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

Type your text

➢ 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.

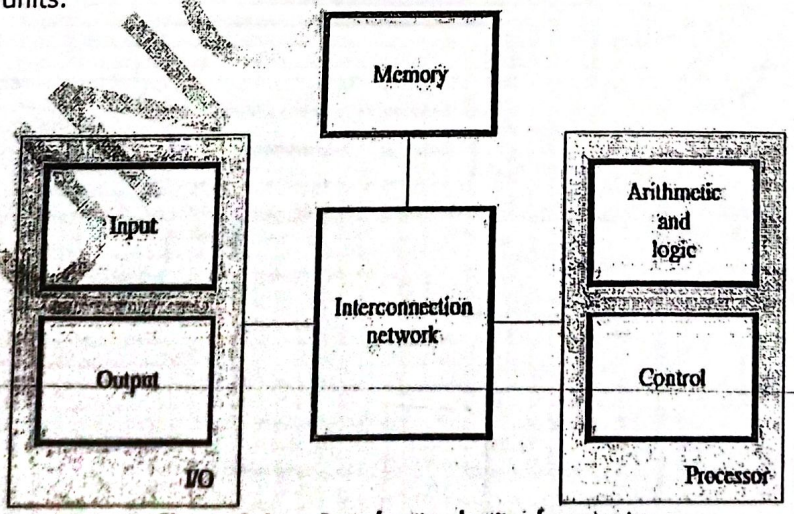


Figure 1.1 Basic functional units of a computer.

COMPUTER ORGANIZATION

BASIC OPERATIONAL CONCEPTS

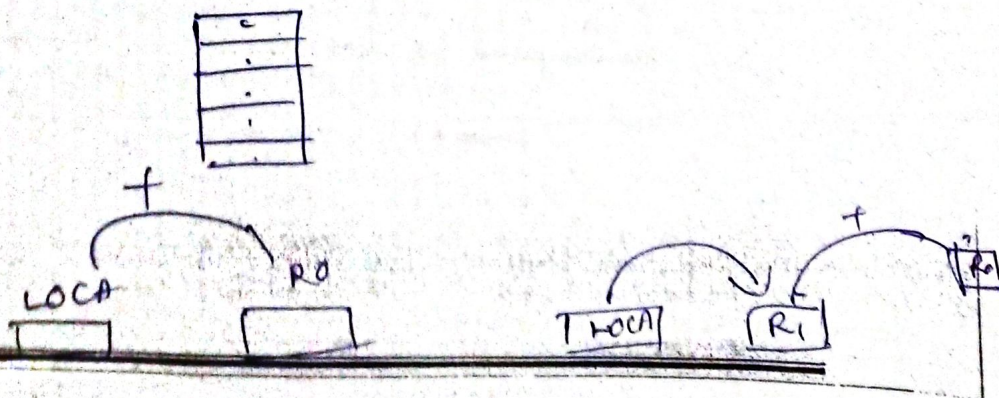
- An Instruction consists of 2 parts, 1) Operation code (Opcode) and 2) Operands.

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.

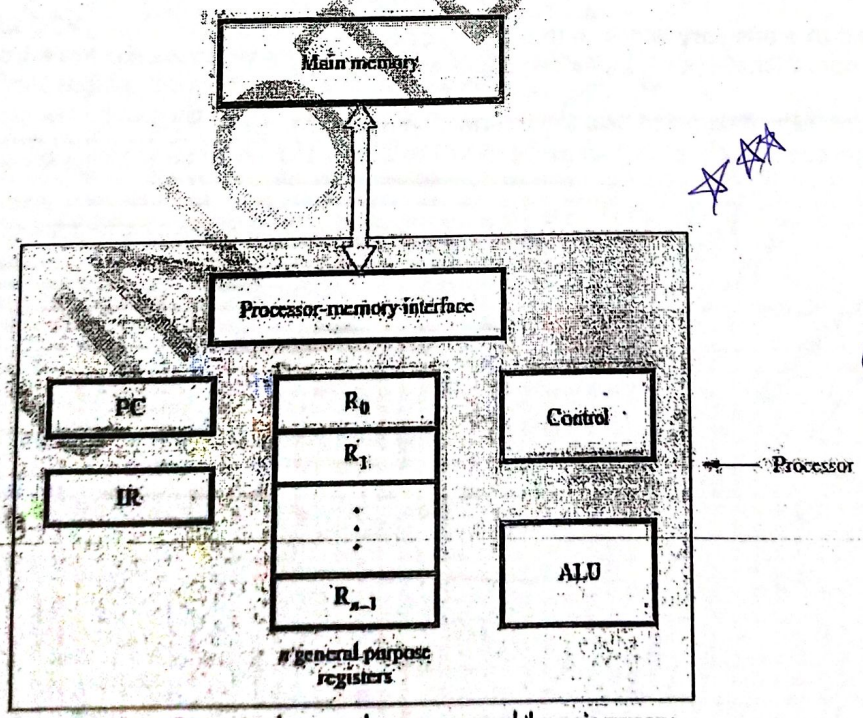


Figure 1.2 Connection between the processor and the main memory.

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.

