Sahyadri College of Engineering and Management
Mangaluru
(Affiliated to Visvesvaraya Technological University, Belagavi)

COMPUTER ORGANIZATION (17CS34)

Prepared By:

Ganaraj K
Assistant Professor
Dept of ISE
Sahyadri College of Engineering and Management
Mangaluru

COMPUTER ORGANIZATION

[As per Choice Based Credit System (CBCS) scheme]
(Effective from the academic year 2015 -2016)
SEMESTER - III

Subject Code	**15CS34**	**IA Marks**	**20**
Number of Lecture Hours/Week	**04**	**Exam Marks**	**80**
Total Number of Lecture Hours	**50**	**Exam Hours**	**03**

CREDITS – 04

Course objectives:

This course will enable students to

- Understand the basics of computer organization: structure and operation of computers and their peripherals.
- Understand the concepts of programs as sequences or machine instructions.
- Expose different ways of communicating with I/O devices and standard I/O interfaces.
- Describe hierarchical memory systems including cache memories and virtual memory.
- Describe arithmetic and logical operations with integer and floating-point operands.
- Understand basic processing unit and organization of simple processor, concept of pipelining and other large computing systems.

Module -1	**Teaching Hours**
Basic Structure of Computers: Basic Operational Concepts, Bus Structures, Performance – Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement. **Machine Instructions and Programs:** Memory Location and Addresses, Memory Operations, Instructions and Instruction Sequencing, Addressing Modes, Assembly Language, Basic Input and Output Operations, Stacks and Queues, Subroutines, Additional Instructions, Encoding of Machine Instructions **Textbook 1: Ch 1: 1.3, 1.4, 1.6.1, 1.6.2, 1.6.4, 1.6.7. Ch 2: 2.2 to 2.10, 2.12**	**10Hours**
Module -2	
Input/Output Organization: Accessing I/O Devices, Interrupts – Interrupt Hardware, Enabling and Disabling Interrupts, Handling Multiple Devices, Controlling Device Requests, Exceptions, Direct Memory Access, Buses, Interface Circuits, Standard I/O Interfaces – PCI Bus, SCSI Bus, USB. **Textbook 1: Ch 4: 4.1, 4.2: 4.2.1 to 4.2.5, 4.4 to 4.7.**	**10 Hours**
Module – 3	
Memory System: Basic Concepts, Semiconductor RAM Memories, Read Only Memories, Speed, Size, and Cost, Cache Memories – Mapping Functions, Replacement Algorithms, Performance Considerations, Virtual Memories, Secondary Storage. **Textbook 1: Ch 5: 5.1 to 5.4, 5.5.1, 5.5.2, 5.6, 5.7, 5.9**	**10 Hours**
Module-4	

Arithmetic: Numbers, Arithmetic Operations and Characters, Addition and Subtraction of Signed Numbers, Design of Fast Adders, Multiplication of Positive Numbers, Signed Operand Multiplication, Fast Multiplication, Integer Division, Floating-point Numbers and Operations. **Textbook 1: Ch 2: 2.1, Ch 6: 6.1 to 6.7**	**10 Hours**
Module-5	
Basic Processing Unit: Some Fundamental Concepts, Execution of a Complete Instruction, Multiple Bus Organization, Hard-wired Control, Micro programmed Control. **Embedded Systems and Large Computer Systems:** Examples of Embedded Systems, Processor chips for embedded applications, Simple Microcontroller. **The structure of General-Purpose Multiprocessors**. **Textbook 1: Ch 7: 7.1 to 7.5, Ch 9:9.1 to 9.3, Ch 12:12.3**	**10 Hours**

Course outcomes:

After studying this course, students will be able to:

- Acquire knowledge of
  - The basic structure of computers & machine instructions and programs, Addressing Modes, Assembly Language, Stacks, Queues and Subroutines.
  - Input/output Organization such as accessing I/O Devices, Interrupts.
  - Memory system basic Concepts, Semiconductor RAM Memories, Static memories, Asynchronous DRAMS, Read Only Memories, Cache Memories and Virtual Memories.
  - Some Fundamental Concepts of Basic Processing Unit, Execution of a Complete Instruction, Multiple Bus Organization, Hardwired Control and Micro programmed Control.
  - Pipelining, embedded and large computing system architecture.
- Analyse and design arithmetic and logical units.
- Apply the knowledge gained in the design of Computer.
- Design and evaluate performance of memory systems
- Understand the importance of life-long learning

Graduate Attributes (as per NBA)

1. Engineering Knowledge
2. Problem Analysis
3. Life-Long Learning

Question paper pattern:

The question paper will have ten questions.
There will be 2 questions from each module.
Each question will have questions covering all the topics under a module.
The students will have to answer 5 full questions, selecting one full question from each module.

Text Books:

1. Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill, 2002.

Reference Books:

1. William Stallings: Computer Organization & Architecture, 9th Edition, Pearson, 2015.

Module 1: BASIC STRUCTURE OF COMPUTERS

TYPES OF COMPUTERS

Desktop Computers

- These are most commonly used computers in home, schools and offices.
- This has
  - → processing- & storage-units
  - → video & audio output-units
  - → Keyboard & mouse input-units.

Notebook Computers (Laptops)

- This is a compact version of a personal-computer (PC) made as a portable-unit.

Workstations

- These have more computational-power than PC.

Enterprise Systems (Mainframes)

- These are used for business data-processing.
- These have large computational-power and larger storage-capacity than workstations.
- These are referred to as
  - → server at low-end and
  - → Super-computers at high end.

Servers

- These have large database storage-units and can also execute requests from other computers.
- These are used in banks & educational institutions.

Super Computers

- These are used for very complex numerical-calculations.
- These are used in weather forecasting, aircraft design and military applications.

FUNCTIONAL UNITS

- A computer consists of 5 functionally independent main parts: 1)input, 2)memory,3)arithmetic & logic, 4)output and 5)control units.

Input Unit

- The computer accepts the information in the form of program & data through an input-device.
  Eg: keyboard
- Whenever a key is pressed, the corresponding letter/digit is automatically translated into its corresponding binary-code and transmitted over a cable to either the memory or the processor.

Memory Unit

- This unit is used to store programs & data.
- There are 2 classes of storage:
  1) Primary-storage is a fast-memory that operates at electronic-speed. Programs must be stored in the memory while they are being executed.
  2) Secondary-storage is used when large amounts of data & many programs have to be stored. Eg: magnetic disks and optical disks(CD-ROMs).
- The memory contains a large number of semiconductor storage cells(i.e. flip-flops), each capable of storing one bit of information.
- The memory is organized so that the contents of one word can be stored or retrieved in one basic operation.

• Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called RAM (Random Access Memory).

ALU (Arithmetic & Logic Unit)

• This unit is used for performing arithmetic & logical operations.

• Any arithmetic operation is initiated by bringing the required operand into the processor (i.e. registers), where the operation is performed by the ALU.

Output Unit

• This unit is used to send processed-results to the outside world.
Eg: printer, graphic displays etc.

Control Unit

• This unit is used for controlling the activities of the other units (such as memory, I/O device).

• This unit sends control-signals (read/write) to other units and senses their states.

• Data transfers between processor and memory are also controlled by the control-unit through timing-signals.

• Timing-signals are signals that determine when a given action is to take place.

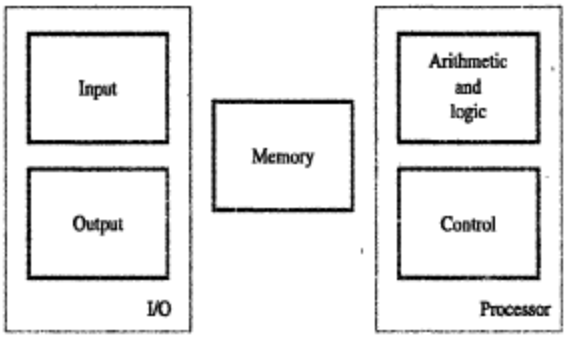


Figure 1.1 Basic functional units of a computer.

BASIC OPERATIONAL CONCEPTS

- The processor contains ALU, control-circuitry and many registers.
- The instruction-register(IR) holds the instruction that is currently being executed.
- The instruction is then passed to the control-unit, which generates the timing-signals that determine when a given action is to take place
- The PC(Program Counter) 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 processor also contains „n" general-purpose registers R0 through Rn-1.
- The MAR (Memory Address Register) holds the address of the memory-location to be accessed.
- The MDR (Memory Data Register) contains the data to be written into or read out of the addressed location.

Following are the steps that take place to execute an instruction

- The address of first instruction(to be executed) gets loaded into PC.
- The contents of PC(i.e. address) are transferred to the MAR & control-unit issues Read signal to memory.
- After certain amount of elapsed time, the first instruction is read out of memory and placed into MDR.
- Next, the contents of MDR are transferred to IR. At this point, the instruction can be decoded & executed.
- 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.
- Likewise required number of operands is fetched into processor.
- Finally, ALU performs the desired operation.
- If the result of this operation is to be stored in the memory, then the result is sent to the MDR.
- The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.
- At some point during execution, contents of PC are incremented to point to next instruction in the program. [The instruction is a combination of opcode and operand].

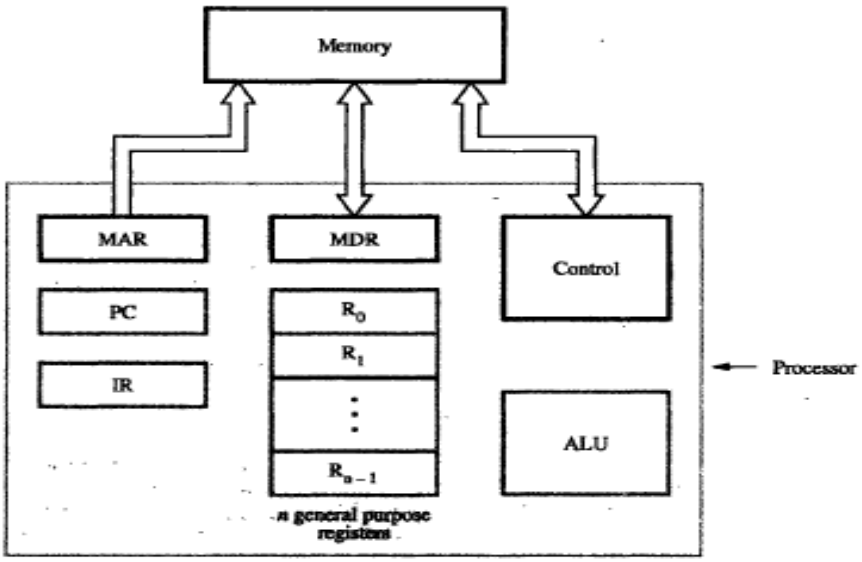


Figure 1.2 Connections between the processor and the memory.

BUS STRUCTURE

• A bus is a group of lines that serves as a connecting path for several devices.
• Bus must have lines for data transfer, address & control purposes.
• 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.
• Main advantage of single bus: Low cost and flexibility for attaching peripheral devices.
• Systems that contain multiple buses achieve more concurrency in operations by allowing 2 or more transfers to be carried out at the same time. Advantage: better performance. Disadvantage: increased cost.
• The devices connected to a bus vary widely in their speed of operation. To synchronize their operational speed, the approach is to include buffer registers with the devices to hold the information during transfers.

Buffer registers prevent a high-speed processor from being locked to a slow I/O device during a sequence of data transfers.

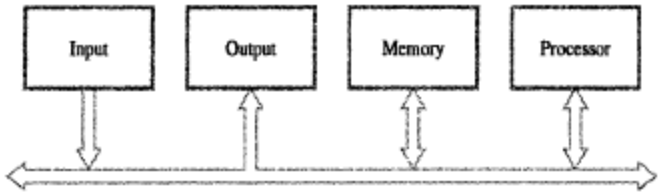


Figure 1.3 Single-bus structure.

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=1/P which is measured in cycles per second.
• Cycles per second is also called hertz(Hz)

BASIC PERFORMANCE EQUATION

• Let T=processor time required to
execute 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.

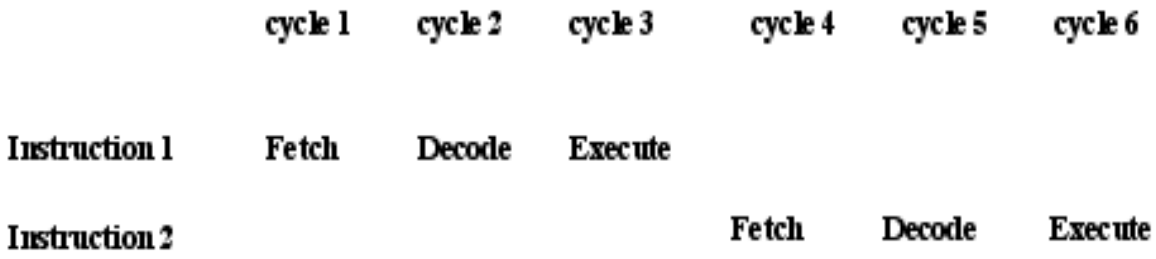
→ The value of N is reduced if source program is compiled into fewer machine instructions.
→ The value of S is reduced if instructions have a smaller number of basic steps to perform.
→ The value of R can be increased by using a higher frequency clock.

• Care has to be taken while modifying the values since changes in one parameter may affect the other.

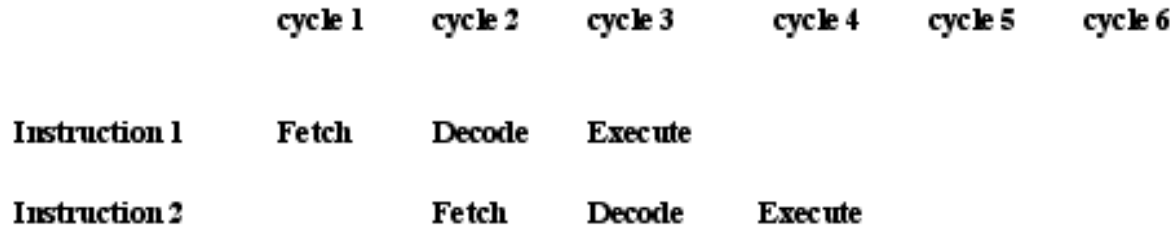
PIPELINING & SUPERSCALAR OPERATION

- Normally, the computer executes the instruction in sequence one by one. An improvement in performance can be achieved by overlapping the execution of successive instructions using a technique called *pipelining.*
- Consider the instruction
  Add R1,R2,R3;instruction 1
  Move R4,R5 ;instruction 2

Let us assume that both operations take 3 clock cycles each for completion.



As shown in above figure, 6 clock cycles are required to complete two operations.



• As shown in above figure, if we use pipelining & prefetching, only 4 cycles are required to complete same two operations.
• While executing the Add instruction, the processor can read the Move instruction from memory.
• In the ideal case, if all instructions are overlapped to the maximum degree possible, execution proceeds at the rate of one instruction completed in each clock cycle.
• A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor i.e. multiple functional units can be used to execute different instructions parallely. This mode of operation is known as *superscalar execution.*
• With Superscalar arrangement, it is possible to complete the execution of more than one instruction in every clock cycle.

PERFORMANCE MEASUREMENT

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

The SPEC rating is computed as follows

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

- If SPEC rating=50 means that the computer under test is 50times as fast as reference computer.
- The test is repeated for all the programs in the SPEC suite, and the geometric mean of the results is computed.

Let $SPEC_i$ be the rating for program i in the suite. The 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=number of programs in the suite

INSTRUCTION SET: CISC AND RISC

Architectural characteristics	***CISC***	***RISC***
Instruction formats	instructions with variable formats	instructions with fixed format and most are register based instruction
addressing modes	12-24	limited to 3-5
General purpose register and cache design	8-24 GPRs. Mostly with a unified cache for instruction and data	large number of GPRs with mostly split data cache and instruction cache
Clock rate & CPI	33-50 Mhz with CPI between 2 and 15	50-150Mhz with 1 clock cycle for almost all instruction and an average CPI<15
CPU control	mostly microcoded using control memory	mostly hardwired without control memory

Module 1(CONT.): MACHINE INSTRUCTIONS & PROGRAMS

MEMORY LOCATIONS & ADDRESSES

• The memory consists of many millions of storage cells (flip-flops), each of which can store a bit of information having the value 0 or 1 (Figure 2.5).

• 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 64bits.

• A unit of 8 bits is called a byte.

• Accessing the memory to store or retrieve a single item of information (either a word or a 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=number 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 (16MB).

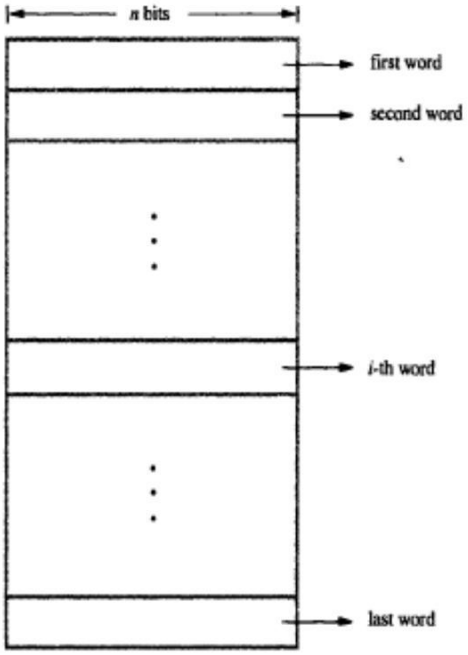


Figure 2.5 Memory words.

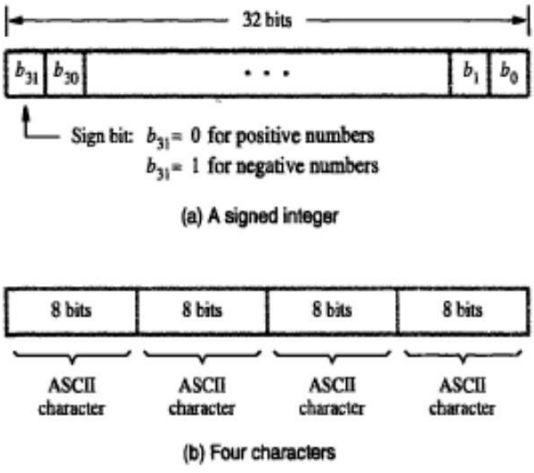


Figure 2.6 Examples of encoded information in a 32-bit word.

• Characters can be letters of the alphabet, decimal digits, punctuation marks and so on.
• Characters are represented by codes that are usually 8 bits long. E.g. ASCII code
• The three basic information quantities are: bit, byte and word.
• A byte is always 8 bits, but the word length typically ranges from 1 to 64 bits.
• It is impractical to assign distinct addresses to individual bit locations in the memory.

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 AND LITTLE-ENDIAN ASSIGNMENTS

• There are two ways in which byte addresses are arranged.
    1) Big-endian assignment: lower byte addresses are used for the more significant bytes of the word (Figure 2.7).
    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.

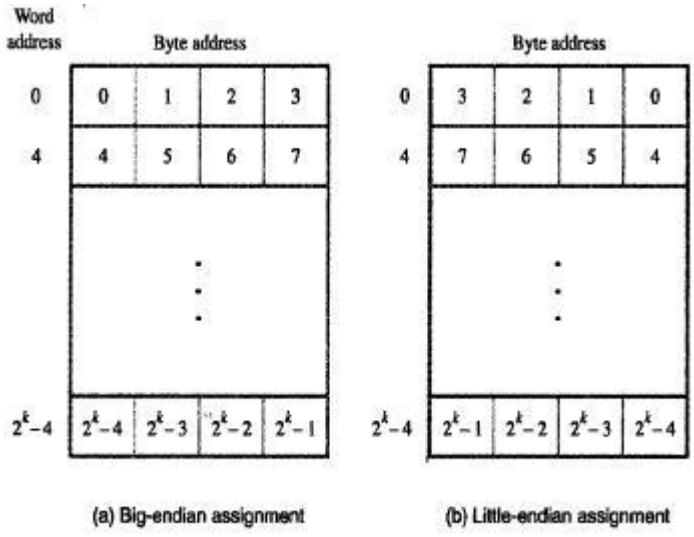


Figure 2.7 Byte and word addressing.

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 . . . . . And for a word length of 64, 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

→ a special control character with the meaning "end of string" can be used as the last character in the string, or

→ a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

MEMORY OPERATIONS

• Two basic operations involving the memory are: Load(Read/Fetch) and Store(Write).

• The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged.

• The 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 processor register to the specified memory location. This will destroy the original contents of that memory location.

• The 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 processor 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)

• We identify a memory location by a symbolic name (in uppercase alphabets).

For example, LOC, PLACE, NUM etc indicate memory locations.R0, R5 etc indicate processor register. DATAIN, OUTSTATUS etc indicate I/O registers.

• For example,

R<-[LOC] means that the contents of memory location LOC are transferred into processor register R1 (The contents of a location are denoted by placing square brackets around the name of the location). *R3<-[R1]+[R2]* indicates the operation that adds the contents of registers R1 and R2 ,and then places their sum into register R3.

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

ASSEMBLY LANGUAGE NOTATION

• To represent machine instructions and programs, assembly language format can be used.

• For example,

Move LOC, R1; This instruction transfers data from memory-location LOC to processor-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; This instruction adds 2 numbers contained in processor-registers R1 and R2, and places their sum in R3.

• A computer performs its task according to the program stored in memory. A program is a collection of instructions which tell the processor to perform a basic operation like addition, reading from keyboard etc.
• Possible locations that may be involved in data transfers are memory locations, processor registers or registers in the I/O subsystem.

BASIC INSTRUCTION TYPES

• *C=A+B;* This statement is a command to the computer to add the current values of the two variables A and B, and to assign the sum to a third variable C.
• When the program is compiled, each variable is assigned a distinct address in memory.
• The contents of these locations represent the values of the three variables
• The statement *C<-[A]+[B]* indicates that the contents of memory locations A and B are fetched from memory, transferred to the processor, sum is computed and then result is stored in memory location C.

Three-Address Instruction

• The instruction has general format

Operation Source1, Source2, Destination

• For example, *Add A, B, C;* operands A and B are called the source operands, C is called the destination operand, and Add is the operation to be performed.

Two-Address Instruction

• The instruction has general format

Operation Source, Destination

• For example, *Add A, B;* performs the operation B<-[A]+[B].
• When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that operand B is both a source and a destination.
• The operation C<-[A]+[B] can be performed by the two-instruction sequence

Move B, C
Add A, C

One-Address Instruction

• The instruction has general format

Operation Source/Destination

• For example, *Add A ;* Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator.
• *Load A;* This instruction copies the contents of memory location A into the accumulator and *Store A;* This instruction copies the contents of the accumulator into memory location A.

The operation C<-[A]+[B] can be performed by executing the sequence of instructions

Load A
Add B
Store C

• The operand may be a source or a destination depending on the instruction. In the Load instruction, address A specifies the source operand, and the destination location, the accumulator, is implied. On the other hand, C denotes the destination location in the Store instruction, whereas the source, the accumulator, is implied.

Zero-Address Instruction

• The locations of all operands are defined implicitly. The operands are stored in a structure called pushdown stack. In this case, the instructions are called zero-address instructions.
• Access to data in the registers is much faster than to data stored in memory locations because the registers are inside the processor.

• 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 processor registers, the C=A+B task can be performed by the instruction sequence

INSTRUCTION EXECUTION & STRAIGHT LINE SEQUENCING

• The program is executed as follows:

1) Initially, the address of the first instruction is loaded into PC (Program counter is a register which holds the address of the next instruction to be executed)

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* (Figure 2.8)

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

4) Executing given instruction is a two-phase procedure.

i) In fetch phase, the instruction is fetched from the memory location (whose address is in the PC) and placed in the IR of the processor

ii) In 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.

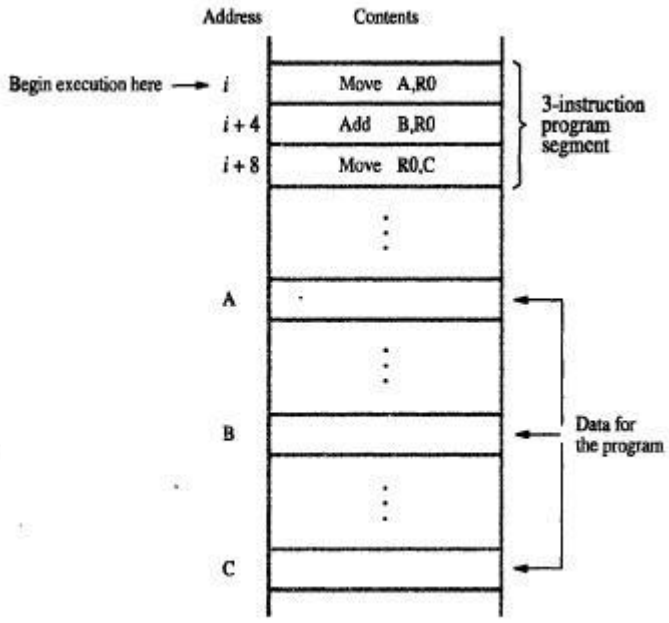


Figure 2.8 A program for C ← [A] + [B].

BRANCHING

• Consider the task of adding a list of n numbers (Figure 2.10).

• The loop is a straight line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch>0.

• During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to R0.

• Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program.

• Within the body of the loop, 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.

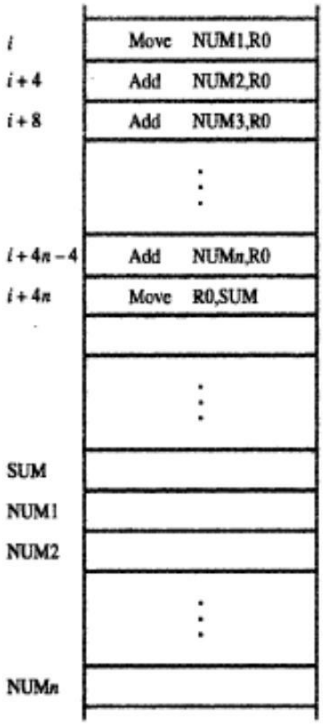


Figure 2.9 A straight-line program for adding *n* numbers.

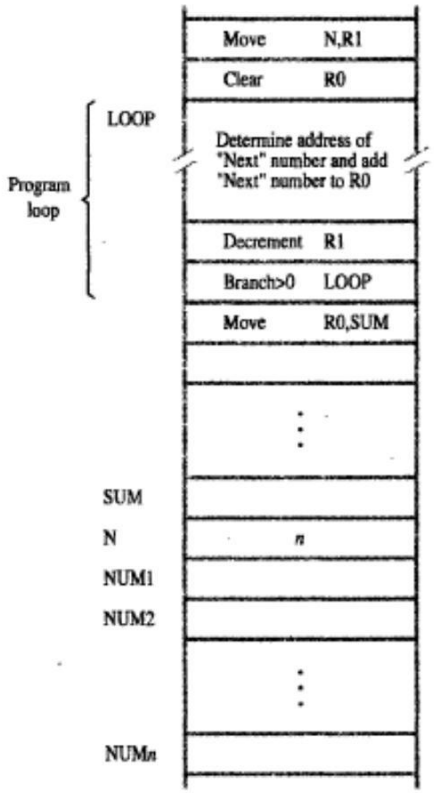


Figure 2.10 Using a loop to add *n* numbers.

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
→ N (negative) set to 1 if the result is negative, otherwise cleared to 0
→ Z (zero) set to 1 if the result is 0; otherwise, cleared to 0
→ V (overflow) set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
→ C (carry) set to 1 if a carry-out results from the 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) (LOC)	EA = [Ri] 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

• Variables & constants are the simplest data-types and are found in almost every computer program.
• In assembly language, a variable is represented by allocating a register (or memory-location) to hold its value. Thus, the value can be changed as needed using appropriate instructions.

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.
• 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

• In this case, the instruction does not give the operand or its address explicitly; instead, it provides information from which the memory-address of the operand can be determined. We refer to this address as the *effective address(EA)* of the operand.

Indirect Mode

• The EA of the operand is the contents of a register(or memory-location) whose address appears in the instruction.

• The register (or memory-location) that contains the address of an operand is called a *pointer*. {The indirection is denoted by ( ) sign around the register or memory-location}.

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

* To execute the Add instruction in fig (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 (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

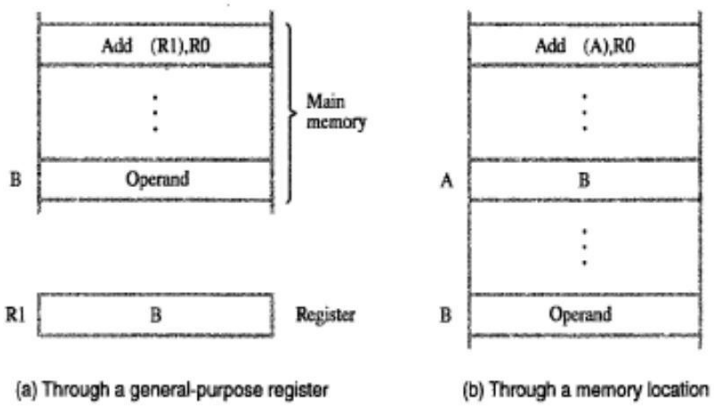


Figure 2.11 Indirect addressing.

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.

• 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 contained in the instruction Ri=the name of the index register

• 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.
• In an assembly language program, the constant X may be given either
  → as an explicit number or
  → as a symbolic-name representing a numerical value.

* 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.
* 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.

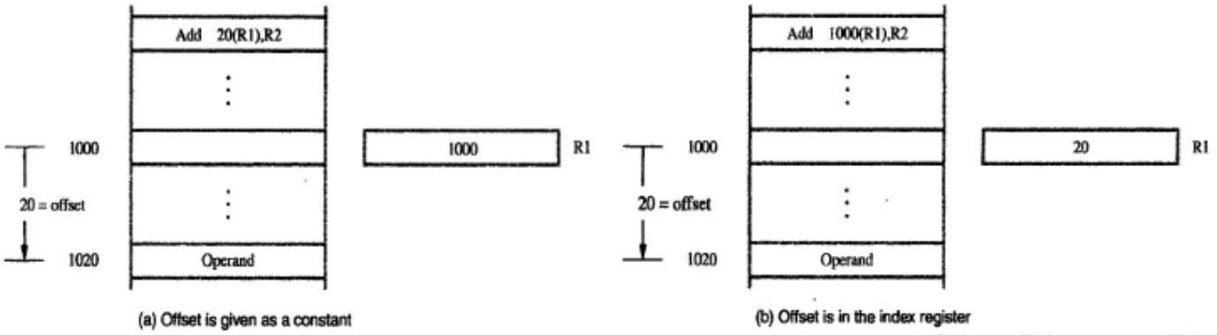


Figure 2.13 Indexed addressing.

