MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: COMPUTER ORGANIZATION

SUBJECT CODE: SE22A

SEMESTER: II

PREPARED BY: PROF. C.SUJDHA

UNIT - I

Data representation: Data types – Complements- fixed point and floating point representation other binary codes. Register Transfer and Microoperations: Register transfer language- Register transfer- Bus and Memory transfers – Arithmetic, logic and shift micro operations.

Data Representation:Data type

- Registers contain either data or controlinformation
- Control information is a bit or group of bits used to specify the sequenceof command signals needed for data manipulation
- Data are numbers and other binary-coded information that are operatedon
- Possible data types inregisters:
  - Numbers used incomputations
  - Letters of the alphabet used in dataprocessing
  - Other discrete symbols used for specificpurposes
- All types of data, except binary numbers, are represented in binary-codedform

- A number system of *base*, or *radix*, *r* is a system that uses distinct symbols for*r* digits
- Numbers are represented by a string of digitsymbols
- The string of digits 724.5 represents thequantity

$$7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

- The string of digits 101101 in the binary number system represents thequantity

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$$

- $(101101)_2 = (45)_{10}$
- We will also use the octal (radix 8) and hexidecimal (radix 16) numbersystems

$$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1} = (478.5)_{10}$$

$$(F3)_{16} = F \times 16^1 + 3 \times 16^0 = (243)_{10}$$

- Conversion from decimal to radix *r* system is carried out by separating the number into its integer and fraction parts and converting each partseparately
- Divide the integer successively by *r* and accumulate theremainders
- Multiply the fraction successively by *r* until the fraction becomeszero

Figure 3-1 Conversion of decimal 41.6875 into binary.

Integer = 41

41	
20	1
10	0
5	0
2	1
1	0
0	1

$(41)_{10} = (101001)_2$

Fraction = 0.6875

0.6875
2
1.3750
x 2
0.7500
x 2
1.5000
x 2
1.0000

$(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

- Each octal digit corresponds to three binarydigits
- Each hexadecimal digit corresponds to four binarydigits
- Rather than specifying numbers in binary form, refer to them in octal or hexadecimal and reduce the number of digits by 1/3 or ¼,respectively

1	2	7	5	4	3	Octal
1 0 1 0	1 1 1 1	0 1 1 0	0 0 1 1			Binary
A	F	6	3			Hexadecimal

Figure 3-2 Binary, octal, and hexadecimal conversion.

TABLE 3-1 Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	↑
1	001	1	
2	010	2	Code
3	011	3	for one
4	100	4	octal
5	101	5	digit
6	110	6	
7	111	7	↓
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

TABLE 3-2 Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	↑
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	Code
7	0111	7	for one
8	1000	8	hexadecimal
9	1001	9	digit
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	↓
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

- A binary code is a group of *n* bits that assume up to 2^n distinctcombinations
- A four bit code is necessary to represent the ten decimal digits – 6 areunused
- The most popular decimal code is called *binary-coded decimal*(BCD)
- BCD is different from converting a decimal number tobinary
- For example 99, when converted to binary, is1100011
- 99 when represented in BCD is 10011001

TABLE 3-3 Binary-Coded Decimal (BCD) Numbers

Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	↓
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

- The standard alphanumeric binary code isASCII
- This uses seven bits to code 128characters
- Binary codes are required since registers can hold binary informationonly

TABLE 3-4 American Standard Code for Information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

Section 3.2 – Complements

- Complements are used in digital computers for simplifying subtraction and logical manipulation
- Two types of complements for each base r system: r's complement and $(r-1)$'s complement
- Given a number N in base r having n digits, the $(r-1)$'s complement of N is defined as $(r^n - 1) - N$

- For decimal, the 9's complement of N is $(10^n - 1) - N$
- The 9's complement of 546700 is 999999 – 546700 =453299

- The 9's complement of 453299 is 999999 – 453299 =546700
- For binary, the 1's complement of N is $(2^n – 1) – N$
- The 1's complement of 1011001 is 1111111 – 1011001 =0100110
- The 1's complement is the true complement of the number – just toggle allbits

- The r's complement of an n-digit number N in base r is defined as $r^n – N$
- This is the same as adding 1 to the $(r – 1)$'scomplement
- The 10's complement of 2389 is 7610 + 1 =7611
- The 2's complement of 101100 is 010011 + 1 =010100

- Subtraction of unsigned n-digit numbers: $M – N$
  - Add M to the r's complement of N – this resultsin
    $M + (r^n – N) = M – N + r^n$
  - If $M \geq N$, the sum will produce an end carry r^nwhich isdiscarded
  - If $M < N$, the sum does not produce an end carry and is equalto $r^n – (N – M)$, which is the r's complement of $(N – M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Example: 72532 – 13250 = 59282. The 10's complement of 13250 is 86750.

M	= 72352
10's comp.ofN	=
+86750Sum	= 159282
Discardendcarry	= -
100000Answer	= 59282

Example for M < N: 13250 – 72532 = -59282

M	= 13250
10's comp.ofN	=
+27468Sum	= 40718
No end carry	
Answer	= -59282 (10's comp. of 40718)

Example for X = 1010100 and Y = 1000011

X	=1010100
2's comp. of Y	= +0111101
Sum	= 10010001
Discard end carry	= -10000000
Answer X – Y	= 0010001

Y	= 1000011
2's comp. of X	=+0101100
Sum	= 1101111

No end carry
Answer = -0010001 (2's comp. of1101111)

Section 3.3 – Fixed-Point Representation

- Positive integers and zero can be represented by unsignednumbers
- Negative numbers must be represented by signed numbers since + and – signs are not available, only 1's and 0's are
- Signed numbers have msb as 0 for positive and 1 for negative – msb is the signbit
- Two ways to designate binary point position in aregister
  - Fixed pointposition
  - Floating-point representation
- Fixed point position usually uses one of the two followingpositions
  - A binary point in the extreme left of the register to make it afraction
  - A binary point in the extreme right of the register to make it aninteger
  - In both cases, a binary point is not actuallypresent
- The floating-point representations uses a second register to designate the position of the binary point in the firstregister

- When an integer is positive, the msb, or sign bit, is 0 and the remaining bits represent themagnitude
- When an integer is negative, the msb, or sign bit, is 1, but the rest of the number can be represented in one of threeways
  - Signed-magnituderepresentation
  - Signed-1's complementrepresentation
  - Signed-2's complementrepresentation

- Consider an 8-bit register and the number+14
  - The only way to represent it is00001110
- Consider an 8-bit register and the number–14
  - Signedmagnitude: 1 0001110
  - Signed1'scomplement: 1 1110001
  - Signed2'scomplement: 1 1110010
- Typically use signed 2'scomplement

- Addition of two signed-magnitude numbers follow the normalrules
  - If same signs, add the two magnitudes and use the commonsign
  - Differing signs, subtract the smaller from the larger and use the sign of the largermagnitude
  - Must compare the signs and magnitudes and then either add orsubtract
- Addition of two signed 2's complement numbers does not require a comparison or subtraction – only addition andcomplementation
  - Add the two numbers, including their signbits
  - Discard any carry out of the sign bitposition
  - All negative numbers must be in the 2's complementform
  - If the sum obtained is negative, then it is in 2's complementform

+6	00000110	-6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111
+6	00000110	-6	11111010
-13	11110011	-13	11110011
-7	11111001	-19	11101101

- Subtraction of two signed 2's complement numbers is asfollows
  - Take the 2's complement form of the subtrahend (including signbit)
  - Add it to the minuend (including the signbit)
  - A carry out of the sign bit position isdiscarded

- An *overflow* occurs when two numbers of *n* digits each are added and the sum occupies *n* + 1digits
- Overflows are problems since the width of a register isfinite
- Therefore, a flag is set if this occurs and can be checked by theuser
- Detection of an overflow depends on if the numbers are signed orunsigned
- For unsigned numbers, an overflow is detected from the end carry out of themsb
- For addition of signed numbers, an overflow cannot occur if one is positive and one is negative – both have to have the samesign
- An overflow can be detected if the carry into the sign bit position and the carry out of the sign bit position are notequal

+70	0 1000110	-70	1 0111010
+80	0 1010000	-80	1 0110000
+150	1 0010110	-150	0 1101010

- The representation of decimal numbers in registers is a function of the binary code used to represent a decimaldigit
- A 4-bit decimal code requires four flip-flops for each decimaldigit
- This takes much more space than the equivalent binary representation and the circuits required to perform decimal arithmetic are morecomplex
- Representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary
- Either signed magnitude or signed complementsystems
- The sign of a number is represented with fourbits
  - 0000 for +
  - 1001 for –
- To obtain the 10's complement of a BCD number, first take the 9's complement and then add one to the least significantdigit

- Example: (+375) + (-240) =+135

0	375	$(0000\ 0011\ 01111010)_{BCD}$
+9	760	$(1001\ 0111\ 01100000)_{BCD}$
0	135	$(0000\ 0001\ 00110101)_{BCD}$

Section 3.4 – Floating-Point Representation

- The floating-point representation of a number has twoparts
- The first part represents a signed, fixed-point number – the*mantissa*
- The second part designates the position of the binary point – the*exponent*
- The mantissa may be a fraction or aninteger
- Example: the decimal number +6132.789is
  - Fraction: +0.6123789
  - Exponent: +04
  - Equivalent to $+0.6132789 \times 10^{+4}$
- A floating-point number is always interpreted to represent $m \times r^e$
- Example: the binary number +1001.11 (with 8-bit fraction and 6-bitexponent)
  - Fraction: 01001110
  - Exponent: 000100
  - Equivalent to $+(.1001110)_2 \times 2^{+4}$
- A floating-point number is said to be *normalized* if the most significant digit of the mantissa isnonzero
- The decimal number 350 is normalized, 00350 isnot
- The 8-bit number 00011010 is notnormalized
- Normalize it by fraction = 11010000 and exponent =-3
- Normalized numbers provide the maximum possible precision for the floating-pointnumber

