INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad -500 043

COMPUTER SCIENCE AND ENGINEERING

COURSE LECTURE NOTES

Course Name	**COMPUTER ORGANIZATION AND ARCHITECTURE**
Course Code	**ACS007**
Programme	B.Tech
Semester	IV
Course Coordinator	Mr. E Sunil Reddy, Assistant Professor
Course Faculty	Dr.P.L.Srinivasa Murthy. Professor Mr. N Rajasekhar, Assistant Professor Ms. B.DhanaLaxmi, Assistant Professor
Lecture Numbers	1-45
Topic Covered	All

COURSE OBJECTIVES:

The course should enable the students to:	
I	Understand the organization and architecture of computer systems and electronic computers.
II	Study the assembly language program execution, instruction format and instruction cycle.
III	Design a simple computer using hardwired and micro programmed control methods.
IV	Study the basic components of computer systems besides the computer arithmetic
V	Understand input-output organization, memory organization and management, and pipelining.

COURSE LEARNING OUTCOMES (CLOs):

Students, who complete the course, will have demonstrated the ability to do the following:

CLOCode	CLO's	At the end of the course, the student will have the ability to:	PO's Mapped	Strength of Mapping
ACS007.01	CLO 1	Describe the various components like input/output MODULEs, memory MODULE, control MODULE, arithmetic logic MODULE connected in the basicorganization of a computer.	PO 1	3
ACS007.02	CLO 2	Understand the interfacing concept with memory subsystem organizationandinput/output subsystemorganization.	PO 1	3
ACS007.03	CLO 3	Understand instruction types, addressing modes and their formats in the assemblylanguage programs.	PO 2	2

ACS007.04	CLO 4	Describe the instruction set architecturedesign for relatively simple microprocessor or Central processing MODULE.	PO 3	1
ACS007.05	CLO 5	Classify the functionalities of various micro operations such as arithmetic, logic and shiftmicro operations.	PO 3	2
ACS007.06	CLO 6	Understand the register transfer languagesand micro operations involved in bus and memory transfers.	PO 2	2
ACS007.07	CLO 7	Describe the design of control MODULE withaddress sequencing and microprogramming Concepts.	PO3	1
ACS007.08	CLO 8	Understand the connections among thecircuits and the functionalities in the hardwired control MODULE.	PO2	1
ACS007.09	CLO 9	Describe the various phases involved in the instruction cycle viz. fetching, decoding, reading effective address and execution ofinstruction.	PO 2	3
ACS007.10	CLO 10	Describe various data representations and explain how arithmetic and logical operationsare performed by computers.	PO 2	1
ACS007.11	CLO 11	Classify the various instructions formats tosolve the arithmetic expressions in different addressing modes.	PO1	1
ACS007.12	CLO 12	Understand the functionality of various instruction formats for writing assemblylanguage programs.	PO 2	1
ACS007.13	CLO 13	Describe the implementation of fixed pointand floating point addition, subtraction operations.	PO 1	3
ACS007.14	CLO 14	Understand the concept of memory hierarchyand different typed of memory chips.	PO 2	2
ACS007.15	CLO 15	Describe various modes of data transferbetween CPU and I/O devices	PO2, PO3	1
ACS007.16	CLO 16	Understand the virtual memory concept with page replacement concept in memory organization	PO 2	2
ACS007.17	CLO 17	Describe the hardware organization of associate memory and understand the readand write operations	PO1, PO2	1
ACS007.18	CLO 18	Describe the parallel processing concept withmultiple functional MODULEs.	PO 2	2
ACS007.19	CLO 19	Understand the multiprocessor concept with system bus structure and the concept of inter processor communication andsynchronization.	PO 1	2
ACS007.20	CLO 20	Understand the different priority interrupts in the input-output organization in the computerarchitecture.	PO 1	2
ACS007.21	CLO 21	Possess the knowledge and skills for employability and to succeed in nationalandinternational level competitiveexaminations.	PO 2	1
ACS007.22	CLO 22	Possess the knowledge and skills to design advanced computer architecture for currentindustry requirements.	PO 1	1

SYLLABUS

MODULE-I	INTRODUCTION TO COMPUTER ORGANIZATION	Classes: 08
Basic computer organization, CPU organization, memory subsystem organization and interfacing, input or output subsystem organization and interfacing, a simple computer levels of programming languages, assembly language instructions, instruction set architecture design, a simple instruction set architecture.		
MODULE-II	**ORGANIZATION OF A COMPUTER**	**Classes: 10**
Register transfer: Register transfer language, register transfer, bus and memory transfers, arithmetic micro operations, logic micro operations, shift micro operations; Control MODULE: Control memory, address sequencing, micro program example, and design of control MODULE.		
MODULE-III	**CPU AND COMPUTER ARITHMETIC**	**Classes: 08**
CPU design: Instruction cycle, data representation, memory reference instructions, input-output, and interrupt, addressing modes, data transfer and manipulation, program control. Computer arithmetic: Addition and subtraction, floating point arithmetic operations, decimal arithmetic MODULE.		
MODULE-IV	**INPUT-OUTPUT ORGANIZATION AND MEMORY ORGANIZATION**	**Classes: 10**
Memory organization: Memory hierarchy, main memory, auxiliary memory, associative memory, cache memory, virtual memory; Input or output organization: Input or output Interface, asynchronous data transfer, modes of transfer, priority interrupt, direct memory access.		
MODULE-V	**MULTIPROCESSORS**	**Classes: 09**
Pipeline: Parallel processing, pipelining-arithmetic pipeline, instruction pipeline; Multiprocessors: Characteristics of multiprocessors, inter connection structures, inter processor arbitration, inter processor communication and Synchronization.		

Text Books:

1. M. Morris Mano, "Computer Systems Architecture", Pearson, 3rdEdition,2007.
2. John D. Carpinelli, "Computer Systems Organization and Architecture", Pearson, 1st Edition,2001.
3. Patterson,Hennessy,"Computer OrganizationandDesign:TheHardware/SoftwareInterface",Morgan Kaufmann, 5th Edition,2013.

Reference Books:

1. John. P. Hayes, "Computer System Architecture", McGraw-Hill, 3rd Edition,1998.
2. Carl Hamacher, Zvonko G Vranesic, Safwat G Zaky, "Computer Organization", McGraw-Hill, 5th Edition,2002.
3. William Stallings, "Computer Organization and Architecture", Pearson Edition, 8th Edition,2010.

Web References:

1. https://www.tutorialspoint.com/computer_logical_organization/
2. https://www.courseera.org/learn/comparch

MODULE-1

INTRODUCTION TO COMPUTER ORGANIZATION

Basic Computer Organization – CPU Organization – Memory Subsystem Organization and Interfacing – I/O Subsystem Organization and Interfacing – A Simple Computer- Levels of Programming Languages, Assembly Language Instructions, Instruction Set Architecture Design, A simple Instruction Set Architecture.

1.1 BASIC COMPUTERORGANIZATION:

Most of the computer systems found in automobiles and consumer appliances to personal computers and main frames have some basic organization. The basic computer organization has three main components:

- CPU
- Memorysubsystem
- I/Osubsystem.

The generic organization of these components is shown in the figure below.

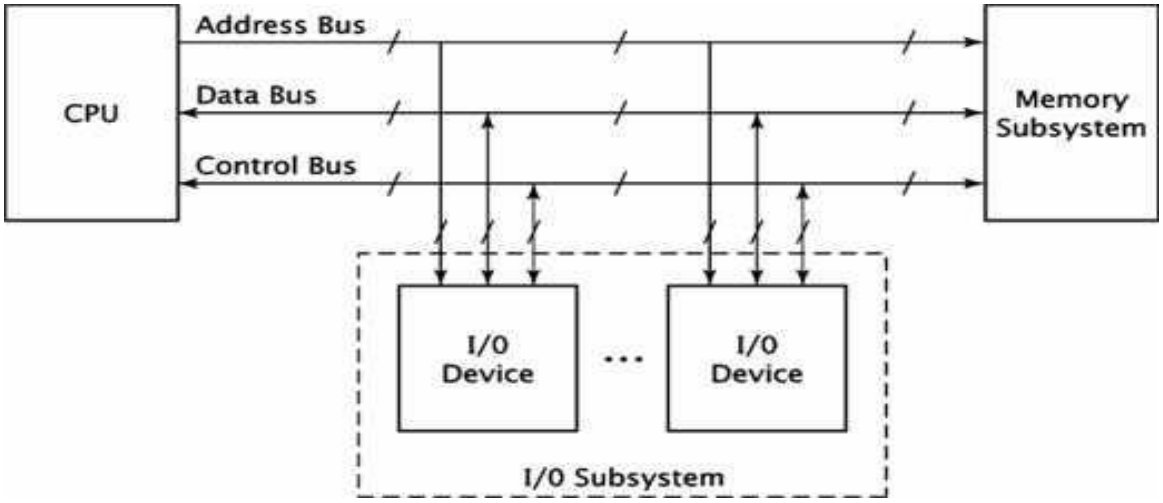


Fig 1.1 Generic computer Organization

1.1.1 System bus:

Physically the bus a set of wires. The components of a computer are connected to the buses. To send information from one component to another the source component outputs data on to the bus. The destination component then inputs the data frombus.

The system has threebuses

- Address bus
- Data bus
- Control bus

- The uppermost bus in this figure is the **address bus**. When the CPU reads data or instructions from or writes data to memory, it must specify the address of the memory location it wishes toaccess.
- Data is transferred via the **data bus**. When CPU fetches data from memory it first outputs the memory address on to its address bus. Then memory outputs the data onto the data bus. Memory then reads and stores the data

at the properlocations.

- **Control bus** carries the control signal. Control signal is the collection of individual control signals. These signals indicate whether data is to be read into or written out of theCPU.

Instruction cycles:

- The instruction cycle is the procedure a microprocessor goes through to process an instruction.
- First the processor **fetches** or reads the instruction from memory. Then it decodes the instruction determining which instruction it has fetched. Finally, it performs the operations necessary to execute theinstruction.
- After fetching it **decodes** the instruction and controls the execution procedure. It performs some Operation internally, and supplies the address, data & control signals needed by memory & I/O devices to **execute** theinstruction.
- The READ signal is a signal on the control bus which the microprocessor asserts when it is ready to read data from memory or I/Odevice.
- When READ signal is asserted the memory subsystem places the instruction code be fetched on to the computer system's data bus. The microprocessor then inputs the data from the bus and stores its internalregister.
- READ signal causes the memory to read the data, the WRITE operation causes the memory to store thedata.

Below figure shows the memory read and memory writeoperations.

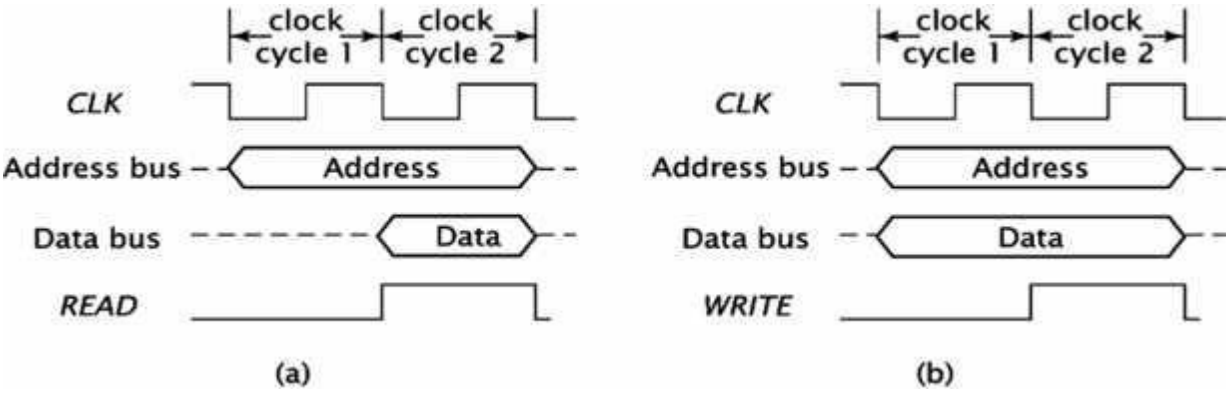


Fig 1.2: Timing diagram for memory read and memory writeoperations

• In the above figure the top symbol is CLK. This is the computer system clock. The processor uses the system clock to synchronize its operations.

• In fig (a) the microprocessor places the address on to the bus at the beginning of a clock cycle, a 0/1 sequence of clock. One clock cycle later, to allow for memory to decode the address and access its data, the microprocessor asserts the READ control signal. This causes the memory to place its data onto the system data bus. During this clock cycle, the microprocessor reads the data off the system bus and stores it in one of the registers. At the end of the clock cycle it removes the address from the address bus and deasserts the READ signal. Memory then removes the data from the data from the data bus completing the memory read operation.

- In fig(b) the processor places the address and data onto the system bus during the first clock pulse. The microprocessor then asserts the WRITE control signal at the end of the second clock cycle. At the end of the second clock cycle the processor completes the memory write operation by removing the address and data from the system bus and DE asserting the WRITE signal.

- I/O read and write operations are similar to the memory read and write operations. Basically the processor may use memory mapped I/O and isolated I/O.

- In memory mapped I/O it fallows the same sequence of operations to input data as to read from or write data into memory.

In isolated I/O fallow same process but have a second control signal to distinguish between I/O and memory accesses. For example in 8085 microprocessor has a control signal called IO/ . The processor set IO/ to 1 for I/O read and write operations and 0 for memory read and writeoperations.

CPUORGANIZATION:

Central processing MODULE(**CPU**) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by theinstructions.

In the computer all the all the major components are connected with the help of the **system bus**. **Data bus** is used to shuffle data between the various components in a computersystem.

To differentiate memory locations and I/O devices the system designer assigns a unique memory address to each memory element and I/O device. When the software wants to access some particular memory location or I/O device it places the corresponding address on the **address bus**. Circuitry associated with the memory or I/O device recognizes this address and instructs the memory or I/O device to read the data from or place data on the data bus. Only the device whose address matches the value on the address busresponds.

The **control bus** is an eclectic collection of signals that control how the processor communicates with the rest of the system. The **read** and **write** control lines control the direction of data on the data bus. When both contain logic one the CPU and memory-I/O are not communicating with one another. If the read line is low (logic zero) the CPU is reading data from memory (that is the system is transferring data from memory to the CPU). If the write line is low the system transfers data from the CPU tomemory.

The CPU controls the computer. It **fetches** instructions from memory, supply the address and control signals needed by the memory to access itsdata.

Internally, CPU has three sections as shown in the figbelow

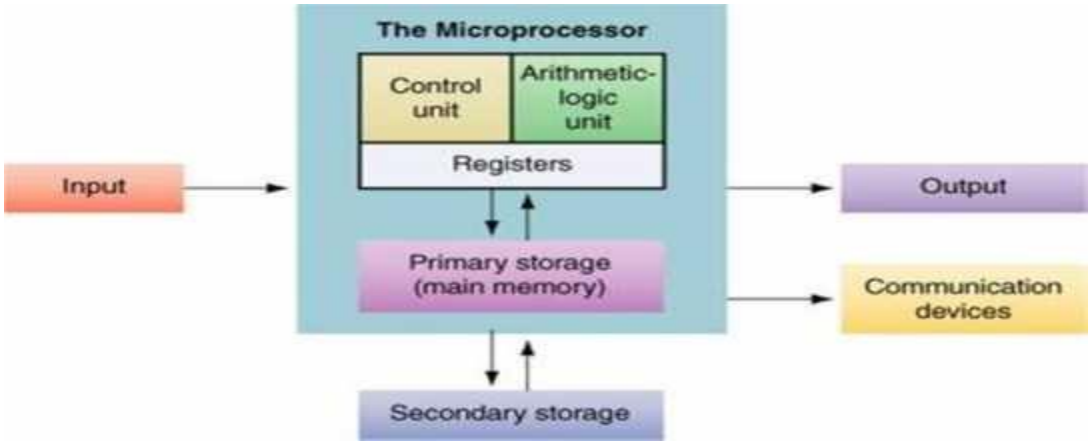


Fig : CPU Organization

- The register section, as its name implies, includes a set of registers and a bus or other communicationmechanism.
- Theregisterinaprocessor'sinstructionsetarchitecturearefoundinthesectionoftheCPU.
- The system address and data buses interact with this section of CPU. The register section also contains other registers that are not directly accessible by theprogrammer.
- The fetch portion of the instruction cycle, the processor first outputs the address of the instruction onto the address bus. The processor has a register called the "**programcounter**".
- The CPU keeps the address of the next instruction to be fetched in this register. Before the CPU outputs the address on to the system bus, it retrieves the address from the program counterregister.
- At the end of the instruction fetch, the CPU reads the instruction code from the system data bus.
- It stores this value in an internal register, usually called the "**instructionregister**".
- The arithmetic / logic MODULE (or) ALU performs most arithmetic and logic operations such as adding and ANDing values. It receives its operands form the register section of the CPU and stores its result back in the registersection.
- Just as CPU controls the computer, the control MODULE controls the CPU. The control MODULE receives some data values from the register MODULE, which it used to generate the control signals. This code generates the instruction codes & the values of some flagregisters.
- The control MODULE also generates the signals for the system control bus such asREAD,WRITE,IO/$\overline{M}$signals.

MEMORY SUBSYSTEM ORGANIZATION ANDINTERFACING:

Memory is the group of circuits used to store data. Memory components have some number of memory locations, each word of which stores a binary value of some fixed length. The number of locations and the size of each location vary from memory chip to memory chip, but they are fixed within individualchip.

The size of the memory chip is denoted as the number of locations times the number of bits in each location. For example, a memory chip of size 512×8 has 512 memory locations, each of which has eight bits.The address inputs of a memory chip choose one of its locations. A memory chip with 2^nlocations requires n address inputs.

- View the memory MODULE as a black box. Data transfer between the memory and the processor takes place through the use of two registers called MAR(Memory Address Register) and MDR (Memory data register).

- MAR is n-bits long and MDR is m-bits long, and data is transferred between the memory and the processor. This transfer takes placeovertheprocessorbus.

Internal organization of the memory chips:

- Memory is usually organized in the form of arrays, in which each cell is capable of storing one bitinformation.
- A possible organization is stored in the figbelow…
- Each row of cell constitutes a memory word, and all cells of a row are connected to a common column called word line, which is driven by the address decoder on thechip.
- The cells in each column are connected to sense/write circuit by two bitlines.
- The sense /write circuits are connected to the data input/output lines of thechip.
- During read operation these circuits sense or read the information stored in cells selected by a word line and transmit the information to the outputlines.
- During write operation the sense/write circuit receives the input information and store in the cell of selectedword.

Types of Memory:

There are two types ofmemorychips

1. Read Only Memory(ROM)
2. Random Access Memory (RAM)

a) ROMChips:

ROM chips are designed for applications in which data is read. These chips are programmed with data by an external programming MODULE before they are added to the computer system. Once it is done the data does not change. A ROM chip always retains its data, even when

Power to chip is turned off so ROM is called **nonvolatile** because of its property. There are several types of ROM chips which are differentiated by how often they are programmed.

- Masked ROM(or) simplyROM
- PROM(Programmed Read OnlyMemory)
- EPROM(Electrically Programmed Read OnlyMemory)
- EEPROM(Electrically ErasablePROM)
- FlashMemory

- A masked ROM or simply ROM is programmed with data as chip isfabricated.
- The mask is used to create the chip and chip is designed with the requireddata hardwired init.

Once chip is designed the data will not change. Figure below shows thepossible configuration of the ROMcell.

- Logic 0 is stored in the cell if the transistor is connected to ground at point P, other wise1 stored.
- A sense circuit at the end of the bit line generates at the high voltage indicating a 1.Data are written into the ROM when it ismanufactured.

PROM

- Some ROM designs allow the data to be loaded by the user, thus providing programmableROM (PROM).

- ProgrammabilityisachievedbyinsertingafuseatpointPintheabovefig.Beforeitis programmed, the memory contains all 0's.
- The user insert 1's at the required locations by burning out the fuse at these locations using high currentpulse.
- The fuses in PROM cannot restore once they are blown, PROM's can only beprogrammed once.

2) EPROM

- EPROM is the another ROM chip allows the stored data to be erased and new data to be loaded. Such an erasable reprogrammable ROM is usually called anEPROM.
- Programming in EPROM is done by charging of capacitors. The charged and uncharged capacitors cause each word of memory to store the correctvalue.
- The chip is erased by being placed under UV light, which causes the capacitor to leak theircharge.

3) EEPROM

- A significant disadvantage of the EPROM is the chip is physically removed from the circuit for reprogramming and that entire contents are erased by the UVlight.
- Another version of EPROM is EEPROM that can be both programmed and erased electrically, such chips called EEPROM, do not have to remove forerasure.
- The only disadvantage of EEPROM is that different voltages are need for erasing, writing, reading and storeddata.

4) FlashMemory

- A special type of EEPROM is called a flash memory is electrically erase data in blocks rather than individuallocations.
- It is well suited for the applications that writes blocks of data and can be used as a solid state hard disk. It is also used for data storage in digitalcomputers.

RAMChips:

- RAM stands for Random access memory. This often referred to as read/writememory. Unlike the ROM it initially contains nodata.
- The digital circuit in which it is used stores data at various locations in the RAMare retrieves data from theselocations.

- The data pins are bidirectional unlike inROM.
- A ROM chip loses its data once power is removed so it is a volatilememory.
- RAM chips are differentiated based on the data theymaintain.

➢ Dynamic RAM(DRAM)

➢ Static RAM(SRAM)

1. DynamicRAM:

- DRAM chips are like leaky capacitors. Initially data is stored in the DRAM chip, charging its memory cells to their maximumvalues.
- The charging slowly leaks out and would eventually go too low to represent validdata.
- Before this a refresher circuit reads the content of the DRAM and rewrites data to its original locations.
- DRAM is used to construct the RAM in personalcomputers.

- DRAM memory cell is shown in the figurebelow.

2. StaticRAM:

- Static RAM are more likely the register .Once the data is written to SRAM, its contents stay valid it does not have to berefreshed.
- Static RAM is faster than DRAM but it is also much more expensive. Cache memory in the personal computer is constructed fromSRAM.

➢ Various factors such as cost, speed, power consumption and size of the chip determine how a RAM is chosen for a givenapplication

➢ StaticRAMs:

- Chosen when speed is the primaryconcern.
- Circuit implementing the basic cell is highly complex, so cost and size areaffected.
- Used mostly in cachememories.

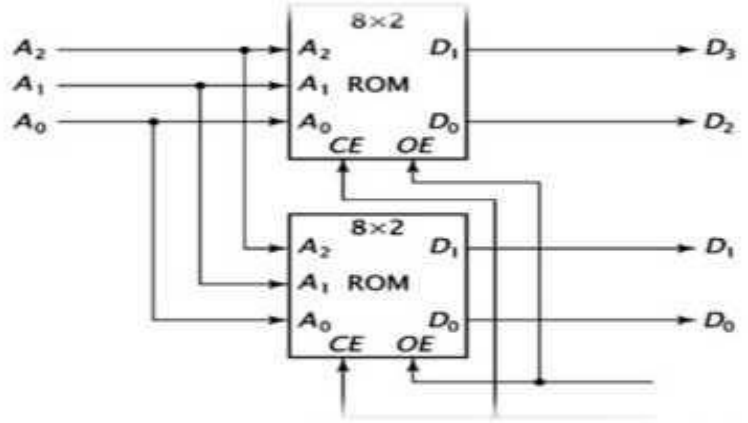
➢ DynamicRAMs:

- Predominantly used for implementing computer mainmemories.
- High densities available in thesechips.
- Economically viable for implementing largememories

Memory subsystem configuration:

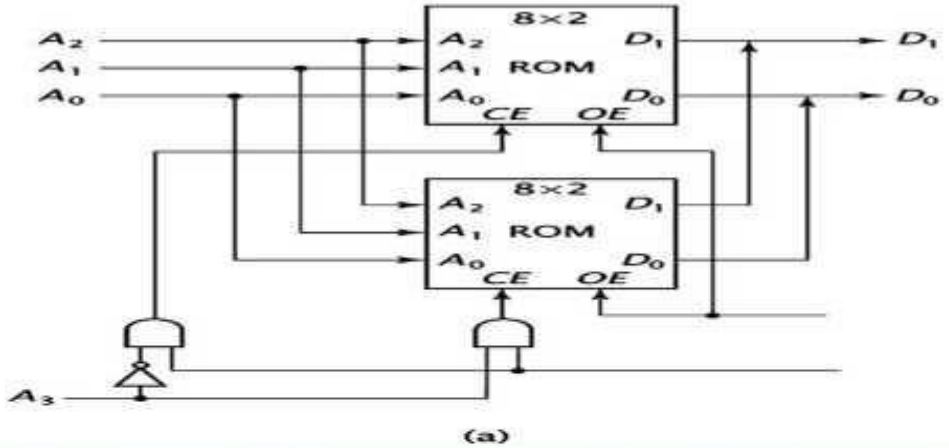
- Two or more chips are combined to create memory with more bits per location. This is done by connecting the corresponding address and control signals of chips and connecting their data pins to different bits of the databus.
- The following figure fallowed the higher order interleaving where the higher order bit A3 is 0and lower chip has A3 1. This difference is used to select one of the twochips.
- In the lower order interleaving we consider lower bit for selecting the two chips. So theupper chip is enabled when A0=0 or by address 0×××(or 0,2,4,6,8,10,12) and lower chip is enabled when A0=1 which is true for the address 1,3,5,7,9,11,13 and 15. Figure for lower order interleaving is shown in the figurebelow.

- For example two 8×2 chips can be combined to create 8×4 memory as shown the figure below.



An 8×4 memory subsystem constructed from two 8×2 ROM chips

- Both receive the same address inputs from the bus as well as the same chips enable and output enable signals. The data pins of the first two connected to bits 3 and 2 of the data bus and other chip are connected to bits 1 and 0.
- When the CPU read the data it places the address on the address bus. Both the chips read the address and perform their internal decoding.
- If CE (Chip Enable) and OE (Output Enable) signals are activated. The chips place the data on the data bus because OE and CE are same for both chips. The same 8×2 configured as a 16×2 shown in the figure below.



(a)

A 16×4 memory subsystem constructed from two 8×2 ROM chips with higher order interleaving

• The upper chip is configured as memory locations **0-7(0000 to 0111).** The lower chip is configured as **8 to 15(1000 to 1111)**.

• The configurations can be done by using two methods **higher order interleaving** and **lower order interleaving**.

Multi byte organization:

- Many data formats use more than one 8-bit byte to represent a value whether it is an integer , floating point, or characterstring.
- Most CPU assign addresses to 8-bit memory locations so these values must be stored in more than one location. It is necessary for every CPU to define the order it expects for the data in theselocations.
- There are two commonly used organizations for multi bytedata.
  - Bigendian
  - Littleendian
- In BIG-ENDIAN systems the most significant byte of a multi-byte data item always has the lowest address, while the least significant byte has the highestaddress.
- In LITTLE-ENDIAN systems, the least significant byte of a multi-byte data item always has the lowest address, while the most significant byte has the highestaddress.

I/O SUBSYSTEM ORGANIZATION ANDINTERFACING

The I/O subsystem is treated as an independent MODULE in the computer The CPU initiates I/O commandsgenerically

- Read, write, scan,etc.
- This simplifies theCPU

Below figure shows the basic connection between the CPU, memory to the I/O device .The I/O device is connected to the computer system address, data and control buses. Each I/O device includes I/O circuitry that interacts with thebuses.

INPUT DEVICE:

- The generic interface circuitry for an input device such as keyboard and also enable logic for tri state buffer is shown in the figure below.

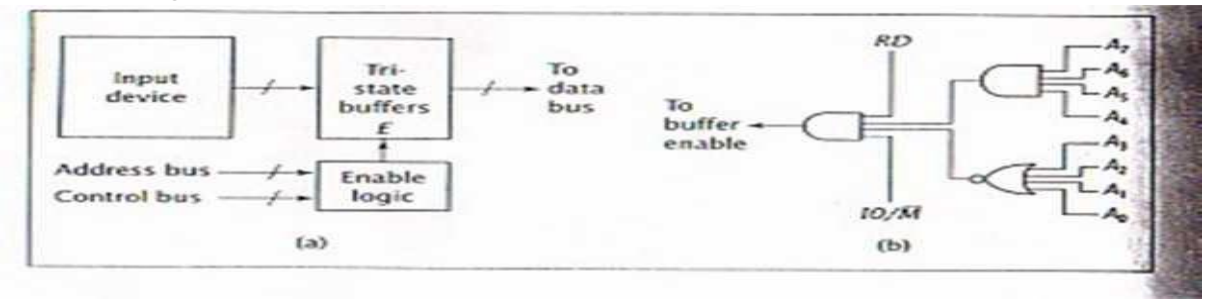


Fig : (a) with its interface and (b) the enable logic for the tri-state buffers

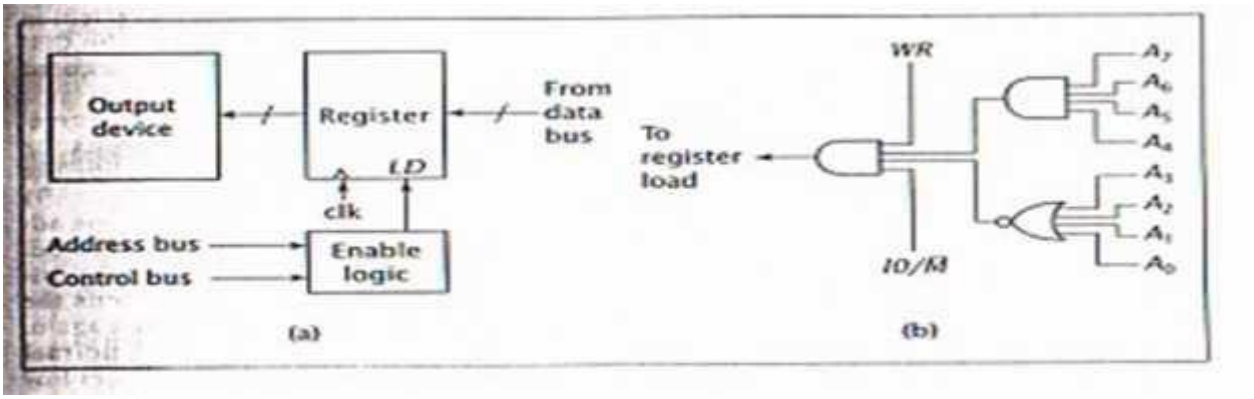
- The data from the input device goes to the tri-state buffers. When the value in the address and control buses are correct, the buffers are enabled and data passes on the data bus.
- The CPU can then read this data. If the conditions are not right the logic block does not enable the buffers and do not place on the bus.
- The enable logic contains 8-bit address and also generates two control signals RD and IO/

OUTPUT DEVICE

- The design of the interface circuitry for an output device such as a computer monitor is somewhat different than for

the input device.

- The design of the interface circuitry for an output device, such as a computer monitor, is somewhat different than that for the input device. Tri-state buffers are replaced by a register.
- The tri-state buffers are used in input device interfaces to make sure that one device writes data to the bus at any time.
- Since the output devices read from the bus, rather that writes data to it, they don't need the buffers.
- The data can be made available to all output devices but the devices only contains the correct address will read it in.
- When the load logic receives the correct address and control signals, it asserts data bus. The output device can read the data from the register at its leisure while the CPU performs the othertasks.



An output device: (a) with its interface and (b) the enable logic for the registers

Some devices are used for both input and output. Personal computer and hard disk devices are falls into this category. Such a devices requires a combined interface that is essential two interfaces. A bidirectional I/O device with its interface and enable/load logic is shown in the figurebelow.

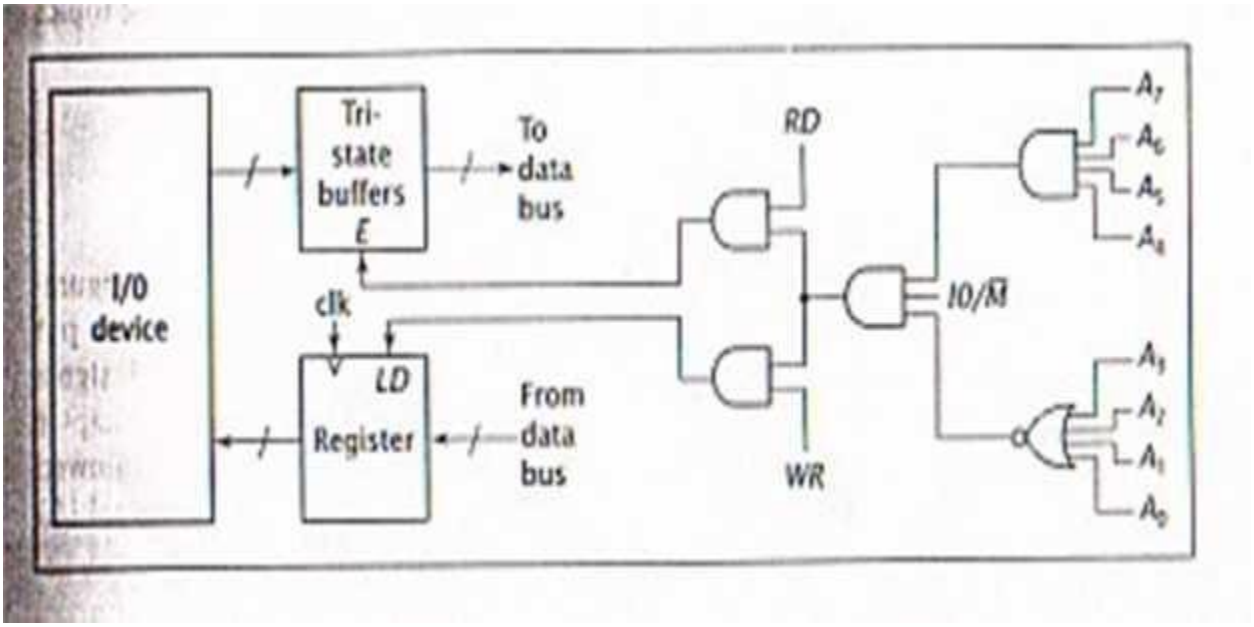
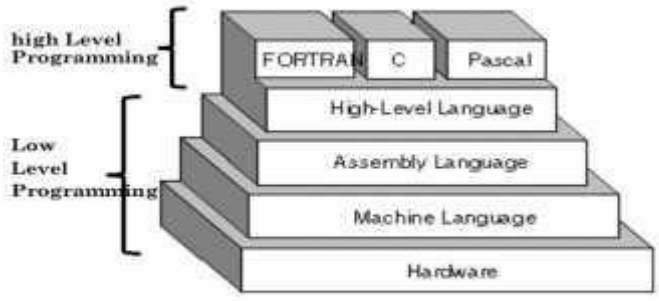


Fig: A bidirectional I/O device with its interface and enable/load logic

LEVELS OF PROGRAMMINGLANGUAGES

- Computer programming languages are divided into 3categories.
- High levellanguage
- Assembly levellanguage
- Machine levellanguage

- **High level languages** are platform independent that is these programs can run on computers with different microprocessor and operating systems without modifications. Languages such as C++, Java and FORTRAN are high levellanguages.
- **Assembly languages** are at much lower level of abstraction. Each processor has its own assemblylanguage.
- A program written in the assembly language of one processor cannot be run on a computer that has different processor so assembly languages are platformdependent.
- Unlike high level languages, instructions in assembly languages can directly manipulate the data stored in microprocessor internal components. Assembly language instructions can load the data from memory into microprocessor registers, add values, and perform many otheroperations.
- The lowest level of programming language is**machine level languages**. These languages contain the binary values that cause the microprocessor to perform certain operations. When microprocessor reads and executes an instruction it's a machine language instruction.
- Levels of programming languages is shown in the figurebelow



- Programmers don't written the programs in machine language ,rather programs written in assembly or high level are converted into machine level and then executed by the microprocessor.
- High level language programs are **compiled** and assembly level language programs are assembled.
- A program written in the high level language is input to the compiler. The compiler checkstomakesureeverystatementintheprogramisvalid.Whentheprogramhasno syntax errors the compiler finishes the compiling the program that is **source code** and generates an **object code file.**
- An object code is the machine language equivalent of sourcecode.
- A **linker** combines the object code to any other object code. This combined code storesin the **executablefile.**
- The process of converting the assembly language program to an executable form is shown the figurebelow.