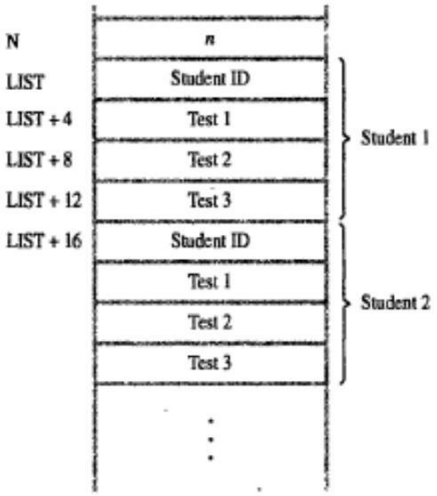


Figure 2.14 A list of students' marks.

```
        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 an exception: 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

• The following 2 modes are useful for accessing data items in successive locations in the memory.

Auto-increment Mode

• The effective-address of operand is the 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

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.

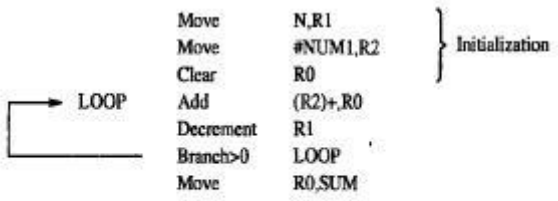


Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

ASSEMBLY LANGUAGE

• 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*.
• Move instruction is written is

MOVE R0,SUM ;The mnemonic MOVE represents the binary pattern, or OP code, for the operation performed by the instruction.

• The instruction

ADD #5,R3 ;Adds the number 5 to the contents of register R3 and puts the result back into register R3.

ASSEMBLER DIRECTIVES

• EQU informs the assembler about the value of an identifier (Figure: 2.18).

Ex*: SUM EQU 200* ; This statement informs the assembler that the name SUM should be replaced by the value 200 wherever it appears in the program.

• ORIGIN tells the assembler about the starting-address of memory-area to place the data block.

• DATAWORD directive tells the assembler to load a value (say 100) into the location (say 204). Ex: *N DATAWORD 100*

• RESERVE directive declares that a memory-block of 400 bytes is to be reserved for data and that the name NUM1 is to be associated with address 208.

Ex: *NUM1 RESERVE 400*

• 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.

	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.

GENERAL FORMAT OF A STATEMENT

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

Label Operation Operands Comment

→Label is an optional name associated with the memory-address where the machine language instruction produced from the statement will be loaded.

→ The Operation field contains the OP-code mnemonic of the desired instruction or assembler → The

Operand field contains addressing information for accessing one or more operands, depending on the type of instruction.

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 the address given in the ORIGIN directive.

→ inserts constants that may be given in DATAWORD directives.

→ reserves memory-space as requested by RESERVE directives.

• 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. Thus, when a name appears a second time, it is replaced with its value from the table. Hence, such an assembler is called a *two-pass assembler.*

• 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 and

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

→The Comment field is used for documentation purposes to make the program easier to understand.

BASIC INPUT/OUTPUT OPERATIONS

• Consider the problem of moving a character-code from the keyboard to the processor. For this transfer, buffer-register(DATAIN) & a status control flags(SIN) are used.

• Striking a key stores the corresponding character-code in an 8-bit buffer-register(DATAIN) associated with the keyboard (Figure: 2.19).

• To inform the processor that a valid character is in DATAIN, a SIN is set to 1.

• A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN.

• When the character is transferred to the processor, SIN is automatically cleared to 0.

• If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.

• An analogous process takes place when characters are transferred from the processor to the display. A buffer-register, DATAOUT, and a status control flag, SOUT are used for this transfer.

• When SOUT=1, the display is ready to receive a character.

• The transfer of a character to DATAOUT clears SOUT to 0.

• The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a *device interface.*

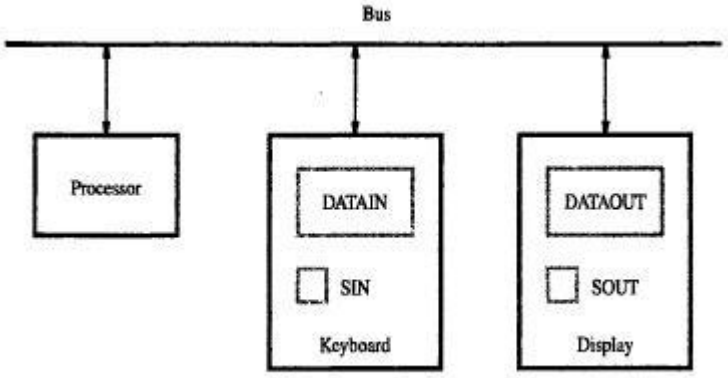


Figure 2.19 Bus connection for processor, keyboard, and display.

• Following is a 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.
```

MEMORY-MAPPED I/O

• Some address values are used to refer to peripheral device buffer-registers such as DATAIN and 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 list of data elements with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom (Figure: 2.21).

• The terms push and pop are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

• 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 SP (Stack Pointer).

• If we assume a byte-addressable memory with a 32-bit word length,

→The push operation can be implemented as

Subtract #4,SR

Move NEWITEM,(SP)

→ 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	#2000,SP	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.
	Branch>0	EMPTYERROR	
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

(a) Routine for a safe pop operation

SAFEPUSH	Compare	#1500,SP	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.
	Branch≤0	FULLERROR	
	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.

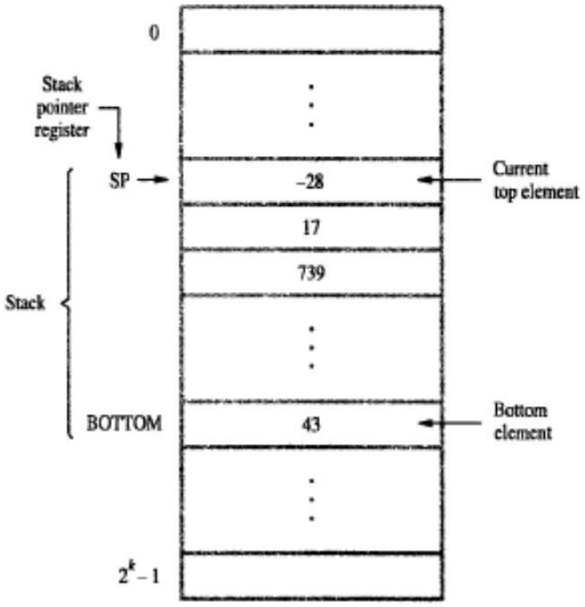


Figure 2.21 A stack of words in the memory.

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) A single pointer is needed to point to the top of the stack at any given time.
On the other hand, both does of a queue move to higher addresses as data are added at the back and removed from the front. So, two pointers are needed to keep track of the two does of the queue.

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.*

• The program branches to a subroutine with a Call instruction (Figure: 2.24).

• Once the subroutine is executed, the calling-program must resume execution starting from the instruction immediately following the Call instructions i.e. control is to be transferred back to the calling-program. This is done by executing a Return instruction at the end of the subroutine.

• 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.

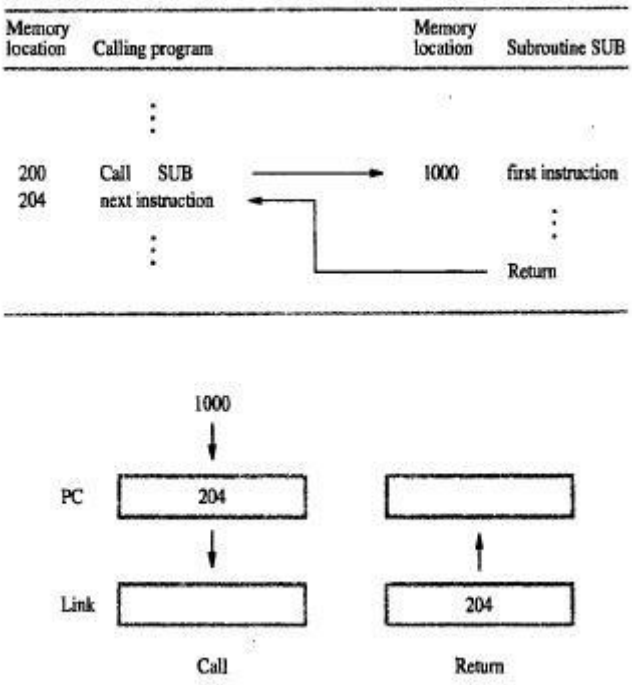


Figure 2.24 Subroutine linkage using a link register.

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.

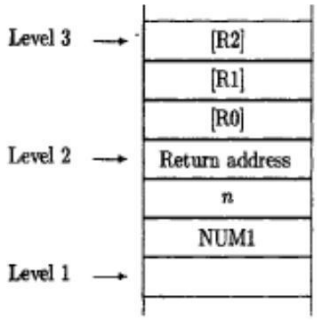
STACK FRAME

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

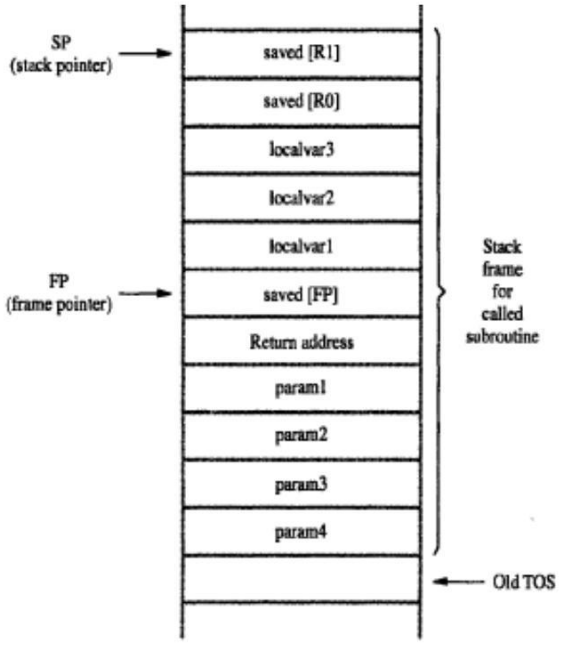


Figure 2.27 A subroutine stack frame example.

- 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 old TOS.
• 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.
• Consider the following program to illustrate stack frames for nested subroutines (refer fig no. 2.28 from text book).

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 needed, one for shifting left(LShiftL) and another for shifting right(LShiftR).
• These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.

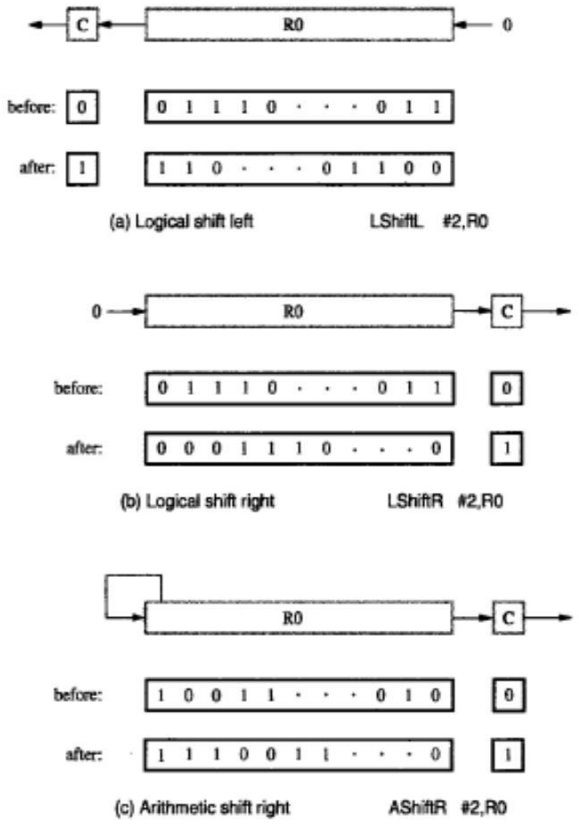


Figure 2.30 Logical and arithmetic shift instructions.

```
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.
MoveByte  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 are simply rotated.
In the other version, the rotation includes the C flag.

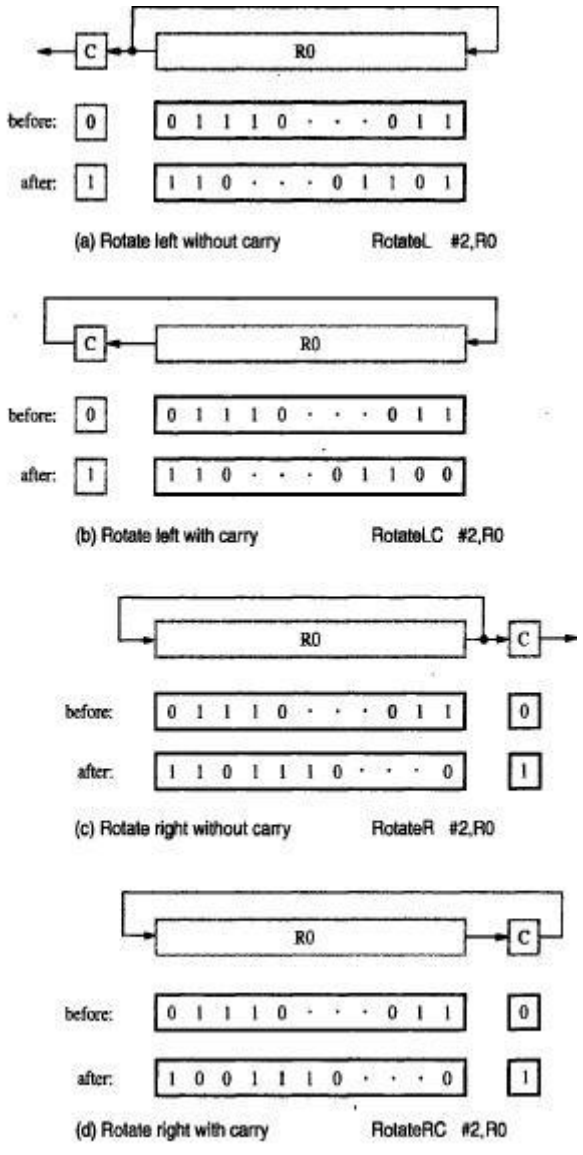


Figure 2.32 Rotate instructions.

Multiplication And Division:

Multiply R_i,R_j
Rj= [Ri]*[Rj]

Division Ri,Rj
Rj=[Ri]/[Rj]

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 processor registers.

Ex: Add R1,R2,R3

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

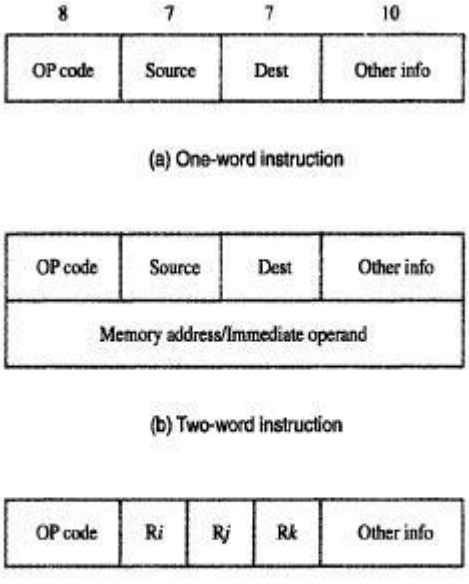


Figure 2.39 Encoding instructions into 32-bit words.

MODULE 2 : INPUT/OUTPUT ORGANIZATION

ACCESSING I/O DEVICES

• There are 2 ways to deal with I/O devices (Figure 4.1).

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 reads data from DATAIN(input-buffer associated with keyboard) & stores them into processor-register R0.

2) In *I/O mapped I/O*, memory and I/O address-spaces are different.
   - 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.

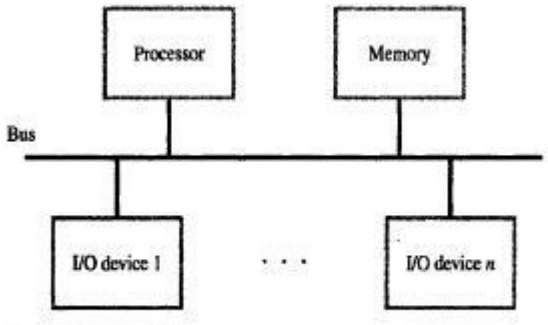


Figure 4.1 A single-bus structure.

I/O Interface for an Input Device

• Address decoder: decodes address sent on bus, so as to enable input-device (Figure 4.2).
• Data register: holds data being transferred to or from the processor.
• Status register: contains information relevant to operation of I/O device.
• Address decoder, data- and status-registers, and control-circuitry required to coordinate I/O transfers constitute device's interface-circuit.

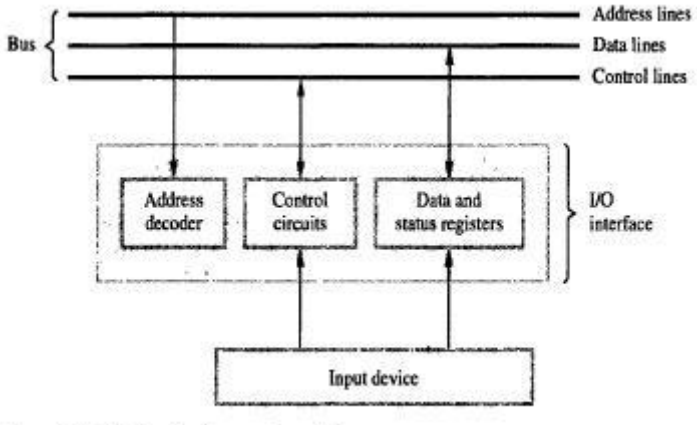


Figure 4.2 I/O interface for an input device.

MECHANISMS USED FOR INTERFACING I/O DEVICES

1) Program Controlled I/O
   - Processor repeatedly checks a status-flag to achieve required synchronization between processor & input/output device. (We say that the processor polls the device).
   - Main drawback: The processor wastes its time in checking the status of the device before actual data transfer takes place.

2) Interrupt I/O
   - Synchronization is achieved by having I/O device send a special signal over bus whenever it is ready for a data transfer operation.

3) Direct Memory Access (DMA)
   - This involves having the device-interface transfer data directly to or from the memory without continuous involvement by the processor.

INTERRUPTS

- I/O device initiates the action instead of the processor. This is done by sending a special hardware signal to the processor called as *interrupt*(INTR), on the interrupt-request line.
- The processor can be performing its own task without the need to continuously check the I/O device.
- When device gets ready, it will "alert" the processor by sending an interrupt-signal (Figure 4.5).
- The routine executed in response to an interrupt-request is called ISR(Interrupt Service Routine).
- Once the interrupt-request signal comes from the device, the processor has to inform the device that its request has been recognized and will be serviced soon. This is indicated by a special control signal on the bus called *interrupt-acknowledge*(INTA).

Difference between subroutine & ISR

- A subroutine performs a function required by the program from which it is called.
  However, the ISR may not have anything in common with the program being executed at the time the interrupt-request is received. Before starting execution of ISR, any information that may be altered during the execution of that routine must be saved. This information must be restored before the interrupted-program resumed.
- Another difference is that an interrupt is a mechanism for coordinating I/O transfers whereas a subroutine is just a linkage of 2 or more function related to each other.

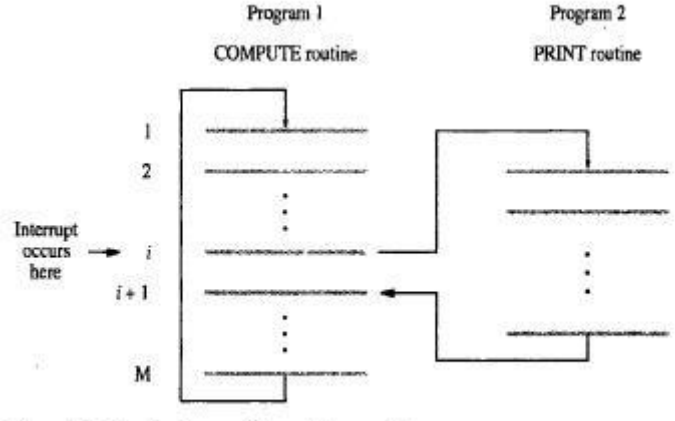


Figure 4.5 Transfer of control through the use of interrupts.

- The speed of operation of the processor and I/O devices differ greatly. Also, since I/O devices are manually operated in many cases (like pressing a key on keyboard), there may not be synchronization between the CPU operations and I/O operations with reference to CPU clock. To cater to the different needs of I/O operations, 3 mechanisms have been developed for interfacing I/O devices. 1) Program controlled I/O 2) Interrupt I/O 3) Direct memory access (DMA).
- Saving registers increases the delay between the time an interrupt request is received and the start of execution of the ISR. This delay is called interrupt latency.
- Since interrupts can arrive at any time, they may alter the sequence of events. Hence, facility must be provided to enable and disable interrupts as desired.
- Consider the case of a single interrupt request from one device. The device keeps the interrupt request signal activated until it is informed that the processor has accepted its request. This activated signal, if not deactivated may lead to successive interruptions, causing the system to enter into an infinite loop.

INTERRUPT HARDWARE

- An I/O device requests an interrupt by activating a bus-line called interrupt-request(IR).
- A single IR line can 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(i.e. if all switches are open), the voltage on the IR line will be equal to V_{dd}.
- When a device requests an interrupt by closing its switch, the voltage on the line drops to 0, causing the INTR received by the processor to goto 1.
- The value of INTR is the logical OR of the requests from individual devices

  $INTR=INTR_1+ INTR_{2+} . . . . . +INTR_n$
- A special gate known as open-collector or open-drain are used to drive the INTR line.
- 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.

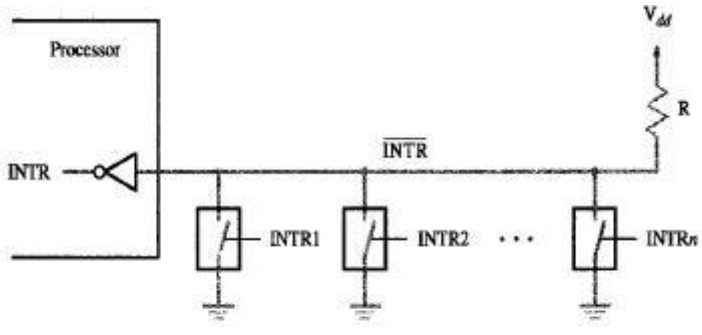


Figure 4.6 An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

ENABLING & DISABLING INTERRUPTS

- To prevent the system from entering into an infinite-loop because of interrupt, there are 3 possibilities:

  1) The first possibility is to have the processor-hardware ignore the interrupt-request line until the execution of the first instruction of the ISR has been completed.

  2) The second option is to have the processor automatically disable interrupts before starting the execution of the ISR.

  3) In the third option, the processor has a special interrupt-request line for which the interrupt-handling circuit responds only to the leading edge of the signal. Such a line is said to be edge-triggered.
- Sequence of events involved in handling an interrupt-request from a single device is as follows:

  1) The device raises an interrupt-request.

  2) The program currently being executed is interrupted.

  3) All interrupts are disabled(by changing the control bits in the PS).

  4) The device is informed that its request has been recognized, and

  in response, the device deactivates the interrupt-request signal.

  5) The action requested by the interrupt is performed by the ISR.

  6) Interrupts are enabled again and execution of the interrupted program is resumed.

HANDLING MULTIPLE DEVICES

Polling

- Information needed to determine whether a device is requesting an interrupt is available in its status-register.
- When a device raises an interrupt-request, it sets IRQ bit to 1 in its status-register (Figure 4.3).
- KIRQ and DIRQ are the interrupt-request bits for keyboard & display.
- Simplest way to identify interrupting device is to have ISR poll all I/O devices connected to bus.
- The first device encountered with its IRQ bit set is the device that should be serviced. After servicing this device, next requests may be serviced.
- Main advantage: Simple & easy to implement.
  Main disadvantage: More time spent polling IRQ bits of all devices (that may not be requesting any service).

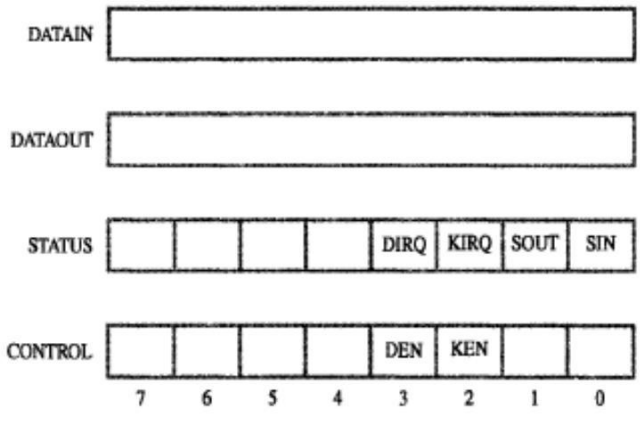


Figure 4.3 Registers in keyboard and display interfaces.

```
        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. (This enables processor to identify individual devices even if they share a single interrupt-request line).
- The code represents starting-address of ISR for that device.
- ISR for a given device must always start at same location.
- The address stored at the location pointed to by interrupting-device is called the interrupt-vector.
- Processor
  → loads interrupt-vector into PC &
  → executes appropriate ISR
- Interrupting-device must wait to put data on bus only when processor is ready to receive it.
- When processor is ready to receive interrupt-vector code, it activates INTA line.
- I/O device responds by sending its interrupt-vector code & turning off the INTR signal.

CONTROLLING DEVICE REQUESTS

• There are 2 independent mechanisms for controlling interrupt requests.

• At device-end, an interrupt-enable bit in a control register determines whether device is allowed to generate an interrupt request.

• At processor-end, either an interrupt-enable bit in the PS register or a priority structure determines whether a given interrupt-request will be accepted.

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 of the INTR lines is assigned a different priority-level (Figure 4.7).
- Priority-level of processor is the priority of program that is currently being executed.
- During execution of an ISR, interrupt-requests will be accepted from some devices but not from others depending upon device's priority.
- Processor accepts interrupts only from devices that have priority higher than its own.
- At the time of execution of an ISR for some device is started, priority of processor is raised to that of the device
- Processor's priority is encoded in a few bits of processor-status (PS) word. This can be changed by program instructions that write into PS. These are called *privileged instructions.*
- Privileged-instructions can be executed only while processor is running in supervisor-mode.
- Processor is in supervisor-mode only when executing operating-system routines. (An attempt to execute a privileged-instruction while in the user-mode leads to a special type of interrupt called a *privileged exception*).

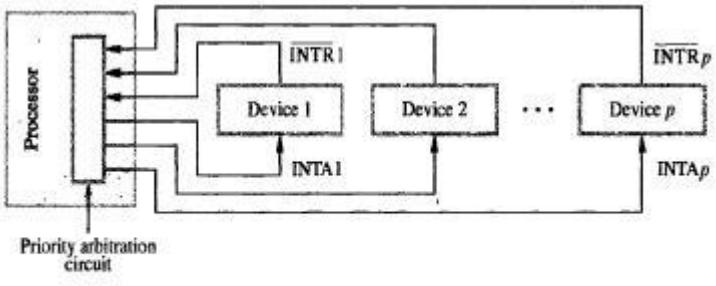
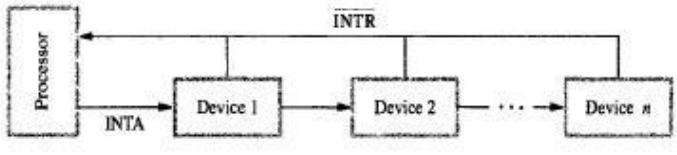


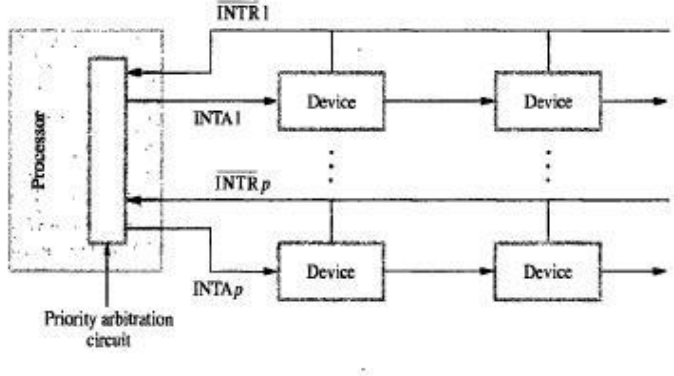
Figure 4.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

SIMULTANEOUS REQUESTS

- INTR line is common to all devices (Figure 4.8).
- INTA line is connected in a daisy-chain fashion such that INTA signal propagates serially through devices.
- When several devices raise an interrupt-request and 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, it blocks INTA signal and proceeds to put its identifying code on data lines.
- Device that is electrically closest to processor has highest priority.
- Main advantage: This allows the processor to accept interrupt-requests from some devices but not from others depending upon their priorities.



(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.
  I/O interrupts are one example of an exception.

Recovery from Errors

• Computers use a variety of techniques to ensure that all hardware-components are operating properly. For e.g. many computers include an error-checking code in main-memory which allows detection of errors in stored-data.
• If an error occurs, control-hardware detects it & informs processor by raising an interrupt.
• When exception processing is initiated (as a result of errors), processor
  → suspends program being executed &
  → starts an ESR(Exception Service Routine). This routine takes appropriate action to recover from the error to inform user about it.

Debugging

• Debugger
  → helps programmer find errors in a program and
  → uses exceptions to provide 2 important facilities: 1) Trace & 2) Breakpoints
• 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 (AX, BX), memory-locations and so on.
• On return from debugging-program,
  next instruction in program being debugged is executed,
    then debugging-program is activated again.
• Breakpoints provide a similar facility except that 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.

Privilege Exception

• To protect OS of computer from being corrupted by user-programs, certain instructions can be executed only while processor is in supervisor-mode. These are called *privileged instructions.*
• For e.g. when the processor is running in user-mode, it will not execute an instruction that changes priority-level of processor.
• An attempt to execute such an 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 between an external device & main memory without continuous involvement by processor is called as *DMA*.

• DMA transfers are performed by a control-circuit that is part of I/O device interface. This circuit is called as a *DMA controller* (Figure 4.19).

• DMA controller performs the functions that would normally be carried out by processor

• In controller, 3 registers are accessed by processor to initiate transfer operations (Figure 4.18):

1) Two registers are used for storing starting-address & word-count

2) Third register contains status- & control-flags

• The R/W bit determines direction of transfer.

When R/W=1, controller performs a read operation(i.e. it transfers data from memory to I/O),

Otherwise it performs a write operation (i.e. it transfers data from I/O device to memory).

• When Done=1, controller

→ completes transferring a block of data &

→ is ready to receive another command.

• When IE=1, controller raises an interrupt after it has completed transferring a block of data (IE=Interrupt Enable).

• Finally, when 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:

2) In one method, processor originates most memory-access cycles. DMA controller is said to "steal" memory cycles from processor. Hence, this technique is usually called *cycle stealing.*

3) In second method, 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).

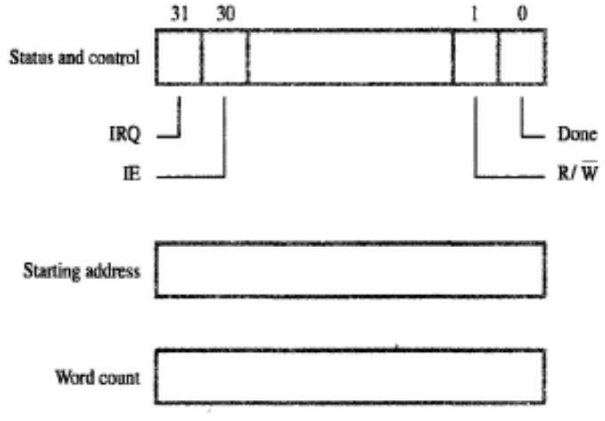


Figure 4.18 Registers in a DMA interface.