Section 3.5 – Other Binary Codes

- Digital systems can process data in discrete formonly
- Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter
- The reflected binary or *Gray code*, is sometimes used for the converted digital data
- The Gray code changes by only one bit as it sequences from one number to the next
- Gray code counters are sometimes used to provide the timing sequencesthat control the operations in a digitalsystem

TABLE 3-5 4-Bit Gray Code

Binary code	Decimal equivalent	Binary code	Decimal equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

- Binary codes for decimal digits require a minimum of fourbits
- Other codes besides BCD exist to represent decimaldigits

TABLE 3-6 Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
	1010	0101	0000	0000
Unused	1011	0110	0001	0001
bit	1100	0111	0010	0011
combi-	1101	1000	1101	1000
nations	1110	1001	1110	1001
	1111	1010	1111	1011

- The 2421 code and the excess-3 code are both*self-complementing*
- The 9's complement of each digit is obtained by complementing each bit in the code
- The 2421 code is a *weightedcode*
- The bits are multiplied by indicated weights and the sum gives the decimaldigit
- The excess-3 code is obtained from the corresponding BCD code added to3

Section 3.6 – Error Detection Codes

- Transmitted binary information is subject to noise that could change bits 1 to 0 and vice versa
- An *error detection code* is a binary code that detects digital errors during transmission
- The detected errors cannot be corrected, but can prompt the data tobe retransmitted
- The most common error detection code used is the *paritybit*
- A parity bit is an extra bit included with a binary message to make the total number of 1's either odd oreven

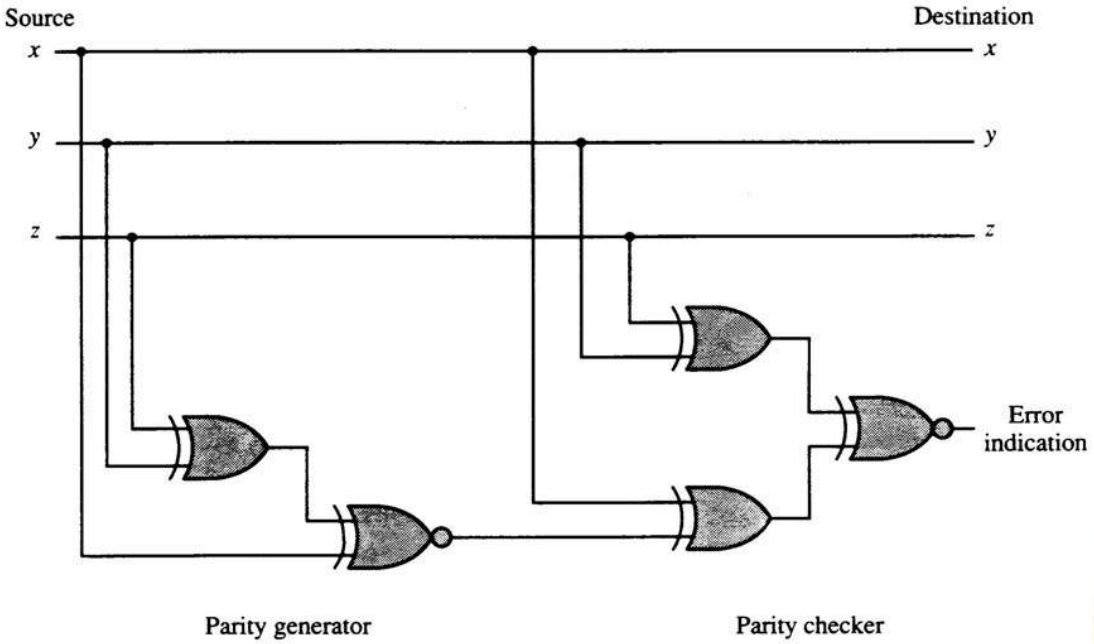
TABLE 3-7 Parity Bit Generation

Message *xyz*	*P*(odd)	*P*(even)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

- The P(odd) bit is chosen to make the sum of 1's in all four bitsodd
- The even-parity scheme has the disadvantage of having a bit combination of all 0's
- Procedure duringtransmission:
    - At the sending end, the message is applied to a *paritygenerator*
    - The message, including the parity bit, istransmitted
    - At the receiving end, all the incoming bits are applied to a *paritychecker*
    - Any odd number of errors are detected

- Parity generators and checkers are constructed with XOR gates (oddfunction)
- An odd function generates 1 iff an odd number if input variables are1

Figure 3-3 Error detection with odd parity bit.



REGISTER TRANSFER AND MICROOPERATIONS

- ✓ Register TransferLanguage
- ✓ RegisterTransfer
- ✓ Bus And MemoryTransfers
- ✓ Types ofMicro-operations
- ✓ ArithmeticMicro-operations
- ✓ LogicMicro-operations
- ✓ ShiftMicro-operations
- ✓ Arithmetic Logic ShiftUnit

BASIC DEFINITIONS:

- A digital system is an interconnection of digital hardwaremodules.
- The modules are registers, decoders, arithmetic elements, and controllogic.
- 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 inthem.
- The operations executed on data stored in registers are called*microoperations*.
- A *microoperation*is an elementary operation performed 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 anotherregister.
- Examples of microoperations are shift, count, clear, andload.
- The internal hardware organization of a digital computer is best defined byspecifying:

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

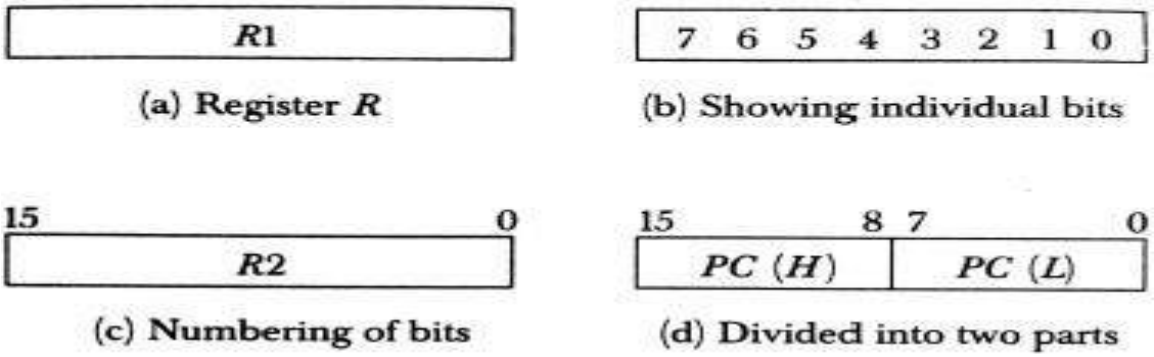
REGISTER TRANSFER LANGUAGE:

- The symbolic notation used to describe the micro-operation transfer among registers is called RTL (Register TransferLanguage).
- The use of ***symbols*** instead of a ***narrative explanation*** provides an organized and concise manner for listing the micro-operation sequences in registers and the control functions that initiatethem.
- A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digitalmodule.
- It is a convenient tool for describing the internal organization of digital computers in concise and precise manner.

Registers:

- Computer registers are designated by upper case letters (and optionally followed by digits or letters) to denote the function of theregister.
- For example, the register that holds an address for the memory unit 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 theleft.
- Figure 4-1 shows the representation of registers in block diagramform.

Figure 4-1 Block diagram of register.



- The most common way to represent a register is by a rectangular box with the name ofthe register inside, as in Fig.4-1(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).
- 16-bitregisterispartitionedintotwopartsin (d).Bits0through7areassignedthesymbolL(for low byte) and bits 8 through 15 are assigned the symbol *H* (for highbyte).
- 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-orderbyte**.**

Register Transfer:

- Information transfer from one register to another is designated in symbolic form by means ofa *replacement operator*.
- Thestatement**R2**←**R1**denotesatransferofthecontentofregisterR1intoregisterR2.
- It designates a replacement of the content of R2 by the content ofR1.
- By definition, the content of the source register R 1 does not change after thetransfer.
- If we want the transfer to occur only under a predetermined control condition then it can be shown by an if-thenstatement.

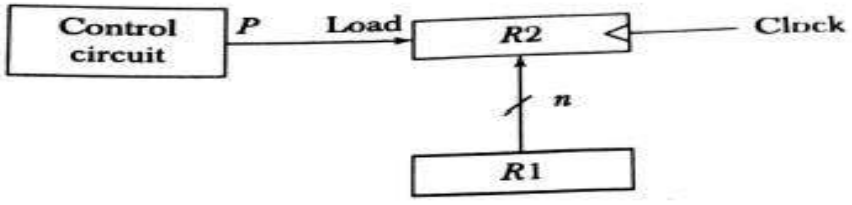
if (P=1) then R2← R1

- P is the control signal generated by a controlsection.
- We can separate the control variables from the register transfer operation by specifying a ***Control Function.***
- Control function is a Boolean variable that is equal to 0 or1.
- control function is included in the statementas

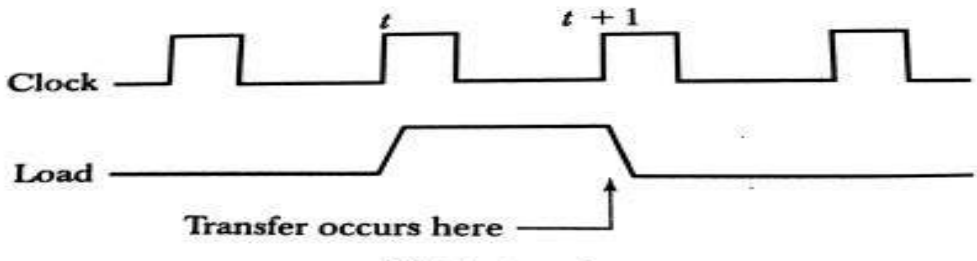
P: R2← R1

- Control condition is terminated by a colon implies transfer operation be executed by the hardware only ifP=1.
- Every statement written in a register transfer notation implies a hardware construction for implementing thetransfer.
- Figure 4-2 shows the block diagram that depicts the transfer from R1 toR2.

Figure 4-2 Transfer from R1 to R2 when p = 1.