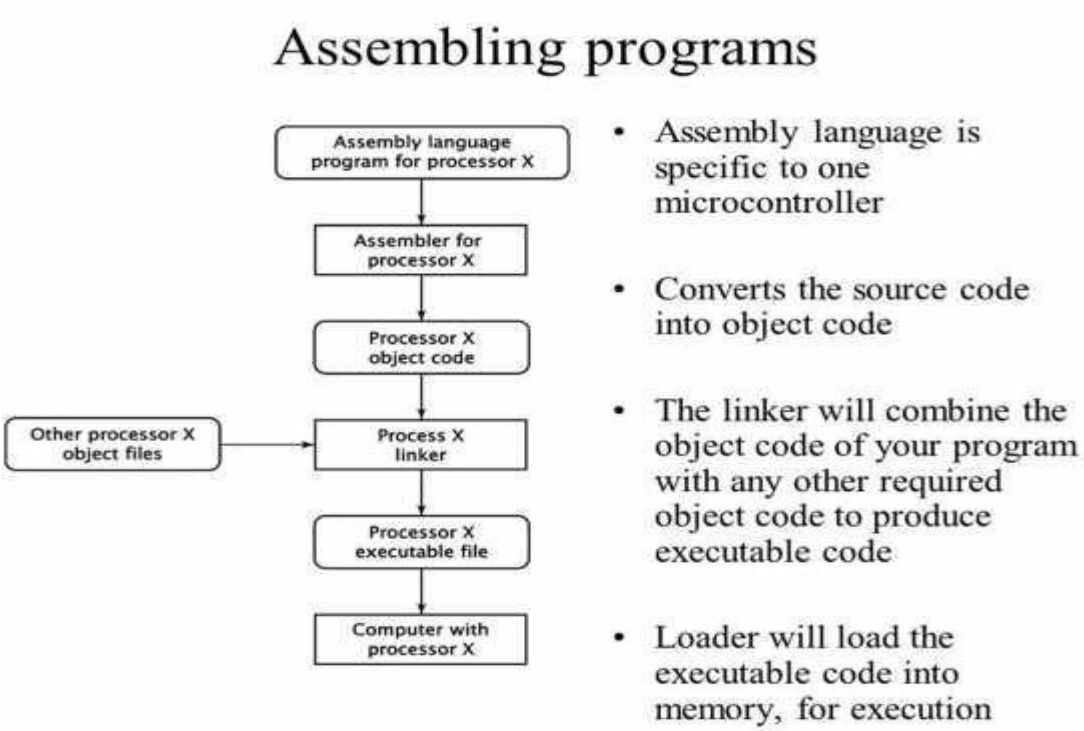


Fig: Assembly process for assembly language programs

ASSEMBLY LANGUAGEINSTRUCTIONS:

- The simplest way to organize a computer is to have one processor register and instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processorregister.
- Below Figure depicts this type of organization. Instructions are stored in one section of memory and data inanother.
- ForamemoryMODULEwith4096wordsweneed12bitstospecifyanaddresssince2^{12}= 4096. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated op code) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.
- The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a16-bit operand from the data portion of memory. It then executes the operation specified by the operationcode.

Instruction formats:

- The basic computer has three instruction code formats, as shown in Fig below. Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation codeencountered.
- A memory reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirectaddress.
- The register-reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of

the instruction. A register-reference instruction specifies an

Operation on or a test of the AC register. An operand from memory is not needed therefore the other 12 bits are used to specify the operation or test to be executed.

- Similarly, an input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining12 bits are used to specify the type of input-output operation or test performed. The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three op code bits in positions 12 through 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode I. If the 3-bit op code is equal to 111, control then inspects the bit in position15. If this bit is 0, the instruction is a register-reference type. If the bit is 1, the instruction is an I/Otype.
- The instruction for the computer is shown the table below. The symbol designation is a three letter word and represents an abbreviation intended for programmers and users. The hexa decimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to fourbits.
- A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I. When I = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When I = 1, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is I.
- Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12bits.
- The input-output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimalF.

Symb	**Hexadecimal code**		**Description**
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	ADD memory word to AC
LDA	2xxx	Axx	Load memory word to AC
STA	3xxx	Bxx	Store content of AC in memory
BUN	4xxx	Cxx	Branch Unconditionally
BSA	5xxx	Dxx	Branch and save return address
ISZ	6xxx	Exx	Increment & skip if skip
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC & E
CIL	7040		Circulate left AC & E
INC	7020		Increment AC
SPA	7010		Skip next address if AC is +ve
SNA	7008		Skip next address if AC is -ve
SZA	7004		Skip next address if AC is zero
SZE	7002		Skip next address if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

Fig: Basic computer instructions

Instructiontypes:

- Assembly languages instructions are grouped together based on the operation they performed.

➢ Data transferinstructions

➢ Data operationalinstructions

➢ Program controlinstructions

1) Data transferinstructions:

- **Load the data from memory into the microprocessor**: These instructions copy data from memory into a microprocessorregister
- **Store the data from the microprocessor into the memory**: This is similar to the load data expect data is copied in the opposite direction from a microprocessor register to memory.
- **Move data within the microprocessor**: These operations copies data from one microprocessor register toanother.
- **Input the data to the microprocessor**: The microprocessor inputs the data from the input devices ex: keyboard in to one of itsregisters.
- **Output the data from the microprocessor**: The microprocessor copies the data from one of the registers to an input device such as digital display of a microwaveoven.

2) Data operationalinstructions:

- Data operational instructions do modify their data values. They typically perform some operations using one or two data values (operands) and storeresult.
- Arithmetic instructions make up a large part of data operations instructions. Instructions that add, subtract, multiply, or divide values fall into this category. An instruction that increment or decrement also falls in to thiscategory.
- Logical instructions perform basic logical operations on data. They AND, OR, or XOR two data values or complement a singlevalue.
- Shift operations as their name implies shift the bits of a data values also comes under thiscategory.

3) Program controlinstructions:

- Program control instructions are used to control the flow of a program. Assembly language instructions may include subroutines like in high level language program may have subroutines, procedures, andfunctions.
- A jump or branch instructions are generally used to go to another part of the program orsubroutine.
- A microprocessor can be designed to accept interrupts. An interrupt causes the processor to stop what is doing and start other instructions. Interrupts may be software orhardware.
- One final type of control instructions is halt instruction. This instruction causes a processor to stop executing instructions such as end of aprogram.

Instruction set architecture(ISA)design:

- Designing of the instruction set is the most important in designing the microprocessor. A poor designed ISA even it is implemented well leads to bad microprocessor.
- A well designed instruction set architecture on the other hand can result in a powerful processor that can meet variety ofneeds.
- In designing ISA the designer must evaluate the tradeoffs in performance and such constrains issues as size and cost when designing ISAspecifications.

- If the processor is to be used for general purpose computing such as personal computer it will probably require a rich set of ISA. It will need a relatively large instruction set to

perform the wide variety of tasks it must accomplish. It may also need many registers for some of thesetasks.

- In contrast consider a microwave oven it require only simple ISA those need to control the oven. This issue of

completeness of the ISA is one of the criteria in designing the processor that means the processor must have complete set of instructions to perform the task of the givenapplication.

- Another criterion is instruction **orthogonally**. Instructions are orthogonal if they do not overlap, or perform the same function. A good instruction set minimizes the overlap betweeninstructions.
- Another area that the designer can optimize the ISA is the **register set**. Registers have a large effect on the performance of a CPU. The CPU Can store data in its internal registers instead of memory. The CPU can retrieve data from its registers much more likely than form thememory.
- Having too few registers causes a program to make more reference to the memory thus reducingperformance.

General purposes CPU have many registers since they have to process different types of programs. Intel processor has 128 general purpose registers for integer data and another 128 for floating point data. For dedicated CPU such as microwave oven having too many registers adds unnecessary hardware to the CPU without providing anybenefits.

A RELATIVELY SIMPLE INSTRUCTION SETARCHITECTURE:

- A relatively simple instruction set architecture describes the ISA of the simpleprocessor orCPU.
- A simple microprocessor can access 64K ($=2^{16}$) bytes of memory with each byte having8

bits or 64K×8 of memory. This does not mean that every computer constructed using this relatively simple CPU must have full 64K of memory. A system based on this processor can have less than memory if doesn't need the maximum 64K of memory.

- This processor inputting the data from and outputting the data to external devices suchas microwave ovens keypad and display are treated as memory accesses. There are two types of input/output interactions that can design a CPU toperform.

- IsolatedI/O
- Memory mappedI/O

- An **isolated I/O** input and output devices are treated as being separated from memory. Different instructions are used for memory andI/O.
- **Memory mapped I/O** treats input and output devices as memory locations the CPU access these I/O devices using the same instructions that it uses to access memory. For relatively simple CPU memory mapped I/O isused.
- There are three registers in ISA of thisprocessor.

- Accumulator(AC)
- RegisterR
- Zero flag(Z)

- The first accumulator is an 8bit register. Accumulator of this CPU receives the result of any arithmetic
- and logical operations. It also provides one of the operands for ALU instructions that
- use two operands. Data is loaded from memory it is loaded into the accumulator and also data stored to memory also comes fromAC.
- Register R is an 8-bit general purpose register. It supplies the second operand of all two operand arithmetic and logical instructions. It also stores the finaldata.
- Finally, there is a 1-bit zero flag Z. Whenever an arithmetic and logical instruction is executed. If the result of the instruction is 0 then Z is set to 1 that a zero result was generated. Otherwise it is set to0.

- The final component is the instruction set architecture for this relatively simple CPU is shown in the tablebelow.

Instruction	Instruction Code	Operation
NOP	0000 0000	No operation
LDAC	0000 0001 Γ	$AC = M[\Gamma]$
STAC	0000 0010 Γ	$M[\Gamma] = AC$
MVAC	0000 0011	$R = AC$
MOVR	0000 0100	$AC = R$
JUMP	0000 0101 Γ	GOTO Γ
JMPZ	0000 0110 Γ	IF (Z=1) THEN GOTO Γ
JPNZ	0000 0111 Γ	IF (Z=0) THEN GOTO Γ
ADD	0000 1000	$AC = AC + R$, If $(AC + R = 0)$ Then $Z = 1$ Else $Z = 0$
SUB	0000 1001	$AC = AC - R$, If $(AC - R = 0)$ Then $Z = 1$ Else $Z = 0$
INAC	0000 1010	$AC = AC + 1$, If $(AC + 1 = 0)$ Then $Z = 1$ Else $Z = 0$
CLAC	0000 1011	$AC = 0$, $Z = 1$
AND	0000 1100	$AC = AC \wedge R$, If $(AC \wedge R = 0)$ Then $Z = 1$ Else $Z = 0$
OR	0000 1101	$AC = AC \vee R$, If $(AC \vee R = 0)$ Then $Z = 1$ Else $Z = 0$
XOR	0000 1110	$AC = AC \oplus R$, If $(AC \oplus R = 0)$ Then $Z = 1$ Else $Z = 0$
NOT	0000 1111	$AC = AC'$, If $(AC' = 0)$ Then $Z = 1$ Else $Z = 0$

- The LDAC, STAC, JUMP, JMPZ AND JPNZ instructions all require a 16-bit memory address represented by the symbol Γ.
- Since each byte of memory is 8-bit wide these instructions requires 3 bytes in memory. The first byte contains the opcode for the instruction and the remaining 2 bytes for theaddress.
- The instructions of this instruction set architecture can be grouped in to 3categories
  - Data transferinstructions
  - Program controlinstructions
  - Data operationalinstructions

- The NOP, LDAC, STAC, MVAC or MOVR instructions are **data transfer instructions**. The NOP operation performs no operation. The LDAC operation loads the data from the memory it reads the data from the memory location M[Γ]. The STAC performs opposite, copying data from AC to the memory locationΓ.
- The MOVAC instruction copies data from R to AC and MOVR instruction copies the data from R toAC.
- There are three **program control instructions** in the instruction set: JUMP, JUPZ and JPNZ. The JUMP is the unconditional it always jumps to the memory location Γ. For example JUMP 1234H instruction always jump to the memory location 1234H. The JUMP instruction uses the immediate addressing mode since the jump address is specified in theinstruction.

The other two program control instructions JUPZ and JPNZ are conditional. If their conditions are met Z=0 for JUPZ and Z=1 for JPNZ these instructions jump to the memory locationΓ.

- Finally data operations instructions are ADD, SUB, INAC and CLAC instructions are arithmetic instructions. The ADD instructions add the content of AC and R and store the result again in AC. The instruction also set the zero flag Z =1 if sum is zero or Z=0 if the sum is non zero value. The SUB operation performs the same but it subtracts the AC andR.
- INAC instruction adds 1 to the AC and sets Z to its proper value. The CLAC instruction always makes Z=1 because it clears the AC value that is AC to0.
- The last four data operation instructions are logical instructions as the name imply the AND, OR, and XOR instructions logically AND,OR and XOR the values AC and R and store the result in AC. The NOT instruction sets AC to its bitwisecomplement.

RELATIVELY SIMPLECOMPUTER:

- In this relatively simple computer we put all the hard ware components of the computer together in one system. This computer will have 8K ROM starting at address 0 fallowed by 8K RAM. It also has a memory mapped bidirectional I/O port at address8000H.
- First let us look at the CPU since it uses 16 bit address labeled A_{15} through A_0. System bus via pins through D_7 to D_0. The CPU also has the two control lines READ and WRITE.
- Since it uses the memory mapped I/O it does not need a control signal such as M.The relatively simple computer is shown in the figure below. It only contains the CPU details. Each part will be developed in thedesign.

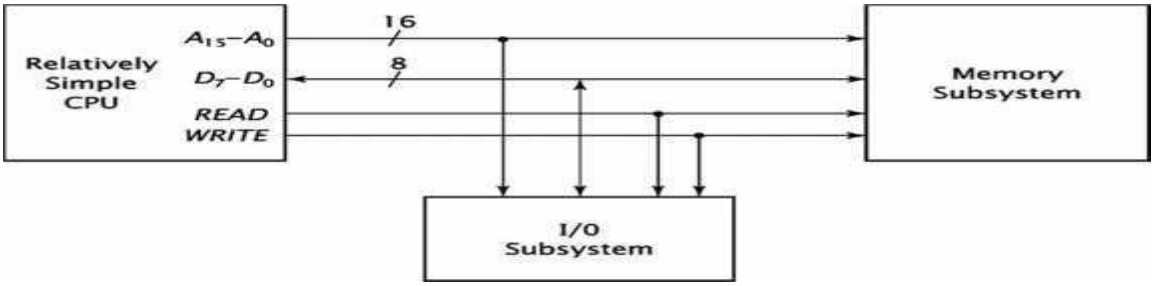


Fig: A relatively simple computer: CPU details only

- Toaccessamemorychiptheprocessormustsupplyanaddressusedbythechip.An8K memory chip has 2^{13} internal memory locations it has 13bit address input to select one of these locations. The address input of each chip receives CPU address bits A_{12} to A_0 from system address bus. The remaining three bits A_{15}, A_{14}, and A_{13} will be used to select one of the memory chips.
- The address range of ROM and RAMis

MODULE-2

ORGANIZATION OF A COMPUTER

Register transfer: Register transfer language, register transfer, bus and memory transfers, arithmetic micro operations, logic micro operations, And shift micro operations; Control MODULE: Control memory, address sequencing, micro program example, and design of control MODULE.

REGISTER TRANSFER:

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory MODULE is usually called a memory address register and is designated by the name MAR. Other designations for registers are PC (for program counter), IR (for instruction register, and R1 (for processor register). The individual flip-flops in an n-bit register are numbered in sequence from 0 through n − 1, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 2-1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 2-1(a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is PC. The symbol PC (0−7) or PC(L) refers to the low-order byte and PC(8−15) or PC(H) to the high-order byte.

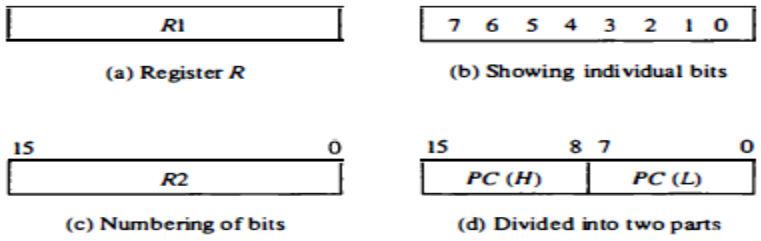
Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement denotes a transfer of the content of

$$R2 \leftarrow R1$$

register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we want the transfer to Occur only under a predetermined control condition. This can be shown by means of an if-then statement.

Figure: Block diagram of register.



$$\text{If } (P = 1) \text{ then } (R2 \leftarrow R1)$$

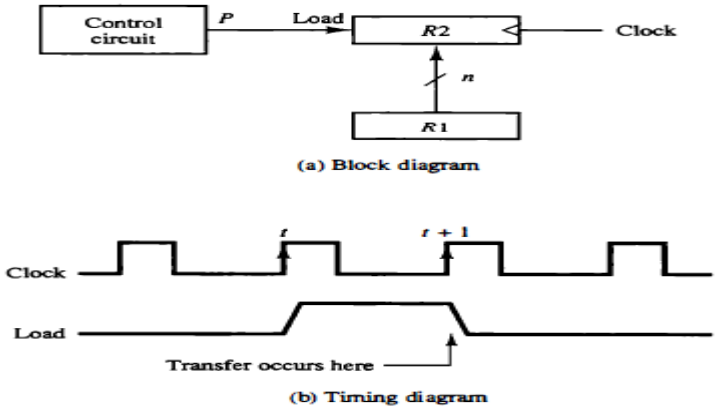
where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a control function. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows

$$P : R2 \leftarrow R1$$

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if P = 1.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.

Figure 2.2 shows the block diagram that depicts the transfer from R1 to R2. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input that is activated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As shown in the timing diagram, P is activated in the control



section by the rising edge of a clock pulse at time t. The next positive transition of the clock at time t + 1 finds the load input active and the data inputs of R2 are then loaded into the register in parallel. P may go back to 0 at time t + 1; otherwise, the transfer will occur with every clock pulse transition while P remains active.

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as P becomes active just after time t, the actual transfer dose not occur until the register is triggered by the next positive transition of the clock at time t + 1.

The basic symbols of the register transfer notation are listed in Table 2-1. Registers are denoted by capital letters, and numbers may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

T: R2 ← R1, R1←R2

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

TABLE: Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR, R2*
Parentheses ( )	Denotes a part of a register	*R2(0–7), R2(L)*
Arrow ←	Denotes transfer of information	*R2 ← R1*
Comma ,	Separates two microoperations	*R2 ← R1, R1 ← R2*

REGISTER TRANSFER LANGUAGE

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting

digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on date stored in registers are called microoperations. A microoperation is an elementary operation performed on the on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. Some of he digital components introduced

here in this chapter are registers that implement microoperations. For example, a counter with parallel load is capable of performing the micro operations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.
The internal hardware organization of a digital computer is best defined by specifying

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbolic representation to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers. The use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language. The term “register transfer” implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word “language” is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the statedmicrooperations. The symbolic designation introduced in this chapter will be utilized in subsequent chapters to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

BUS AND MEMORY TRANSFERS

A typical digital computer has many registers, and paths must be provided to transfer in formation form one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a

common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the onus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown is Fig. 2-3. Each register has four bits, numbered 0 through 3. The bus consists of four 4 × 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S_1 and S_0. In order not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labeled A_1. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When $S_1 S_0 = 00$, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0data inputs of the multiplexers. Similarly, register B is selected if $S_1 S_0 = 01$, and so on. Table 2-2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

In general, a bus system will multiples k registers of n bits each to produce an n-line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multiplexer must be k ×1 since it multiplexes k data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

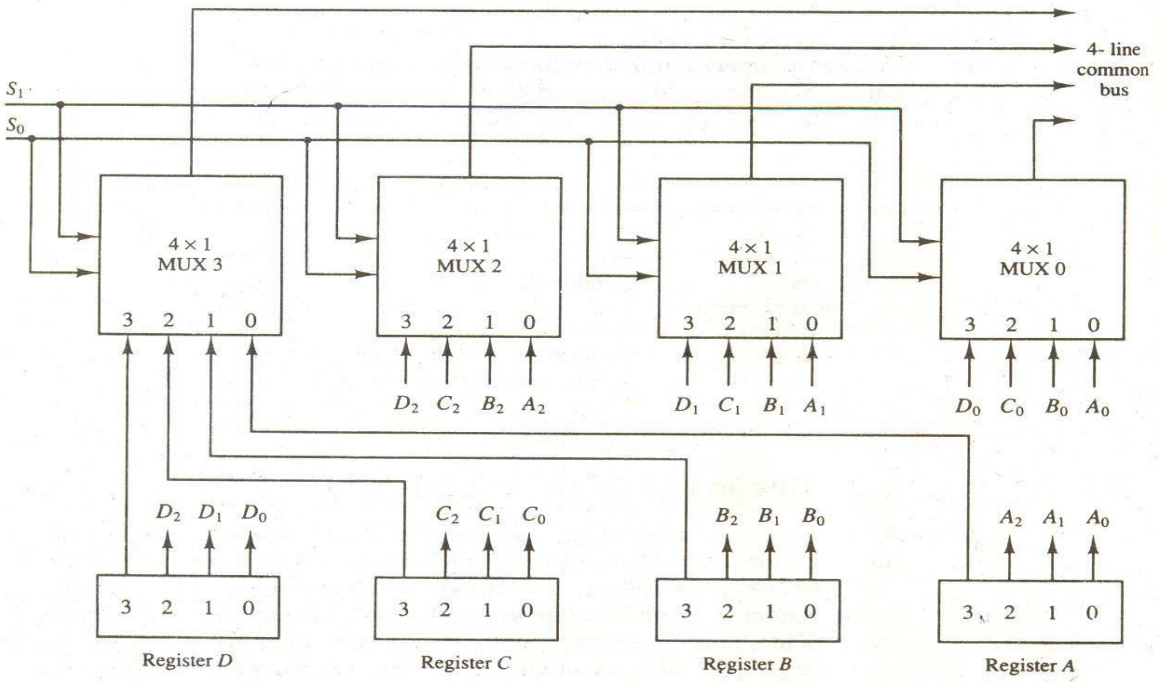


Figure : Bus system for four registers.

TABLE: Function Table for Bus of Fig

S_1	S_0	Register selected
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is includes in the statement, the register transfer is symbolized as follows:

BUS ← C,R1← BUS

The content of register C is placed on the bus, and the content of the bus is loaded into register R1 by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

R1 ← C

From this statement the designer knows which control signals must be activated to produce the transfer through the bus

THREE-STATE BUS BUFFERS

A bus system can be constructed with three-state gates instead of multiplexes. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state. The high-impedance state behaves like an open circuit which means that the output is disconnected and does not have a logic significance. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a three-state buffer gate is shown in Fig. 2-4. it is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate gives to a high-impedance state, regard less of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

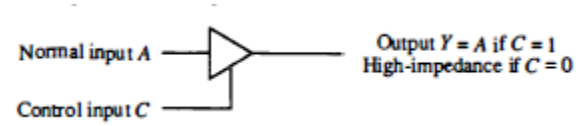
The construction of a bus system with three- state buffers is demonstrated in Fig. 2-5. The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs.) the control inputs to the buffers determine which of the four normal inputs will communicate with the us line. No more than one buffer may be in the active state at any given time. ;the connected buffers must be controlled so that only one three-state buffer bas access to the bus line while all other buffers are maintained in a high-impedance state.

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of tits four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation will reveal that Fig. 2-5 is another way of constructing a 4 × 1 multiplexer since the circuit can replace the multiplexer in Fig. 2-3.

To construct a common bus for four registers on n bits each using three- state buffers, we need n circuits with four buffers in each as shown in Fig. 2-5. Each group of four buffers receives one significant bit from the four registers. Each common

output produces one of the lines for the common bus for a total of n lines. Only one decoder is necessary to select between the four register.

Figure : Graphic symbols for three-state buffer.



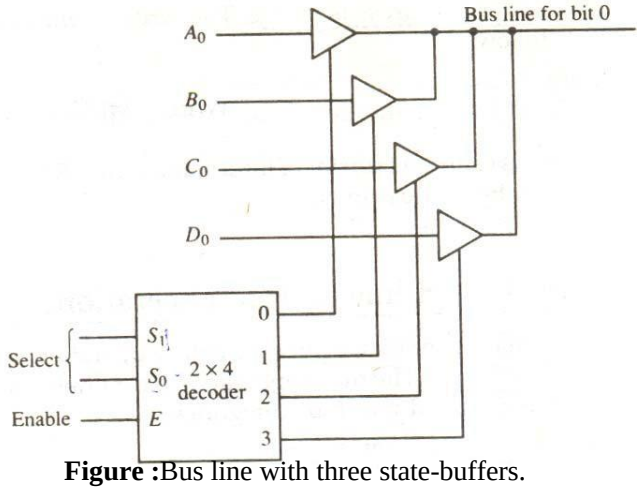


Figure : Bus line with three state-buffers.

ARITHMETIC MICRO OPERATIONS

A micro operation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations transfer binary information from one register to another.
2. Arithmetic micro operations perform arithmetic operation on numeric data stored in registers.
3. Logic micro operations perform bit manipulation operations on non-numeric data stored in registers.
4. Shift microoperations perform shift operations on data stored in registers.

The register transfer microoperation was introduced in Sec. 1 -2. This type of microoperation dies not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperation change the information content during the transfer. In this section we introduce a set of arithmetic microoperations.

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement

$$R3 \leftarrow R1 + R2$$

specifies an add microoperation. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 4-3. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

$$R3 \leftarrow \overline{R1} + R2 + 1$$

R2 is the symbol for the 1's complement of R2. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of R1 to the 2's complement of R2 is equivalent to R1 – R2.

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

TABLE 2-3 Arithmetic Microoperations

The increment and decrement microoperations are symbolized by plus-one and minus-one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.

The arithmetic operations of multiply and divide are not listed in Table 2-3. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shiftmicrooperations. To specify the hardware in such a case requires a list of statements that use the basic microoperations of add, subtract, and shift (see Chapter 10).

BINARY ADDER

To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder (see Fig. 2-6). The digital circuit that generates the arithmetic sum of two binary numbers of any lengths is called a binryadder. The binary adder is constructed with full-adder circuitsers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder (see Fig. 2-6). The digital circuit that generates the arithmetic sum of two binary

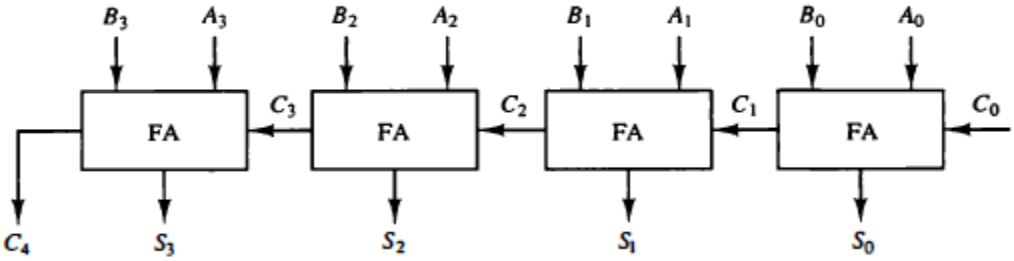


Figure 2-6 4-bits binary adder.

connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 2-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augends bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C_0 and the output carry is C_4. This S outputs of the full-adders generate the required sum bits.

An n-bit binary adder requires n full –adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The n data bits for the A inputs come from tone register (such as R1), and the n data bits for

the B inputs come from another register (such as R2). The sum can be transferred to a third register or to one of the sourc3e registers (R1 or R2), replacing its previous content,

Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements. Remember that the subtraction A –B can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4 -bit adder-subtract or circuit is shown in Fig. 2-7. The mode input M controls the operation. When M = 0 the circuit is an adder and when M = 1 the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When M = 0, we have $B \oplus 0 = B$. the full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When M = 1, we have $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the

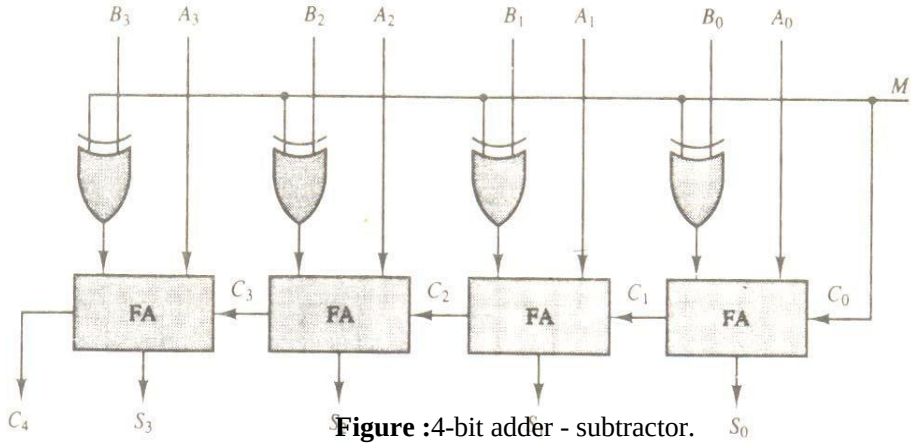


Figure :4-bit adder - subtractor.

2's complement of B. For unsigned, this gives A – B if A ≥ B or the 2's complement of (B−A) if A < B. For signed numbers, the result is A – B provided that there is no overflow.

BINARY INCREMENTER

The increment microoperation adds one to a number in a register. For example, if a 4-bit register ahs a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter. Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders connected in cascade.

The diagram of a 4-bit combinational circuit incrementer is shown in Fig. 2-8. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3. The output carry C_4 will be 1 only after incrementing binary 1111. This also causes outputs S_0through $S_3$to go to 0.

The circuit of Fig. 2-8 can be extended to an N-bit binary incremented by extending thediagram to include n half-adders. The least significant bit must have one input connected to logic-. The other inputs receive the number to be incremented or the

carry from the previous stage.

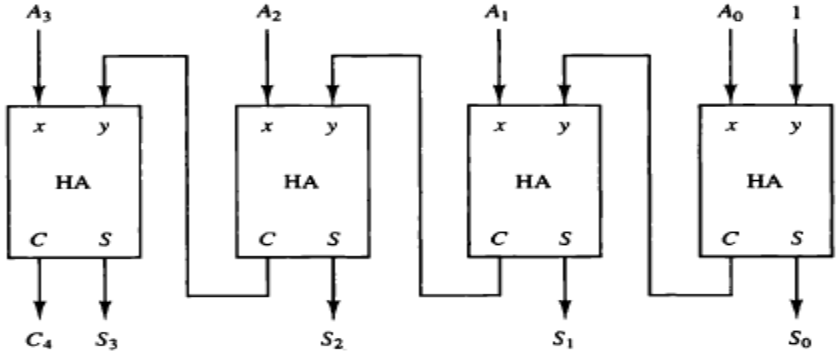


Figure :4-bit binary incrementer

ARITHMETIC CIRCUIT

The arithmetic microoperations listed in Table 2-3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

The diagram of a 4-bit arithmetic circuit is shown in Fig. 2-9. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the for inputs from B are connected to the data inputs of the multiplexers. The multiplexer's data inputs also receive the complement of B. The other two data inputs are connected to logic-0 ad logic -1. Logic-0 is fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, S_1 and S_0. The input carry C_{in} goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

The output of the binary adder is calculated from the following arithmetic

$$\text{sum: } D = A + Y + C_{in}$$

where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. C_{in} is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S_1 and S_0 ad making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 2-4.

TABLE 2-4 Arithmetic Circuit Function Table

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

When S_1 S_0 = 00, the value of B is applied to the Y inputs of the adder. If C_{in} = 0, the output D = A + B. If C_{in} = 1, output D = A + B + 1. Both cases perform the add microoperation with or without adding the input carry.

When S_1S_0 = 01, the complement of B is applied to the Y inputs of the adder. If C_{in} = 1, thenD =A + B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtract with borrow, that is, A – B – 1.

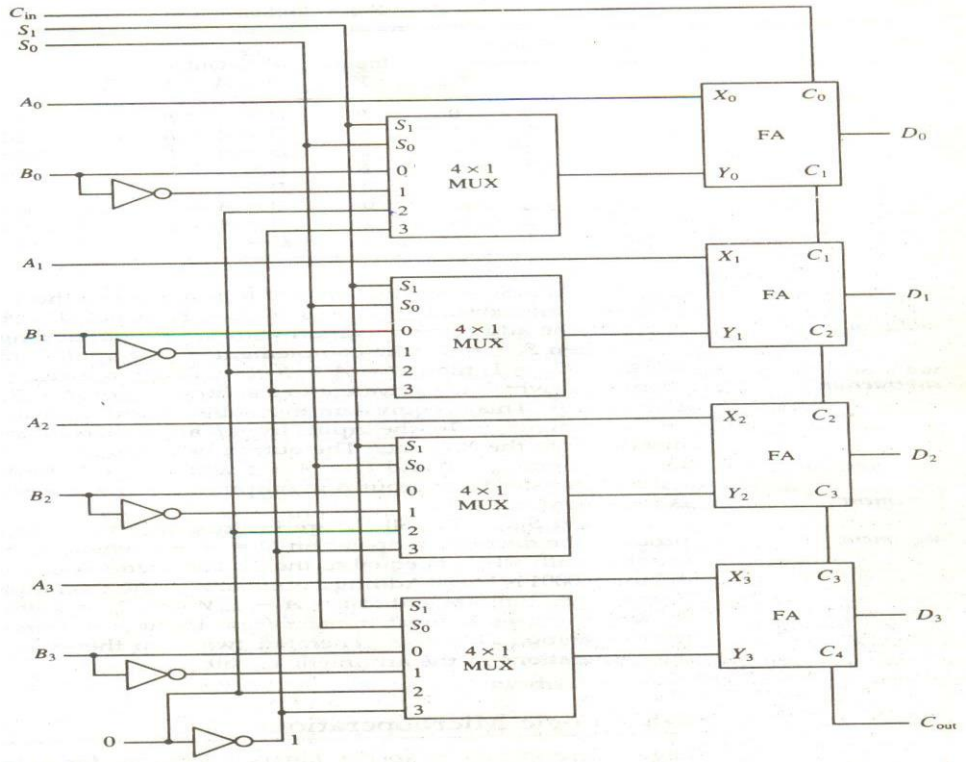


Figure :4-bit arithmetic circuit

complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces F = A +2's complement of 1 = A – 1. When C_{in} = 1, then D = A – 1 + 1 = A, which causes a direct transfer from input A to output D. Note that the microoperation D = A is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

LOGIC MICROOPERATIONS

Logic micro operations specify binary operations for strings of bits stored in registers. These operations consider each it of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers R1 and R2 is symbolized by the statement

$$P: R1 \leftarrow R1 \oplus R2$$

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1. As a numerical example, assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation:

1010 Content of R1
1100 Content of R2
0110 Content of R1 after P = 1

The content of R1, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in R2 and previous values of R1. the logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The Symbol $\vee$ will be used to denote an OR microoperation and the symbol $\wedge$ to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol +, when used to symbolize an arithmetic plus, from a logic OR operation. Although the +

symbol has two meaning, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation. For example, in the statement

$$P + Q: R1 \leftarrow R2 + R3, R4 \leftarrow R5 \vee R6$$

the + between P and Q is an OR operation between two binary variables of a control function. The + between R2 and R3 specifies an add microoperation. The OR microoperation is designated by the symbol $\vee$ between register R5 and R6.