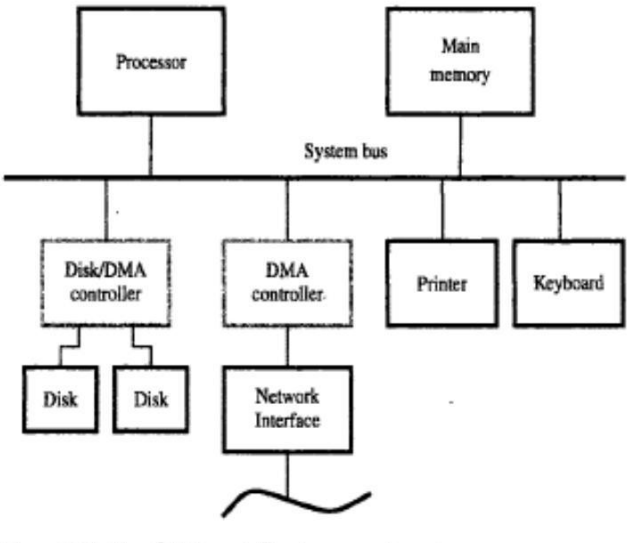


Figure 4.19 Use of DMA controllers in a computer system.

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 and bus-mastership is transferred to it.
• There are 2 approaches to bus arbitration:
1) In centralized arbitration, a single bus-arbiter performs the required arbitration.
2) In distributed arbitration, all device participate in selection of next bus-master.

CENTRALIZED ARBITRATION

• A single bus-arbiter performs the required arbitration (Figure: 4.20 & 4.21).
• Normally, processor is the bus. master unless it grants bus mastership to one of the DMA controllers.
• A DMA controller indicates that it needs to become busmaster by activating Bus-Request line(BR).
• The signal on the BR line is the logical OR of bus-requests from all devices connected to it.
• When BR is activated, processor activates Bus-Grant signal(BG1) indicating to DMA controllers that they may use bus when it becomes free. (This signal is connected to all DMA controllers using a daisy-chain arrangement).
• If DMA controller-1 is requesting the bus, it blocks propagation of grant-signal to other devices.
Otherwise, it passes the grant downstream by asserting BG2.
• Current bus-master indicates to all devices that it is using bus by activating Bus-Busy line (BBSY).
• Arbiter circuit ensures that only one request is granted at any given time according to a predefined priority scheme

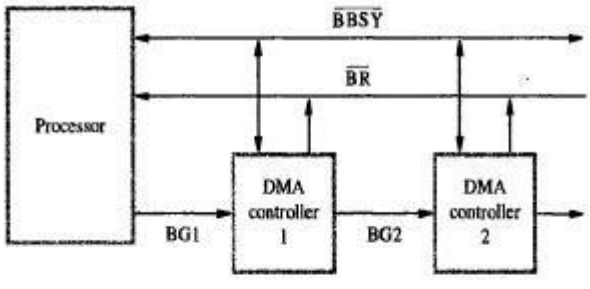


Figure 4.20 A simple arrangement for bus arbitration using a daisy chain.

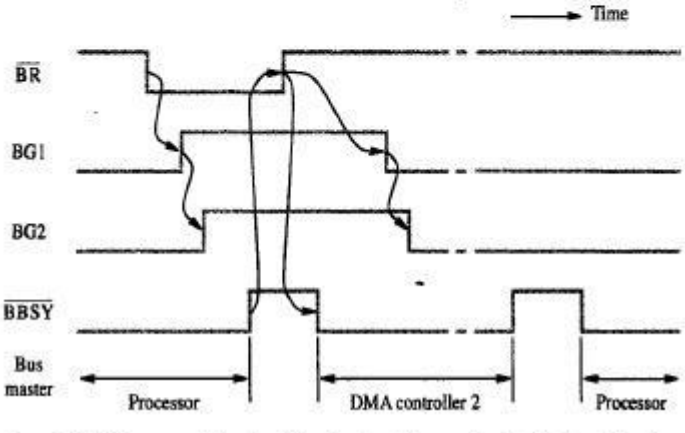


Figure 4.21 Sequence of signals during transfer of bus mastership for the devices in Figure 4.20.

A conflict may arise if both the processor and a DMA controller try to use the bus at the same time to access the main memory. To resolve these conflicts, a special circuit called the bus arbiter is provided to coordinate the activities of all devices requesting memory transfers

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 $\overline{ARB\ 0}$ through $\overline{ARB\ 3}$.
- A winner is selected as a result of interaction among signals transmitted over these lines by all contenders.
- Net outcome is that the code on 4 lines represents request that has the highest ID number.
- Main advantage: This approach offers higher reliability since operation of bus is not dependent on any single device.

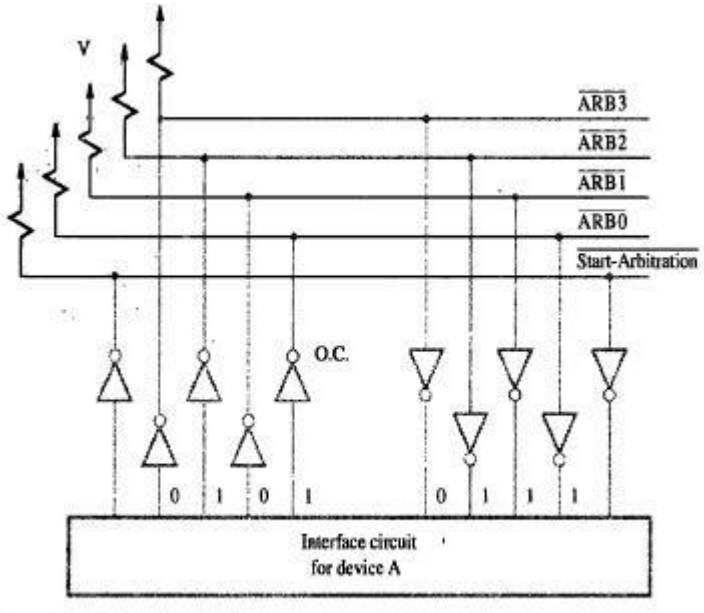


Figure 4.22 A distributed arbitration scheme.

BUSES

- 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 behaviour 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 and 3) Control lines.
- Control-signals specify whether a read or a write operation is to be performed.
- R/W line specifies
  - → read operation when R/W=1
  - → write operation when R/W=0
- In data-transfer operation, one device plays the role of a bus-master which initiates data transfers by issuing Read or Write commands on bus (Hence it may be called an initiator).
- Device addressed by master is referred to as a slave (or target).
- Timing of data transfers over a bus is classified into 2 types:
  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.
- Each of these intervals constitutes a bus-cycle during which 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 4.23).
- Information travels over bus at a speed determined by its physical & electrical characteristics.
- Clock pulse width (t_1-t_0) must be longer than the maximum propagation-delay between 2 devices connected to bus.
- 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 any storage device (such as a register built with flip-flops), data must be available at input of that device for a period greater than setup-time of device.

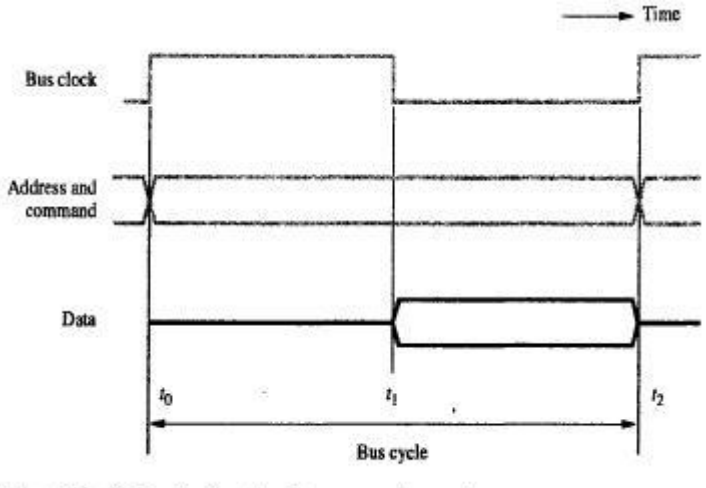


Figure 4.23 Timing of an input transfer on a synchronous bus.

ASYNCHRONOUS BUS

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

The read operation proceeds as follows:

- At t0, master places address- & command-information on bus. All devices on bus begin to decode this information.
- At t1, master sets MR-signal to 1 to inform all devices that the address- & command-information is ready.
- At t2, selected slave performs required input-operation & sets SR signal to 1 (Figure 4.26).
- At t3, SR signal arrives at master indicating that the input-data are available on bus skew.
- At t4, master removes address- & command-information from bus.
- At t5, 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

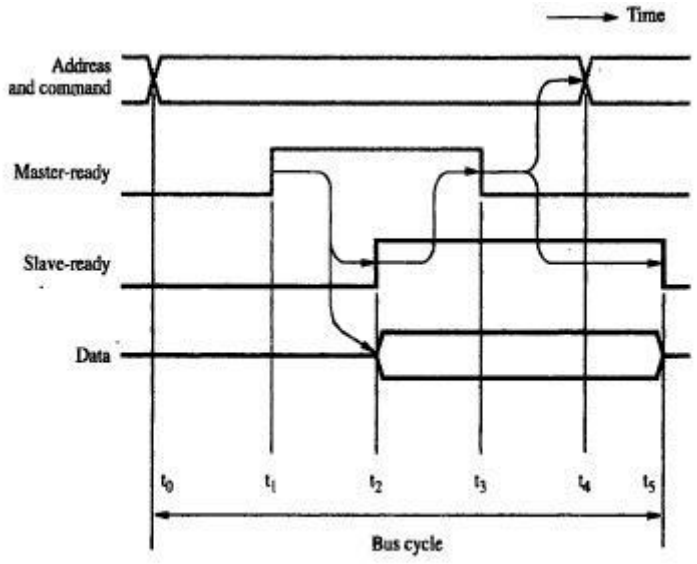
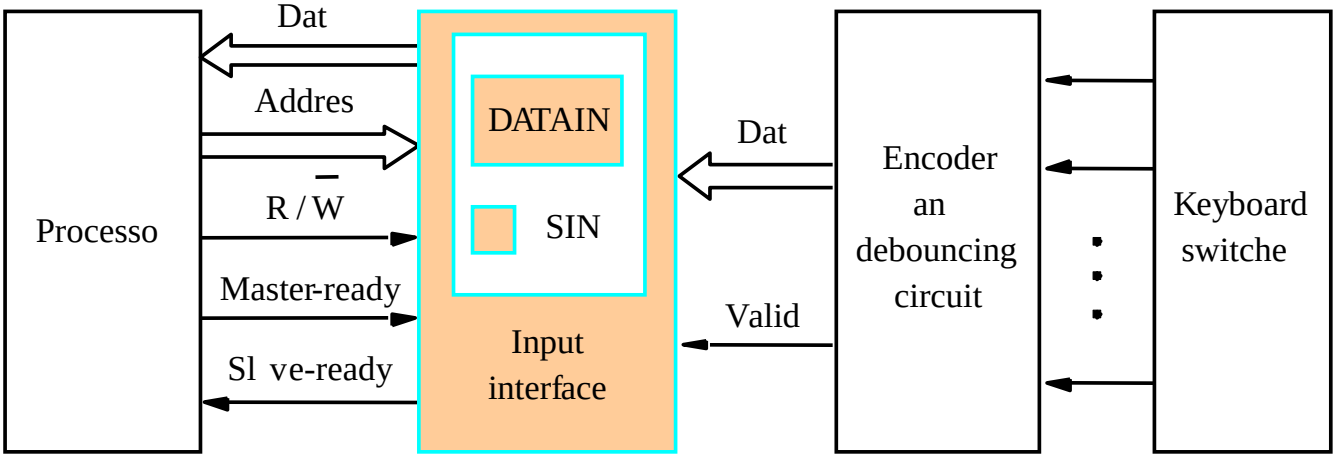


Figure 4.26 Handshake control of data transfer during an input operation.

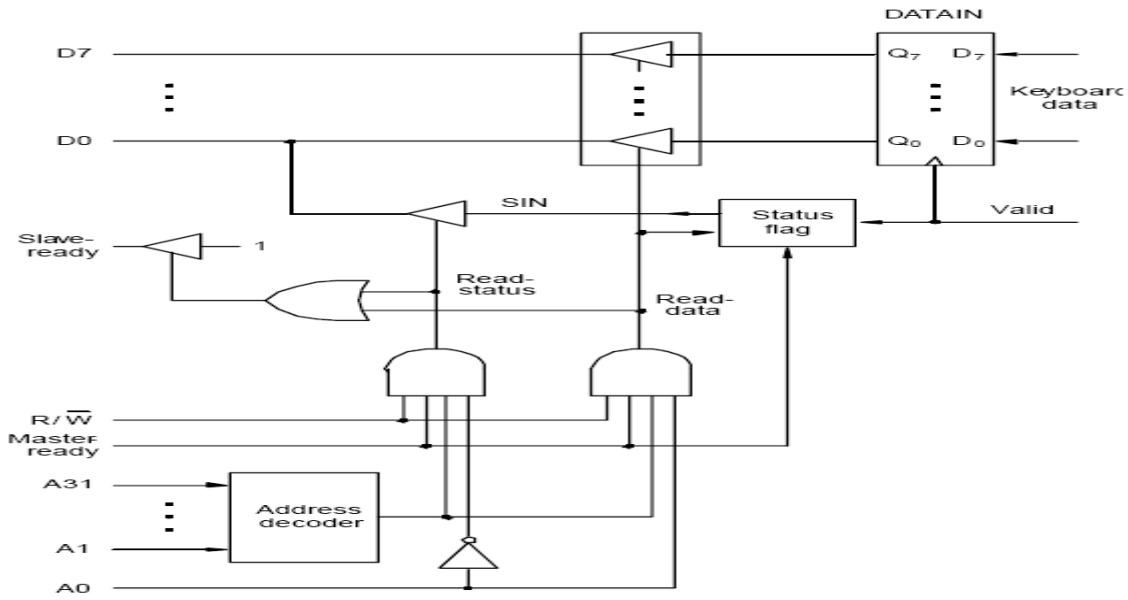
INTERFACE CIRCUITS

- I/O interface consists of the circuitry required to connect an I/O device to a computer bus.
- Side of the interface which connects to the computer has bus signals for:
    - Address,
    - Data
    - Control
- Side of the interface which connects to the I/O device has:
    - Datapath and associated controls to transfer data between the interface and the I/O device.
    - This side is called as a "port".
- Ports can be classified into two:
    - Parallel port,
    - Serial port.
- Parallel port transfers data in the form of a number of bits, normally 8 or 16 to or from the device.
- Serial port transfers and receives data one bit at a time.
- Processor communicates with the bus in the same way, whether it is a parallel port or a serial port.
    - Conversion from the parallel to serial and vice versa takes place inside the interface circuit.



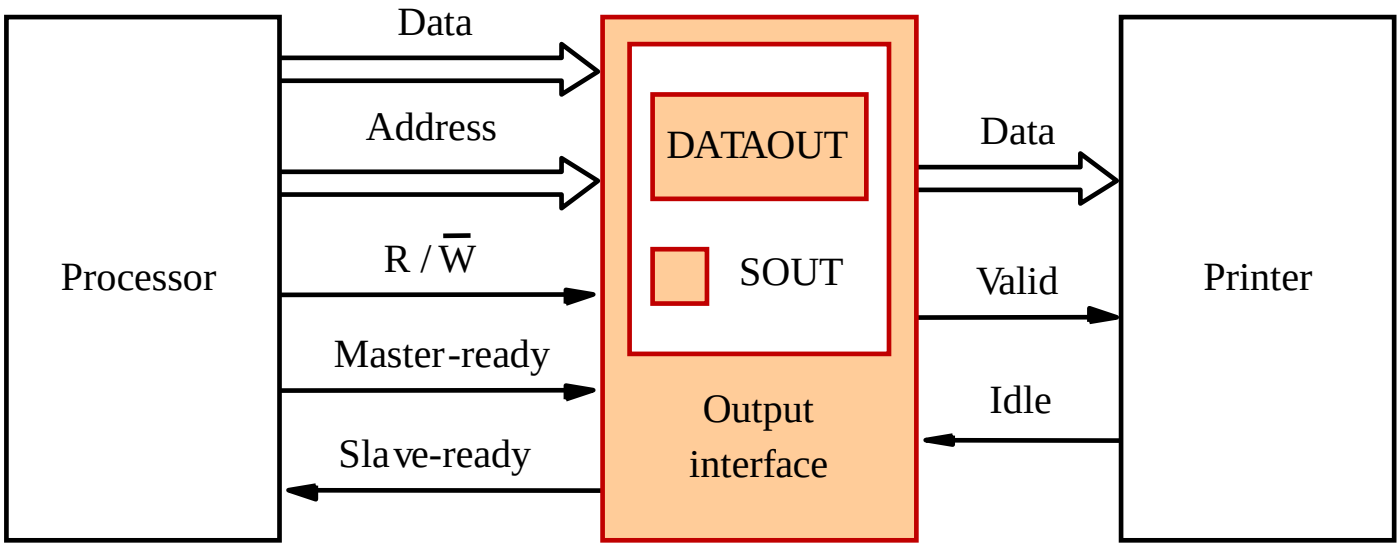
- Keyboard is connected to a processor using a parallel port.
- Processor is 32-bits and uses memory-mapped I/O and the asynchronous bus protocol.
- On the processor side of the interface we have:
  - Data lines.
  - Address lines
  - Control or R/W line.
  - Master-ready signal and
  - Slave-ready signal.
- On the keyboard side of the interface:
  - Encoder circuit which generates a code for the key pressed.
  - Debouncing circuit which eliminates the effect of a key bounce (a single key stroke may appear as multiple events to a processor).
  - Data lines 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.

INPUT INTERFACE CIRCUIT



- Output lines of DATAIN are connected to the data lines of the bus by means of 3 state drivers
- Drivers are turned on when the processor issues a read signal and the address selects this register.
- *SIN signal is generated using a status flag circuit.*
- *It is connected to line D_0 of the processor bus using a three-state driver.*
- *Address decoder selects the input interface based on bits A_1 through A_{31}.*
- *Bit A0 determines whether the status or data register is to be read, when Master-ready is*
- *active.*
- *In response, the processor activates the Slave-ready signal, when either the Read-status or Read-data is equal to 1, which depends on line A_0.*

PRINTER TO PROCESSOR CONNECTION



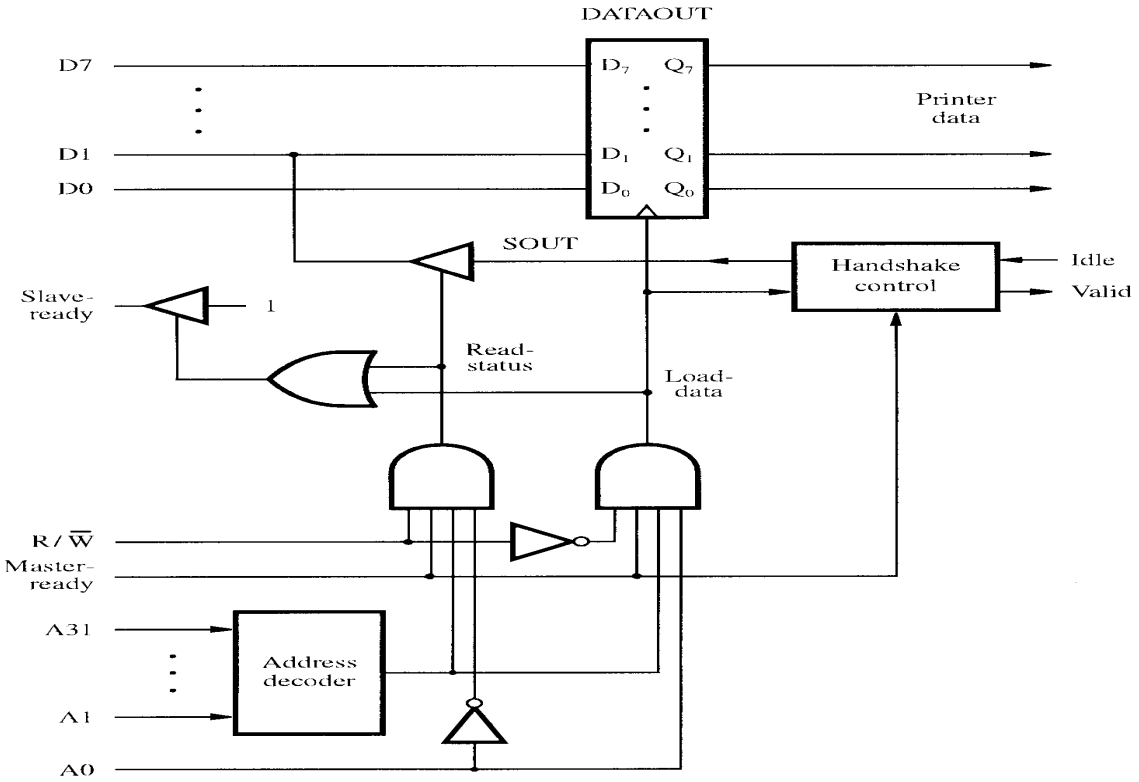
- *Printer is connected to a processor using a parallel port.*
- *Processor is 32 bits, uses memory-mapped I/O and asynchronous bus protocol.*
- *On the processor side:*

- Data lines.
- Address lines
- Control or R/W line.
- Master-ready signal and
- Slave-ready signal.

- *On the printer side:*

- Idle signal line which the printer asserts when it is ready to accept a character. This causes the SOUT flag to be set to 1.
- Processor places a new character into a DATAOUT register.
- Valid signal, asserted by the interface circuit when it places a new character on the data lines.

OUTPUT INTERFACE CIRCUIT



- Data lines of the processor bus are connected to the DATAOUT register of the interface.
- The status flag SOUT is connected to the data line D1 using a three-state driver.
- The three-state driver is turned on, when the control Read-status line is 1.
- Address decoder selects the output interface using address lines A1 through A31.
- Address line A0 determines whether the data is to be loaded into the DATAOUT register or status flag is to be read.
- If the Load-data line is 1, then the Valid line is set to 1.
- If the Idle line is 1, then the status flag SOUT is set to 1.

COMBINED I/O INTERFACE CIRCUITS

- Address bits A2 through A31, that is 30 bits are used to select the overall interface.
- Address bits A1 through A0, that is, 2 bits select one of the three registers, namely, DATAIN, DATAOUT, and the status register.
- Status register contains the flags SIN and SOUT in bits 0 and 1.
- Data lines PA0 through PA7 connect the input device to the DATAIN register.
- DATAOUT register connects the data lines on the processor bus to lines PB0 through PB7 which connect to the output device.
- Separate input and output data lines for connection to an I/O device. Refer fig no. 4.33

SERIAL PORT

- Serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time.
- Serial port communicates in a bit-serial fashion on the device side and bit parallel fashion on the bus side.
    - Transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability.
- Input shift register accepts input one bit at a time from the I/O device. Refer fig no.4.37
- Once all the 8 bits are received, the contents of the input shift register are loaded in parallel into DATAIN register.
- Output data in the DATAOUT register are loaded into the output shift register.

- Bits are shifted out of the output shift register and sent out to the I/O device one bit at a time.
- As soon as data from the input shift registers are loaded into DATAIN, it can start accepting another 8 bits of data.
- Input shift register and DATAIN registers are both used at input so that the input shift register can start receiving another set of 8 bits from the input device after loading the contents to DATAIN, before the processor reads the contents of DATAIN. This is called as double-buffering.
- Serial interfaces require fewer wires, and hence serial transmission is convenient for connecting devices that are physically distant from the computer.
- Speed of transmission of the data over a serial interface is known as the "bit rate".
    - Bit rate depends on the nature of the devices connected.
- In order to accommodate devices with a range of speeds, a serial interface must be able to use a range of clock speeds.
- Several standard serial interfaces have been developed:
    - Universal Asynchronous Receiver Transmitter (UART) for low-speed serial devices.
    - RS-232-C for connection to communication links.

STANDARD I/O INTERFACES

- I/O device is connected to a computer using an interface circuit.
- Do we have to design a different interface for every combination of an I/O device and a computer?
- A practical approach is to develop standard interfaces and protocols.
- A personal computer has:
    - A motherboard which houses the processor chip, main memory and some I/O interfaces.
    - A few connectors into which additional interfaces can be plugged.
- Processor bus is defined by the signals on the processor chip.
    - Devices which require high-speed connection to the processor are connected directly to this bus.
- Because of electrical reasons only a few devices can be connected directly to the processor bus.
- Motherboard usually provides another bus that can support more devices.
    - Processor bus and the other bus (called as expansion bus) are interconnected by a circuit called "bridge".
    - Devices connected to the expansion bus experience a small delay in data transfers.
- Design of a processor bus is closely tied to the architecture of the processor.
    - No uniform standard can be defined.
- Expansion bus however can have uniform standard defined.
- A number of standards have been developed for the expansion bus.
    - Some have evolved by default.
    - For example, IBM's Industry Standard Architecture.
- Three widely used bus standards:
    - PCI (Peripheral Component Interconnect)
    - SCSI (Small Computer System Interface)
    - USB (Universal Serial Bus)

Refer fig no. 4.38

PCI BUS

- *Peripheral Component Interconnect*
- Introduced in 1992
- Low-cost bus
- Processor independent
- Plug-and-play capability
- In today's computers, most memory transfers involve a burst of data rather than just one word. The PCI is designed primarily to support this mode of operation.
- The bus supports three independent address spaces: memory, I/O, and configuration.
- we assumed that the master maintains the address information on the bus until data transfer is completed. But, the address is needed only long enough for the slave to be selected. Thus, the address is needed on the bus for one clock cycle only, freeing the address lines to be used for sending data in subsequent clock cycles. The result is a significant cost reduction.

- A master is called an initiator in PCI terminology. The addressed device that responds to read and write commands is called a target.

Refer table 4.3 and 4.40 from text

Device configuration

- When an I/O device is connected to a computer, several actions are needed to configure both the device and the software that communicates with it.
- PCI incorporates in each I/O device interface a small configuration ROM memory that stores information about that device.
- The configuration ROMs of all devices are accessible in the configuration address space. The PCI initialization software reads these ROMs and determines whether the device is a printer, a keyboard, an Ethernet interface, or a disk controller. It can further learn bout various device options and characteristics.
- Devices are assigned addresses during the initialization process.
- This means that during the bus configuration operation, devices cannot be accessed based on their address, as they have not yet been assigned one.
- Hence, the configuration address space uses a different mechanism. Each device has an input signal called Initialization Device Select, IDSEL#
- Electrical characteristics:
    - PCI bus has been defined for operation with either a 5 or 3.3 V power supply

SCSI BUS

- The acronym SCSI stands for Small Computer System Interface.
- It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131 .
- In the original specifications of the standard, devices such as disks are connected to a computer via a 50-wire cable, which can be up to 25 meters in length and can transfer data at rates up to 5 megabytes/s.
- The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years.
- SCSI-2 and SCSI-3 have been defined, and each has several options.
- Because of various options SCSI connector may have 50, 68 or 80 pins.
- Devices connected to the SCSI bus are not part of the address space of the processor
- The SCSI bus is connected to the processor bus through a SCSI controller. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa.
- A packet may contain a block of data, commands from the processor to the device, or status information about the device.
- A controller connected to a SCSI bus is one of two types – an initiator or a target.
- An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. The disk controller operates as a target. It carries out the commands it receives from the initiator.
- The initiator establishes a logical connection with the intended target.
- Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data.
- While a particular connection is suspended, other device can use the bus to transfer information.
- This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.
- Data transfers on the SCSI bus are always controlled by the target controller.
- To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it.
- Then the controller starts a data transfer operation to receive a command from the initiator.
- Assume that processor needs to read block of data from a disk drive and that data are stored in disk sectors that are not contiguous.
- The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:
    - The SCSI controller, acting as an initiator, contends for control of the bus.
    - When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
    - The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.

- The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
- 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. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator, the target requests control of the bus. After it wins arbitration, it reselects the initiator controller, thus restoring the suspended connection.
- The target transfers the contents of the data buffer to the initiator and then suspends the connection again
- The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator as before. At the end of this transfers, the logical connection between the two controllers is terminated.
- As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
- The SCSI controller sends as interrupt to the processor to inform it that the requested operation has been completed

MAIN PHASES INVOLVED

- Arbitration
  - A controller requests the bus by asserting BSY and by asserting it's associated data line
  - When BSY becomes active, all controllers that are requesting bus examine data lines
- Selection
  - Controller that won arbitration selects target by asserting SEL and data line of target. After that initiator releases BSY line.
  - Target responds by asserting BSY line
  - Target controller will have control on the bus from then
- Information Transfer
  - Handshaking signals are used between initiator and target
  - At the end target releases BSY line
- Reselection

USB

- Universal Serial Bus (USB) is an industry standard developed through a collaborative effort of several computer and communication companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, and Philips.
- Speed
  - Low-speed(1.5 Mb/s)
  - Full-speed(12 Mb/s)
  - High-speed(480 Mb/s)
- Port Limitation
- Device Characteristics
- Plug-and-play

USB TREE STRUCTURE

Refer fig 4.44

- 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.
- Each node of the tree has a device called a hub, which 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, Internet connection, speaker, or digital TV)
- In normal operation, 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. However, a message from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, the USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.
- When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual

devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree.

- Each device on the USB, whether it is a hub or an I/O device, 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 to which this device is connected is capable of detecting that the device 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 a 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, where 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 the USB can be divided into two broad categories: control and data.
    - Control packets perform such tasks 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 the type of that packet.
- They are transmitted twice. The first time they are sent with their true values, and 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.

ELECTRICAL CHARACTERISTICS

- The cables used for USB connections consist of four wires.
- Two are used to carry power, +5V and Ground.
    - Thus, a hub or an I/O device may be powered directly from the bus, or it may have its own external power connection.
- The other two wires are used to carry data.
- Different signaling schemes are used for different speeds of transmission.
    - At low speed, 1s and 0s are transmitted by sending a high voltage state (5V) on one or the other o the two signal wires. For high-speed links, differential transmission is used.

MODULE – 3

THE MEMORY SYSTEM

5.1 BASIC CONCEPTS:

The maximum size of the Main Memory (MM) that can be used in any computer is determined by its addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing upto 2^{16} =64K memory locations. If a machine generates 32-bit addresses, it can access upto 2^{32} = 4G memory locations. This number represents the size of address space of the computer.

If the smallest addressable unit of information is a memory word, the machine is called word-addressable. If individual memory bytes are assigned distinct addresses, the computer is called byte-addressable. Most of the commercial machines are byte-addressable. For example in a byte-addressable 32-bit computer, each memory word contains 4 bytes. A possible word-address assignment would be:

Word Address	**Byte Address**			
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
.		…..		
.		…..		
.		…..		

With the above structure a READ or WRITE may involve an entire memory word or it may involve only a byte. In the case of byte read, other bytes can also be read but ignored by the CPU. However, during a write cycle, the control circuitry of the MM must ensure that only the specified byte is altered. In this case, the higher-order 30 bits can specify the word and the lower-order 2 bits can specify the byte within the word.

CPU-Main Memory Connection – A block schematic: -

Data transfer between CPU and MM takes place through the use of two CPU registers, usually called MAR (Memory Address Register) and MDR (Memory Data Register). If MAR is K bits long and MDR is 'n' bits long, then the MM unit may contain upto 2^k addressable locations and each location will be 'n' bits wide, while the word length is equal to 'n' bits. During a "memory cycle", n bits of data may be transferred between the MM and CPU. This transfer takes place over the processor bus, which has k address lines (address bus), n data lines (data bus) and control lines like Read, Write, Memory Function completed (MFC), Bytes specifiers etc (control bus). For a read operation, the CPU loads the address into MAR, set R/$\overline{W}$ to 1 and sets other control signals if required. The data from the MM is loaded into MDR and MFC is set to 1. For a write operation, MAR, MDR are suitably loaded by the CPU,R/$\overline{W}$ is set to 0 and other control signals are set suitably. The MM control circuitry loads the data into appropriate locations and sets MFC to 1. This organization is shown in the following block schematic.