(a) Block diagram



(b) Timing diagram

- The n outputs of register R1 are connected to the n inputs of registerR2.
- 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 isknown.
- Register R2 has a load input that is activated by the control variableP.
- It is assumed that the control variable is synchronized with the same clock as the one applied to theregister.
- 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 remainsactive.
- Even though the control condition such as P becomes active just after time *t*, the actual transfer doesnotoccuruntiltheregisteristriggeredbythenext positivetransitionoftheclockattime $t+1$.
- The basic symbols of the register transfer notation are listed in belowtable

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

- A comma is used to separate two or more operations that are executed at the sametime.
- Thestatement

T:R2←R1,R1←R2 (exchangeoperation)

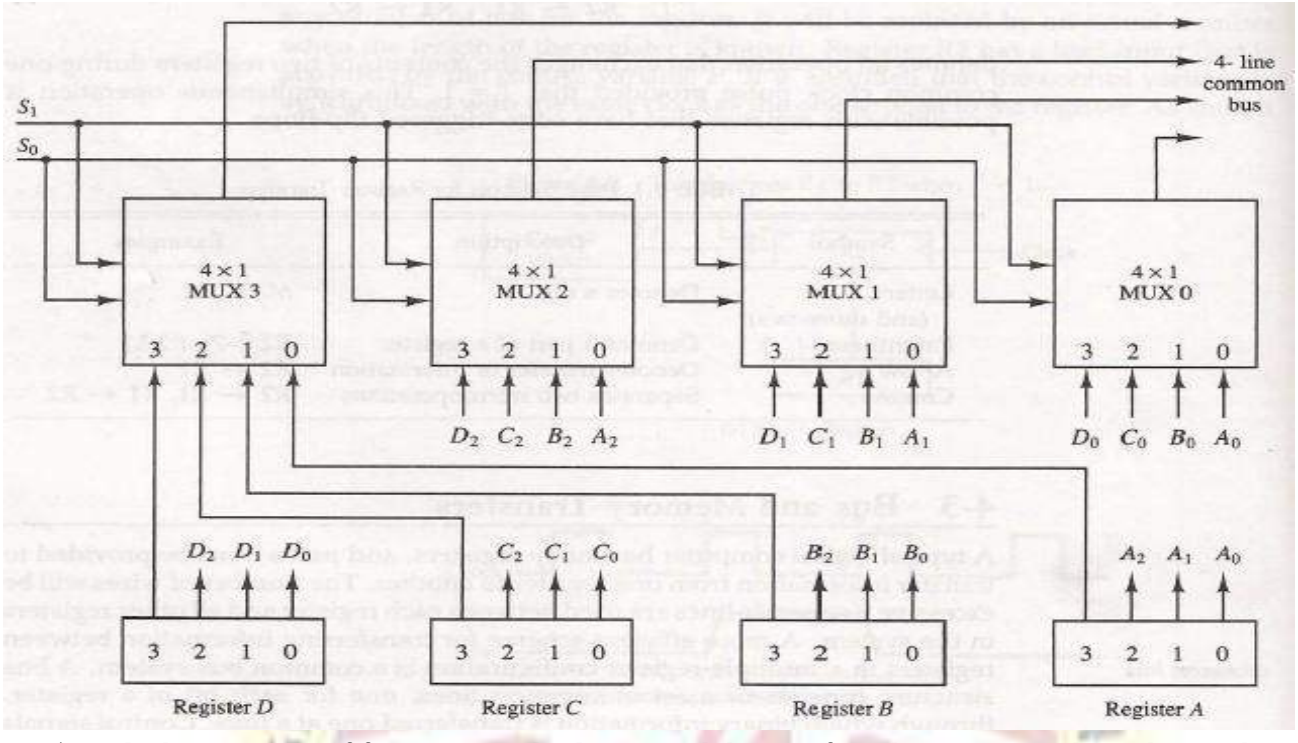
denotes an operation that exchanges the contents of two rgisters during one common clock pulse provided that T=1.

Bus and Memory Transfers:

- A more efficient scheme for transferring information between registers in ***a multiple-register configuration*** is a ***Common BusSystem.***
- A common bus consists of a set of common lines, one for each bit of aregister.
- Controlsignalsdeterminewhichregisterisselectedbythebusduringeachparticularregister transfer.
- Different ways of constructing a Common BusSystem
  - ✓ UsingMultiplexers
  - ✓ Using Tri-state Buffers

Common bus system is with multiplexers:

- The multiplexers select the source register whose binary information is then placed on thebus.
- The construction of a bus system for four registers is shown in belowFigure.



- The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S_1 andS_0.
- For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labelled 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 thebus.
- 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 twobits.
- The two selection lines Si and So are connected to the selection inputs of all fourmultiplexers.
- The selection lines choose the four bits of one register and transfer them into the four-line commonbus.
- When S_1S_0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form thebus.
- This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of themultiplexers.
- Similarly, register *B* is selected if S_1S_0 = 01, and soon.
- Table 4-2 shows the register that is selected by the bus for each of the four possible binary value of the selectionlines.

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

- In general a bus systemhas
  - ✓ multiplex "k"Registers
  - ✓ eachregisterof"n"bits
  - ✓ toproduce"n-linebus"
  - ✓ no. of multiplexers required =n
  - ✓ size of each multiplexer = k x 1
- When the bus is includes in the statement, the register transfer is symbolized asfollows:

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 directtransfer.

R1← C

Three-State Bus Buffers:

- A bus system can be constructed with three-state gates instead ofmultiplexers.
- A three-state gate is a digital circuit that exhibits threestates.
- Two of the states are signals equivalent to logic 1 and 0 as in a conventionalgate.
- The third state is a *high-impedancestate.*
- The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logicsignificance.
- Becauseofthisfeature,alargenumberofthree-stategateoutputscanbeconnectedwithwires toformacommonbuslinewithoutendangeringloadingeffects.
- The graphic symbol of a three-state buffer gate is shown in Fig.4-4.

Figure 4-4 Graphic symbols for three-state buffer.

- It is distinguished from a normal buffer by having both a normal input and a controlinput.
- 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 goes to a high-impedance state, regardless of the value in the normalinput.
- The construction of a bus system with three-state buffers is shown in Fig.4

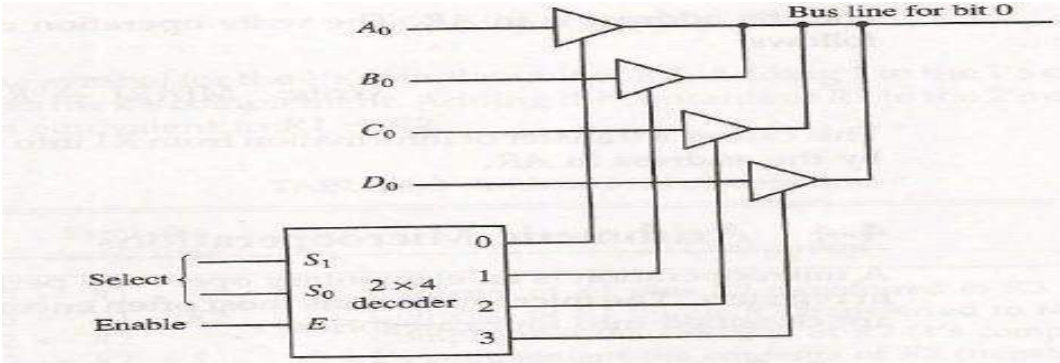


Figure 4-5 Bus line with three state-buffers.

- The outputs of four buffers are connected together to form a single busline.
- Thecontrolinputstothebuffersdeterminewhichofthefournormalinputswillcommunicatewith the busline.
- 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 has access to the bus line while all other buffers are maintained in a high impedancestate.
- 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 thediagram.
- When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers aredisabled.
- 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 thedecoder.

Memory Transfer:

- The transfer of information from a memory word to the outside environment is called a*read* operation.
- The transfer of new information to be stored into the memory is called a *write*operation.
- A memory word will be symbolized by the letterM.
- The particular memory word among the many available is selected by the memory address during thetransfer.
- It is necessary to specify the address of M when writing memory transferoperations.
- This will be done by enclosing the address in square brackets following the letterM.
- Consider a memory unit that receives the address from a register, called the address register, symbolized byAR.
- The data are transferred to another register, called the data register, symbolized byDR.
- The read operation can be stated asfollows:

Read: DR<- M [AR]

- This causes a transfer of information into DR from the memory word M selected by the address in AR.
- The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is inAR.
- The write operation can be stated asfollows:

Write: M [AR] <- R1

Types of Micro-operations:

- *Register Transfer Micro-operations:*Transfer binary information from one register toanother.

- *Arithmetic Micro-operations:*Perform arithmetic operation on numeric data stored inregisters.
- *Logical Micro-operations:*Perform bit manipulation operations on data stored inregisters.
- *Shift Micro-operations:*Perform shift operations on data stored inregisters.

- RegisterTransferMicro-operationdoesn'tchangetheinformationcontentwhenthebinary information moves from source register to destinationregister.
- Other three types of micro-operations change the information change the information content during thetransfer.

Arithmetic Micro-operations:

- The basic arithmetic micro-operationsare
  - Addition
  - Subtraction
  - Increment
  - Decrement
  - Shift
- The arithmetic Micro-operation defined by the statement below specifies the add micro-operation.

$$R3 \leftarrow R1 + R2$$

- It states that the contents of R1 are added to contents of R2 and sum is transferred toR3.
- To implement this statement hardware requires 3 registers and digital component that performs addition
- Subtraction is most often implemented through complementation andaddition.
- The subtract operation is specified by the followingstatement

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

- instead of minus operator, we can write as
- $\overline{R2}$isthesymbolforthe1'scomplementofR2
- Adding1to1'scomplementproduces2'scomplement
- Addingthecontentsof*R1*tothe2'scomplementofR2isequivalenttoR1-R2.

Binary Adder:

- Digitalcircuitthatformsthearithmeticsumof2bitsandthepreviouscarryiscalled**FULLADDER.**
- Digital circuit that generates the arithmetic sum of 2 binary numbers of any lengths iscalled **BINARY ADDER.**
- Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binaryadder.