LIST OF LOGIC MICROOPERATIONS

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 2-5. in this table, each of the 16 columns F_0 through F_{15} represents a truth table of one possible Boolean function for the
two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 2-6. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B. It is important to realize that the Boolean functions listed in the first column of Table 2 -6 represent a relationship between two binary variables x and y. The logic microoperations listed in the second column represent a relationship between the binary content of two registers A and B. Each bit of the register is treated as a binary variable and the microoperation is performed on the string of bits stored in the registers.

Table 2-5 Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

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

TABLE: Sixteen Logic Micro operations

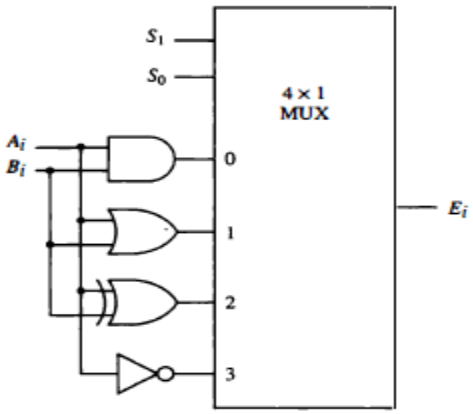
HARDWARE IMPLEMENTATION

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four --- AND, OR, XOR (exclusive-OR), and complement by which all others can be derived.

Figure 4-10 shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for i = 0, 1, 2, …, N –1. The selection variables are applied to all stages. The function table in Fig. 2-10 (b) lists the logic microoperations obtained for each combination of the selection variables

SOME APPLICATIONS

Logic micro operations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated



(a) Logic diagram

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

(b) Function table

Figure : One stage of logic circuit

by logic microoperations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

The *selective-set* operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

1010	A before
1100	B (logic operand)
1110	A after

The two leftmost bits of B are 1's, so the corresponding bits of A are set to 1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of A with corresponding 0's in B remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A. Therefore, the OR microoperation can be used to selectively set bits of a register.

The *selective-complement* operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

1010	A before
1100	B (logic operand)
0110	A after

Again the two leftmost bits of B are 1's, so the corresponding bits of A are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

1010	A before
1100	B (logic operand)
0010	A after

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB'. The corresponding logic microoperation is

$$A \leftarrow A \wedge \overline{B}$$

The *mask* operation is similar to the selective-clear operation except that the bits of *A* are cleared only where there are corresponding 0's in *B*. The mask operation is an AND micro operation as seen from the following numerical example:

```
1010    A before
1100    B (logic operand)
1000    A after masking
```

The two rightmost bits of *A* are cleared because the corresponding bits of *B* are 0's. The two leftmost bits are left unchanged because the corresponding bits of *B* are 1's. The mask operation is more convenient to use than the selective-clear operation because most computers provide an AND instruction, and few provide an instruction that executes the microoperation for selective-clear.

The *insert* operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an *A* register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

```
0110 1010    A before
0000 1111    B (mask)
0000 1010    A after masking
```

and then insert the new value:

```
0000 1010    A before
1001 0000    B (insert)
1001 1010    A after insertion
```

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The *clear* operation compares the words in *A* and *B* and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

```
1010    A
1010    B
0000    A ← A ⊕ B
```

When *A* and *B* are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

SHIFT MICRO OPERATIONS

Shift micro operations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the right most position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

A logical shift is one that transfers 0 through the serial input. We will adopt the symbols **shl** and **shr** for logical shift-left and shift-right microoperations. For example:

R1 ← Sh1 R1
R2 ←shr R2

are two microoperations that specify a 1-bit shift to the left of the content of register R1 and a 1- bit shift to the right of the content of register R2. The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols cil and cir for the circular shift left and right, respectively. The symbolic notation for the shift microoperation is shown in Table 2-7.

Symbolic designation	Description
$R \leftarrow$shl R	Shift-left register R
$R \leftarrow$shr R	Shift-right register R
$R \leftarrow$cil R	Circular shift-left register R
$R \leftarrow$cir R	Circular shift-right register R
$R \leftarrow$ashl R	Arithmetic shift-left R
$R \leftarrow$ashr R	Arithmetic shift-right R

TABLE: Shift Microoperations

An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or

Figure : Arithmetic shift right.

divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus R_{n-1}remains the same, R_{n-2} receives the bit from R_{n-1}, and so on tfor the other bits in the register. The bits in R_0 is lost.

The arithmetic shift-left inserts a 0 into R_0, and shifts all other bits to the left. The initial bit ofR_{n-1} is lost and replaced by the bit from R_{n-2}.. A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occursafter an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to R_{n-2}. An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow.

$$V_s = R_{n-1} \oplus R_{n-2}$$

If $V_s = 0$, there is no overflow, bit if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into

the overflow flip-flop with the same clock pulse that shifts the register.

Control memory

The function of the control MODULE in a digital computer is to initiate sequences of microoperations. The number of different types of microoperations that are available in a given system is finite. The complexity of the digital system is derived from the number of sequences of microoperations that are performed. When the control signals are generated by hardware using conventional logic design techniques, the control MODULE is said to be hardwired. Microprogramming is a second alternative for designing the control MODULE of a digital computer. The principle of microprogramming is an elegant and systematic method for controlling the micro operation sequence in a digital computer.

The control function that specifies a microoperation is a binary variable. When it is in one binary state, the corresponding microoperation is executed. A control variable in the opposite binary state dies not change the state of the registers in the system. The active state of a control variable may be either the 1 state or the 0 state, depending on the application. In a bus-organized system, the control signals that specify micro operation are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic MODULEs.

The control MODULE initiates a series of sequential steps of microoperations. During any given time, certain microoperations are to be initiated, while others remain idle. ;the control variables at any given time can be represented by a string of 1's and 0' a called a control word. As such, control words can be programmed to perform various operations on the components are stored in memory is called a micro programmed control MODULE. Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more micro operations fro the system. A sequence of microinstructions constitutes a microprogram. Since alterations of the microprogram are not need once the control MODULE is in operation, the control memory can be a read-only memory (ROM). The content of the words in ROM are fixed and cannot be altered by simple programming since no writing capability is available in the ROM. ROM words are made permanent during he hardware production of the MODULE.

The use of a microprogram involves placing all control variables in words of ROM for use by the control MODULE through successive read operations. The content of the word in ROM at a given address specifies a microinstruction.

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control MODULEs that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory that is part of a control MODULE is referred to as a control memory.

A computer that employs a micro programmed control MODULE will have two separate memories: a main memory and a control memory. The main memory is available to the user for storing the program. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data. In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations. Each machine instruction initials a series of microinstructions in control memory. These microinstructions
generate the rations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.

The general configuration of a microprogrammed control MODULE is demonstrated in the block diagram of Fig. 4-1. The control memory is assumed to be a ROM, within which all control information is permanently stored. Thecontrol

memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor.

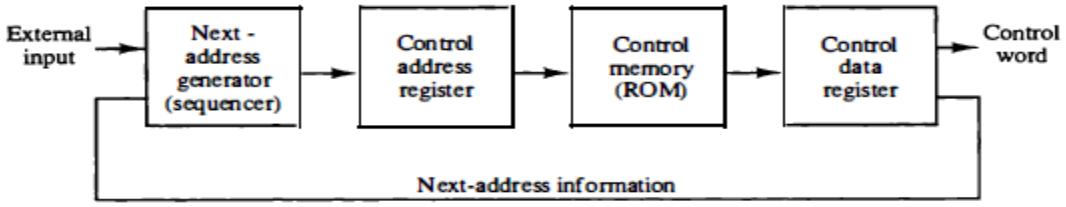


Fig.Microprogrammed control organization

Once these operations are executed, the control must determine the nest address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the microoperations are being executed, the next address is computer in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory. ;the address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.

The control data register holds the present microinstruction while the next address is computed and read from memory; the data register is sometimes called a pipeline register. It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock., with one clock applied to the address register and the other to the data register.

The system can operate without the control data register by applying a single-phase clock to the address register. The control word and next-address information are taken directly from the control memory. ;it must be realized that a ROM operates as a combinational circuit, with the address value as the input and the corresponding word as the output. The content of the specified word in ROM remains in the output wires as long as its address value remains in the address register. No read signal is needed as in a random-access memory.Each clock pulse will execute the microoperations specified by the control word and also transfer a new address to the control address register. In the example that follows we assume a single-phase clock and therefore we do not use a control data register. In this way the address register is the only component in the control system that receives clock pulses. The other two components: the sequencer and the control memory are combinational circuits and do not need a clock.

The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstruction for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

It should be mentioned that must computers based on the reduced instruction set computer (RISC) architecture concept, use hardwired control rather than a control memory with a microprogram.

ADDRESS SEQUENCING

Microinstructions are stored in control memory in groups, with each group specifying routine. Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. The hardware

that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another. To appreciate the address sequencing in a microprogram control MODULE, let us enumerate the steps that the control must undergo during the execution of a single computer instruction.

An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions. At the end of the fetch routine, the instruction is in the instruction register of the computer.

The control memory next must go through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers. The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction. When the effective address computation routine is completed, the address of the operand is available in the memory address register.

The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor register depend on the operation code part of the instruction. Each instruction has its own microprogram routine stored in a given location of control memory. The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process. A mapping procedure is a rule that transforms the instruction code into a control memory address. Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register, but sometimes the sequence of microoperations will depend on values of certain status bits in processor registers. Micro programs that employ subroutines will require an external register for storing the return address. Return addresses cannot be stored in ROM because the MODULE has no writing capability.

When the execution of the instruction is completed, control must return to the fetch routine. This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine. In summary, the address sequencing capabilities required in control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on statues bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Figure shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction. The microinstruction in control memory contains a set of bits to initiate micro oprerations in computer registers and other bits to specify the method by which the next address is obtained. The diagram shows four different paths from which the control address register (CAR) receive the address. The incremented increments the content of the control address register by one, to select the next microinstruction in sequence. Branching is achieved by specifying the branch address in one of the fields of the microinstruction. Conditional branching is obtained by using part of the microinstruction to select a specific statues bit in order to determine its condition. An external address is transferred into control memory via a mapping logic circuit. The return address for a subroutine is stored in a special register whose value is then used when the microprogram wishes to return form the subroutine

CONDITIONAL BRANCHING

The branch logic provides decision-making capabilities in the control MODULE. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions. Information in these bits can be tested and actions initiated based on their

condition: whether their value is 1 or 0. The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.

The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented.

This can be implemented with a multiplexer. Suppose that there are eight status bit conditions in the system. Three bits in the microinstruction are used to specify any one of eight status bit conditions. These three bits provide the selection variables for the multiplexer. If the selected status bits in the 1 state, the output of the multiplexer is 1; otherwise, it is 0. A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register. A 0 output in the multiplexer causes the address register to be incremented. In this configuration, the microprogram follows one of two possible paths, depending on the value of the selected status bit.

An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register. This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to 1. A reference to this bit by the status bit select lines from control memory causes the branch address to loaded into the control address register unconditionally

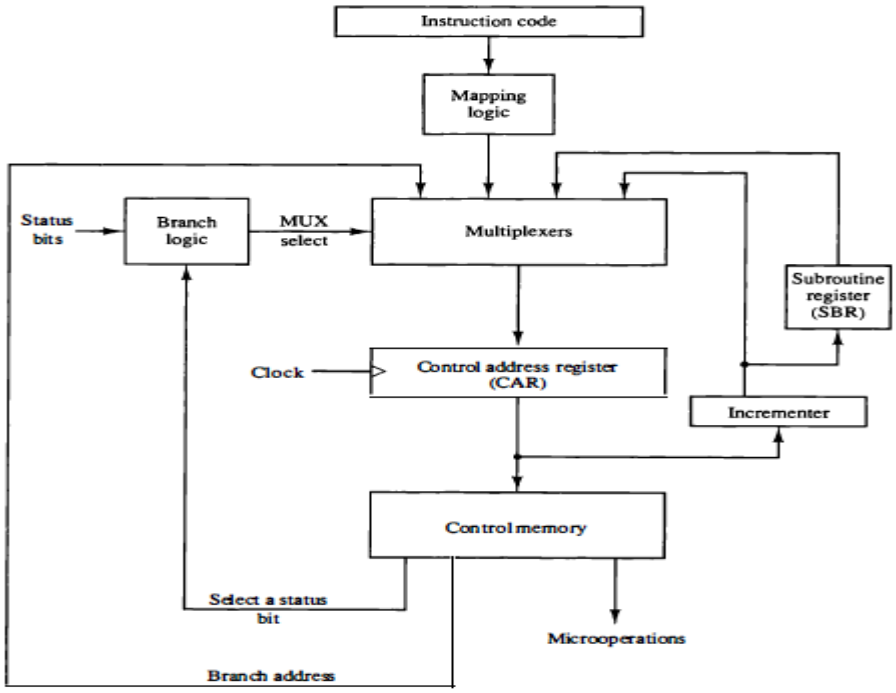


Fig. Selection of Address for control memory.

MAPPING OF INSTRUCTION

A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in Fig. 4-3 has an operation code of four bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a micro program routine in control memory that executes the instruction. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in fig. 4-3. This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two lest significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions. If the routine needs more than four microinstructions, it can

use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory cautions would be available for other routines One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function. In this configuration, the bits of the instruction specify the address of a mapping ROM. The contents of the mapping ROM give the bits for the control address register. In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory. The mapping concept provides flexibility for adding instruction for control memory as the need arises.

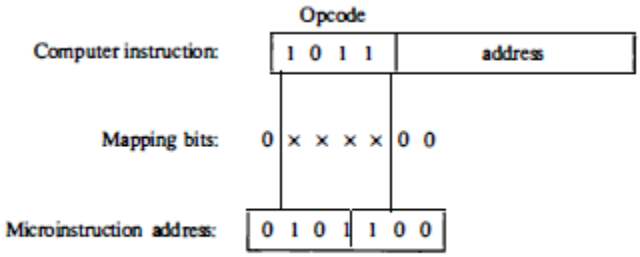


Figure : Mapping from instruction code to microinstruction address.

The mapping function is sometimes implemented by means of an integrated circuit called programmable logic device or PLD. A PLD is similar to ROM in concept except that it uses AND and OR gates with internal electronic fuses. The interconnection between inputs, AND gates, OR gates, and outputs can be programmed as in ROM. A mapping function that can be expressed in terms of Boolean expressions can be implemented conveniently

SUBROUTINES

Subroutines are programs that are used by other routines to accomplish a particular task. A subroutine can be called from any point within the main body of the microprogram. Frequently, many micro programs contain identical sections of code. Micro instructions can be saved by employing subroutines that use common sections of microcode. For example, the sequence of microoperation needed to generate the effective address of the operand for an instruction is common to all memory reference instructions. This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Micro programs that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return. This may be accomplished by placing the incremented output form the control address register into a subroutine register and branching to the beginning of the subroutine. The subroutine register can then become the source for transferring the address for the return to the main routine. The best way to structure a register file that stores addresses for subroutines is to organize the registers in a last-in, first-out (LIFO) stack.

MICROPROGRAM EXAMPLE

Once the configuration of a computer and its micro programmed control MODULE is established, the designer's task is to generate the microcode for the control memory. This code generation is called microprogramming and is a process similar to conventional machine language programming. To appreciate this process, we present here a simple digital computer and show how it is micro programmed. The computer used here is similar but not identical to the basic computer.

COMPUTER CONFIGURATION

The block diagram of the computer is shown in Fig. It consists of two memory MODULEs: a main memory for storing instructions and data, and a control memory for storing the microprogram. Four register are associated with the processor MODULE and two with the control MODULE. The processor registers are program counter PC, address register AR, data register DR, and accumulator register

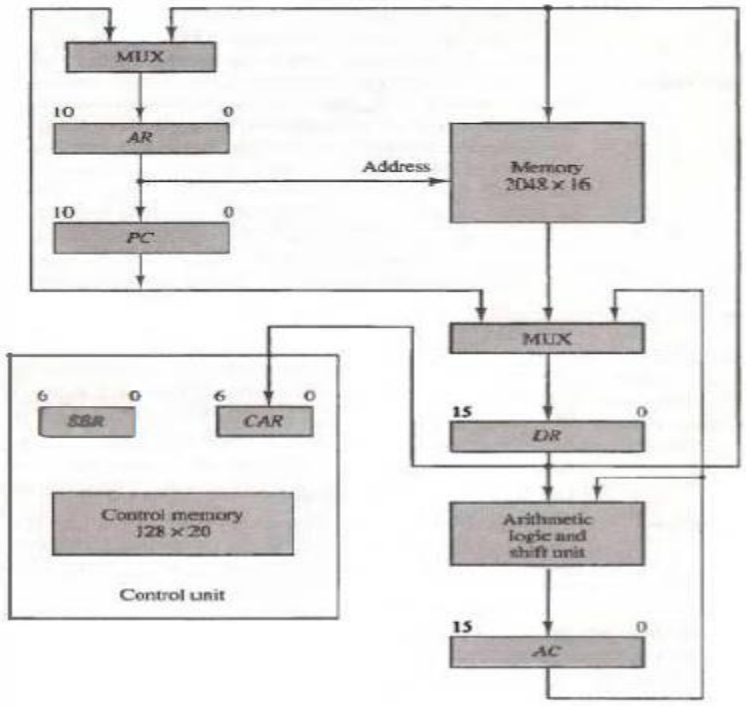


Figure: Computer hardware configuration

DR, and accumulator register AC. The function of these registers is similar to the basic computer. The control MODULE has a control address register CAR and a subroutine register SBR. The control memory and its registers are organized as a micro programmed control MODULE, as shown in Fig. 4-2.

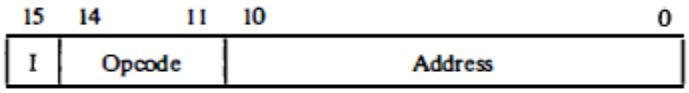
The transfer of information among the register in the processor is done through multiplexers rather than a common bus. DR can receive information from AC, PC, or memory. AR can receive information from PC or DR, PC can receive information only from AR. The arithmetic, logic, and shift MODULE performs micro opreations with data from AC and DR and places the result in AC. Note that memory receive its address from AR. Input data written to memory come from DR, and data read from memory can go only to DR.

The computer instruction format is depicted in Fig. 4-5(a). It consists of three fields: a 1-bit field for indirect addressing symbolized by I, a 4-bit operation code (opcode), and an 11-bitaddress field. Figure 4-5(b) lists four of the 16 possible memory-reference instructions. The ADD instruction adds the content of the operand found in the effective address to the content of AC. The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of AC into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.

It will be shown subsequently that each computer instruction must be micro programmed. In order not to complicate the microprogramming example, only four instructions are considered here. It should be realized that 12 other instructions can be included and each instruction must be micro programmed by the procedure outlined below.

MICROINSTRUCTION FORMAT

The microinstruction format for the control memory is shown in Fig. 4-6. The 20 bits of the microinstruction are divided into four functional parts. The three fields F1, F2, and F3 specify microoperations for the computer. The CD field

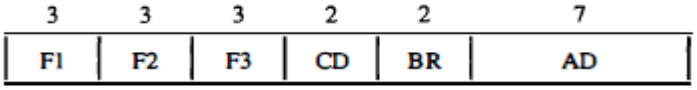


(a) Instruction format

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

EA is the effective address

(b) Four computer instructions



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure : Microinstruction code format (20 bits).

selects status bit conditions. The BR field specifies the type of branch to be used. The AD fieldcontains a branch address. The address field is seven bits wide, since the control memory has $128 = 2^7$ words.

The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations as listed in Table 4-1. This gives a total of 21 microoperations. No more than three microoperations can be chosen for a microinstruction, one from each field. If fewer than three microoperations are used, one or more of the fields will use the binary code 000 for no operation. As an illustration, a microinstruction can specify two simultaneous micro operation from F2 and F3 and none from F1.

M [AR]	2 = 100
C ← PC + 1	3 = 101

The nine bits of the microoperation fields will than be 000 100 101. It is important to realize that two or more conflicting microoperations cannot be specified simultaneously. For example, a microoperation field 010 001 000 has no meaning because it specifies the operations to clear AC to 0 and subtract DR from AC at the same time.

Each microoperation in Table 4-1 is defined with a register transfer statement and is assigned a symbol for use in a symbolic microprogram. All transfer-type microoperations symbols use five letters. The first two letters designate the source register, the third letters is always a T, and the last two letters designate the destination register. For example, the microoperation that specifies the transfer AC ← DR (F1 = 100) has the symbol DRTAC, which stands for a transfer from DR to AC.

The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 7-1. The first condition is always a 1, so that a reference to CD = 00 (or the symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation. The indirect bit I is available from bit 15 of DR after an instruction is read from memory. The sign bit of AC provides the next

status bit. The zero value, symbolized by Z, is a binary variable whose value is equal to 1 if all the bits in AC are equal to zero. We will use the symbols U, I, S, and Z for the four status bits when we write micro programs in symbolic form.

The BR (branch) field consists of two bits. It is used, in conjunction with address field AD, to choose the address of the next microinstruction. As shown in Table 4 -1, when BR = 00, the control performs a jump (JMP) operation (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation. The two operations are identical except that a call microinstruction stores the return address in the subroutine register SBR. The jump and call operations depend on the value of the CD field. If the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register CAR. Otherwise, CAR is incremented by 1.

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

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

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

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

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

Table: Symbols and Binary Code for Microinstruction Fields

The return from subroutine is accomplished with a BR field equal to 10. This causes the transfer of the return address from SBR to CAR. The mapping from the operation code bits of the instruction to an address for CAR is accomplished when the BR field is equal to 11. This mapping is as depicted in Fig. 4-3. the bits of the operation code are in DR after an instruction is read from memory. Note that the last two conditions in the BR field are independent of the values in the CD and AD fields

SYMBOLIC MICROPROGRAM

The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address 0××××0, where ×××× are the four bits of the operation code. For example, if the instruction is an ADD instruction whose operation code is 0000, the MAP microinstruction will transfer to CAR the address 0000000, which is the start address for the ADD routine in control memory. The first address for the BRANCH and STORE routines are 0 0001 00 (decimal 4) and 0 0010 00 (decimal 8), respectively. The first address for the other 13 routines are at address values 12, 16, 20,…., 60. This gives four words in control memory for each routine.

In each routine we must provide microinstructions for evaluating the effective address and for executing the instruction. The indirect address mode is associated with all memory-reference instructions. A saving in the number of control memory words may be achieved if the microinstructions for the indirect address are stored as a subroutine. This subroutine, symbolized by INDRCT, is located right after the fetch routine, as shown in Table 7-2. the table also shows the symbolic microprogram for the fetch routine and the microinstruction in the ADD routine calls subroutine INDRCT, conditioned on status bit I. If I = 1 , a branch to INDRCT occurs and the return address (address 1 in this case) is stored in the subroutine register SBR. The INDRCT subroutine has two microinstructions:

```
INDRCT:    READ      U     JMP     NEXT
           DRTAR     U     RET
```

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

Remember that an indirect address considers the address part of the instruction as the address where the effective address is stored rather than the address of the operand. Therefore, the memory has to be accessed to get the effective address, which is then transferred to AR. The return from subroutine (RET) transfers the address from SBR to CAR, thus returning to the second microinstruction of the ADD routine.

The execution of the ADD instruction is carried out by the microinstruction at addresses 1 and 2. The first microinstruction reads the operand from memory into DR. the second microinstruction performs an add microoperation with the content of DR

and AC and then jumps back to the beginning of the fetch routine.

The BRANCH instruction should cause a branch to the effective address if AC < 0. The AC will be less than zero if its sign is negative, which is detected from status bit S being a 1. The BRANCH routine in Table 4-2 starts by checking the value of S. If S is equal to 0, no branch occurs and the next microinstruction causes a jump back to the fetch routine without altering the content of PC. If S is equal to 1, the first JMP microinstruction transfers control to location OVER. The microinstruction at this location calls the INDRCT subroutine if I = 1. The effective address is then transferred from AR to PC and the microprogram jumps back to the fetch routine.

The STORE routine again uses the INDRCT subroutine if I = 1. The content of AC is transferred into DR. A memory write operation is initiated to store the content of DR in a location specified by the effective address in AR.

The EXCHANGE routine reads the operand from the effective address and places it in DR and AC are interchanged in the third microinstruction. This interchange is possible when the registers are of the edge-triggered type. The original content of AC that is now in now in DR is stored back in memory.

Note that Table 4-2 contains a partial list of the microprogram. Only four out of 16 possible computer instructions have been micro programmed. Also control memory words at locations 69 to 127 have not be4en used. Instructions such as multiply, divide, and others that require a long sequence of microoperations will need more than four microinstructions for their execution. Control memory words 69 to 127 can be used for this purpose.

DESIGN OF CONTROL MODULE

The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in instruction formats provide control bits to initiate microoperations in the system,

special bits to specify the way that the next address is to be evaluated, and an address field for branching. The number of control bits that initiate micro operation can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2 microoperations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

The encoding of control bits was demonstrated in the programming example of the preceding section. The nine bits of the microoperation field are divided into three subfields of three bits each. The control memory output of each subfield must be decoded to provide the distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the processor MODULE.

Figure 4-7 shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 × 8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation as specified in Table 7-1. For example, when FI = 101 (binary 5), the next clock pulse transition transfers the content of DR (0-10) to AR (symbolized by DRTAR in Table 7-1). Similarly, when F1 = 101 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig. 4-7, outputs 5 and 6 of decoder F1 are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR. The multiplexers select the information from DR when output 5 is active and from PC when output 5 in inactive. The transfer into AR occurs with a clock pulse transition only when output 5 or output 6 of the decoder are active. The other outputs of the decoders that initiate transfers between registers must be connected in a similar fashion.

The arithmetic logic shift MODULE can be designed, instead of using gates to generate the control signals marked by the

symbols AND, ADD, and DR in Fig 4.7, these inputs will now come from the outputs of the decoders associated with the symbols AND, ADD, and DRTAC, respectively as shown in Fig. 4-7. the other output of the decoders that are associated with an AC operation must also be connected to the arithmetic logic shift MODULE in a similar fashion .

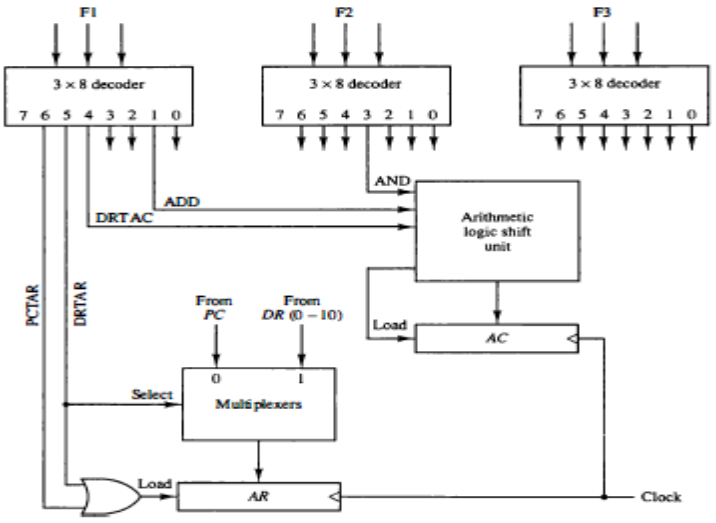


Figure : Decoding of microoperation fields

MICROPROGRAM SEQUENCER

The basic components of a micro programmed control MODULE are the control MODULE are the control memory and the circuits that select the next address. The address selection part is called a microprogram sequencer. A microprogram sequencer can be constructed with digital functions to suit a particular application. However, just as there are large ROM MODULEs available in integrated circuit packages, so are general purpose sequencers suited for the construction of microprogram control MODULEs. To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.

The purpose of a microprogram sequencer is to present is to present an address to the control memory so that a microinstruction may be read and executed. The next-address logic of the sequencer determines the specific address source to be loaded into the control address register. The choice of the address source is guided by the next-

address information bits that the sequencer receives from the present microinstruction. Commercial sequencers include within the MODULE an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls. ;some sequencers provide an output register which can function as the address register for the control memory.

To illustrate the internal structure of a typical microprogram sequencer we will show a particular MODULE that is suitable for use in the microprogram computer example developed in the preceding section. The block diagram of the microprogram sequencer is shown in Fig. 4 -8. The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register CAR. The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit. The output from CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR. The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction. Although the diagram shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time. A push and pop operation, in conjunction with a stack pointer, stores and retrieves the return address during the call and return microinstructions.

The CD (condition) field of the microinstruction selecting one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test)variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the MODULE. Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operations. Some commercial sequencers have three or four inputs in addition to the T input and thus provide a wider range of operations.

The input logic circuit in Fig. 4-8 has three inputs, I_0, I_1, and T, and three outputs, S_0 , S_1 and L. variables S_0 and S_1 select one of the source addresses for CAR. Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer. For example, with S_1 S_0 = 10, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR. Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

The truth table for the input logic circuit is shown in Table 4-4. inputs I_1 and I_0 are identical to the bit values in the BR field. The function listed in each entry was defined in Table 4-1. The bit values for S_1 and S_0 are determined from the stated function and the path in the multiplexer that establishes the required transfer. The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T = 1). The truth table can be used to obtain the simplified Boolean functions for the input logic:

$$S_1 = I_1$$

$$S_0 = I_1 I_0 + I'_1 T L = I'_1 I_0 T L = I'_1 I_0 T$$

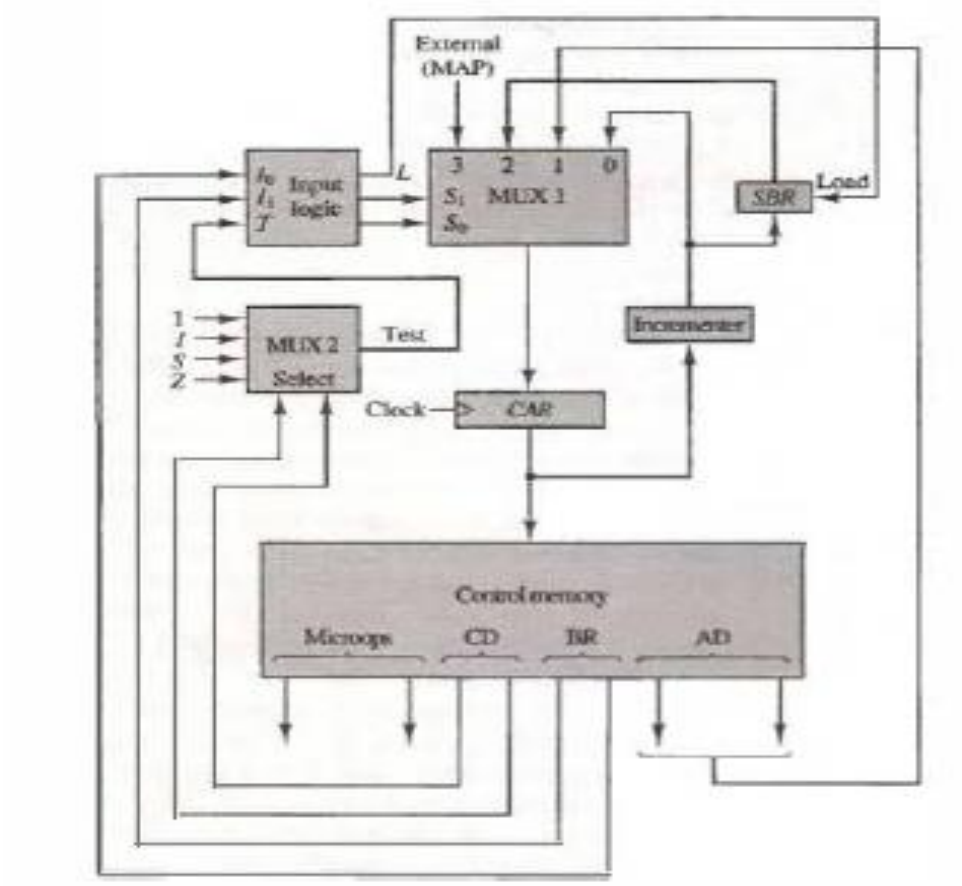


Figure: Microprogramsequencer for a control memory

BR Field		Input			MUX 1		Load *SBR*
		I_1	I_0	T	S_1	S_0	L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	x	1	0	0
1	1	1	1	x	1	1	0

Table: Input Logic Truth Table for Microprogram Sequencer

The circuit cab be constructed with three AND gates, an OR gate, and an inverter.

Note that the incremented circuit in the sequencer of Fig. 4-8 is not a counter constructed with flip-flops but rather a combinational circuit constructed with gates. A combinational circuit incrementer can be designed by cascading a series of half-adder circuits. The output carry from one state must be applied to the input of the next stage. One input in the first least significant stage must be equal to 1 to provide the increment-by-one operation

MODULE-3

CPU AND COMPUTER ARITHMETIC

CPU design: Instruction cycle, data representation, memory reference instructions, input-output, and interrupt, addressing modes, data transfer and manipulation, program control. Computer arithmetic: Addition and subtraction, floating point arithmetic operations, decimal arithmetic MODULE.

INSTRUCTION CYCLE

A program residing in the memory MODULE of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases. In the basic computer each instruction cycle consists or the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

FETCH AND DECODE

Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0. After each clock pulse, SC is incremented by one so that the timing signals go through a sequence T_0, T_1, T_2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T_0: AR ← PC

T_1: IR ←M[AR], PC ← PC + 1

T_2: D_0,…, D_1← Decode IR (12-14), AR ← IR(0 −11), 1 ← IR (15)

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T_0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T_1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At time T_2 , the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. Note that SC is incremented after each clock pulse to produce the sequence T_0, T_1, and T_2..

Figure 3.4 shows how the first two register transfer statements are implemented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to achieve the following connection.:

1. Place the content of PC onto the bus by making the bus selection inputs $S_2S_1S_0$ equal to 010.
2. Transfer the content of the bus to AR y enabling the LD input of AR.

The next clock transition initiates the transfer form PC to AR since $T_0 = 1$. In order to implement the second statement

T_1: IR ←M[AR], PC ← PC + 1

it is necessary to use timing signal T_1 to provide the following connections in the bus system.

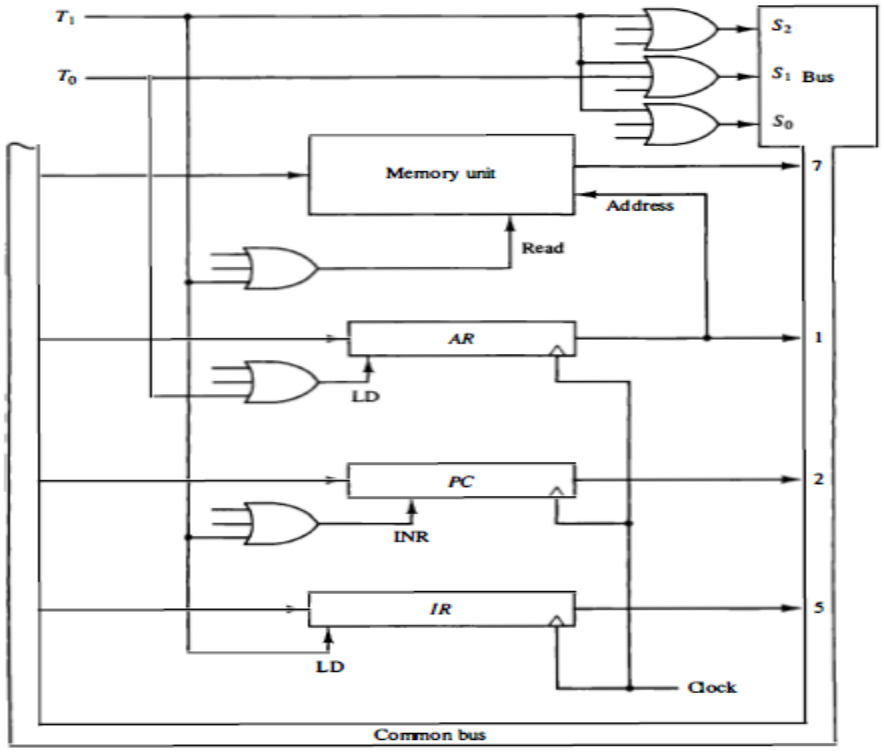


Figure: Register transfers for the fetch phase.