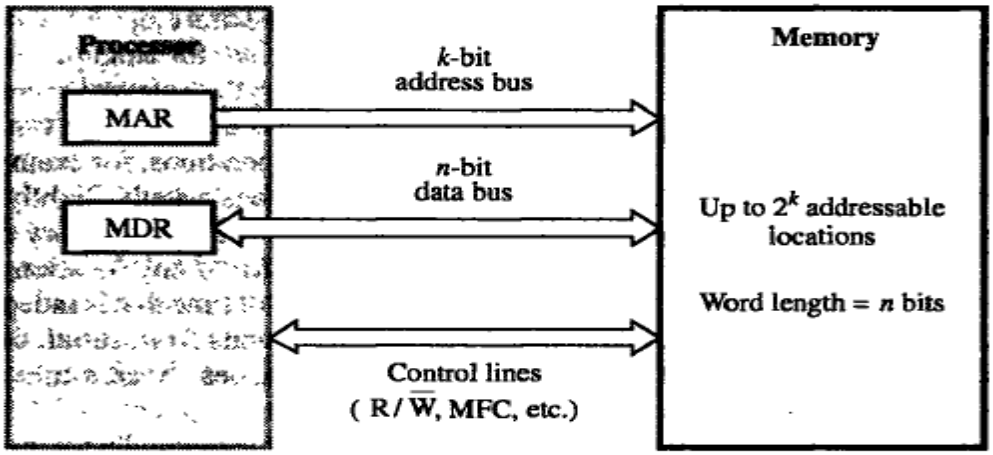


Figure 5.1 Connection of the memory to the processor.

Some Basic Concepts

Memory Access Times: -

It is a useful measure of the speed of the memory unit. It is the time that elapses between the initiation of an operation and the completion of that operation (for example, the time between READ and MFC).

Memory Cycle Time :-

It is an important measure of the memory system. It is the minimum time delay required between the initiations of two successive memory operations (for example, the time between two successive READ operations). The cycle time is usually slightly longer than the access time.

RAM: A memory unit is called a Random Access Memory if any location can be accessed for a READ or WRITE operation in some fixed amount of time that is independent of the location's address. Main memory units are of this type. This distinguishes them from serial or partly serial access storage devices such as magnetic tapes and disks which are used as the secondary storage device.

Cache Memory:-

The CPU of a computer can usually process instructions and data faster than they can be fetched from compatibly priced main memory unit. Thus the memory cycle time becomes the bottleneck in the system. One way to reduce the memory access time is to use cache memory. This is a small and fast memory that is inserted between the larger, slower main memory and the CPU. This holds the currently active segments of a program and its data. Because of the locality of address references, the CPU can, most of the time, find the relevant information in the cache memory itself (cache hit) and infrequently needs access to the main memory (cache miss) with suitable size of the cache memory, cache hit rates of over 90% are possible leading to a cost-effective increase in the performance of the system.

Memory Interleaving: -

This technique divides the memory system into a number of memory modules and arranges addressing so that successive words in the address space are placed in different modules. When requests for memory access involve consecutive addresses, the access will be to different modules. Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.

Virtual Memory: -

In a virtual memory System, the address generated by the CPU is referred to as a virtual or logical address. The corresponding physical address can be different and the required mapping is implemented by a special memory control unit, often called the memory

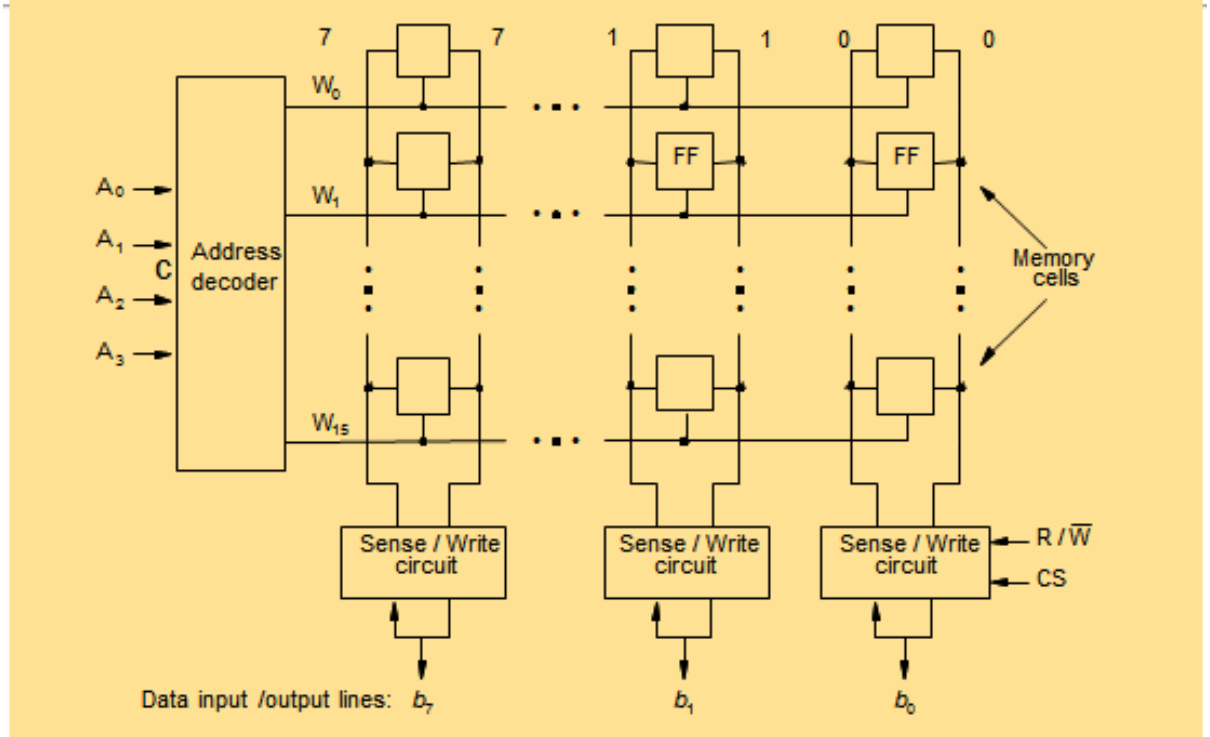
management unit. The mapping function itself may be changed during program execution according to system requirements.

Because of the distinction made between the logical (virtual) address space and the physical address space; while the former can be as large as the addressing capability of the CPU, the actual physical memory can be much smaller. Only the active portion of the virtual address space is mapped onto the physical memory and the rest of the virtual address space is mapped onto the bulk storage device used. If the addressed information is in the Main Memory (MM), it is accessed and execution proceeds. Otherwise, an exception is generated, in response to which the memory management unit transfers a contigious block of words containing the desired word from the bulk storage unit to the MM, displacing some block that is currently inactive. If the memory is managed in such a way that, such transfers are required relatively infrequency (ie the CPU will generally find the required information in the MM), the virtual memory system can provide a reasonably good performance and succeed in creating an illusion of a large memory with a small, in expensive MM.

5.2 SEMICONDUCTOR RAM MEMORIES

5.2.1 Internal Organization of Memory Chips

Memory cells are usually organized in the form of an array, in which each cell is capable of storing on bit of information. Each row of cells constitutes a memory word, and all cells of a row are connected to a common line referred to as the word line, which is driven by the address decoder on the chip. The cells in each column are connected to a Sense/Write circuit by two bit lines. The Sense/Write circuits are connected to the data I/O lines of the chip. During the read operation, these circuits' sense, or read, the information stored in the cells selected by a word line and transmit this information to the output data lines. During the write operation, the Sense/Write circuits receive the input information and store in the cells of the selected word.



The above figure is an example of a very small memory chip consisting of 16 words of 8 bits each. This is referred to as a 16×8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that can be connected to the data bus of a computer. Two control lines, R/W (Read/ Write) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

The memory circuit given above stores 128 and requires 14 external connections for address, data and control lines. Of course, it also needs two lines for power supply and ground connections. Consider now a slightly larger memory circuit, one that has a 1k (1024) memory cells. For a 1k×1 memory organization, the representation is given next. The required 10-bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. However, according to the column address, only one of these cells is connected to the external data line by the output multiplexer and input de-multiplexer.

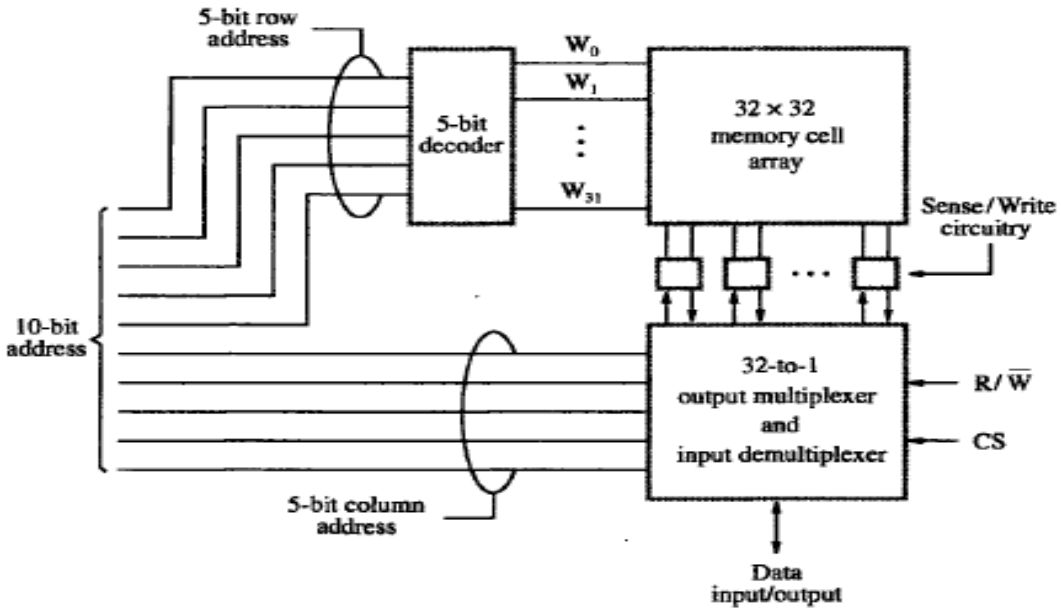
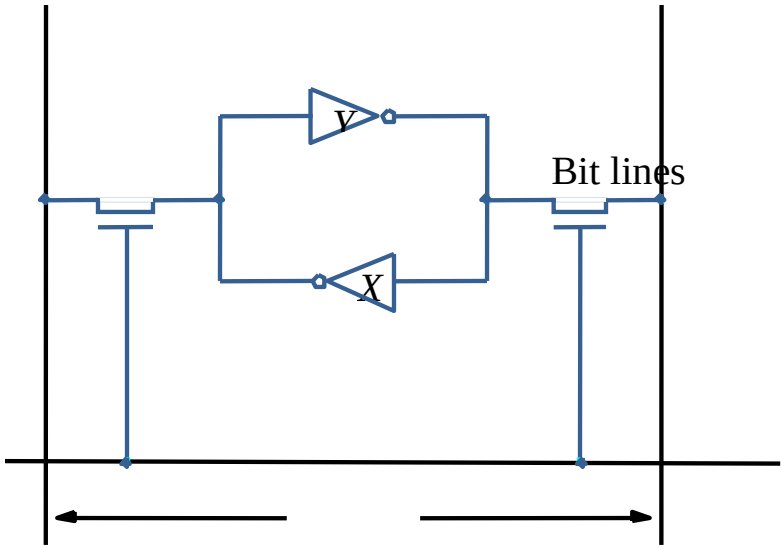


Figure 5.3 Organization of a 1K × 1 memory chip.

5.2.2 Static Memories

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



The above figure illustrates how a static RAM (SRAM) cell may be implemented. Two inverters are cross- connected to form a latch. The latch is connected to two bit lines by transistors T_1 and T_2. These transistors act as switches that can be opened or closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state. For example, let us assume that the cell is in state 1 if the

logic value at point X is 1 and at point Y is 0. This state is maintained as long as the signal on the word line is at ground level.

Read Operation

In order to read the state of the SRAM cell, the word line is activated to close switches T_1 and T_2. If the cell is in state 1, the signal on the bit line b is high and the signal on the bit line b' is low. The opposite is true if the cell is in state 0. Thus b and b' are compliments of each other. Sense/Write circuits at the end of the bit lines monitor the state of b and b' and set the output accordingly.

Write Operation

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

CMOS Cell

A CMOS realization of the static RAM cell is given below:

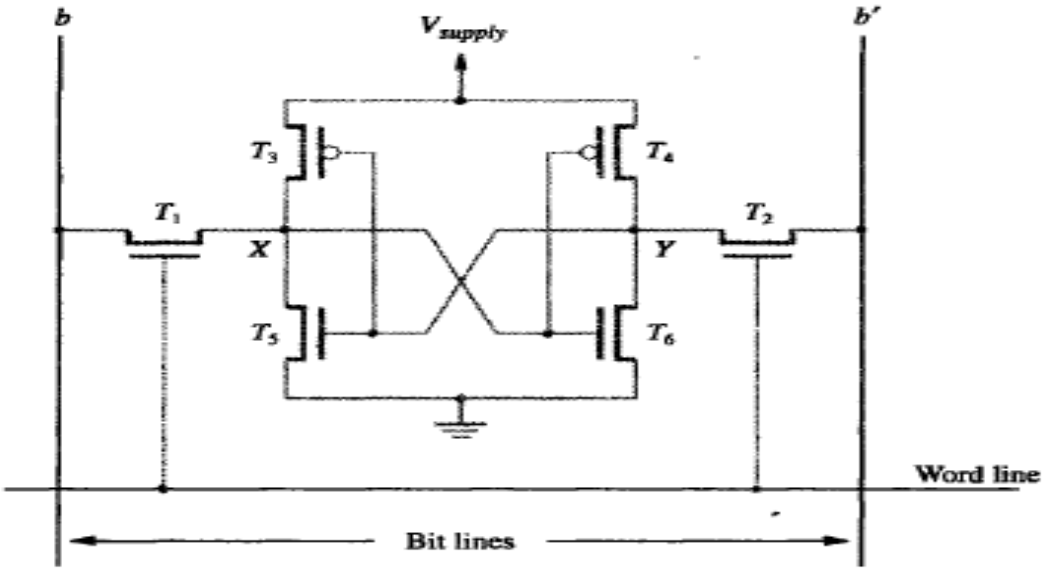


Figure 5.5 An example of a CMOS memory cell.

Transistor pairs (T_3, T_5) and (T_4, T_6) form the inverters in the latch (see Appendix A). The state of the cell is read or written as just explained. For example, in state 1, the voltage at point X is maintained high by having transistors T_3 and T_6 on, while T_4 and T_5 are off. Thus, if T_1 and T_2 are turned on (closed), bit lines b and b' will have high and low signals, respectively.

5.2.3 Asynchronous DRAMS

Information is stored in a dynamic memory cell in the form of a charge on a capacitor, and this charge can be maintained for only tens of milliseconds. Since the cell is required to store information for a much longer time, its contents must be periodically refreshed by restoring the capacitor charge to its full value. An example of a dynamic memory cell that consists of a capacitor, C, and a transistor, T, is shown below:

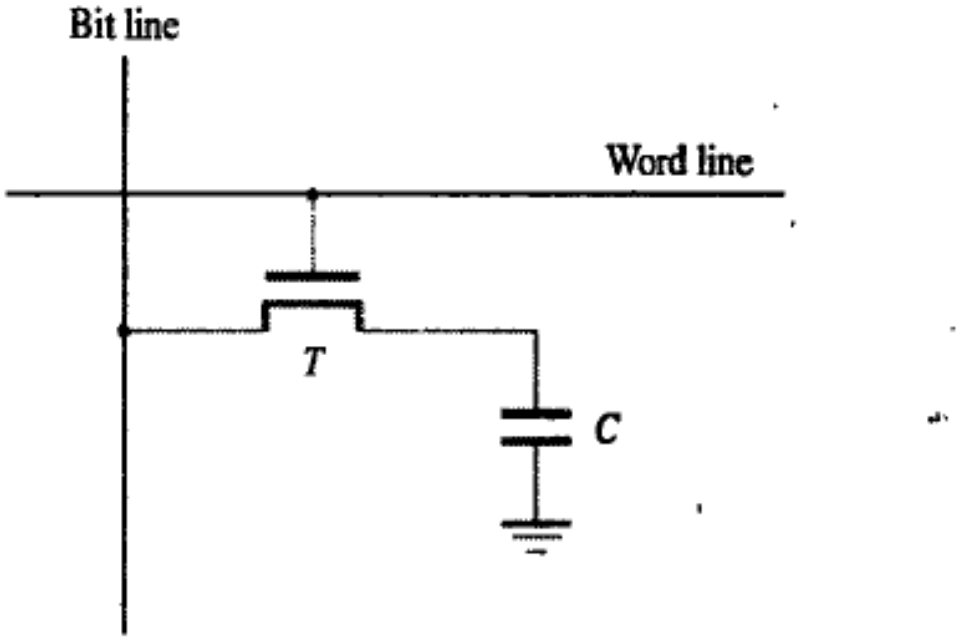
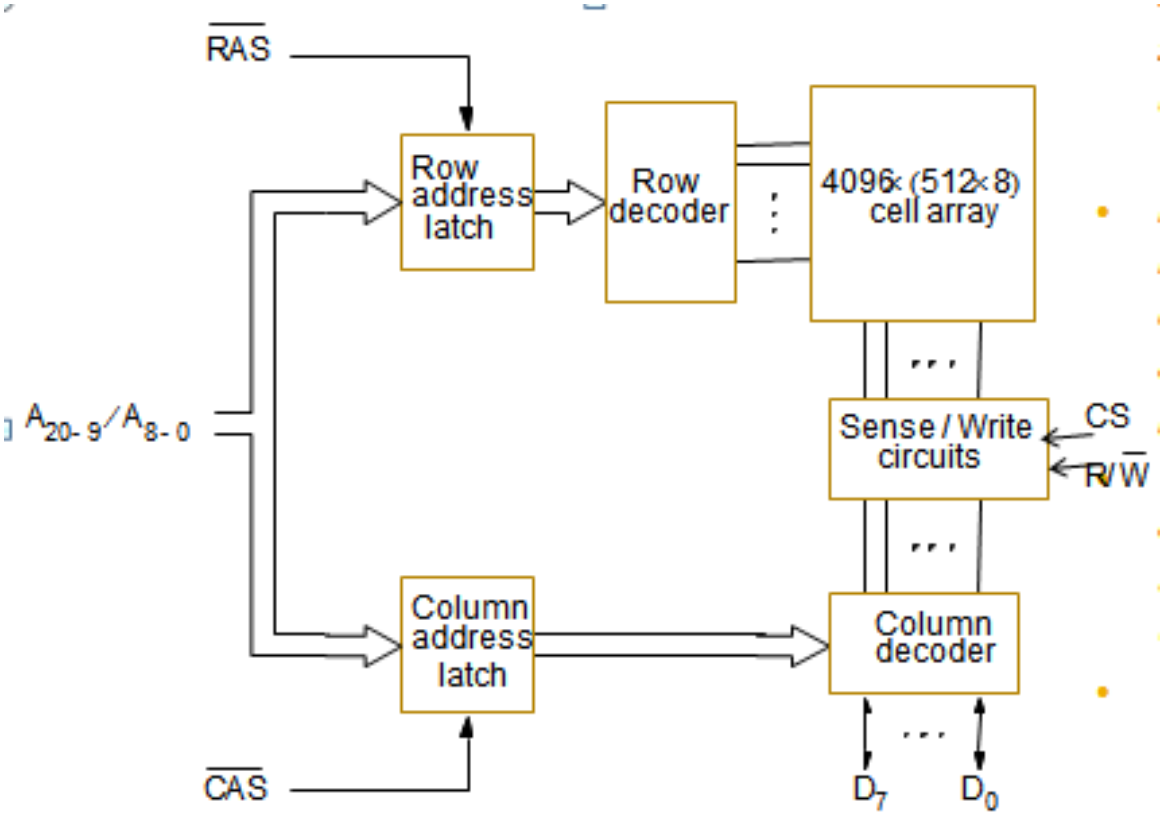


Figure 5.6 A single-transistor dynamic memory cell.

A sense amplifier connected to the bit line detects whether the charge stored on the capacitor is above the threshold. If so, it drives the bit line to a full voltage that represents logic value 1. This voltage recharges the capacitor to full charge that corresponds to logic value 1. If the sense amplifier detects that the charge on the capacitor will have no charge, representing logic value 0.

A 16-megabit DRAM chip, configured as 2M×8, is shown below.

- ***Each row can store 512 bytes. 12 bits to select a row, and 9 bits to select a group in a row. Total of 21 bits.***
- ***First apply the row address; RAS signal latches the row address. Then apply the column address, CAS signal latches the address.***
- ***Timing of the memory unit is controlled by a specialized unit which generates RAS and CAS.***
- ***This is asynchronous DRAM***



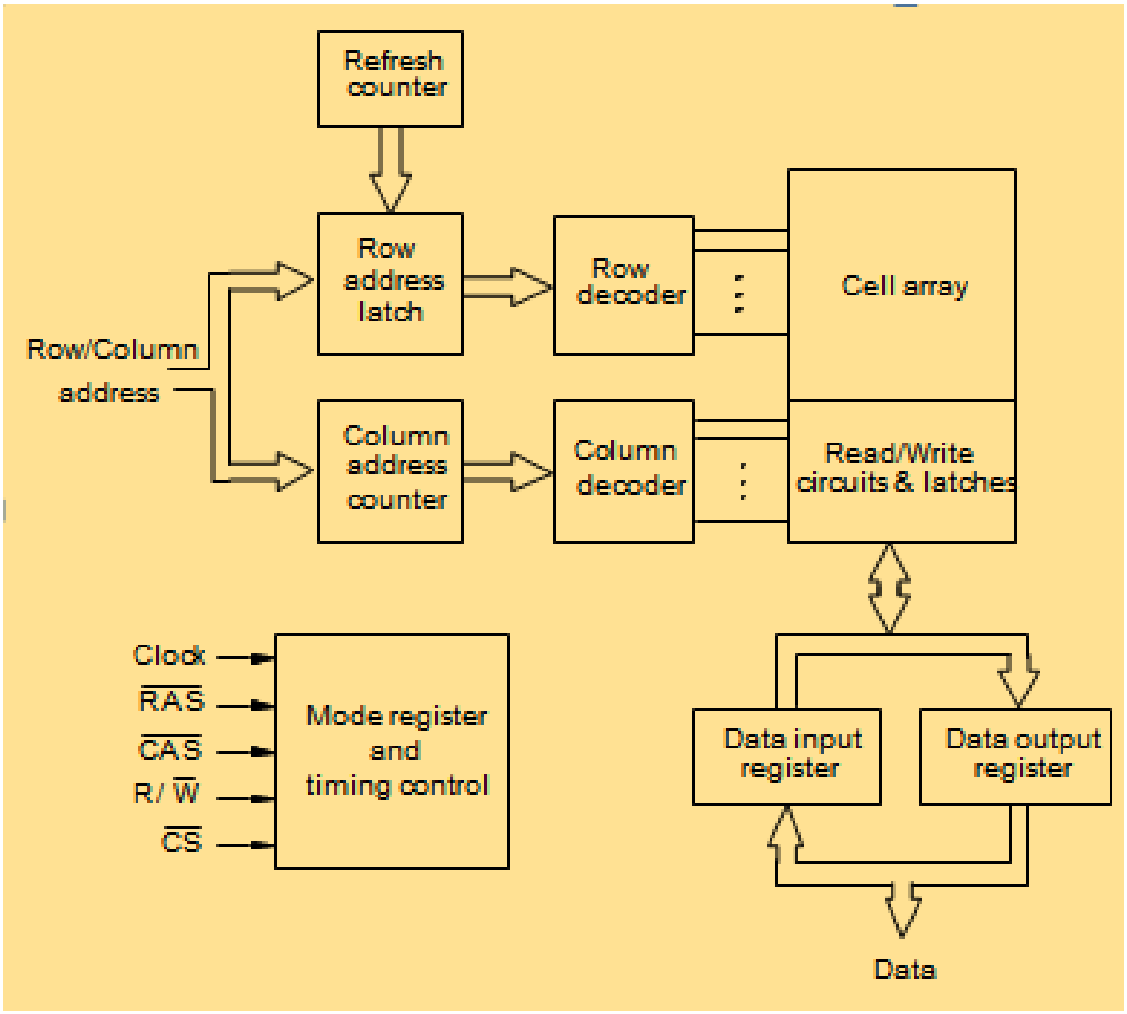
Fast Page Mode

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

- Consecutive sequence of column addresses can be applied under the control signal CAS, without reselecting the row.
    - Allows a block of data to be transferred at a much faster rate than random accesses.
    - A small collection/group of bytes is usually referred to as a block.
- This transfer capability is referred to as the fast page mode feature.

5.2.4 Synchronous DRAMs

In these DRAMs, operation is directly synchronized with a clock signal. The below given figure indicates the structure of an SDRAM.



- 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 into these latches.
- But, if an access is made for refreshing purpose only, it will not change the contents of these latches; it will merely refresh the contents of the cells.
- Data held in the latches that correspond to the selected column(s) are transferred into the output register, thus becoming available on the data output pins.

- SDRAMs have several different modes of operation, which can be selected by writing control information into a mode register. For example, burst operations of different lengths are specified.
- The burst operations use the block transfer capability described before as fast page mode feature.
- In SDRAMs, it is not necessary to provide externally generated pulses on the CAS line to select successive columns. The necessary control signals are provided internally using a column counter and the clock signal. New data can be placed on the data lines in each clock cycles. All actions are triggered by the rising edge of the clock.

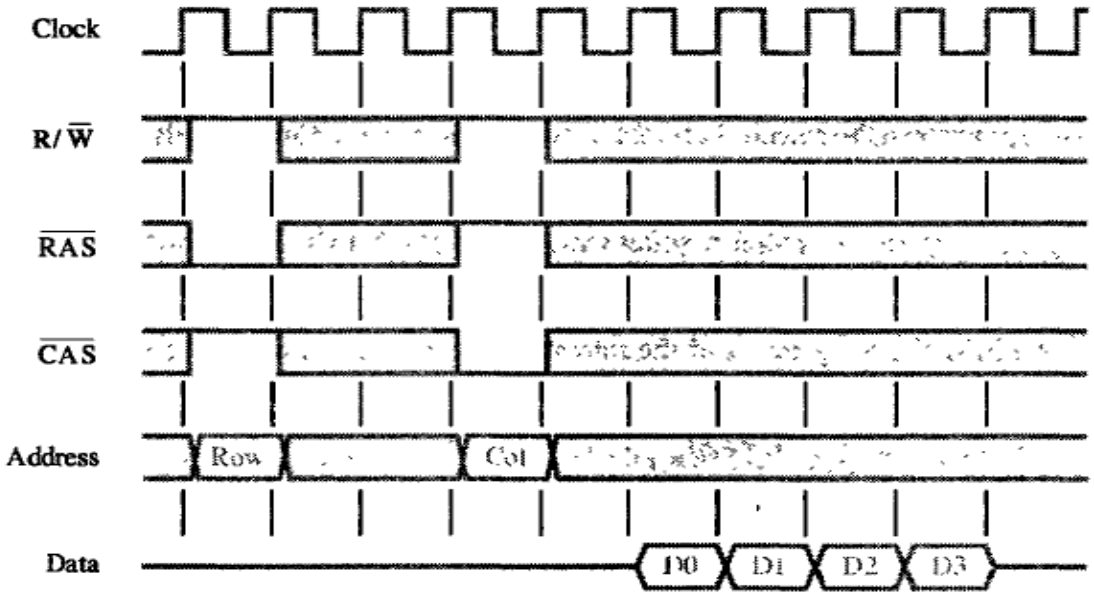


Figure 5.9 Burst read of length 4 in an SDRAM.

The above figure shows the timing diagram for a burst read of length 4.

- First, the row address is latched under control of the RAS signal.
- Then, the column address latched under control of the CAS signal.
- After a delay of one clock cycle, the first set of data bits is placed on the data lines.
- The SDRAM automatically increments the column address to access next three sets of the bits in the selected row, which are placed on the data lines in the next clock cycles.

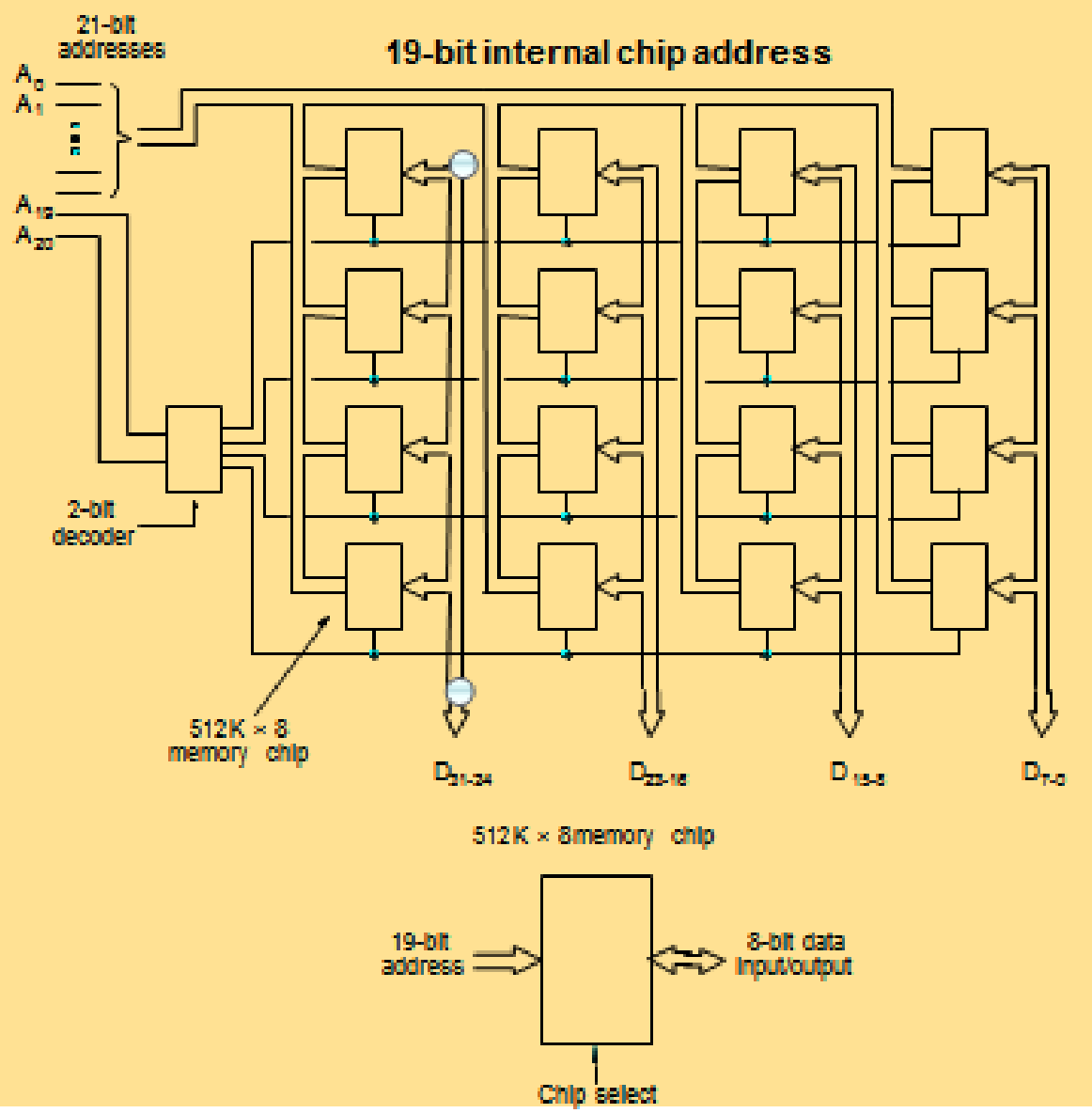
Latency and Bandwidth

- Memory latency is the time it takes to transfer a word of data to or from memory
- Memory bandwidth is the number of bits or bytes that can be transferred in one second.
- DDRSDRAMs- Cell array is organized in two banks.

