is to perform X-Y), take the 2's complement of Y and then add it to X as in rule 1.

➢ Result will be algebraically correct, if it lies in the range (2^{n-1}) to $+(2^{n-1}-1)$.

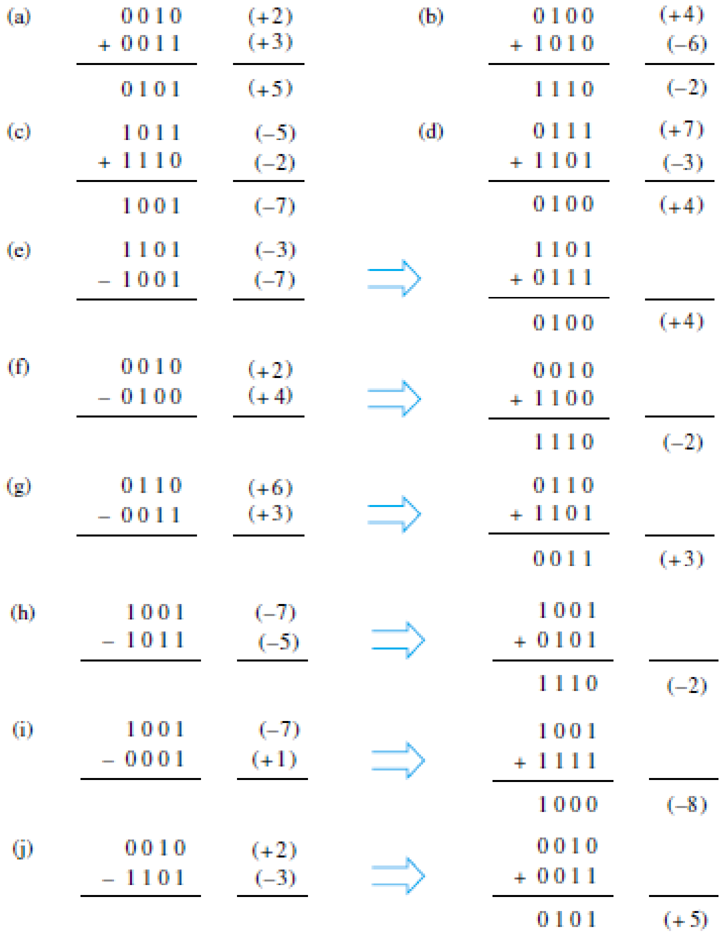
- When the result of an arithmetic operation is outside the representable-range, an arithmetic overflow is said to occur.
- To represent a signed in 2's complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called **sign extension**.
- In 1's complement representation, the result obtained after an addition operation is not always correct. The carry-out(c_n) cannot be ignored. If c_n=0, the result obtained is correct. If c_n=1, then a 1 must be added to the result to make it correct.

OVERFLOW IN INTEGER ARITHMETIC

- When result of an arithmetic operation is outside the representable-range, an **arithmetic overflow** is said to occur.
- For example: If we add two numbers +7 and +4, then the output sum S is 1011(←0111+0100), which is the code for -5, an incorrect result.
- An overflow occurs in following 2 cases

1) Overflow can occur only when adding two numbers that have the same sign.

2) The carry-out signal from the sign-bit position is not a sufficient indicator of overflow when adding signed numbers.



```
(a)     0010    (+2)          (b)     0100    (+4)
      + 0011    (+3)                + 1010    (-6)
      ------    ----                ------    ----
        0101    (+5)                  1110    (-2)

(c)     1011    (-5)          (d)     0111    (+7)
      + 1110    (-2)                + 1101    (-3)
      ------    ----                ------    ----
        1001    (-7)                  0100    (+4)

(e)     1101    (-3)                  1101
      - 1001    (-7)     ⇒          + 0111
      ------    ----                ------    ----
                                      0100    (+4)

(f)     0010    (+2)                  0010
      - 0100    (+4)     ⇒          + 1100
      ------    ----                ------    ----
                                      1110    (-2)

(g)     0110    (+6)                  0110
      - 0011    (+3)     ⇒          + 1101
      ------    ----                ------    ----
                                      0011    (+3)

(h)     1001    (-7)                  1001
      - 1011    (-5)     ⇒          + 0101
      ------    ----                ------    ----
                                      1110    (-2)

(i)     1001    (-7)                  1001
      - 0001    (+1)     ⇒          + 1111
      ------    ----                ------    ----
                                      1000    (-8)

(j)     0010    (+2)                  0010
      - 1101    (-3)     ⇒          + 0011
      ------    ----                ------    ----
                                      0101    (+5)
```

Figure 1.6 2's-complement Add and Subtract operations.

ADDITION & SUBTRACTION OF SIGNED NUMBERS

n-BIT RIPPLE CARRY ADDER

- A cascaded connection of n full-adder blocks can be used to add 2-bit numbers.
- Since carries must propagate (or ripple) through cascade, the configuration is called an n-bit ripple carry adder (Figure 9.1).

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i \overline{y}_i \overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

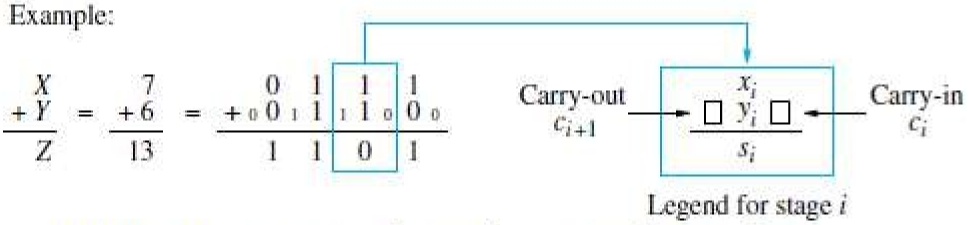
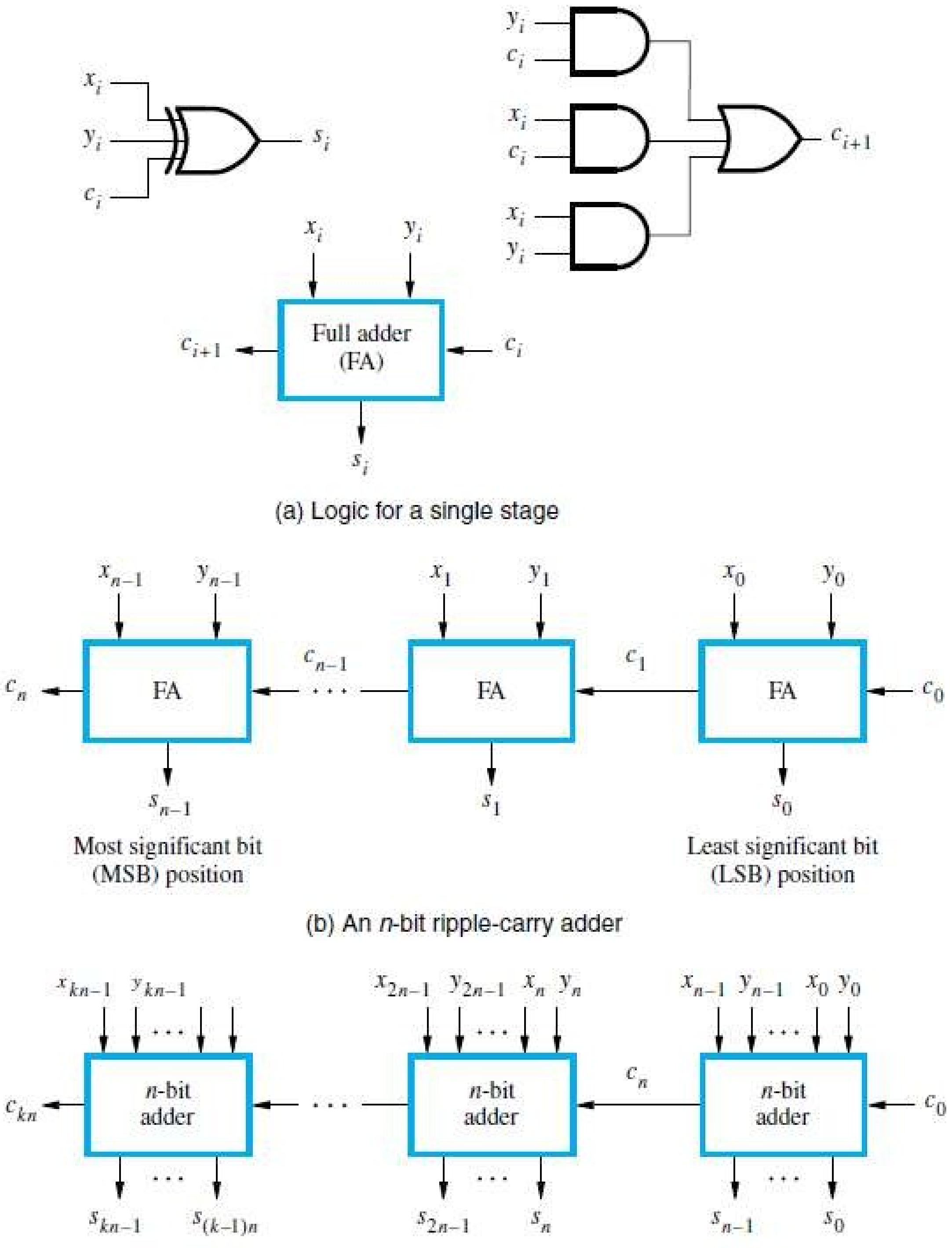


Figure 9.1 Logic specification for a stage of binary addition.



(a) Logic for a single stage

(b) An n-bit ripple-carry adder

(c) Cascade of k n-bit adders

Figure 9.2 Logic for addition of binary numbers.

ADDITION/SUBTRACTION LOGIC UNIT

- The n-bit adder can be used to add 2's complement numbers X and Y (Figure 9.3).
- **Overflow** can only occur when the signs of the 2 operands are the same.
- In order to perform the subtraction operation X-Y on 2's complement numbers X and Y; we form the 2's complement of Y and add it to X.
- Addition or subtraction operation is done based on value applied to the Add/Sub input control-line.
- Control-line=0 for addition, applying the Y vector unchanged to one of the adder inputs.
  Control-line=1 for subtraction, the Y vector is 2's complemented.

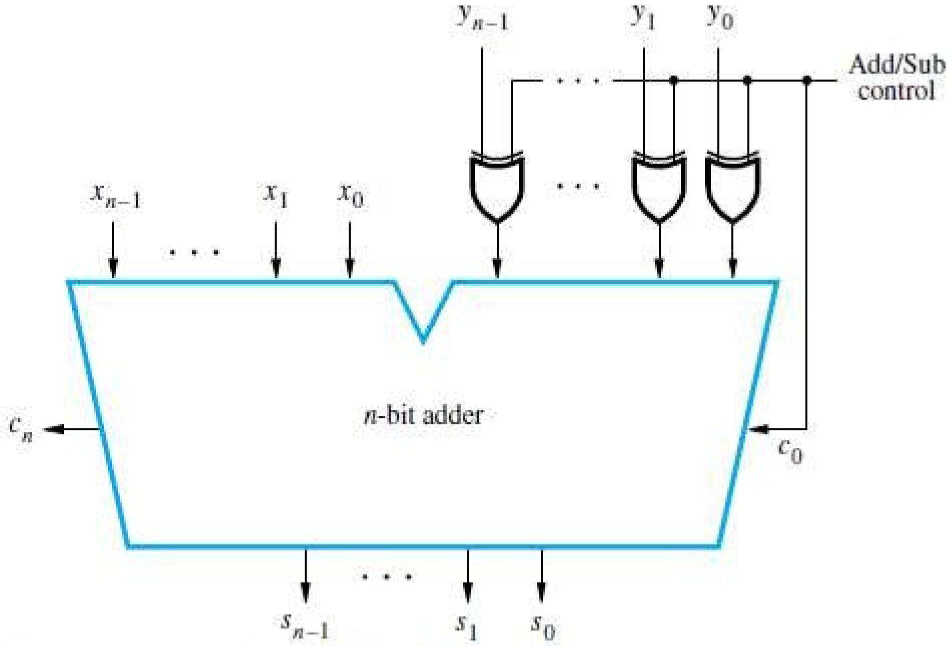


Figure 9.3 Binary addition/subtraction logic circuit.

DESIGN OF FAST ADDERS

- **Drawback of ripple carry adder:** If the adder is used to implement the addition/subtraction, all sum bits are available in 2n gate delays.
- Two approaches can be used to reduce delay in adders:
  1) Use the fastest possible electronic-technology in implementing the ripple-carry design.
  2) Use an augmented logic-gate network structure.

CARRY-LOOKAHEAD ADDITIONS

- The logic expression for s_i(sum) and c_{i+1}(carry-out) of stage i are

$s_i=x_i+y_i+c_i$ ------(1) $c_{i+1}=x_iy_i+x_ic_i+y_ic_i$ ------(2)

- Factoring (2) into

$c_{i+1}=x_iy_i+(x_i+y_i)c_i$

we can write

$c_{i+1}=G_i+P_iC_i$ where $G_i=x_iy_i$ and $P_i=x_i+y_i$

- The expressions G_i and P_i are called generate and propagate functions (Figure 9.4).
- If $G_i=1$, then $c_{i+1}=1$, independent of the input carry c_i. This occurs when both x_i and y_i are 1. Propagate function means that an input-carry will produce an output-carry when either $x_i=1$ or $y_i=1$.
- All G_i and P_i functions can be formed independently and in parallel in one logic-gate delay.
- Expanding c_i terms of i-1 subscripted variables and substituting into the c_{i+1} expression, we obtain

$c_{i+1}=G_i+P_iG_{i-1}+P_iP_{i-1}G_{i-2.}\ .\ .\ .\ .\ .\ .+P_1G_0+P_iP_{i-1}\ .\ .\ .\ P_0c_0$

- Conclusion: Delay through the adder is 3 gate delays for all carry-bits & 4 gate delays for all sum-bits.
- Consider the design of a 4-bit adder. The carries can be implemented as

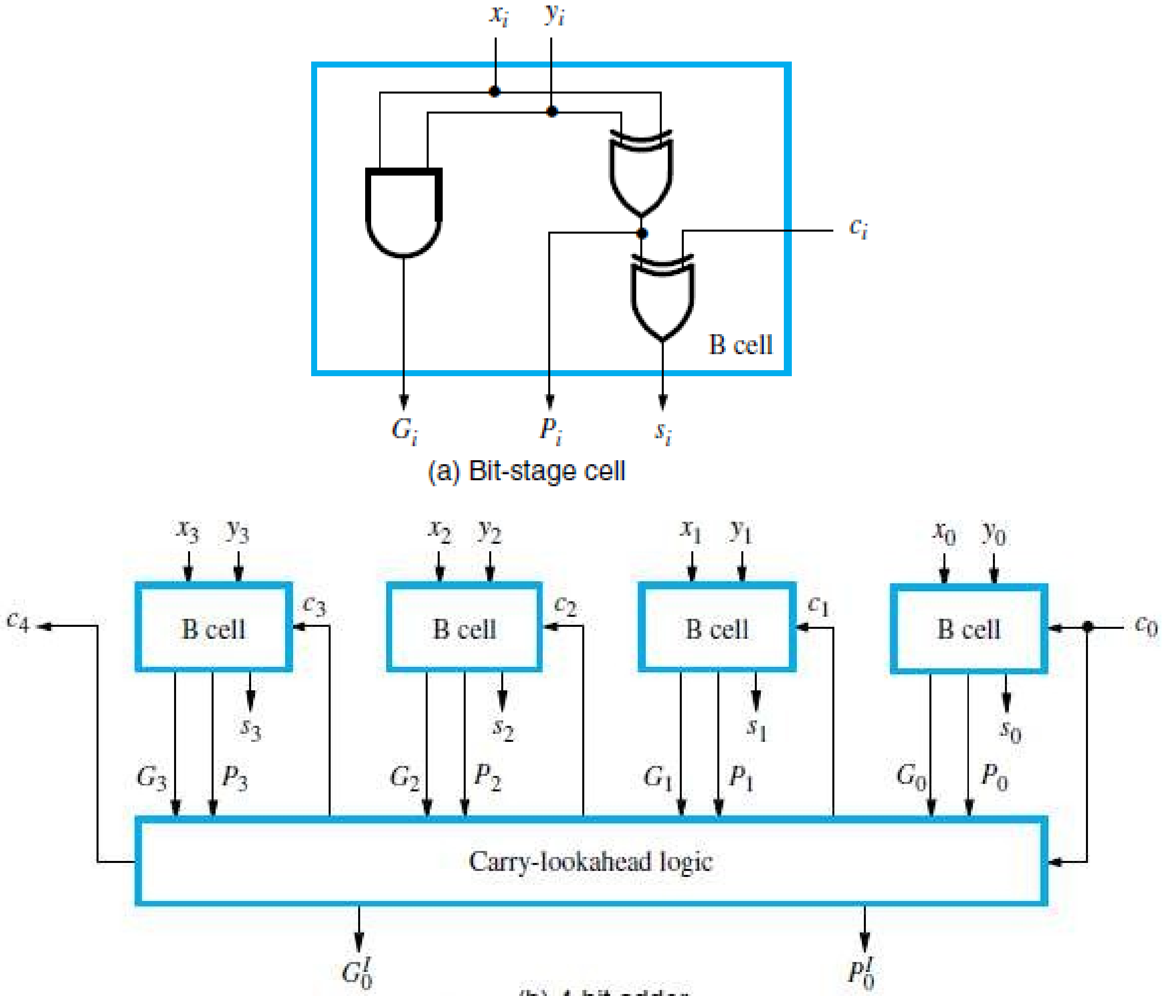
$c_1=G_0+P_0c_0$

$c_2=G_1+P_1G_0+P_1P_0c_0$

$c_3=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0c_0$

$c_4=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0c_0$

- The carries are implemented in the block labeled carry-lookahead logic. An adder implemented in this form is called a **Carry-Lookahead Adder**.
- Limitation: If we try to extend the carry-lookahead adder for longer operands, we run into a problem of gate fan-in constraints.



(a) Bit-stage cell

(b) 4-bit adder

Figure 9.4 A 4-bit carry-lookahead adder.

HIGHER-LEVEL GENERATE & PROPAGATE FUNCTIONS

- 16-bit adder can be built from four 4-bit adder blocks (Figure 9.5).
- These blocks provide new output functions defined as G_k and P_k,
  where k=0 for the first 4-bit block,
  k=1 for the second 4-bit block and so on.
- In the first block,

  $P_0=P_3P_2P_1P_0$

  &

  $G_0=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0$
- The first-level G_i and P_i functions determine whether bit stage i generates or propagates a carry, and the second level G_k and P_k functions determine whether block k generates or propagates a carry.
- Carry c_{16} is formed by one of the carry-lookahead circuits as

  $c_{16}=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0c_0$
- Conclusion: All carries are available 5 gate delays after X, Y and c_0 are applied as inputs.

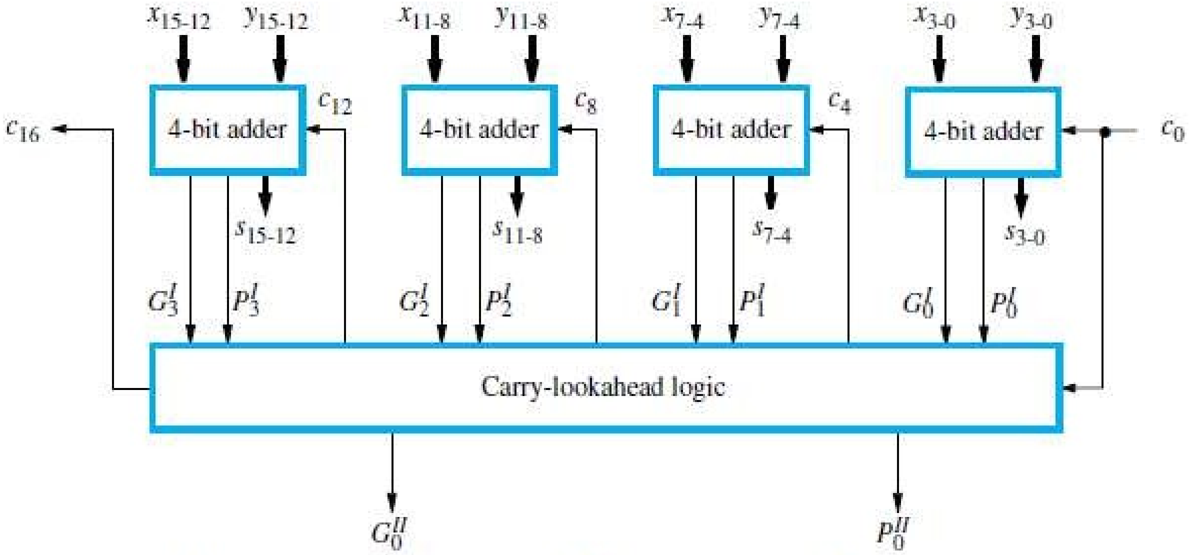
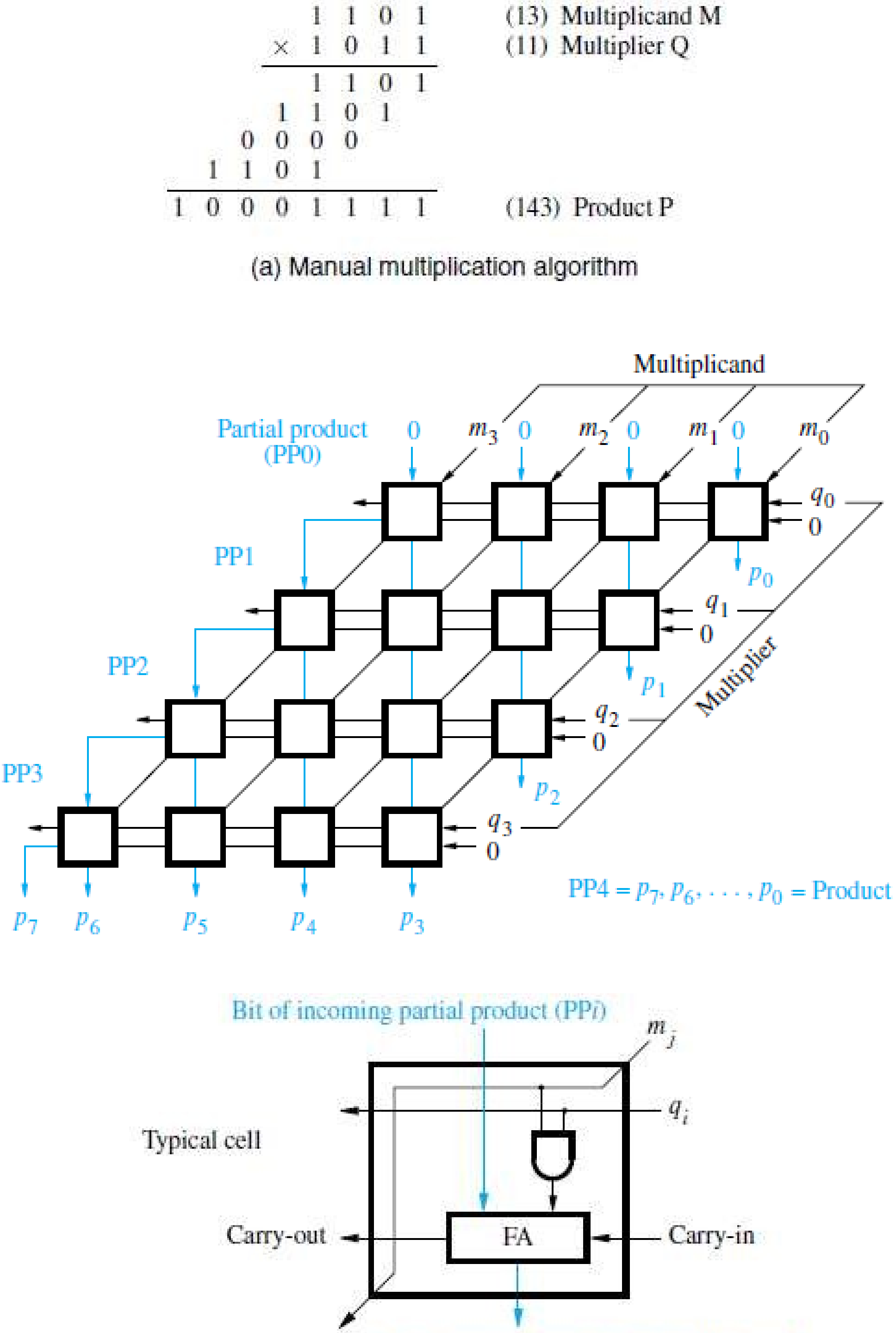


Figure 9.5 A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4*b*).