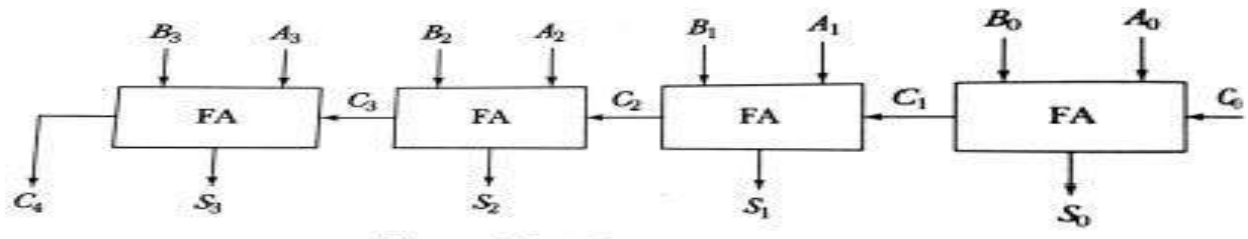


Figure 4-6 4-bit binary adder.

- 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-orderbit.

- The carries are connected in a chain through the full-adders. The input carry to the binaryadderisCoandtheoutputcarryisC4.TheSoutputsofthefull-addersgenerate the required sumbits.
- An n-bit binary adder requires nfull-adders.

Binary Adder –Subtractor:

- The addition and subtraction operations can be combined into one common circuit by includingan exclusive-OR gate with eachfull-adder.
- A 4-bit adder-subtractor circuit is shown in Fig.4-7.

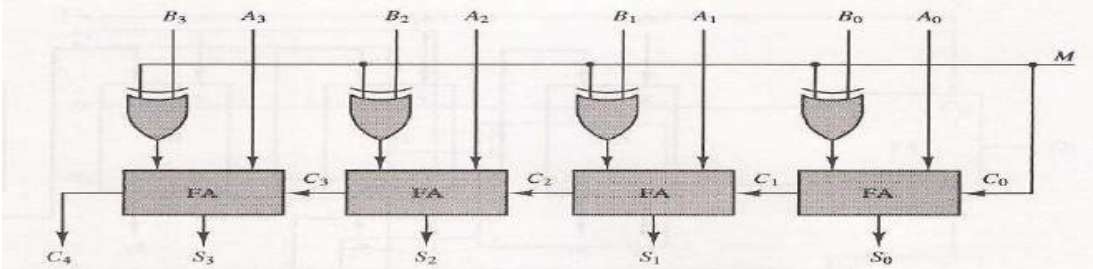


Figure 4-7 4-bit adder-subtractor.

- The mode input M controls the operation. When M = 0 the circuit is an adder and when M = 1 the circuit becomes asubtractor.
- Each exclusive-OR gate receives input M and one of the inputs ofB
- When M = 0, we have *B* xor 0 = B. The full-adders receive the value of *B*, the input carry is 0, and the circuit performs A plusB.
- When M = 1, we have *B xor 1 = B'* and Co =1.
- The *B* inputs are all complemented and a 1 is added through the inputcarry.
- The circuit performs the operation A plus the 2's complement of*B*.

Binary Incrementer:

- The increment microoperation adds one to a number in aregister.
- For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it isincremented.
- This can be accomplished by means of half-adders connected incascade.
- The diagram of a 4-bit 'combinational circuit incrementer is shown in Fig.4-8.

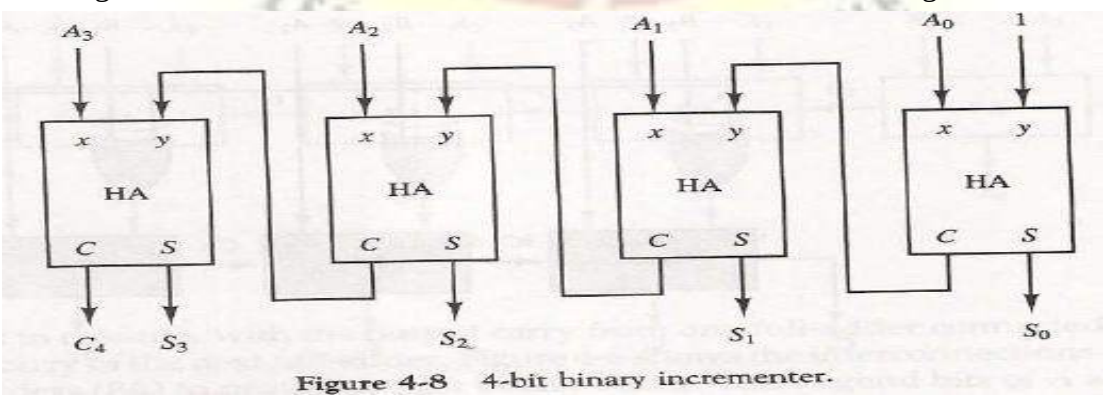


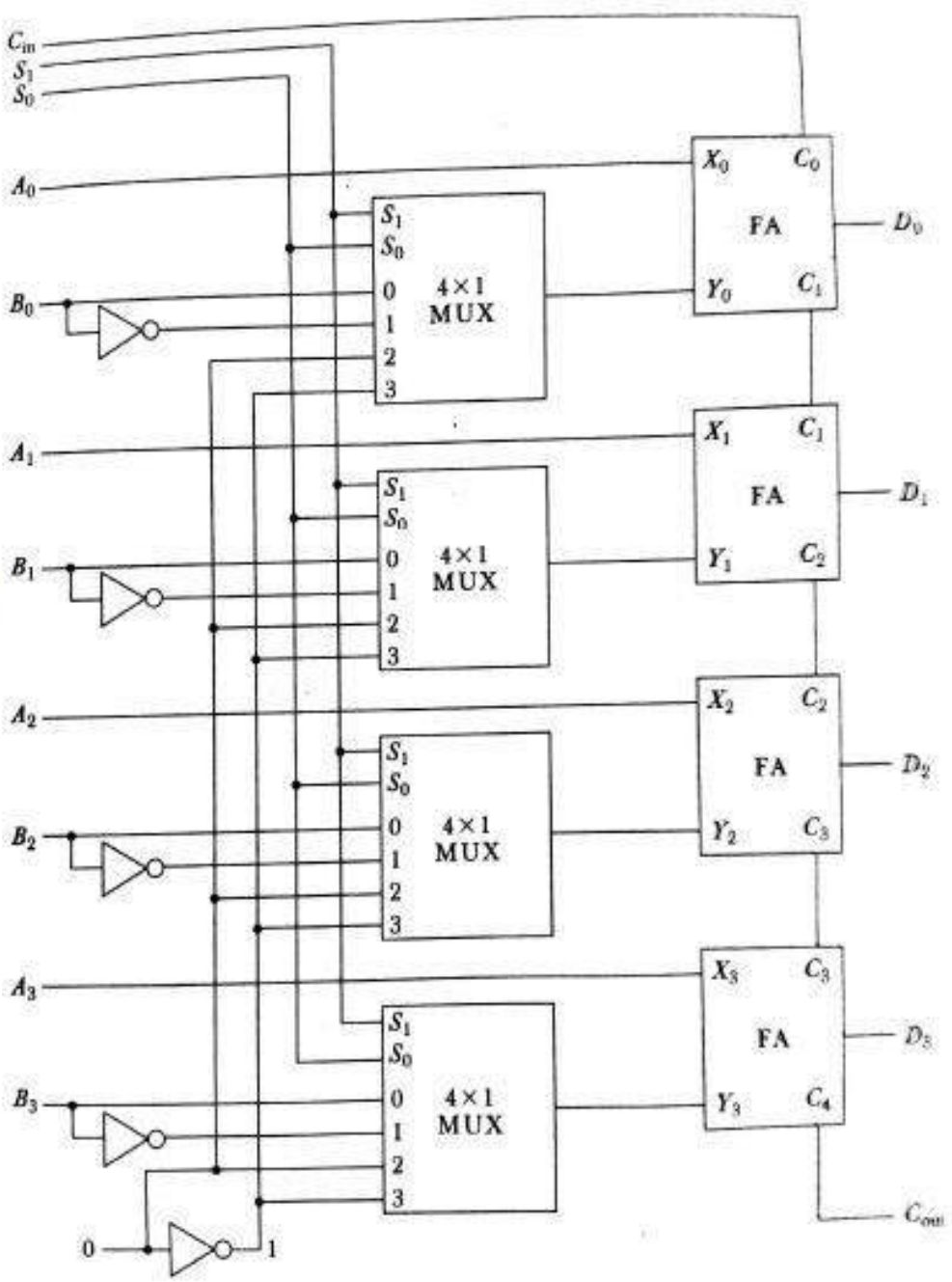
Figure 4-8 4-bit binary incrementer.

- 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 beincremented.
- 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 throughS_3.

- The output carry C_4will be 1 only after incrementing binary 1111. This also causes outputs S_0 through S_3 to go to0.
- The circuit of Fig. 4-8 can be extended to an *n* -bit binary incrementer by extending the diagram to include *n*half-adders.
- The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previousstage.

Arithmetic Circuit:

- The basic component of an arithmetic circuit is the paralleladder.
- 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. 4-9. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing differentoperations.



- 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 binaryadder.
- Each of the four inputs from B are connected to the data inputs of themultiplexers.
- The multiplexers data inputs also receive the complement ofB.
- The other two data inputs are connected to logic-0 andlogic-1.
- 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 thenext.
- By controlling the value of Y with the two selection inputs $S_1$and S_0 and making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table44.

TABLE 4-4 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	$\bar{B}$	$D = A + \bar{B}$	Subtract with borrow
0	1	1	$\bar{B}$	$D = A + \bar{B} + 1$	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

Addition:

- When S_1S_0= 00, the value of *B* is applied to the Y inputs of theadder.
  - ✓ If Cir, = 0, the output *D=A+B*.
  - ✓ If Cin = 1, output *D=A+B +1*.
- Both cases perform the add microoperation with or without adding the inputcarry.