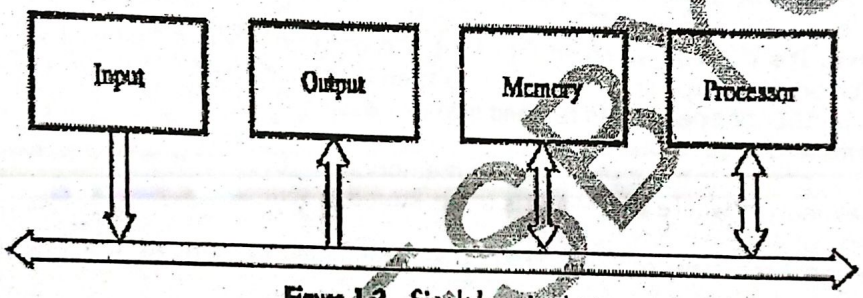
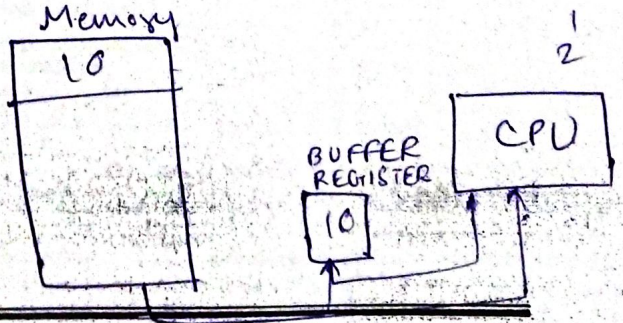


Figure 1.3 Single-bus structure.

- 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.

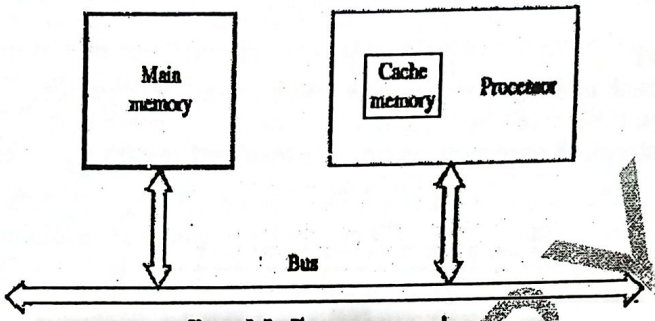


Figure 1.5 The processor cache.

- 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

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

- 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.
$T = (N*S)/R = (250*4+400*5+350*3)/1X10^9 = (1000+2000+1050)/1*10^9 = 4.05\ \mu s.$

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 $T_R = (N_R \times S_R)/R_R$ & $T_C = (N_C \times S_C)/R_C$ 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

$$N_C/N_R = 1.2/(1.15 \times 1.5) = 0.696$$

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).

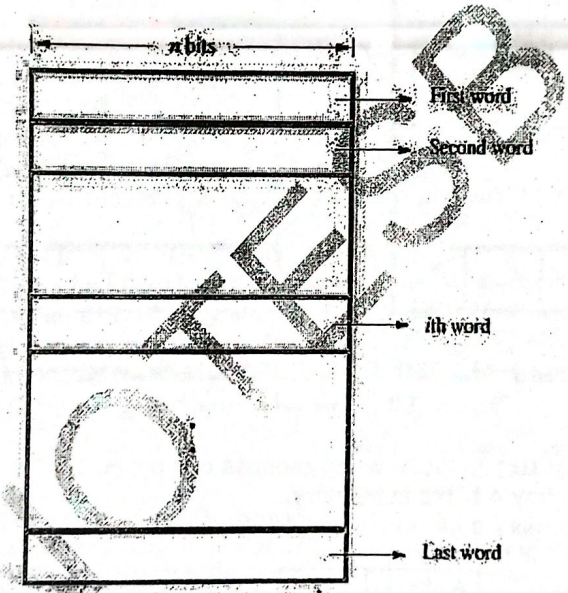
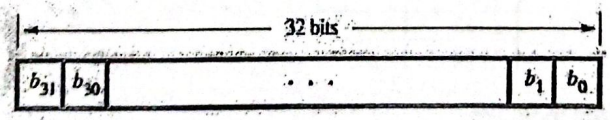
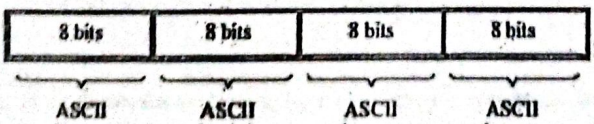


Figure 2.1 Memory words.



Sign bit: $b_{31} = 0$ for positive numbers
$b_{31} = 1$ for negative numbers

(a) A signed integer



(b) Four characters

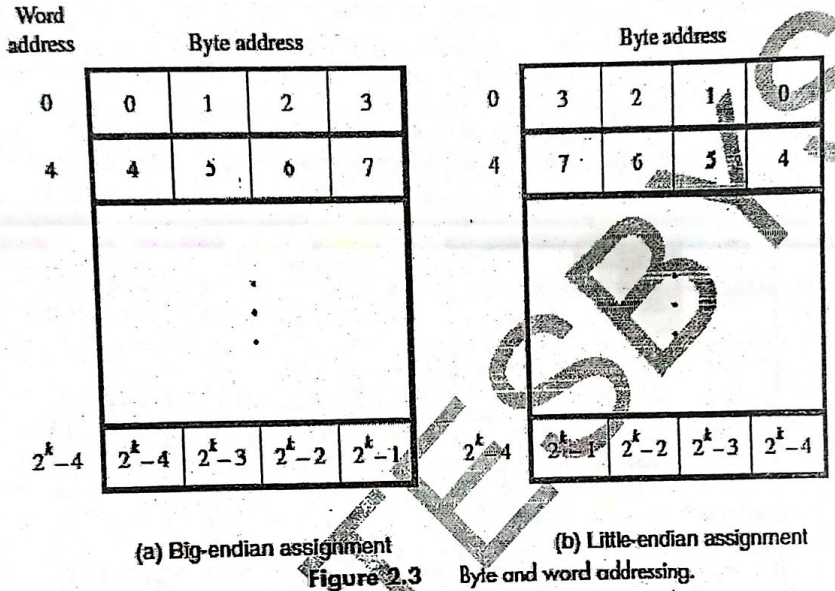
Figure 2.2 Examples of encoded information in a 32-bit word.