MULTIPLICATION OF POSITIVE NUMBERS



(a) Manual multiplication algorithm

(b) Array implementation

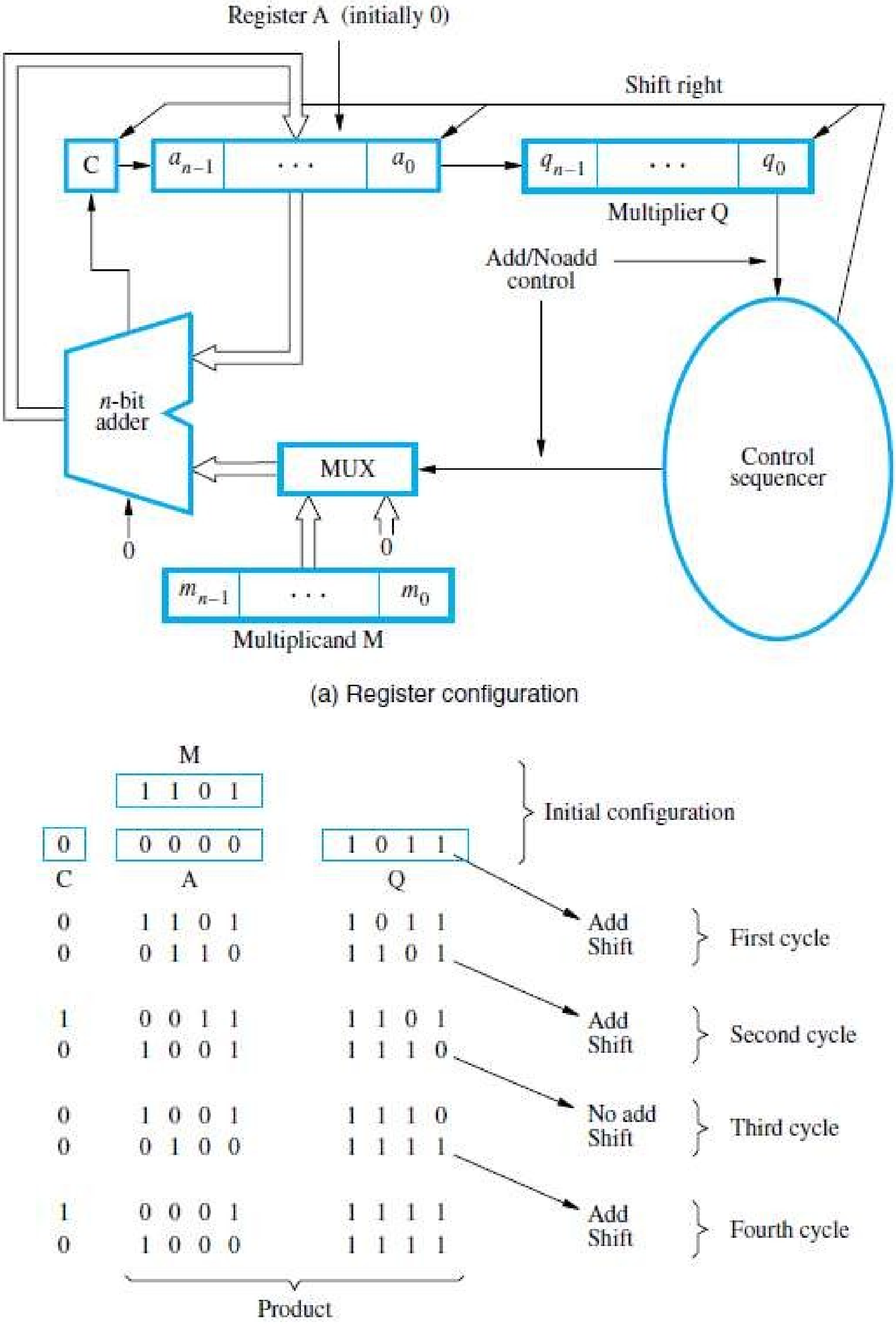
Figure 9.6 Array multiplication of unsigned binary operands.

ARRAY MULTIPLICATION

- The main component in each cell is a full adder(FA)..
- The AND gate in each cell determines whether a multiplicand bit m_j, is added to the incoming partial-product bit, based on the value of the multiplier bit q_i (Figure 9.6).

SEQUENTIAL CIRCUIT BINARY MULTIPLIER

- Registers A and Q combined hold PP_i(partial product)
  while the multiplier bit q_i generates the signal Add/Noadd.
- The carry-out from the adder is stored in flip-flop C (Figure 9.7).
- Procedure for multiplication:
  1) Multiplier is loaded into register Q,
     Multiplicand is loaded into register M and
     C & A are cleared to 0.
  2) If q_0=1, add M to A and store sum in A. Then C, A and Q are shifted right one bit-position.
     If q_0=0, no addition performed and C, A & Q are shifted right one bit-position.
  3) After n cycles, the high-order half of the product is held in register A and
     the low-order half is held in register Q.



(a) Register configuration

C	A	Q		
	M: 1 1 0 1			
0	0 0 0 0	1 0 1 1		Initial configuration
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

(b) Multiplication example

Figure 9.7 Sequential circuit binary multiplier.

SIGNED OPERAND MULTIPLICATION

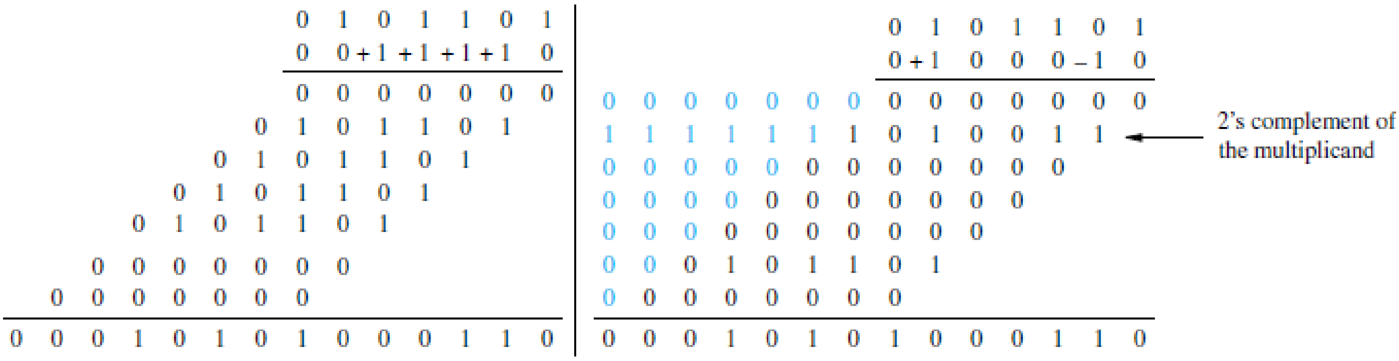
BOOTH ALGORITHM

- This algorithm
  - → generates a 2n-bit product
  - → treats both positive & negative 2's-complement n-bit operands uniformly(Figure 9.9-9.12).
- Attractive feature: This algorithm achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s.
- This algorithm suggests that we can reduce the number of operations required for multiplication by representing multiplier as a difference between 2 numbers.

For e.g. multiplier(Q) 14(001110) can be represented as

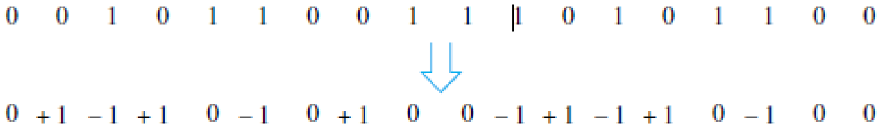
 010000 (16)
-000010 (2)
001110 (14)

- Therefore, product P=M*Q can be computed by adding 2^4 times the M to the 2's complement of 2^1 times the M.



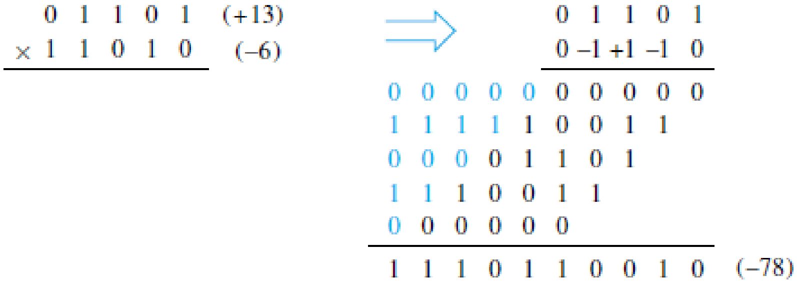
```
                  0  1  0  1  1  0  1          |                      0  1  0  1  1  0  1
                  0  0 +1 +1 +1 +1  0          |                      0 +1  0  0  0 -1  0
                  --------------------         |                      --------------------
                  0  0  0  0  0  0  0          |  0  0  0  0  0  0  0  0  0  0  0  0  0  0
               0  1  0  1  1  0  1             |  1  1  1  1  1  1  1  0  1  0  0  1  1     <--- 2's complement of
            0  1  0  1  1  0  1                |  0  0  0  0  0  0  0  0  0  0  0  0             the multiplicand
         0  1  0  1  1  0  1                   |  0  0  0  0  0  0  0  0  0  0  0
      0  1  0  1  1  0  1                      |  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0                         |  0  0  0  1  0  1  1  0  1
0  0  0  0  0  0  0                            |  0  0  0  0  0  0  0  0
-------------------------------------------    |  -----------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0       |  0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

Figure 9.9 Normal and Booth multiplication schemes.



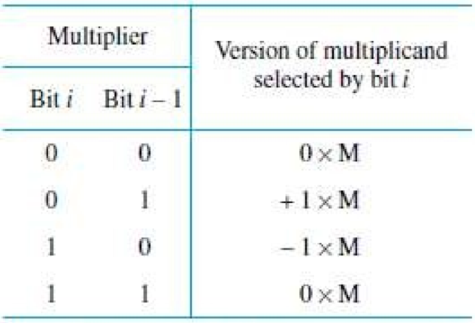
```
0   0   1   0   1   1   0   0   1   1  |1   0   1   0   1   1   0   0
                                    ⇓
0  +1  -1  +1   0  -1   0  +1   0   0  -1  +1  -1  +1   0  -1   0   0
```

Figure 9.10 Booth recoding of a multiplier.