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0 = 111$.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

The next clock transition initiates the read and increment operations since $T_1 = 1$.

Figure 3.4 duplicates a portion of the bus system and shows how T_0 and T_1 are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

DETERMINE THE TYPE OF INSTRUCTION

The timing signal that is active after the decoding is T_3. During time T_3, the control MODULE determines the type of instruction that was just read from memory. The flowchart of Fig. 3.5 presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. We determine that if $D_1 = 1$, the instruction must be a register-reference or input-output type. If $D_7 = 0$, the operation code must be one of the other seven values 000 through 110, specifying memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If $D_7 = 0$ and I = 1, we have a memory-reference instruction with an indirect address. It is then necessary to read the effective address from memory. The micro operation for the indirect address condition can be symbolized by the register transfer statement

$AR \leftarrow M [AR]$

Initially, AR holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock

transition associated with timing signal T_3. This can be symbolized as follows:

D_1 = 0	I = 1	’3:	AR← M [AR]
0	0	Γ3;	ıg
1	0	Γ3:	te a register-reference instruction
1	1	’3:	te an input-output instruction

When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when D’$_7$ T_3 = 1, so that the execution of the memory-reference instruction can be continued with timing variable T_4. a register -reference or input-output instruction can be executed with the clock associated with timing signal T_3 . After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with T_0 = 1.

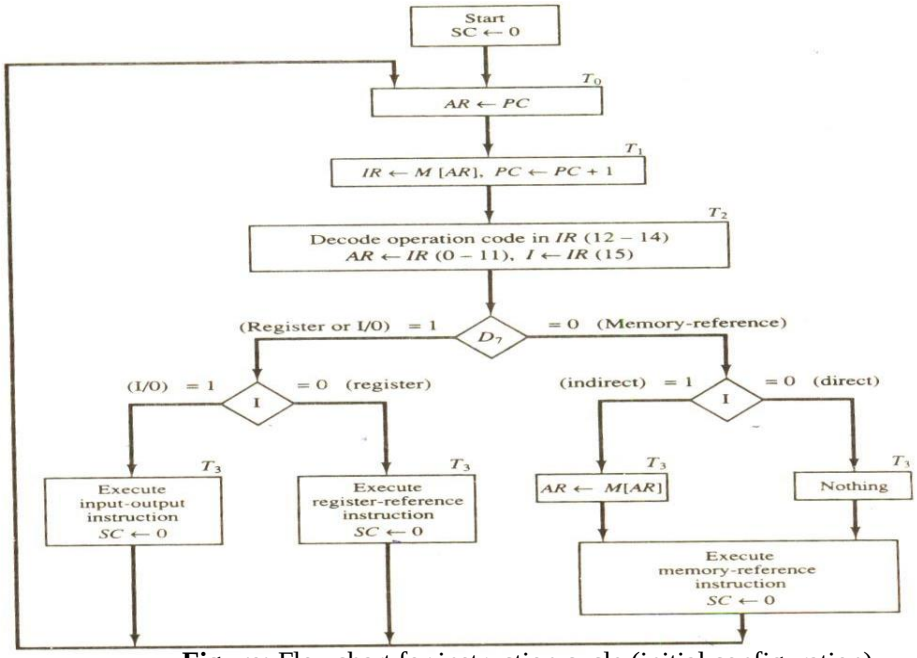


Figure: Flowchart for instruction cycle (initial configuration).

Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement SC ← SC + 1, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement Sc← 0.

The register transfers needed for the execution of the register-reference instructions are presented in this section. The memory-reference instructions are explained in the next section.

REGISTER-REFERENCE INSTRUCTIONS

Register-reference instructions are recognized by the control when D_7 = 1 and I = 0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR (0-11). They were also transferred to AR during time T_2.

The control functions and microoperations for the register -reference instructions are listed in Table 3.2. These instructions are executed with the clock transition associated with timing variable T_3. Each control function needs the Boolean relation D $_7$ I’ T_3, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in IR (0-11). By assigning the symbol B_i to bit i of IR, all control functions can be simply denoted by rB_i. For

example, the instruction CLA has the hexadecimal code 7800 (see Table 3.1), which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to I'. The next three bits constitute the operation code and are recognized from decoder output D_7. Bit 11 in IR is 1 and is recognized from B_{11}. The control function that initiates the microoperation for this instruction is D_7 I' T_3 B_{11} = rB_{11} . The execution of a register-reference instruction is completed at time T_3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0.

The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers. The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again (in addition, it is being incremented during the fetch phase at time T_1). The condition control statements must be recognized as part o the control conditions. The AC is positive when the sign bit in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero. The HLT instruction clears a start-stop flip-flops S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

TABLE 3.2 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

INPUT-OUTPUT AND INTERRUPT

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices. To demonstrate the most basic requirements for

input and output communication, we will use as an illustration a terminal MODULE with a keyboard and printer. Input-output organization is discussed further in Chap. 11.

INPUT-OUTPUT CONFIGURATION

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Fig. 3.8. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.

The input register INPR consists of eight bits and holds an alphanumeric input

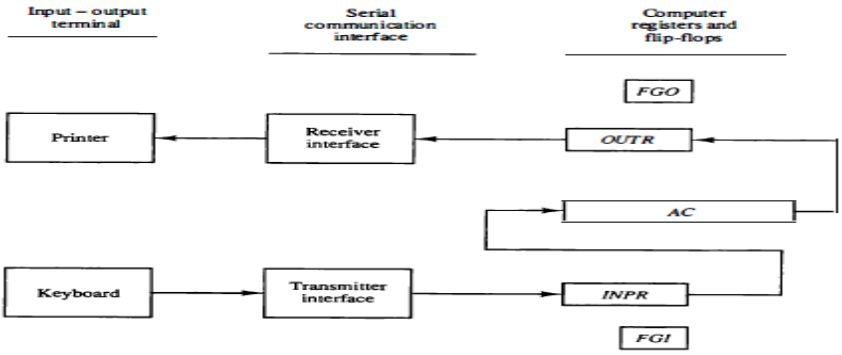


Figure: Input Output Configuration

Information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8- bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

The output register OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1. The computer checks the flag bit, if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to the computer does not load a new

character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

INPUT OUTPUT INSTRUCTIONS

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-output instructions have an operation code 1111 and are recognized by the control when D_7 = and I = 1. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed in Table 3.4. These instructions are executed with the clock transition associated with timing signal T_3 . Each control function needs a Boolean relation D-IT_3, which we designate for convenience by the symbol p. The control function is distinguished by one of the bits in IR. By assigning the symbol B_i to bit i of IR, all control functions can be

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Table 3.4 Input-Output Instructions

denoted by pB_1 for i = 6 though 11. The sequence counter SC is cleared to 0 when $p = D_7IT_3 = 1$.

The INP instruction transfers the input information from INPR into the eight low-order bits of AC and

also clears the input flag to 0. The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag to 0. The next two instructions in Table 3.4 check the status of the flags and cause a skip of the next instruction if the flag is 1. The instruction that is skipped will normally be a branch instruction to return and check the flag again. The branch instruction is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed. The last two instructions set and clear an interrupt enable flip-flop IEN. The purpose of IEN is explained in conjunction with the interrupt operation.

PROGRAM INTERRUPT

The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input-output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 μs. Assume that the input -output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip -flop IEN can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer.When IEN is set to 1 (with the ION instruction), the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. 5-13. An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts
address 1 into PC and clears IEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.

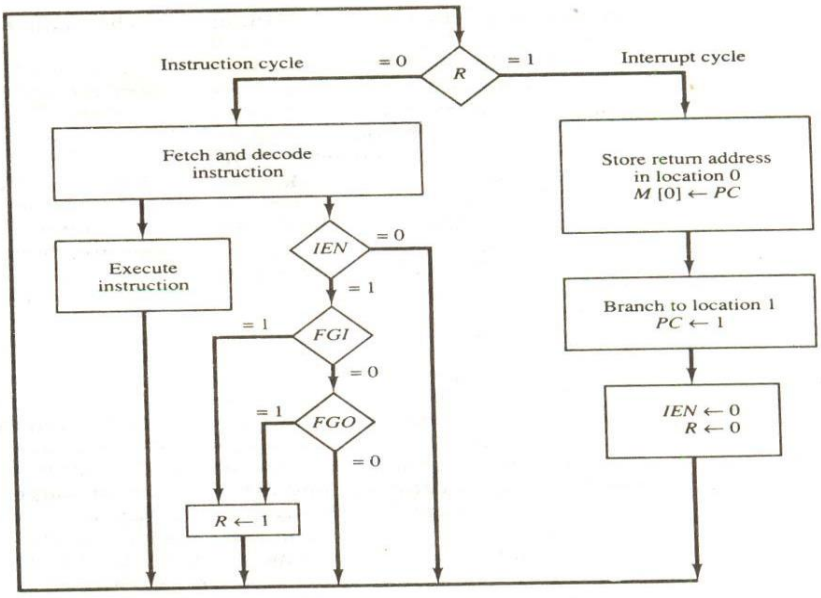


Figure : Flowchart for interrupt cycle

An example that shows what happens during the interrupt cycle is shown in Fig. 3.10. Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Fig. 3.10 (a).

When control reaches timing signal T_0 and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. One this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Fig. 3.10 (b).

The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program. After this instruction is read from memory during the fetch phase, control goes to the indirect phase (because I = 1) to read the effective address. The effective address is location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

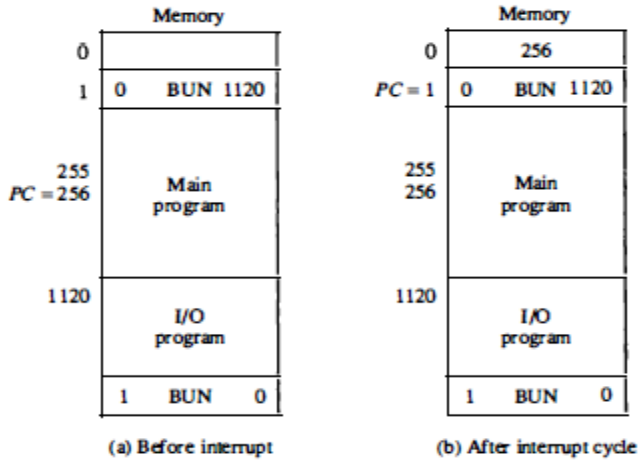


Figure : Demonstration of the interrupt cycle.

INTERRUPT CYCLE

We are now ready to list the register transfer statements for the interrupt cycle. The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if IEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T_0, T_1, or T_2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement :

$$T'_0T'_1T'_2 \text{ (IEN) (FGI + FGO): } R \leftarrow 1$$

The symbol + between FGI and FGO in the control function designates a logic OR operation. This is ANDed with IEN and $T'_0T'_1T'_2$.

We now modify the fetch and decode phases of the instruction cycle. Instead of using only timing signals T_0, T_1, and T_2 (as shown in Fig. 3.5) we will AND the three timing signals with R′ so that the fetch and decode phases will be recognized from the three control functions $R'T_0$, $R'T_1$, and $R'T_2$. The reason for this is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if R = 0. Otherwise, if R = I, the control will go through an interrupt cycle. The interrupt cycle stores the return address (available in PC) into memory location 0, branches to memory location 1, and clears IEN, R, and SC to 0. This can be done with the following sequence of micro operations.

$$\begin{aligned} RT_0&: \quad AR \leftarrow 0, \quad TR \leftarrow PC \\ RT_1&: \quad M[AR] \leftarrow TR, \quad PC \leftarrow 0 \\ RT_2&: \quad PC \leftarrow PC + 1, \quad IEN \leftarrow 0, \quad R \leftarrow 0, \quad SC \leftarrow 0 \end{aligned}$$

During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR. With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increments PC to 1, clears IEN and R, and control goes back to T_0 by clearing SC to 0. The beginning of the next instruction cycle has the condition $R' T_0$ and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

ADDRESSING MODES

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution in dependent on the addressing mode of the instruction. The addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.
3. The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the computer. The control MODULE of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory
2. Decode the instruction.
3. Execute the instruction.

There is one register in the computer called the program counter of PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing

mode of the instruction
and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

An example of an instruction format with a distinct addressing mode field is shown in Fig. 5 -6. The operation code specified the operation to be performed. The mode field is sued to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

Implied Mode: In this mode the operands are specified implicitly in the definition of theinstruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator

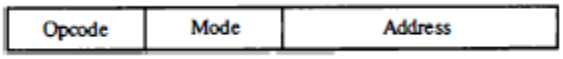


Figure: Instruction format with mode field

register is implied in the definition of the instruction. In fact, all register reference instructions that sue an accumulator are implied-mode instructions.

Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode : In this mode the operand is specified in the instruction itself. Inother words, an immediate-mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

Register Mode: In this mode the operands are in registers that reside within the CPU.The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.

Register Indirect Mode : In this mode the instruction specifies a register in the CPUwhose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address fo the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction sues fewer bits to select a register than would have been required to specify a memory address directly.

Auto increment or Auto decrement Mode: This is similar to the register indirect modeexcept that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction. However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control MODULE in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the instruction. The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.

The effective address is the address of the operand in a computational-type instruction. It is the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in previous chapter.

Direct Address Mode: In this mode the effective address is equal to the address part ofthe instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode: In this mode the address field of the instruction gives theaddress where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

Relative Address Mode: In this mode the content of the program counter is added to theaddress part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction. To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826 + 24 = 850. This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

Indexed Addressing Mode: In this mode the content of an index register is added to theaddress part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stores in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

Base Register Addressing Mode: In this mode the content of a base register is added tothe address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7. The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500. The first word of the instruction specifies the operation codeand mode, and the second word specifies the address part. PC has the value200 for fetching this instruction. The content of processor register R 1 is 400,and the content of an index register XR is 100. AC receives the operand afterthe instruction is executed. The figure lists a few pertinent addresses andshows the memory content at each of these addresses.

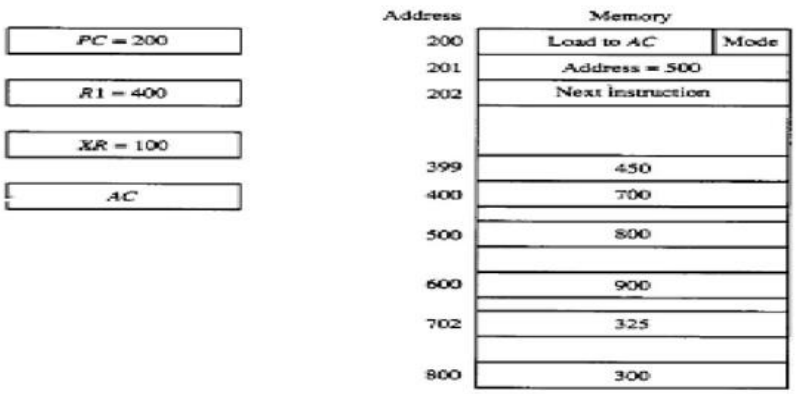


Figure 8-7 Numerical example for addressing modes.

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC. In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201 .) In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode the effective address is 500 + 202 =702 and the operand is 325. (Note that the value in PC after the fetch phase and during the execute phase is 202.) In the index mode the effective address is XR + 500 = 100 + 500 = 600 and the operand is 900. In the register mode the operand is in R 1 and 400 is loaded into AC. (There is no effective address in this case.) In the register indirect mode the effective address is 400, equal to the content of R 1 and the operand loaded into AC is 700. The auto increment mode is the same as the register indirect mode except that R 1 is incremented to 401 after the execution of the instruction. The auto decrement mode decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450. Table 8-4 lists the values of the effective address and theoperand loaded into AC for the nine addressing modes.

datatransferandmanipulation

Most computer instructions can be classified into three categories:
1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer instructions cause transfer of data from one location to another without changing the binary information content. Data manipulation instructions are those that perform arithmetic, logic, and shift operations. Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer. The instruction set of a particular computer determines the register transfer operations and control decisions that are available to the user.

Data Transfer Instructions

Data transfer instructions move data from one place in the computer to another without changing the data content. The most

common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. Table 8-5 gives a list of eight data transfer instructions used in many computers. Accompanying each instruction is a mnemonic symbol. It must be realized that different computers use different mnemonics for the same instruction name. The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator. The store instruction designates a transfer from a processor register into memory. The move instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers

between CPU registers and memory or between two memory words. The exchange instruction swaps information between two registers or a register and a memory word. The input and output instructions transfer data among processor registers and input or output terminals. The push and pop instructions
transfer data between processor registers and a memory stack.

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

TABLE :Typical Data Transfer Instructions

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

TABLE: Eight Addressing Modes for the Load Instruction

Table shows the recommended assembly language convention and theactual transfer accomplished in each case. ADR stands for an address, NBR isa number or operand, X is an index register, Rl is a processor register, and ACis the accumulator register. The @ character symbolizes an indirect address. The $ character before an address makes the address relative to the programcounter PC. The # character precedes the operand in an immediate-modeinstruction. An indexed mode instruction is recognized by a register that isplaced in parentheses after the symbolic address. The register mode is symbolized by giving the name of a processor register. In the register indirect mode,the name of the register that holds the memory address is enclosed in parentheses. The auto increment mode is distinguished from the register indirectmode by placing a plus after the parenthesized register. The auto decrement mode would use a minus instead.

Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

Arithmetic instructions

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide

instructions for all four operations. Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by means of
software subroutines. The four basic arithmetic operations are sufficient for formulating solutions to scientific problems when expressed in terms of numerical analysis methods.
A list of typical arithmetic instructions is given in Table 8-7. The increment instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all 1' s when incremented produces a result of all 0' s. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all D's, when decremented, produces a number with all 1's. The add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed ·to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

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

TABLE: Typical Arithmetic Instructions

Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The logical instructions considereach bit of the operand separately and treat it as a Boolean variable.
Some typical logical and bit manipulation instructions are listed in Table 8-8. The clear instruction causes the specified operand to be replaced by 0's. The complement instruction produces the 1's complement by inverting all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. There are three bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented.

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

TABLE: Typical Logical and Bit Manipulation Instructions

Shift Instructions

Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

Table 8-9 lists four types of shift instructions. The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. Arithmetic shifts usually conform with the rules for signed-2' s complement numbers. These rules are given in Sec. 4-6. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction. For this reason many computers do not provide a distinct arithmetic shift-left instruction when the logical shift-left instruction is already available.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus a rotate-left through carry

instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

TABLE: Typical Shift Instructions

Some typical program control instructions are listed in Table. The branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes. The branch is usually a one-address instruction. It is written in assembly languageas BR ADR, where ADR is a symbolic name for an address. When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR. Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

TABLE: Typical Program Control Instructions

from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence. The skip instruction does not need an address field and is therefore a zero-address instruction. A conditional skip instruction will skip the next

instruction if the condition is met. This is accomplished by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction. Thus a skip-branch pair of instructions causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met. The

call and return instructions are used in conjunction with subroutines. Their performance and implementation are discussed later in this section. The compare and test instructions do not change the program sequence directly.
They are listed in Table 8-10 because of their application in setting conditions for subsequent conditional branch instructions. The compare instruction per forms a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation. Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands. The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition. The generation of these status bits will be discussed first and then we will show how they are used in conditional branch instructions.

Status Bit Conditions

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits. Figure 8-8 shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C. S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit C (carry) is set to 1 if the end carry C8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest-order bit F, is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all O's. !t is cleared to 0 otherwise. In other Words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0
Otherwise. This is the condition for an

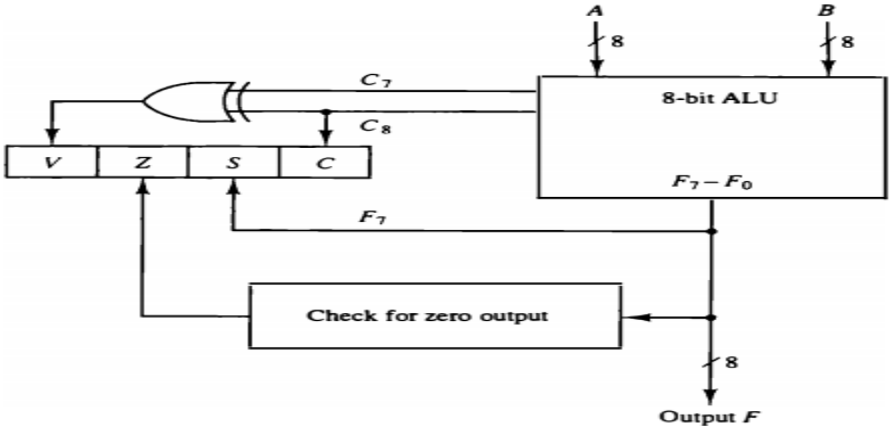


Figure: Status register bits.

Overflow when negative numbers are in 2's complement (see Sec. 3-3).For the 8-bit ALU, V = 1 if the output is greater than + 127 or less than - 128.
The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that A = B. This is so because x E!) x = 0, and the exclusive-OR of two equal operands gives an all-0' s result which sets the Z bit. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let A = 101xll00, where x is the bit to

be checked. The AND operation of A with B = 00010000 produces a result OOOxOOOO. If x = 0, the Z Status bit is set, but if x = 1, the Z bit is cleared since the result is not zero. The AND operation can be generated with the TEST instruction listed in Table 8-10 if the original content of A must be preserved.

Conditional Branch Instructions

Table 8-1 1 gives a list of the most common branch instructions. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

TABLE : Conditional Branch Instructions

Inserted to define the 0 state. Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions. The zero status bit is used for testing if the result of an ALU operation is equal to zero or not. The carry bit is used to check if there is a carry out of the most significant bit position of the ALU. It is also used in conjunction with
the rotate instructions to check the bit shifted from the end position of a register into the carry position. The sign bit reflects the state of the most significant bit of the output from the ALU. S = 0 denotes a positive sign and S = 1, a negative sign. Therefore, a branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1. It must be realized, however, that these two conditional branch instructions can be used to check the value of the most significant bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations done on signed numbers in 2's complement representation.

Subroutine Call and Return

A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save address. A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two operations: (1) the address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return, and (2) control is transferred to the beginning of the subroutine. The last instruction of every subroutine, commonly called return from subroutine, transfers the return address from the temporary location into on the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.
The advantage of usinga stack for the return address is that when a succession of subroutines is called,the sequential return addresses can be pushed into the stack. The return from subroutine instruction causes the stack to pop and the contents of the

top of the stack are transferred to the program counter. In this way, the return is always to the program that last called a subroutine. A subroutine call is implemented with the following micro operations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$ effective address	Transfer control to the subroutine

If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the micro operations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

By using a subroutine stack, all return addresses are automatically stored by the hardware in one MODULE. The programmer does not have to be concerned or remember where the return address was stored.
A recursive subroutine is a subroutine that calls itself. If only one register or memory location is used to store the return address, and the recursive subroutine calls itself, it destroys the previous return address.

Types of Interrupts

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:
1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input-output (l/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data, elapsed time of an event, or power failure. Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation. Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. These error conditions usually occur as a result of a premature termination of the instruction execution. The service program that processes the internal interrupt determines the corrective measure to be taken.
The difference between internal and external interrupts is that the internal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event. Internal interrupts are synchronous with the program while external interrupts are asynchronous.

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction.

Addition and subtraction:

Addition and Subtraction with Signed-Magnitude Data The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations. The procedure for adding or subtracting two signed binary numbers with paper and pencil is simple and straightforward. A review of this procedure will be helpful for deriving the hardware algorithm.

We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed.

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words inside parentheses should be used for the subtraction algorithm):

Addition (subtraction) algorithm: when the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different (identical), compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to

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

TABLE: Addition and Subtraction of Signed-Magnitude Numbers

be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Hardware Implementation:

Figure 10-1 shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops As and Bs . Subtraction is done by adding A to the 2' s complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers. The add-overflow flip-flop AVF holds the overflow bit when A and B are added. The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm. The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B depending on the state of the mode control M. The complementer consists of exclusive-OR gates and the parallel adder consists of full-adder circuits as shown in Fig. 4-7 in Chap. 4. The M signal is also applied to the input carry of the adder. When M = 0, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum A + B. When M a I, the l's complement of B is applied to the adder, the input carry is I, and output S = A + H + 1. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction, A - B.

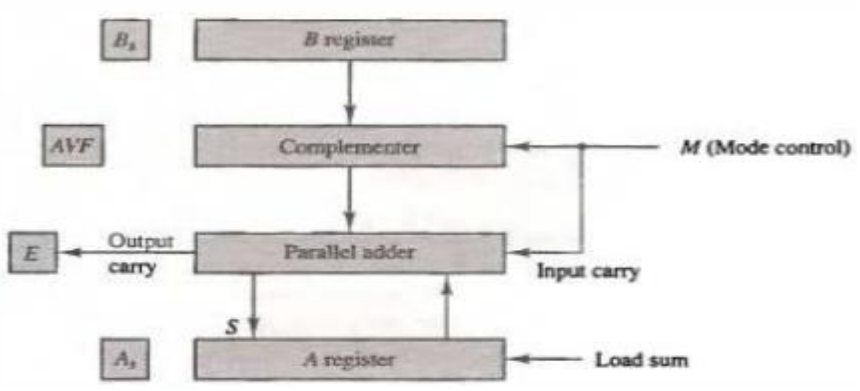


Figure: Hardware for a signed-magnitude addition and subtraction.

The flowchart for the hardware algorithm is presented in Fig. 10-2. The two signs As and Bs are compared by an exclusive-OR gate. If the output of the gate is 0, the signs are identical; if it is 1, the signs are different For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added. The magnitudes are added with a micro operation EA <--A + B. where EA is a register that combines E: and A. The carry in E: after the addition constitutes an overflow if it is equal to 1. The value of E: is transferred into the add-over flow flip-flop AVF.

The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1 in E indicates that A >=B and the number in A is the correct result. If this number is zero.The sign As must be made positive to avoid a negative zero. A 0 in E: indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one microoperation A ←A'+ 1 . However, we assume that the A register has circuits for microoperations complement and increment, so the 2' s complement is obtained from these two microoperations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in As is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement As to obtain the correct sign. The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.

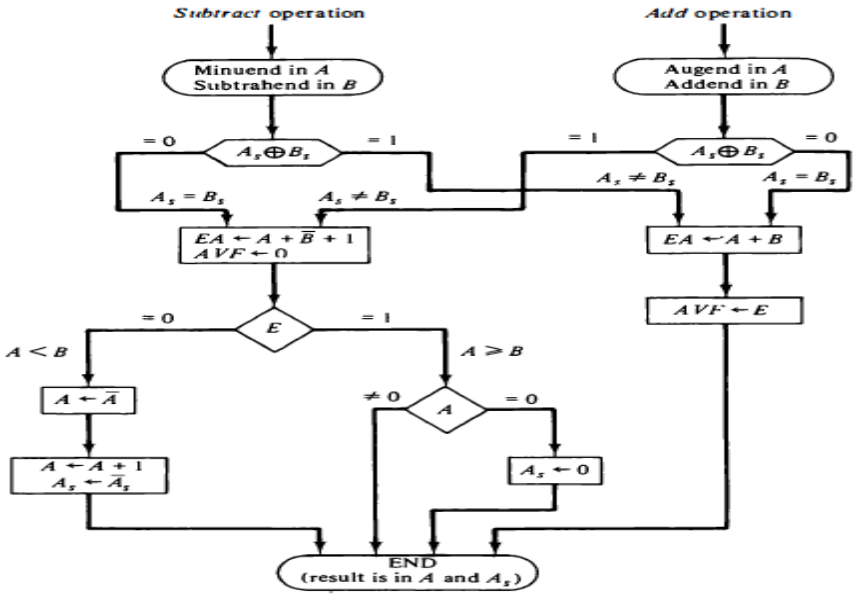


Figure: Flowchart for add and subtract operations.

Addition and Subtraction with Signed-2's Complement Data

The register configuration for the hardware implementation i s shown in Fig. 10-3. This is the same configuration as in Fig. 10-1 except that the sign bits are not separated from the rest of the registers. We name the A register AC (accumulator) and the B register BR . The leftmost bit in AC and BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded**.**

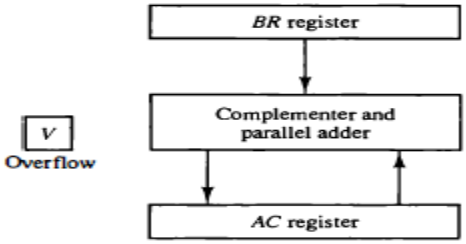


Figure: Hardware for signed-2's complement addition and subtraction.

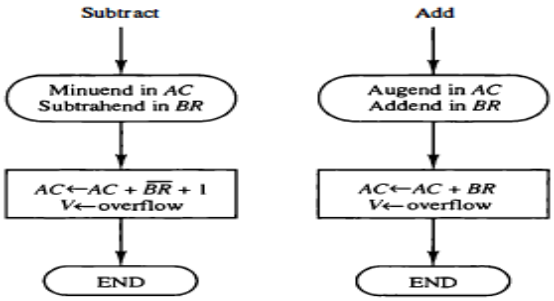


Figure : Algorithm for adding and subtracting numbers in signed-2'scomplement representation.

Multiplication Algorithms:

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example.

```
 23      10111    Multiplicand
 19    × 10011    Multiplier
       ------
       10111
      10111
     00000    +
    00000
   10111
  ---------
437 110110101    Product
```

The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, thenumbers are added and their sum forms the product.The sign of the product i s determined from the signs of the multiplicand and multiplier.

Hardware Implementation for Signed-Magnitude Data

The hardware for multiplication consists of the equipment shown in Fig. 10-1 plus two more registers. These registers together with registers A and B are shown in Fig. 10-5. The multiplier is stored in the Q register and its sign in Qs. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.
Initially, the multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig. 10-5. The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E.

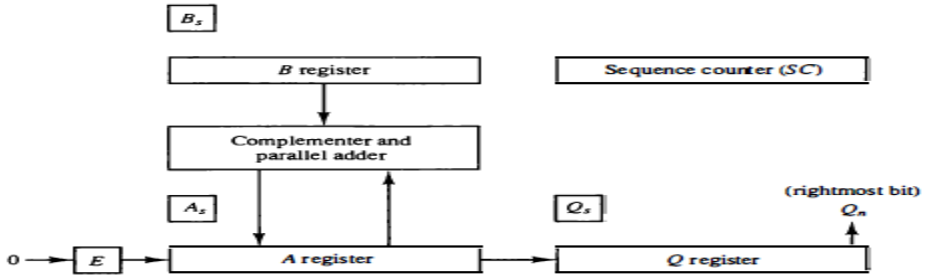


Figure: Hardware for multiply operation.

After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by Qn, will hold the bit of the multiplier, which must be inspected next.

Hardware Algorithm

Figure 10-6 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to registers from a memory MODULE that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n - 1 bits.

After the initialization, the low-order bit of the multiplier in Qn is tested. If it is a 1, the multiplicand in B is added to the present partial product in A . If it is a 0 , nothing i s done. Register EAQ i s then shifted once to the right to form the new partial product The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

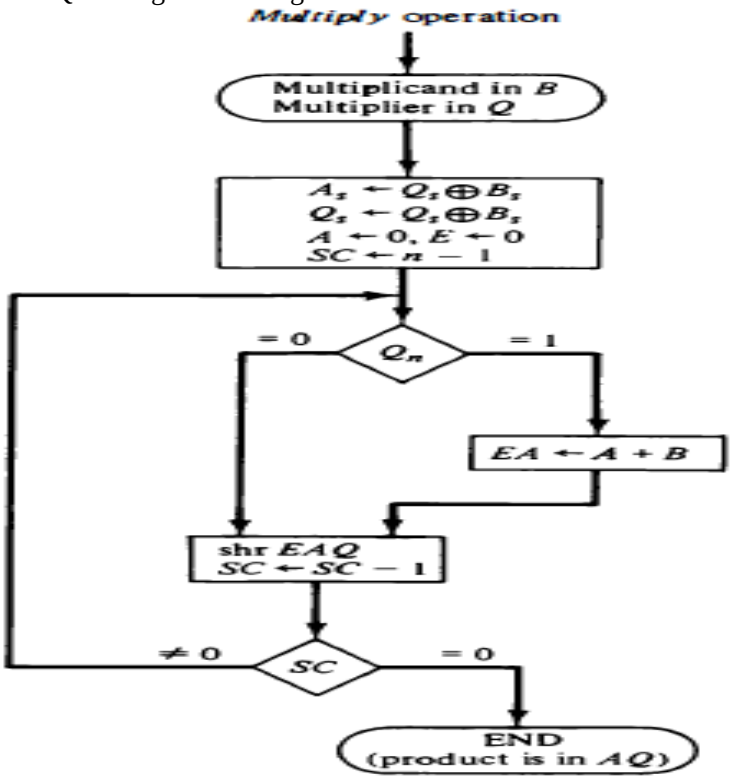


Figure: Flowchart for multiply operation.

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

TABLE :Numerical Example for Binary Multiplier

Booth Multiplication Algorithm

The hardware implementation of Booth algorithm requires the register configuration shown in Fig. This is similar to Fig. except that the sign bits are not separated from the rest of the registers. To show this difference, we rename registers A, B, and Q, as AC, BR, and QR, respectively. Qn designates the least significant bit of the multiplier in register QR. An extra flip-flop Qn+1, is appended to QR to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Fig. AC and the appended.

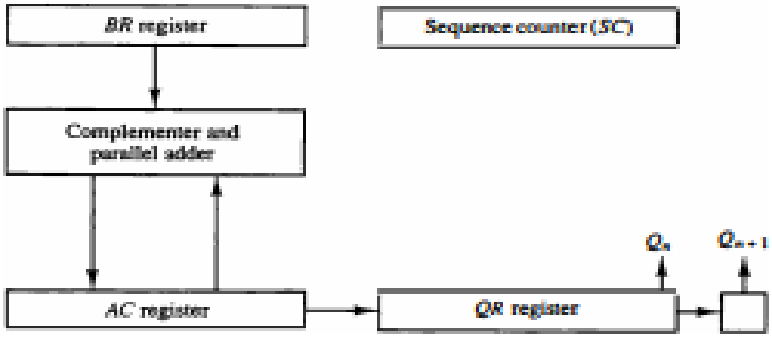


Figure- Hardware for Booth algorithm.

bit Qn+1 are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Qn and Qn+1 are inspected. If the two bits are equal to 10, it means that the first 1 in a string of 1' s has been encountered. This requires a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01, it means that the first 0 in a string of 0' s has been encountered. This requires the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including bit Qn+1). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged (see Sec. 4-6). The sequence counter is decremented and the computational loop is repeated n times.