Double Data Rate- Synchronous DRAMs (DDR- SDRAMs)

To assist the processor in accessing data at high enough rate, the cell array is organized in 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 that are transferred on the successive edges of the clock. This type of SDRAM is called Double Data Rate SDRAM (DDR- SDRAM).

5.2.5 Structure of larger memories



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

Dynamic Memory System

- Large dynamic memory systems can be implemented using DRAM chips in a similar way to static memory systems.
- Placing large memory systems directly on the motherboard will occupy a large amount of space.
    - Also, this arrangement is inflexible since the memory system cannot be expanded easily.
- Packaging considerations have led to the development of larger memory units known as SIMMs (Single In-line Memory Modules) and DIMMs (Dual In-line Memory Modules).
- Memory modules are an assembly of memory chips on a small board that plugs vertically onto a single socket on the motherboard.
    - Occupy less space on the motherboard.
    - Allows for easy expansion by replacement.

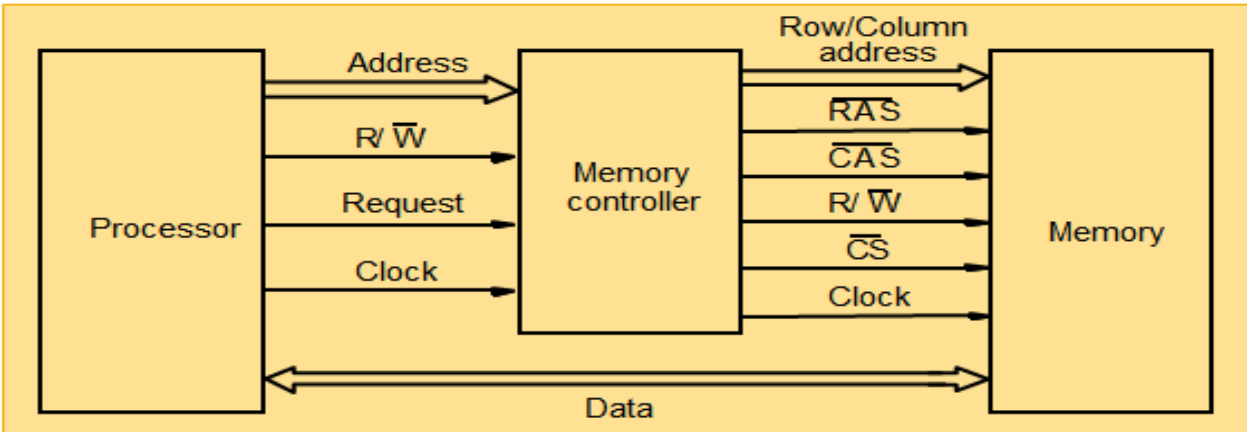
Recall that in a dynamic memory chip, to reduce the number of pins, multiplexed addresses are used.

- Address is divided into two parts:
    - High-order address bits select a row in the array.
    - They are provided first, and latched using RAS signal.
    - Low-order address bits select a column in the row.
    - They are provided later, and latched using CAS signal.
- However, a processor issues all address bits at the same time.
- In order to achieve the multiplexing, memory controller circuit is inserted between the processor and memory.

5.2.6 Memory System Considerations

- Recall that in a dynamic memory chip, to reduce the number of pins, multiplexed addresses are used.
- Address is divided into two parts:
    - High-order address bits select a row in the array.
    - They are provided first, and latched using RAS signal.
    - Low-order address bits select a column in the row.
    - They are provided later, and latched using CAS signal.

- However, a processor issues all address bits at the same time.
- In order to achieve the multiplexing, memory controller circuit is inserted between the processor and memory.



Refresh Operation:-

The Refresh control block periodically generates Refresh, requests, causing the access control block to start a memory cycle in the normal way. This block allows the refresh operation by activating the Refresh Grant line. The access control block arbitrates between Memory Access requests and Refresh requests, with priority to refresh requests in the case of a tie to ensure the integrity of the stored data.

As soon as the Refresh control block receives the Refresh Grant signal, it activates the Refresh line. This causes the address multiplexer to select the Refresh counter as the source and its contents are thus loaded into the row address latches of all memory chips when the RAS signal is activated.

5.3 Semi-Conductor Rom Memories: -

Semiconductor read-only memory (ROM) units are well suited as the control store components in micro programmed processors and also as the parts of the main memory that contain fixed programs or data. The following figure shows a possible configuration for a bipolar ROM cell.

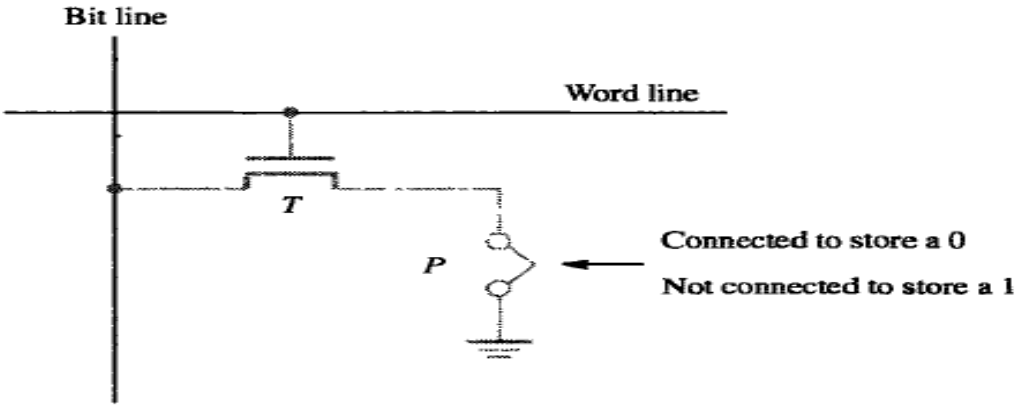


Figure 5.12 A ROM cell.

The word line is normally held at a low voltage. If a word is to be selected, the voltage of the corresponding word line is momentarily raised, which causes all transistors whose emitters are connected to their corresponding bit lines to be turned on. The current that flows from the voltage supply to the bit line can be detected by a sense circuit. The bit positions in which current is detected are read as 1s, and the remaining bits are read as O_s. Therefore, the contents of a given word are determined by the pattern of emitter to bit-line connections similar configurations are possible in MOS technology.

Data are written into a ROM at the time of manufacture programmable ROM (PROM) devices allow the data to be loaded by the user. Programmability is achieved by connecting a fuse between the emitter and the bit line. Thus, prior to programming, the memory contains all 1s. The user can inserts O_s at the required locations by burning out the fuses at these locations using high-current pulses. This process is irreversible.

ROMs are attractive when high production volumes are involved. For smaller numbers, PROMs provide a faster and considerably less expensive approach. Chips which allow the stored data to be erased and new data to be loaded. Such a chip is an erasable, programmable ROM, usually called an EPROM. It provides considerable flexibility during the development phase. An EPROM cell bears considerable resemblance to the dynamic memory cell. As in the case of dynamic memory, information is stored in the form of a charge on a capacitor. The main difference is that the capacitor in an EPROM cell is very well insulated. Its rate of discharge is so low that it retains the stored information for very long periods. To write information, allowing charge to be stored on the capacitor.

The contents of EPROM cells can be erased by increasing the discharge rate of the storage capacitor by several orders of magnitude. This can be accomplished by allowing

ultraviolet light into the chip through a window provided for that purpose, or by the application of a high voltage similar to that used in a write operation. If ultraviolet light is used, all cells in the chip are erased at the same time. When electrical erasure is used, however, the process can be made selective. An electrically erasable EPROM, often referred to as EEPROM. However, the circuit must now include high voltage generation.

Some EEPROM chips incorporate the circuitry for generating these voltages o the chip itself. Depending on the requirements, suitable device can be selected.

Flash memory:

- Has similar approach to EEPROM.
- Read the contents of a single cell, but write the contents of an entire block of cells.
- Flash devices have greater density.
    - Higher capacity and low storage cost per bit.
- Power consumption of flash memory is very low, making it attractive for use in equipment that is battery-driven.
- Single flash chips are not sufficiently large, so larger memory modules are implemented using flash cards and flash drives.

(REFER slides for point wise notes on RoM and types of ROM)

5.4 Speed, Size and Cost

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

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

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

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

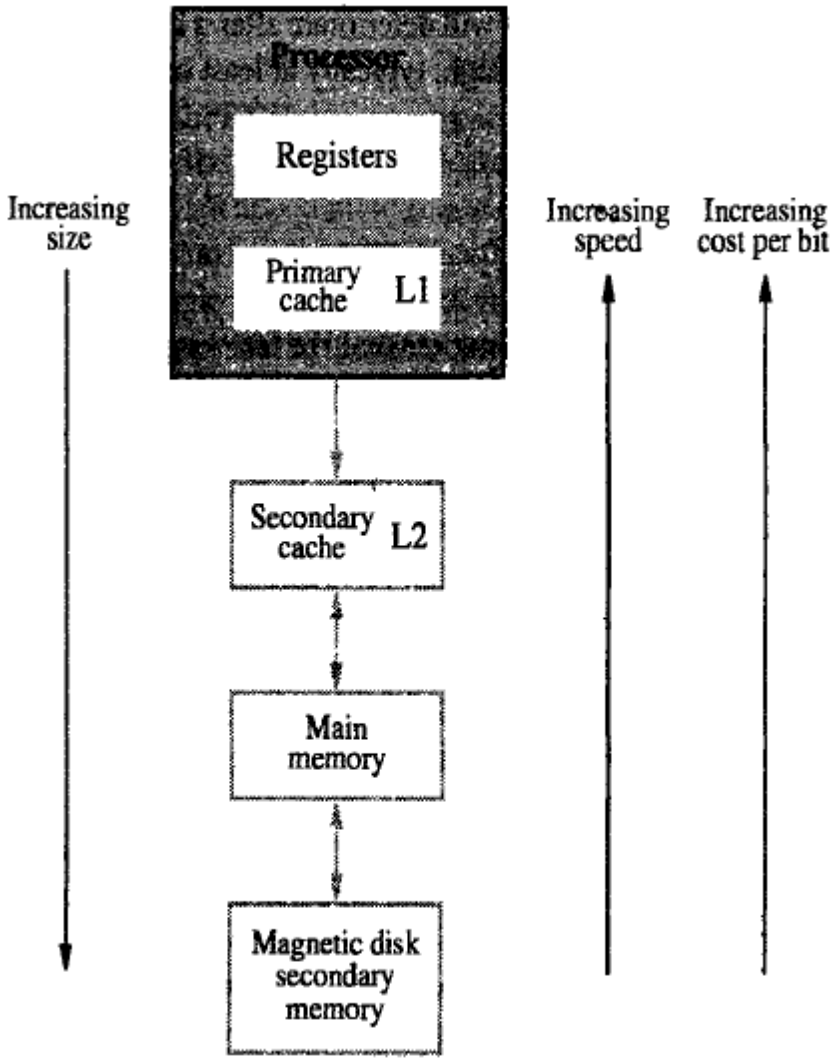


Figure 5.13 Memory hierarchy.

Fastest access is to the data held in processor registers. Registers are at the top of the memory hierarchy. Relatively small amount of memory that can be implemented on the processor chip. This is processor cache. Two levels of cache. Level 1 (L1) cache is on the processor chip. Level 2 (L2) cache is in between main memory and processor. Next level is main memory, implemented as SIMMs. Much larger, but much slower than cache memory. Next level is magnetic disks. Huge amount of inexpensive storage. Speed of memory access is critical, the idea is to bring instructions and data that will be used in the near future as close to the processor as possible.

5.5 Cache memories

Processor is much faster than the main memory. As a result, the processor has to spend much of its time waiting while instructions and data are being fetched from the main memory. This serves as a major obstacle towards achieving good performance. Speed of the main memory cannot be increased beyond a certain point. So we use Cache memories. Cache memory is an architectural arrangement which makes the main memory appear faster to the processor than it really is. Cache memory is based on the property of computer programs known as "locality of reference".

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

Temporal locality of reference: Recently executed instruction is likely to be executed again very soon.

Spatial locality of reference: Instructions with addresses close to a recently instruction are likely to be executed soon.

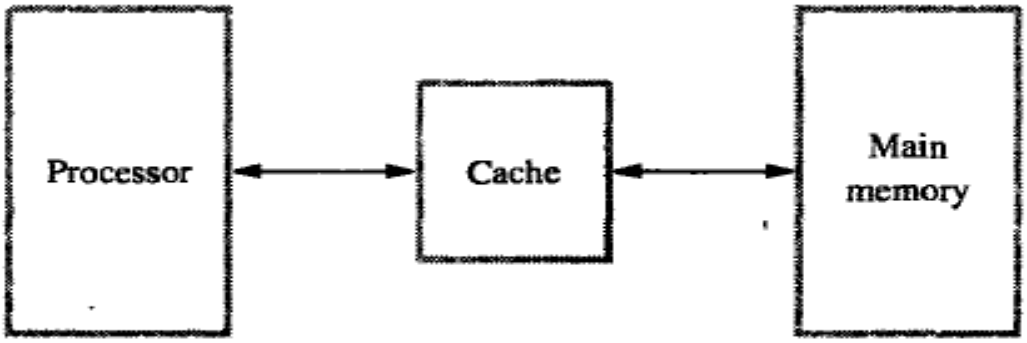


Figure 5.14 Use of a cache memory.

A simple arrangement of cache memory is as shown above.

- Processor issues a Read request, a block of words is transferred from the main memory to the cache, one word at a time.
- Subsequent references to the data in this block of words are found in the cache.
- At any given time, only some blocks in the main memory are held in the cache. Which blocks in the main memory are in the cache is determined by a "mapping function".

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

Cache hit:

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

Read hit: The data is obtained from the cache.

Write hit: Cache has a replica of the contents of the main memory. Contents of the cache and the main memory may be updated simultaneously. This is the write-through protocol.

Update the contents of the cache, and mark it as updated by setting a bit known as the dirty bit or modified bit. The contents of the main memory are updated when this block is replaced. This is write-back or copy-back protocol.

Cache miss:

- If the data is not present in the cache, then a Read miss or Write miss occurs.
- Read miss: Block of words containing this requested word is transferred from the memory. After the block is transferred, the desired word is forwarded to the processor. The desired word may also be forwarded to the processor as soon as it is transferred without waiting for the entire block to be transferred. This is called load-through or early-restart.
- Write-miss: Write-through protocol is used, then the contents of the main memory are updated directly. If write-back protocol is used, the block containing the addressed word is first brought into the cache. The desired word is overwritten with new information.

Cache Coherence Problem:

A bit called as "valid bit" is provided for each block. If the block contains valid data, then the bit is set to 1, else it is 0. Valid bits are set to 0, when the power is just turned on.

When a block is loaded into the cache for the first time, the valid bit is set to 1. Data transfers between main memory and disk occur directly bypassing the cache. When the data on a disk changes, the main memory block is also updated. However, if the data is also resident in the cache, then the valid bit is set to 0.

The copies of the data in the cache, and the main memory are different. This is called the cache coherence problem

Mapping functions: Mapping functions determine how memory blocks are placed in the cache.

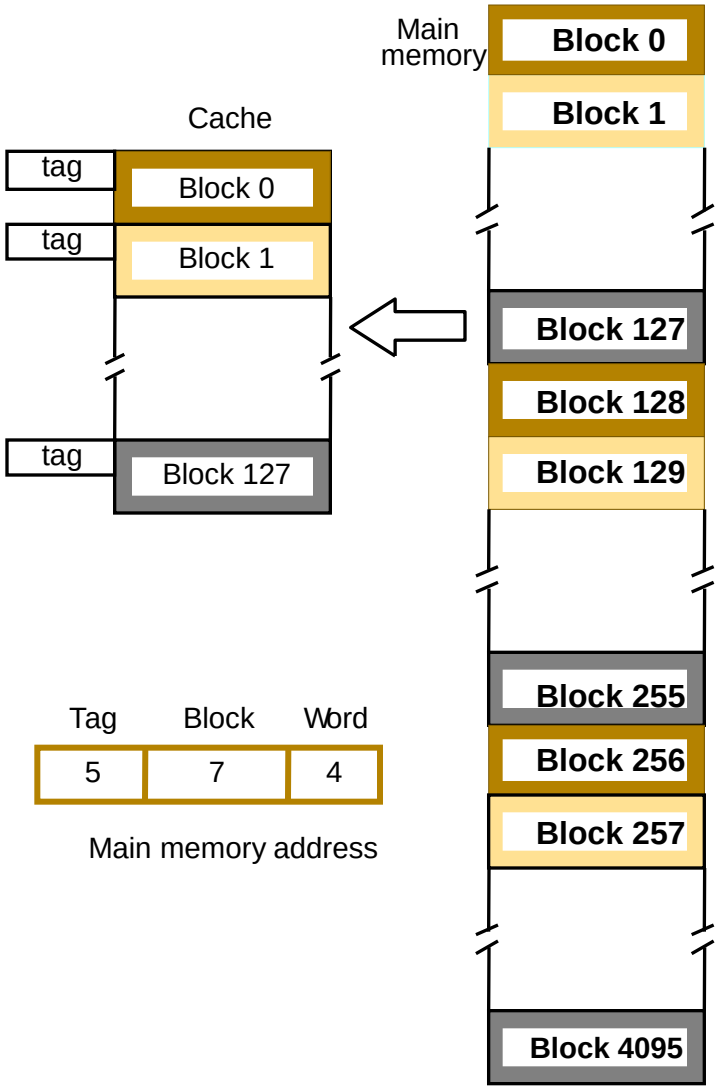
A simple processor example:

- Cache consisting of 128 blocks of 16 words each.
- Total size of cache is 2048 (2K) words.
- Main memory is addressable by a 16-bit address.
- Main memory has 64K words.
- Main memory has 4K blocks of 16 words each.

Three mapping functions can be used.

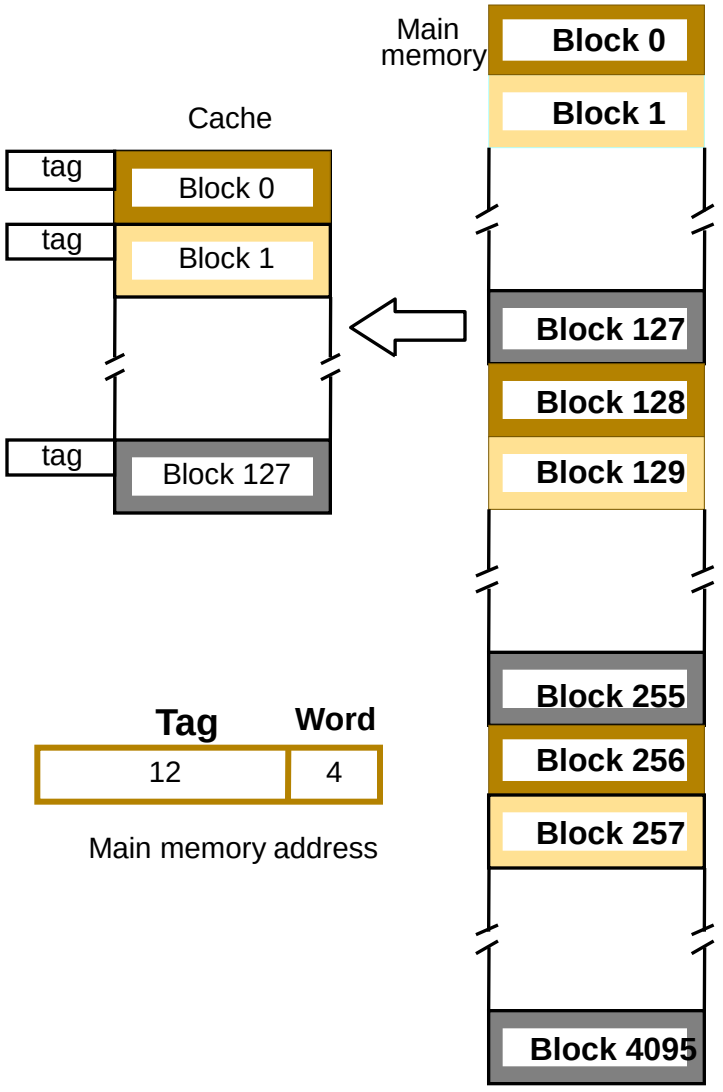
1. Direct mapping
2. Associative mapping
3. Set-associative mapping.

Direct mapping



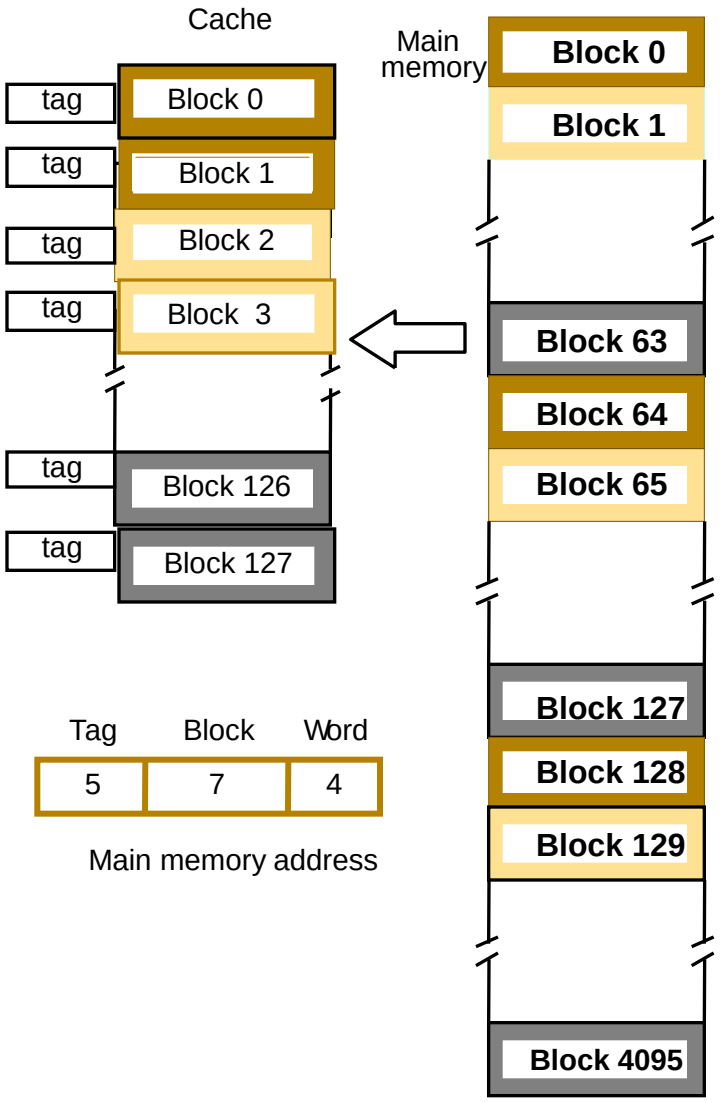
- *Block j of the main memory maps to j modulo 128 of the cache. 0 maps to 0, 129 maps to 1.*
- *More than one memory block is mapped onto the same position in the cache.*
- *May lead to contention for cache blocks even if the cache is not full.*
- *Resolve the contention by allowing new block to replace the old block, leading to a trivial replacement algorithm.*
- *Memory address is divided into three fields:*
  - *Low order 4 bits determine one of the 16 words in a block.*
  - *When a new block is brought into the cache, the the next 7 bits determine which cache block this new block is placed in.*
  - *High order 5 bits determine which of the possible 32 blocks is currently present in the cache. These are tag bits.*
- *Simple to implement but not very flexible.*

Associative mapping



- *Main memory block can be placed into any cache position.*
- *Memory address is divided into two fields:*
  - *Low order 4 bits identify the word within a block.*
  - *High order 12 bits or tag bits identify a memory block when it is resident in the cache.*
- *Flexible, and uses cache space efficiently.*
- *Replacement algorithms can be used to replace an existing block in the cache when the cache is full.*
- *Cost is higher than direct-mapped cache because of the need to search all 128 patterns to determine whether a given block is in the cache.*

Set-Associative mapping



Blocks of cache are grouped into sets.
Mapping function allows a block of the main memory to reside in any block of a specific set.
Divide the cache into 64 sets, with two blocks per set.
Memory block 0, 64, 128 etc. map to block 0, and they can occupy either of the two positions.
Memory address is divided into three fields:

- *6 bit field determines the set number.*
- *High order 6 bit fields are compared to the tag fields of the two blocks in a set.*

Set-associative mapping combination of direct and associative mapping.
Number of blocks per set is a design parameter.

- *One extreme is to have all the blocks in one set, requiring no set bits (fully associative mapping).*
- *Other extreme is to have one block per set, is the same as direct mapping.*

Replacement Algorithm

In a direct-mapped cache, the position of each block is fixed, hence no replacement strategy exists. In associative and set-associative caches, when a new block is to be brought into the cache and all the Positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This is important issue because the decision can be factor in system performance. The objective is to keep blocks in the cache that are likely to be referenced in the near future. Its not easy to determine which blocks are about to be referenced. The property of locality of reference gives a clue to a reasonable strategy. When a block is to be over written, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the least recently used (LRU) block, and technique is called the LRU Replacement algorithm. The LRU algorithm has been used extensively for many access patterns, but it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache. Performance of LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.

Solved Problems:-

1. A block set associative cache consists of a total of 64 blocks divided into 4 block sets. The MM contains 4096 blocks each containing 128 words.

a) How many bits are there in MM address?

b) How many bits are there in each of the TAG, SET & word fields

Solution:- Number of sets = 64/4 = 16

Set bits = 4(2^4 = 16)

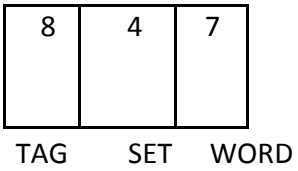
Number of words = 128

Word bits = 7 bits (2^7 = 128)

MM capacity : 4096 x 128 (2^{12} x 2^7 = 2^{19})

a) Number of bits in memory address = 19 bits

b)



TAG bits = 19 – (7+4) = 8 bits.

2. A computer system has a MM capacity of a total of 1M 16 bits words. It also has a 4K words cache organized in the block set associative manner, with 4 blocks per set & 64 words per block. Calculate the number of bits in each of the TAG, SET & WORD fields of MM address format.

Solution: Capacity: 1M (2^{20} = 1M)

Number of words per block = 64

Number of blocks in cache = 4k/64 = 64

Number of sets = 64/4 = 16

Set bits = 4 (2^4 = 16)

Word bits = 6 bits (2^6 = 64)

Tag bits = 20-(6+4) = 10 bits

MM address format: 10 tag bits, 6 word bits and 4 set bits.

5.6 PERFORMANCE CONSIDERATIONS

A key design objective of a computer system is to achieve the best possible performance at the lowest possible cost. Price/performance ratio is a common measure of success.

Performance of a processor depends on: How fast machine instructions can be brought into the processor for execution. How fast the instructions can be executed.

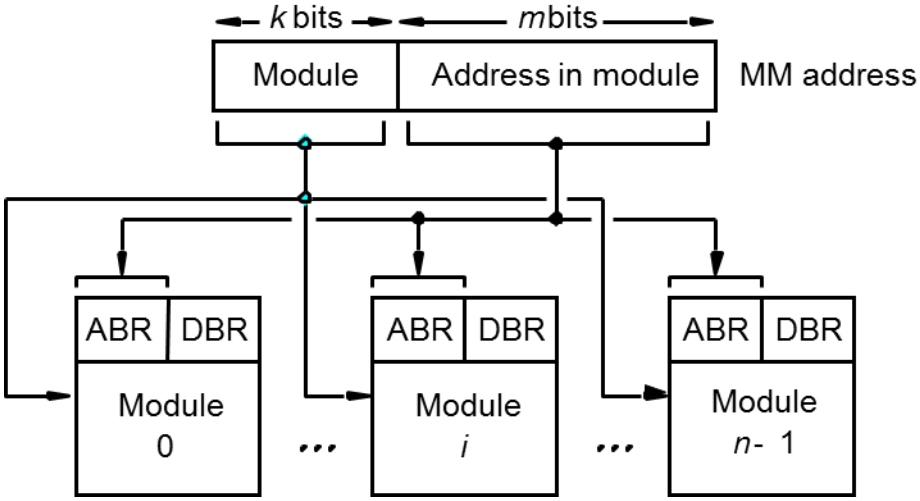
Interleaving

Divides the memory system into a number of memory modules. Each module has its own address buffer register (ABR) and data buffer register (DBR). Arranges addressing so that successive words in the address space are placed in different modules. When requests for memory access involve consecutive addresses, the access will be to different modules.

Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.

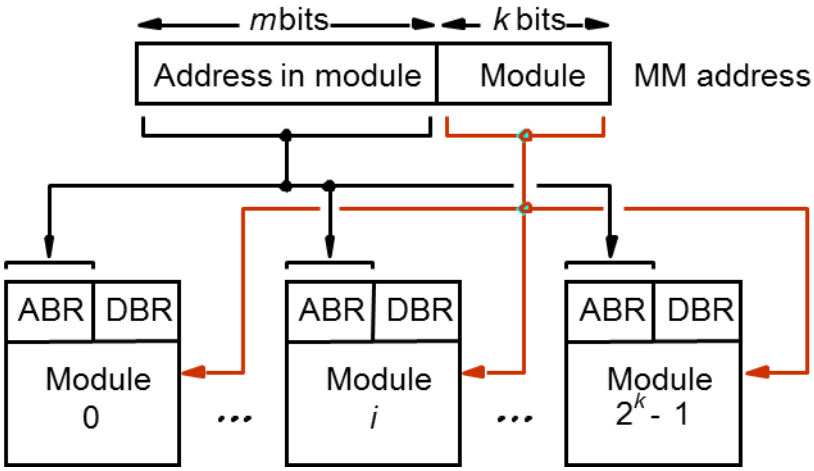
Methods of address layout:

1)



Consecutive words are placed in a module. High-order k bits of a memory address determine the module. Low-order m bits of a memory address determine the word within a module. When a block of words is transferred from main memory to cache, only one module is busy at a time.

2)



Consecutive words are located in consecutive modules. Consecutive addresses can be located in consecutive modules. While transferring a block of data, several memory modules can be kept busy at the same time.