```
    0  1  1  0  1   (+13)                     0  1  1  0  1
 ×  1  1  0  1  0   (−6)      ⟹               0 −1 +1 −1  0
 ----------------                  ----------------------------
                               0  0  0  0  0  0  0  0  0  0
                               1  1  1  1  1  0  0  1  1
                               0  0  0  0  1  1  0  1
                               1  1  1  0  0  1  1
                               0  0  0  0  0  0
                               ----------------------------
                               1  1  1  0  1  1  0  0  1  0   (−78)
```

Figure 9.11 Booth multiplication with a negative multiplier.



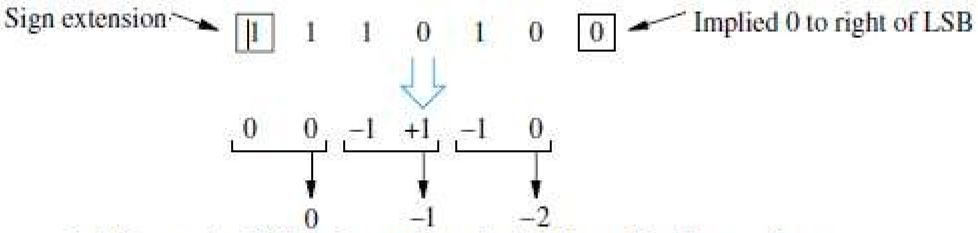
Multiplier		Version of multiplicand selected by bit *i*
Bit *i*	Bit *i* – 1	
0	0	0 × M
0	1	+ 1 × M
1	0	– 1 × M
1	1	0 × M

Figure 9.12 Booth multiplier recoding table.

FAST MULTIPLICATION

BIT-PAIR RECODING OF MULTIPLIERS

- This method
  - → derived from the booth algorithm
  - → reduces the number of summands by a factor of 2
- Group the Booth-recoded multiplier bits in pairs. (Figure 9.14 & 9.15).
- The pair (+1 -1) is equivalent to the pair (0 +1).



(a) Example of bit-pair recoding derived from Booth recoding

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	0 × M
0	0	1	+ 1 × M
0	1	0	+ 1 × M
0	1	1	+ 2 × M
1	0	0	− 2 × M
1	0	1	− 1 × M
1	1	0	− 1 × M
1	1	1	0 × M

(b) Table of multiplicand selection decisions

Figure 9.14 Multiplier bit-pair recoding.

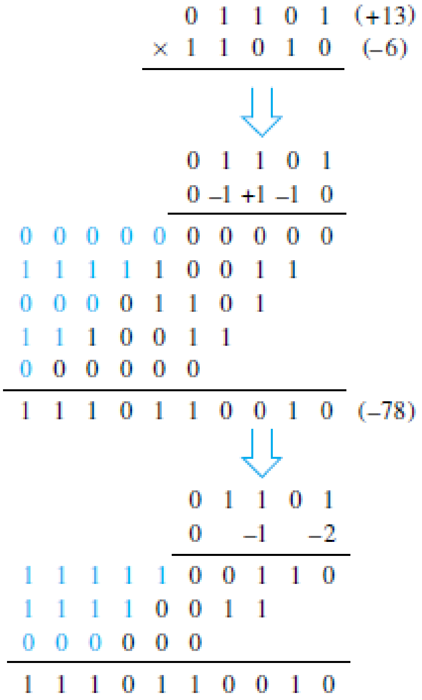
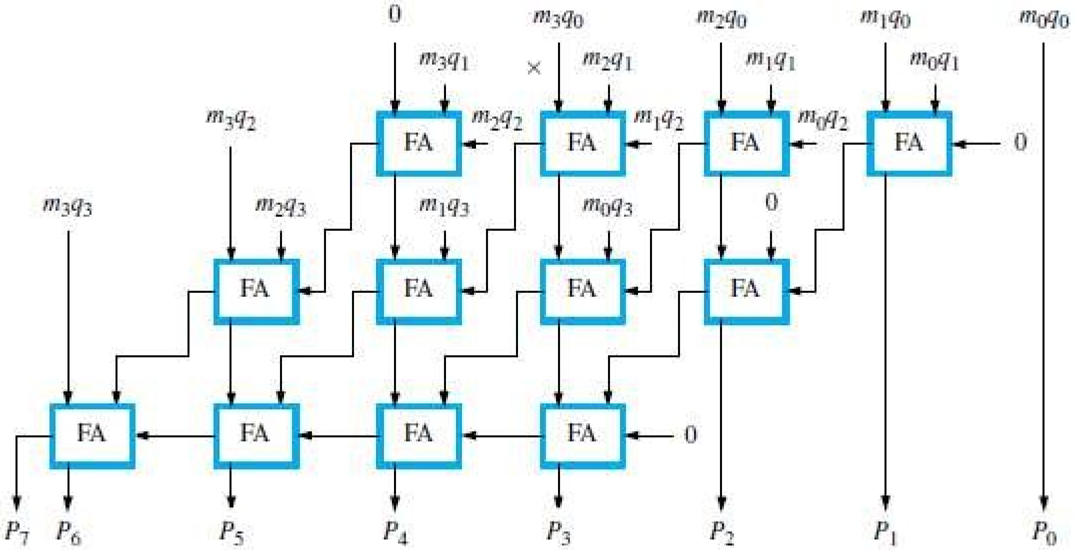


Figure 9.15 Multiplication requiring only $n/2$ summands.

CARRY-SAVE ADDITION OF SUMMANDS

- Consider the array for 4*4 multiplication. (Figure 9.16 & 9.18).
- Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions.



(b) Carry-save array

Figure 9.16 carry-save arrays for a 4 × 4 multiplier.

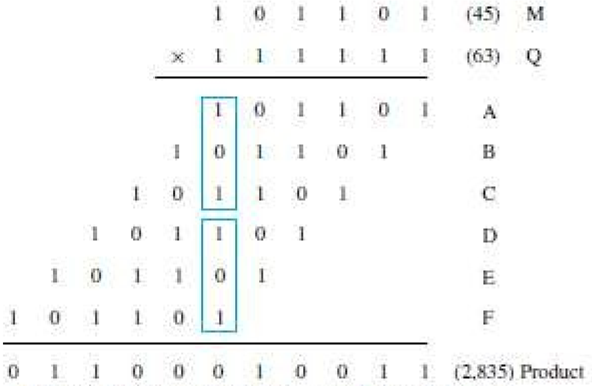
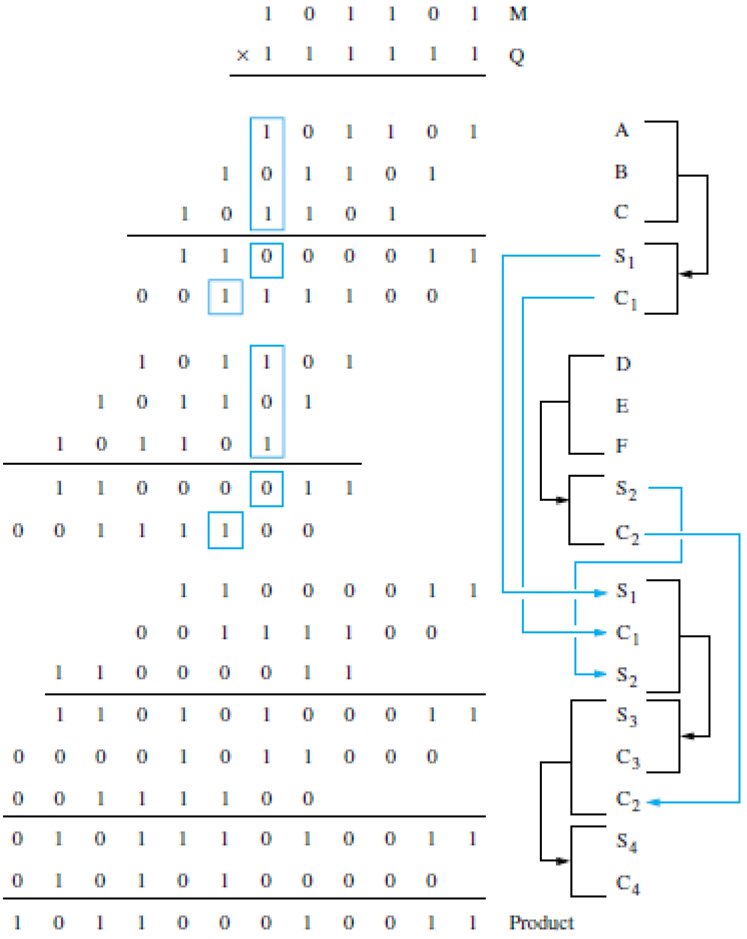


Figure 9.17 A multiplication example used to illustrate carry-save addition as shown in Figure 9.18.



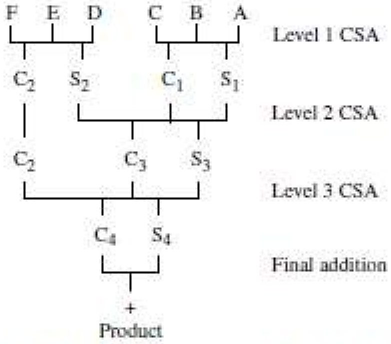


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

- The full adder is input with three partial bit products in the first row.
- Multiplication requires the addition of several summands.
- CSA speeds up the addition process.
- Consider the array for 4x4 multiplication shown in fig 9.16.
- First row consisting of just the AND gates that implement the bit products m_3q_0, m_2q_0, m_1q_0 and m_0q_0.
- The delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.
- Consider the addition of many summands in fig 9.18.
- Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
- Group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay
- Continue with this process until there are only two vectors remaining
- They can be added in a RCA or CLA to produce the desired product.
- When the number of summands is large, the time saved is proportionally much greater.
- Delay: AND gate + 2 gate/CSA level + CLA gate delay, Eg., 6 bit number require 15 gate delay, array 6x6 require 6(n-1)-1 = 29 gate Delay.
- In general, CSA takes 1.7 log_2k-1.7 levels of CSA to reduce k summands.

INTEGER DIVISION

- An n-bit positive-divisor is loaded into register M.
  An n-bit positive-dividend is loaded into register Q at the start of the operation.
  Register A is set to 0 (Figure 9.21).
- After division operation, the n-bit quotient is in register Q, and
  the remainder is in register A.

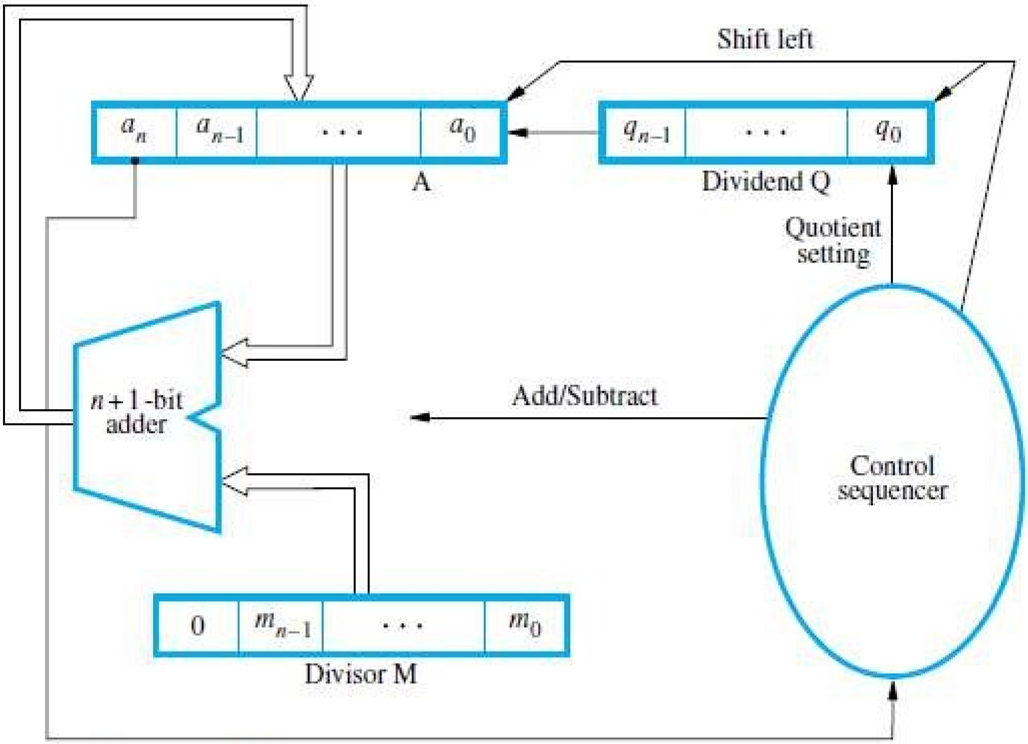


Figure 9.23 Circuit arrangement for binary division.

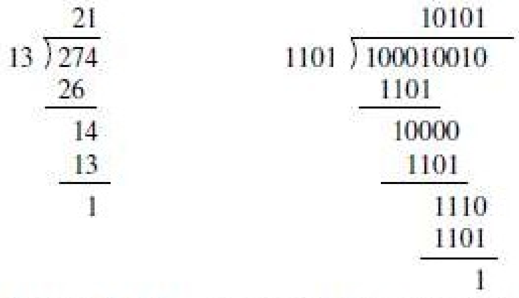


Figure 9.22 Longhand division examples.

NON-RESTORING DIVISION

- Procedure:

    Step 1: Do the following n times

    i) If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A (Figure 9.23).

    ii) Now, if the sign of A is 0, set q_0 to 1; otherwise set q_0 to 0.

    Step 2: If the sign of A is 1, add M to A (restore).

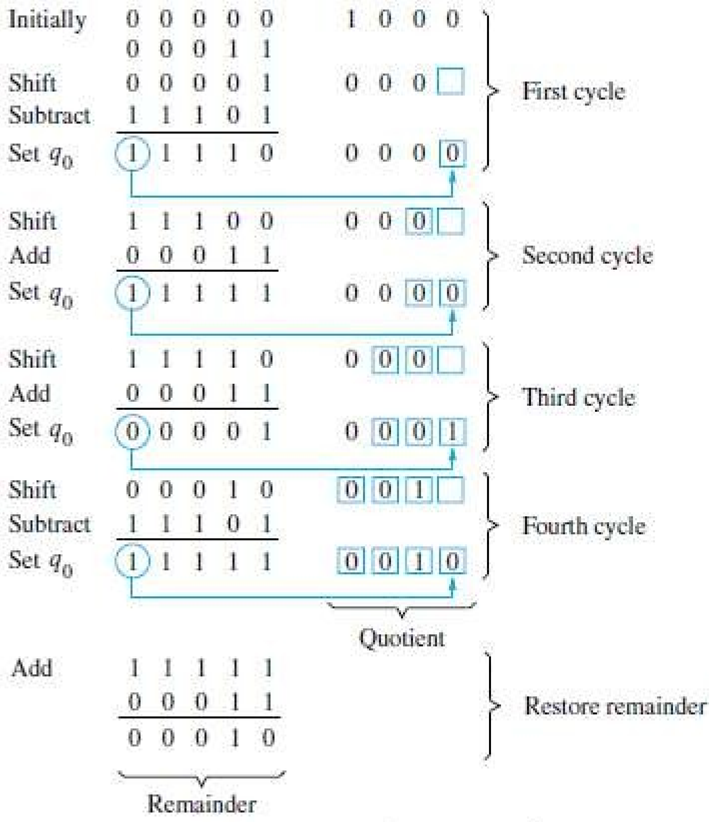


Figure 9.25 A non-restoring division example.

RESTORING DIVISION

- Procedure: Do the following n times

  1) Shift A and Q left one binary position (Figure 9.22).
  2) Subtract M from A, and place the answer back in A
  3) If the sign of A is 1, set q_0 to 0 and add M back to A(restore A).
     If the sign of A is 0, set q_0 to 1 and no restoring done.

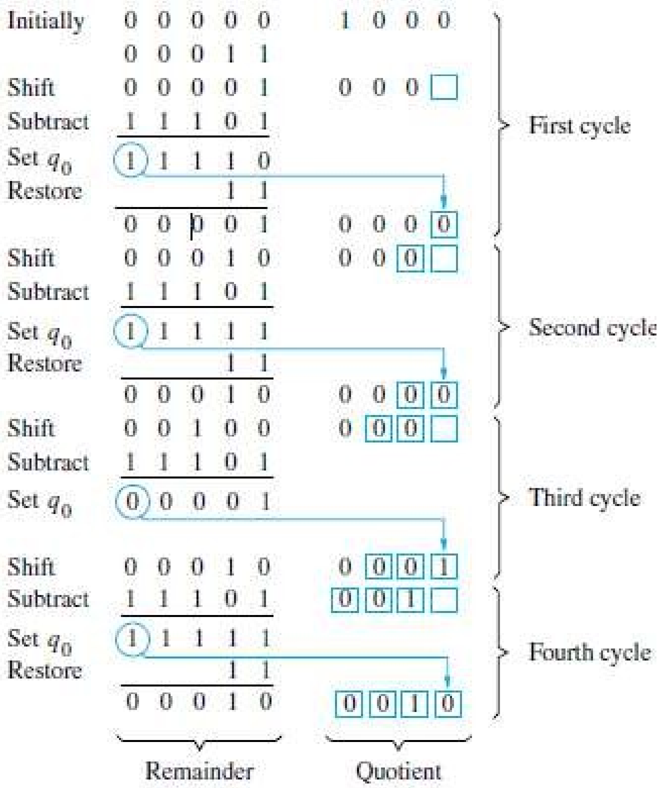


Figure 9.24 A restoring division example.

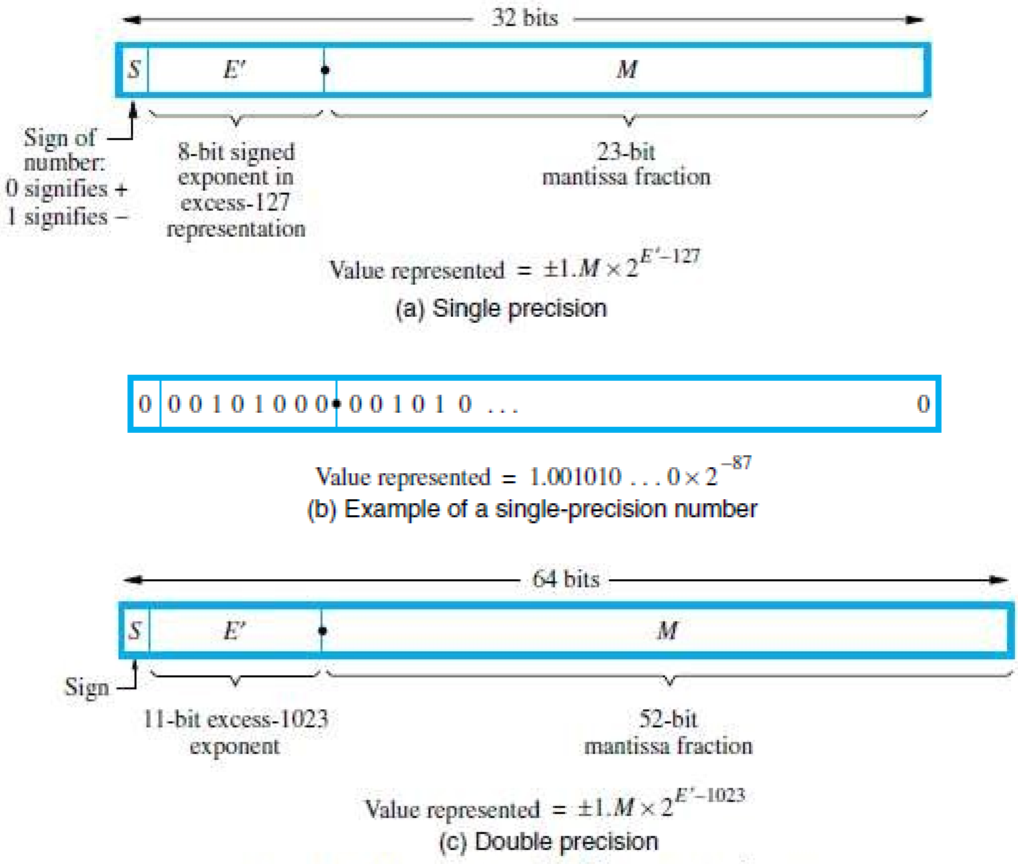
FLOATING-POINT NUMBERS & OPERATIONS

IEEE STANDARD FOR FLOATING POINT NUMBERS

- Single precision representation occupies a single 32-bit word.

  The scale factor has a range of 2^{-126} to 2^{+127}(which is approximately equal to 10^{+38}).
- The 32 bit word is divided into 3 fields: sign(1 bit), exponent(8 bits) and mantissa(23 bits).
- Signed exponent=E.

  Unsigned exponent E'=E+127. Thus, E' is in the range 0<E'<255.
- The last 23 bits represent the mantissa. Since binary normalization is used, the MSB of the mantissa is always equal to 1. (M represents fractional-part).
- The 24-bit mantissa provides a precision equivalent to about 7 decimal-digits (Figure 9.24).
- Double precision representation occupies a single 64-bit word. And E' is in the range 1<E'<2046.
- The 53-bit mantissa provides a precision equivalent to about 16 decimal-digits.


32 bits
S
E'
M
Sign of number: 0 signifies + 1 signifies −
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction
0
0 0 1 0 1 0 0 0
0 0 1 0 1 0 ...
0
64 bits
S
E'
M
Sign
11-bit excess-1023 exponent
52-bit mantissa fraction

Value represented $= \pm 1.M \times 2^{E'-127}$

(a) Single precision

Value represented $= 1.001010\ldots0 \times 2^{-87}$

(b) Example of a single-precision number

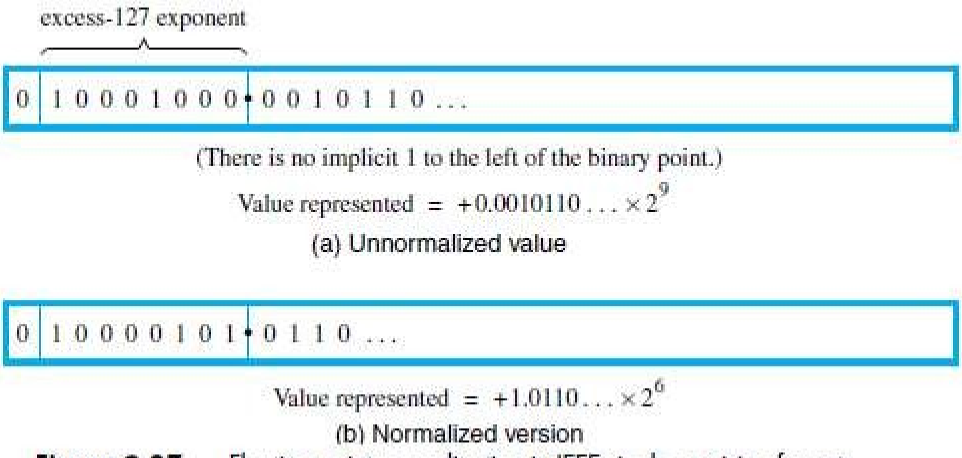
Value represented $= \pm 1.M \times 2^{E'-1023}$

(c) Double precision

Figure 9.26 IEEE standard floating-point formats.

NORMALIZATION

- When the decimal point is placed to the right of the first(non zero) significant digit, the number is said to be normalized.
- If a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent. As computations proceed, a number that does not fall in the representable range of normal numbers might be generated.
- In single precision, it requires an exponent less than -126 (underflow) or greater than +127 (overflow). Both are exceptions that need to be considered.


excess-127 exponent
0 | 1 0 0 0 1 0 0 0 • 0 0 1 0 1 1 0 . . .
(There is no implicit 1 to the left of the binary point.)
Value represented = +0.0010110 . . . × 2^9
0 | 1 0 0 0 0 1 0 1 • 0 1 1 0 . . .
Value represented = +1.0110 . . . × 2^6

(a) Unnormalized value

(b) Normalized version

Figure 9.27 Floating-point normalization in IEEE single-precision format.

SPECIAL VALUES

- The end values 0 and 255 of the excess-127 exponent E' are used to represent special values.
- When E'=0 and the mantissa fraction m is zero, the value exact 0 is represented.
- When E'=255 and M=0, the value ∞ is represented, where ∞ is the result of dividing a normal number by zero.
- when E'=0 and M!=-, denormal numbers are represented. Their value is $\pm 0.M X 2^{-126}$
- When E'=255 and M!=0, the value represented is called not a number(NaN). A NaN is the result of performing an invalied operation such as 0/0 or $\sqrt{0}$.

ARITHMETIC OPERATIONS ON FLOATING-POINT NUMBERS

Multiply Rule

1) Add the exponents & subtract 127.
2) Multiply the mantissas & determine sign of the result.
3) Normalize the resulting value if necessary.

Divide Rule

1) Subtract the exponents & add 127.
2) Divide the mantissas & determine sign of the result.
3) Normalize the resulting value if necessary.

Add/Subtract Rule

1) Choose the number with the smaller exponent & shift its mantissa right a number of steps equal to the difference in exponents(n).
2) Set exponent of the result equal to larger exponent.
3) Perform addition/subtraction on the mantissas & determine sign of the result.
4) Normalize the resulting value if necessary.

IMPLEMENTING FLOATING-POINT OPERATIONS

- First compare exponents to determine how far to shift the mantissa of the number with the smaller exponent.
- The shift-count value n
  → is determined by 8 bit subtractor &
  → is sent to SHIFTER unit.
- In step 1, sign is sent to SWAP network (Figure 9.26).
  If sign=0, then $E_A>E_B$ and mantissas M_A & M_B are sent straight through SWAP network.
  If sign=1, then $E_A<E_B$ and the mantissas are swapped before they are sent to SHIFTER.
- In step 2, 2:! MUX is used. The exponent of result E is tentatively determined as E_A if $E_A>E_B$ or
  E_B if $E_A<E_B$
- In step 3, CONTROL logic
  → determines whether mantissas are to be added or subtracted.
  → determines sign of the result.
- In step 4, result of step 3 is normalized. The number of leading zeros in M determines number of bit shifts(X) to be applied to M.

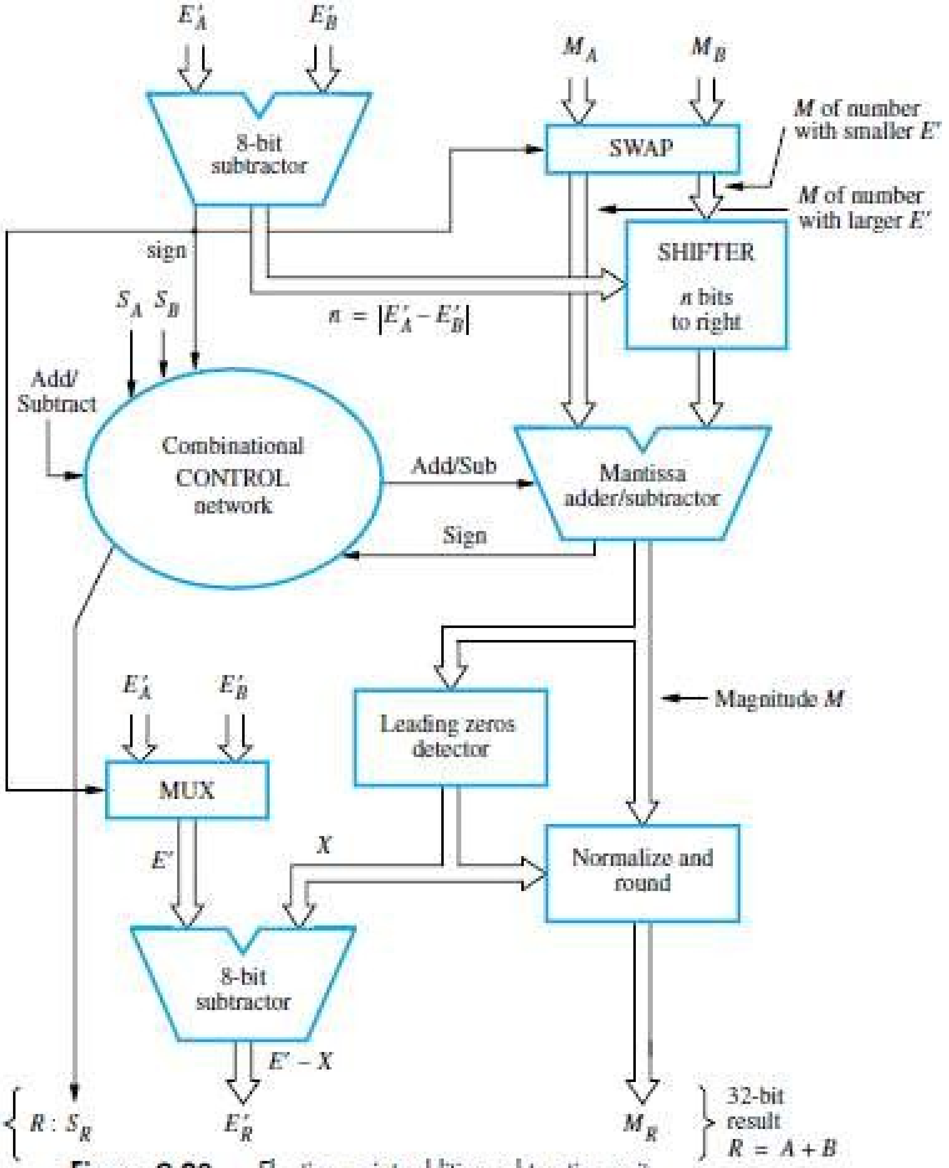


Figure 9.28 Floating-point addition-subtraction unit.

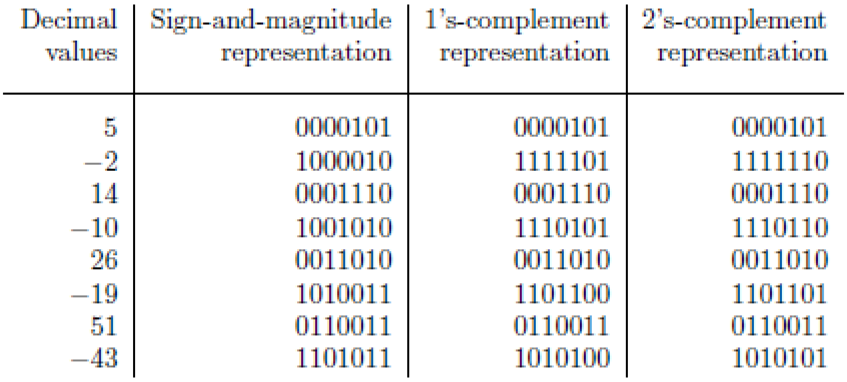
Problem 1:

Represent the decimal values 5, -2, 14, -10, 26, -19, 51 and -43 as signed 7-bit numbers in the following binary formats:

(a) sign-and-magnitude
(b) 1’s-complement
(c) 2’s-complement

Solution:

The three binary representations are given as:



Decimal values	Sign-and-magnitude representation	1’s-complement representation	2’s-complement representation
5	0000101	0000101	0000101
−2	1000010	1111101	1111110
14	0001110	0001110	0001110
−10	1001010	1110101	1110110
26	0011010	0011010	0011010
−19	1010011	1101100	1101101
51	0110011	0110011	0110011
−43	1101011	1010100	1010101

Problem 2:

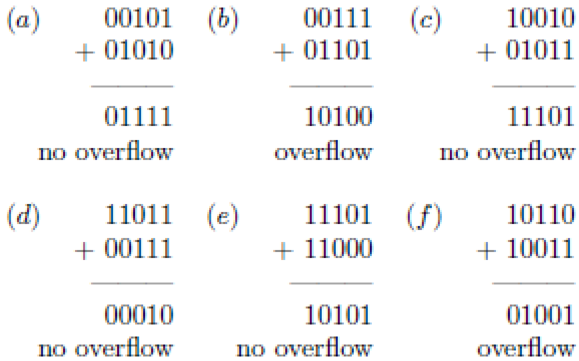
(a) Convert the following pairs of decimal numbers to 5-bit 2’s-complement numbers, then add them. State whether or not overflow occurs in each case.

a) 5 and 10   b) 7 and 13
c) −14 and 11   d) −5 and 7
e) −3 and −8

(b) Repeat Problem 1.7 for the subtract operation, where the second number of each pair is to be subtracted from the first number. State whether or not overflow occurs in each case.

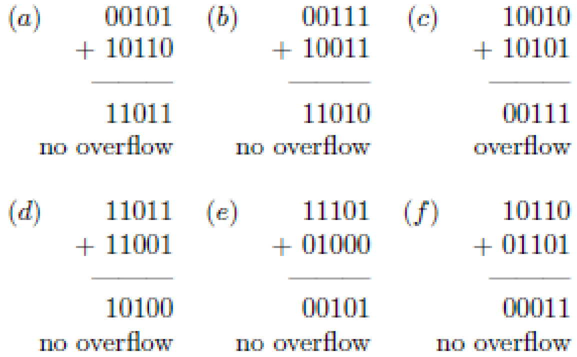
Solution:

(a)



```
(a)      00101   (b)      00111   (c)      10010
       + 01010          + 01101          + 01011
       -------          -------          -------
         01111            10100            11101
   no overflow         overflow      no overflow

(d)      11011   (e)      11101   (f)      10110
       + 00111          + 11000          + 10011
       -------          -------          -------
         00010            10101            01001
   no overflow      no overflow         overflow
```

(b) To subtract the second number, form its 2's-complement and add it to the first number.