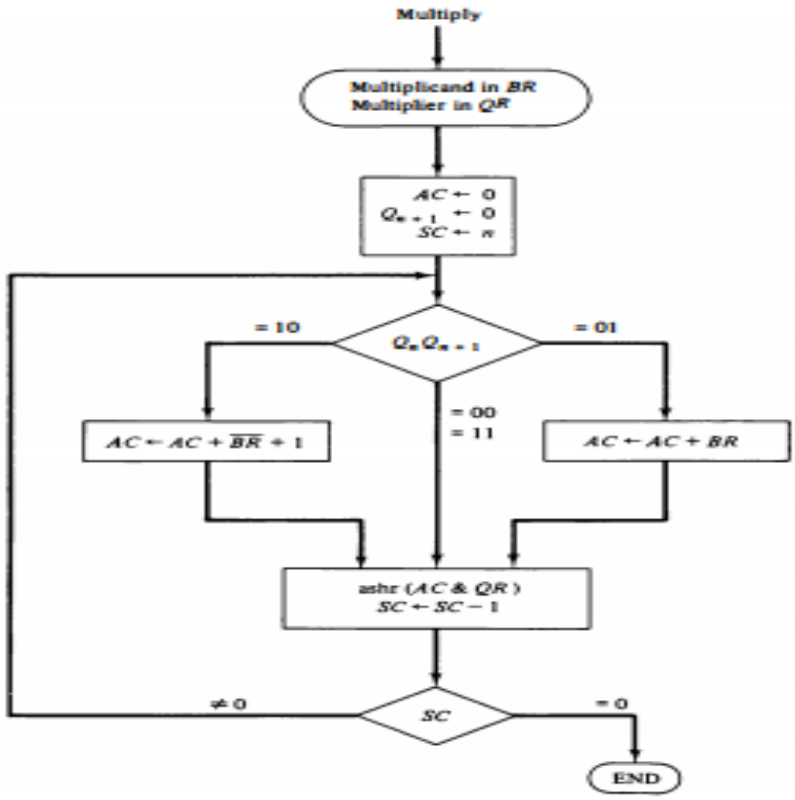


Figure- Booth algorithm for multiplication of signed 2's complement numbers.

A numerical example of Booth algorithm is shown in Table for n = 5. It shows the step-by-step multiplication of (-9) x (- 13) = + 117. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive. The final value of Qn+1 is the original sign bit of the multiplier and should not be taken as part of the product.

Q_nQ_{n+1}	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

Array Multiplier

To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 10-9. The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3 c2 c1 c0• The first partial product is formed by multiplying a0 by b1 b0• The multiplication of two bits such as a0 and b0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a1 by b1 b0 and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need j x k AND gates and (j - 1) k-bit adders to produce a product of j + k bits.

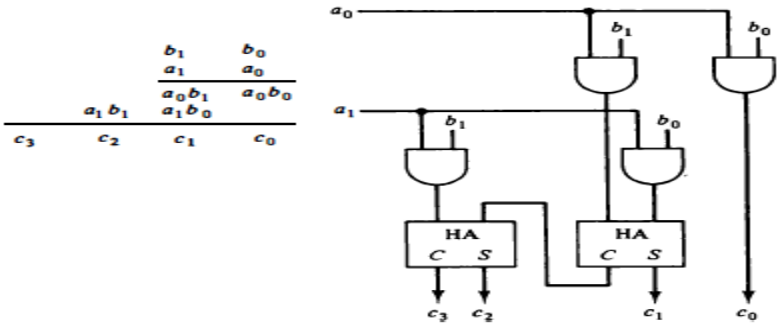


Figure : 2-bit by 2-bit array multiplier.

As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be

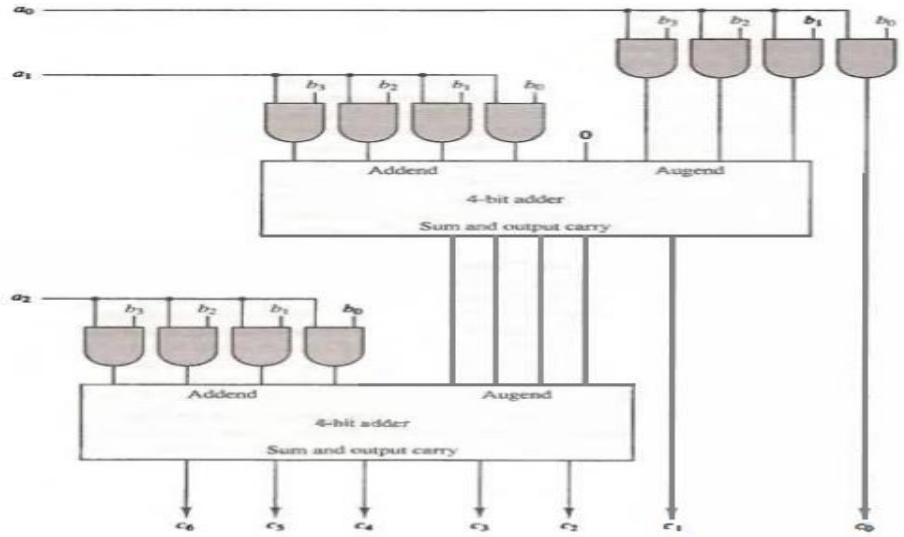


Fig:4-bit by 3-bit array multiplier

Represented by b3 b2 b1 b0 and the multiplier by a2 a1 a0. Since k=4 and j=3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Fig.

Division Algorithms:

Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations. Binary division is simpler than decimal division because the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is illustrated by a numerical example in Fig. 10-1 1. The divisor B consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bit number is smaller than B, we try again by taking the six most significant bits of A and compare this number with B. The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The divisor is then shifted once to the right and subtracted from the dividend. The difference is called a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder.

If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

Hardware Implementation for Signed-Magnitude Data

Instead of shifting the divisor to the right, the divided, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position. Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end-carry.

Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E lost. The numerical example is repeated in Fig. 10-12 to clarify the

Divisor:		
$B = 10001$	11010	Quotient = Q
	)0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Figure :Example of binary division.

Divisor $B = 10001$, $\overline{B} + 1 = 01111$

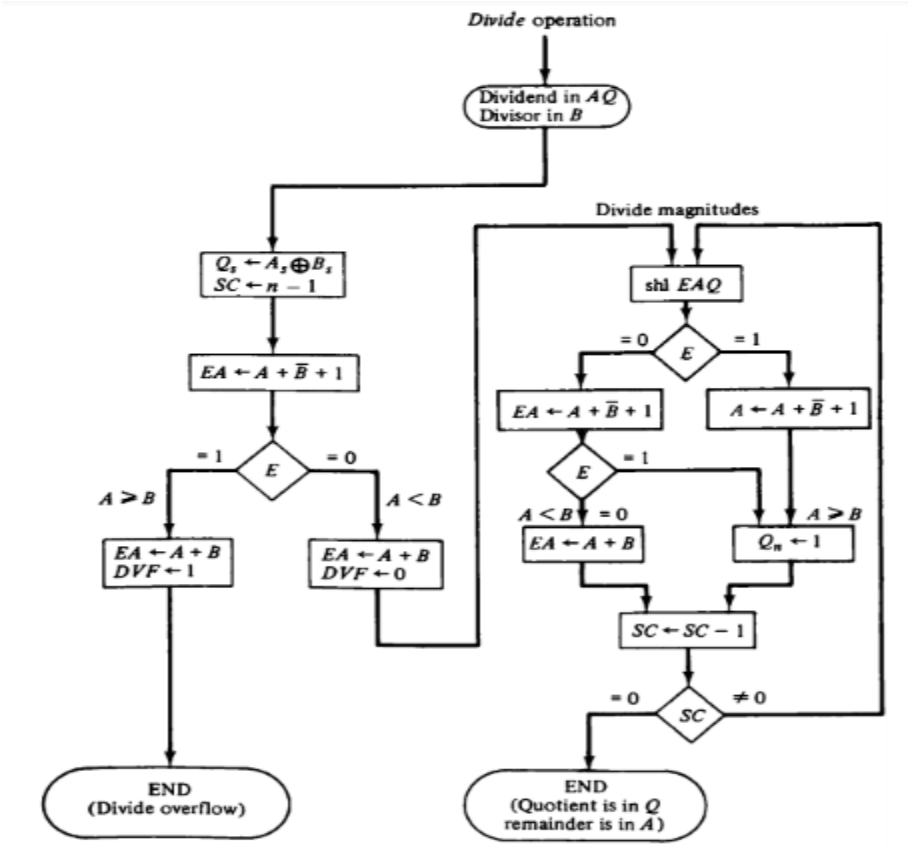
	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure: Example of binary division with digital hardware.

Proposed division process. The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E. If E = 1, it signifies that A 2: B. A quotient bit 1 is inserted into Q, and the partial remainder is shifted to the left to repeat the process. If E = 0, it signifies that A < B so the quotient in Q, remains a 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed. Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A.

Hardware Algorithm

The hardware divide algorithm is shown in the flowchart of Fig. 10-13. The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Q, to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory MODULE that has



words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n -1 bits.A divide-overflow condition is tested by subtracting the divisor in B fromhalf of the bits of the dividend stored in A. If A ;, B, the divide-overflowflip-flop DVF is set and the operation is terminated prematurely. If A < B,no divide overflow occurs so the value of the dividend is restored by addingB to A.The division of the magnitudes starts by shifting the dividend in AQ tothe left with the high-order bit shifted into E. If the bit shifted into E is 1, weknow that EA > B because EA consists of a 1 followed by n -1 bits while Bconsists of only n -1 bits.

Floating-Point Arithmetic Operations

Register Configuration

The register configuration for floating-point operations is quite similar to the layout for fixed-point operations. As a general rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled.

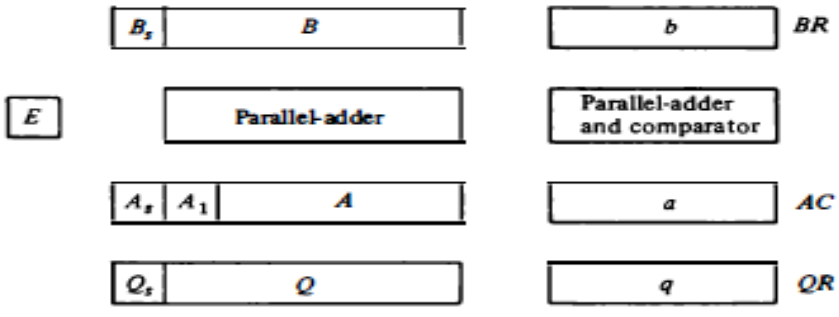


Figure :Registers for floating-point arithmetic operations.

The register organization for floating-point operations is shown in Fig. there are three registers, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lower case letter symbol.

It is assumed that each floating-point number has a mantissa in signed magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in A, and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. The diagram shows explicitly the most significant bit of A, labeled by A1 •The bit in this position must be a 1 for the number to be normalized. Note that the symbol AC represents the entire register, that is, the concatenation of As, A, and a.

Similarly, register B R i s subdivided into Bs, B, and b, and QR into Qs, Q, and q. A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E. A separate parallel-adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity. It is assumed that the floating-point numbers are so large that the chance of an exponent overflow is very remote, and for this reason the exponent overflow will be neglected. The exponents are also connected to a magnitude comparator that provides three binary outputs to indicate their relative magnitude.

Addition and Subtraction

During addition or subtraction, the two floating-point operands are in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. 10-15. If BR is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer

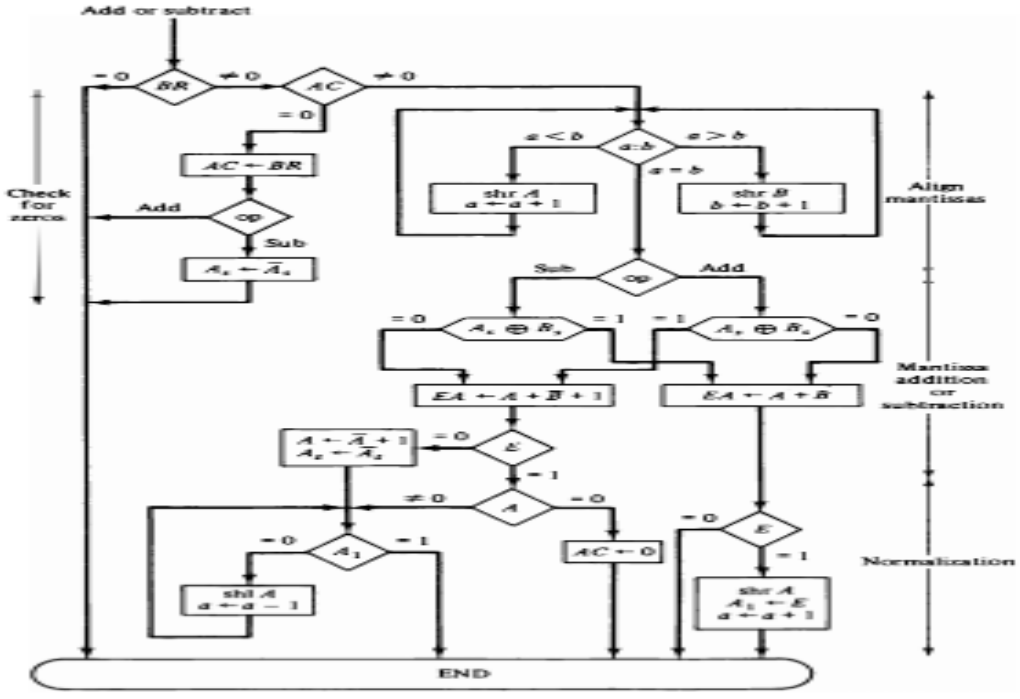


Figure : Addition and subtraction of floating-point numbers.

the content of BR into AC and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas. The magnitude comparator attached to exponents a and b provides three outputs that indicate their relative magnitude. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.

The addition and subtraction of the two mantissas is identical to the fixed-point addition and subtraction algorithm presented in Fig. 10-2. The magnitude part is added or subtracted depending on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E is equal to 1, the bit is transferred into A1 and all other bits of A are shifted right. The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position. If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number in the AC is made zero. Otherwise, the mantissa must have at least one bit that is
equal to 1. The mantissa has an underflow if the most significant bit in position A1 is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A1 is checked again and the process is repeated until it is equal to 1. When A1 = 1, the mantissa is normalized and the operation is completed.

Multiplication

The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents.
The multiplication algorithm can be subdivided into four parts:
1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.

Steps 2 and 3 can be done simultaneously if separate adders are available for the mantissas and exponents.
The flowchart for floating-point multiplication is shown in Fig. 10-16. The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the AC is set to zero and the operation is

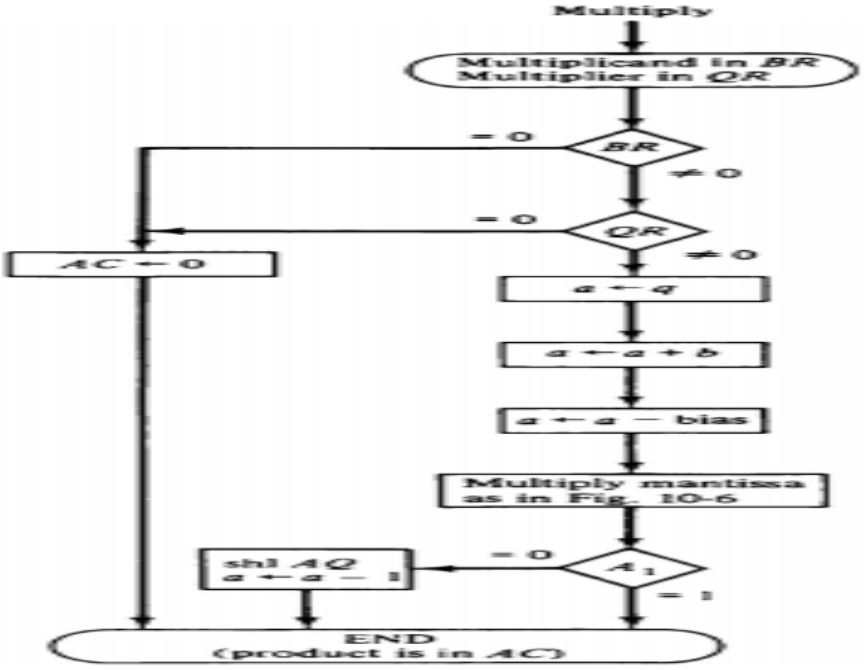


Figure: Multiplication of floating-point numbers.

terminated. If neither of the operands is equal to zero, the process continues with the exponent addition.

The exponent of the multiplier is in q and the adder is between exponents a and b. It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a. Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias. The correct biased exponent for the product is obtained by subtracting the bias number from the sum.

The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and Q. Overflow cannot occur during multiplication,so there is no need to check for it.The product may have an underflow, so the most significant bit in A ischecked. If it is a 1, the product is already normalized. If it is a 0, the mantissain AQ is shifted left and the exponent decremented. Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is0.1, so the smallest possible product is 0.01. Therefore, only one leading zeromay occur.

Although the low-order half of the mantissa is in Q, we do not use it forthe floating-point product. Only the value in the AC is taken as the product.

Division

Floating-point division requires that the exponents be subtracted and the mantissas divided. The mantissa division is done as in fixed-point except that the dividend has a single-precision mantissa that is placed in the AC. Remember that the mantissa dividend is a fraction and not an integer. For integer representation, a single-precision dividend must be placed in register Q and register A must be cleared. The zeros in A are to the left of the binary point and have no significance. In fraction representation, a single-precision dividend is placed in register A and register Q is cleared. The zeros in Q are to the right of the binary point and have no significance.

The check for divide-overflow is the same as in fixed-point representation. However, with floating-point numbers the divide-overflow imposes no problems. If the dividend is greater than or equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1. For normalized operands this is a sufficient operation to ensure that no mantissa divide dividend alignment overflow will occur. The operation above is referred to as a dividend alignment.

The division of two normalized floating-point numbers will always result in a normalized quotient provided that a dividend alignment is carried out before the division. Therefore, unlike the other operations, the quotient obtained after the division does not require normalization.

The division algorithm can be subdivided into five parts:
1. Check for zeros.
2. Initialize registers and evaluates the sign.
3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.

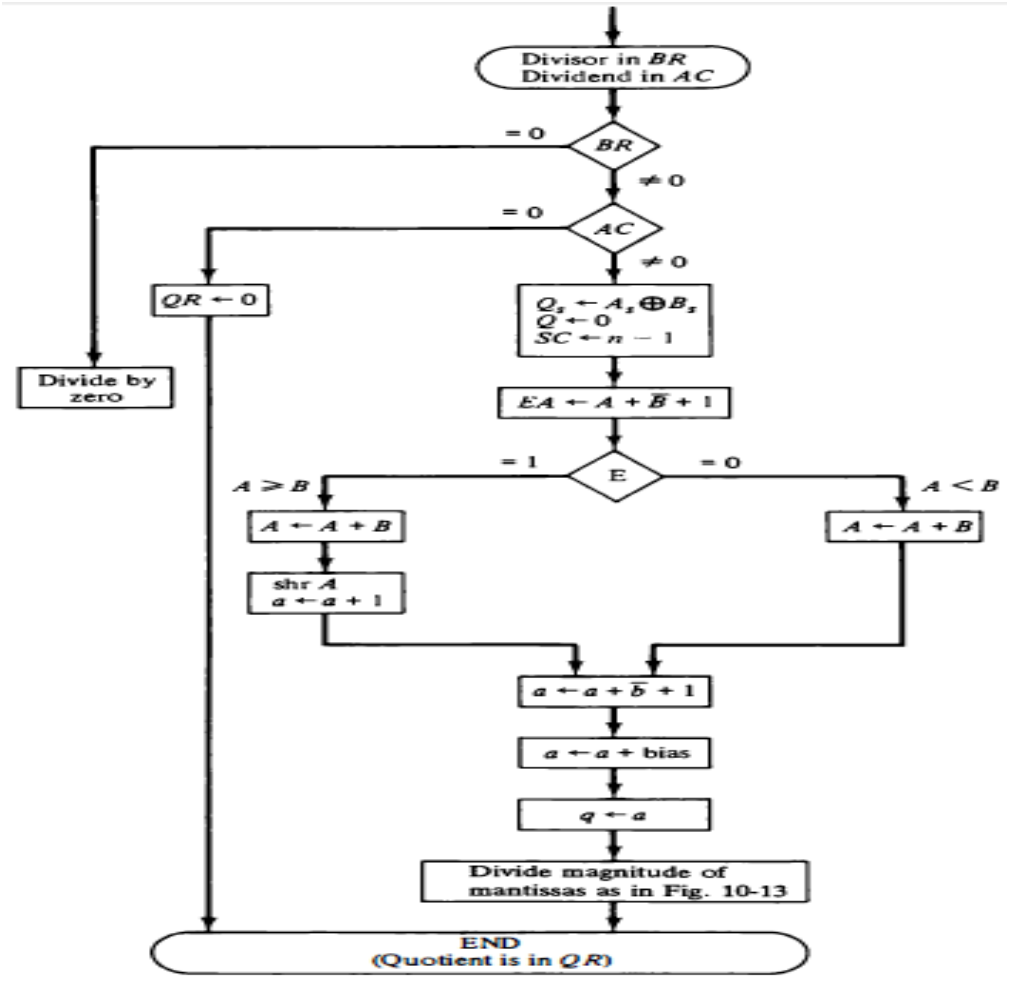
The flowchart for floating-point division is shown in Fig. 10-17. The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error message. An alternative procedure would be to set the quotient in QR to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative). If the dividend in AC is zero, the quotient in QR is made zero and the operation terminates.

If the operands are not zero, we proceed to determine the sign of the quotient and store it in Qs. The sign of the dividend in As, is left unchanged to be the sign of the remainder. The Q register is cleared and the sequence counter SC is set to a number equal to the number of bits in the quotient.

The dividend alignment is similar to the divide-overflow check in the fixed-point operation. The proper alignment requires that the fraction dividend be smaller than the divisor. The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor. If A>=B, it is necessary to shift A once

to the right and increment the dividend exponent. Since both operands are normalized, this alignment

ensures that A < B. Next, the divisor exponent is subtracted from the dividend exponent. Since both exponents were originally biased, the subtraction operation gives the difference without the bias. The bias is then added and the result transferred into q because the quotient is formed in QR.



The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q and the remainder in A. The floating-point quotient is already normalized and resides in QR. The exponent of the remainder should be the same as the exponent of the dividend.

The binary point for the remainder mantissa lies (n - 1) positions to the left of A1• The remainder can be converted to a normalized fraction by subtracting n - 1 from the dividend exponent and by shift and decrement until the bit in A1 is equal to 1. This is not shown in the flow chart and is left as an exercise.

Decimal Arithmetic MODULE:

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input-carry. Suppose that we apply two BCD digits to a 4-bit binary adder.

The adder will form the sum in binary and produce a result that may range from 0 to 19. These binary numbers are listed in Table 10-4 and are labeled by symbols K, Z8, z,, Z2, and Z1• K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit binary adder. The output sum of two decimal numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple

rule by which the binary number in the first column can be converted to the correct BCD digit

representation of the number in the second column. In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical

Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

TABLE: Derivation of BCD Adder

and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a no valid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required. One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum. If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. This second operation will automatically produce an output-carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until all decimal digits are added.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry K = 1. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z8• To distinguish them from binary 1000 and 1001 which also have a 1 in position z, we specify further that either z, or z, must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function.

$$C = K + Z8Z4 + Z8Z2$$

When C = 1, it is necessary to add 0110 to the binary sum and provide an output-carry for the next stage.

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as

shown in Fig. 10-18. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum.When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.

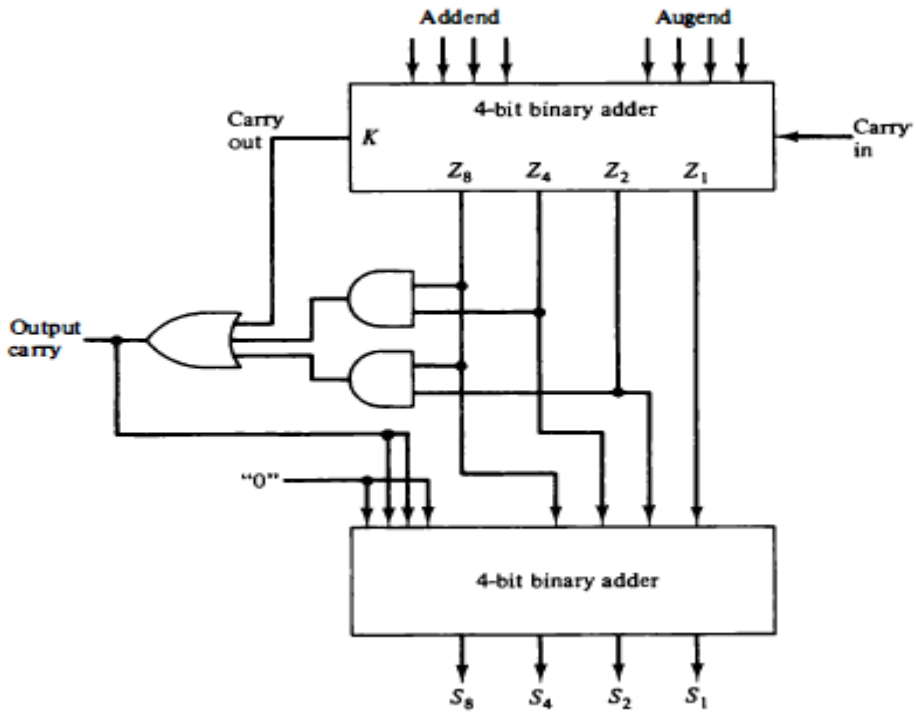


Figure: Block diagram of BCD adder.

MODULE-4
MEMORY ORGANIZATION

Memory organization: Memory hierarchy, main memory, auxiliary memory, associative memory, cache memory, virtual memory; **Input or output organization**: Input or output Interface, asynchronous data transfer, modes of transfer, priority interrupt, direct memory access.

MEMORY HIERARCHY

The memory MODULE is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory MODULE to accommodate all the programs used in a typical computer. Moreover, most computer users accumulate and continue to accumulate large amounts of data-processing software. Not all accumulated information is needed by the processor at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU. The memory MODULE that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.

The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of tall storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic. **Figure** illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

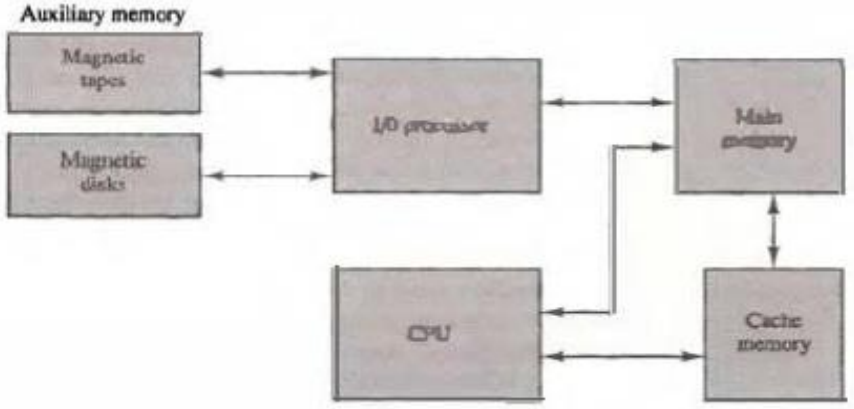


Figure - Memory hierarchy in a computer system.

A special very-high speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is
limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ in extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations by Making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

While the I/O processor manages data transfers between auxiliary memory and main memory,

the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100ns, while main memory access time may be 700ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from256 to 2048 words, while cache block size is typically from 1 to 16 words.

Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called multiprogramming, refers to the existence of two or more programs indifferent parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose that a program is being executed in the CPU and an I/O transfer is required. The CPU initiates the I/O processor to start executing the transfer. This leaves the CPU free to execute another program. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

MAIN MEMORY

The main memory is the central storage MODULE in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, static and dynamic. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to MODULE. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charges on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorted read and write cycles.

Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips. Originally, RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a

read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are

permanently resident in the computer and for tables of constants that do not change in value one the production of the

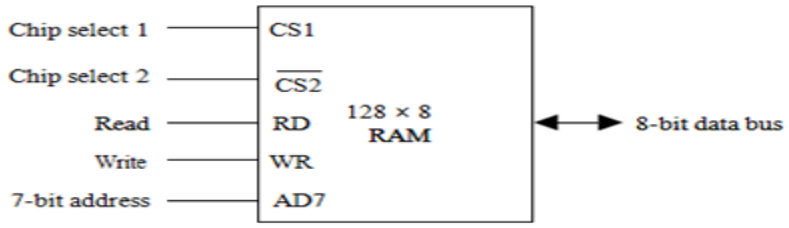
computer is completed.

Among other things, the ROM portion of main memory is needed for storing an initial program called a bootstrap loader. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer from general use.

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024 × 8 memory constructed with 128 × 8 RAM chips and 512 × 8 ROM chips.

RAM AND ROM CHIPS

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance. The block
diagram of a RAM chip is shown in Fig. The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit.



(a) Block diagram

CSI	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

Figure- Typical RAM Chip.

Address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.

The function table listed in Fig. (b) Specifies the operation of the RAM chip. The MODULEis in operation only when CSI = 1 and CS2 = 0. The bar on top of the second select variable indicates that this input in enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When SC1 = 1 and CS2 = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in below Fig. For the same-size chip, it is possible to have more bits of ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and CS2 = 0 for the MODULE to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the MODULE can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

MEMORY ADDRESS MAP

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established form knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a memory address map, is a pictorial representation of assigned address space for each chip in the system.

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips

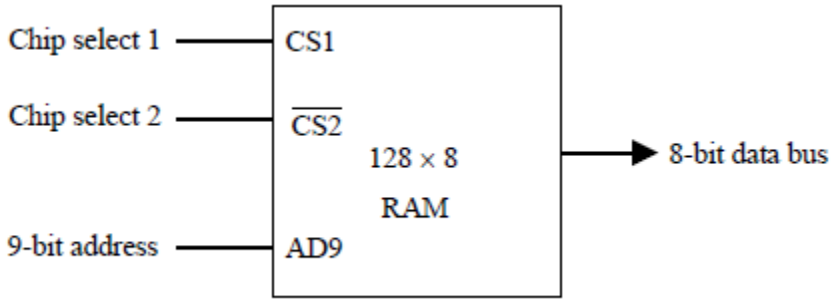


Figure-Typical ROM chip.

To be used are specified in Fig Typical RAM chip and Typical ROM chip. The memory address map for this configuration is shown in Table 7-1. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space from RAM equal to $2^9 = 512$ bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

The equivalent hexadecimal address for each chip is obtained forms the information under the address bus assignment. The address bus lines are subdivided into groups of four bits each so

TABLE-Memory Address Map for Microprocomputer

Component	Hexadecimal address	Address bus 10 9	8 7 6 5	4 3 2 1
RAM 1	0000—007F	0 0	0 x x x	x x x x
RAM 2	0080—00FF	0 0	1 x x x	x x x x
RAM 3	0100—017F	0 1	0 x x x	x x x x
RAM 4	0180—01FF	0 1	1 x x x	x x x x
ROM	0200—03FF	1 x	x x x x	x x x x

That each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. This x's represent a binary number that can range from an all-0's to an all-1's value.

MEMORY CONNECTION TO CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. The connection of memory chips to the CPU is shown in belowFig. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of Table 7-1. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2 × 4 decoder whose outputs go to the SCI input in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

The example just shown gives an indication of the interconnection complexity that can exist between memory chips and the CPU. The more chips that are connected, the more external decoders are required for selection among the chips. The designer must establish a memory map that assigns addresses to the various chips from which the required connections are determined.

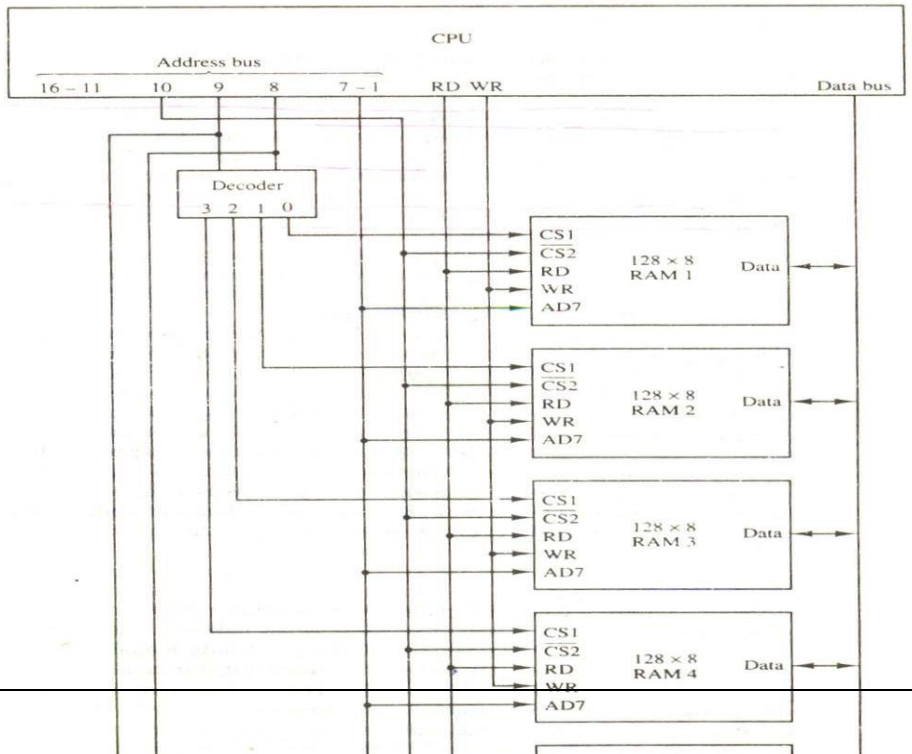


Figure -Memory connection to the CPU.

ASSOCIATIVE MEMORY

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. An account number may be searched in a file to determine the holder's name and account status. The established way to search a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory MODULE accessed by content is called an associative memory or content addressable memory (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given,. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locaters all words which match the specified content and marks them for reading.

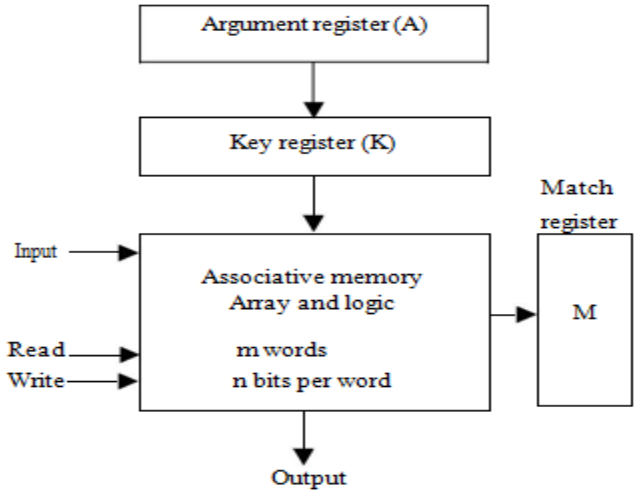
Because of its organization, the associative memory is uniquely suited to do parallel searches by data association. Moreover, searches can be done on an entire word or on a specific field within a word. An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

HARDWARE ORGANIZATION

The block diagram of an associative memory is shown in below Fig. It consists of a memory array and logic from words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared. Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.