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): 0x12345678 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.

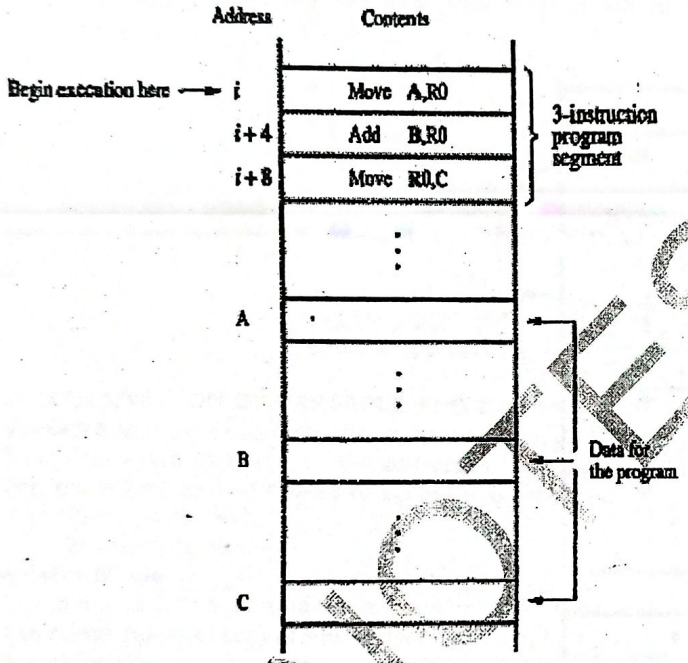


Figure 2.8 A program for C ← [A] + [B].

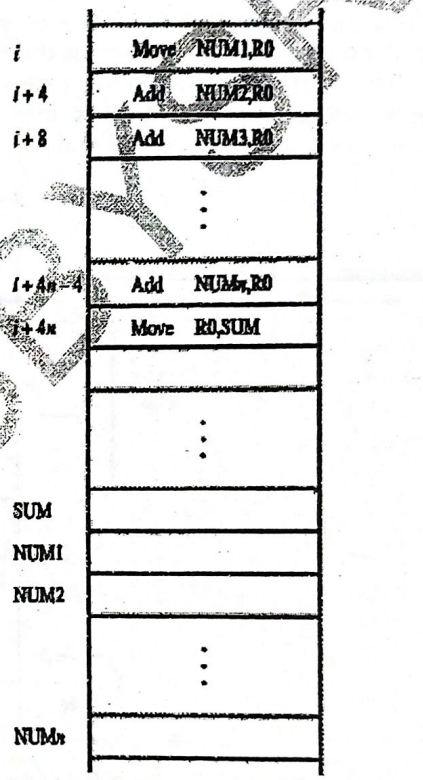


Figure 2.9 A straight-line program for adding n numbers.

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.

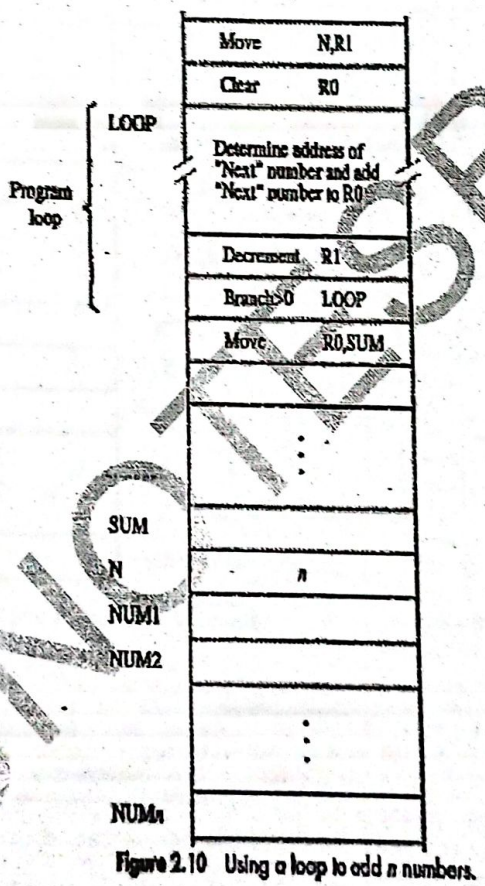


Figure 2.10 Using a loop to add *n* numbers.

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	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri,Rj)	EA = [Ri] + [Rj]
Base with index and offset	X(Ri,Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri]

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.

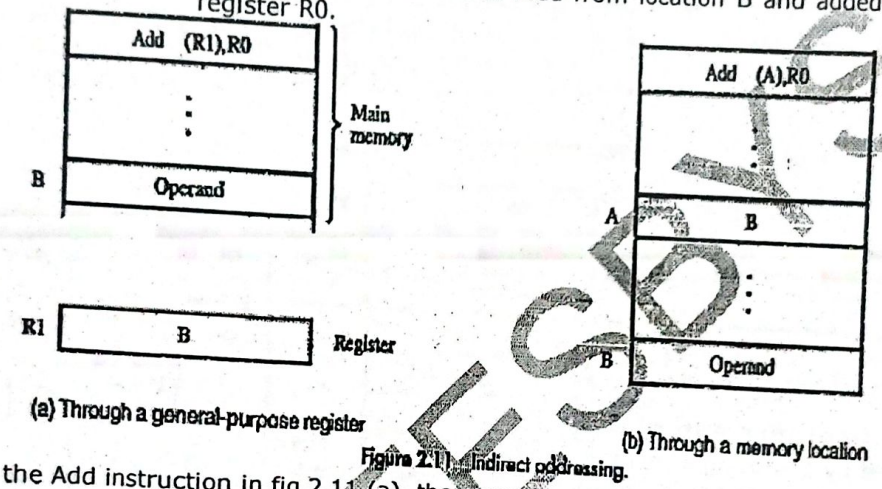


Figure 2.11 Indirect addressing.