```
(a)      00101   (b)      00111   (c)      10010
       + 10110          + 10011          + 10101
       -------          -------          -------
         11011            11010            00111
   no overflow      no overflow         overflow

(d)      11011   (e)      11101   (f)      10110
       + 11001          + 01000          + 01101
       -------          -------          -------
         10100            00101            00011
   no overflow      no overflow      no overflow
```

Problem 3:
Perform following operations on the 6-bit signed numbers using 2's complement representation system. Also indicate whether overflow has occurred.

$$\begin{array}{rrr}
010110 & 101011 & 111111 \\
\underline{+001001} & \underline{+100101} & \underline{+000111} \\
011001 & 110111 & 010101 \\
\underline{+010000} & \underline{+111001} & \underline{+101011} \\
010110 & 111110 & 100001 \\
\underline{-011111} & \underline{-100101} & \underline{-011101} \\
111111 & 000111 & 011010 \\
\underline{-000111} & \underline{-111000} & \underline{-100010}
\end{array}$$

Solution:

$$\begin{array}{rrrrrr}
010110 & (+22) & 101011 & (-21) & 111111 & (-1) \\
\underline{+\,001001} & \underline{+\,(+9)} & \underline{+\,100101} & \underline{+\,(-27)} & \underline{+\,000111} & \underline{+\,(+7)} \\
011111 & (+31) & 010000 & (-48) & 000110 & (+6) \\
 & & \text{overflow} & & &
\end{array}$$

$$\begin{array}{rrrrrr}
011001 & (+25) & 110111 & (-9) & 010101 & (+21) \\
\underline{+\,010000} & \underline{+\,(+16)} & \underline{+\,111001} & \underline{+\,(-7)} & \underline{+\,101011} & \underline{+\,(-21)} \\
101001 & (+41) & 110000 & (-16) & 000000 & (0) \\
\text{overflow} & & & & &
\end{array}$$

$$\begin{array}{rrr}
010110 & (+22) & 010110 \\
\underline{-\,011111} & \underline{-\,(+31)} & \underline{+\,100001} \\
 & (-9) & 110111 \\
\\
111110 & (-2) & 111110 \\
\underline{-\,100101} & \underline{-\,(-27)} & \underline{+\,011011} \\
 & (+25) & 011001 \\
\\
100001 & (-31) & 100001 \\
\underline{-\,011101} & \underline{-\,(+29)} & \underline{+\,100011} \\
 & (-60) & 000100 \\
 & & \text{overflow} \\
\\
111111 & (-1) & 111111 \\
\underline{-\,000111} & \underline{-\,(+7)} & \underline{+\,111001} \\
 & (-8) & 111000 \\
\\
000111 & (+7) & 000111 \\
\underline{-\,111000} & \underline{-\,(-8)} & \underline{+\,001000} \\
 & (+15) & 001111 \\
\\
011010 & (+26) & 011010 \\
\underline{-\,100010} & \underline{-\,(-30)} & \underline{+\,011110} \\
 & (+56) & 111000 \\
 & & \text{overflow}
\end{array}$$

Problem 4:

Perform signed multiplication of following 2's complement numbers using Booth's algorithm.

(a) A=010111 and B=110110 (b) A=110011 and B=101100
(c) A=110101 and B=011011 (d) A=001111 and B=001111
(e) A=10100 and B=10101 (f) A=01110 and B=11000

Solution:

```
  010111      +23                              0  1  0  1  1  1
 ×110110     × -10                          ×  0 -1 +1  0 -1  0
 -------     -----                       -------------------------------------
              -230                                            0
                                   1  1  1  1  1  1  0  1  0  0  1
                  sign                                     0
                  extension        0  0  0  0  1  0  1  1  1
                                   1₁ 1₁ 1₁ 0₂ 1₁ 0₁ 0₁ 1
                                   -------------------------------------
                                   1  1  1  1  0  0  0  1  1  0  1  0
```

```
  110011      -13                              1  1  0  0  1  1
 ×101100     × -20                          × -1 +1  0 -1  0  0
 -------     -----                       -------------------------------------
               260                                            0
                                                           0
                                   0  0  0  0  0  0  1  1  0  1
                  sign                                  0
                  extension        1  1  1  1  0  0  1  1
                                   0₁ 0₁ 0₁ 1₁ 1₁ 0₂ 1₁
                                   -------------------------------------
                                   0  0  0  1  0  0  0  0  0  1  0  0
```

```
  110101      -11                              1  1  0  1  0  1
 ×011011     ×  27                          × +1  0 -1 +1  0 -1
 -------     -----                       -------------------------------------
              -297                 0  0  0  0  0  0  0  0  1  0  1  1
                                                           0
                  sign             1  1  1  1  1  1  0  1  0  1
                  extension        0  0  0  0  0  1  0  1  1
                                                     0
                                   1₁ 1₁ 1₁ 0₁ 1₁ 0₁ 1₁  ₁
                                   -------------------------------------
                                   1  1  1  0  1  1  0  1  0  1  1  1
```

```
  001111       15                              0  0  1  1  1  1
 ×001111     × 15                           ×  0 +1  0  0  0 -1
 -------     -----                       -------------------------------------
              225                  1  1  1  1  1  1  1  1  0  0  0  1
                                   0  0  0  0  1  1  1  1
                                   -------------------------------------
                                   0  0  0  0  1  1  1  0  0  0  0  1
```

```
   1 0 1 0 0 (-12)                              1  0  1  0  0
 X 1 0 1 0 1 (-11)        →                    -1  1 -1  1 -1   (recoded multiplier)
 -----------------                  ---------------------------
                                    0  0  0  0  0  1  1  0  0
                                    1  1  1  1  0  1  0  0
                                    0  0  0  1  1  0  0
                                    1  1  0  1  0  0
                                    0  1  1  0  0
                                    ---------------------------
                                    0  1  0  0  0  0  1  0  0   (+132)
```

```
   0 1 1 1 0 (+14)                                 0  1  1  1  0
 X 1 1 0 0 0 (-8)         →                        0 -1  0  0  0   (recoded multiplier)
 -----------------                  ------------------------------
                                    0  0  0  0  0  0  0  0  0  0
                                    0  0  0  0  0  0  0  0  0
                                    0  0  0  0  0  0  0  0
                                    1  1  1  0  0  1  0
                                    0  0  0  0  0  0
                                    ------------------------------
                                    1  1  1  0  0  1  0  0  0  0   (-112)
```

Problem 5:

Perform signed multiplication of following 2’s complement numbers using bit-pair recoding method.

(a) A=010111 and B=110110 (b) A=110011 and B=101100

(c) A=110101 and B=011011 (d) A=001111 and B=001111

Solution:

```
  010111                           0 1 0 1 1 1
×110110                            -1  +2  -2
                      1 1 1 1 1 1 0 1 0 0 1 0
                      0 0 0 0 1 0 1 1 1 0
                      1₁1₁1₁0₂1₁0₁0₁1
                      1 1 1 1 0 0 0 1 1 0 1 0
```

```
  110011                           1 1 0 0 1 1
×101100                            -1  -1   0
                                            0
                      0 0 0 0 0 0 1 1 0 1
                      0 0 0 0₁1₁1₁0₁1
                      0 0 0 1 0 0 0 0 0 1 0 0
```

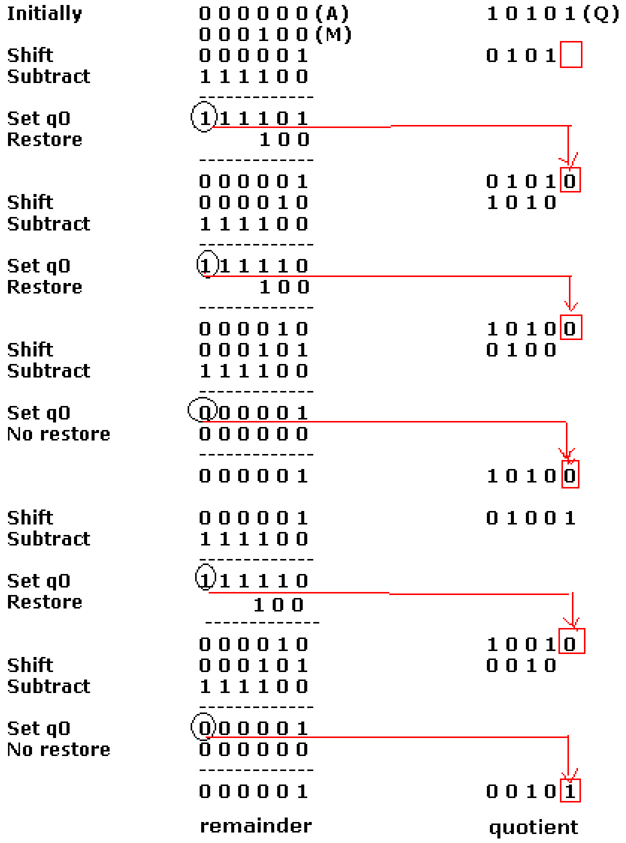
```
  110101                           1 1 0 1 0 1
×011011                            +2  -1  -1
                      0 0 0 0 0 0 0 0 1 0 1 1
                      0 0 0 0 0 0 1 0 1 1
                      1 1 1 0 1 0₁1   ₁
                      1 1 1 0 1 1 0 1 0 1 1 1
```

```
  001111                           0 0 1 1 1 1
×001111                            +1      -1
                      1 1 1 1 1 1 1 1 0 0 0 1
                      0 0 0 0 1 1 1 1
                      0 0 0 0 1 1 1 0 0 0 0 1
```

Problem 6:

Given A=10101 and B=00100, perform A/B using restoring division algorithm.

Solution:

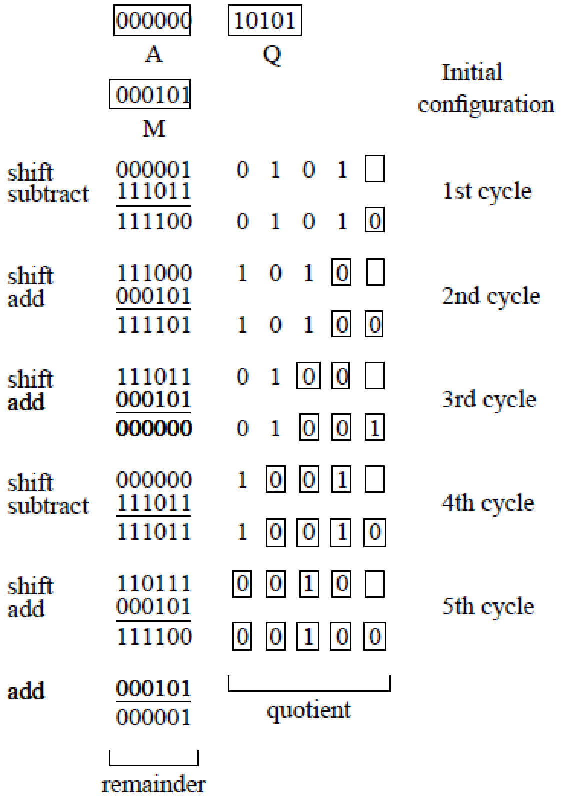


```
Initially      0 0 0 0 0 0 (A)      1 0 1 0 1 (Q)
               0 0 0 1 0 0 (M)
Shift          0 0 0 0 0 1          0 1 0 1 □
Subtract       1 1 1 1 0 0
               ------------
Set q0         1 1 1 1 0 1
Restore              1 0 0
               ------------
               0 0 0 0 0 1          0 1 0 1 0
Shift          0 0 0 0 1 0          1 0 1 0
Subtract       1 1 1 1 0 0
               ------------
Set q0         1 1 1 1 1 0
Restore              1 0 0
               ------------
               0 0 0 0 1 0          1 0 1 0 0
Shift          0 0 0 1 0 1          0 1 0 0
Subtract       1 1 1 1 0 0
               ------------
Set q0         0 0 0 0 0 1
No restore     0 0 0 0 0 0
               ------------
               0 0 0 0 0 1          1 0 1 0 0

Shift          0 0 0 0 0 1          0 1 0 0 1
Subtract       1 1 1 1 0 0
               ------------
Set q0         1 1 1 1 1 0
Restore              1 0 0
               ------------
               0 0 0 0 1 0          1 0 0 1 0
Shift          0 0 0 1 0 1          0 0 1 0
Subtract       1 1 1 1 0 0
               ------------
Set q0         0 0 0 0 0 1
No restore     0 0 0 0 0 0
               ------------
               0 0 0 0 0 1          0 0 1 0 1
               remainder            quotient
```

Problem 7:

Given A=10101 and B=00101, perform A/B using non-restoring division algorithm.

Solution:



```
           000000    10101
             A         Q
                                  Initial
           000101                 configuration
             M

shift      000001    0 1 0 1 □
subtract   111011                 1st cycle
           111100    0 1 0 1 0

shift      111000    1 0 1 0 □
add        000101                 2nd cycle
           111101    1 0 1 0 0

shift      111011    0 1 0 0 □
add        000101                 3rd cycle
           000000    0 1 0 0 1

shift      000000    1 0 0 1 □
subtract   111011                 4th cycle
           111011    1 0 0 1 0

shift      110111    0 0 1 0 □
add        000101                 5th cycle
           111100    0 0 1 0 0