Subtraction:

- When S_1S_0 = 01, the complement of B is applied to the Y inputs of theadder.
  - ✓ If C_{in} = 1, then D = A $\bar{+}$ B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtraction of A-B.
  - ✓ When C_{in} = 0 then D = A $\bar{+}$ B. This is equivalent to a subtract with borrow, that is, A-B-1.

Increment:

- When S_1S_0 = 10, the inputs from *B* are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes D = A + 0 + C_{in}. This gives *D* = A when C_{in} = 0 and D = A + 1 when C_{in}= 1.
- In the first case we have a direct transfer from input A to outputD.
- In the second case, the value of A is incremented by1.

Decrement:

- When S_1S_0= 11, all l's are inserted into the Y inputs of the adder to produce the decrement operation $D = A$ -1 when C_{in} =0.
- This is because a number with all 1's is equal to the 2's 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 complementof1=A—1.WhenC_{in}=1,then$D=A$ -1+1=A,whichcausesadirecttransferfrom input A to output D.

Logic Micro-operations:

- Logic microoperations specify binary operations for strings of bits stored inregisters.
- These operations consider each bit of the register separately and treat them as binaryvariables.
- For example, the exclusive-OR microoperation with the contents of two registers RI and R2 is symbolized by thestatement

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

- It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P =1.

List of Logic Microoperations:

- There are 16 different logic operations that can be performed with two binaryvariables.
- They can be determined from all possible truth tables obtained with two binary variables as shown in Table4-5.

TABLE 4-5 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

- The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table4-6.
- The 16 logic microoperations are derived from these functions by replacing variable x bythe binary content of register A and variable y by the binary content of registerB.
- The logic micro-operations listed in the second column represent a relationship between the binary content of two registers A andB.

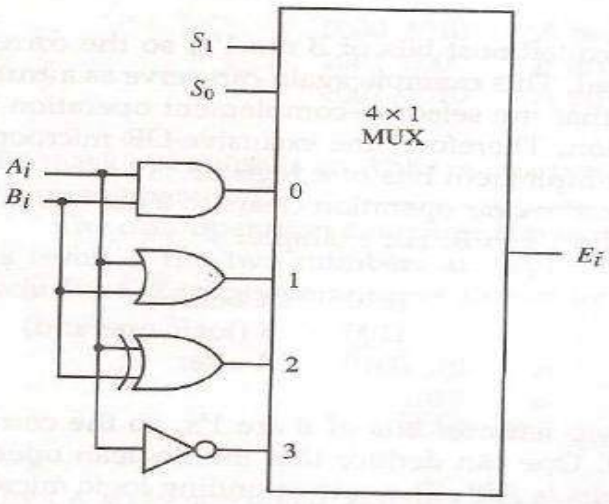
TABLE 4-6 Sixteen Logic Microoperations

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

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 logicfunction.
- Although there are 16 logic microoperations, most computers use only four--AND, OR, XOR (exclusive-OR), and complement from which all others can bederived.
- Figure 4-10 shows one stage of a circuit that generates the four basic logicmicrooperations.
- It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the requiredlogic.
- The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputsS_1andS_0chooseoneofthedatainputsofthemultiplexeranddirectitsvaluetotheoutput.

Figure 4-10 One stage of logic circuit.



(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

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 bits values into aregister.
- The following example shows how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated byB).
- Selectiveset
  - ✓The *selective-set* operation sets to 1 the bits in register A where there are correspondingl'sinregisterB.ItdoesnotaffectbitpositionsthathaveO'sinB.Thefollowing numerical example clarifies thisoperation:

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

  - ✓ The OR microoperation can be used to selectively set bits of aregister.

- Selectivecomplement
  - ✓ These*lective-complement*operationcomplementsbitsinAwheretherearecorresponding 1's in B. It does not affect bit positions that have 0's in *B*. Forexample:

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

  - ✓ The exclusive-OR microoperation can be used to selectively complement bits of aregister.
- Selectiveclear
  - ✓ The *selective-clear* operation clears to 0 the bits in Aonly where there are corresponding l's in B. Forexample:

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

  - ✓ The corresponding logic microoperationis $A \leftarrow A \wedge \overline{B}$
- Mask
  - ✓ The *mask* operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding O's in B . The mask operation is an AND micro operation as seen from the following numericalexample:

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

- Insert
  - ✓ 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 requiredvalue.