- 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	Initialization
	Move	#NUM1,R2	
	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.

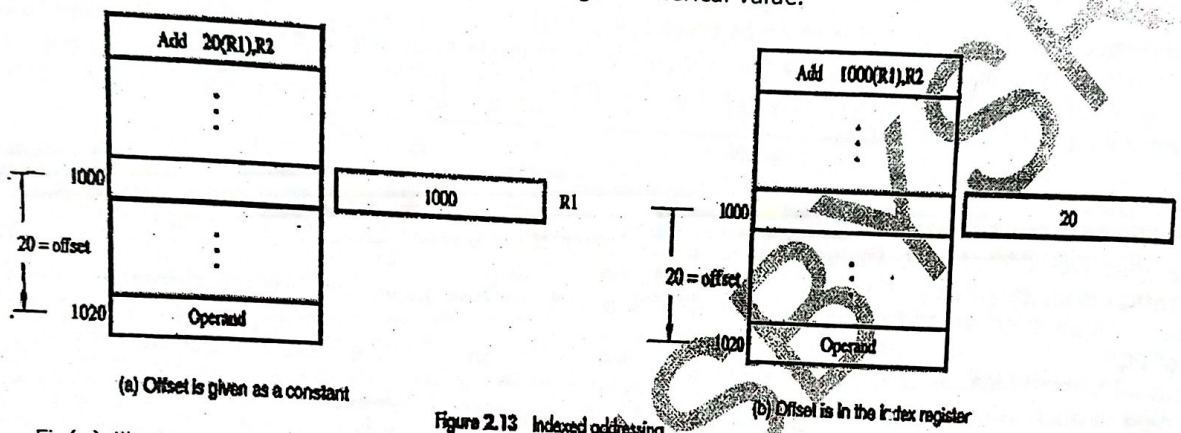


Figure 2.13 Indexed addressing.

• 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.