add        000101    quotient
           000001
          remainder
```

Problem 8:

Represent 1259.12510 in single precision and double precision formats

Solution:

Step 1: Convert decimal number to binary format

$1259_{(10)} = 10011101011_{(2)}$

Fractional Part

$0.125_{(10)} = 0.001$

Binary number = 10011101011+0.001

= 10011101011.001

Step 2: Normalize the number

$10011101011.001 = 1.0011101011001 \times 2^{10}$

Step 3: Single precision format:

For a given number S=0, E=10 and M=0011101011001

Bias for single precision format is = 127

$E' = E+127 = 10+127 = 137_{(10)}$

$= 10001001_{(2)}$

Number in single precision format is given as

$$\underbrace{0}_{\text{Sign}} \quad \underbrace{10001001}_{\text{Exponent}} \quad \underbrace{0011101011001\ldots0}_{\text{Mantissa(23 bit)}}$$

Step 4: Double precision format:

For a given number S=0, E=10 and M=0011101011001

Bias for double precision format is = 1023

$E' = E+1023 = 10+1023 = 1033_{(10)}$

$= 10000001001_{(2)}$

Number in double precision format is given as

$$\underbrace{0}_{\text{Sign}} \quad \underbrace{10001001}_{\text{Exponent}} \quad \underbrace{0011101011001\ldots0}_{\text{Mantissa(23 bit)}}$$

MODULE 5: BASIC PROCESSING UNIT

SOME FUNDAMENTAL CONCEPTS

• To execute an instruction, processor has to perform following 3 steps:

1) Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation is written as:
   IR← [[PC]]
2) Increment PC by 4.
   PC← [PC] +4
3) Carry out the actions specified by instruction (in the IR).

• The first 2 steps are referred to as **Fetch Phase**.
Step 3 is referred to as **Execution Phase**.

• The operation specified by an instruction can be carried out by performing one or more of the following actions:

1) Read the contents of a given memory-location and load them into a register.
2) Read data from one or more registers.
3) Perform an arithmetic or logic operation and place the result into a register.
4) Store data from a register into a given memory-location.

• The hardware-components needed to perform these actions are shown in Figure 5.1.

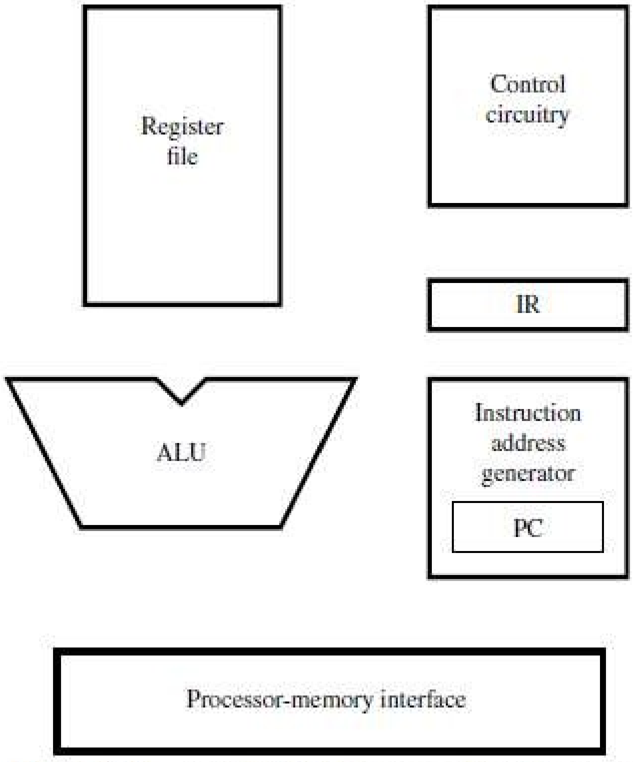


Figure 5.1 Main hardware components of a processor.

SINGLE BUS ORGANIZATION

- ALU and all the registers are interconnected via a **Single Common Bus** (Figure 7.1).
- Data & address lines of the external memory-bus is connected to the internal processor-bus via MDR & MAR respectively. (MDR→ Memory Data Register, MAR → Memory Address Register).
- **MDR** has 2 inputs and 2 outputs. Data may be loaded
  → into MDR either from memory-bus (external) or
  → from processor-bus (internal).
- **MAR**"s input is connected to internal-bus;
  MAR"s output is connected to external-bus.
- **Instruction Decoder & Control Unit** is responsible for
  → issuing the control-signals to all the units inside the processor.
  → implementing the actions specified by the instruction (loaded in the IR).
- Register R0 through R(n-1) are the **Processor Registers**.
  The programmer can access these registers for general-purpose use.
- Only processor can access 3 registers **Y**, **Z** & **Temp** for temporary storage during program-execution.
  The programmer cannot access these 3 registers.
- In **ALU**, 1) „A" input gets the operand from the output of the multiplexer (MUX).
  2) „B" input gets the operand directly from the processor-bus.
- There are 2 options provided for „A" input of the ALU.
- MUX is used to select one of the 2 inputs.
- **MUX** selects either
  → output of Y or
  → constant-value 4(which is used to increment PC content).

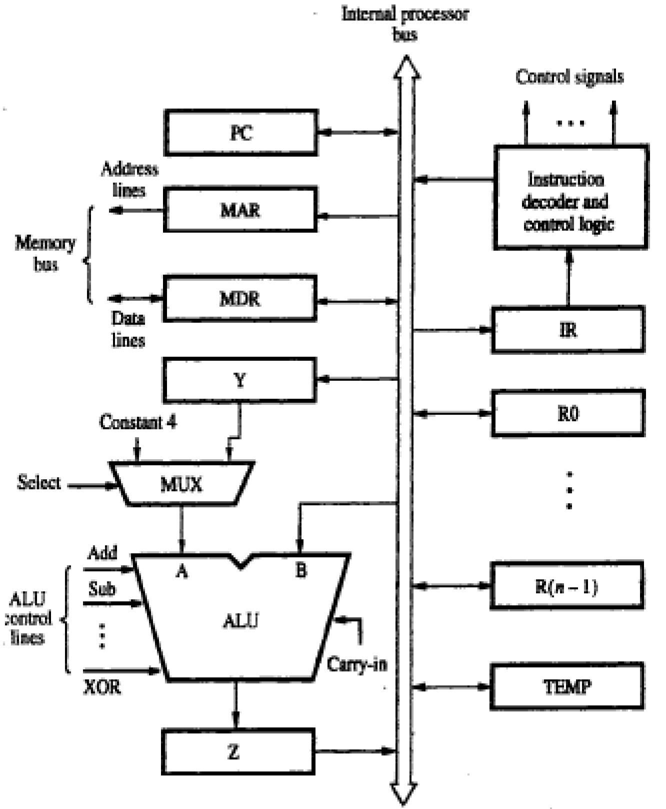


Figure 7.1 Single-bus organization of the datapath inside a processor.

- An instruction is executed by performing one or more of the following operations:
  1) Transfer a word of data from one register to another or to the ALU.
  2) Perform arithmetic or a logic operation and store the result in a register.
  3) Fetch the contents of a given memory-location and load them into a register.
  4) Store a word of data from a register into a given memory-location.
- **Disadvantage:** Only one data-word can be transferred over the bus in a clock cycle.
  Solution: Provide multiple internal-paths. Multiple paths allow several data-transfers to take place in parallel.

REGISTER TRANSFERS

- Instruction execution involves a sequence of steps in which data are transferred from one register to another.
- For each register, two control-signals are used: Ri_{in} & Ri_{out}. These are called **Gating Signals.**
- Ri_{in}=1 → data on bus is loaded into Ri. Ri_{out}=1

  → content of Ri is placed on bus.

  Ri_{out}=0, → bus can be used for transferring data from other registers.
- For example, *Move R1, R2;* This transfers the contents of register R1 to register R2. This can be accomplished as follows:

  1) Enable the output of registers R1 by setting $R1_{out}$ to 1 (Figure 7.2).

     This places the contents of R1 on processor-bus.

  2) Enable the input of register R2 by setting $R2_{out}$ to 1.

     This loads data from processor-bus into register R4.
- All operations and data transfers within the processor take place within time-periods defined by the **processor-clock**.
- The control-signals that govern a particular transfer are asserted at the start of the clock cycle.

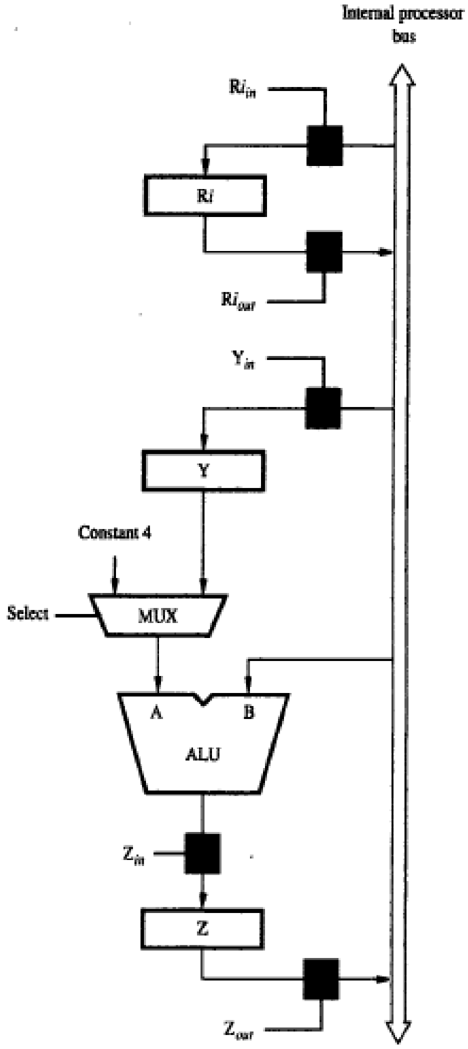


Figure 7.2 Input and output gating for the registers in Figure 7.1.

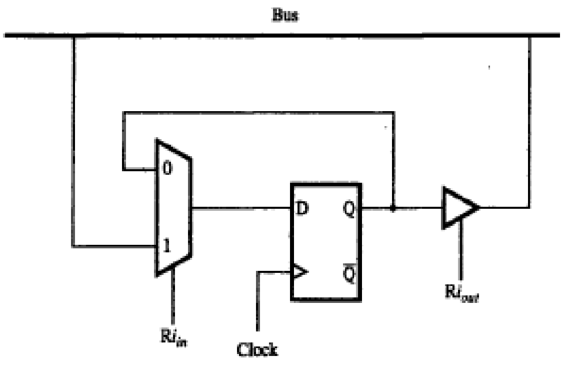


Figure 7.3 Input and output gating for one register bit.

Input & Output Gating for one Register Bit

- A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop.
- Ri_{in}=1 → mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock.

  Ri_{in}=0 → mux feeds back the value currently stored in flip-flop (Figure 7.3).
- Q output of flip-flop is connected to bus via a tri-state gate.

  Ri_{out}=0 → gate's output is in the high-impedance state.

  Ri_{out}=1 → the gate drives the bus to 0 or 1, depending on the value of Q.

PERFORMING AN ARITHMETIC OR LOGIC OPERATION

- The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs.
- One of the operands is output of MUX;
  And, the other operand is obtained directly from processor-bus.
- The result (produced by the ALU) is stored temporarily in register Z.
- The sequence of operations for [R3]🡐[R1]+[R2] is as follows:
  1) $R1_{out}$, Y_{in}
  2) $R2_{out}$, SelectY, Add, Z_{in}
  3) Z_{out}, $R3_{in}$
- Instruction execution proceeds as follows:
  Step 1 --> Contents from register R1 are loaded into register Y.
  Step2 --> Contents from Y and from register R2 are applied to the A and B inputs of ALU;
  Addition is performed &
  Result is stored in the Z register.
  Step 3 --> The contents of Z register is stored in the R3 register.
- The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

CONTROL-SIGNALS OF MDR

- The MDR register has 4 control-signals (Figure 7.4):
  1) MDR_{in} & MDR_{out} control the connection to the internal processor data bus &
  2) MDR_{inE} & MDR_{outE} control the connection to the memory Data bus.
- MAR register has 2 control-signals.
  1) MAR_{in} controls the connection to the internal processor address bus &
  2) MAR_{out} controls the connection to the memory address bus.

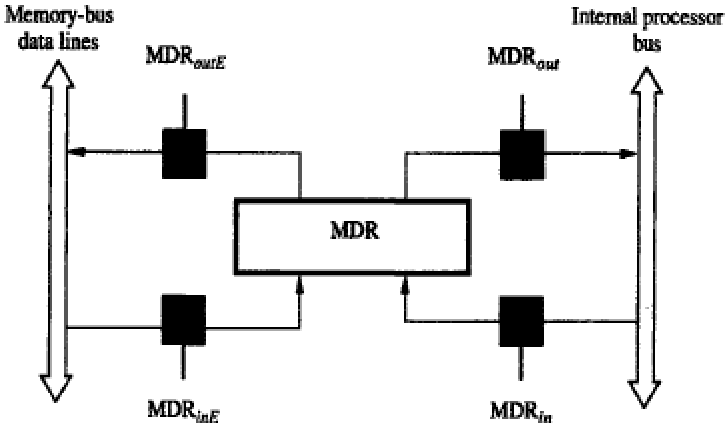


Figure 7.4 Connection and control signals for register MDR.

FETCHING A WORD FROM MEMORY

• To fetch instruction/data from memory, processor transfers required address to MAR.
At the same time, processor issues Read signal on control-lines of memory-bus.

• When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers.

• The response time of each memory access varies (based on cache miss, memory-mapped I/O). To accommodate this, MFC is used. (MFC → Memory Function Completed).

• MFC is a signal sent from addressed-device to the processor. MFC informs the processor that the requested operation has been completed by addressed-device.

• Consider the instruction Move (R1),R2. The sequence of steps is (Figure 7.5):

1) $R1_{out}$, MAR_{in}, Read ;desired address is loaded into MAR & Read command is issued.
2) MDR_{inE}, WMFC ;load MDR from memory-bus & Wait for MFC response from memory.
3) MDR_{out}, $R2_{in}$;load R2 from MDR.
where WMFC=control-signal that causes processor's control.
circuitry to wait for arrival of MFC signal.

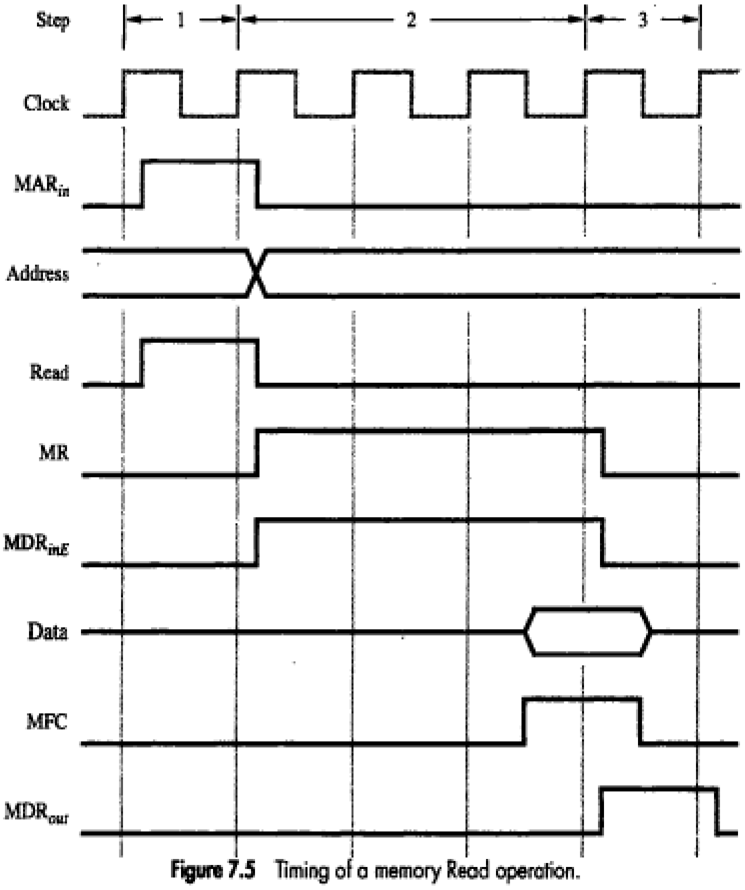


Figure 7.5 Timing of a memory Read operation.

Storing a Word in Memory

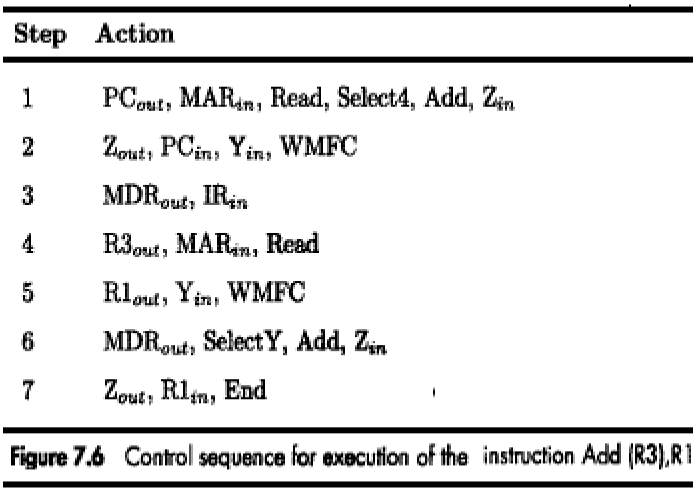
• Consider the instruction *Move R2,(R1).* This requires the following sequence:

1) $R1_{out}$, MAR_{in} ;desired address is loaded into MAR.
2) $R2_{out}$, MDR_{in}, Write ;data to be written are loaded into MDR & Write command is issued.
3) MDR_{outE}, WMFC ;load data into memory-location pointed by R1 from MDR.

EXECUTION OF A COMPLETE INSTRUCTION

• Consider the instruction *Add (R3),R1* which adds the contents of a memory-location pointed by R3 to register R1. Executing this instruction requires the following actions:

1) Fetch the instruction.
2) Fetch the first operand.
3) Perform the addition &
4) Load the result into R1.



Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1

• Instruction execution proceeds as follows:

Step1--> The instruction-fetch operation is initiated by
→ loading contents of PC into MAR &
→ sending a Read request to memory.
The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC"s content), and the result is stored in Z.

Step2--> Updated value in Z is moved to PC. This completes the PC increment operation and PC will now point to next instruction.

Step3--> Fetched instruction is moved into MDR and then to IR.
The step 1 through 3 constitutes the **Fetch Phase**.
At the beginning of step 4, the instruction decoder interprets the contents of the IR. This enables the control circuitry to activate the control-signals for steps 4 through 7.
The step 4 through 7 constitutes the **Execution Phase**.

Step4--> Contents of R3 are loaded into MAR & a memory read signal is issued.

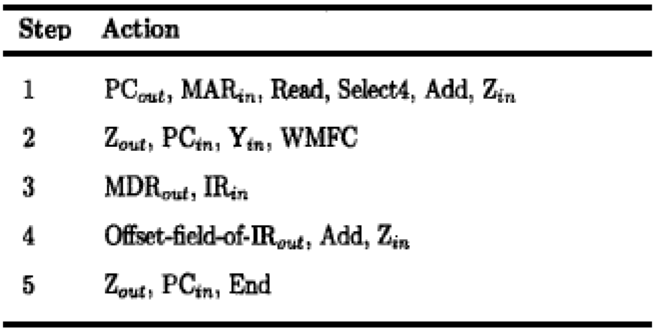
Step5--> Contents of R1 are transferred to Y to prepare for addition.

Step6--> When Read operation is completed, memory-operand is available in MDR, and the addition is performed.

Step7--> Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

BRANCHING INSTRUCTIONS

- Control sequence for an **unconditional branch instruction** is as follows:



Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7 Control sequence for an unconditional Branch instruction.

- Instruction execution proceeds as follows:

  Step 1-3--> The processing starts & the fetch phase ends in step3.

  Step 4--> The offset-value is extracted from IR by instruction-decoding circuit.

  Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.

  Step 5--> the result, which is the branch-address, is loaded into the PC.

- The branch instruction loads the branch target address in PC so that PC will fetch the next instruction from the branch target address.
- The branch target address is usually obtained by adding the offset in the contents of PC.
- The offset X is usually the difference between the branch target-address and the address immediately following the branch instruction.
- In case of **conditional branch**,

  we have to check the status of the condition-codes before loading a new value into the PC.

  e.g.: Offset-field-of-IR_{out}, Add, $Z_{in,}$ If N=0 then End

  If N=0, processor returns to step 1 immediately after step 4.

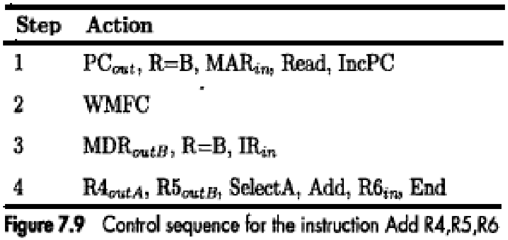
  If N=1, step 5 is performed to load a new value into PC.

MULTIPLE BUS ORGANIZATION

- **Disadvantage of Single-bus organization:** Only one data-word can be transferred over the bus in a clock cycle. This increases the steps required to complete the execution of the instruction

Solution: To reduce the number of steps, most processors provide multiple internal-paths. Multiple paths enable several transfers to take place in parallel.

- As shown in fig 7.8, three buses can be used to connect registers and the ALU of the processor.
- All general-purpose registers are grouped into a single block called the **Register File**.
- Register-file has 3 ports:
  1) Two output-ports allow the contents of 2 different registers to be simultaneously placed on buses A & B.
  2) Third input-port allows data on bus C to be loaded into a third register during the same clock-cycle.