Figure- Block diagram of associative memory



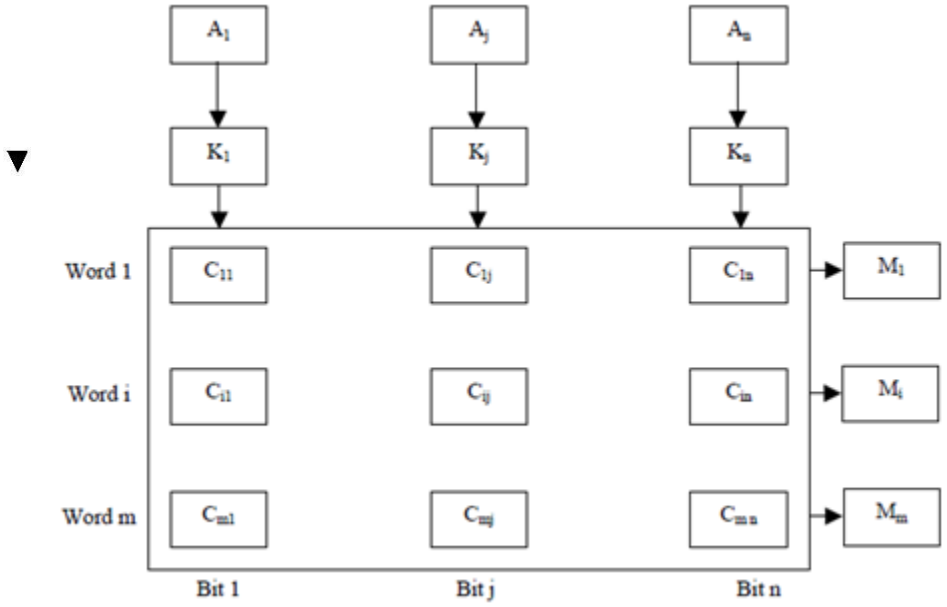
To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

A	101	111100	
K	111	000000	
Word 1	100	111100	no match
Word 2	101	000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in below Fig. The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that $K_j = 1$. This is done for all columns j = 1, 2,…,n. If a matchoccurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

Figure -Associative memory of m word, n cells per word



The internal organization of a typical cell C_{ij} is shown in Fig.

It consists of a flip-Flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i.

MATCH LOGIC

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we neglect the key bits and compare the argument in A with the bits stored in the cells of the words. Word i is equal to the argument in A if $A_j = F_{ij}$ for j = 1, 2,…, n. Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function

$x_j = A_jF_{ij} + A'_jF_{ij}'$

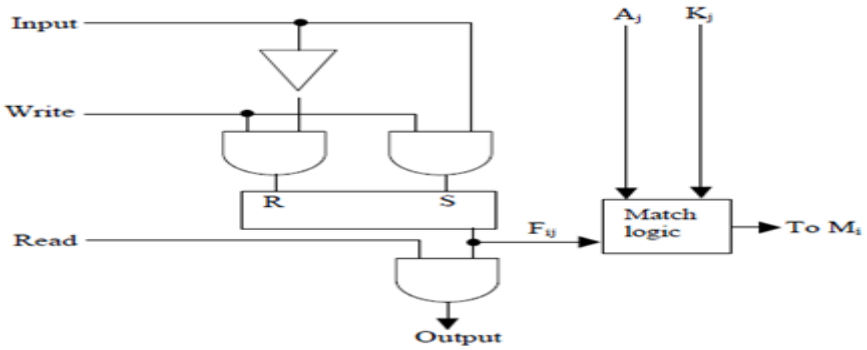
Wherex_j = 1 if the pair of bits in position j are equal; otherwise, x_j = 0.

For a word i to be equal to the argument in a we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is

$$M_i = x_1 x_2 x_3 \ldots x_n$$

And constitutes the AND operation of all pairs of matched bits in a word.

Figure : Onecell of associative memory.



We now include the key bit K_j in the comparison logic. The requirement is that if K_j = 0, the corresponding bits of A_j and F_{ij} need no comparison. Only when K_j = 1 must they be compared. This requirement is achieved by ORing each term with K_j' , thus:

$$x_j + K'_j = \begin{cases} x_j & \text{if } K_j = 1 \\ 1 & \text{if } K_j = 0 \end{cases}$$

1

j

When $K_j = 1$, we have $K_j' = 0$ and $x_j + 0 = x_j$. When $K_j = 0$, then $K_j' = 1$ $x_j + 1 = 1$. A term $(x_j + K_j')$ will be in the 1 state if its pair of bitsis not compared. This is necessary because each term isANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when $K_j = 1$.

The match logic for word i in an associative memory can now be expressed by the following Boolean function:

$M_i = (x_1 + K_j')(x_2 + K_j')(x_3 + K_j') \ldots (x_n + K_j')$

Each term in the expression will be equal to 1 if its corresponding $K_j' = 0$. if$K_j = 1$, the term willbe either 0 or 1 depending on the value of x_j. A match will occur and M_i will be equal to 1 if all terms are equal to 1.

If we substitute the original definition of x_j. The Boolean function above can be expressed as follows:

$M_i = \prod_{j=1} (A_j F_{ij} + A_j' F_j' + K_j')$

Where $\prod$ is a product symbol designating the AND operation of all n terms. We need m such functions, one for each word i = 1, 2, 3, …., m.

The circuit for catching one word is shown in below Fig. Each cell requires two AND gates and one OR gate. The inverters for A_j and K_j are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for M_i.M_i will be logic 1 if a catch occurs and 0 if no match occurs. Note that if the key register contains all 0's, output M_i will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

READ OPERATION

If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the catch register. It is then necessary to scan the bits of the match register on eat a time. The matched words are read in sequence by applying a read signal to each word line whose corresponding M_i bit is a 1.

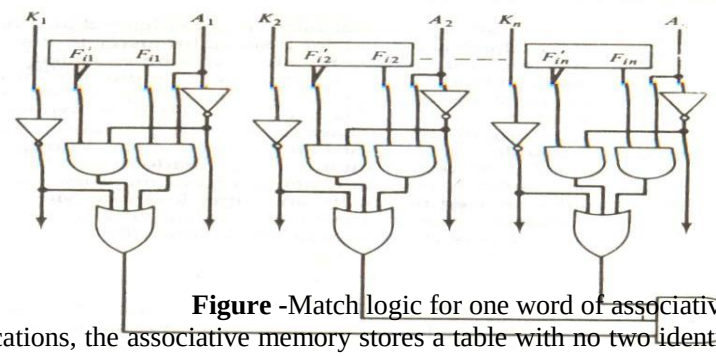


Figure -Match logic for one word of associative memory

In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output M_i directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude words having zero content, an all-zero output will indicate that no match occurred and that the searched item is not available in memory.

WRITE OPERATION

An associative memory must have a write capability for storing the information to be searched. Writing in an associative memory can take different forms, depending on the application. If the
entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of address lines can be reduced by the decoder to d lines, where $m = 2^d$.

If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a tag register, would have as many bits as there are words in the memory. For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when

deleted from memory can be cleared to all 0's if this value is used to specify an empty location. Moreover, the words that have a tag bit of 0 must be masked (together with the K_j bits) with the argument word so that only active words are compared.

CACHE MEMORY

Analysis of a large number of typical programs has shown that the references, to memory at any given interval of time tend to be confined within a few localized areas in memory. The phenomenon is known as the property of locality of reference. The reason for this property may be understood considering that a typical computer program flows in a straight-line fashion with program loops and subroutine calls encountered frequently. When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, its set of instructions is fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table-lookup procedures repeatedly refer to that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory. The result of all these observations is the locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively frequently.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between theCPU and main memory as illustrated in below Fig. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

The performance of cache memory is frequently measured in terms of a quantity called hit ratio. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If the word is not found in cache, it is in main memory and it counts as a miss. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The hit ratio is best measured experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

The average memory access time of a computer system can be improved considerably by use of a cache. If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory, the average access time is closer to the access time of the fast cache memory. For example, a computer with cache access time of 100 ns, a main memory access time of 1000 ns, and a hit ratio of 0.9 produces an average access time of 200 ns. This is a considerable improvement over a similar computer without a cache memory, whose access time is 1000 ns.

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is referred to as a mapping process. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

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

To helping the discussion of these three mapping procedures we will use a specific example of a memory organization as shown in below Fig. The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory.

The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU

accepts the 12 -bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

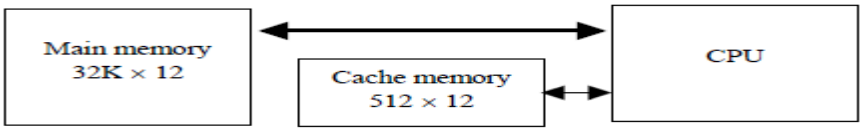
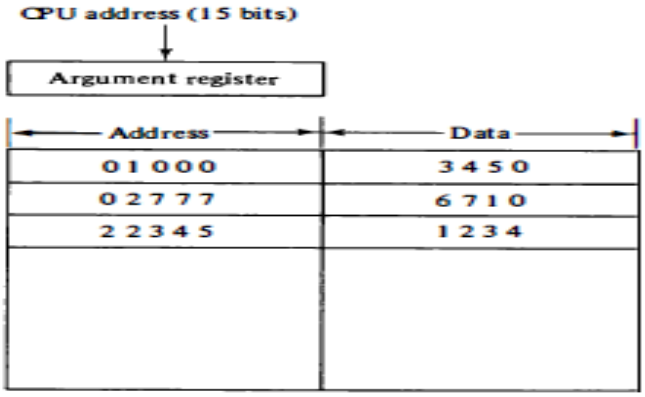


Figure - Example of cache memory

ASSOCIATIVE MAPPING

The fasters and most flexible cache organization use an associative memory. This organization is illustrated in below Fig. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read

Figure-Associative mapping cache (all numbers in octal)



And sent to the CPU. If no match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory. If the cache is full, an address−data pair must be displaced to make room for a pair that is needed and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

DIRECT MAPPING

Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Fig. The CPU address of 15 bits is divided into two fields. Thenine least significant bits constitute the index field and the remaining six bits from the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n-bit memory address is divided into two fields: k bits for the index field and n − k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache. The internal organization of the words in the cache memory is as shown in Fig. (b). Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.

The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value. The disadvantage of direct mapping is that the hit ratio

can droop considerably if two or more words whose

addresses have the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such words are relatively far apart in the address range (multiples of 512 locations in this example).

To see how the direct-mapping organization operates, consider the numerical example shown in Fig. The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is sued to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

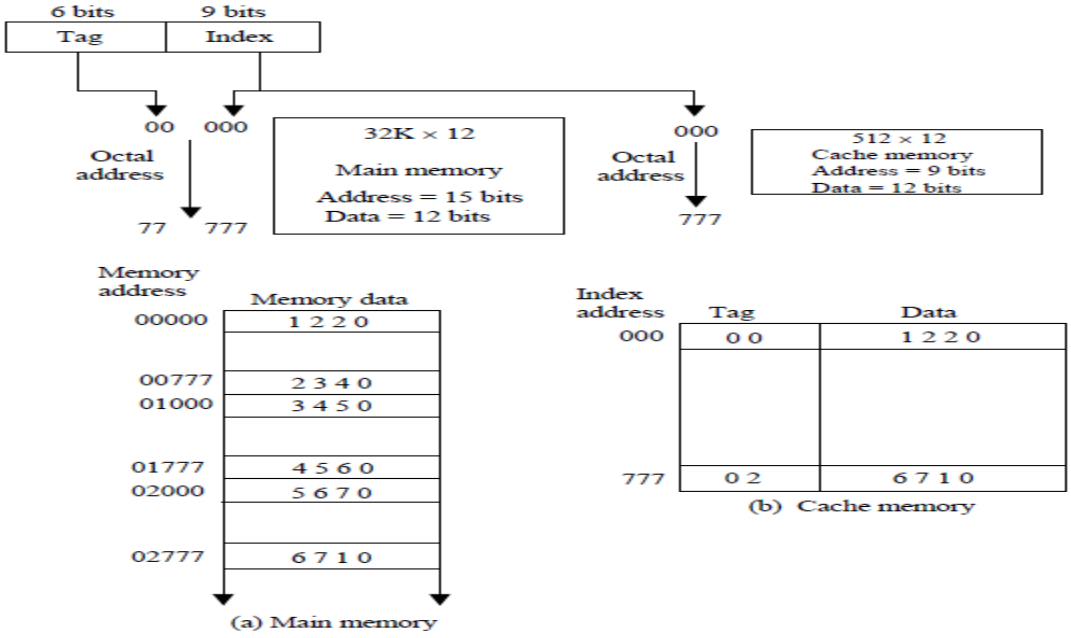


Fig-Direct mapping cache organization

The direct-mapping example just described uses a block size of one word. The same organization but using a block size of 8 words is shown in below Fig. The index

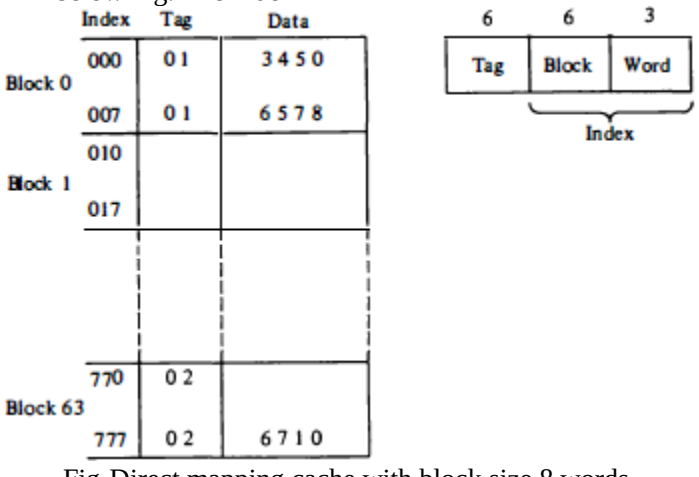


Fig-Direct mapping cache with block size 8 words

Field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 block

of 8 words each, since 64 × 8 = 512. The block number is specified with a 6-bit field and the word within the block is specified with a 3-bit field. The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.

SET-ASSOCIATIVE MAPPING

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time. A third type of cache organization, called set-associative mapping, is an improvement over the direct-mapping organization in that each word of cache can store two or more words ofmemory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set. An example of a set-associative cache organization for a set size of two is shown in Fig. Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is 2(6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512 × 36. It can accommodate 1024 words of main memory since each word of cache contains two data words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

Index	Tag	Data	Tag	Data
000	0 1	3 4 5 0	0 2	5 6 7 0
777		6 7 1 0	0 0	2 3 4 0

Figure- Two-way set-associative mapping cache.

The octal numbers listed in above Fig.are with reference to the main memory content illustrated in Fig.(a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a catch occurs. The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name “set-associative”. The hit ratio will improve as the set size increases because more words with the same index but different tag can reside in cache. However, an increase in the set size increases the number of bit s in words of cache and requires more complex comparison logic.

When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: random replacement, first-in, first out (FIFO), and least recently used (LRU). With the random replacement policy the control chooses one tag-data item for replacement at random. The FIFO procedure selects for replacement the item that has been in the set the longest. The LRU algorithm selects for replacement the item that has been least recently used by the CPU. Both FIFO and LRU can be implemented by adding a few extra bits in each word of cache.

WRITING INTO CACHE

An important aspect of cache organization is concerned with memory write requests. When the CPU finds a word in cache during read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

The simplest and most commonly used procedure is to up data main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the write-through method. This method has the advantage that main memory always contains the same data as the cache,. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at tall times so that an I/O device communicating through DMA would receive the most recent updated data.

The second procedure is called the write-back method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the words are removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory. Analytical results indicate that the number of memory writes in a typical program ranges between 10 and 30 percent of the total references to memory.

CACHE INITIALIZATION

One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, built in effect it contains some non-valid data. It is customary to include with each word in cache a valid bit to indicate whether or not the word contains valid data.

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The introduction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data. Thus the initialization condition has the effect of forcing misses from the cache until it fills with valid data.

VIRTUAL MEMORY

In a memory hierarchy system, programs and data are brought into main memory as they are needed by the CPU. Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

ADDRESS SPACE AND MEMORY SPACE

An address used by a programmer will be called a virtual address, and the set of such addresses the address space. An address in main memory is called a location or physical address. The set of such locations is called the memory space. Thus the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing. In most computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space in computers with virtual memory.

As an illustration, consider a computer with a main -memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since $32K = 2^{15}$. Suppose that the computer has available auxiliary memory for storing $2^{20} = 1024K$ words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

In a multiprogramming computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data re moved from auxiliary memory into main memory as shown in Fig. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

In a virtual memory system, programmers are told that they have the total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long. (Remember

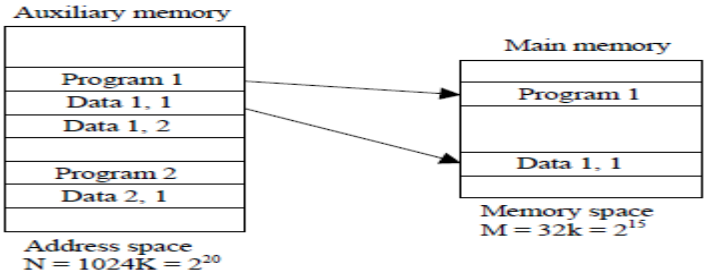


Fig-Relation between address and memory space in a virtual memory system

That for efficient transfers, auxiliary storage moves an entire record to the main memory). A table is then needed, as shown in Fig, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

The mapping table may be stored in a separate memory as shown in Fig. or in main memory. In the first case, an additional memory MODULE is required as well as one extra memory access time. In the second case, the table

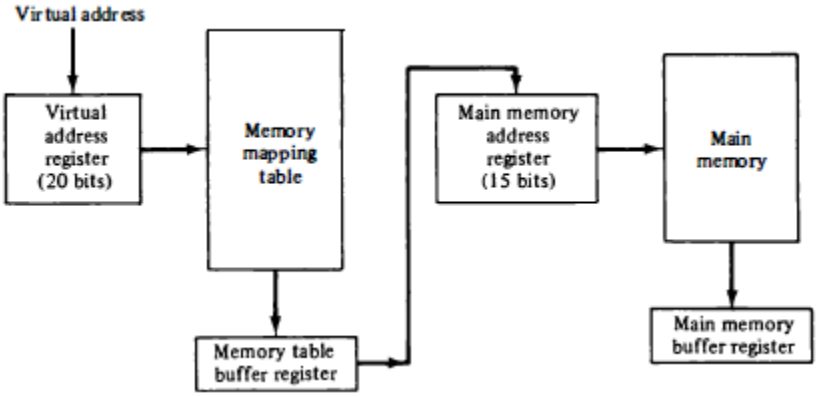


Figure - Memory table for mapping a virtual address.

Takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory as explained below.

ADDRESS MAPPING USING PAGES

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size. For example, if a page or block consists of 1K words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks. Although both a page and a block are split into groups of 1K words, a page refers to the organization of address space, while a block refers to the organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term “page frame” is sometimes used to denote a block.

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example of Fig, a virtual address has 13 bits. Since each page consists of $2^{10} = 1024$ words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the

line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

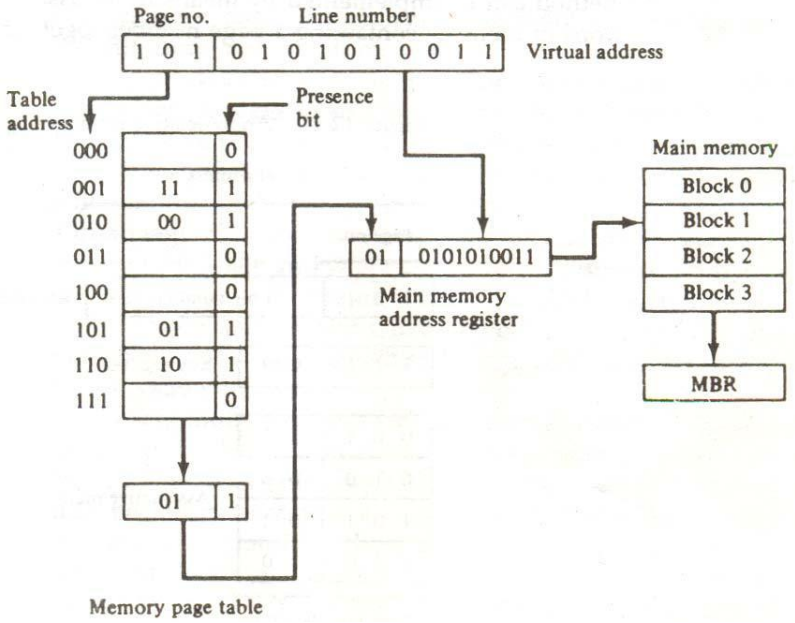


Figure-Memory table in a paged system.

The word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before resuming computation.

ASSOCIATIVE MEMORY PAGE TABLE

A random-access memory page table is inefficient with respect to storage utilization. In the example of below Fig. we observe that eight words of memory are needed, one for each page, but at least four words will always be marked empty because main memory cannot accommodate more than four blocks. In general, system with n pages and m blocks would require a memory-page table of n locations of which up to m blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding block number The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is

extracted.

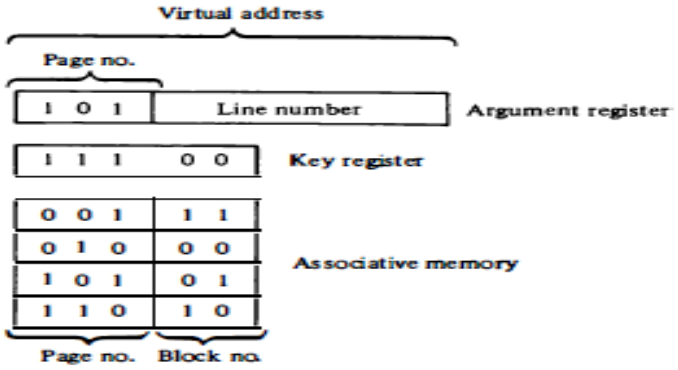


Figure -An associative memory page table.

Consider again the case of eight pages and four blocks as in the example of Fig. We replace the random access memory-page table with an associative memory of four words as shown in Fig. Each entry in the associative memory array consists of two fields. The first three bits specify a field from storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

INPUT-OUTPUT INTERFACE

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing MODULE. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are:

1 Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.
2 The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be need.
3 Data codes and formats in peripherals differ from the word format in the CPU and memory.
4 The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called interface MODULEs because they interface between the processor bus and the peripheral device. In addition, each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

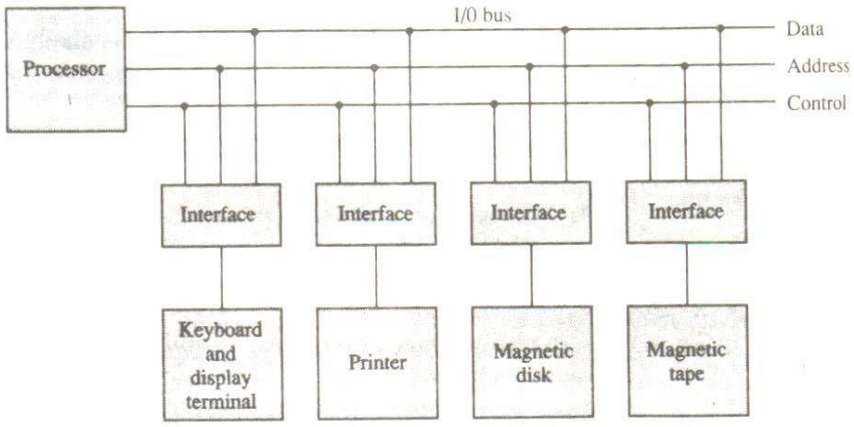
I/O BUS AND INTERFACE MODULES

A typical communication link between the processor and several peripherals is shown in Fig. 6-1 The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface MODULE. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device,

the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled their interface.

At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute it.



The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral MODULE. The interpretation of the command depends on the peripheral that the processor is addressing. There are four types of commands that an interface may receive. They are classified as control, status, status, data output, and data input.

A control command is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape MODULE may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

A status command is used to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

A data output command causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape MODULE. The computer starts the tape moving by issuing a control

command. The processor then monitors the status of the tape by means of a status command. When the tape is in the correct position, the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.

The data input command is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

I/O VERSUS MEMORY BUS

In addition to communicating with I/O, the processor must communicate with the memory MODULE. Like the I/O bus, the memory bus contains data, address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

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

In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing MODULE (CPU). The memory communicates with both the CPU and the IOP through a memory

bus. The IOP communicates also with the input and output devices through a separate I/O bus with its own address, data and control lines. The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory.

ISOLATED VERSUS MEMORY-MAPPED I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through

separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines. The I/O read and I/O write control lines are enabled during an I/O transfer. The memory read and memory write control lines are enabled during a memory transfer. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the isolated I/O method for assigning addresses in a common bus.

In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.

The isolated I/O method isolates memory word and not for an I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other

alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory-mapped I/O. The computer treats an interface
Register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduce the memory address range available.

In a memory-mapped I/O organization there are no specific inputs or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as

there is not also a memory word that responds to the same address.

Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input-output transfers or for memory transfers. The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers. In a typical computer, there are more memory-reference instructions than I/O instructions. With memory-mapped I/O all instructions that refer to memory are also available for I/O.

ASYNCHRONOUS DATA TRANSFER

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. Clock pulses are applied to all registers within a MODULE and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two MODULEs, such as a CPU and an I/O interface, are designed independently of each other. If the registers in the interface share a common clock with the CPU registers, the transfer between the two MODULEs is said to be synchronous. In most cases, the internal timing in each MODULE is independent from the other in that each uses its own private clock for internal registers. In that case, the two MODULEs are said to be asynchronous to each other. This approach is widely used in most computer systems.

Asynchronous data transfer between two independent MODULEs requires that control signals be transmitted between the communicating MODULEs to indicate the time at which data is being transmitted. One way of achieving this is by means of a strobe pulse supplied by one of the MODULEs to indicate to the other MODULE when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence ofdata in the bus. The MODULE receiving the data item responds with another

control signal to acknowledge receipt of the data. This type of agreement between two independent MODULEs is referred to as handshaking.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent MODULEs. In the general case we consider the transmitting MODULE as the source and the receiving MODULE as the destination. For example, the

CPU is the source MODULE during an output or a write transfer and it is the destination MODULE during an input or a read transfer. It is customary to specify the asynchronous transfer between two independent MODULEs by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in buses. The sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination MODULE.

STROBE CONTROL

The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination MODULE. Figure 6-3(a) shows a source-initiated transfer.

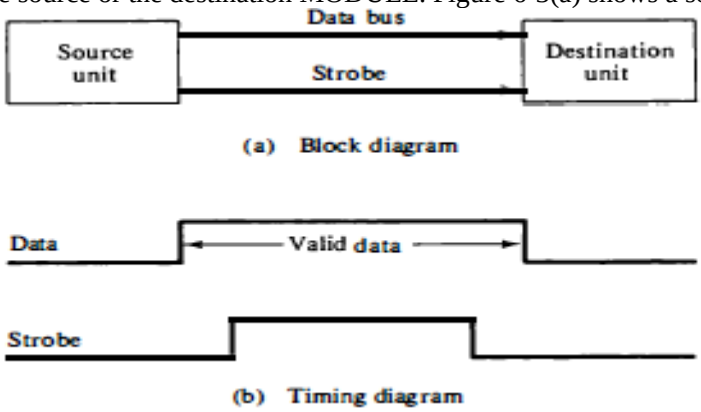


Figure 6-3 Source-initiated strobe for data transfer.

The data bus carries the binary information from source MODULE to the destination MODULE. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination MODULE when a valid data word is available in the bus.

As shown in the timing diagram of Fig. 6-3(b), the source MODULE first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse. The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination MODULE to receive the data. Often, the destination MODULE uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers. The source removes the data from the bus a brief period after it disables its strobe pulse. Actually, the source does not have to change the information in the data bus. The fact that the strobe signal is disabled indicates that the data bus does not contain valued data. New valid data will be available only after the strobe is enabled again.

Figure 6-4 shows a data transfer initiated by the destination MODULE. In this case the destination MODULE activates the strobe pulse, informing the source to provide the data. The source MODULE responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination MODULE to accept it. The falling edge of the Strobe pulse can be
used again to trigger a destination register. The destination MODULE then disables the strobe. The source removes the data from the bus after a predetermined time interval.

In many computers the strobe pulse is actually controlled by the clock pulses in the CPU. The CPU is always in control of the buses and informs the external MODULEs how to transfer data. For example, the strobe of Fig. 6-3 could be a memory -write control signal from the CPU to a memory MODULE. The source, being the CPU, places a word on the data bus and informs the memory MODULEs.

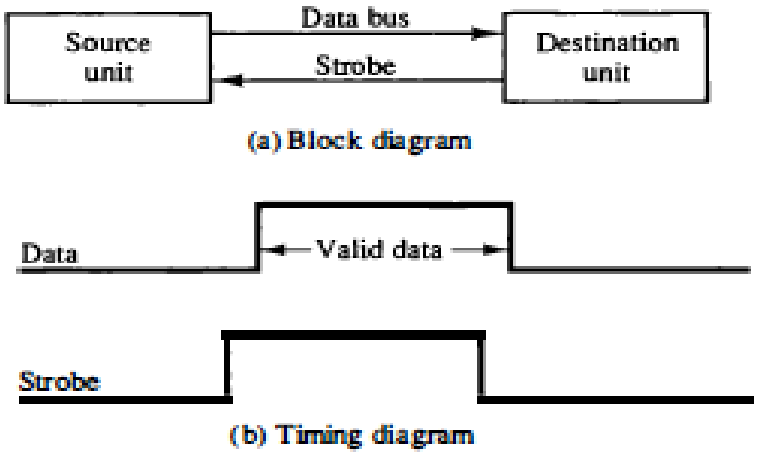


Figure 6-4 Destination-initiated strobe for data transfer.

This is the destination, that this is a write operation. Similarly, the strobe of fig. 6-4 could be a memory -read control signal from the CPU to a memory MODULE. The destination, the CPU, initiates the read operation to inform the memory, which is the source, to place a selected word into the data bus.

The transfer of data between the CPU and an interface MODULE is similar to the strobe transfer just described. Data transfer between an interface and an I/O device is commonly controlled by a set of handshaking lines.

HANDSHAKING

The disadvantage of the strobe method is that the source MODULE that initiates the transfer has no way of knowing whether the destination MODULE has actually received the data item that was placed in the bus. Similarly, a destination MODULE that initiates the transfer has no way of knowing whether the source MODULE has actually placed the data on the bus,. The handshake method solves this problem by introducing a second control signal that provides a reply to the MODULE that initiates the transfer. The basic principle of the two-write handshaking method of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source MODULE to inform the destination MODULE whether there are valued data in the bus. The other control line is in the other direction from the destination to the source. It is used by the destination MODULE to inform the source whether it can accept data. The sequence of control during the transfer depends on the MODULE that initiates the transfer.

Figure 6-5 shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source MODULE, and data accepted, generated by the destination MODULE. The timing diagram shows the exchange of signals between the two MODULEs. The sequence of events listed in part (c) shows the four possible states that the system can be at any given time. The source MODULE initiates the transfer by placing the data on the bus and enabling its data valid signal. The data accepted signal is activated by the destination MODULE after it accepts the data from the bus. The source MODULE then disables its data valid signal, which invalidates the data on the bus. The destination MODULE then disables its data accepted signal and the system goes into its initial state.The source dies not send the next data item until after the destination MODULE shows its readiness to accept new data by disabling its data accepted signal. This scheme allows arbitrary delays from one state to the next and permits each MODULE to respond at its own data transfer rate. The rate of transfer is determined by the slowest MODULE.

The destination -initiated transfer using handshaking lines is shown in Fig. 6-6. Note that the name of the signal generated by the destination MODULE has been changed to ready from data to reflect its new meaning. The source MODULE in this case dies not place data on the bus until after it receives the ready for data signal from the destination MODULE. From there on, the handshaking procedure follows the same pattern as in the source-initiated case. Note that the sequence of events in both cases would be identical if we consider the ready for data signal as the complement of data accepted.In fact, the only difference between the source-initiated and the destination-initiated transfer is in their choice of initial state.

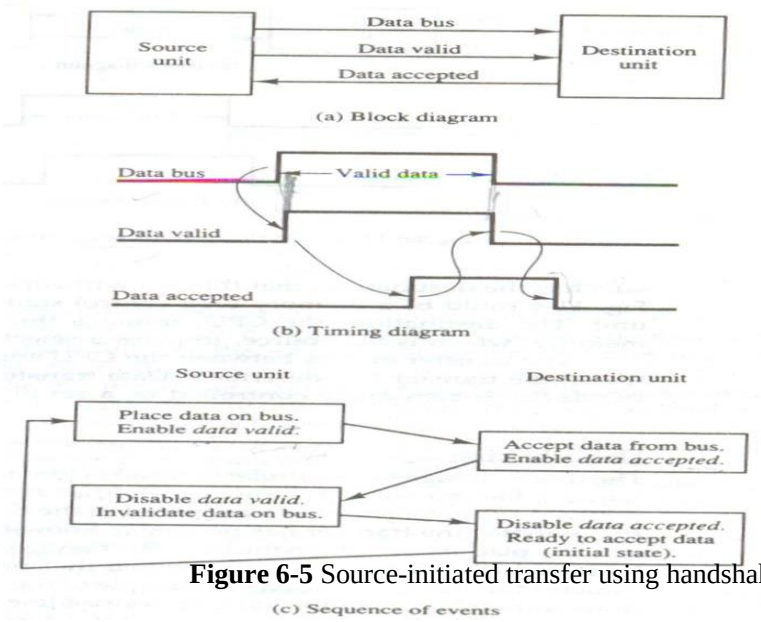


Figure 6-5 Source-initiated transfer using handshaking.

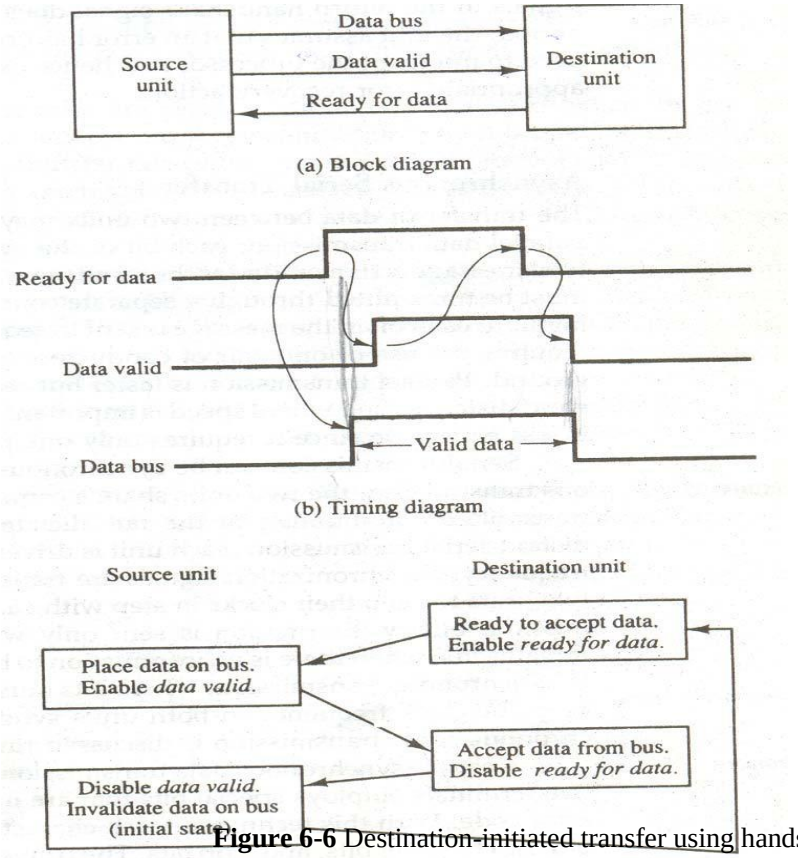


Figure 6-6 Destination-initiated transfer using handshaking.