Hit rate and miss penalty

- The number of hits stated as fraction of all attempted accesses is called the **hit rate** and the number of misses stated as a fraction of all attempted accesses is called the **miss rate.**
- The extra time needed to bring the desired information into cache is called as miss penalty.
- Hit rate can be improved by increasing block size, while keeping cache size constant
- Block sizes that are neither very small nor very large give best results.
- Miss penalty can be reduced if load-through approach is used when loading new blocks into cache.
- Avg access time experienced by processor is

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

h- hitrate, M – miss penalty, C- time to access info from cache.

Caches on the processor chips

- In high performance processors 2 levels of caches are normally used.
- Avg access time in a system with 2 levels of caches is
- T ave = h1c1+(1-h1)h2c2+(1-h1)(1-h2)M

h1- hitrate of Cache 1, h2- hit rate of Cache 2, M – miss penalty, c1- time to access info from cache1, c2- time to access info from cache2.

Write buffer

Write-through: Each write operation involves writing to the main memory. If the processor has to wait for the write operation to be complete, it slows down the processor. Processor does not depend on the results of the write operation. Write buffer can be included for temporary storage of write requests. Processor places each write request into the buffer and continues execution. If a subsequent Read request references data which is still in the write buffer, then this data is referenced in the write buffer.

Write-back: Block is written back to the main memory when it is replaced. If the processor waits for this write to complete, before reading the new block, it is slowed down.Fast write buffer can hold the block to be written, and the new block can be read first.

In addition to these prefetching and lock up free cache can also be used to improve performance of the processor.

5.7 Virtual Memory

An important challenge in the design of a computer system is to provide a large, fast memory system at an affordable cost. Cache memories were developed to increase the effective speed of the memory system. Virtual memory is an architectural solution to increase the effective size of the memory system.

The addressable memory space depends on the number of address bits in a computer. For example, if a computer issues 32-bit addresses, the addressable memory space is 4G bytes. Physical main memory in a computer is generally not as large as the entire possible addressable space. Physical memory typically ranges from a few hundred megabytes to 1G bytes. Large programs that cannot fit completely into the main memory have their parts stored on secondary storage devices such as magnetic disks. Pieces of programs must be transferred to the main memory from secondary storage before they can be executed.

Techniques that automatically move program and data between main memory and secondary storage when they are required for execution are called virtual-memory techniques. Programs and processors reference an instruction or data independent of the size of the main memory. Processor issues binary addresses for instructions and data. These binary addresses are called logical or virtual addresses. Virtual addresses are translated into physical addresses by a combination of hardware and software subsystems. If virtual address refers to a part of the program that is currently in the main memory, it is accessed immediately. If the address refers to a part of the program that is not currently in the main memory, it is first transferred to the main memory before it can be used.

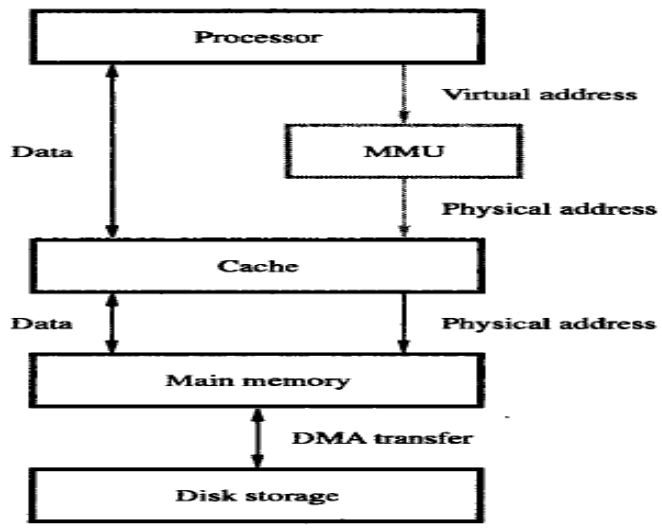


Figure 5.26 Virtual memory organization.

Memory management unit (MMU) translates virtual addresses into physical addresses. If the desired data or instructions are in the main memory they are fetched as described previously. If the desired data or instructions are not in the main memory, they must be transferred from secondary storage to the main memory. MMU causes the operating system to bring the data from the secondary storage into the main memory.

Address Translation

Assume that program and data are composed of fixed-length units called pages. A page consists of a block of words that occupy contiguous locations in the main memory. Page is a basic unit of information that is transferred between secondary storage and main memory. Size of a page commonly ranges from 2K to 16K bytes. Pages should not be too small, because the access time of a secondary storage device is much larger than the main memory. Pages should not be too large, else a large portion of the page may not be used, and it will occupy valuable space in the main memory.

Concepts of virtual memory are similar to the concepts of cache memory.

Cache memory: Introduced to bridge the speed gap between the processor and the main memory. Implemented in hardware.

Virtual memory: Introduced to bridge the speed gap between the main memory and secondary storage. Implemented in part by software.

Each virtual or logical address generated by a processor is interpreted as a virtual page number (high-order bits) plus an offset (low-order bits) that specifies the location of a particular byte within that page. Information about the main memory location of each page is kept in the page table. Main memory address where the page is stored. Current status of the page. Area of the main memory that can hold a page is called as page frame. Starting address of the page table is kept in a page table base register.

Virtual page number generated by the processor is added to the contents of the page table base register. This provides the address of the corresponding entry in the page table. The contents of this location in the page table give the starting address of the page if the page is currently in the main memory.

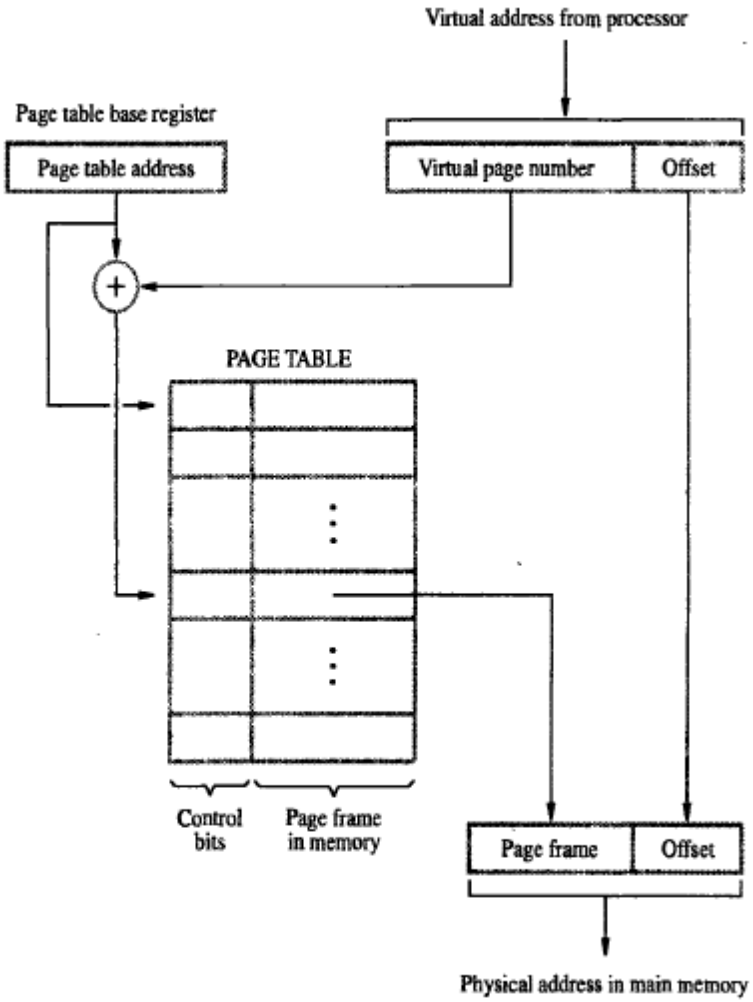


Figure 5.27 Virtual-memory address translation.

Page table entry for a page also includes some control bits which describe the status of the page while it is in the main memory. One bit indicates the validity of the page. Indicates whether the page is actually loaded into the main memory. Allows the operating system to invalidate the page without actually removing it. One bit indicates whether the page has been modified during its residency in the main memory. This bit determines whether the page should be written back to the disk when it is removed from the main memory. Similar to the dirty or modified bit in case of cache memory. Other control bits for various other types of restrictions that may be imposed. For example, a program may only have read permission for a page, but not write or modify permissions.

The page table is used by the MMU for every read and write access to the memory. Ideal location for the page table is within the MMU. Page table is quite large. MMU is implemented as part of the processor chip. Impossible to include a complete page table on the chip. Page table is kept in the main memory. A copy of a small portion of the page table can

be accommodated within the MMU. Portion consists of page table entries that correspond to the most recently accessed pages.

A small cache called as Translation Lookaside Buffer (TLB) is included in the MMU. TLB holds page table entries of the most recently accessed pages. The cache memory holds most recently accessed blocks from the main memory. Operation of the TLB and page table in the main memory is similar to the operation of the cache and main memory. Page table entry for a page includes: Address of the page frame where the page resides in the main memory. Some control bits. In addition to the above for each page, TLB must hold the virtual page number for each page.

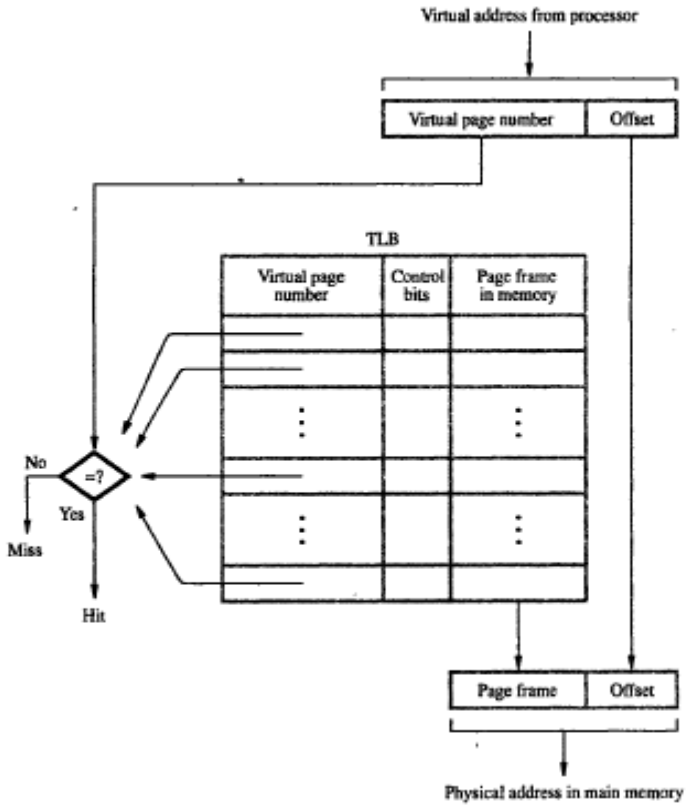


Figure 5.28 Use of an associative-mapped TLB.

Associative-mapped TLB: High-order bits of the virtual address generated by the processor select the virtual page.These bits are compared to the virtual page numbers in the TLB. If there is a match, a hit occurs and the corresponding address of the page frame is read. If there is no match, a miss occurs and the page table within the main memory must be consulted. Set-associative mapped TLBs are found in commercial processors.

If a program generates an access to a page that is not in the main memory a page fault is said to occur. Whole page must be brought into the main memory from the disk, before the execution can proceed. Upon detecting a page fault by the MMU, following actions occur: MMU asks the operating system to intervene by raising an exception. Processing of the active task which caused the page fault is interrupted. Control is transferred to the operating system. Operating system copies the requested page from secondary storage to the main memory. Once the page is copied, control is returned to the task which was interrupted.

5.8 Secondary Storage

1. Magnetic Disk Drives: Hard disk Drive organization:

The modern hard disk drive is a system in itself. It contains not only the disks that are used as the storage medium and the read write heads that access the raw data encoded on them, but also the signal conditioning circuitry and the interface electronics that separate the system user from the details & getting bits on and off the magnetic surface. The drive has 4 platters with read/write heads on the top and bottom of each platter. The drive rotates at a constant 3600rpm.

Platters and Read/Write Heads: - The heart of the disk drive is the stack of rotating platters that contain the encoded data, and the read and write heads that access that data. The drive contains five or more platters. There are read/write heads on the top and bottom of each platter, so information can be recorded on both surfaces. All heads move together across the platters. The platters rotate at constant speed usually 3600 rpm.

Drive Electronics: - The disk drive electronics are located on a printed circuit board attached to the disk drive. After a read request, the electronics must seek out and find the block requested, stream is off of the surface, error check and correct it, assembly into bytes, store it in an on-board buffer and signal the processor that the task is complete. To assist in the task, the drive electronics include a disk controller, a special purpose processor.

Data organization on the Disk:- The drive needs to know where the data to be accessed is located on the disk. In order to provide that location information, data is organized on the disk platters by tracks and sectors. Fig below shows simplified view of the organization of tracks and sectors on a disk. The fig. shows a disk with 1024 tracks, each of which has 64 sectors. The head can determine which track it is on by counting tracks from a known

location and sector identities are encoded in a header written on the disk at the front of each sector. The number of bytes per sector is fixed for a given disk drive, varying in size from 512 bytes to 2KB. All tracks with the same number, but as different surfaces, form a cylinder. The information is recorded on the disk surface 1 bit at a time by magnetizing a small area on the track with the write head. That bit is detected by sending the direction of that magnetization as the magnetized area passes under the read head as shown in fig below.

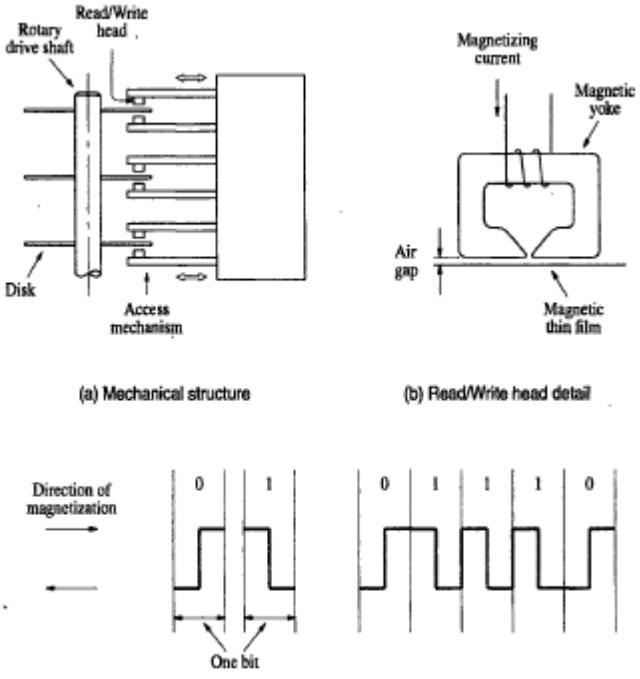


Figure 5.29 Magnetic disk principles.

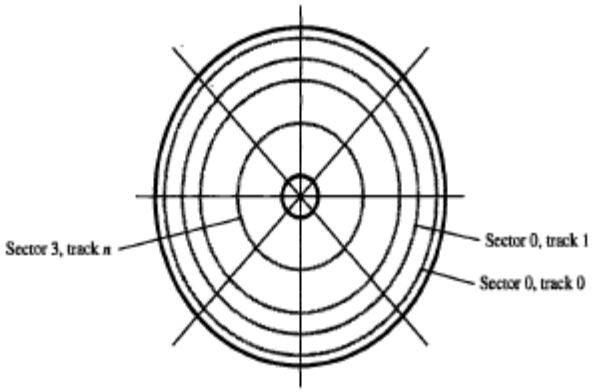


Figure 5.30 Organization of one surface of a disk.

The header usually contains both synchronization and location information. The synchronization information allows the head positioning circuitry to keep the heads centered on the track and the location information allows the disk controller to determine the sectors & identifies as the header passes, so that the data can be captured if it is read or stored, if it is a write. The 12 bytes of ECC (Error Correcting Code) information are used to detect and correct errors in the 512 byte data field.

Disk Drive Dynamic Properties: -

Dynamic properties are those that deal with the access time for the reading and writing of data. The calculation of data access time is not simple. It depends not only as the rotational speed of the disk, but also the location of the read/write head when it begins the access. There are several measures of data access times.

1. **Seek time: -** Is the average time required to move the read/write head to the desired track. Actual seek time which depend on where the head is when the request is received and how far it has to travel, but since there is no way to know what these values will be when an access request is made, the average figure is used. Average seek time must be determined by measurement. It will depend on the physical size of the drive components and how fast the heads can be accelerated and decelerated. Seek times are generally in the range of 8-20 m sec and have not changed much in recent years.

2. **Track to track access time: -** Is the time required to move the head from one track to adjoining one. This time is in the range of 1-2 m sec.

3. **Rotational latency: -** Is the average time required for the needed sector to pass under head once and head has been positioned once at the correct track. Since on the average the desired sector will be half way around the track from where the head is when the head first arrives at the track, rotational latency is taken to be ½ the rotation time. Current rotation speeds are from 3600 to 7200 rpm, which yield rotational latencies in the 4-8 ms range.

4. **Average Access time:-** Is equal to seek time plus rotational latency.

5. **Burst rate: -** Is the maximum rate at which the drive produces or accepts data once the head reaches the desired sector, It is equal to the rate at which data bits stream by the head, provided that the rest of the system can produce or accept data at that rate

   Burst rate (byte/sec) = rows/sec * sector/row * bytes/sector

6. **Sustained data rate: -** Is the rate at which data can be accessed over a sustained period of time.

Optical Disks

Compact Disk (CD) Technology:- The optical technology that is used for CD system is based on laser light source. A laser beam is directed onto the surface of the spinning disk. Physical indentations in the surface are arranged along the tracks of the disk. They reflect the focused beam towards a photo detector, which detects the stored binary patterns.

The laser emits a coherent light beam that is sharply focused on the surface of the disk. Coherent light consists of Synchronized waves that have the same wavelength. If a coherent light beam is combined with another beam of the same kind, and the two beams are in phase, then the result will be brighter beam. But, if a photo detector is used to detect the beams, it will detect a bright spot in the first case and a dark spot in the second case.A cross section of a small portion of a CD shown in fig. below. The bottom payer is Polycarbonate plastic, which functions as a clear glass base. The surface of this plastic is Programmed to store data by indenting it with pits. The unindented parts are called lands. A thin layer of reflecting aluminium material is placed on top of a programmed disk. The aluminium is then covered by a protective acrylic. Finally the topmost layer is deposited and stamped with a label.

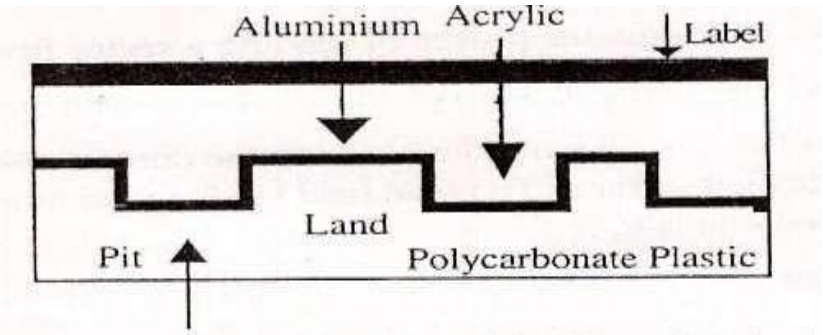


Fig. 16 Optical disk (Cross Section)

The laser source and the Photo detector are positioned below the polycarbonate plastic. The emitted bean travels through this plastic, reflects off the aluminium layer and travels back toward photo detector.

Some important optical disks are listed below

1. CD-ROM
2. CD-RWs (CD-re writables)
3. DVD technology (Digital Versatile disk)

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 the sign-and-magnitude system, negative value is obtained by changing the MSB from 0 to 1 of the corresponding positive value. For example, +5 is represented by 0101 &
-5 is represented by 1101.

• In 1's complement representation, negative values are obtained by complementing each bit of the corresponding positive number. For example, -3 is obtained by complementing each bit in 0011 to yield 1100. (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 the 2's complement system, forming the 2's complement of a number is done by subtracting that number from 2^n.

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

• The 2's complement system yields the most efficient way to carry out addition and subtraction operations.

B	Values represented		
$b_3b_2b_1b_0$	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 2.1 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

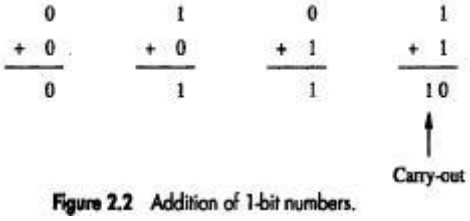


Figure 2.2 Addition of 1-bit numbers.

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.

1) To add two numbers, add their n-bits and ignore the carry-out signal from the MSB position. The sum will be algebraically correct value as long as the answer is in the range -2^{n-1} through $+2^{n-1}-1$ (Figure 2.4).

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 the result of an arithmetic operation is outside the representable-range, an arithmetic overflow is said to occur.

• For example, when using 4-bit signed numbers, if we try to add the numbers +7 and +4, the output sum S is 1011, 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.

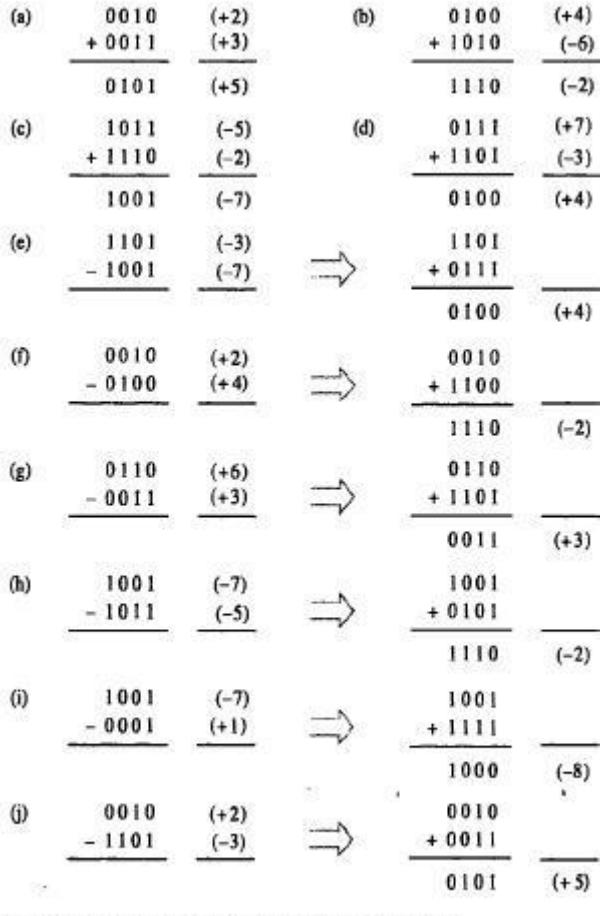


Figure 2.4 2's-complement add and subtract operations.

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.(Fig 6.2).

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

$$s_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i\bar{y}_i\bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

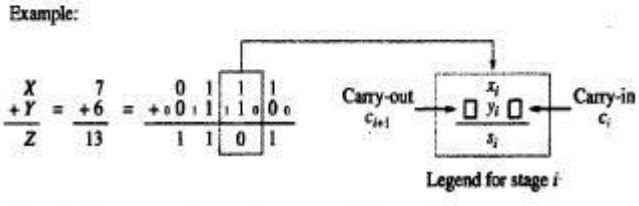
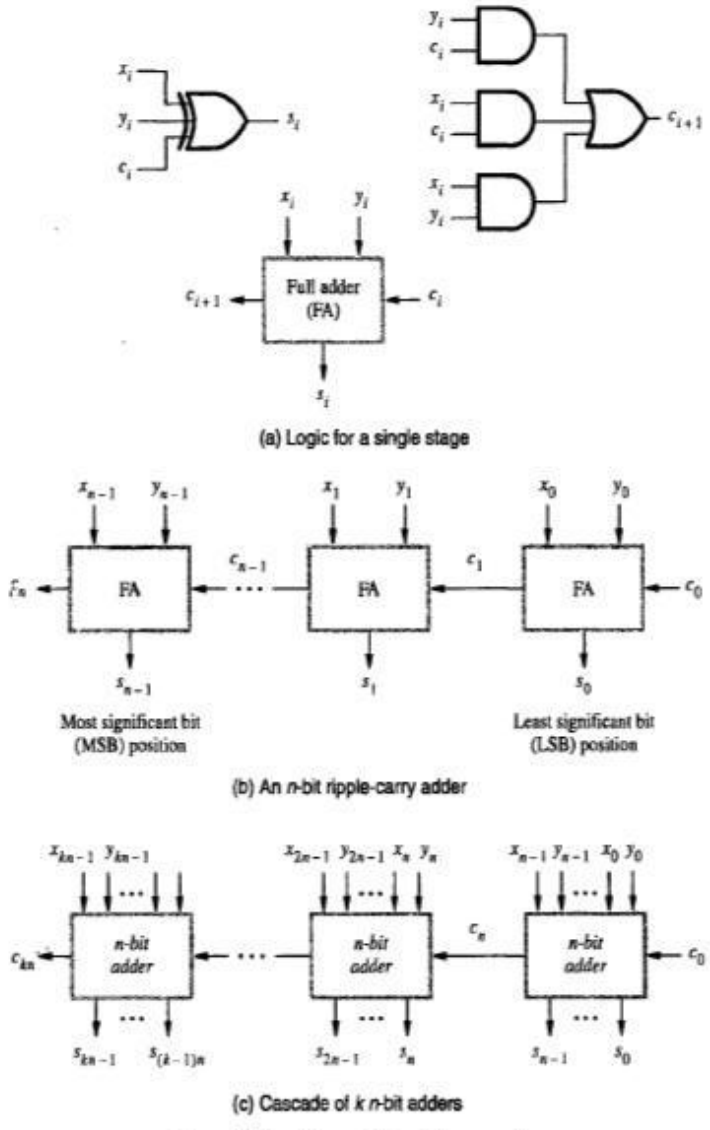


Figure 6.1 Logic specification for a stage of binary addition.



(a) Logic for a single stage

(b) An *n*-bit ripple-carry adder

(c) Cascade of *k* *n*-bit adders

Figure 6.2 Logic for addition of binary vectors.

ADDITION/SUBTRACTION LOGIC UNIT

- The n-bit adder can be used to add 2's complement numbers X and Y (Figure 6.3).
- 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.

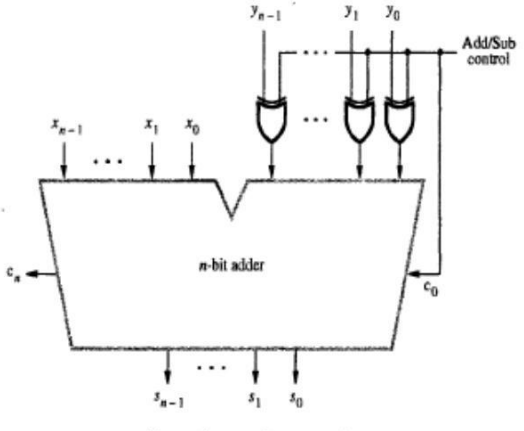


Figure 6.3 Binary addition-subtraction logic network.

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:

i) Use the fastest possible electronic-technology in implementing the ripple-carry design

ii) 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 6.4).

• 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

$c_1=G_0+P_0c_0$

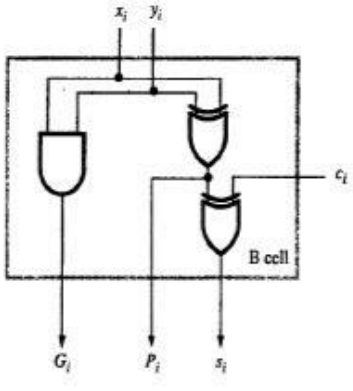
$c_2=G_1+P_1G_0+P_1P_0c_0$

$c_3=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0c_0$

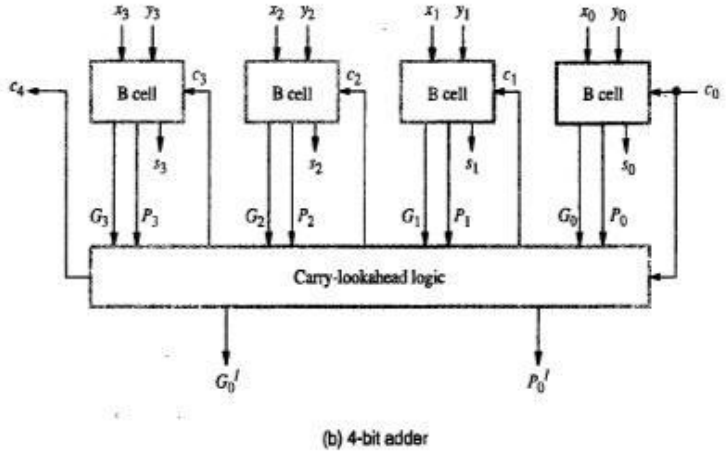
$c_4=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0c_0$

• 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 6.4 4-bit carry-lookahead adder.

HIGHER-LEVEL GENERATE & PROPAGATE FUNCTIONS

- 16-bit adder can be built from four 4-bit adder blocks (Figure 6.5).
- 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.

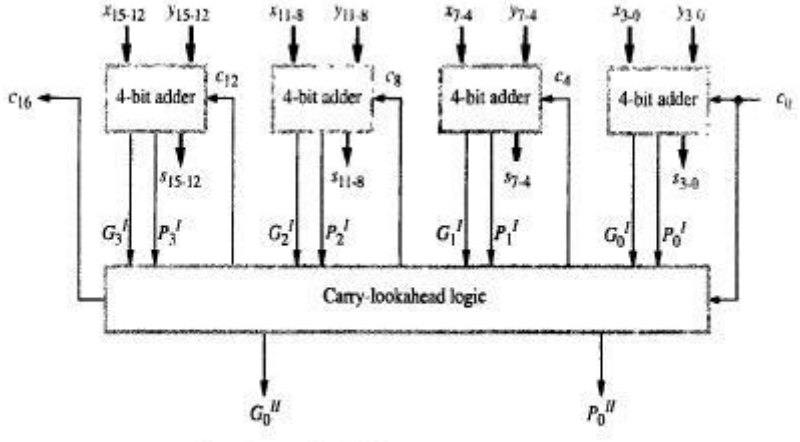
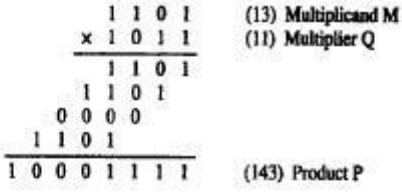
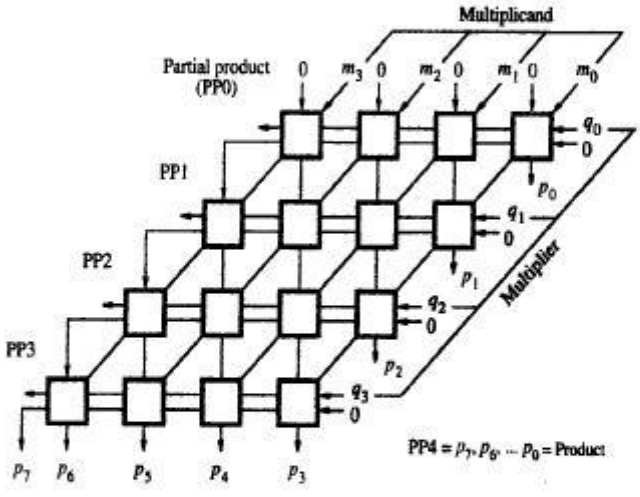


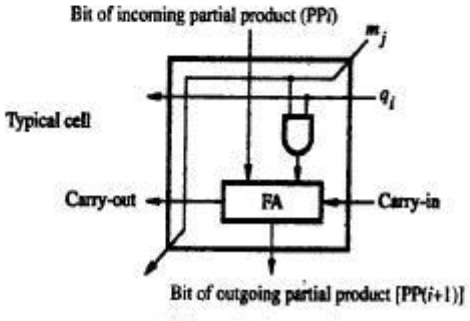
Figure 6.5 16-bit carry-lookahead adder built from 4-bit adders (see Figure 6.4b).