  - ✓ 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 unwantedbits:

```
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.

- Clear
  - ✓ 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 followingexample

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

Shift Microoperations:

- Shift microoperations are used for serial transfer ofdata.
- The contents of a register can be shifted to the left or theright.
- Duringashift-leftoperationtheserialinputtransfersabitintotherightmostposition.
- Duringashift-rightoperationtheserialinputtransfersabitintotheleftmostposition.
- There are three types of shifts: logical, circular, andarithmetic.
- The symbolic notation for the shift microoperations is shown in Table4-7.

TABLE 4-7 Shift Microoperations

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

- **LogicalShift:**
  - A *logical* shift is one that transfers 0 through the serialinput.
  - The symbols *shl*and shr for logical shift-left and shift-rightmicrooperations.

- The microoperations that specify a 1-bit shift to the left of the content of register R and a 1-bit shift to the right of the content of register R shown in table4.7.
- The bit transferred to the end position through the serial input is assumed to be 0 during a logicalshift.

➢ CircularShift:

- The *circular* shift (also known as a *rotate* operation) circulates the bits of the register around the two ends without loss ofinformation.
- This is accomplished by connecting the serial output of the shift register to its serialinput.
- We will use the symbols *cil*and *cir*for the circular shift left and right,respectively.

➢ ArithmeticShift:

- 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 by2.
- An arithmetic shift-right divides the number by2.
- Arithmetic shifts must leave the sign bit unchanged because the sign of thenumber remains the same when it is multiplied or divided by2.

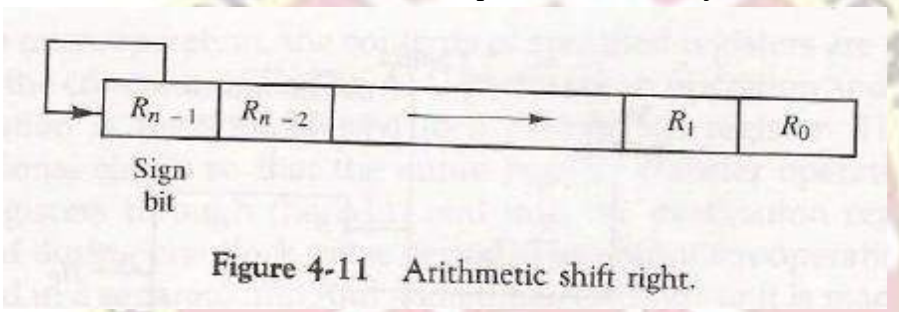
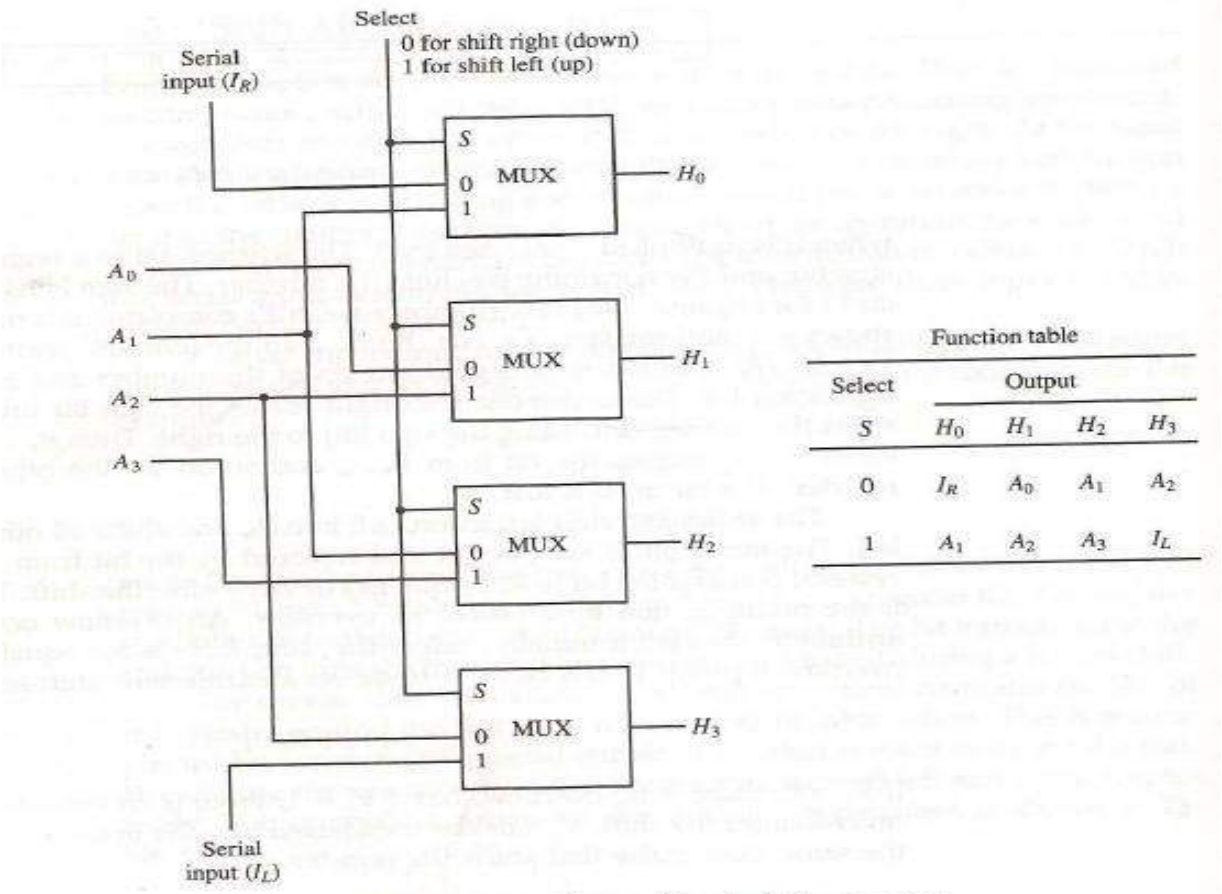


Figure 4-11 Arithmetic shift right.

Hardware Implementation:

- A combinational circuit shifter can be constructed with multiplexers as shown in Fig.4-12.
- The4-bitshifterhasfourdatainputs,A_0throughA_3,andfourdataoutputs,H_0throughH_3.
- There are two serial inputs, one for shift left (I_L) and the other for shift right(I_R).
- When the selection input *S=0* the input data are shifted right (down in thediagram).
- When *S* = 1, the input data are shifted left (up in thediagram).
- The function table in Fig. 4-12 shows which input goes to each output after theshift.
- A shifter with n data inputs and outputs requires nmultiplexers.
- Thetwoserialinputscanbecontrolledbyanothermultiplexertoprovidethethreepossibletypes of shifts.



Function table

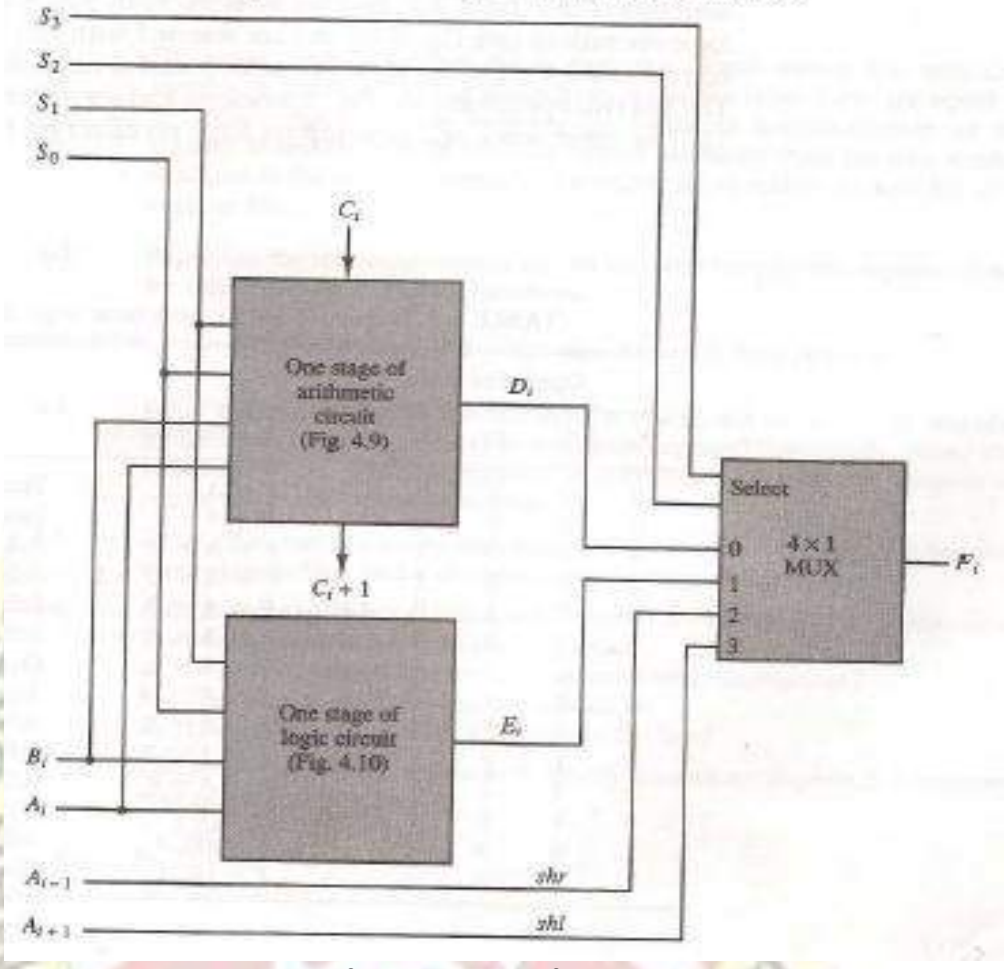
Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4-12 4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit:

- Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviatedALU.
- The ALU is a combinational circuit so that the entire register transfer operation fromthe source registers through the ALU and into the destination register can be performed during one clock pulse period.
- The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overallALU.
- The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 4- 13.
- Particular microoperation is selected with inputs S_1 and S_0. A 4 x 1 multiplexer at the output chooses between an arithmetic output in $D_i$and a logic output inE_i.
- The data in the multiplexer are selected with inputs $S_3$and S_2. The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and A_{i+1} for the shift-leftoperation.
- The circuit whose one stage is specified in Fig. 4-13 provides eight arithmetic operation, four logic operations, and two shiftoperations.
- Each operation is selected with the five variables S_3, S_2, S_1, S_0 andC_{in}.
- The input carry C_{in} is used for selecting an arithmetic operationonly.

Figure 4-13 One stage of arithmetic logic shift unit.



- Table 4-8 lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with S_3S_2 =00.
- The next four are logic and are selected with S_3S_2 =01.
- Theinputcarryhasnoeffectduringthelogicoperationsandismarkedwithdon't-carex's.
- The last two operations are shift operations and are selected with S_3S_2= 10 and11.

- The other three selection inputs have no effect on theshift.

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

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

UNIT - II

Central processing unit: General register and stack organizations- instruction formats - Addressing modes- Data transfer and manipulation - program control- RISC - Pipelining - Arithmetic and instruction- RISC pipeline - Vector processing and Array processors.

Components of CPU and their functions:

CPU or **Central processing unit** is the brain of the computer system. A function of CPU varies from data processing to controlling input-output devices. Each and every instruction no matter how complex or simple, it has to go through the CPU. In this article we will learn various components of CPU and their functions.

The central processing unit is also responsible for storing data or information, intermediate results and instructions in the memory system. It also controls the operations of all other parts of the computer system.

Functions of a CPU:

CPU generally performs the arithmetical and logical operations, controlling of different input-output devices. These operations are performed based on some predefined algorithms and instructions normally referred as computer programs. A computer program is a set of instructions written by a human to perform a specific operation by the CPU. A computer program is normally stored in the memory unit of the Central Processing Unit.

A CPU mainly consists of ALU (Arithmetic & Logic Unit), Control Unit and Memory Unit. These 3 units are the primary components of a CPU. Various functions of CPU and operations are generally performed by these 3 units are described below.

Components of CPU and their functions :

Memory unit(storage component):

The primary job of the **memory unit** is to store data or instructions and intermediate results. Memory unit supplies data to the other units of a CPU. In Computer Organization, memory

can be divided into two major parts primary memory and secondary memory. Speed and power and performance of a memory depends on the size and type of the memory.
When an instruction is processed by the central processing unit, the main memory or the RAM (Random Access Memory) stores the final result before it is sent to the output device. All inputs and outputs are intermediate and are transmitted through the main memory.

Control unit (Control Component)

It is the unit which controls all the operations of the different units but does not carry out any actual data processing operation. **Control unit** transfers data or instruction among different units of a computer system. It receives the instructions from the memory, interprets them and sends the operation to various units as instructed.
Control unit is also responsible for communicating with all input and output devices for transferring or receiving the instruction from the storage units. So, the control unit is the main coordinator since it sends signals and find the sequence of instructions to be executed.

Arithmetic and logic unit(Execution Component)

ALU can also be subdivided into 2 sections namely, **arithmetic unit** and **logic unit**. It is a complex digital circuit which consists of registers and which performs arithmetic and logical operations. Arithmetic sections perform arithmetic operations like addition, subtraction, multiplication, division etc. All other Complex operations can also be performed by repetition of these above basic operations.
The logic unit is responsible for performing logical operations such as comparing, selecting, matching and merging of different data or information.
So basically ALU is the major part of the computer system which handles different calculations. Depending on the design of ALU it makes the CPU more powerful and efficient.

A decoder is a combinational logic circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs. They are used in a wide variety of applications, including data demultiplexing, seven segment displays, and memory address **decoding**.

A mutliplexer (**Mux**) is a device used to select a single line of input from multiple input lines using control signals. In this diagram, D0 to D3 are input data lines and Y is the output.

General Register organization

Generally CPU has seven general registers. Register organization show how registers are selected and how data flow between register and ALU. A decoder is used to select a particular register. The output of each register is connected to two multiplexers to form the two buses A and B. The selection lines in each multiplexer select the input data for the particular bus.

The A and B buses form the two inputs of an ALU. The operation select lines decide the micro operation to be performed by ALU. The result of the micro operation is available at the output bus. The output bus connected to the inputs of all registers, thus by selecting a destination register it is possible to store the result in it.
Introduction:

- The main part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as theCPU.

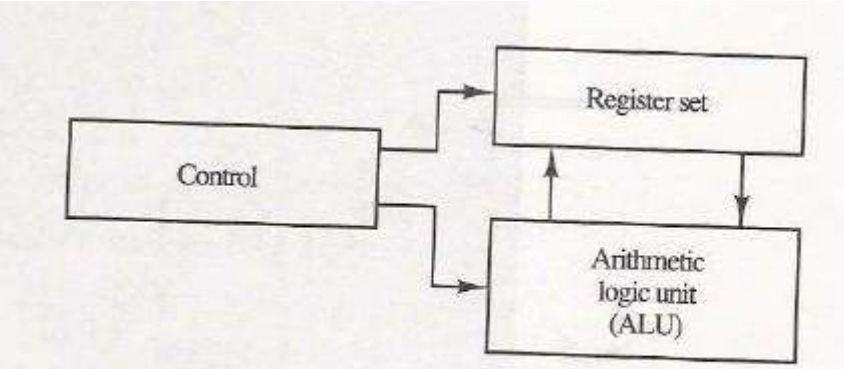


Figure 8-1 Major components of CPU.

- The CPU is made up of three major parts, as shown in Fig.8-1.
- The register set stores intermediate data used during the execution of theinstructions.
- The arithmetic logic unit (ALU) performs the required microoperations for executing theinstructions.
- The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation toperform.

StackOrganization:

A stack or last-in first-out (LIFO) is useful feature that is included in the CPU of mostcomputers.

Stack:

- A stack is a storage device that stores information in such a manner that the item stored last is the first itemretrieved.
- The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be takenoff.
- In the computer stack is a memory unit with an address register that can count the addressonly.
- The register that holds the address for the stack is called a stack pointer (SP). It always points at the top item in the stack.
- The two operations that are performed on stack are the insertion anddeletion.
- The operation of insertion is called*PUSH*.
- The operation of deletion is called*POP*.
- These operations are simulated by incrementing and decrementing the stack pointer register(SP).

Register Stack:

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words orregisters.

The below figure shows the organization of a 64-word registerstack.

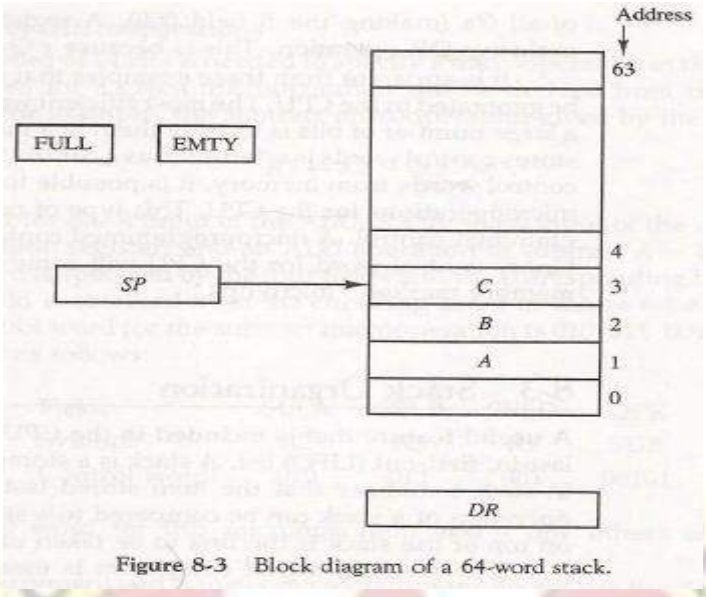


Figure 8-3 Block diagram of a 64-word stack.

- The stack pointer register SP contains a binary number whose value is equal to the address of the word is currently on top of the stack. Three items are placed in the stack: A, *B, C, in* thatorder.
- In above figure C is on top of the stack so that the content of *SP* is3.
- For removing the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of stackSP.
- Now the top of the stack is B, so that the content of SP is2.
- Similarly for inserting the new item, the stack is pushed by incrementing SP and writing a word in the next- higher location in thestack.
- In a 64-word stack, the stack pointer contains 6 bits because 2^6 =64.
- Since *SP* has only six bits, it cannot exceed a number greater than 63 (111111 inbinary).