The handshaking scheme provides a high degree of flexibility and reality because the successful completion of a data transfer relies on active participation by both MODULEs. If one MODULE is faulty, the data transfer will not be completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a predetermined time. The timeout is implemented by means of an internal clock that starts counting time when the MODULE enables one of its handshaking control signals. If the return handshake signal does not respond within a given time period, the MODULE assumes that an error has occurred. The timeout signal can

be used to interrupt the processor and hence execute a service routine that takes appropriates error recovery action.

ASYNCHRONOUS SERIAL TRANSFER

The transfer of data between two MODULEs may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time. This means that an n-bit message must be transmitted through in *n* separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and a common ground. Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors.

Serial transmission can be synchronous or asynchronous. In synchronous transmission, the two MODULEs share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses. In long-distant serial transmission, each MODULE is driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two MODULEs to keep their clocks in step with each other. In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is

in contrast to synchronous transmission, where bits must be transmitted continuously to deep the clock frequency in both MODULEs synchronized with each other.

Serial asynchronous data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1. An example of this format is shown in Fig. 6-7.

A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1When a character is not being sent, the line is kept in the 1-state.

2The initiation of a character transmission is detected from the start bit, which is always 0.

3The character bits always follow the start bit.

4After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

Using these rules, the receiver can detect the start bit when the line gives from 1 to 0. A clock in the receiver examines the line at proper bit times. The receiver knows the transfer rate of the bits and the number of character bits to accept. After the character bits are transmitted, one or two stop bits are sent. The stop bits are always in the 1-state and frame the end of the character to signify the idle or wait state.

At the end of the character the line is held at the 1-state for a period of at least one or two bit times so that both the transmitter and receiver can resynchronize. The length of time that the line stays in this state depends on the amount of time required for the equipment to resynchronize. Some older electromechanical terminals use two stop bits, but newer terminals use one stop bit. The line remains in the 1-state until another character is transmitted. The stop time ensures that a new character will not follow for one or two bit times.

As illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists of a start bit, eight information bits, of

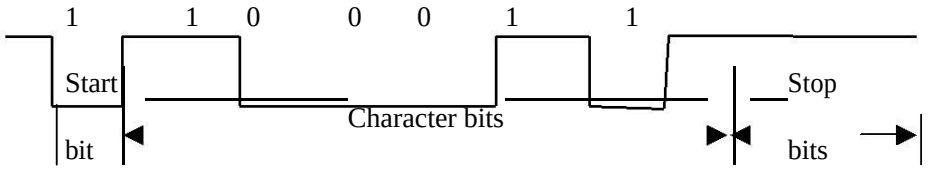


Figure 6-7 Asynchronous serial transmission

a start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten characters per second means that each character takes 0.1s for transfer. Since there are 11 bits to be

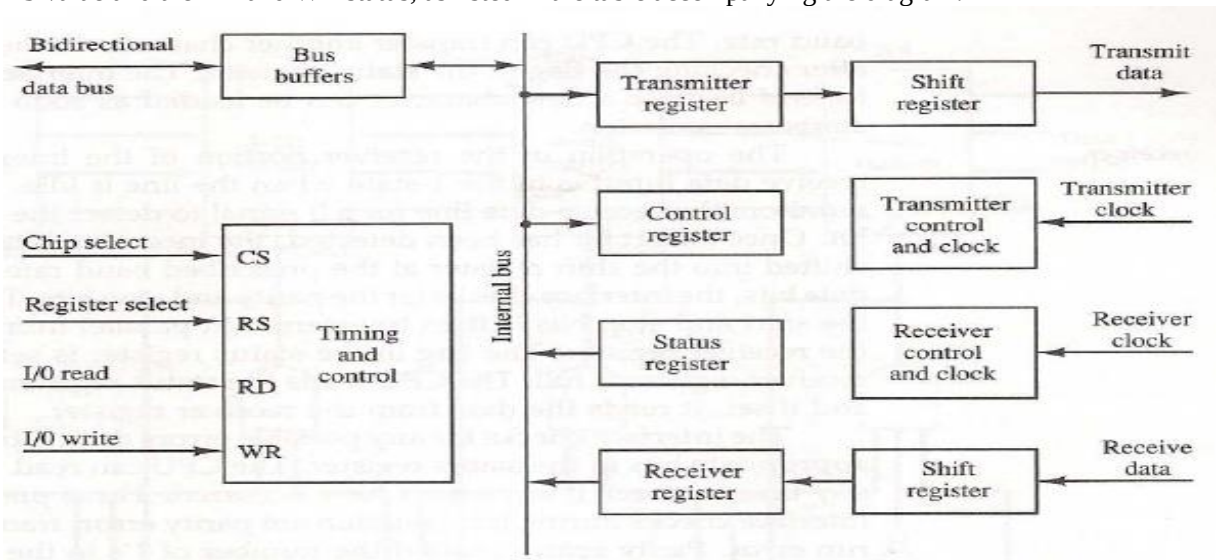
transmitted, it follows that the bit time is 9.09 ms.the baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

The terminal has a keyboard and a printer. Every time a key is depressed, the terminal sends 11 bits serially

along a wire. To print a character. To print a character in the printer, an 11-bit message must be received along another wire. The terminal interface consists of a transmitter and a receiver. The transmitter accepts an 8-bit character room the computer and proceeds to send a serial 11-bit message into the printer line. The receiver accepts a serial 11-bit message from the keyboard line and forwards the 8-bit character code into the computer. Integrated circuits are available which are specifically designed to provide the interface between computer and similar interactive terminals. Such a circuit is called an asynchronous communication interface or a universal asynchronous receiver-transmitter (UART).

Asynchronous Communication Interface

Fig shows the block diagram of an asynchronous communication interface is shown in Fig. It acts as both a transmitter and a receiver. The interface is initialized for a particular mode of transfer by means of a control byte that is loaded into its control register. The transmitter register accepts a data byte from the CPU through the data bus. This byte is transferred to a shift register for serial transmission. The receiver portion receives serial information into another shift register, and when a complete data byte is accumulated, it is transferred to the receiver register. The CPU can select the receiver register to read the byte through the data bus. The bits in the status register are used for input and output flags and for recording certain errors that may occur during the transmission. The CPU can read the status register to check the status of the flag bits and to determine if any errors have occurred. The chip select and the read and write control lines communicate with the CPU. The chip select (CS) input is used to select the interface through the address bus. The register select (RS) is associated with the read (RD)
and write (WR) controls. Two registers are write-only and two are read-only. The register selected is a function of the RS value and the RD and WR status, as listed in the table accompanying the diagram.



CS	RS	Operation	Register selected
0	×	×	None: data bus in high-impedance
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

MODES OF TRANSFER

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory MODULE. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory MODULE. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; other transfer the data directly to and from the memory

MODULE. Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O

2. Interrupt-initiated I/O

3. Direct memory access (DMA)

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface MODULE and the I/O device.

In the programmed I/O method, the CPU stays in a program loop until the I/O MODULE indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from thedevice. In the meantime the CU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory MODULE through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory.

Many computers combine the interface logic with the requirements for direct memory access into one MODULE and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMPA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory MODULE, the CPU, and the IOP.

EXAMPLE OF PROGRAMMED I/O

In the programmed I/O method, the I/O device dies not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU, and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig. 6.8. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets ait in the status register that we will refer to as an F or “flag” bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface. This is according to the handshaking procedure established in Fig. 6-5.A program is written for the computer to check the flag in the status register to determine if a

byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of

the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program that must be written for the CPU is shown in Fig. 6-9. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words form an I/O device and store them in a memory buffer. A program that stores input characters in a memory buffer using the instructions mentioned in the earlier chapter.

Figure 6-8 Data transfer form I/O device to CPU

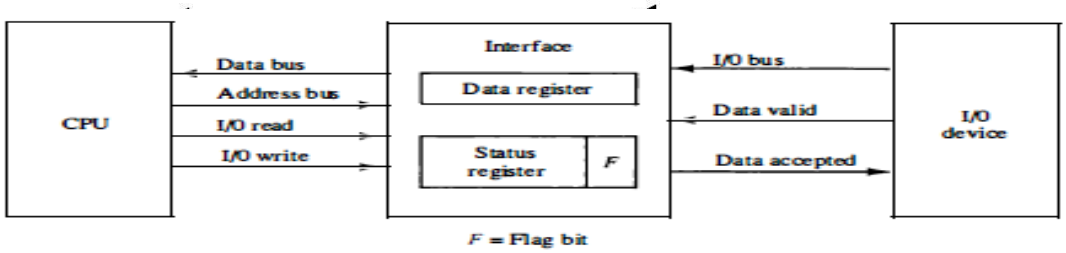
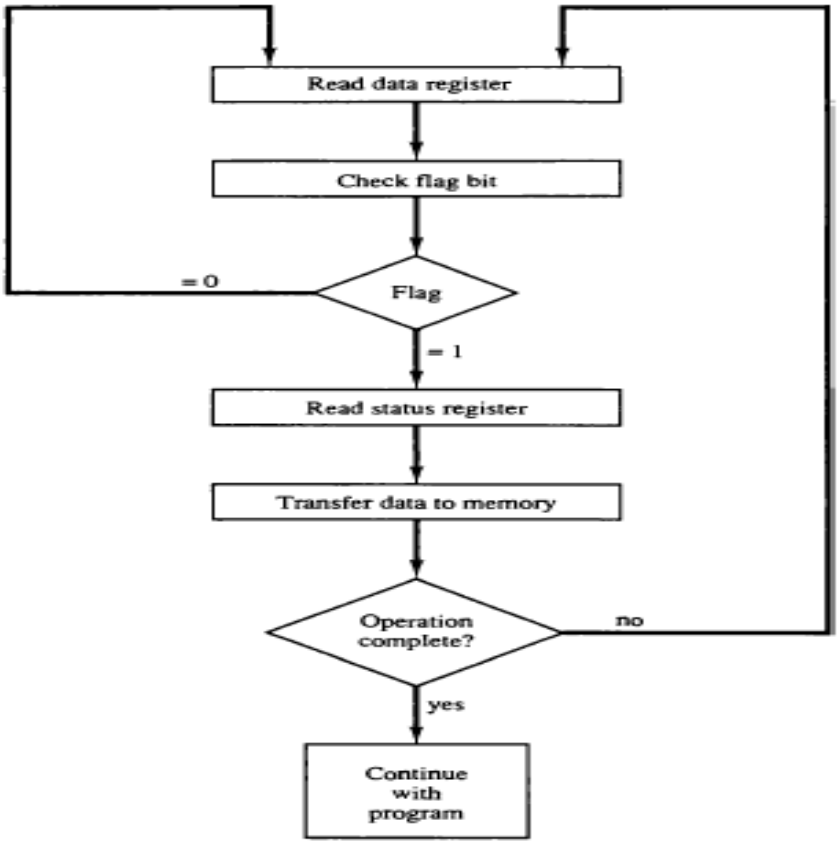


Figure 6-9 Flowcharts for CPU program to input data



The programmed I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient. To see why this is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1 πs. Assume that the input device transfers its data at an average rate of 100 bytes per second. This is equivalent to one byte every 10,000 π s. This means that the CPU will check the flag 10,000 times between each transfer. The CPU is wasting time while checking the flag instead of doing some other useful processing task.

INTERRUPT-INITIATED I/O

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from tone MODULE to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, no vectored interrupt. In a non-vectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

DIRECT MEMORY ACCESS (DMA)

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

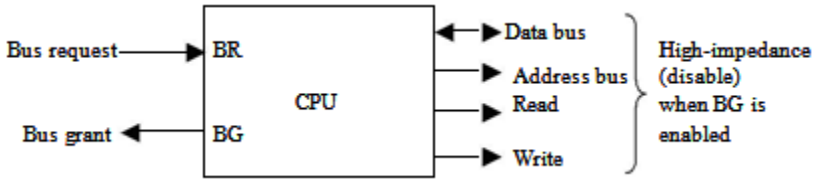
The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure 6.13 shows two control signals in the CPU that facilitate the DMA transfer. The bus request (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state behaves like an open circuit, which means that the output is disconnected and dies not have a logic significance. The CPU activates the Bus grant (BG) output to inform the external DMA that the buses are in the high -impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA burst transfer, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique called cycle stealing allows the DMA controller to transfer one data word at a time after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to “steal” one memory cycle.

DMA CONTROLLER

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines. The address registers and addresses lines

Figure 6.13 CPU bus signals for DMA transfer.



Are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Figure 6.14 shows the block diagram of a typical DMA controller. The MODULE communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional. When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When BG = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control. ;the DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory. The word count register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral MODULE until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

1The starting address of the memory block where data are available (for read) or where data are to be stored (for write)

2 The word count, which is the number of words in the memory block

3 Control to specify the mode of transfer such as read or write

4 A control to start the DMA transfer

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA TRANSFER

The position of the DMA controller among the other components in a computer system is illustrated in Fig. 6.15. The CPU communicates with the DMA through the address and data buses as with any interface MODULE. The DMA has its own address, which activates the DS and RS lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device. Note that the RD and WR lines in the DMA controller are bidirectional. Thedirection of transfer depends on the status of the BG line. When BG line. When BG = 0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG = 1, the RD and WR and output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.

When the peripheral device receives a DMA acknowledge, it puts a word in the data us (for write) or receives a

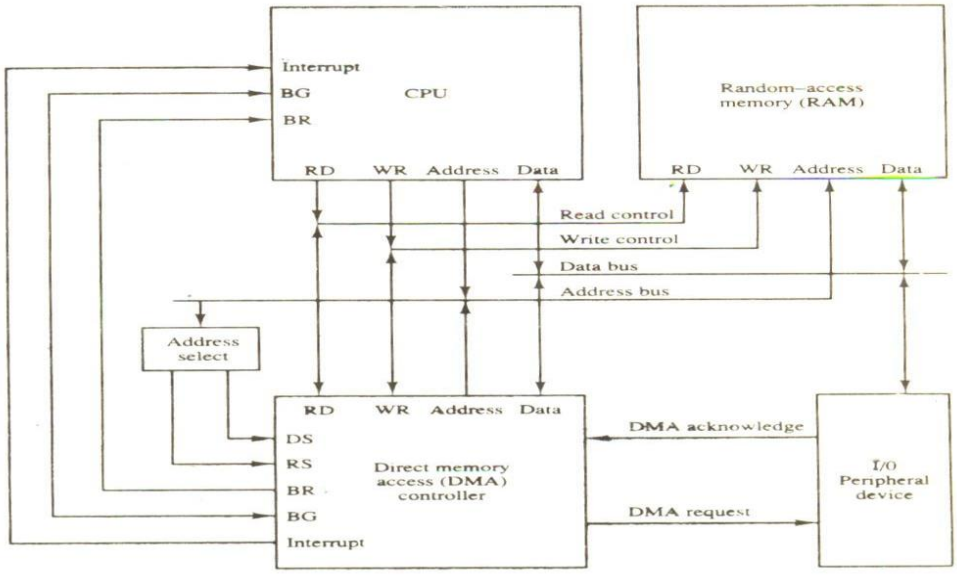


Figure 6.15 DMA transfer in a computer system.

word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral MODULE can then communicate with memory through the data bus for direct transfer between the two MODULEs while the CPU is momentarily disabled.

For each word that is transferred, the DMA increments its address registers and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For a high-speed device, the line will be active as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

It the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.

A DMA controller may have more than on channel. In this case, each channel has a request and acknowledges pair of control signals which are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.

DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal. Typically, an image of the screen display of the terminal is kept in memory which can be updated under program control. The contents of the memory can be transferred to the screen periodically by means of DMA transfer.

Input-output Processor (IOP)

The IOP is similar to a CPU except that it is designed to handle the details of I/O processing. Unlike the DMA controller that must be set up entirely by the CPU, the IOP can fetch and execute its own instructions. IOP instructions are specially designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks, such as arithmetic, logic, branching, and code translation.

The block diagram of a computer with two processors is shown in Figure 6.39. The memory MODULE occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transfer of data between various peripheral devices and the memory MODULE.

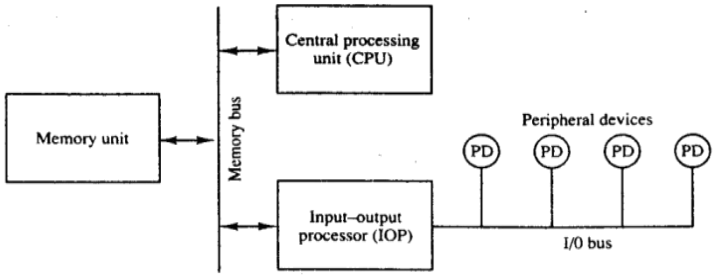


Figure : Block diagram of a computer with I/O processor

The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory. Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program. After the input data are assembled into a memory word, they are transferred from IOP directly into memory by "stealing" one memory cycle from the CPU. Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output device at the device rate and bit capacity.

The communication between the IOP and the devices attached to it is similar to the program control method of transfer. The way by which the CPU and IOP communicate depends on the level of sophistication included in the system. In most computer systems, the CPU is the master while the IOP is a slave processor. The CPU is assigned the task of initiating all operations, but I/O instructions are execute in the IOP. CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities. The IOP, in turn, typically asks for CPU attention by means of an interrupt. It also responds to CPU requests by placing a status word in a prescribed location in memory to be examined later by a CPU program. When an I/O operation is desired, the CPU informs the IOP where to find the I/O program and then leaves the transfer details to the IOP.

CPU-IOP Communication

The communication between CPU and IOP. These are depending on the particular computer considered. In most cases the memory MODULE acts as a message center where each processor leaves information for the other. To appreciate the operation of a typical IOP, we will illustrate by a specific example the method by which the CPU and IOP communicate. This is a simplified example that omits many operating details in order to provide an overview of basic concepts.

The sequence of operations may be carried out as shown in the flowchart of Fig. 6.40. The CPU sends an instruction to test the IOP path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the IOP where to find its program.

The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the IOP terminates the execution of its program, it sends an interrupt request to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The IOP responds by placing the contents of its status report into a specified memory location.

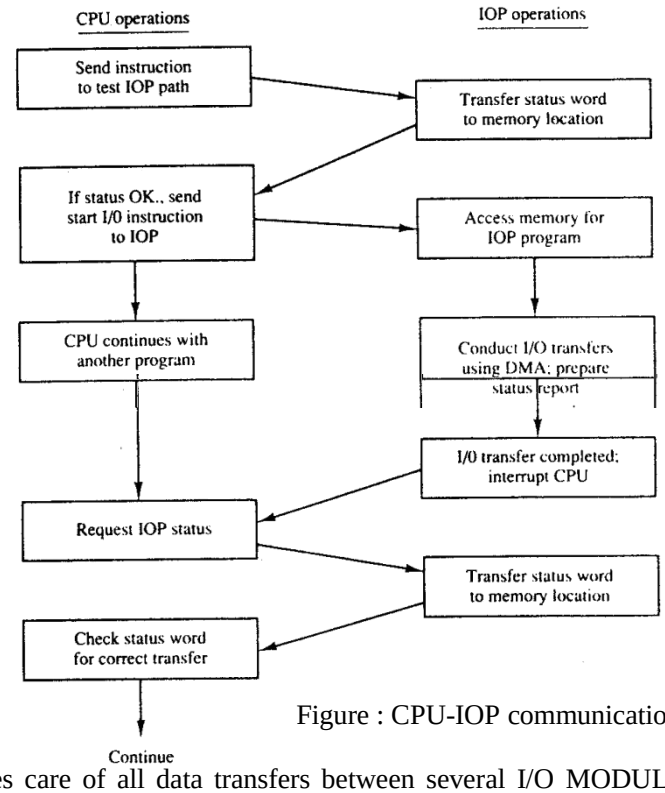


Figure : CPU-IOP communication

The IOP takes care of all data transfers between several I/O MODULEs and the memory while the CPU is processing another program. The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory.

PRIORITY INTERRUPT

Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal. The CPU responds to the interrupt
request by storing the return address from PC into a memory stack and then the program branches to a service routine that processes the required transfer. Some processors also push the current PSW for the service routine. We neglect the PSW here in order not to complicate the discussion of I/O interrupts.

In a typical application a number of I/O devices are attached to the computer, with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

A priority interrupts is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more request arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to request which, if delayed of interrupted, could have serious consequences. Devices with high-speed transfers such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

Establishing the priority of simultaneous interrupts can be done by software or hardware. A polling procedure

is used to identify the highest-priority source by software means. In this method there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The highest-priority source is tested first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower-priority source is tested, and so on. Thus the initial service routine for all interrupt consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest-priority device among all devices that interrupted the computer. The disadvantage of the soft ware method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation a hardware priority-interrupt MODULE can be used to speed up the operation.

A hardware priority-interrupt MODULE functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt MODULE. The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy chaining method.

DAISY-CHAINING PRIORITY

The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Fig. 6-10 The interrupt request line is common to all devices and forms a wired logic connection. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU. This is equivalent to a negative logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.

A device with a 0 in its PI input generates a 0 in its PO output to inform the next-lower-priority device that the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 in its PI input will intercept the acknowledge signal by placing a 0 in its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device

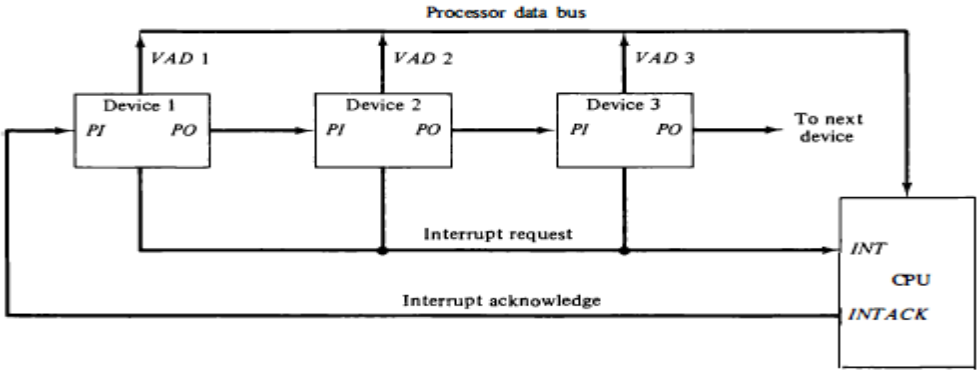


Figure 6-10 Daisy-chain priority interrupt

by placing a 1 in its PO output. Thus the device with PI = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

Figure 6.11 shows the internal logic that must be included with in each device when connected in the daisy-chaining scheme. The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line. If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF. If PI = 1 and RF = 0, then PO = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO. The device is active when PI = 1 and RF = 1. This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

PARALLEL PRIORITY INTERRUPT

The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register the circuit may include a mask register whose
purpose is to control the status of each interrupt request. The mask register can be programmed to disable

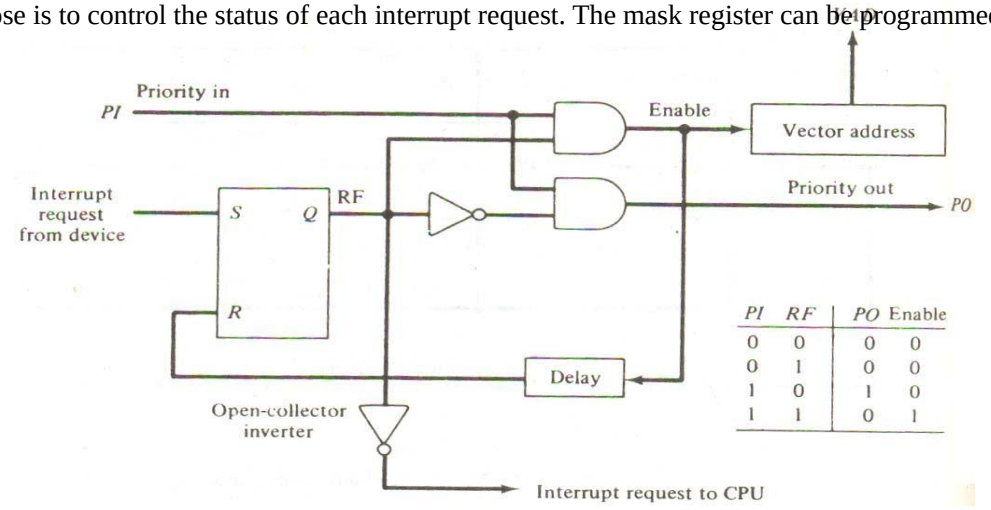


Figure : One state of the daisy-chain priority arrangement.

lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

The priority logic for a system of four interrupt sources is shown in Fig. 6.12. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs. The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system. The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus. We will now explain the priority encoder circuit and then discuss the interaction between the priority interrupt controller and the CPU.

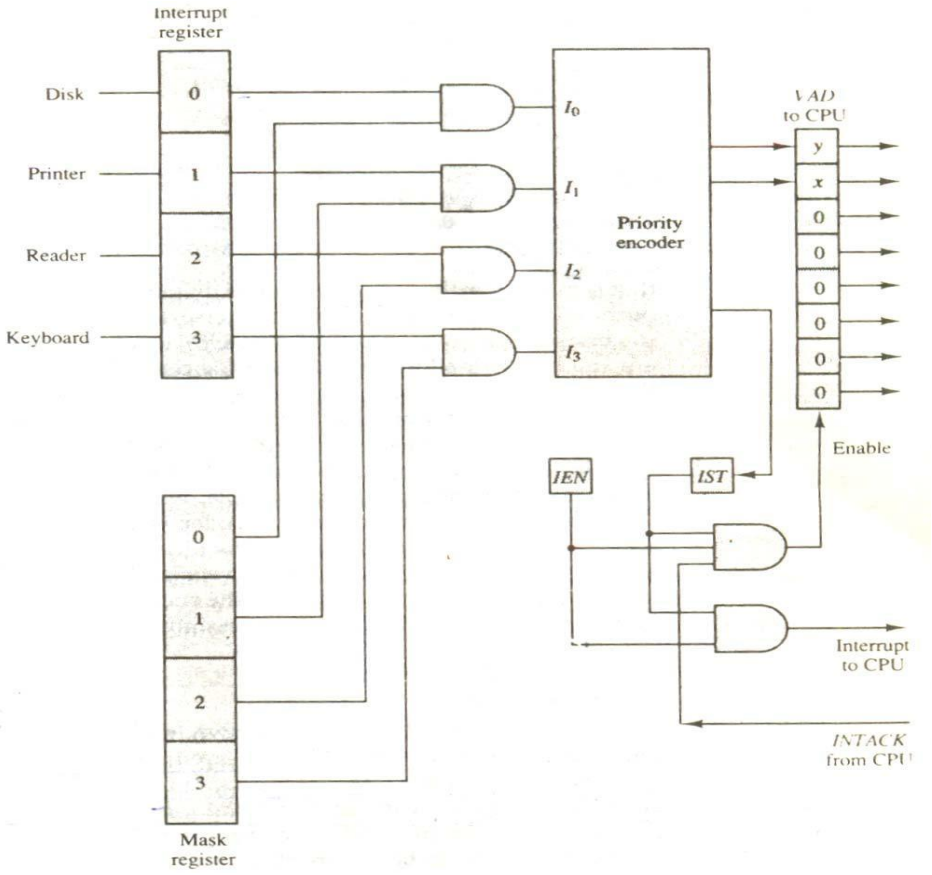


Figure : Priority interrupt hardware.

MODULE-5

PIPELINE

Pipeline: Parallel processing, pipelining-arithmetic pipeline, instruction pipeline; Multiprocessors: Characteristics of multiprocessors, inter connection structures, inter processor arbitration, and inter processor communication and synchronization

Parallel processing

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

Figure shows one possible way of separating the execution MODULE into eight functional MODULEs operating in parallel. The operands in the registers are applied to one of the MODULEs depending on the operation specified by the instruction.

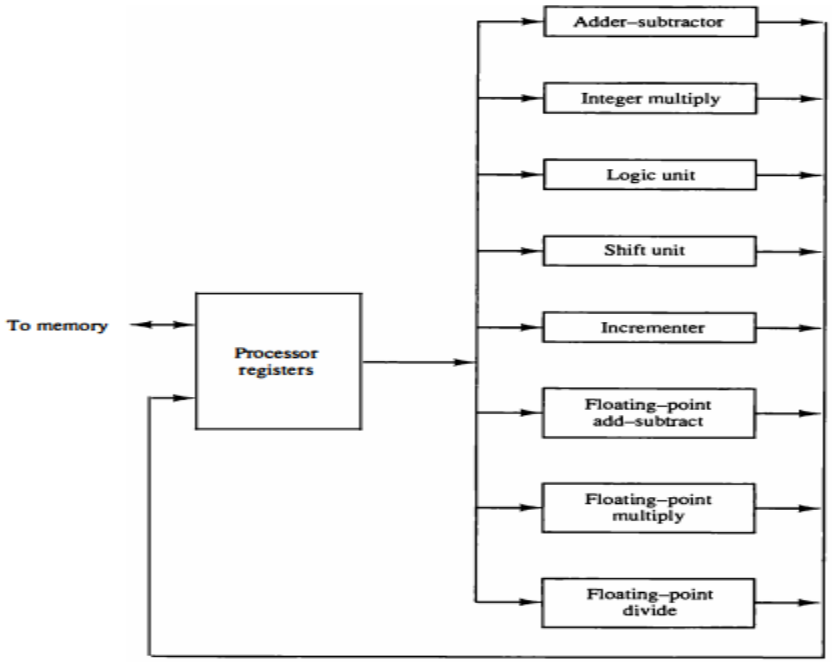


Figure: Processor with multiple functional MODULEs.

associated with the operands. The operation performed in each functional MODULE is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating-point operations are separated into three circuits operating in parallel. The logic, shift, and increment operations can be performed concurrently on different data. All MODULEs are independent of each other, so one number can be shifted while another number is being incremented. A multifunctional organization is usually associated with a complex control MODULE to coordinate all the activities among the various components.

There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One classification introduced by M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously. The normal operation of a computer is to fetch instructions from memory and execute them in the processor. The sequence of instructions read from memory constitutes an instruction stream. The operations performed on the data in the processor constitutes a data stream .Parallel processing may occur in the instruction stream, in the data stream, or in both. Flynn's classification divides computers into four major groups as follows:

Single instruction stream, single data stream (SISD)

Single instruction stream, multiple data stream (SIMD)

Multiple instruction stream, single data stream (MISD)

Multiple instruction stream, multiple data stream (MIMD)

SISD represents the organization of a single computer containing a control MODULE, a processor MODULE, and a memory MODULE. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional MODULEs or by pipeline processing. SIMD represents an organization that includes many processing MODULEs under the supervision of a common control MODULE. All processors receive the same instruction from the control MODULE but operate on different items of data. The shared memory MODULE must contain multiple modules so that it can communicate with all the processors simultaneously. MISD structure is only of theoretical interest since no practical system has been constructed using this organization. MIMD organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multicomputer systems can be classified in this category. Flynn's classification depends on the distinction between the performance of the control MODULE and the data-processing MODULE. It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining.

In this chapter we consider parallel processing under the following main topics:

1. Pipeline processing

2. Vector processing

3. Array processors

Pipeline processing is an implementation technique where arithmetic sub operations or the phases of a computer instruction cycle overlap in execution. Vector processing deals with computations involving large vectors and matrices. Array processors perform computations on large arrays of data.

Pipelining

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segments in the pipeline. The final result is obtained after the data have passed through all segments. The

pipeline organization will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i*B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. R 1 through RS are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

R 1 <-- A_i, R2 <-- B_i Input A_i and B_i

R3 <--R 1 * R2, R4 <-- Ci Multiply and input C_i

R5 <--R3 + R4 Add $C_i$to product

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table. The first clock pulse transfers A1 and 81 into R 1 and R2.

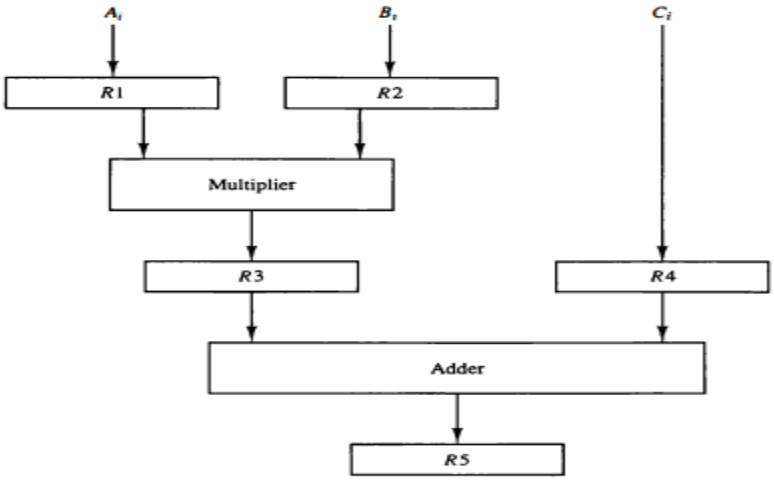


Figure : Example of pipeline processing.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	A_1*B_1	C_1	—
3	A_3	B_3	A_2*B_2	C_2	$A_1*B_1 + C_1$
4	A_4	B_4	A_3*B_3	C_3	$A_2*B_2 + C_2$
5	A_5	B_5	A_4*B_4	C_4	$A_3*B_3 + C_3$
6	A_6	B_6	A_5*B_5	C_5	$A_4*B_4 + C_4$
7	A_7	B_7	A_6*B_6	C_6	$A_5*B_5 + C_5$
8	—	—	A_7*B_7	C_7	$A_6*B_6 + C_6$
9	—	—	—	—	$A_7*B_7 + C_7$

TABLE: Content of Registers in Pipeline Example

The second dock pulse transfers the product of R 1 and R2 into R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R 1 and R2. The third clock pulse operates on all three segments simultaneously. It places A, and B, into R1 and R2, transfers the product of R1 and R2 into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS.

General Considerations

The general structure of a four-segment pipeline is illustrated in Fig. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S; that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers R; that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We task define a task as the total operation performed going through all the segments in the pipeline.

The behavior of a pipeline can be illustrated with a space-time diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Fig. The horizontal axis diagram the time in clock cycles and the vertical axis gives the

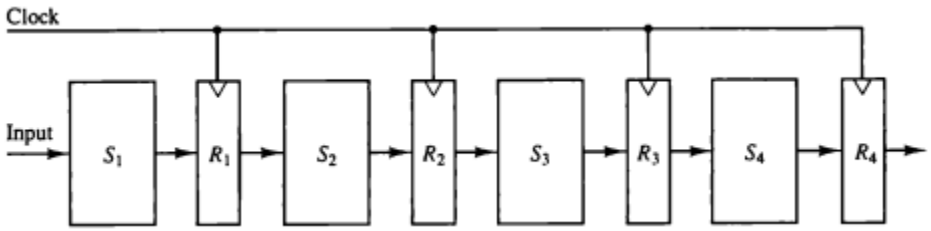


Figure: Four-segment pipeline.

Segment number. The diagram shows six tasks T1 through T6 executed in four segments. Initially, task T_1 is handled by segment 1. After the first clock, segment 2 is busy with T_1, while segment 1 is busy with task T_2.Continuing in this manner, the first task T_1 is completed after the fourth clock cycle. Now consider the case where a k-segment pipeline with a clock cycle time t, is used to execute n tasks. The first task T1 requires a time equal to kt, to complete its operation since there are k segments in the pipe. The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n - 1)tp. Therefore, to complete n tasks using a k-segment pipeline requires k + (n - 1) clock cycles. For example, the diagram of Fig. 9-4 shows four segments and six tasks. The time required to complete all the operations is 4 + (6 - 1) = 9 clock cycles, as indicated in the diagram.