MULTIPLICATION OF POSITIVE NUMBERS



(a) Manual multiplication algorithm





(b) Array implementation

Figure 6.6 Array multiplication of positive binary operands.

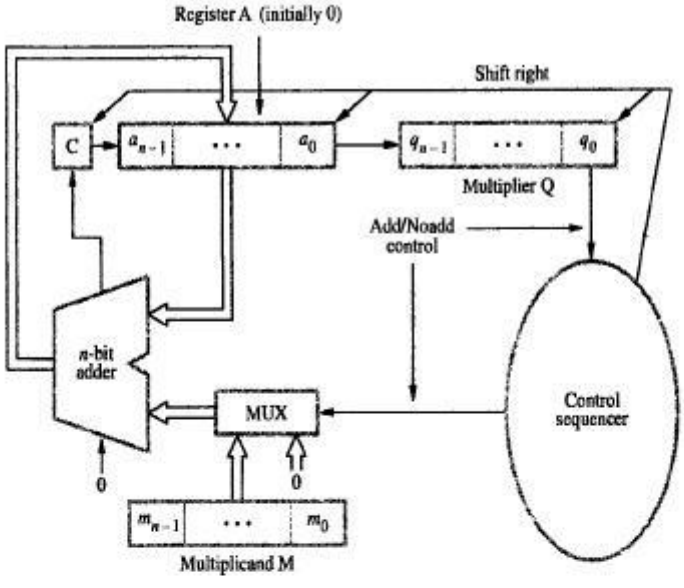
ARRAY MULTIPLICATION

• The main component in each cell is a full adder(FA)..

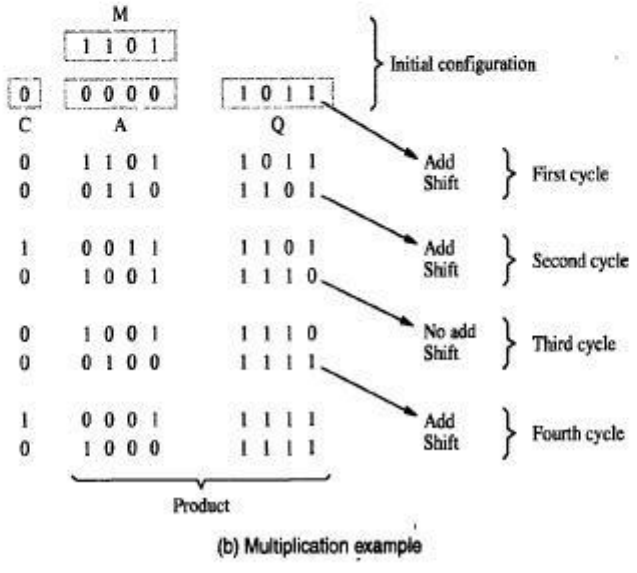
• The AND gate in each cell determines whether a multiplicand bit m_j, is added to the incoming partial-product bit, based on the value of the multiplier bit q_i (Figure 6.6).

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 6.7).
- 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



(b) Multiplication example

Figure 6.7 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 6.9-6.12).
- 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

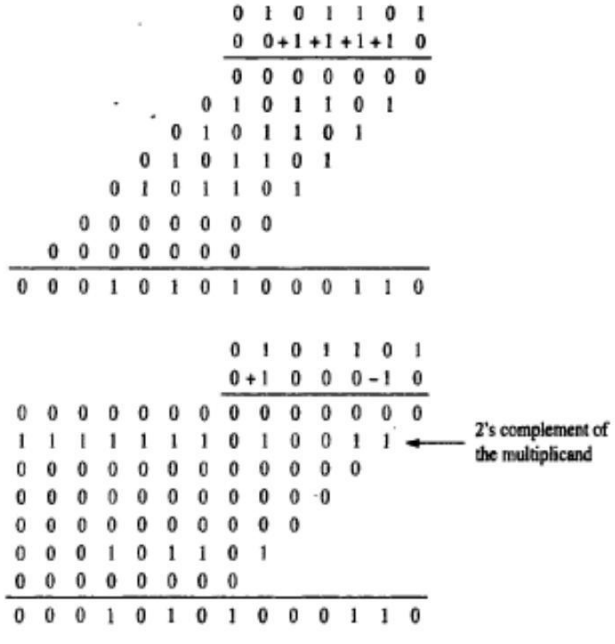


Figure 6.9 Normal and Booth multiplication schemes.

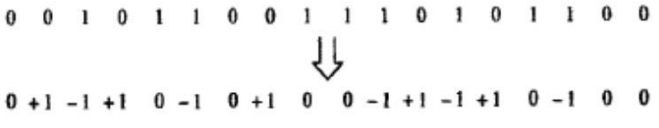


Figure 6.10 Booth recoding of a multiplier.

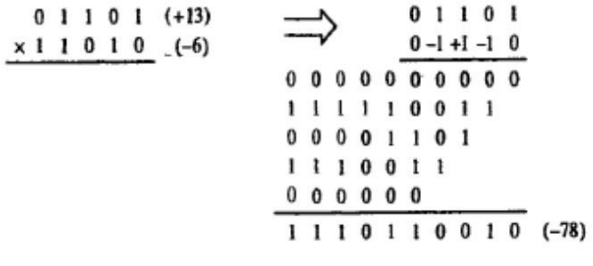


Figure 6.11 Booth multiplication with a negative multiplier.

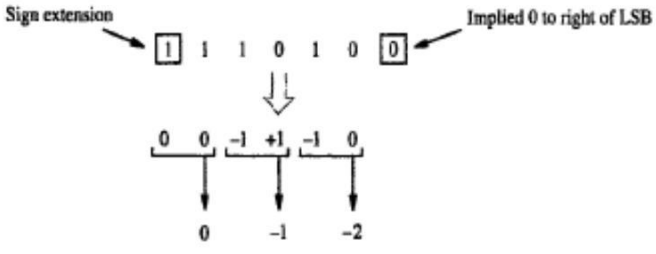
Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 6.12 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 6.14 & 6.15).
- 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 6.14 Multiplier bit-pair recoding.

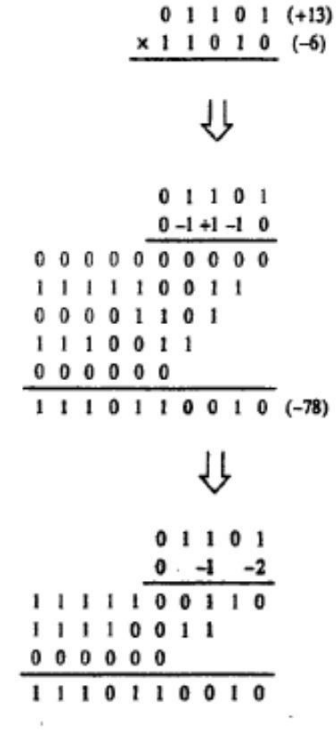
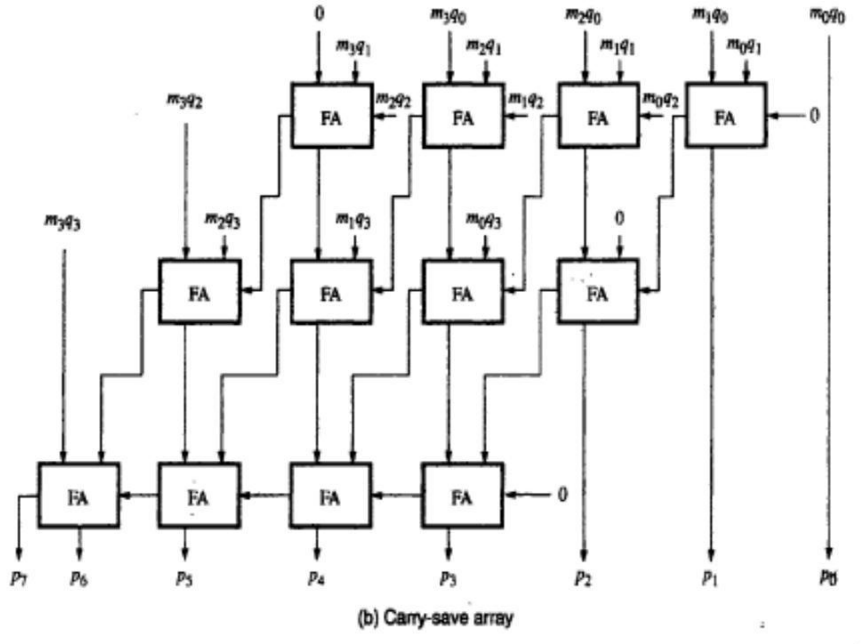


Figure 6.15 Multiplication requiring only $n/2$ summands.

CARRY-SAVE ADDITION OF SUMMANDS

• Consider the array for 4*4 multiplication. (Figure 6.16 & 6.18).

• 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 6.16 Ripple-carry and carry-save arrays for the multiplication operation M × Q = P for 4-bit operands.

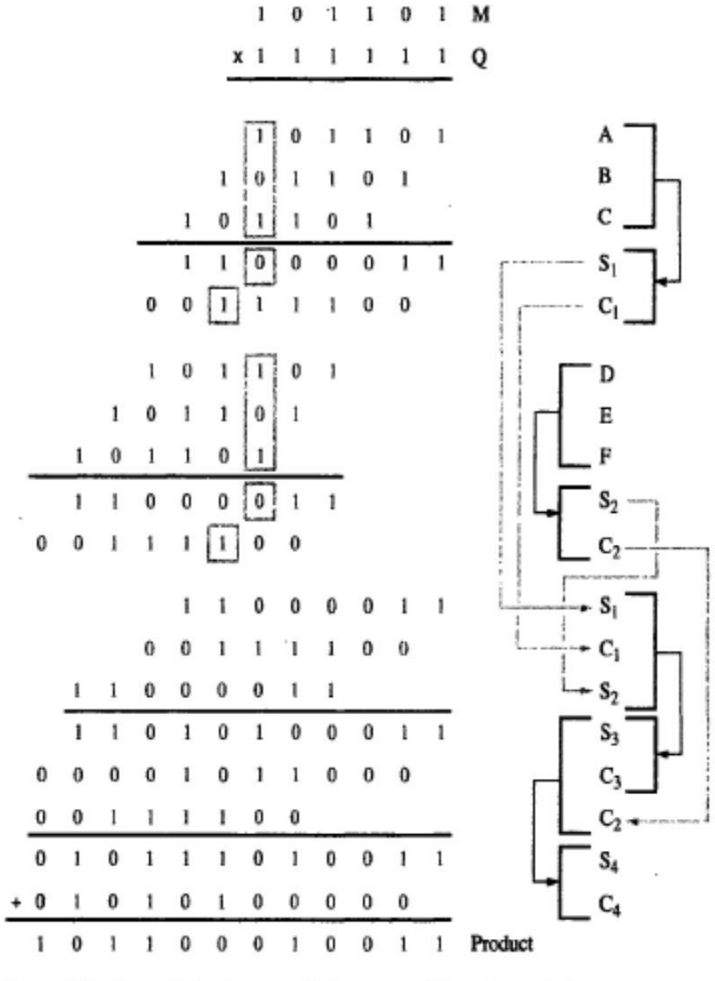


Figure 6.18 The multiplication example from Figure 6.17 performed using carry-save addition.

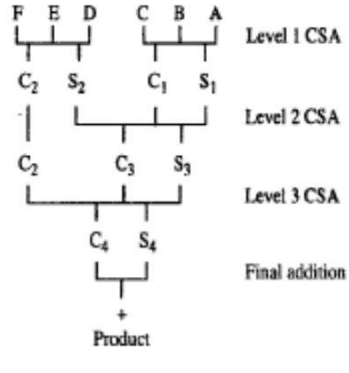


Figure 6.19 Schematic representation of the carry-save addition operations in Figure 6.18.

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 6.21).
- After division operation, the n-bit quotient is in register Q,
  and the remainder is in register A.

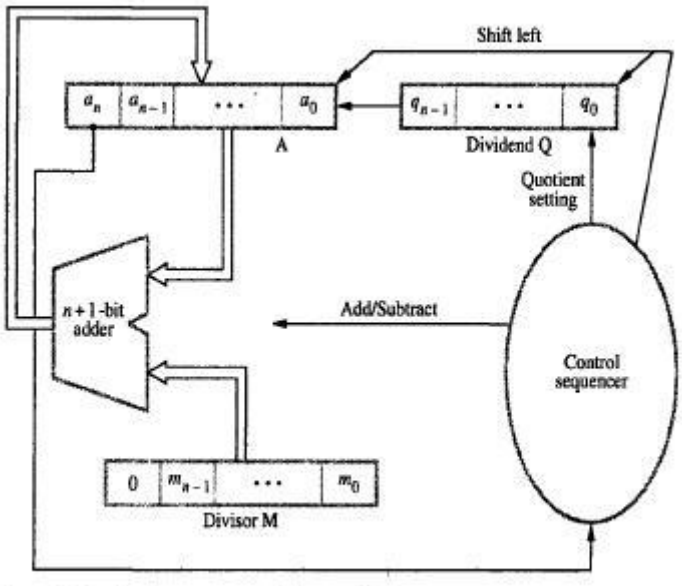


Figure 6.21 Circuit arrangement for binary division.

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 6.23).

  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).

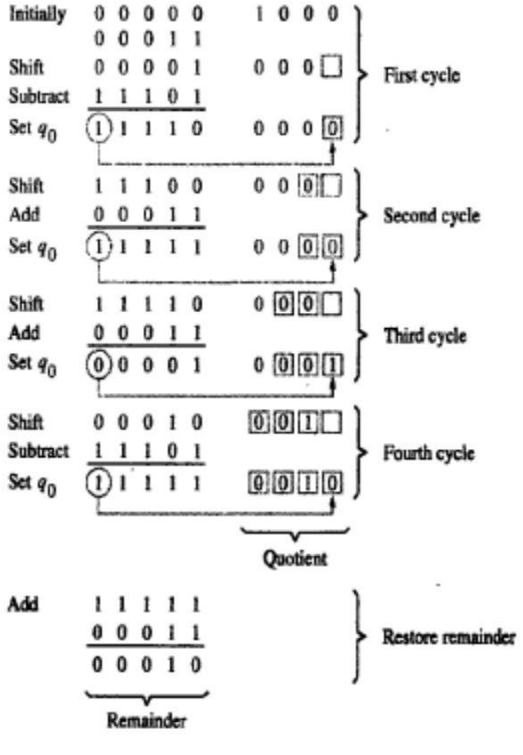


Figure 6.23 A nonrestoring-division example.

RESTORING DIVISION

• Procedure: Do the following n times

1) Shift A and Q left one binary position (Figure 6.22).
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.

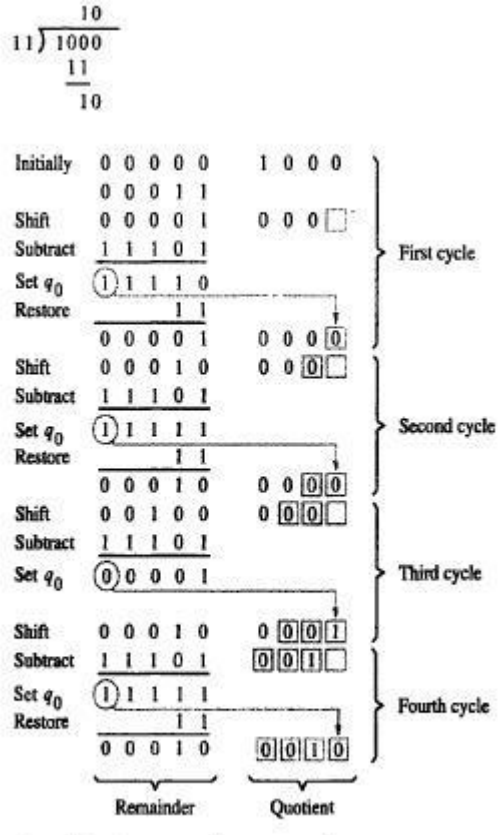


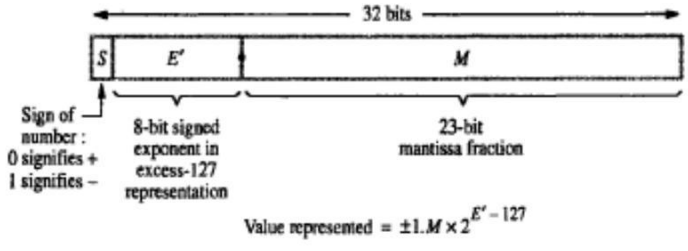
Figure 6.22 A restoring-division example.

c

FLOATING-POINT NUMBERS & OPERATIONS

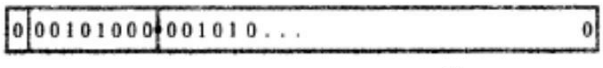
IEEE STANDARD FOR FLOATING POINT NUMBERS

- 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 6.24).
- 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.



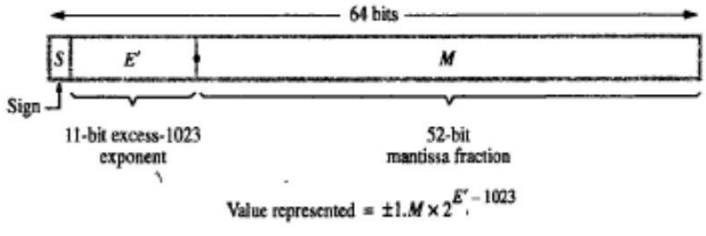
Value represented = $\pm 1.M \times 2^{E'-127}$

(a) Single precision



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

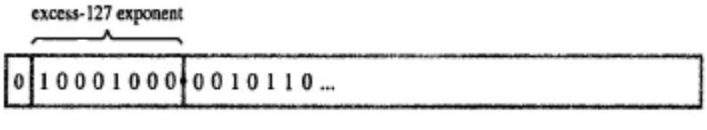
(b) Example of a single-precision number



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

(c) Double precision

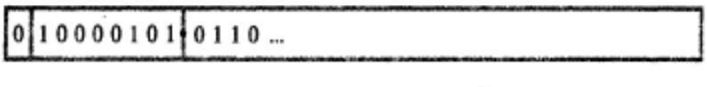
Figure 6.24 IEEE standard floating-point formats.



(There is no implicit 1 to the left of the binary point.)

Value represented = $+0.0010110\ldots \times 2^{9}$

(a) Unnormalized value



Value represented = $+1.0110\ldots \times 2^{6}$

(b) Normalized version

Figure 6.25 Floating-point normalization in IEEE single-precision format.

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 6.26).
  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.

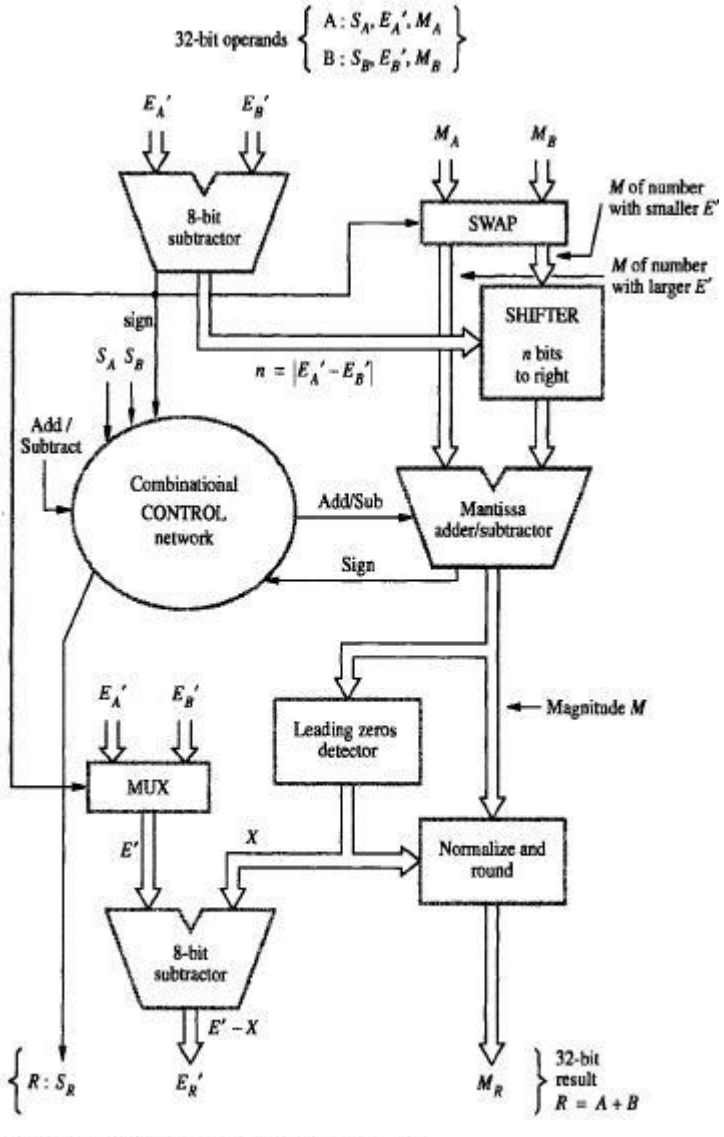


Figure 6.26 Floating-point addition-subtraction unit.

Chapter 7
Basic processing Unit

Chapter Objectives

- How a processor executes instructions
- Internal functional units and how they are connected
- Hardware for generating internal control signals
- The micro programming approach
- Micro program organization

Fundamental Concepts

- Processor fetches one instruction at a time, and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

$$IR \leftarrow [[PC]]$$

- Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).

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

- Carry out the actions specified by the instruction in the IR (execution phase).

Processor Organization

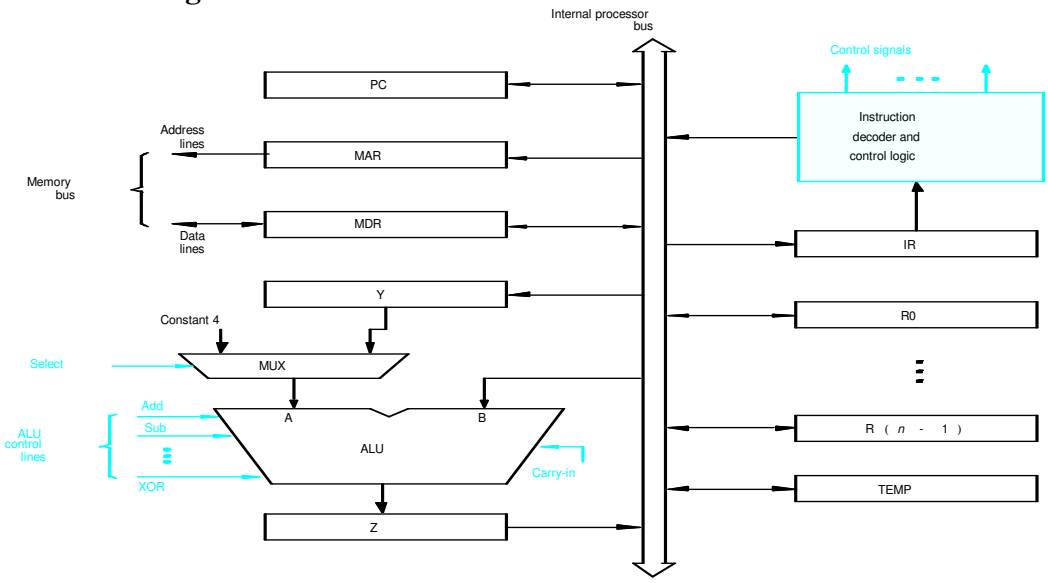


Figure 7.1. Single-bus organization of the datapath inside a processor.

- ALU and all the registers are interconnected via a single common bus.
- The data and address lines of the external memory bus connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR respectively.
- Register MDR has two inputs and two outputs.
- Data may be loaded into MDR either from the memory bus or from the internal processor bus.
- The data stored in MDR may be placed on either bus.
- The input of MAR is connected to the internal bus, and its output is connected to the external bus.
- The control lines of the memory bus are connected to the instruction decoder and control logic.
- This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for increasing with the memory bus.
- The MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU.
- The constant 4 is used to increment the contents of the program counter.

Register Transfers

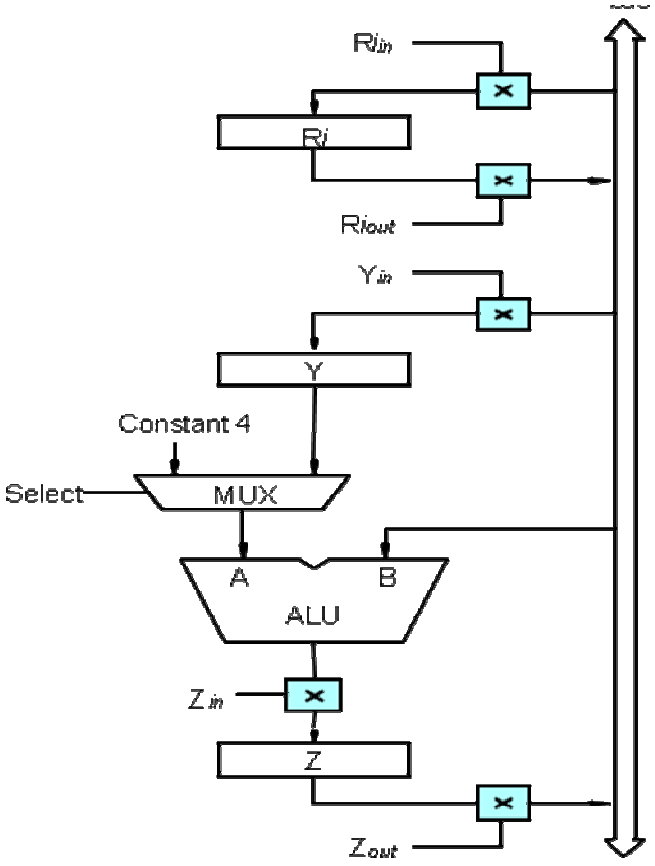


Figure 7.2. Input and output gating for the registers in Figure 7.1.

- 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 to place the contents of that register on the bus or to load the data on the bus into register.(in figure)
- The input and output of register Ri_{in} and Ri_{out} is set to 1, the data on the bus are loaded into R_i.
- Similarly, when Ri_{out} is set to 1, the contents of register Ri are placed on the bus.
- While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Example

- Suppose we wish to transfer the contents of register R1 to register R4. This can be accomplished as follows.
- Enable the output of registers R1 by setting R1out to 1. This places the contents of R1 on the processor bus.
- Enable the input of register R4 by setting R4out to 1. This loads data from the processor bus into register R4.
- All operations and data transfers with in the processor take place with in time periods defined by the processor clock.
- The control signals that govern a particular transfer are asserted at the start of the clock cycle.

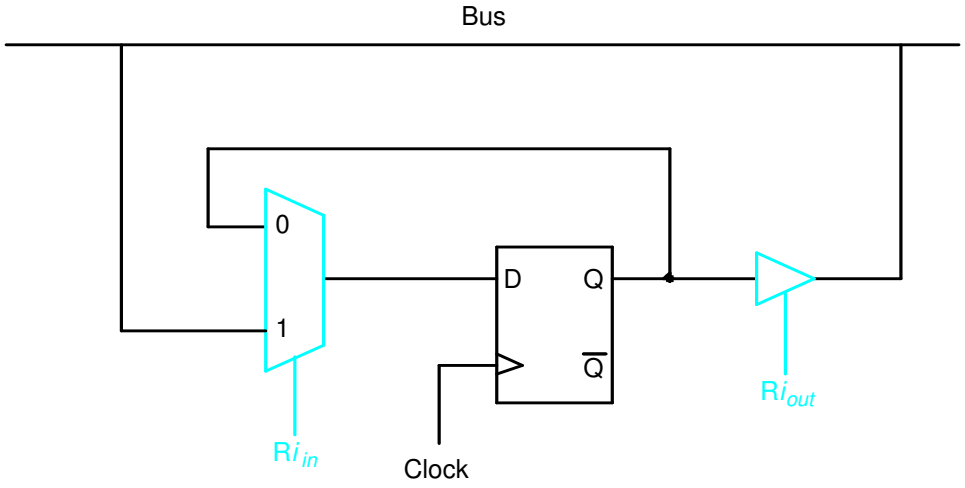


Figure 7.3. Input and output g ating for one register bit.

Performing an Arithmetic or Logic Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?
    - R1out, Yin

- R2out, SelectY, Add, Zin
- Zout, R3in

- All other signals are inactive.
- In step 1, the output of register R1 and the input of register Y are enabled, causing the contents of R1 to be transferred over the bus to Y.
- Step 2, the multiplexer's select signal is set to Select Y, causing the multiplexer to gate the contents of register Y to input A of the ALU.
- At the same time, the contents of register R2 are gated onto the bus and, hence, to input B.
- The function performed by the ALU depends on the signals applied to its control lines.
- In this case, the ADD line is set to 1, causing the output of the ALU to be the sum of the two numbers at inputs A and B.
- This sum is loaded into register Z because its input control signal is activated.
- In step 3, the contents of register Z are transferred to the destination register R3. This last transfer cannot be carried out during step 2, because only one register output can be connected to the bus during any clock cycle.

Fetching a Word from Memory

- The processor has to specify the address of the memory location where this information is stored and request a Read operation.
- This applies whether the information to be fetched represents an instruction in a program or an operand specified by an instruction.
- The processor transfers the required address to the MAR, whose output is connected to the address lines of the memory bus.

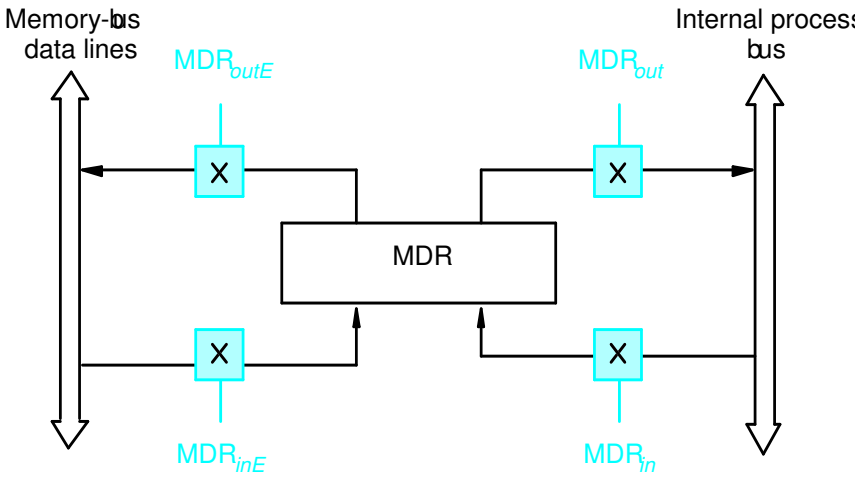


Figure 7.4. Connection and control signals for register MDR.

- At the same time , the processor uses the control lines of the memory bus to indicate that a Read operation is needed.
- When the requested data are received from the memory they are stored in register MDR, from where they can be transferred to other registers in the processor.