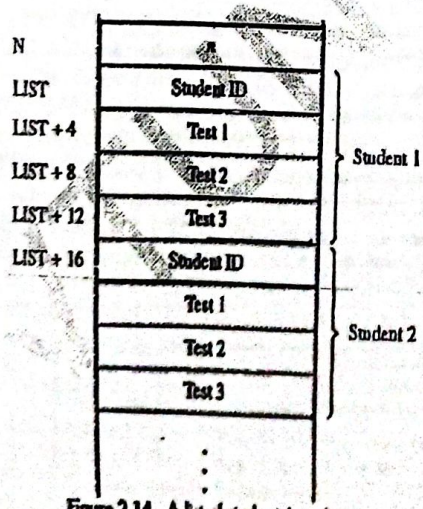


Figure 2.14 A list of students' marks.

```
        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.

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**.

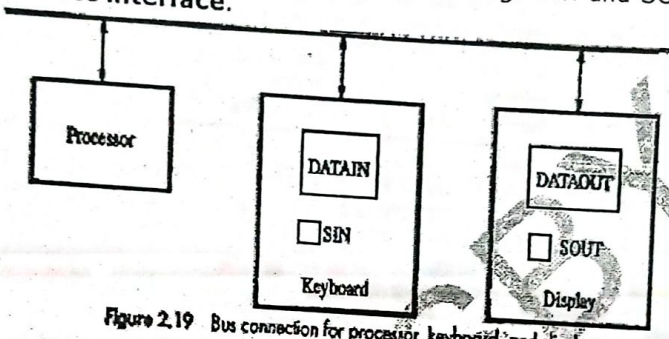


Figure 2.19 Bus connection for processor, keyboard, and display.

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.

MEMORY-MAPPED I/O

- 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.

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.

UNIT 2 : BASIC STRUCTURE OF COMPUTERS
INPUT/OUTPUT ORGANIZATION

Type your text
Type your text

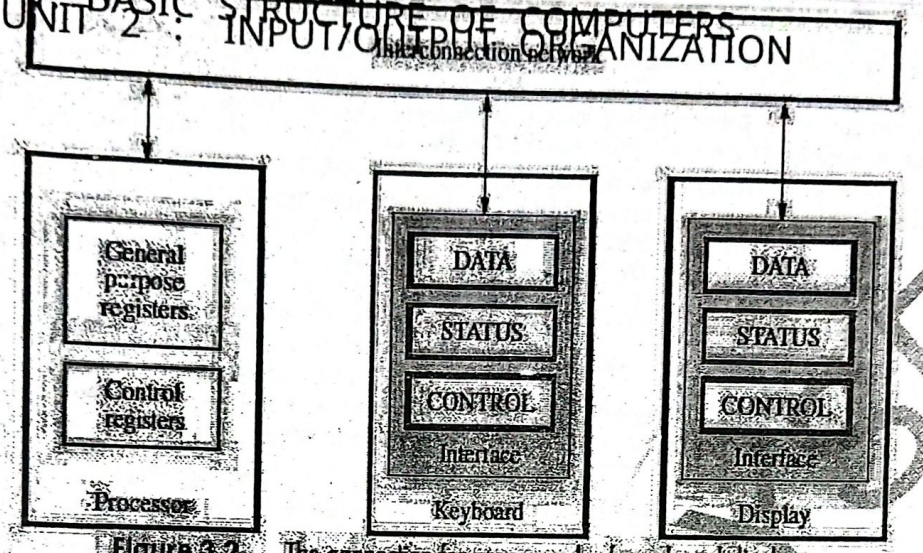


Figure 3.2 The connection for processor, keyboard, and display.

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).

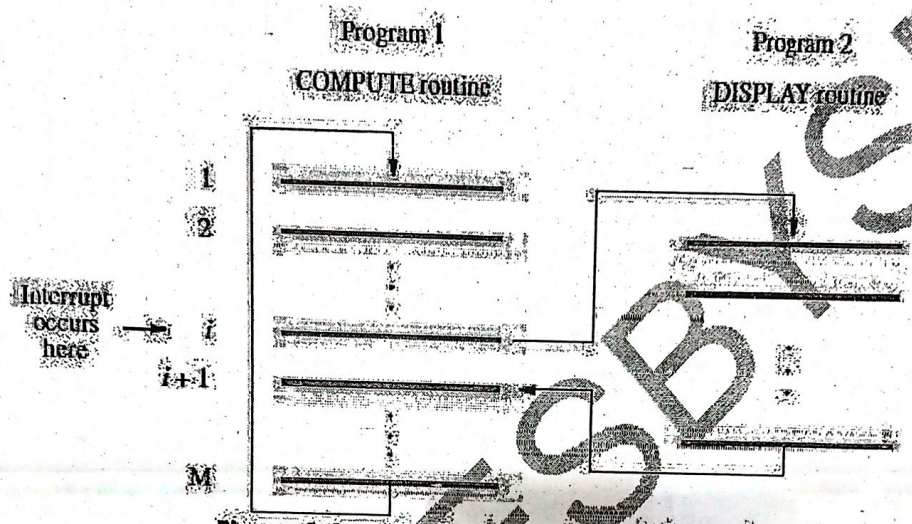


Figure 3.6 Transfer of control through the use of interrupts.

- 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.

$$INTR = INTR_1 + INTR_{2+} \ldots\ldots + INTR_n$$

- 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.

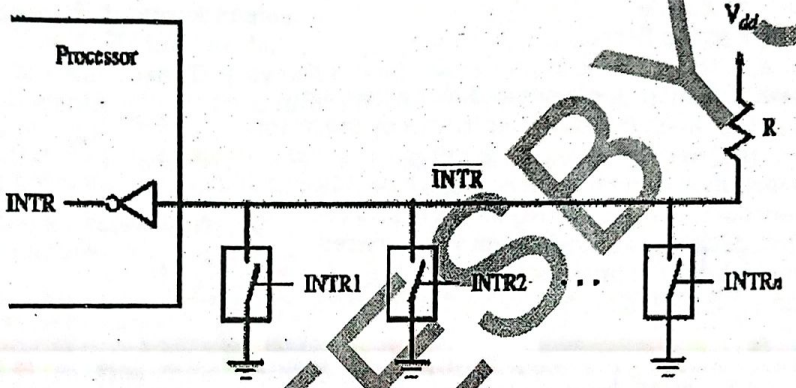


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

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.

COMPUTER ORGANIZATION

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.

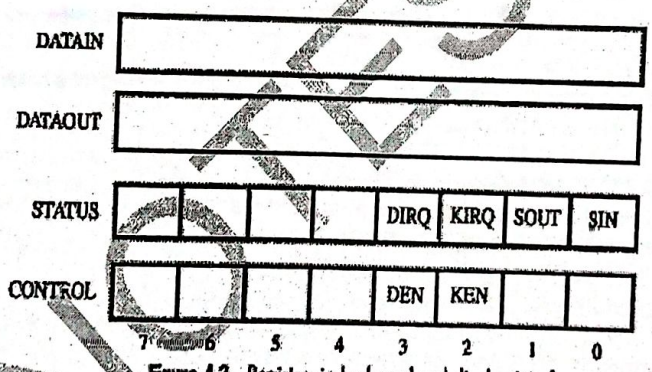


Figure 4.3 Registers in keyboard and display interfaces.

```
        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.**

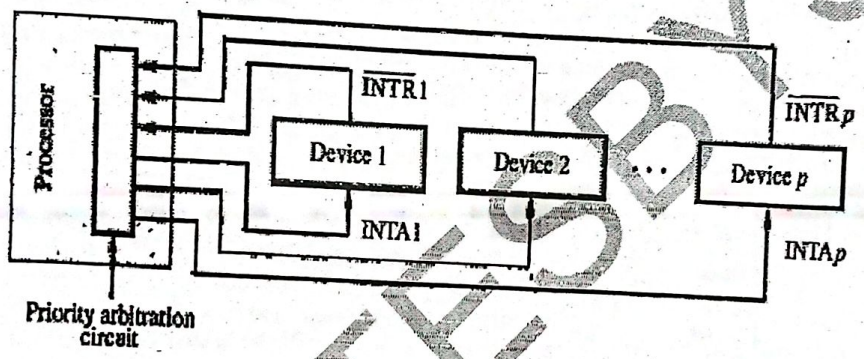


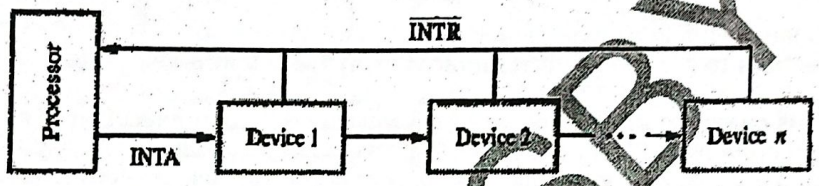
Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