- Buses A and B are used to transfer source-operands to A & B inputs of ALU.
- The result is transferred to destination over bus C.
- **Incrementer Unit** is used to increment PC by 4.



Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Figure 7.9 Control sequence for the instruction Add R4,R5,R6

- Instruction execution proceeds as follows:

Step 1--> Contents of PC are

→ passed through ALU using R=B control-signal &

→ loaded into MAR to start memory Read operation. At the same time, PC is incremented by 4.

Step2--> Processor waits for MFC signal from memory.

Step3--> Processor loads requested-data into MDR, and then transfers them to IR.

Step4--> The instruction is decoded and add operation takes place in a single step.

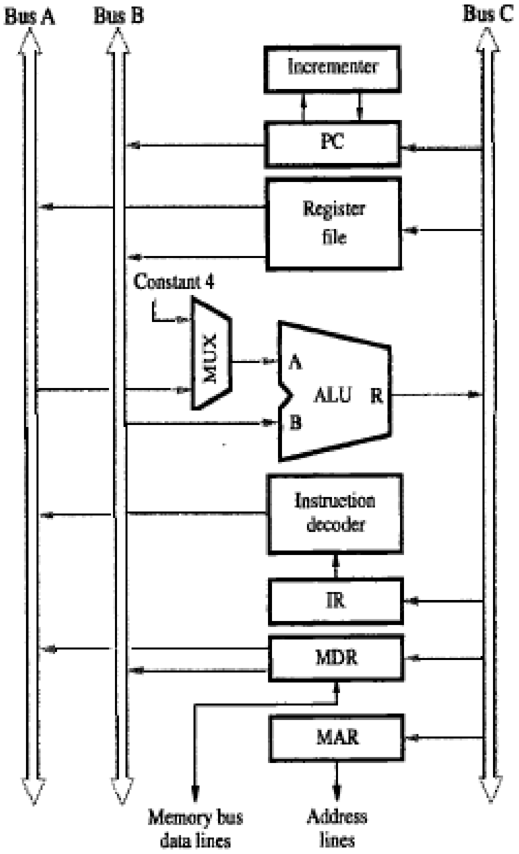


Figure 7.8 Three-bus organization of the datapath.

COMPLETE PROCESSOR

- This has separate processing-units to deal with integer data and floating-point data.
  Integer Unit → To process integer data. (Figure 7.14).
  Floating Unit → To process floating –point data.
- **Data-Cache** is inserted between these processing-units & main-memory.
  The integer and floating unit gets data from data cache.
- **Instruction-Unit** fetches instructions
  → from an instruction-cache or
  → from main-memory when desired instructions are not already in cache.
- Processor is connected to system-bus &
  hence to the rest of the computer by means of a **Bus Interface.**
- Using separate caches for instructions & data is common practice in many processors today.
- A processor may include several units of each type to increase the potential for concurrent operations.
- The 80486 processor has 8-kbytes single cache for both instruction and data.
  Whereas the Pentium processor has two separate 8 kbytes caches for instruction and data.

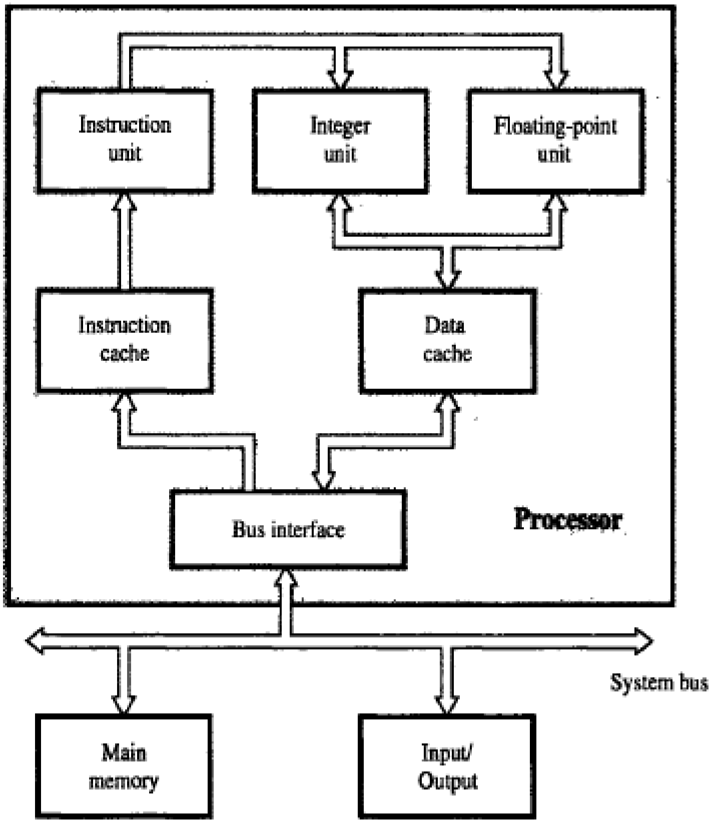


Figure 7.14 Block diagram of a complete processor.

Note:

To execute instructions, the processor must have some means of generating the control-signals. There are two approaches for this purpose:

1) Hardwired control and 2) Microprogrammed control.

HARDWIRED CONTROL

- Hardwired control is a method of control unit design (Figure 7.11).
- The control-signals are generated by using logic circuits such as gates, flip-flops, decoders etc.
- **Decoder/Encoder Block** is a combinational-circuit that generates required control-outputs depending on state of all its inputs.
- **Instruction Decoder**
  - ➢ It decodes the instruction loaded in the IR.
  - ➢ If IR is an 8 bit register, then instruction decoder generates 2^8(256 lines); one for each instruction.
  - ➢ It consists of a separate output-lines INS_1 through INS_m for each machine instruction.
  - ➢ According to code in the IR, one of the output-lines INS_1 through INS_m is set to 1, and all other lines are set to 0.
- **Step-Decoder** provides a separate signal line for each step in the control sequence.
- **Encoder**
  - ➢ It gets the input from instruction decoder, step decoder, external inputs and condition codes.
  - ➢ It uses all these inputs to generate individual control-signals: Y_{in}, PC_{out}, Add, End and so on.
  - ➢ For example (Figure 7.12), $Z_{in}=T_1+T_6.ADD+T_4.BR$

    ;This signal is asserted during time-slot T_1 for all instructions.
    during T_6 for an Add instruction.
    during T_4 for unconditional branch instruction
- When **RUN**=1, counter is incremented by 1 at the end of every clock cycle.
  When RUN=0, counter stops counting.
- After execution of each instruction, **end** signal is generated. End signal resets step counter.
- Sequence of operations carried out by this machine is determined by wiring of logic circuits, hence the name "**hardwired**".
- **Advantage**: Can operate at high speed.
- **Disadvantages:**
  1) Since no. of instructions/control-lines is often in hundreds, the complexity of control unit is very high.
  2) It is costly and difficult to design.
  3) The control unit is inflexible because it is difficult to change the design.

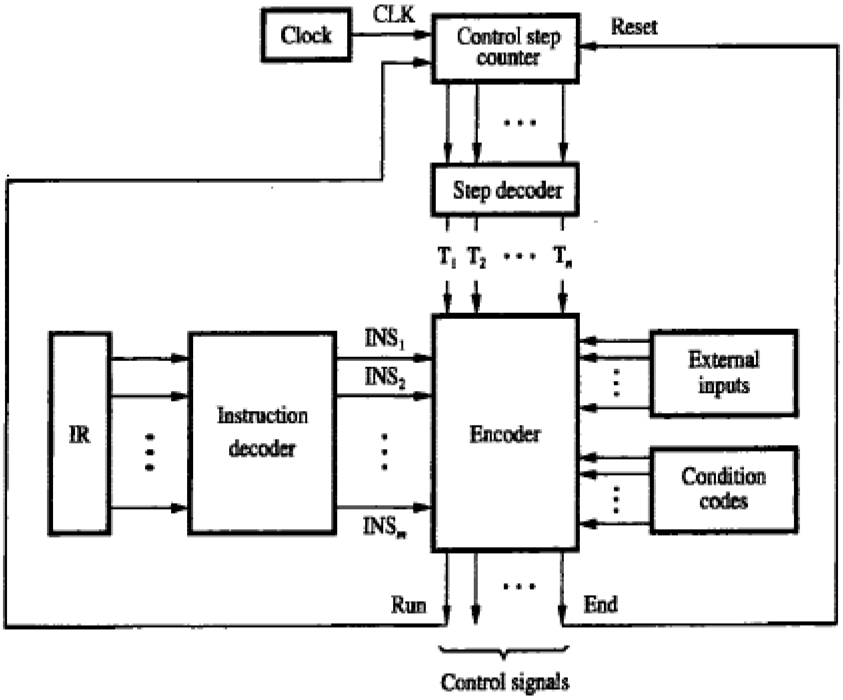


Figure 7.11 Separation of the decoding and encoding functions.

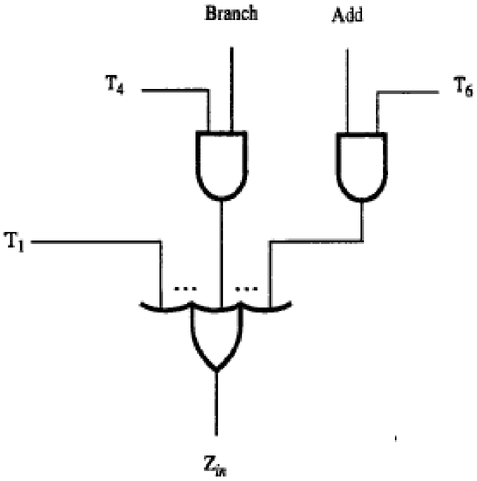
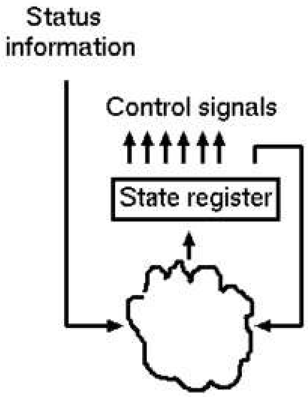
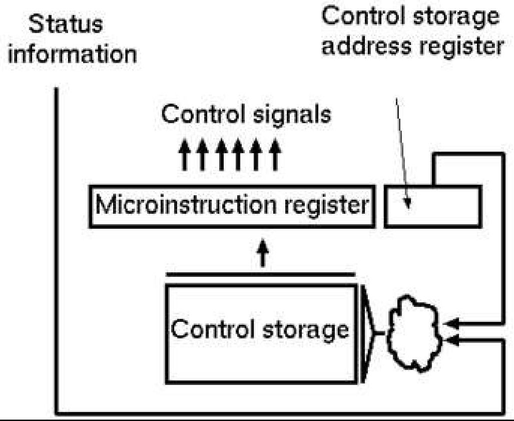


Figure 7.12 Generation of the Z_{in} control signal

HARDWIRED CONTROL VS MICROPROGRAMMED CONTROL

Attribute	Hardwired Control	Microprogrammed Control
Definition	Hardwired control is a control mechanism to generate control-signals by using gates, flip-flops, decoders, and other digital circuits.	Micro programmed control is a control mechanism to generate control-signals by using a memory called control store (CS), which contains the control-signals.
Speed	Fast	Slow
Control functions	Implemented in hardware.	Implemented in software.
Flexibility	Not flexible to accommodate new system specifications or new instructions.	More flexible, to accommodate new system specification or new instructions redesign is required.
Ability to handle large or complex instruction sets	Difficult.	Easier.
Ability to support operating systems & diagnostic features	Very difficult.	Easy.
Design process	Complicated.	Orderly and systematic.
Applications	Mostly RISC microprocessors.	Mainframes, some microprocessors.
Instructionset size	Usually under 100 instructions.	Usually over 100 instructions.
ROM size	-	2K to 10K by 20-400 bit microinstructions.
Chip area efficiency	Uses least area.	Uses more area.
Diagram		 Status information Control storage address register Control signals Microinstruction register Control storage

MICROPROGRAMMED CONTROL

- Microprogramming is a method of control unit design (Figure 7.16).
- Control-signals are generated by a program similar to machine language programs.
- **Control Word(CW*)*** is a word whose individual bits represent various control-signals (like Add, PC_{in}).
- Each of the control-steps in control sequence of an instruction defines a unique combination of 1s & 0s in CW.
- Individual control-words in microroutine are referred to as **microinstructions** (Figure 7.15).
- A sequence of CWs corresponding to control-sequence of a machine instruction constitutes the **microroutine.**
- The microroutines for all instructions in the instruction-set of a computer are stored in a special memory called the **Control Store (CS)***.*
- Control-unit generates control-signals for any instruction by sequentially reading CWs of corresponding microroutine from CS.
- **µPC** is used to read CWs sequentially from CS. (µPC→ Microprogram Counter).
- Every time new instruction is loaded into IR, o/p of **Starting Address Generator** is loaded into µPC.
- Then, µPC is automatically incremented by clock;

  causing successive microinstructions to be read from CS.

  Hence, control-signals are delivered to various parts of processor in correct sequence.

Advantages

- It simplifies the design of c ror prone implement.
- Control functions are imple
- The design process is orde
- More flexible, can be chanç or to correct the design errors quickly and cheaply.
- Complex function such as f tly.

Disadvantages

- A microprogrammed contr :ontrol unit, because time is required to access the microi
- The flexibility is achieved ε memory and its access circuitry.

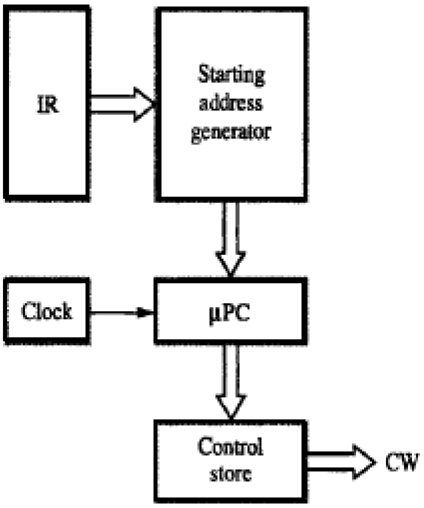
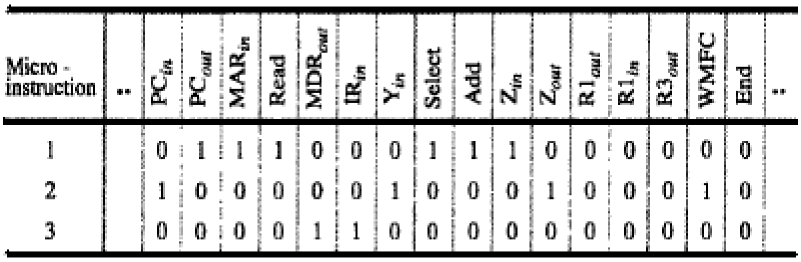


Figure 7.16 Basic organization of a microprogrammed control unit.



Micro - instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	..
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	

Figure 7.15 An example of microinstructions for Figure 7.6.

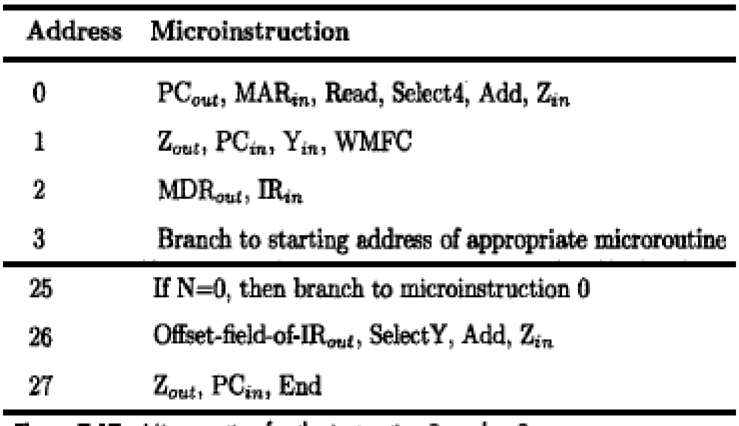
ORGANIZATION OF MICROPROGRAMMED CONTROL UNIT TO SUPPORT CONDITIONAL BRANCHING

- **Drawback of previous Microprogram control:**
  - ➢ It cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.

Solution:

- ➢ Use conditional branch microinstruction.

- In case of conditional branching, microinstructions specify which of the external inputs, condition-codes should be checked as a condition for branching to take place.
- **Starting and Branch Address Generator Block** loads a new address into μPC when a microinstruction instructs it to do so (Figure 7.18).
- To allow implementation of a conditional branch, inputs to this block consist of
  - → external inputs and condition-codes &
  - → contents of IR.
- μPC is incremented every time a new microinstruction is fetched from microprogram memory except in following situations:
  1) When a new instruction is loaded into IR, μPC is loaded with starting-address of microroutine for that instruction.
  2) When a Branch microinstruction is encountered and branch condition is satisfied, μPC is loaded with branch-address.
  3) When an End microinstruction is encountered, μPC is loaded with address of first CW in microroutine for instruction fetch cycle.



Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17 Microroutine for the instruction Branch < 0.

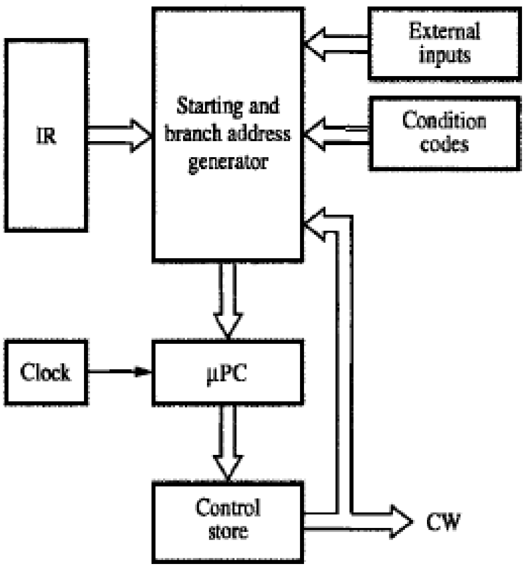
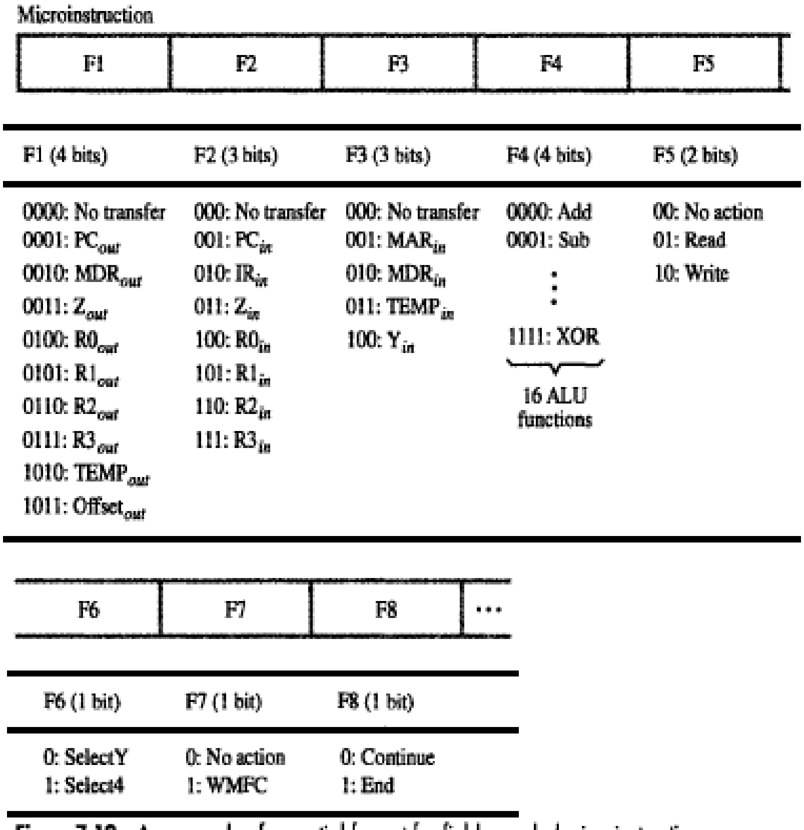


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

MICROINSTRUCTIONS

- A simple way to structure microinstructions is to assign one bit position to each control-signal required in the CPU.
- There are 42 signals and hence each microinstruction will have 42 bits.
- **Drawbacks of microprogrammed control:**
    1) Assigning individual bits to each control-signal results in long microinstructions because the number of required signals is usually large.
    2) Available bit-space is poorly used because only a few bits are set to 1 in any given microinstruction.
- **Solution:** Signals can be grouped because
    1) Most signals are not needed simultaneously.
    2) Many signals are mutually exclusive. E.g. only 1 function of ALU can be activated at a time.

    For ex: Gating signals: IN and OUT signals (Figure 7.19).
    Control-signals: Read, Write.
    ALU signals: Add, Sub, Mul, Div, Mod.
- Grouping control-signals into fields requires a little more hardware because decoding-circuits must be used to decode bit patterns of each field into individual control-signals.
- **Advantage:** This method results in a smaller control-store (only 20 bits are needed to store the patterns for the 42 signals).



Microinstruction

F1	F2	F3	F4	F5

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$		
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	1111: XOR	
0101: $R1_{out}$	101: $R1_{in}$		16 ALU functions	
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

F6	F7	F8	⋯

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY	0: No action	0: Continue
1: Select4	1: WMFC	1: End

Figure 7.19 An example of a partial format for field-encoded microinstructions.

TECHNIQUES OF GROUPING OF CONTROL-SIGNALS

- The grouping of control-signal can be done either by using
  1) Vertical organization &
  2) Horizontal organisation.

Vertical Organization	Horizontal Organization
Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a vertical organization.	The minimally encoded scheme in which many resources can be controlled with a single microinstuction is called a horizontal organization.
Slower operating-speeds.	Useful when higher operating-speed is desired.
Short formats.	Long formats.
Limited ability to express parallel microoperations.	Ability to express a high degree of parallelism.
Considerable encoding of the control information.	Little encoding of the control information.

MICROPROGRAM SEQUENCING

- The task of microprogram sequencing is done by microprogram sequencer.
- Two important factors must be considered while designing the microprogram sequencer:
  1) The size of the microinstruction &
  2) The address generation time.
- The size of the microinstruction should be minimum so that the size of control memory required to store microinstructions is also less.
- This reduces the cost of control memory.
- With less address generation time, microinstruction can be executed in less time resulting better throughout.
- During execution of a microprogram the address of the next microinstruction to be executed has 3 sources:
  1) Determined by instruction register.
  2) Next sequential address &
  3) Branch.
- Microinstructions can be shared using microinstruction branching.
- **Disadvantage of microprogrammed branching**:
  1) Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control-store.