Next consider a nonpipeline MODULE that performs the same operation and takes a time equal to t. to complete each task. The total time required for n tasks is nt_n. The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

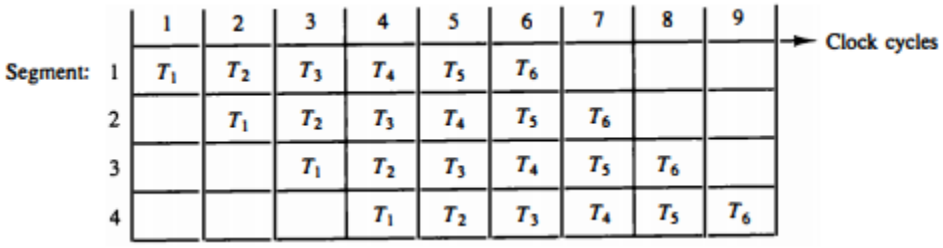


Figure: Space-time diagram for pipeline.

As the number of tasks increases, n becomes much larger than k - 1, and k + n - 1 approaches the value of n. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and non pipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to this show that the theoretical maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline. To clarify the meaning of the speedup ratio, consider the following numerical example. Let the time it takes to process a suboperation in each segment be equal to tp = 20 ns. Assume that the pipeline has k = 4 segments and executes n = 100 tasks in sequence. The pipeline system will take (k + n - 1)tp= (4 + 99) x 20 = 2060 ns to complete. Assuming that t, = ktp = 4 x 20 = 80 ns, a non-pipeline system requires nktp= 100 x 80 = 8000 ns to complete the 100 tasks. The speedup ratio is equal to 8000/2060 = 3.88. As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that t, = 60 ns, the speedup becomes 60/20 = 3.

Arithmetic Pipeline

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X=A \times 2^a$$

$$Y = B \times 2^b$$

A and B are two fractions that represent the mantissas and a and b are the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Fig. The registers labeled R are placed between the segments to store intermediate results. The sub operations that are performed in the four segments are:

1. Compare the exponents.

2. Align the mantissas.

3. Add or subtract the mantissas.

4. Normalize the result

This follows the procedure outlined in the flowchart of Fig. 10-15 but with some variations that are used to reduce the execution time of the suboperations. The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two

mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

The following numerical example may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Fig. 9-6 refers to binary numbers. Consider the two normalized floating-point numbers:

$X = 0.9504 \text{ X } 10^3$

$Y = 0.8200 \text{ X } 10^2$

The two exponents are subtracted in the first segment to obtain 3 - 2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$X= 0.9504 \text{ X } 10^3$

$Y = 0.0820 \text{ X } 10^3$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$Z = 1 .0324 \text{ X } 10^3$

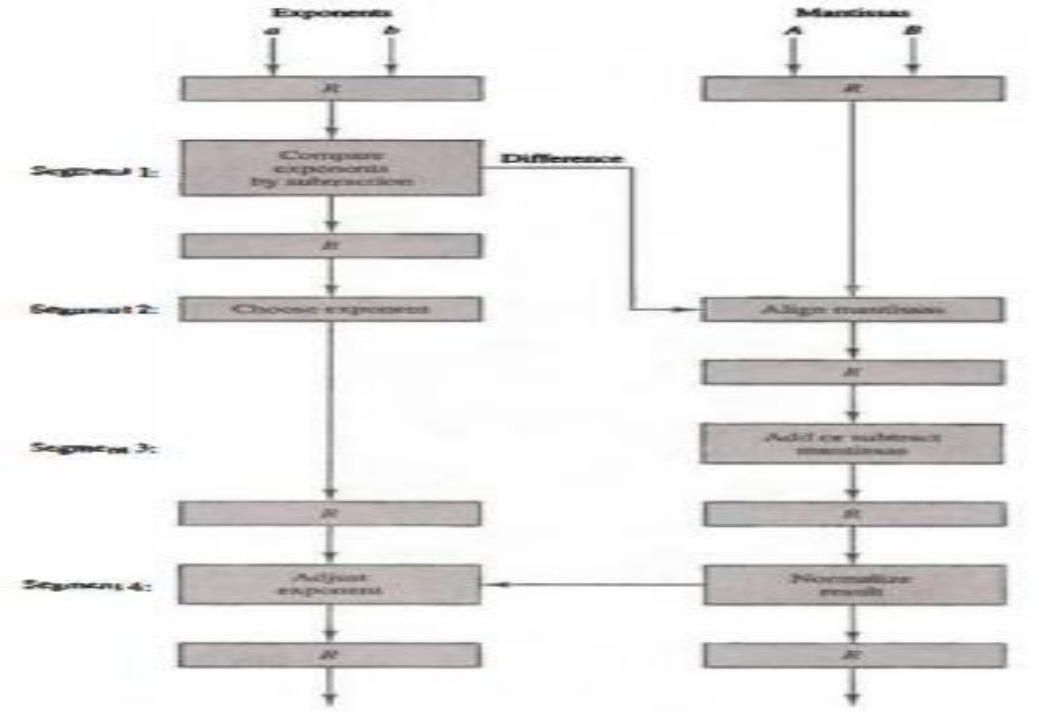


Fig: Pipeline for floating point Addition and subtraction

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \text{ X } 10^4$$

The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are t_1 = 60 ns, t_2 = 70 ns, t_3 = 100 ns, t_4 = 80 ns, and the interface registers have a delay of t_r = 10 ns. The dock cycle is chosen to be tp = t 3 + tr = 110 ns. An equivalent nonpipelinefloatingpoint adder-subtractor will have a delay time tn = t_1 + t2 + t_3 + t_4 + tr = 320 ns. In this case the pipelined adder has a speedup of 320/110 = 2. 9 over the nonpipelined adder.

Instruction Pipeline

An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. One possible digression associated with such a scheme is that an instruction may cause a branch out of sequence. In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

In the most general case, the computer needs to process each instruction with the following sequence of steps.

1. Fetch the instruction from memory.

2. Decode the instruction.

3. Calculate the effective address.

4. Fetch the operands from memory.

5. Execute the instruction.

6. Store the result in the proper place

For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. Ln this way, an instruction word and a data word can be read simultaneously from two different modules The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration.

Example: Four-Segment Instruction Pipeline

Figure shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3. The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO.

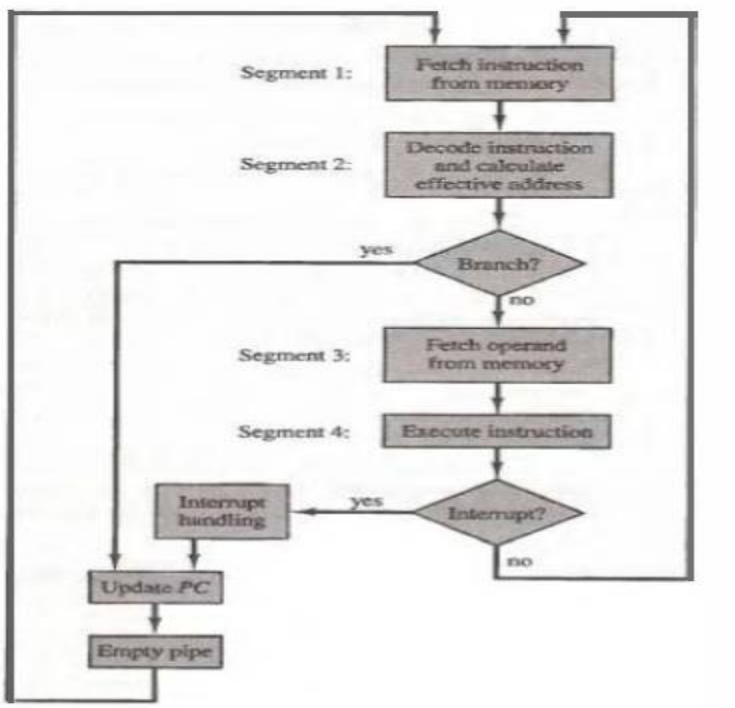


Figure- Four-segment CPU pipeline.

The Below Figure shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

1. Fl is the segment that fetches an instruction.

2. DA is the segment that decodes the instruction and calculates theeffective address.

3. FO is the segment that fetches the operand.

4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in Fl and FO can proceed at the same time.In the absence.of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. Resource conflicts caused by access to memory by two segments at the same time. Most of instruction and data memories.
2. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. Branch difficulties arise from branch and other instructions that change the value of PC.

Multiprocessors

Characteristics of Multiprocessors: A multiprocessors system is an interconnection of two or more CPUs with memory and input-output equipment. The term "processor" in multiprocessor can mean either a central processing MODULE (CPU) or an input-output processor (IOP).

Computers are interconnected with each other by means of communication lines to form a computer network. The network consists of several autonomous computers that may or may not communicate with each other. A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.

Multiprocessing improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to function correctly with perhaps some loss in efficiency.

The benefit derived from a multiprocessors organization is an improved system performance. The system derives its high performance from the fact that computations can proceed in parallel in one of two ways.

1. Multiple independent jobs can be made to operate in parallel.
2. A single job can be partitioned into multiple parallel tasks.

Interconnection Structures:

The components that form a multiprocessors system are CPUs, IOPs connected to input-output devices, and a memory MODULE that may be partitioned into a number of separate modules. The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system. There are several physical forms available for establishing an interconnection network. Some of these schemes are presented in this section:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

Time-shared Common Bus:

A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory MODULE. A time-shared common bus for five processors is shown in fig. Only one processor can communicate with the memory or another processor at any given time. Transfer operations are conducted by the processor that is in control of the bus at the time. Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination MODULE to initiate the transfer. A command is issued to inform the destination MODULE what operation is to be performed. The receiving MODULE recognizes its address in the bus and responds to the control signals from the

sender, after which the transfer is initiated. The system may exhibit transfer conflicts since one common bus is shared by all processors. These conflicts must be resolved by incorporating a bus controller that establishes priorities among the requesting MODULEs.

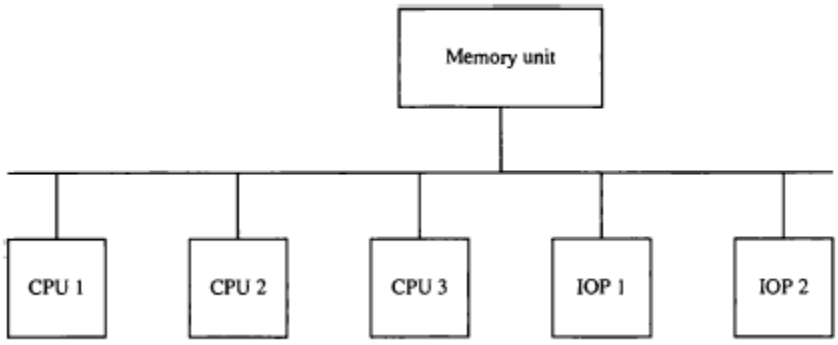


Figure: Time-shared common bus organization.

A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path. The processors in the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers. However, this increases the system cost and complexity.

A more economical implementation of a dual bus structure is depicted in Fig. Here we have a number of local buses each connected to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combination of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus, the I/O devices attached to it may be designed as a cache memory attached to the CPU (see Sec. 12-6). In this way, the average access time of the local memory can be made to approach the cycle time of the CPU to which it is attached.

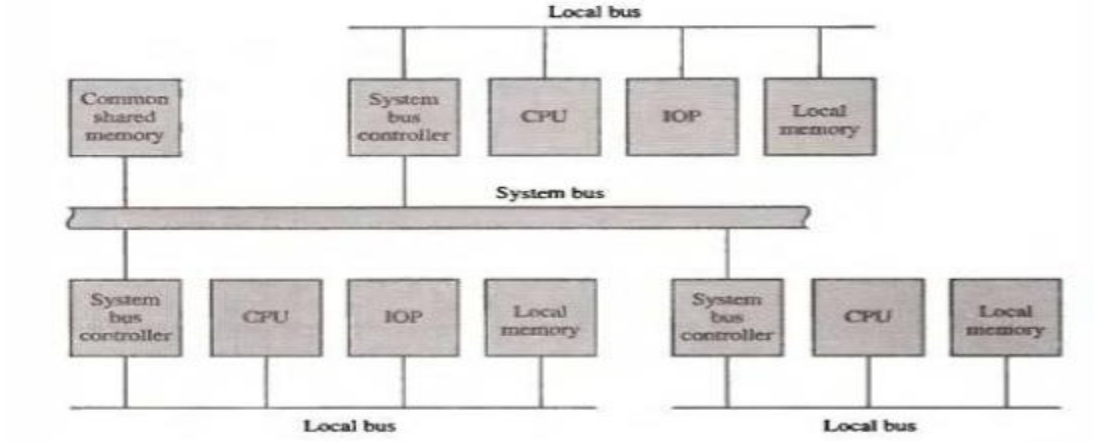


Fig: System bus structure for multi-processors

Multi-port Memory:

A multiport memory system employs separate buses between each memory module and each CPU. This is shown in Fig. 13-3 for four CPUs and four memory modules (MMs). Each processor bus is connected to each memory module. A processor bus consists of the address, data and control lines required to communicate with memory. The memory module is said to have four ports and each port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port. The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module. Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.

The advantage of the multiport memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory. The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors. As a consequence, this interconnection structure is usually appropriate for systems with a small number of processors.

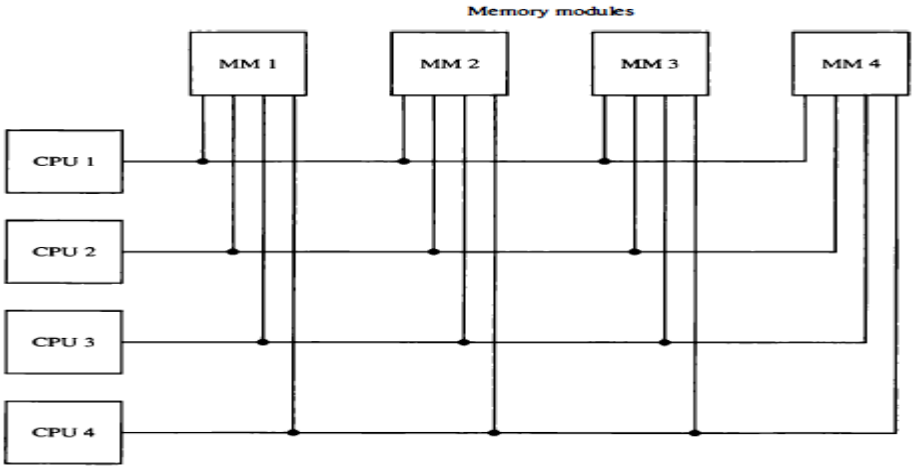


Figure: Multiport memory organization.

Crossbar Switch:

The crossbar switch organization consists of a number of cross prints that are placed at intersections between processor buses and memory module paths. Figure shows a crossbar switch interconnection between four CPUs and four memory modules. The small square in each cross point is a switch that determines the path from a processor to a memory module. Each switch point has control logic to set up the transfer path between a processor and memory. It examines the address that is placed in the bus to determine whether its particular module is being addressed. It also resolves multiple requests for access to the same memory module on a predetermined priority basis.

Figure shows the functional design of a crossbar switch connected to one memory module. The circuit consists of multiplexes that select the data, address, and control from one CPU for communication with the memory module. Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory. The multiplexes are controlled with the binary code that is generated by a priority encoder within the arbitration logic.

A crossbar switch organization supports simultaneous transfers from all memory modules because there is a separate path associated with each module. However, the hardware required to implement the switch could be quite large and complex.

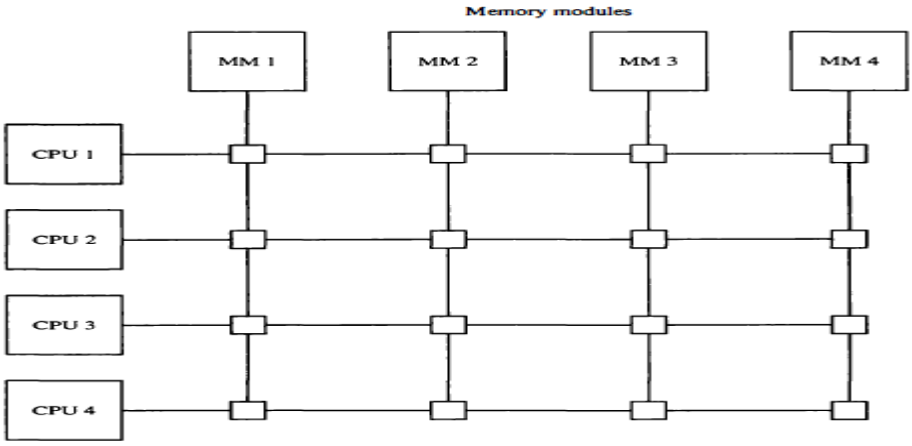


Figure: Crossbar Switch

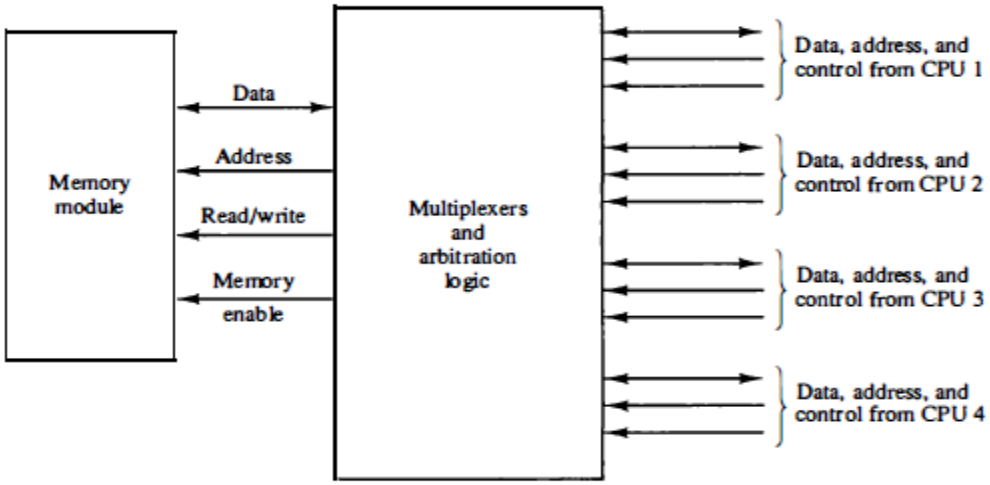


Figure: Block diagram of crossbar switch.

Multistage Switching Network:

The basic component of a multistage network is a two-input, two-output interchange switch. As shown in fig. the 2 X 2 switch has two inputs, labeled A and B, and two outputs, labeled 0 and 1. There are control signals (not shown) associated with the switch that establish the interconnection between the input and output terminals. The switch has the capability of connecting input A to either of the outputs. Terminal B of the switch behaves in a similar fashion. The switch also has the capability to arbitrate between conflicting requests. If inputs A and B both request the same output terminal, only one of them will be connected; the other will be blocked.

Using the 2 X 2 switch as a building block, it is possible to build a multistage network to control the communication between a number of source and destinations. To see how this is done, consider the binary tree shown Fig. 13-7. The two processors P1 and P2 are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to a destination is determined from the binary bits of the destination number. The first bit of the destination number determines the switch output in the first level. The second bit specifies the output of the switch in the second level, and third bit specifies the output of the switch in the third level. For example, to connect P1 to memory 101, it is necessary to form a path from P1 to output 1 in the first level switch, output 0 in the second-level switch, and output 1 in the third-level switch. It is clear that either P1 or P2 can be connected to any one of the eight memories. Certain request patterns, however, cannot be satisfied simultaneously. For example, if P1 is connected to one of the destinations 000 through 011,P2 can be connected to only one of the destinations 100 through 111.

Many different topologies have been proposed for multistage switching networks to control processor-memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system. One such topology is the omega-switching network shown in Fig. 13-8. In this configuration, there is exactly one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

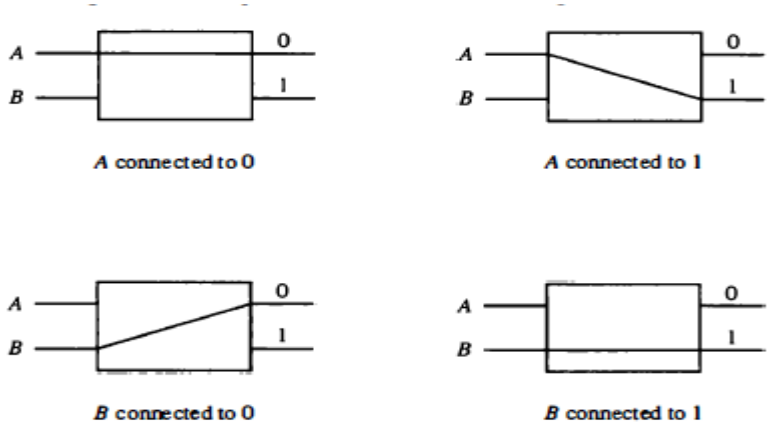


Figure: Operation of 2 X 2 interchange switch.

A particular request is initiated in the switching network by the source, which sends a 3- bit pattern representing the destination number. As the binary pattern moves through the network, each level examines a different bit to determine the 2 X 2 switch setting. Level 1 inspects the most significant bit, level 2 inspects the middle bit, and level 3 inspects the least significant bit. When the request arrives on either input of the 2 X 2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if the bit is 1.

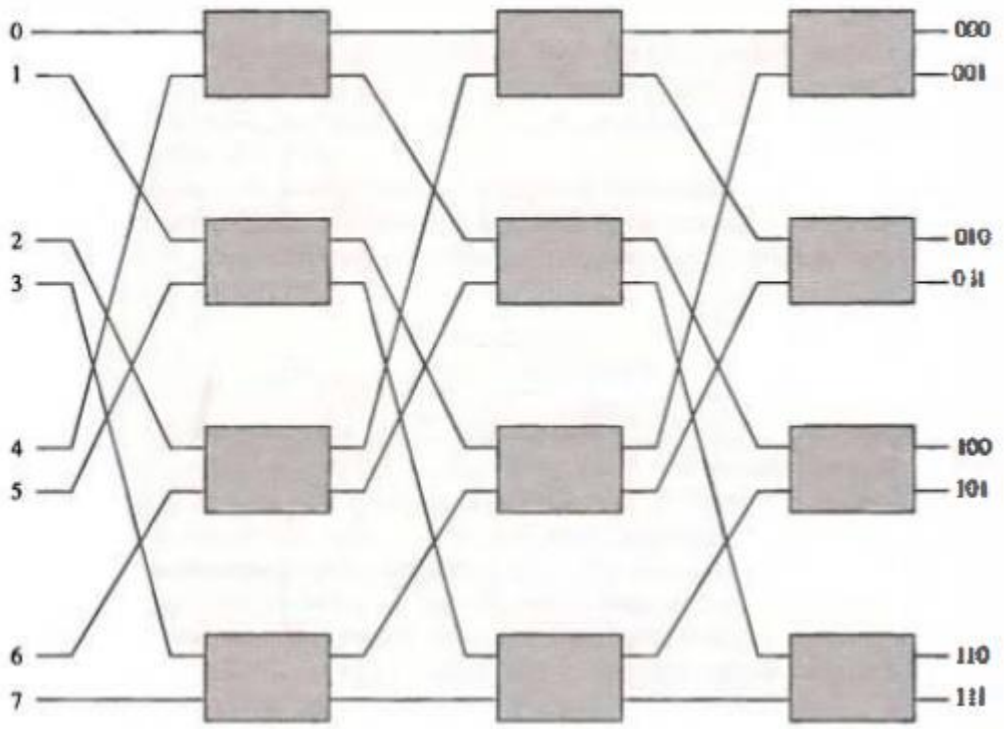


Fig: 8 x 8 omega switching network

Hypercube Interconnection:

The hypercube or binary n – cube multiprocessor structure is a loosely coupled system composed of $N = 2^n$ processor interconnected in an n- dimensional binary cube. Each processor forms a node of the cube. Although it is customary to refer to each node as having a processor, in effect it contains not only a CPU but also local memory and I/O interface. Each processor has direct communication paths to n other neighbor processors. These paths correspond to the edges of the cube. There are 2^n distinct n- bit binary addresses that can be assigned to the processors. Each processor address differs from that of each of its n neighbors by exactly one bit position.

Figure shows the hypercube structure for n = 1,2, and 3. A one-cube structure has n = 1 and 2" = 2. It contains two processor interconnected by a single path. A two-cube structure has n= 2 and 2" = 4. It contains four nodes interconnected as a square. A three- cube structure has eight nodes interconnected as a cube. An n – cube structure has 2" nodes with a processor residing in each node. Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position. For example, the three neighbors of the node with address 100 in a three – cube structure are 000, 110,and 101 each of these binary numbers differs from address 100 by one bit value.

Routing message through an n-cube structure may take from one to n links from a source node to a destination node. For example, in a three- cube structure, node to a destination node. For example, in a three- cube structure, node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011 (from 000 to 001 to 011 or from 000 to 010 to 011). It is necessary to go through at least three links to communicate from node 000 to node 111. A routing procedure can be developed by computing the exclusive –OR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes. For example, in a three- cube structure, a message at 010 going to 001 produces an exclusive- OR of the two-address equal to 011. The message can be sent along the

second axis to 000 and then through the third axis to 001.

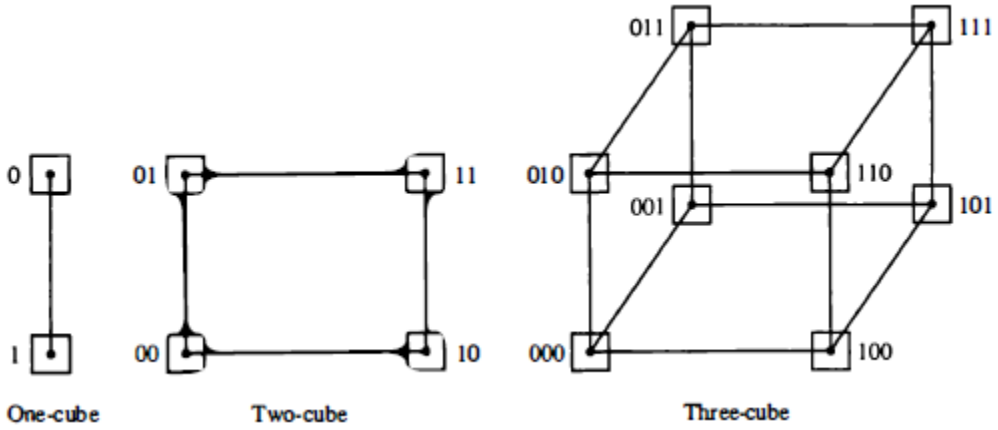


Figure: Hypercube structures for n = 1 , 2 ,3.

A representative of the hypercube architecture is the Intel Ipsc computer complex. It consists of 128 (n = 7) microcomputers connected through communication channels. Each node consists of CPU, a floating-point processor, local memory, and serial communication interface MODULEs. The individual nodes operate independently on data stored in memory according to resident programs. The data and programs to each node come through a message-passing system from other nodes or from a cube manager. Application program are developed and complied on the cube manager and then download to the individual nodes. Computations are distributed through the system and executed concurrently.

Inter processor arbitration:

Arbitration procedures service all processor requests on the basis of established priorities. A hardware bus priority resolving technique can be established by means of a serial or parallel connection of the MODULEs requesting control of the system bus. The serial priority resolving technique is obtained from a daisy-chain connection of bus arbitration circuits similar to the priority interrupt logic presented in Sec. 1 1-5. The processors connected to the system bus are assigned priority according to their position along the priority control line.
Figure 13-10 shows the daisy-chain connection of four arbiters. It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines. The priority out (PO) of each arbiter is connected to the

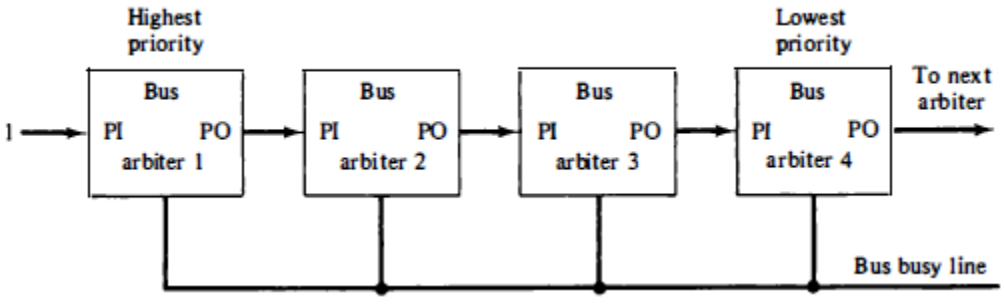


Figure: Serial (daisy-chain) arbitration.

Priority in (PI) of the next-lower-priority arbiter. The PI of the highest-priority MODULE is maintained at logic 1 value. The highest-priority MODULE in the system will always receive access to the system bus when it requests it. The PO output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus. This is the way that priority is passed to the next MODULE in the

chain. If the processor requests control of the bus and the corresponding arbiter finds its PI input equal to 1, it sets its PO output to 0. Lower-priority arbiters receive a 0 in PI and generate a 0 in PO. Thus the processor whose arbiter has a PI = 1 and PO = 0 is the one that is given control of the system bus.
A processor may be in the middle of a bus operation when a higher priority processor requests the bus. The lower-priority processor must complete its bus operation before it relinquishes control of the bus. The bus busy line shown in Fig. 13-10 provides a mechanism for an orderly transfer of control. The busy line comes from open-collector circuits in each MODULE and provides a wired-OR logic connection. When an arbiter receives control of the bus (because its PI = 1 and PO = 0) it examines the busy line. If the line is
Inactive, it means that no other processor is using the bus. The arbiter activates the busy line and its processor takes control of the bus. However, if the arbiter finds the busy line active, it means that another processor is currently using the bus. The arbiter keeps examining the busy line while the lower-priority
Processor that lost control of the bus completes its operation. When the bus busy line returns to its inactive state, the higher-priority arbiter enables the busy line, and its corresponding processor can then conduct the required bus transfers.

Parallel Arbitration Logic:

The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in Fig. Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line. Each arbiter enables the request line when its processor is requesting access to the system bus. The processor takes control of the bus if its acknowledge input line is enabled. The bus busy line provides an orderly transfer of control, as in the daisy-chaining case.

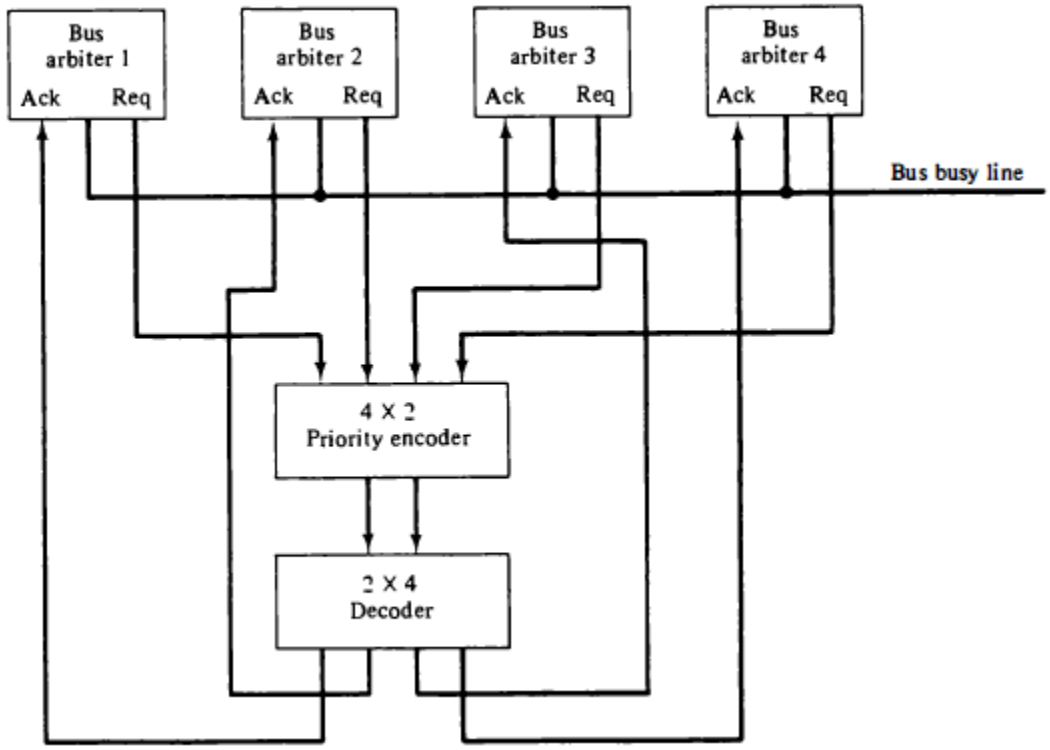


Figure: Parallel arbitration.

Figure shows the request lines from four arbiters going into a 4 x 2 priority encoder. The output of the encoder generates a 2-bit code which represents the highest-priority MODULE among those requesting the bus. The truth table of the priority encoder can be found in Table 11-2 (Sec. 11-5). The 2-bit code from the encoder output drives a 2 x 4 decoder which enables the proper acknowledge line to grant bus access to the highest-priority MODULE.

Inter processor communication and synchronization:

The various processors in a multiprocessor system must be provided with a facility for communicating with each other. A communication path can be established through common input-output channels. In a shared memory multiprocessor system, the most common procedure is to set aside a portion of memory that is accessible to all processors.
that a new message was inserted by the interrupting processor. In addition to shared memory, a multiprocessor system may have other shared resources. For example, a magnetic disk storage MODULE connected to an IOP may be available to all CPUs. This provides a facility for sharing of system programs stored in the disk. A communication path between two CPUs can be established through a link between two IOPs associated with two different CPUs. This type of link allows each CPU to treat the other as an I/O device so that messages can be transferred through the I/O path. To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. This task is given to the operating system. There are three organizations that have been used in the design of operating system for multiprocessors: master-slave configuration, separate operating system, and distributed operating system. In a master-slave mode, one processor, designated the master, always executes the operating system functions. The remaining processors, denoted as slaves, do not perform operating system functions. If a slave processor needs.
In a loosely coupled multiprocessor system the memory is distributed among the processors and there is no shared memory for passing information. The communication between processors is by means of message passing through VO channels. The communication is initiated by one processor calling a procedure that resides in the memory of the processor with which it wishes to communicate.

Interprocessor Synchronization:

The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes. Communication refers to the exchange of data between different processes. Synchronization refers to the special case where the data used to communicate between processors is control information. Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.
Multiprocessor systems usually include various mechanisms to deal with the synchronization of resources. Low-level primitives are implemented directly by the hardware. These primitives are the basic mechanisms that enforce mutual exclusion for more complex mechanisms implemented in software. A number of hardware mechanisms for mutual exclusion have been developed. One of the most popular methods is through the use of a binary semaphore.

Mutual Exclusion with a Semaphore:

A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources. This is necessary to protect data from being changed simultaneously by two or more processors. This mechanism has been termed mutual exclusion. Mutual exclusion must be provided in a multiprocessor system to enable one processor to exclude or lock out access to a shared resource by other processors when it is in a critical section. A critical section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource.
A binary variable called a semaphore is often used to indicate whether or not a processor is executing a critical section. A semaphore is a software controlled flag that is stored in a memory location that all processors can access. When the semaphore is equal to 1, it means that a processor is executing a critical program, so that the shared memory is not available to other processors. When the semaphore is equal to 0, the shared memory is available to any requesting processor. Processors that share the same memory segment agree by convention not to use the memory segment unless the semaphore is equal to 0, indicating that memory is available. They also agree to set the semaphore to 1 when they are executing a critical section and to clear it to 0 when they are finished.