- When 63 is incremented by 1, the result is 0 since 111111 + 1. = 1000000 in binary, but SP can accommodate only the six least significantbits.
- Then the one-bit register FULL is set to 1, when the stack isfull.
- Similarly when 000000 is decremented by 1, the result is 111111, and then the one-bit register EMTY is set 1 when the stack is empty ofitems.
- DR is the data register that holds the binary data to be written into or read out of thestack.

PUSH:

Initially, *SP* is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and notfull.

If the stack is not full (if FULL = 0), a new item is inserted with a pushoperation.

- The push operation is implemented with the following sequence ofmicrooperations:

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

- The stack pointer is incremented so that it points to the address of next-higherword.
- A memory write operation inserts the word from DR the top of thestack.
- The first item stored in the stack is at address1.
- The last item is stored at address0.
- If *SP* reaches 0, the stack is full of items, so FULL is to1.
- This condition is reached if the top item prior to the last push way location 63 and, after incrementing *SP*, the last item is stored in location0.
- Once an item is stored in location 0, there are no more empty registers in the stack, so the EMTY is cleared to0.

POP:

A new item is deleted from the stack if the stack is not empty (if EMTY =0).

- The pop operation consists of the following sequence of minoperations:

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

- The top item is read from the stack intoDR.
- The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set1.
- This condition is reached if the item read was in location 1. Once this it is read out, SP is decremented and reaches the value 0, which is the initial value ofSP.
- If a pop operation reads the item from location 0 and then is decremented, *SP* changes to 111111, which is equivalent to decimal 63 in above configuration, the word in address 0 receives the last item in thestack.

Memory Stack:

In the above discussion a stack can exist as a stand-alone unit. But in the CPU implementation of a stack is done by assigning a portion of memory to a stack operation and using a processor register as stackpointer.

The below figure shows a portion computer memory partitioned into three segments: program, data, and stack.

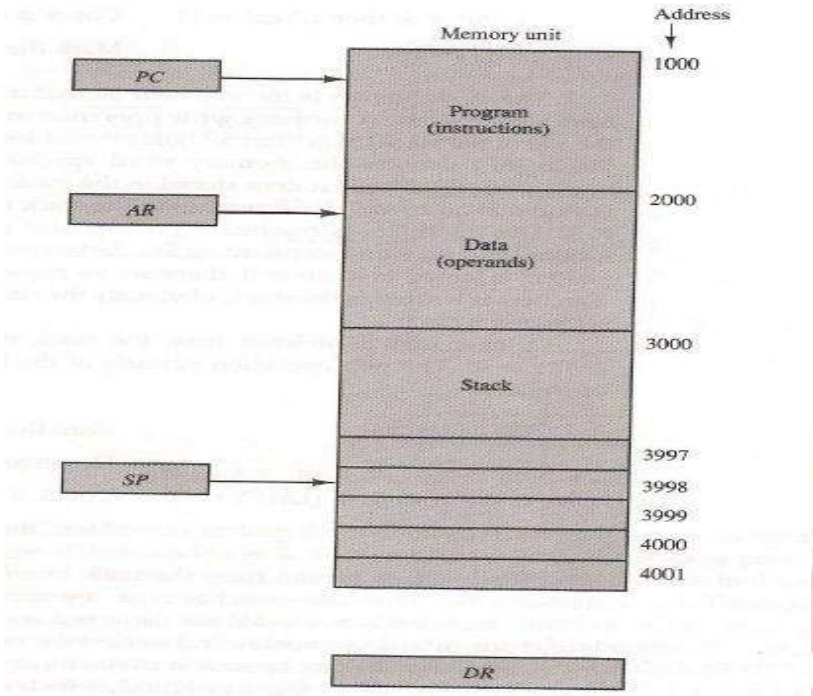


Figure 8-4 Computer memory with program, data, and stack segments.

- The program counter*PC* points at the address of the next instruction inprogram.
- The address register AR points at an array ofdata.
- The stack pointer SP points at the top of thestack.
- The three registers are connected to a common address bus, and either one can provide an address for memory.
    - PC is used during the fetch phase to read aninstruction.
    - AR is used during the exec phase to read anoperand.
    - *SP* is used to push or pop items into or fromstack.

As shown in Fig. 8-4, the initial value of SP is 4001 and the stack grows with decreasingaddresses.
Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is3000. No provisions are available for stack limitchecks. The items in the stack communicate with a data register *DR.*

A new item is inserted with the push operation as follows:

SP← SP-1

M [SP] ← DR

- The stack pointer is decremented so that it points at the address of the nextword.
- A memory write operation inserts the word from DR into the top of stack. A new item is deleted with a pop operation asfollows:

DR ← M [SP]

SP←SP+1

- The top item is read from the stack into DR. The stack pointer is then decremented to point at the next item in the stack.

- Most computers do not provide hardware to check for stack overflow (full stack) or underflow (emptystack).
- The stack limits can be checked by using processorregisters:
  - one to hold the upper limit (3000 in thiscase)
  - Other to hold the lower limit (4001 in thiscase).
- After a push operation, *SP* compared with the upper-limit register and after a pop operation, *SP* is a compared with the lower-limitregister.
- The two microoperations needed for either the push or popare
  - An access to memory throughSP
  - UpdatingSP.
- The advantage of a memory stack is that the CPU can refer to it without having specify an address, since the address is always available and automatically updated in the stackpointer.

Reverse Polish Notation:

A stack organization is very effective for evaluating arithmeticexpressions. The common arithmetic expressions are written in *infix notation,* with each operator written *between* the operands.

Consider the simple arithmeticexpression.

A*B+C*D

- For evaluating the above expression it is necessary to compute the product A*B, store this product result while computing C*D, and then sum the twoproducts.
- For doing this type of infix notation, it is necessary to scan back and forth along the expression to determine the next operation to beperformed.
- The Polish mathematician Lukasiewicz showed that arithmetic expression can be represented in *prefixnotation.*
- This representation, often referred to as *Polish notation,* places the operator before the operands. So it is also called as *prefixnotation*.
- The *Postfix notation,* referred to as *reverse Polish notation (RPN),* places the operator after theoperands.
- The following examples demonstrate the three representations

Eg: A+B > Infix notation
+AB > Prefix or Polishnotation
AB+ > Post or reverse Polishnotation

The reverse Polish notation is in a form suitable for stack manipulation. Theexpression

A*B+C*D

Is written in reverse polish notation as

AB* CD* +

And it is evaluated as follows

- Scan the expression from left toright.
- When operator is reached, perform the operation with the two operands found on the left side of the operator.

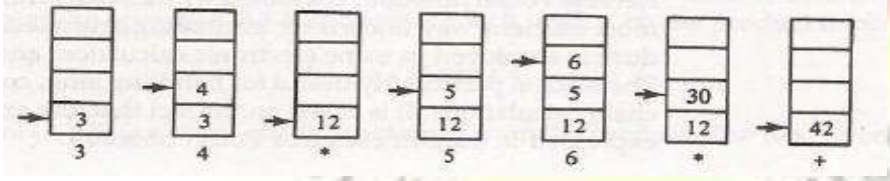
- Remove the two operands and the operator and replace them by the number obtained from the result of theoperation.
- Continue to scan the expression and repeat the procedure for every operation encountered until there are no moreoperators.

For the expression above it find the operator * after A and B. So it perform the operation A*B and replace A, B and * with theresult.
The next operator is a * and it previous two operands are C and D, so it perform the operation C*D and places the result in places C, D and*.
The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain theresult.
The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infixnotation.
This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtractionoperations.

Evaluation of Arithmetic Expressions:
Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmeticexpressions.
This procedure is employed in some electronic calculators and also in somecomputer.
The following numerical example may clarify this procedure. Consider the arithmeticexpression
(3*4) + (5*6)

In reverse polish notation, it is expressed as 34 * 56* +
Now consider the stack operations shown in Fig.8-5.

Figure 8-5 Stack operations to evaluate 3 · 4 + 5 · 6.



Each box represents one stack operation and the arrow always points to the top of thestack.
Scanning the expression from left to right, we encounter twooperands.
First the number 3 is pushed into the stack, then the number4.
The next symbol is the multiplication operator*.
This causes a multiplication of the two top most items thestack.
The stack is then popped and the product is placed on top of the stack, replacing the two originaloperands.
Next we encounter the two operands 5 and 6, so they are pushed into thestack.
The stack operation results from the next * replaces these two numbers by theirproduct.
The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of42.

InstructionFormats:

- The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a controlregister.
- The bits of the instruction are divided into groups calledfields.
- The most common fields found in instruction formatsare:
    - An operation code field that specifies the operation to beperform
    - An address field that designates a memory address or a processorregister.
    - A mode field that specifies the way the operand or the effective address isdetermined.
- Computers may have instructions of several different lengths containing varying number ofaddresses.
- The number of address fields in the instruct format of a computer depends on the internal organization of its registers.
- Most computers fall into one of three types of CPUorganizations:
    - Single accumulatororganization.
    - General registerorganization.
    - Stackorganization.

Single Accumulator Organization:

- In an accumulator type organization all the operations are performed with an implied accumulatorregister.
- The instruction format in this type of computer uses one addressfield.
- For example, the instruction that specifies an arithmetic addition defined by an assembly language instructionas

  ADDX
- Where X is the address of the operand. The ADD instruction in this case results in the operation *AC* □*AC*

+M[X]. AC is the accumulator register and *M[X]* symbolizes the memory word located at address X.

General register organization:

- The instruction format in this type of computer needs three register addressfields.
- Thus the instruction for an arithmetic addition may be written in an assembly languageas

  ADD R1, R2, R3
- to denote the operation R1□R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
- Thusthe instruction**ADDR1,R2**wouldd*e*notetheoperationR1□*R1* +R2.OnlyregisteraddressesforR1 and R2 need be specified in thisinstruction.
- General register-type computers employ two or three address fields in their instructionformat.

- Each address field may specify a processor register or a memoryword.
- An instruction symbolized by **ADD R1, X** would specify the operation R1□R1 +M[X].
- It has two address fields, one for register R1 and the other for the memory address X**.**

Stackorganization:

- The stack-organized CPU has PUSH and POP instructions which require an addressfield.
- Thus the instruction **PUSH X** will push the word at address X to the top of thestack.
- The stack pointer is updatedautomatically.
- Operation-type instructions do not need an address field in stack-organizedcomputers.
- This is because the operation is performed on the two items that are on top of thestack.
- The instruction **ADD** in a stack computer consists of an operation code only with no addressfield.
- This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into thestack.
- There is no need to specify operands with an address field since all operands are implied to be in thestack.
- Most computers fall into one of the three types oforganizations.
- Some computers combine features from more than one organizationalstructure.
- The influence of the number of addresses on computer programs, we will evaluate the arithmeticstatement
- X= (A+B) * (C+D)
- Using zero, one, two, or three address instructions and using the symbols ADD, SUB, MUL and DIV for four arithmetic operations; MOV for the transfer type operations; and LOAD and STORE for transfer to and from memory and ACregister.
- Assuming that the operands are in memory addresses A, B, C, and D and the result must be stored in memory ar address X and also the CPU has general purpose registers R1, R2, R3 andR4.

Three Address Instructions:

- Three-address instruction formats can use each address field to specify either a processor register or a memoryoperand.
- The program assembly language that evaluates **X = (A+B) * (*C*+D**) is shown below, together with comments that explain the register transfer operation of eachinstruction.