SIMULTANEOUS REQUESTS

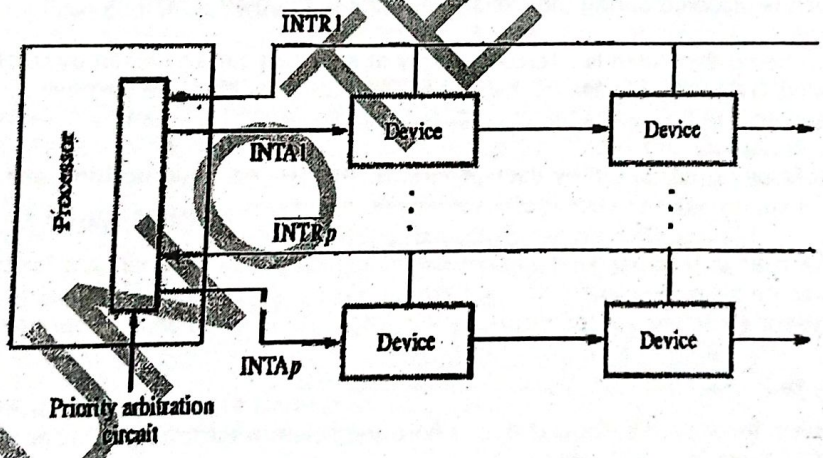
- The processor must have some mechanisms to decide which request to service when simultaneous requests arrive.
- INTR line is common to all devices (Figure 4.8a).
- INTA line is connected in a daisy-chain fashion.
- INTA signal propagates serially through devices.
- When several devices raise an interrupt-request, 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



(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.

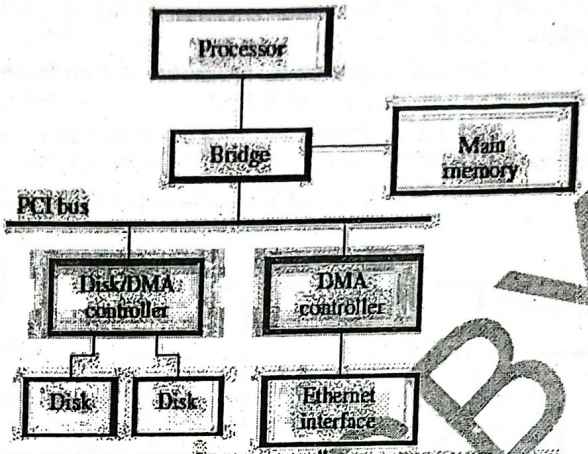


Figure 8.13 Use of DMA controllers in a computer system.

- 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.

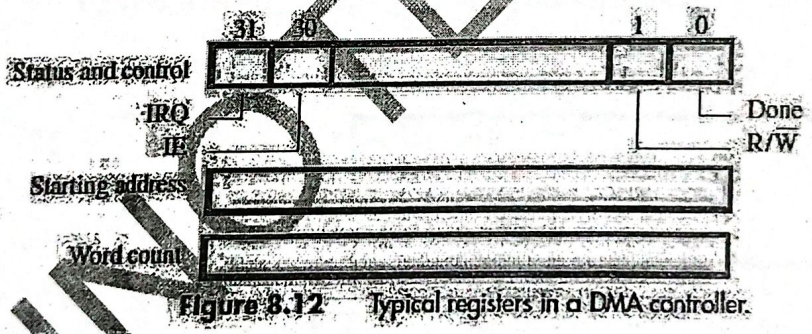


Figure 8.12 Typical registers in a DMA controller.

- 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).

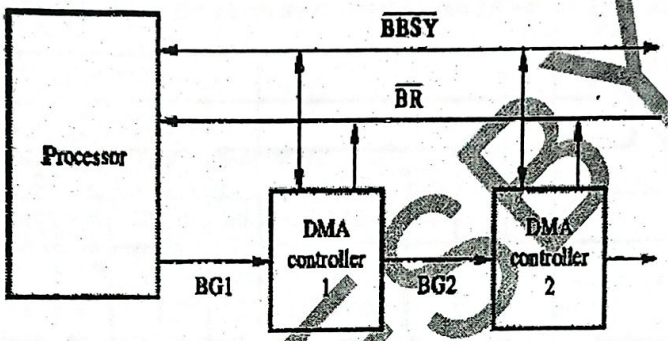


Figure 4.20 A simple arrangement for bus arbitration using a daisy chain.

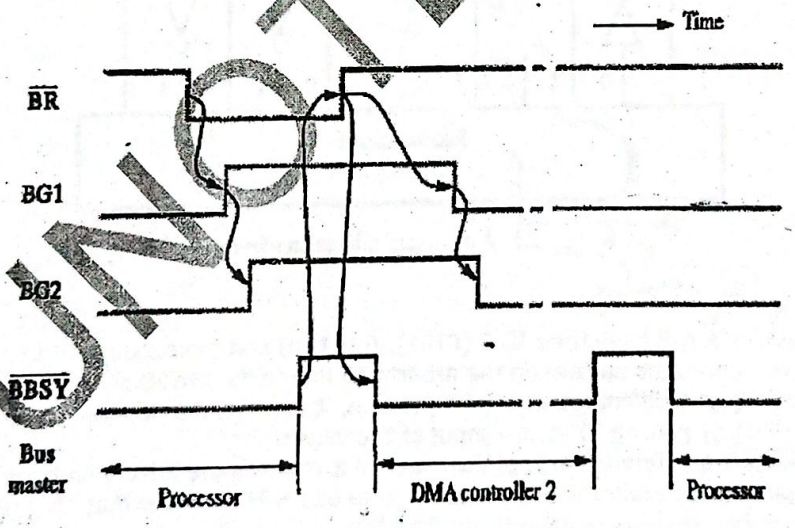


Figure 4.21 Sequence of signals during transfer of bus mastership for the devices in Figure 4.20.