  2) Execution time is longer because it takes more time to carry out the required branches.
- Consider the instruction *Add src,Rdst* ;which adds the source-operand to the contents of Rdst and places the sum in Rdst.
- Let source-operand can be specified in following addressing modes (Figure 7.20):
  a) Indexed
  b) Autoincrement
  c) Autodecrement
  d) Register indirect &
  e) Register direct
- Each box in the chart corresponds to a microinstruction that controls the transfers and operations indicated within the box.
- The microinstruction is located at the address indicated by the octal number (001,002).

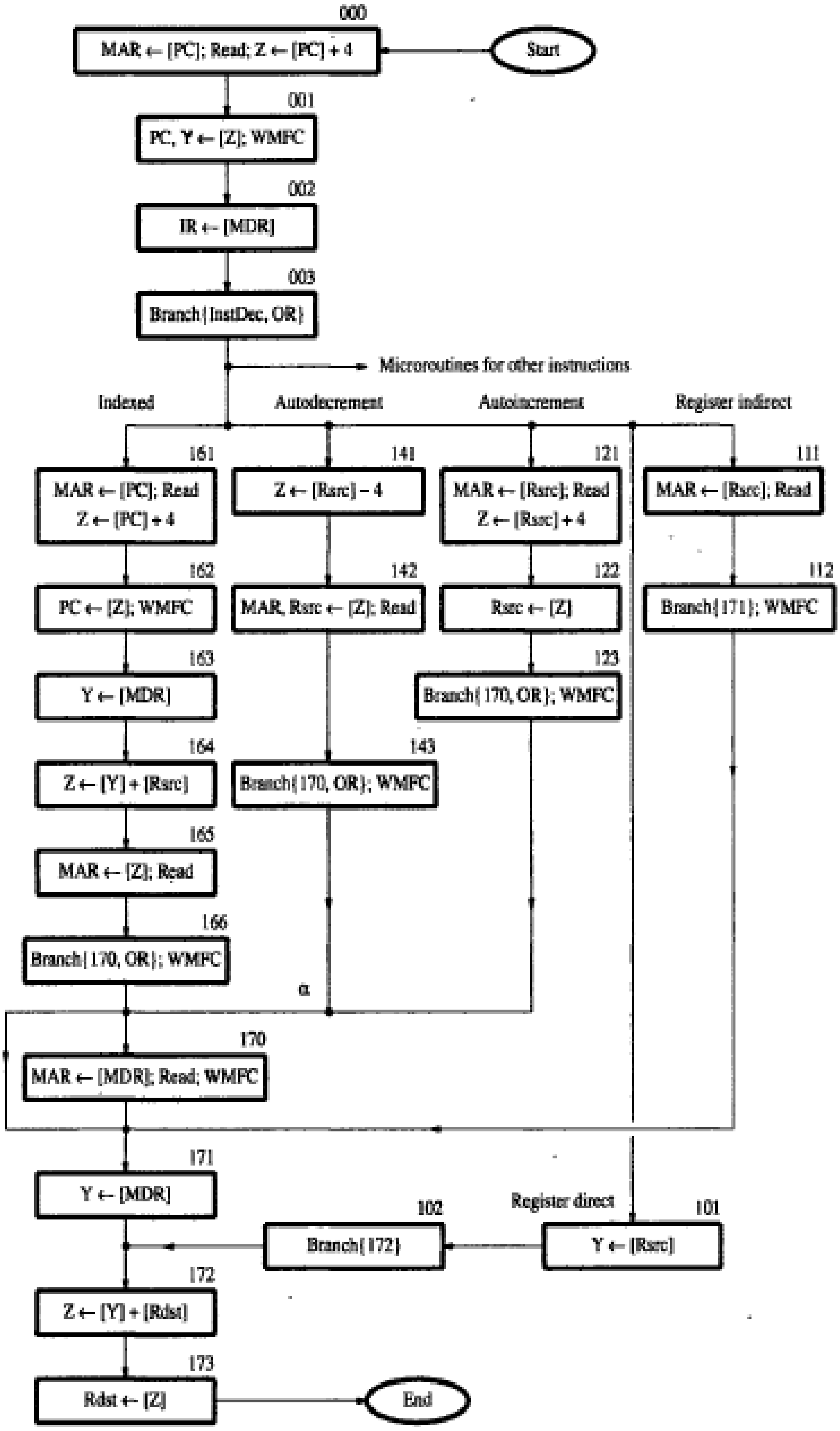


Figure 7.20 Flowchart of a microprogram for the Add src,Rdst instruction.

BRANCH ADDRESS MODIFICATION USING BIT-ORING

- The branch address is determined by ORing particular bit or bits with the current address of microinstruction.
- **Eg:** If the current address is 170 and branch address is 171 then the branch address can be generated by ORing 01(bit 1), with the current address.
- Consider the point labeled **α** in the figure. At this point, it is necessary to choose between direct and indirect addressing modes.
- If indirect-mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory.

  If direct-mode is specified, this fetch must be bypassed by branching immediately to location 171.
- The most efficient way to bypass microinstruction 170 is to have bit-ORing of

  → current address 170 &

  → branch address 171.

WIDE BRANCH ADDRESSING

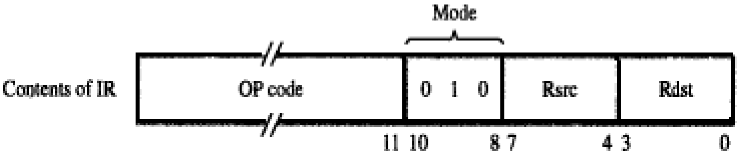
- The instruction-decoder (InstDec) generates the starting-address of the microroutine that implements the instruction that has just been loaded into the IR.
- Here, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101. (However, this address cannot be loaded as is into the μPC).
- The source-operand can be specified in any of several addressing-modes. The bit-ORing technique can be used to modify the starting-address generated by the instruction-decoder to reach the appropriate path.

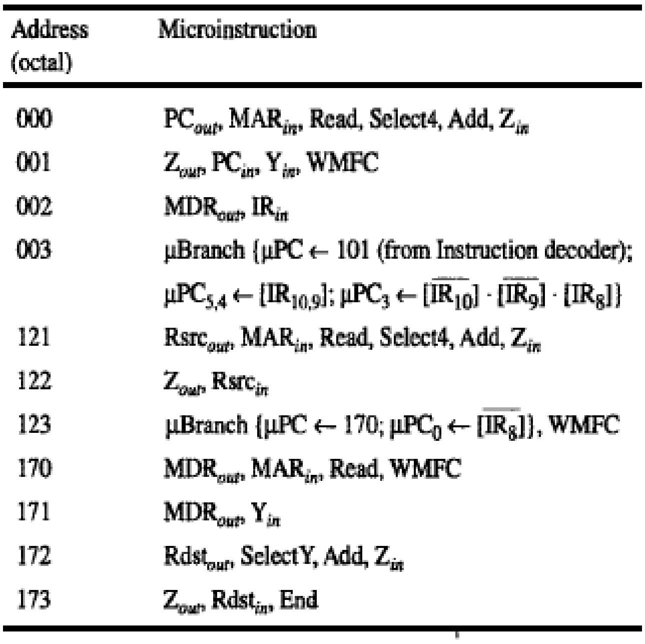
Use of WMFC

- WMFC signal is issued at location 112 which causes a branch to the microinstruction in location 171.
- WMFC signal means that the microinstruction may take several clock cycles to complete. If the branch is allowed to happen in the first clock cycle, the microinstruction at location 171 would be fetched and executed prematurely. To avoid this problem, WMFC signal must inhibit any change in the contents of the μPC during the waiting-period.

Detailed Examination of Add (Rsrc)+,Rdst

- Consider *Add (Rsrc)+,Rdst;* which adds Rsrc content to Rdst content, then stores the sum in Rdst and finally increments Rsrc by 4 (i.e. auto-increment mode).
- In bit 10 and 9, bit-patterns 11, 10, 01 and 00 denote indexed, auto-decrement, auto-increment and register modes respectively. For each of these modes, bit 8 is used to specify the indirect version.
- The processor has 16 registers that can be used for addressing purposes; each specified using a 4-bit-code (Figure 7.21).
- There are 2 stages of decoding:
  1) The microinstruction field must be decoded to determine that an Rsrc or Rdst register is involved.
  2) The decoded output is then used to gate the contents of the Rsrc or Rdst fields in the IR into a second decoder, which produces the gating-signals for the actual registers R0 to R15.





Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {μPC ← 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {μPC ← 170; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21 Microinstruction for Add (Rsrc)+,Rdst.

MICROINSTRUCTIONS WITH NEXT-ADDRESS FIELDS

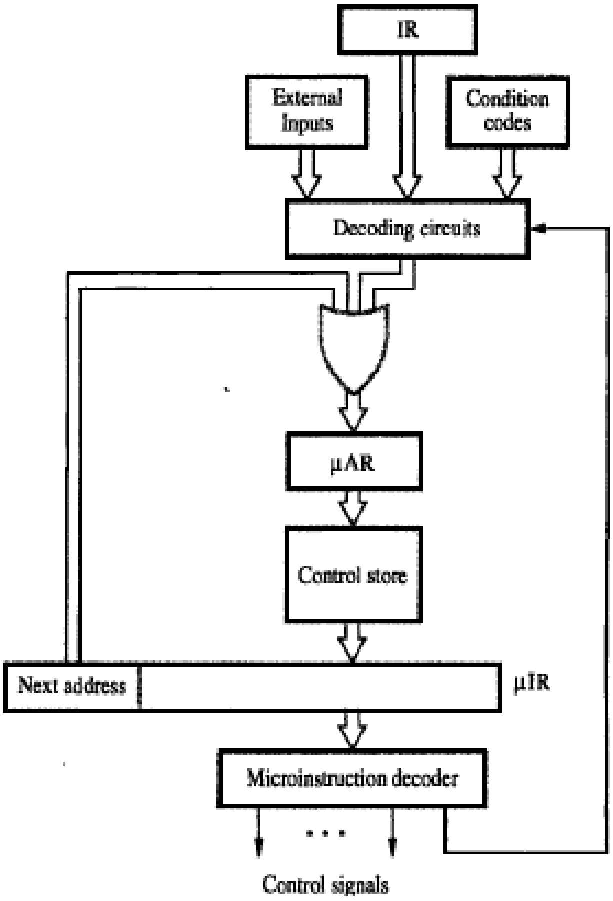


Figure 7.22 Microinstruction-sequencing organization.

- **Drawback of previous organization:**
  - ➢ The microprogram requires several branch microinstructions which perform no useful operation. Thus, they detract from the operating-speed of the computer.

Solution:

- ➢ Include an address-field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched. (Thus, every microinstruction becomes a branch microinstruction).

- The flexibility of this approach comes at the expense of additional bits for the address-field(Fig 7.22).
- **Advantage:** Separate branch microinstructions are virtually eliminated. (Figure 7.23-24).
- **Disadvantage:** Additional bits for the address field (around 1/6).
- There is no need for a counter to keep track of sequential address. Hence, µPC is replaced with µAR.
- The next-address bits are fed through the OR gate to the µAR, so that the address can be modified on the basis of the data in the IR, external inputs and condition-codes.
- The decoding circuits generate the starting-address of a given microroutine on the basis of the opcode in the IR. (µAR → Microinstruction Address Register).

Microinstruction

F0	F1	F2	F3

F0 (8 bits)	F1 (3 bits)	F2 (3 bits)	F3 (3 bits)
Address of next microinstruction	000: No transfer 001: PC_{out} 010: MDR_{out} 011: Z_{out} 100: $Rsrc_{out}$ 101: $Rdst_{out}$ 110: $TEMP_{out}$	000: No transfer 001: PC_{in} 010: IR_{in} 011: Z_{in} 100: $Rsrc_{in}$ 101: $Rdst_{in}$	000: No transfer 001: MAR_{in} 010: MDR_{in} 011: $TEMP_{in}$ 100: Y_{in}

F4	F5	F6	F7

F4 (4 bits)	F5 (2 bits)	F6 (1 bit)	F7 (1 bit)
0000: Add 0001: Sub ⋮ 1111: XOR	00: No action 01: Read 10: Write	0: SelectY 1: Select4	0: No action 1: WMFC

F8	F9	F10

F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
0: NextAdrs 1: InstDec	0: No action 1: OR_{mode}	0: No action 1: OR_{indsrc}

Figure 7.23 Format for microinstructions in the example of Section 7.5.3.

Octal address	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
0 0 0	0 0 0 0 0 0 0 1	0 0 1	0 1 1	0 0 1	0 0 0 0	0 1	1	0	0	0	0
0 0 1	0 0 0 0 0 0 1 0	0 1 1	0 0 1	1 0 0	0 0 0 0	0 0	0	1	0	0	0
0 0 2	0 0 0 0 0 0 1 1	0 1 0	0 1 0	0 0 0	0 0 0 0	0 0	0	0	0	0	0
0 0 3	0 0 0 0 0 0 0 0	0 0 0	0 0 0	0 0 0	0 0 0 0	0 0	0	0	1	1	0
1 2 1	0 1 0 1 0 0 1 0	1 0 0	0 1 1	0 0 1	0 0 0 0	0 1	1	0	0	0	0
1 2 2	0 1 1 1 1 0 0 0	0 1 1	1 0 0	0 0 0	0 0 0 0	0 0	0	1	0	0	1
1 7 0	0 1 1 1 1 0 0 1	0 1 0	0 0 0	0 0 1	0 0 0 0	0 1	0	1	0	0	0
1 7 1	0 1 1 1 1 0 1 0	0 1 0	0 0 0	1 0 0	0 0 0 0	0 0	0	0	0	0	0
1 7 2	0 1 1 1 1 0 1 1	1 0 1	0 1 1	0 0 0	0 0 0 0	0 0	0	0	0	0	0
1 7 3	0 0 0 0 0 0 0 0	0 1 1	1 0 1	0 0 0	0 0 0 0	0 0	0	0	0	0	0

Figure 7.24 Implementation of the microroutine of Figure 7.21 using a next-microinstruction address field. (See Figure 7.23 for encoded signals.)

PREFETCHING MICROINSTRUCTIONS

• **Disadvantage of Microprogrammed Control:** Slower operating-speed because of the time it takes to fetch microinstructions from the control-store.

Solution: Faster operation is achieved if the next microinstruction is pre-fetched while the current one is being executed.

Emulation

• The main function of microprogrammed control is to provide a means for simple, flexible and relatively inexpensive execution of machine instruction.

• Its flexibility in using a machine's resources allows diverse classes of instructions to be implemented.

• Suppose we add to the instruction-repository of a given computer M1, an entirely new set of instructions that is in fact the instruction-set of a different computer M2.

• Programs written in the machine language of M2 can be then be run on computer M1 i.e. M1 emulates M2.

• Emulation allows us to replace obsolete equipment with more up-to-date machines.

• If the replacement computer fully emulates the original one, then no software changes have to be made to run existing programs.

• Emulation is easiest when the machines involved have similar architectures.

Problem 1:
Why is the Wait-for-memory-function-completed step needed for reading from or writing to the main memory?
Solution:

The WMFC step is needed to synchronize the operation of the processor and the main memory.

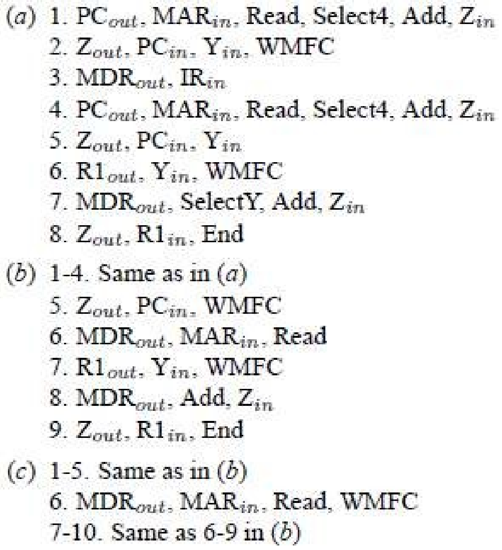
Problem 2:
For the single bus organization, write the complete control sequence for the instruction: Move (R1), R1
Solution:

PC$_{out}$, MAR$_{in}$, Read, Select4, Add, Z$_{in}$
2) Z$_{out}$, PC$_{in}$, Y$_{in}$, WMFC
3) MDR$_{out}$, IR$_{in}$
4) R1$_{out}$, MAR$_{in}$, Read
5) MDR$_{inE}$, WMFC
6) MDR$_{out}$, R2$_{in}$, End

Problem 3:
1)
Write the sequence of control steps required for the single bus organization in each of the following instructions:

a) Add the immediate number NUM to register R1.
b) Add the contents of memory-location NUM to register R1.
c) Add the contents of the memory-location whose address is at memory-location NUM to register R1.

Assume that each instruction consists of two words. The first word specifies the operation andN the addressing mode, and the second word contains the number NUM
Solution:

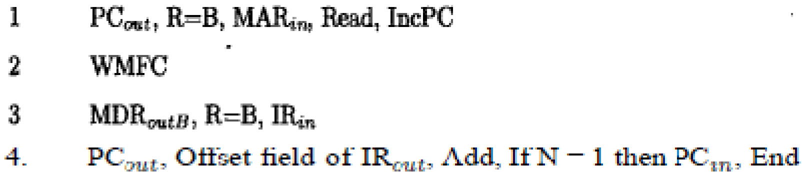


(*a*) 1. PC$_{out}$, MAR$_{in}$, Read, Select4, Add, Z$_{in}$
2. Z$_{out}$, PC$_{in}$, Y$_{in}$, WMFC
3. MDR$_{out}$, IR$_{in}$
4. PC$_{out}$, MAR$_{in}$, Read, Select4, Add, Z$_{in}$
5. Z$_{out}$, PC$_{in}$, Y$_{in}$
6. R1$_{out}$, Y$_{in}$, WMFC
7. MDR$_{out}$, SelectY, Add, Z$_{in}$
8. Z$_{out}$, R1$_{in}$, End

(*b*) 1-4. Same as in (*a*)
5. Z$_{out}$, PC$_{in}$, WMFC
6. MDR$_{out}$, MAR$_{in}$, Read
7. R1$_{out}$, Y$_{in}$, WMFC
8. MDR$_{out}$, Add, Z$_{in}$
9. Z$_{out}$, R1$_{in}$, End

(*c*) 1-5. Same as in (*b*)
6. MDR$_{out}$, MAR$_{in}$, Read, WMFC
7-10. Same as 6-9 in (*b*)

Problem 4:
Show the control steps for the Branch on Negative instruction for a processor with three-bus organization of the data path
Solution:



1 PC$_{out}$, R=B, MAR$_{in}$, Read, IncPC
2 WMFC
3 MDR$_{outB}$, R=B, IR$_{in}$
4. PC$_{out}$, Offset field of IR$_{out}$, Add, If N − 1 then PC$_{in}$, End

MODULE 5(CONT.): EMBEDDED SYSTEMS & LARGE COMPUTER SYSTEMS

MICROWAVE OVEN

- Microwave-oven is one of the examples of embedded-system.
- This appliance is based on **magnetron** power-unit that generates the microwaves used to heat food.
- When turned-on, the magnetron generates its maximum power-output.
  Lower power-levels can be obtained by turning the magnetron on & off for controlled time-intervals.
- **Cooking Options** include:
  → Manual selection of the power-level and cooking-time.
  → Manual selection of the sequence of different cooking-steps.
  → Automatic melting of food by specifying the weight.
- **Display (or Monitor)** can show following information:
  → Time-of-day clock.
  → Decrementing clock-timer while cooking.
  → Information-messages to the user.
- **I/O Capabilities** include:
  → Input-keys that comprise a 0 to 9 number pad.
  → Function-keys such as Start, Stop, Reset, Power-level etc.
  → Visual output in the form of a LCD.
  → Small speaker that produces the beep-tone.
- **Computational Tasks** executed are:
  → Maintaining the time-of-day clock.
  → Determining the actions needed for the various cooking-options.
  → Generating the control-signals needed to turn on/off devices.
  → Generating display information.

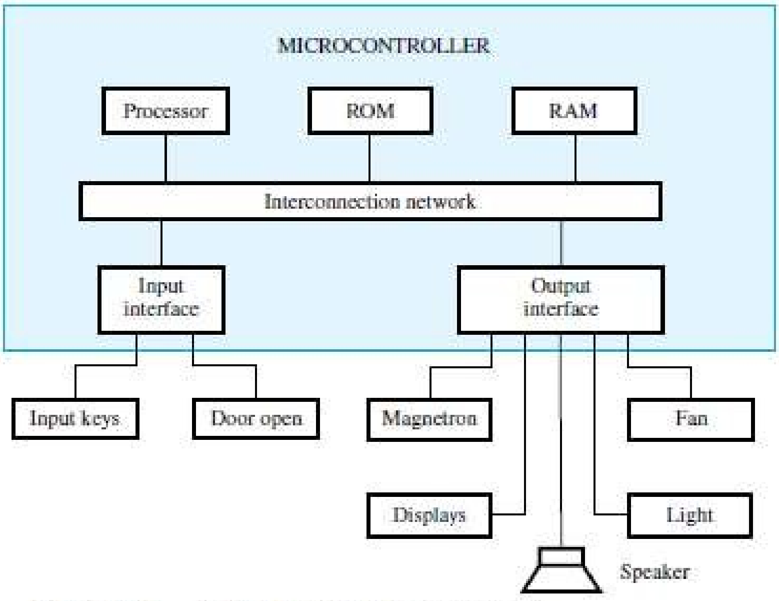


Figure 10.1 A block diagram of a microwave oven.

- **Non-volatile ROM** is used to store the program required to implement the desired actions.
  So, the program will not be lost when the power is turned off (Figure 10.1).
- Most important requirement: The microcontroller must have sufficient I/O capability.
  Parallel I/O Ports are used for dealing with the external I/O signals.
  Basic I/O Interfaces are used to connect to the rest of the system.

DIGITAL CAMERA

- Digital Camera is one of the examples of embedded system.
- An array of **Optical Sensors** is used to capture images (Figure 10.2).
- The optical-sensors convert light into electrical charge.

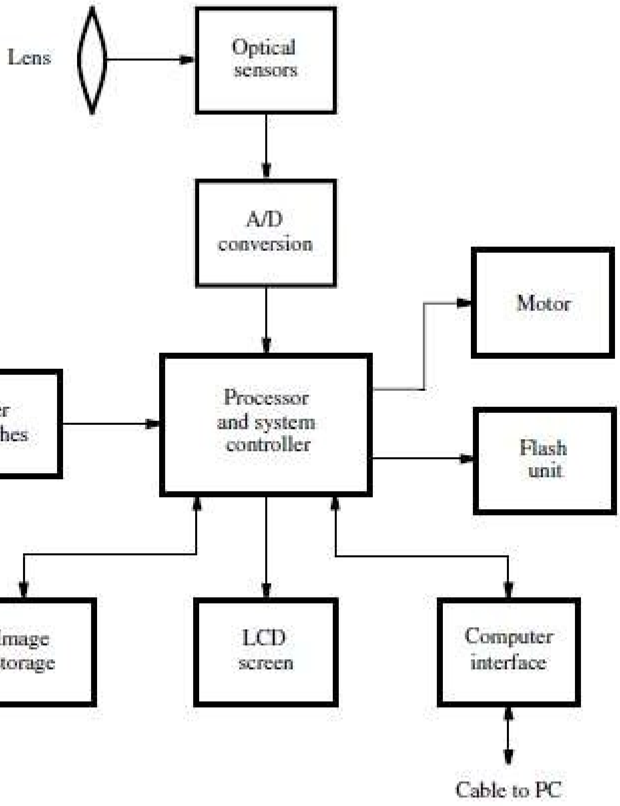


Figure 10.2 A simplified block diagram of a digital camera.

- Each sensing-element generates a charge that corresponds to one **pixel**.
  One pixel is one point of a pictorial image.
  The number of pixels determines the quality of pictures that can be recorded & displayed.
- **ADC** is used to convert the charge which is an analog quantity into a digital representation.
- **Processor**
  → manages the operation of the camera.
  → processes the raw image-data obtained from the ADCs to generate images.
- The images are represented in standard-formats, so that they are suitable for use in computers.
- Two standard-formats are:
  1) TIFF is used for uncompressed images &
  2) JPEG is used for compressed images.