- The response time of each memory access varies (cache miss, memory-mapped I/O,…).
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed (Memory-Function-Completed, MFC).
- **Move (R1), R2**

  MAR ← [R1]
  Start a Read operation on the memory bus
  Wait for the MFC response from the memory
  Load MDR from the memory bus
  R2 ← [MDR]
- The output of MAR is enabled all the time.
- Thus the contents of MAR are always available on the address lines of the memory bus.
- When a new address is loaded into MAR, it will appear on the memory bus at the beginning of the next clock cycle.(in fig)
- A read control signal is activated at the same time MAR is loaded.
- This means memory read operations requires three steps, which can be described by the signals being activated as follows

  $R1_{out}$,MAR_{in},Read
  MDR_{inE},WMFC
  MDR_{out},$R2_{in}$

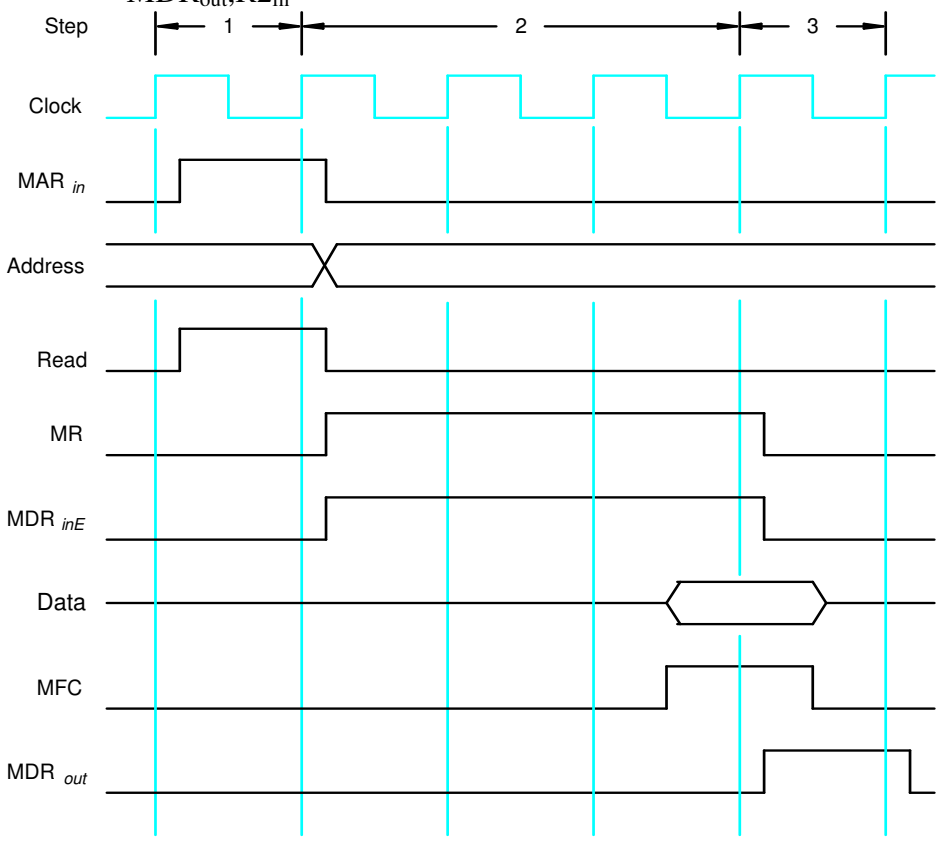


Figure 7.5. Timing of a memory Read operation.

Storing a word in Memory

- Writing a word into a memory location follows a similar procedure.
- The desired address is loaded into MAR.
- Then , the data to be written are loaded into MDR, and a write command is issued.

Example

- Executing the instruction
- Move R2,(R1) requires the following steps
  - **1 $R1_{out}$,MAR_{in}**
  - **2.$R2_{out}$,MDR_{in},Write**
  - **3.MDR_{outE},WMFC**

Execution of a Complete Instruction

- Add (R3), R1
- Fetch the instruction
- Fetch the first operand (the contents of the memory location pointed to by R3)
- Perform the addition
- 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.

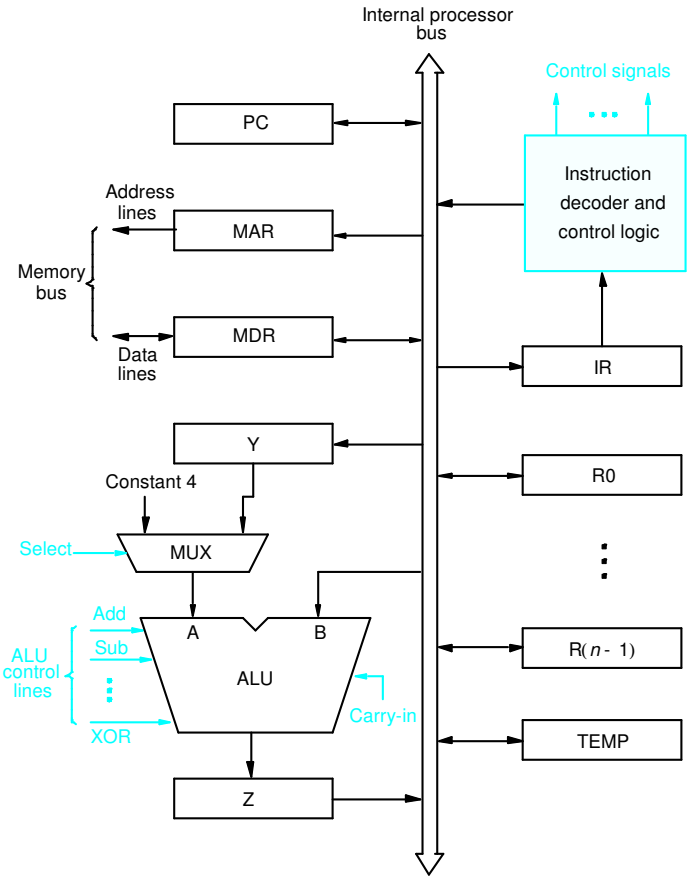


Figure 7.1. Single-bus organization of the datapath inside a proc

Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Conditional branch

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.

Multiple-Bus Organization

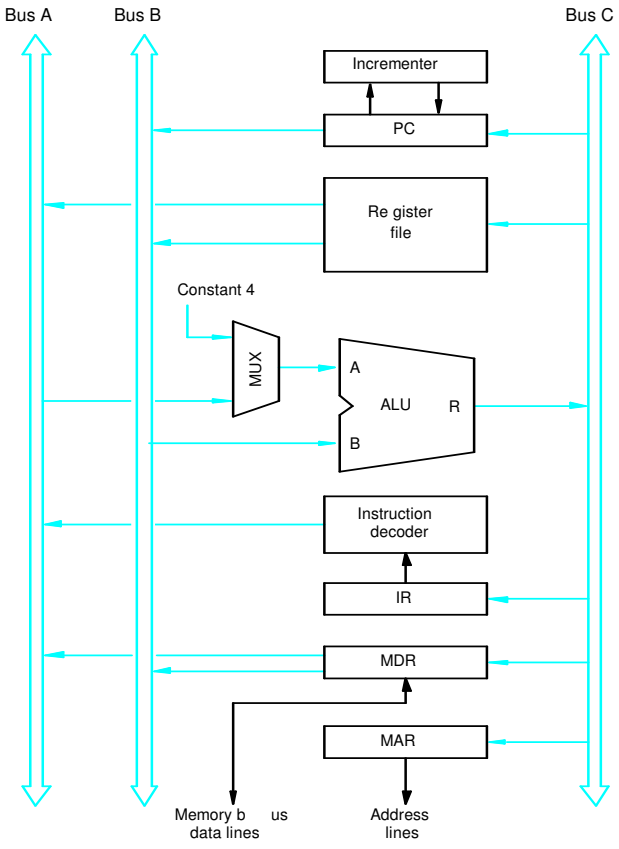


Figure 7.8. Three-b us or g anization of the datapath.

Example : Add R4, R5, R6

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, for the three-bus organization in Figure 7.8.

Hardwired Control

- To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence.
- Two categories: hardwired control and micro programmed control
- Hardwired system can operate at high speed; but with little flexibility.

Control Unit Organization

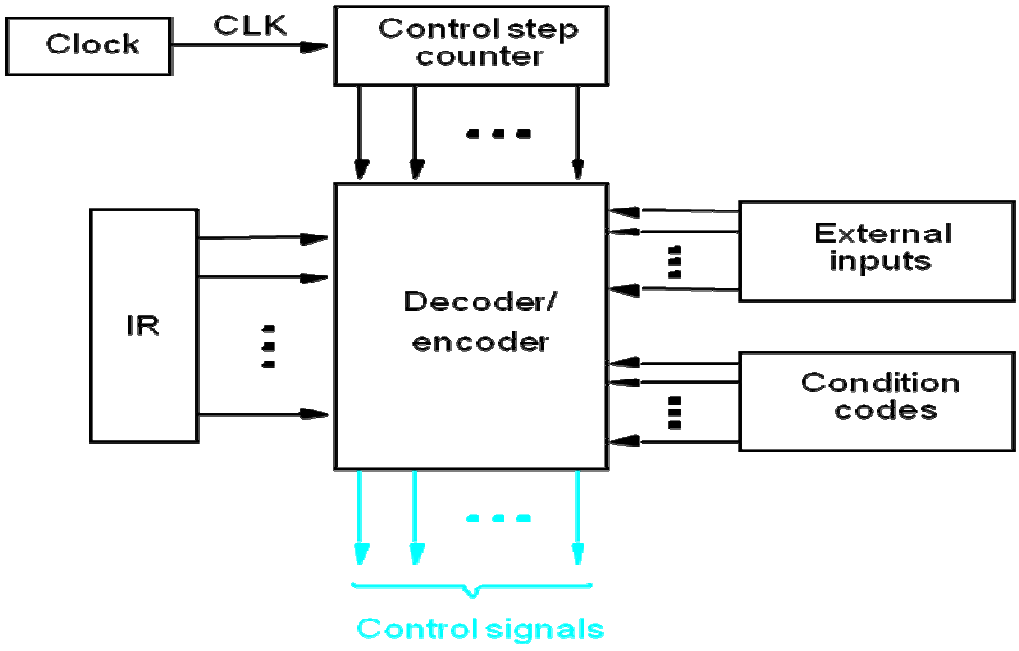


Figure 7.10. Control unit organization.

n

Detailed Control design

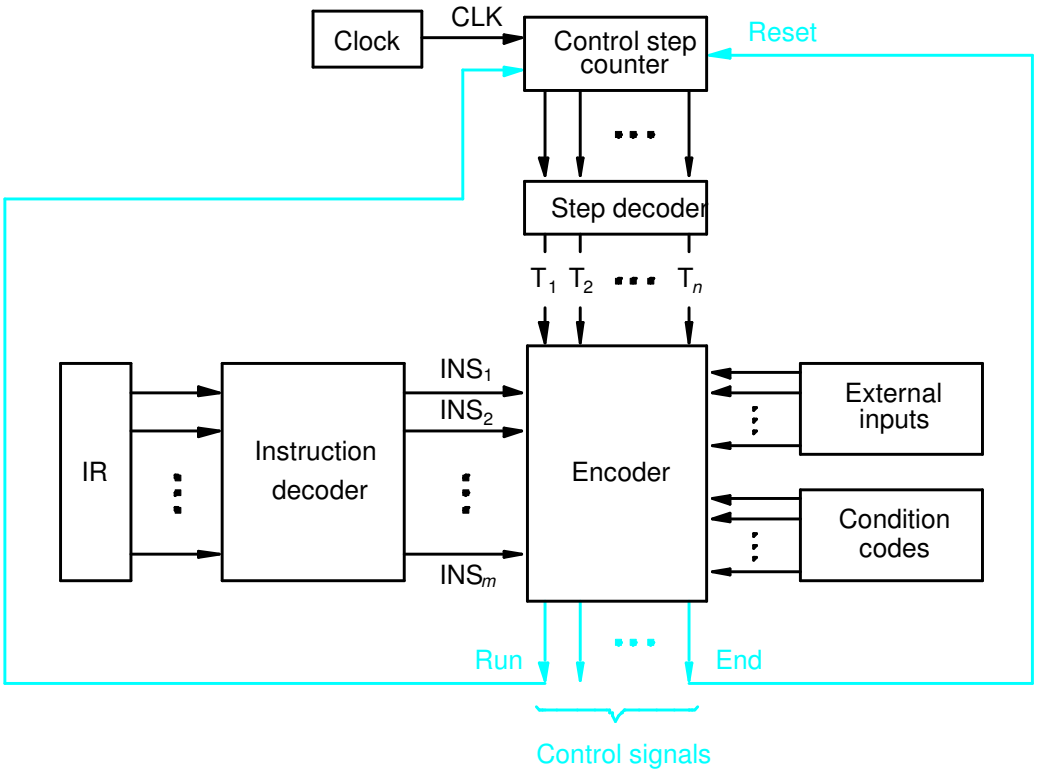


Figure 7.11. Separation of the decoding and encoding functio

Generating Z_{in}

- **$Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \ldots$**

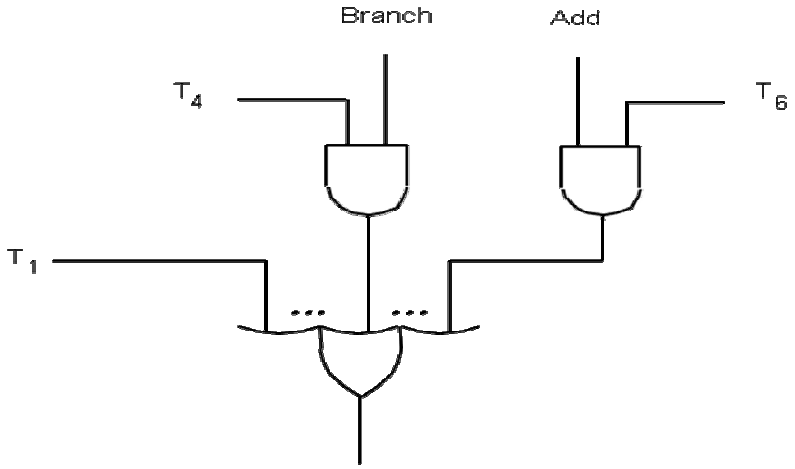


Figure 7.12. Generation of the Z_{in} control signal for the processor in Figure 7.1.

Generating End

- **End = T_7 • ADD + T_5 • BR + (T_5 • N + T_4 • N) • BRN +…**

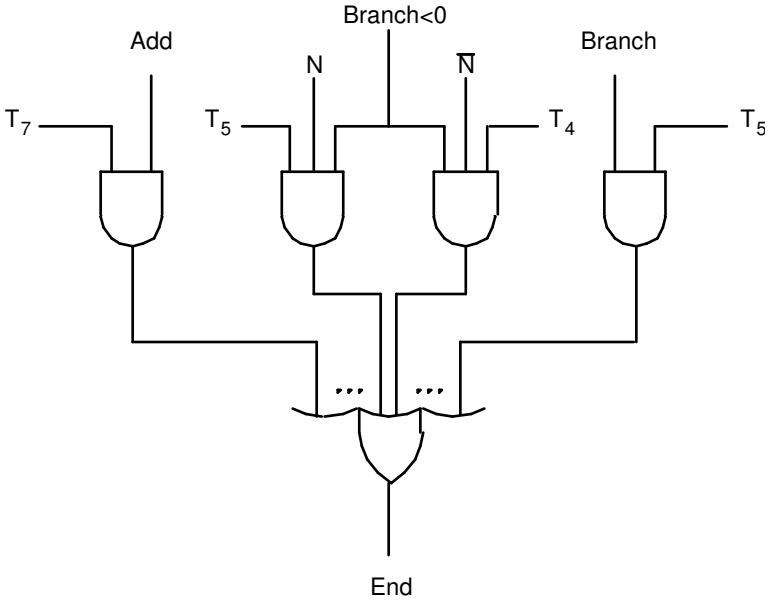


Figure 7.13. Generation of the End control signal.

A Complete Processor

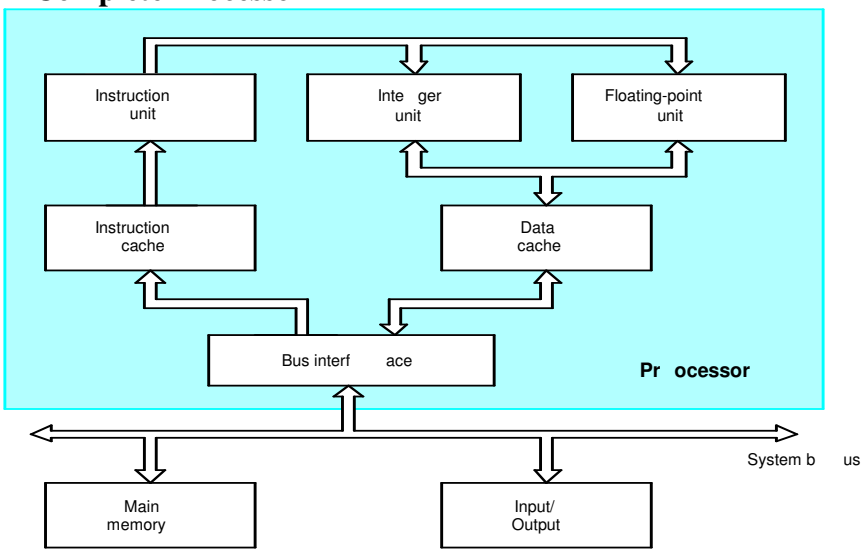


Figure 7.14. Block diagram of a complete processor .

Microprogrammed Control

- Control signals are generated by a program similar to machine language programs.
- Control Word (CW); microroutine; microinstruction

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	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6

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.

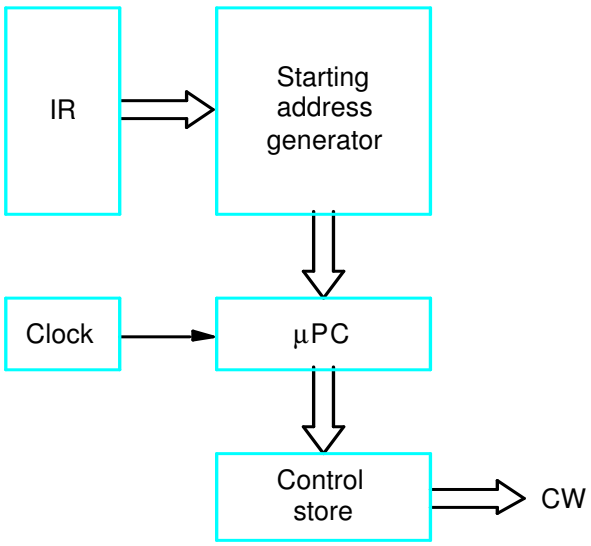


Figure 7.16. Basic organization of a microprogrammed control unit.

- The previous organization 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.
- Use conditional branch microinstruction.

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

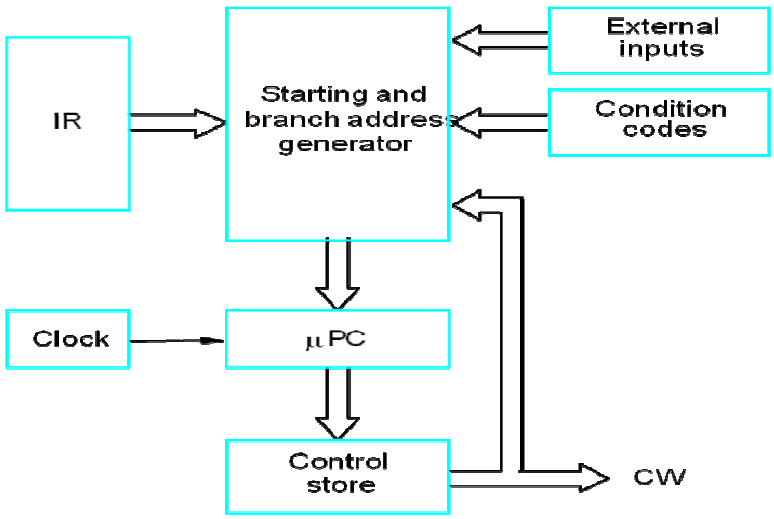


Figure 7.18. Organization of the control unit to allow conditional branching in the microprogram.

Microinstructions

- A straightforward way to structure microinstructions is to assign one bit position to each control signal.
- However, this is very inefficient.
- The length can be reduced: most signals are not needed simultaneously, and many signals are mutually exclusive.
- All mutually exclusive signals are placed in the same group in binary coding.

Microinstruction

F1	F2	F3	F4	F5

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer 0001: PC *out* 0010: MDR *out* 0011: Z *out* 0100: R0 *out* 0101: R1 *out* 0110: R2 *out* 0111: R3 *out* 1010: TEMP *out* 1011: Offset *out*	000: No transfer 001: PC *in* 010: IR *in* 011: Z *in* 100: R0 *in* 101: R1 *in* 110: R2 *in* 111: R3 *in*	000: No transfer 001: MAR *in* 010: MDR *in* 011: TEMP *in* 100: Y *in*	0000: Add 0001: Sub ⋮ 1111: XOR 16 ALU functions	00: No action 01: Read 10: Write

F6	F7	F8	...

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY 1: Select4	0: No action 1: WMFC	0: Continue 1: End

Figure 7.19. An example of a partial format for field-encoded microinstructions.

Further Improvement

- Enumerate the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can then be assigned a distinct code.
- Vertical organization
- Horizontal organization

Micro program Sequencing

- If all micro programs require only straightforward sequential execution of microinstructions except for branches, letting a μPC governs the sequencing would be efficient.
- However, two disadvantages:
    - Having a separate micro routine for each machine instruction results in a large total number of microinstructions and a large control store.
    - Longer execution time because it takes more time to carry out the required branches.
- Example: Add src, Rdst
- Four addressing modes: register, autoincrement, autodecrement, and indexed (with indirect forms).

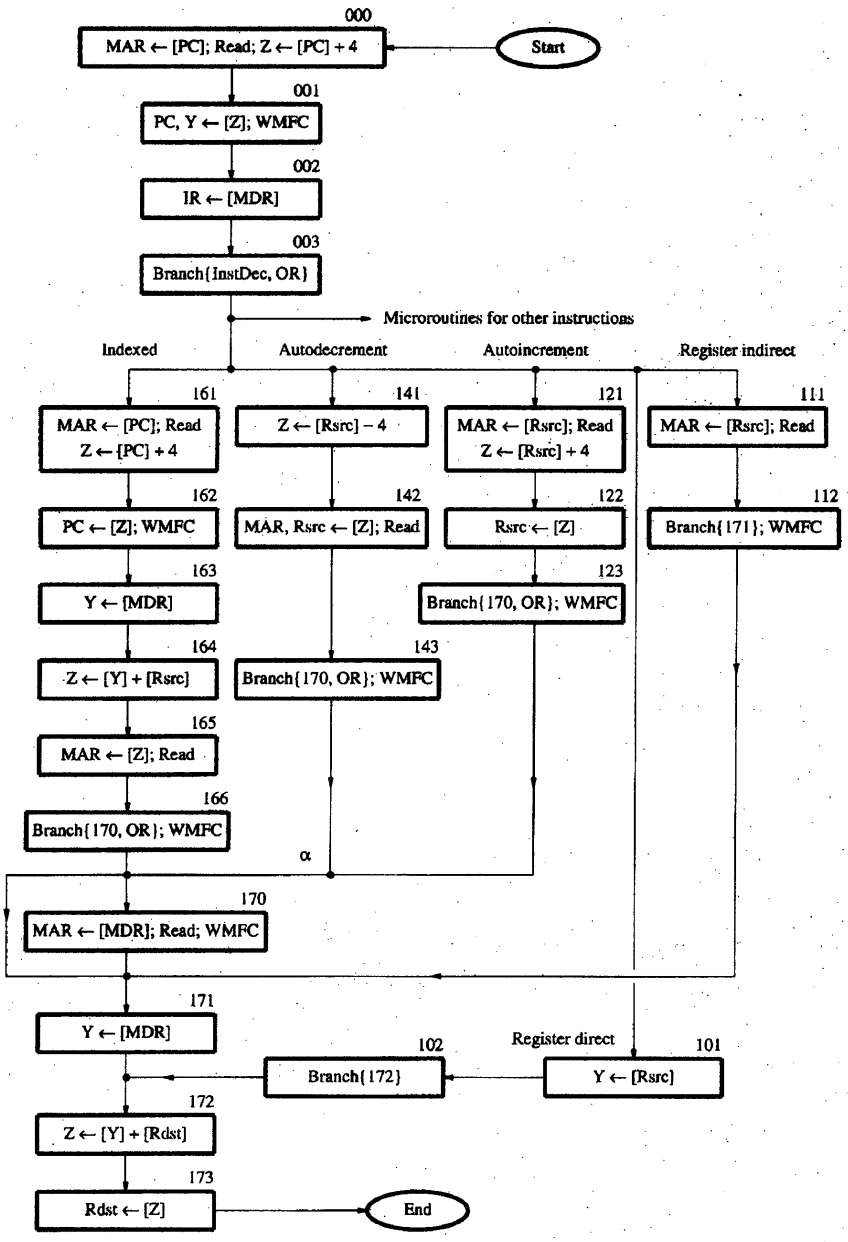


Figure 7.20. Flowchart of a microprogram for the Add src,Rdst instruction.

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 {$\mu PC \leftarrow 101$ (from Instruction decoder); $\mu PC_{3,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {$\mu PC \leftarrow 170$; $\mu PC_8 \leftarrow [\overline{IR_8}]$}, 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 Field

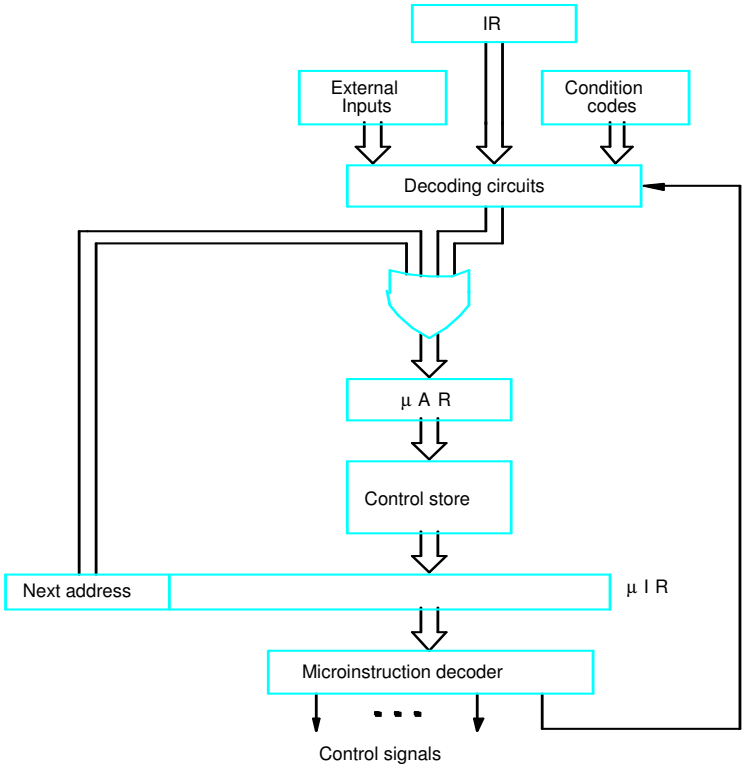


Figure 7.22. Microinstruction-sequencing organization.

- The microprogram we discussed requires several branch microinstructions, which perform no useful operation in the datapath.
- A powerful alternative approach is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.
- Pros: separate branch microinstructions are virtually eliminated; few limitations in assigning addresses to microinstructions.
- Cons: additional bits for the address field (around 1/6)

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

Implementation of the Microroutine

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.)

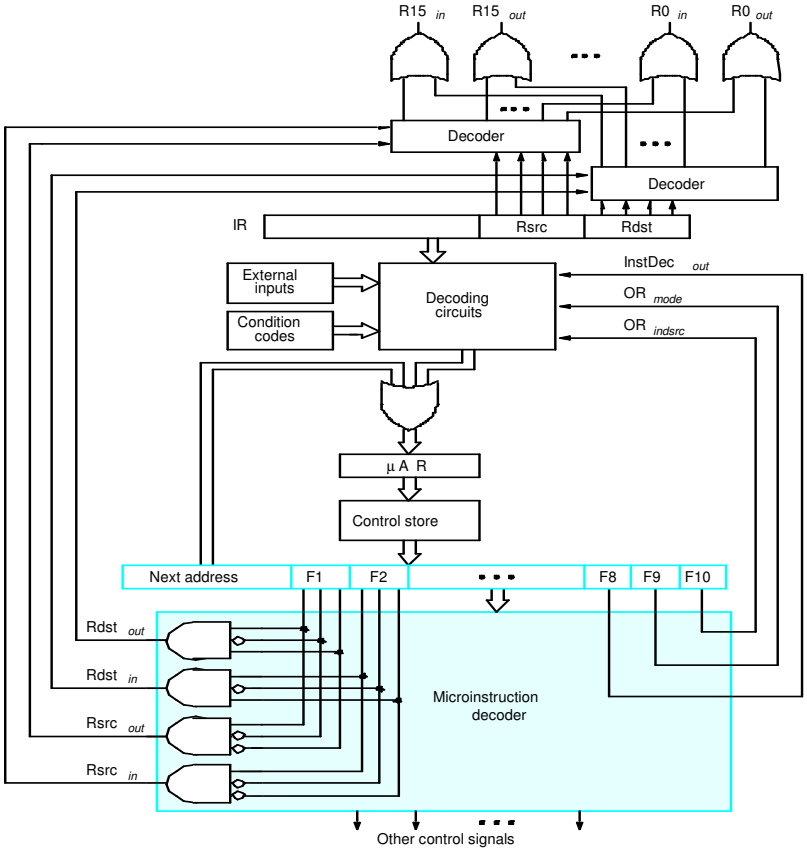


Figure 7.25. Some details of the control-signal-generating circuitry.

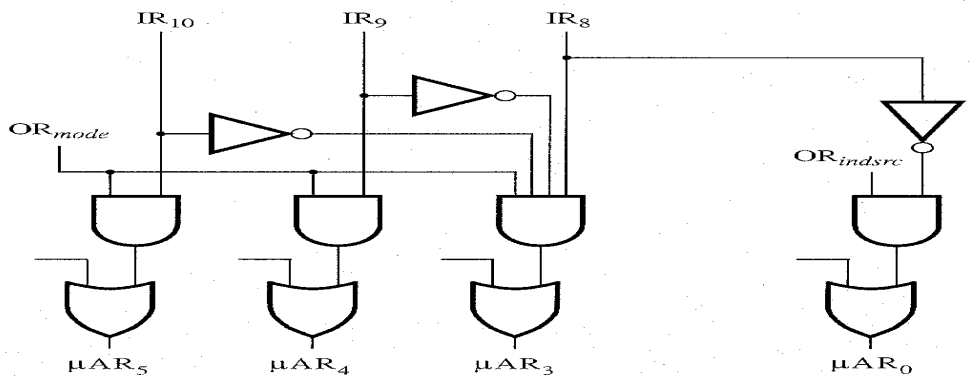


Figure 7.26. Control circuitry for bit-ORing (part of the decoding circuits in Figure 7.25).

Further Discussions

- Prefetching
- Emulation