- The timing diagram shows the sequence of events for the devices connected to the processor.
- 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).
  - → assert Start-Arbitration signal &
  - → place their 4-bit ID numbers on four open-collector lines $\overline{ARB0}$ through $\overline{ARB3}$.
- 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.

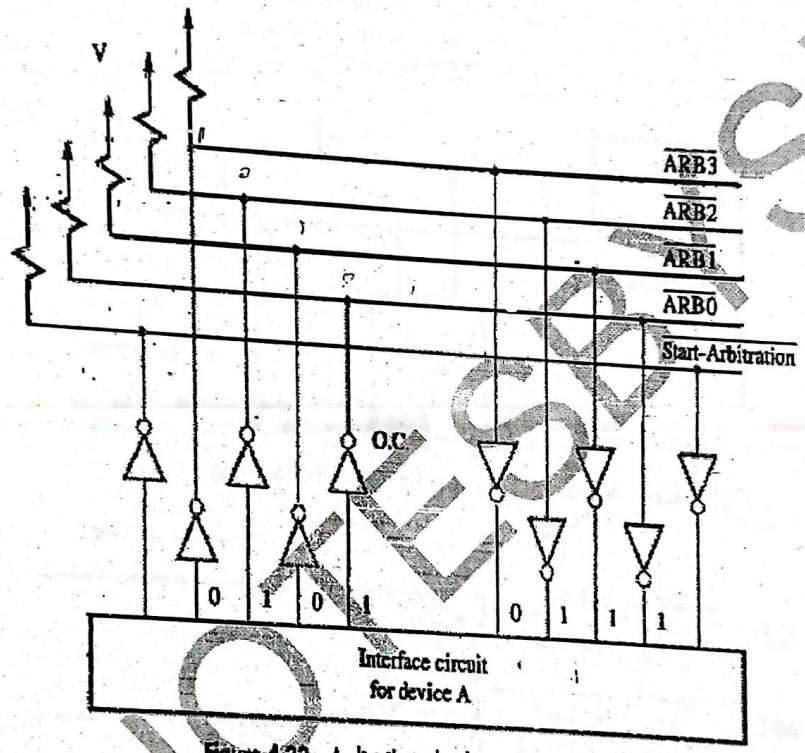


Figure 4.22 A distributed arbitration scheme.

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

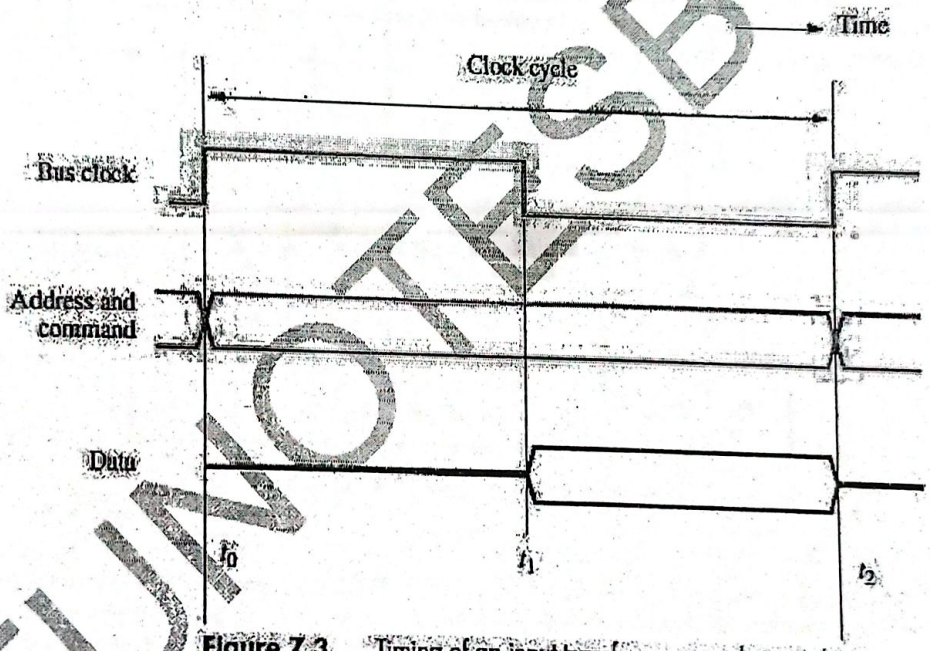


Figure 7.3 Timing of an input transfer on a synchronous bus.