- The processed-images are stored in a larger storage-device. For ex: Flash memory cards.
- A captured & processed image can be displayed on a LCD screen of camera.
- The number of saved-images depends on the size of the storage-unit.
- Typically, **USB Cable** is used for transferring the images from camera to the computer.
- **System Controller** generates the signals needed to control the operation of
  i) Focusing mechanism and
  ii) Flash unit.

(ADC → Analog-to-digital converter, LCD → liquid-crystal display)
(TIFF →Tagged Image File Format, JPEG →Joint Photographic Experts Group)

HOME TELEMETRY (DISPLAY TELEPHONE)

- Home Telemetry is one of the examples of embedded system.
- The display-telephone has an embedded processor which enables a remote access to other devices in the home.
- Display telephone can perform following functions:
  1) Communicate with a computer-controlled home security-system.
  2) Set a desired temperature to be maintained by an air conditioner.
  3) Set start-time, cooking-time & temperature for food in the microwave-oven.
  4) Read the electricity, gas, and water meters.
- All of this is easily implementable if each of these devices is controlled by a **microcontroller**.
- A link (wired or wireless) has to be provided between
  1) Device microcontroller &
  2) Microprocessor in the telephone.
- Using signaling from a remote location to observe/control state of device is referred to as **telemetry**.

MICROCONTROLLER CHIPS FOR EMBEDDED APPLICATIONS

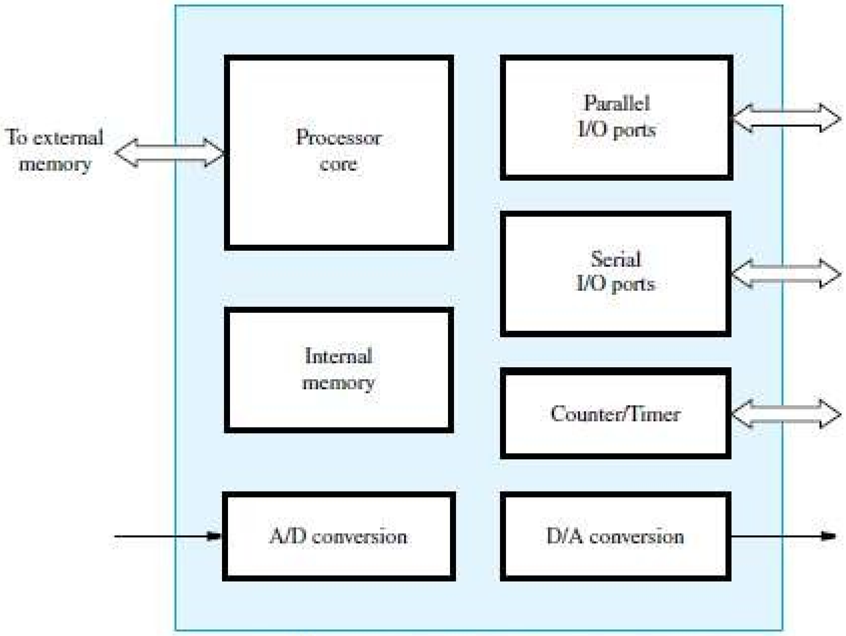


Figure 10.3 A block diagram of a microcontroller.

• **Processor Core** may be a basic version of a commercially available microprocessor (Figure 10.3).
• Well-known popular microprocessor architecture must be chosen. This is because, design of new products is facilitated by
→ numerous CAD tools
→ good examples &
→ large amount of knowledge/experience.
• **Memory-Unit** must be included on the microcontroller-chip.
• The memory-size must be sufficient to satisfy the memory-requirements found in small applications.
• Some memory should be of **RAM** type to hold the data that change during computations.
Some memory should be of **Read-Only** type to hold the software.
This is because an embedded system usually does not include a magnetic-disk.
• A field-programmable type of ROM storage must be provided to allow cost-effective use.
For example: EEPROM and Flash memory.
• **I/O ports** are provided for both parallel and serial interfaces.
• **Parallel and Serial Interfaces** allow easy implementation of standard I/O connections.
• **Timer Circuit** can be used
→ to generate control-signals at programmable time intervals &
→ for event-counting purposes.
• An embedded system may include some **analog devices**.
• **ADC & DAC** are used to convert analog signals into digital representations, and vice versa.

PARALLEL I/O INTERFACE

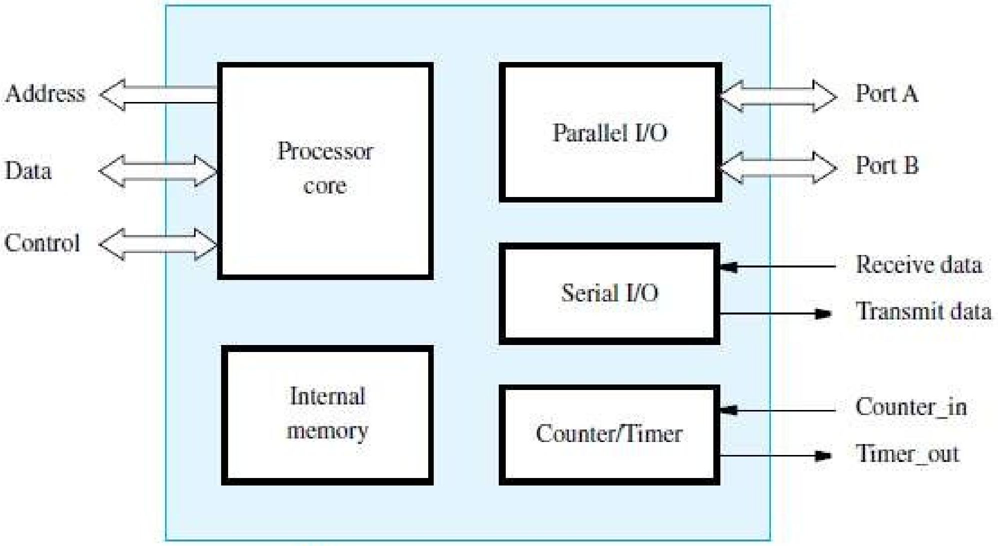


Figure 10.4 An example microcontroller.

- Each parallel port has an associated 8-bit DDR (Data Direction Register) (Figure 10.4).
- **DDR** can be used to configure individual data lines as either input or output.

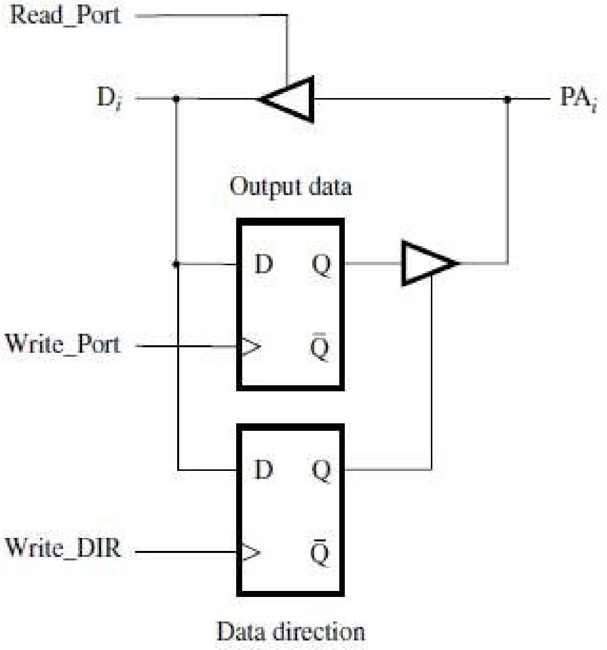


Figure 10.5 Access to one bit in port A in Figure 10.4.

- If the data direction flip-flop contains a 0, then Port pin **PA$_i$** is treated as an input (Figure 10.5).
  If the data direction flip-flop contains a 1, then Port pin PA$_i$ is treated as an output.
- Activation of control-signal **Read_Port**, places the logic value on the port-pin onto the data line D$_i$.
  Activation of control-signal **Write_Port**, places value loaded into output data flip-flop onto port-pin.
- **Addressable Registers** are (Figure 10.6):
  1) Input registers (PAIN for port A, PBIN for port B)
  2) Output registers (PAOUT for port A, PBOUT for port B)
  3) Direction registers (PADIR for port A, PBDIR for port B)
  4) Status-register (PSTAT) &
  5) Control register (PCONT).

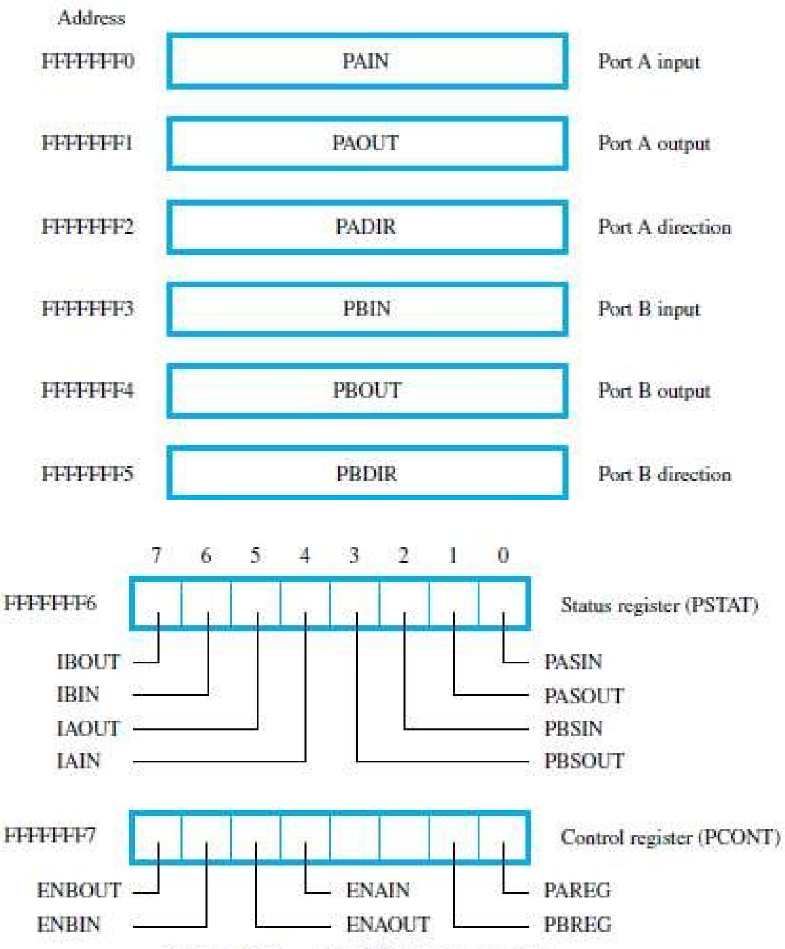


Figure 10.6 Parallel interface registers.

- **Status Register** provides information about the current status of
  1) Input registers &
  2) Output registers.
- PASIN =1 → When there are new data on port A (Figure 10.6).
  PASIN =0 → When the processor accepts the data by reading the PAIN register.
- The interface uses a separate control line to indicate availability of new data to the connected-device.
- PASOUT = 1 → When the data in register PAOUT are accepted by the connected-device.
  PASOUT = 0 → When the processor writes data into PAOUT.
- Like PASIN & PAOUT, the flags PBSIN and PBSOUT perform the same function on port B.
- The status register also contains four interrupt flags. They are IAIN, IAOUT, IBIN & IBOUT.
- IAIN = 1 → When interrupt is enabled and the corresponding I/O action occurs.
- The interrupt-enable bits are held in control register PCONT.
- ENAIN= 1 → when the corresponding interrupt is enabled.
- For ex: If ENAIN=1 & PASIN=1, then interrupt flag IAIN is set to 1 and an interrupt request is raised. Thus,
  IAIN = ENAIN * PASIN
- **Control Registers** is used for controlling data transfers to/from the devices connected to ports A/B.
- Port A has two control lines: CAIN and CAOUT.
- CAIN and CAOUT are be used to provide an automatic signaling mechanism b/w
  i) Interface and
  ii) Attached device.
- PAREG and PBREG are used to select the mode of operation of inputs to ports A and B respectively.
- If PAREG =1;
  Then, a register is used to store the input data.
  Otherwise, a direct path from the pins is used.

SERIAL I/O INTERFACE

- The serial interface provides the UART capability to transfer data (Figure 10.7).
  (UART → Universal Asynchronous Receiver/Transmitter).
- Double buffering is
  → used in both the transmit- and receive-paths.
  → needed to handle bursts in I/O transfers correctly.

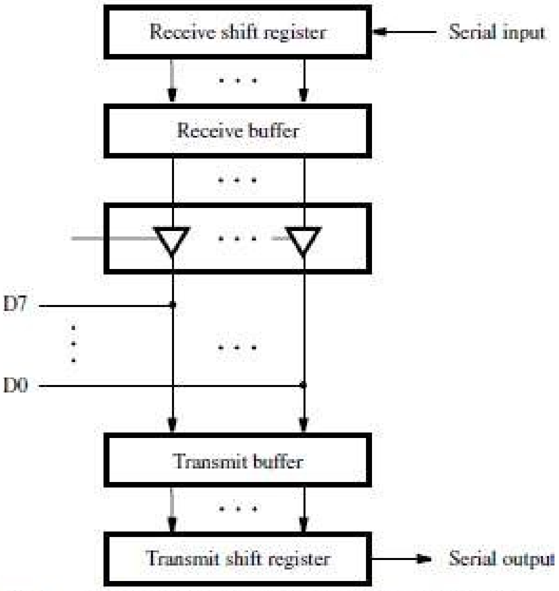


Figure 10.7 Receive and transmit structure of the serial interface.

- **Addressable Registers** are (Figure 10.8):
  1) Receive-buffer
  2) Transmit-buffer
  3) Status-register (SSTAT)
  4) Control register (SCONT) &
  5) Clock-divisor register (DIV).

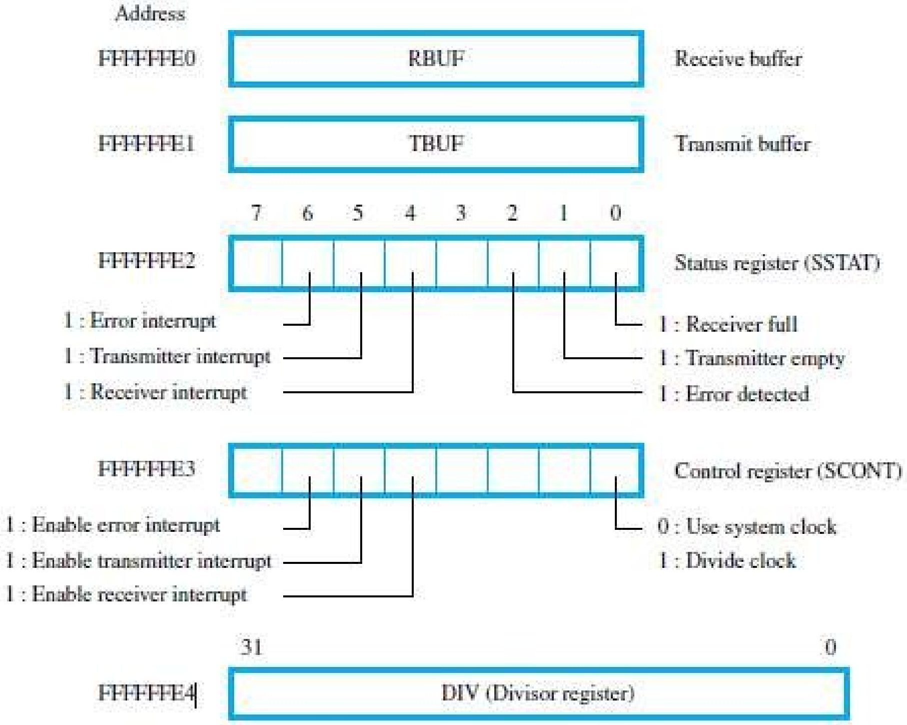


Figure 10.8 Serial interface registers.

- Input data are read from the **Receive-buffer**.
  Output data are loaded into the **Transmit-buffer**.
- **Status Register** (SSTAT) provides information about the current status of
  i) Receive-units and
  ii) Transmit-units.
- Bit SSTAT0 = 1 → When there are new data in the receive-buffer.
  Bit SSTAT0 = 0 → When the processor accepts the data by reading the receive-buffer.
- SSTAT1 = 1 → When the data in transmit-buffer are accepted by the connected-device.
  SSTAT1 = 0 → When the processor writes data into transmit-buffer.
  (SSTAT0 & SSTAT1 similar to SIN & SOUT)
- SSTAT2 = 1 → if an error occurs during the receive process.
- The status-register also contains the interrupt flags.
- SSTAT4 =1 → When the receive-buffer becomes full and the receiver-interrupt is enabled.
  SSTAT5 = 1 → When the transmit-buffer becomes empty & the transmitter-interrupt is enabled.
- **Control Register** (SCONT) is used to hold the interrupt-enable bits.
- If SCONT6−4 = 1.
  Then the corresponding interrupts are enabled.
  Otherwise, the corresponding interrupts are disabled.
- Control register also indicates how the transmit clock is generated
- If SCONT0 = 0.
  Then, the transmit clock is the same as the system (processor) clock.
  Otherwise, a lower frequency transmit clock is obtained using a clock-dividing circuit.
- **Clock-divisor register** (DIV) divides system-clock signal to generate the serial transmission clock.
- The counter generates a clock signal whose frequency is equal to

$$= \frac{\text{Frequency of system clock}}{\text{Contents of DIV register}}$$

COUNTER/TIMER

- A 32-bit down-counter-circuit is provided for use as either a counter or a timer.
- Basic operation of the circuit involves
  → loading a starting value into the counter and
  → then decrementing the counter-contents using either
  i) Internal system clock or
  ii) External clock signal.
- The circuit can be programmed to raise an interrupt when the counter-contents reach 0.

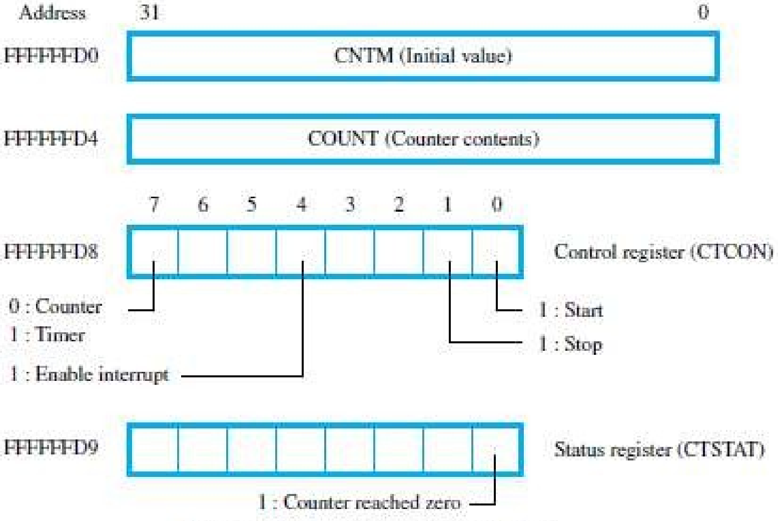


Figure 10.9 Counter/Timer registers.

- **Counter/Timer Register** (CNTM) can be loaded with an initial value (Figure 10.9).
- The initial value is then transferred into the counter-circuit.
- The current contents of the counter can be read by accessing memory-address FFFFFFD4.
- **Control Register** (CTCON) is used to specify the operating mode of the counter/timer circuit.
- The control register provides a mechanism for
  → starting & stopping the counting-process &
  → enabling interrupts when the counter-contents are decremented to 0.
- **Status Register** (CTSTAT) reflects the state of the circuit.
- There are 2 modes: 1) Counter mode 2) Timer mode.

Counter Mode

- CTCON7 = 0 → When the counter mode is selected.
- The starting value is loaded into the counter by writing it into register CNTM.
- The counting-process begins when bit CTCON0 is set to 1 by a program.
- Once counting starts, bit CTCON0 is automatically cleared to 0.
- The counter is decremented by pulses on the Counter.
- Upon reaching 0, the counter-circuit
  → sets the status flag CTSTAT0 to 1 &
  → raises an interrupt if the corresponding interrupt-enable bit has been set to 1.
- The next clock pulse causes the counter to reload the starting value.
- The starting value is held in register CNTM, and counting continues.
- The counting-process is stopped by setting bit CTCON1 to 1.

Timer Mode

- CTCON7 = 1 → When the timer mode is selected.
- This mode can be used to generate periodic interrupts.
- It is also suitable for generating a square-wave signal.
- The process starts as explained above for the counter mode.
- As the counter counts down, the value on the output line is held constant.
- Upon reaching 0, the counter is reloaded automatically with the starting value, and the output signal on the line is inverted.
- Thus, the period of the output signal is twice the starting counter value multiplied by the period of the controlling clock pulse.
- In the timer mode, the counter is decremented by the system clock.

MODULE 5(CONT.): THE STRUCTURE OF GENERAL-PURPOSE MULTIPROCESSORS

THE STRUCTURE OF GENERAL-PURPOSE MULTIPROCESSORS

1. UMA (Uniform Memory Access) Multiprocessor

- An interconnection-network permits n processors to access k memories (Figure 12.2).
  Thus, any of the processors can access any of the memories.
- The interconnection-network may introduce network-delay between
  1) Processor &
  2) Memory.
- A system which has the same network-latency for all accesses from the processors to the memory-modules is called a **UMA Multiprocessor**.
- Although the latency is uniform, it may be large for a network that connects
  → many processors &
  → many memory-modules.
- For better performance, it is desirable to place a memory-module close to each processor.
- **Disadvantage:**
  ➢ Interconnection-networks with very short delays are costly and complex to implement.

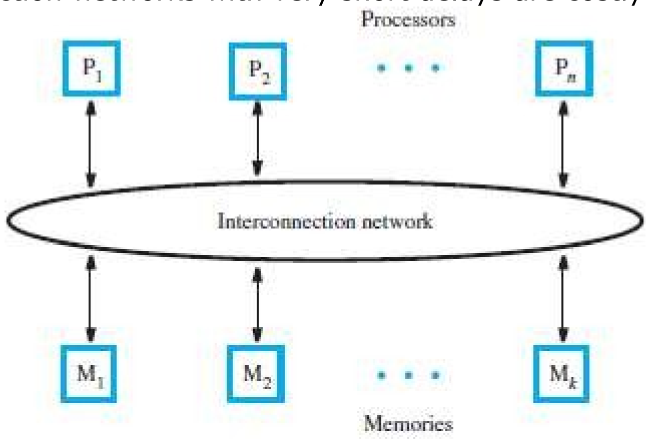


Figure 12.2 A UMA multiprocessor.

2. NUMA (Non-Uniform Memory Access) Multiprocessors

- Memory-modules are attached directly to the processors (Figure 12.3).
- The network-latency is avoided when a processor makes a request to access its local memory.
- However, a request to access a remote-memory-module must pass through the network.
- Because of the difference in latencies for accessing local and remote portions of the shared memory, systems of this type are called **NUMA** multiprocessors.
- **Advantage:**
  ➢ A high computation rate is achieved in all processors
- **Disadvantage:**
  ➢ The remote accesses take considerably longer than accesses to the local memory.

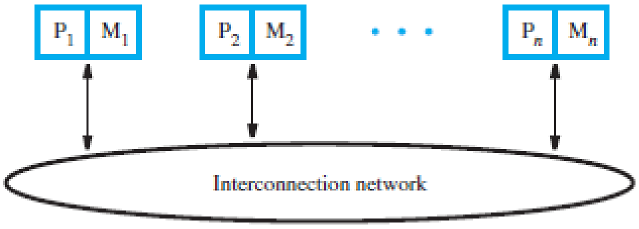


Figure 12.3 A NUMA multiprocessor.

3. Distributed Memory Systems

- All memory-modules serve as private memories for processors that are directly connected to them.
- A processor cannot access a remote-memory without the cooperation of the remote-processor.
- This cooperation takes place in the form of messages exchanged by the processors.
- Such systems are often called **Distributed-Memory Systems** (Figure 12.4).

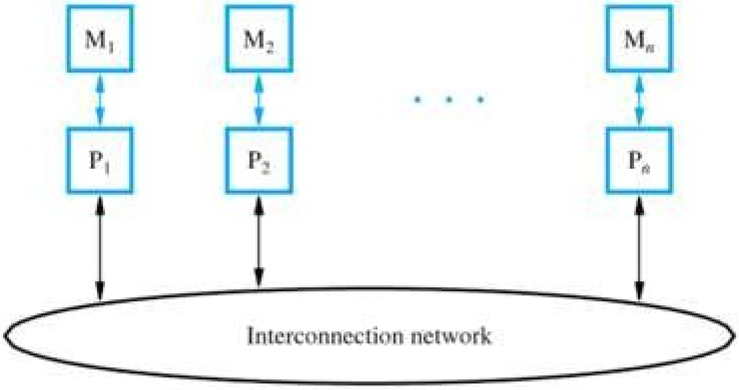


Figure 12.4 A distributed memory system.