A Detailed Timing Diagram for the Read-operation

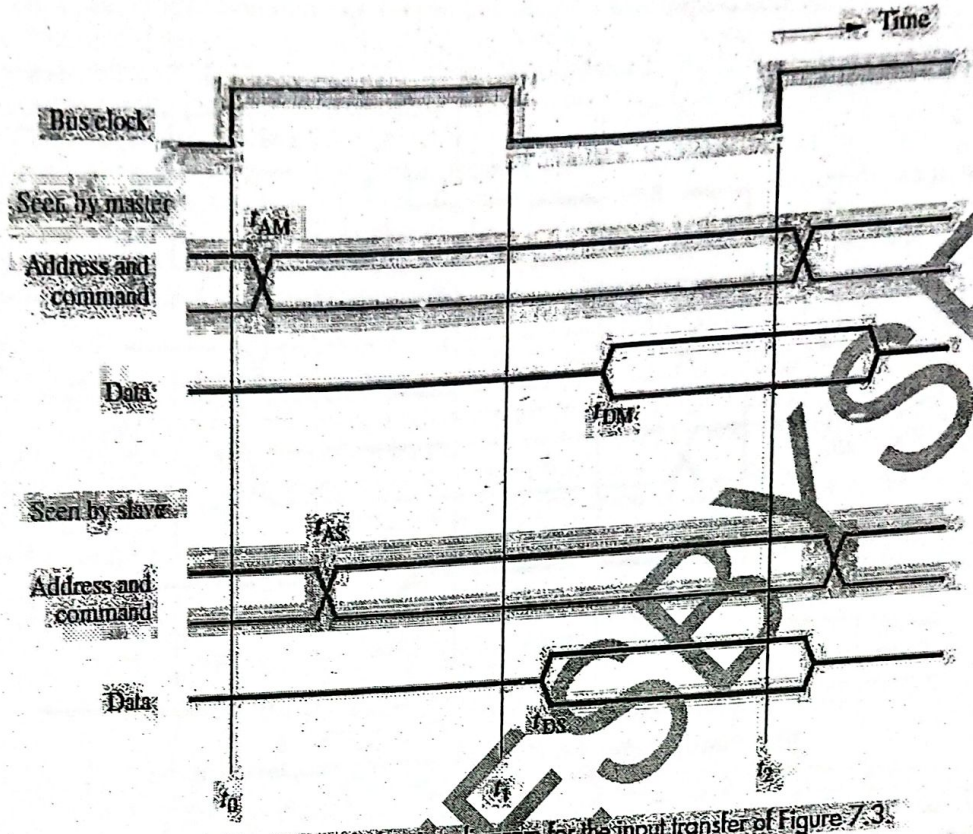


Figure 7.4 A detailed timing diagram for the input transfer of Figure 7.3.

- 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

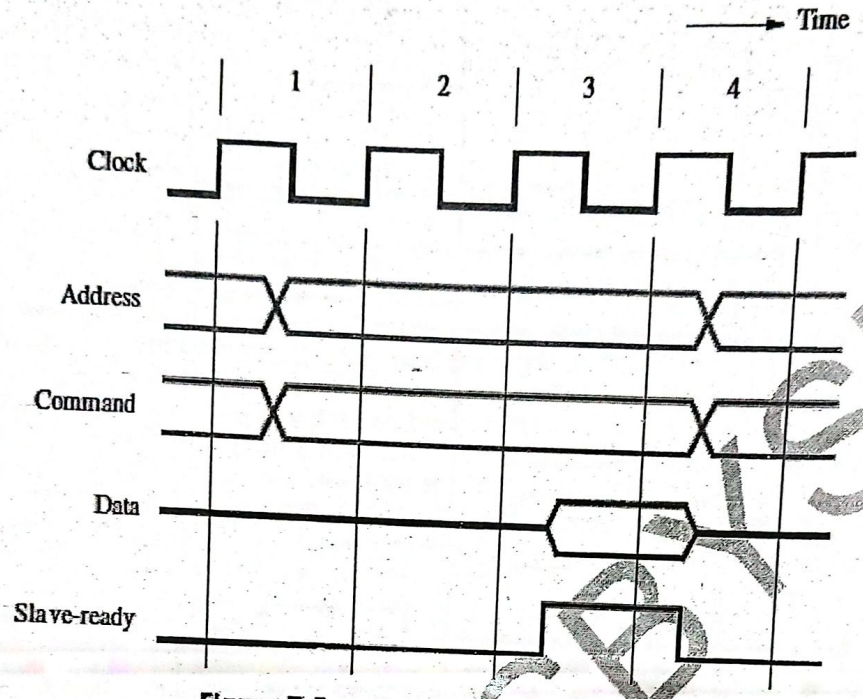


Figure 7.5 - An input transfer using multiple clock cycles.

- 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.

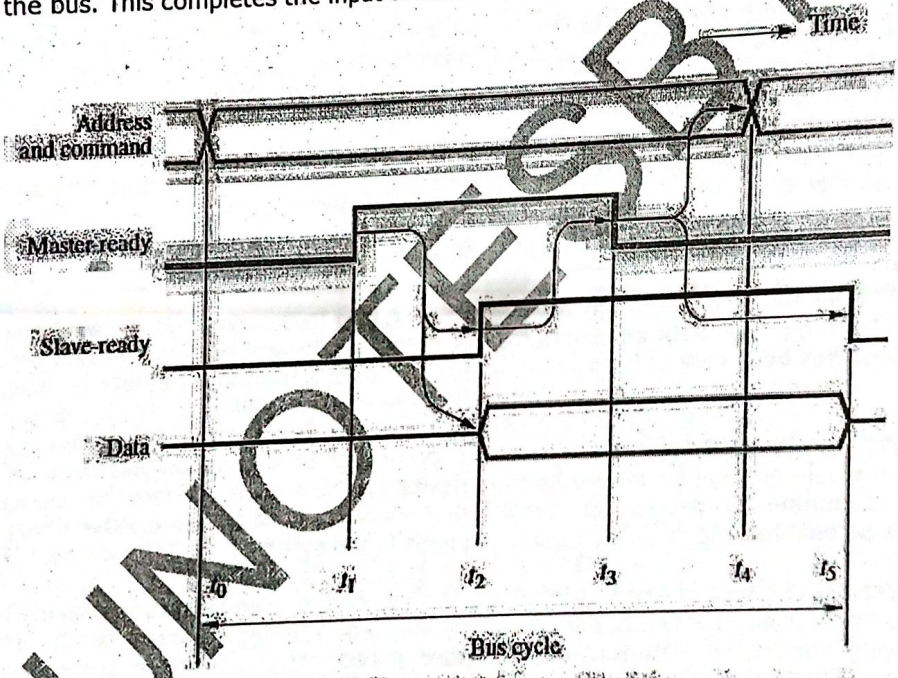


Figure 7.6 Handshake control of data transfer during an input operation.

- 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.

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.