```
ADD   R1, A, B     R1←M[A] + M[B]
ADD   R2, C, D     R2←M[C] + M[D]
MUL   X, R1, R2    M[X]←R1*R2
```

- The symbol M [A] denotes the operand at memory address symbolized byA.
- The advantage of the three-address format is that it results in short programs when evaluating arithmeticexpressions.

- The disadvantage is that the binary-coded instructions require too many bits to specify threeaddresses.

Two Address Instructions:

- Two-address instructions formats use each address field can specify either a processor register or memoryword.
- The program to evaluate X = (A+B) * *(C*+D) is asfollows

```
MOV     R1, A     R1←M[A]
ADD     R1, B     R1←R1 + M[B]
MOV     R2, C     R2←M[C]
ADD     R2, D     R2←R2 + M[D]
MUL     R1,R2     R1←R1*R2
MOV     X, R1     M[X]←R1
```

The MOV instruction moves or transfers the operands to and from memory and processorregisters.

- The first symbol listed in an instruction is assumed be both a source and the destination where the result of the operationtransferred.

One Address Instructions:

- One-address instructions use an implied accumulator *(AC)* register for all datamanipulation.
- For multiplication and division there is a need for a second register. But for the basic discussion we will neglect the second register and assume that the *AC* contains the result of alloperations.
- The program to evaluate X=(A+B) * *(C*+D)is

```
LOAD    A    AC←M[A]
ADD     B    AC←AC + M[B]
STORE   T    M[T]←AC
LOAD    C    AC←M[C]
ADD     D    AC←AC + M[D]
MUL     T    AC←AC*M[T]
STORE   X    M[X]←AC
```

- All operations are done between the AC register and a memoryoperand.
- T is the address of a temporary memory location required for storing the intermediateresult.

Zero Address Instructions:

- A stack-organized computer does not use an address field for the instructions ADD andMUL.
- The PUSH and POP instructions, however, need an address field to specify the operand that communicates with thestack.
- The following program shows how X = (A+B) * *(C*+D) will be written for a stack-organizedcomputer.
- *(TOS* stands for top of stack).

```
PUSH    A    TOS ← A
PUSH    B    TOS ← B
ADD          TOS ← (A + B)
PUSH    C    TOS ← C
PUSH    D    TOS ← D
ADD          TOS ← (C + D)
MUL          TOS ← (C + D) * (A + B)
POP     X    M[X] ← TOS
```

- To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polishnotation.
- Thename"zero-address"isgiventothistypeofcomputerbecauseoftheabsenceofanaddressfieldin the computationalinstructions.

RISC Instructions:

- The instruction set of a typical RISC processor is use only load and store instructions for communicating between memory andCPU.
- All other instructions are executed within the registers of CPU without referring tomemory.
    - LOAD and STORE instructions that have one memory and one register address, and computational type instructions that have three addresses with all three specifying processorregisters.
- The following is a program to evaluateX=(A+B)*(C+D)

```
LOAD    R1, A          R1 ← M[A]
LOAD    R2, B          R2 ← M[B]
LOAD    R3, C          R3 ← M[C]
LOAD    R4, D          R4 ← M[D]
ADD     R1, R1, R2     R1 ← R1 + R2
ADD     R3, R3, R2     R3 ← R3 + R4
MUL     R1, R1, R3     R1 ← R1 * R3
STORE   X, R1          M[X] ← R1
```

- The load instructions transfer the operands from memory to CPUregister.
- The add and multiply operations are executed with data in the register without accessingmemory.
- The result of the computations is then stored memory with a store ininstruction.

AddressingModes

- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:
- To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and programrelocation.
- To reduce the number of bits in the addressing field of theinstruction
- Most addressing modes modify the address field of the instruction; there are two modes that need no address field at all. These are *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 register is implied in the definition of theinstruction.
- All register reference instructions that use an accumulator are implied modeinstructions.
- Zero address in a stack organization computer is implied modeinstructions.

Immediate Mode:

- In this mode the operand is specified in the instructionitself.
- In other words an immediate-mode instruction has an operand rather than an addressfield.
- Immediate-mode instructions are useful for initializing registers to a constantvalue.
- The address field of an instruction may specify either a memory word or a processorregister.
- When the address specifies a processor register, the instruction is said to be in the registermode.

Register Mode:

- In this mode the operands are in registers that reside within theCPU.
- The particular register is selected from a register field in theinstruction.

Register Indirect Mode:

- In this mode the instruction specifies a register in CPU whose contents give the address of the operand inmemory.
- In other words, the selected register contains the address of the operand rather than the operanditself.
- The advantage of a register indirect mode instruction is that the address field of the instruction uses few bits to select a register than would have been required to specify a memory addressdirectly.

Auto-increment or Auto-Decrement Mode:

- This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to accessmemory.
- The address field of an instruction is used by the control unit in the CPU to obtain the operand frommemory.
- 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 iscalculated.
- The basic two mode of addressing used in CPU are *direct* and *indirect* addressmode.

Direct Address Mode:

- In this mode the effective address is equal to the address part of theinstruction.

- The operand resides in memory and its address is given directly by the address field of theinstruction.
- In a branch-type instruction the address field specifies the actual branchaddress.
- Indirect Address Mode:
- In this mode the address field of the instruction gives the address where the effective address is stored inmemory.
- Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.
- A few addressing modes require that the address field of the instruction be added to the content of a specific register in theCPU.
- The effective address in these modes is obtained from the followingcomputation:
- Effective address =address part of instruction + content of CPU register
- The CPU register used in the computation may be the program counter, an index register, or a baseregister.
- We have a different addressing mode which is used for a differentapplication.

Relative Address Mode**:**

- In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effectiveaddress.

Indexed Addressing Mode:

- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- An index register is a special CPU register that contains an indexvalue.

Base Register Addressing Mode:

- In this mode the content of a base register is added to the address part of the instruction to obtain the effectiveaddress.
- This is similar to the indexed addressing mode except that the register is now called a base register instead of an indexregister.

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.

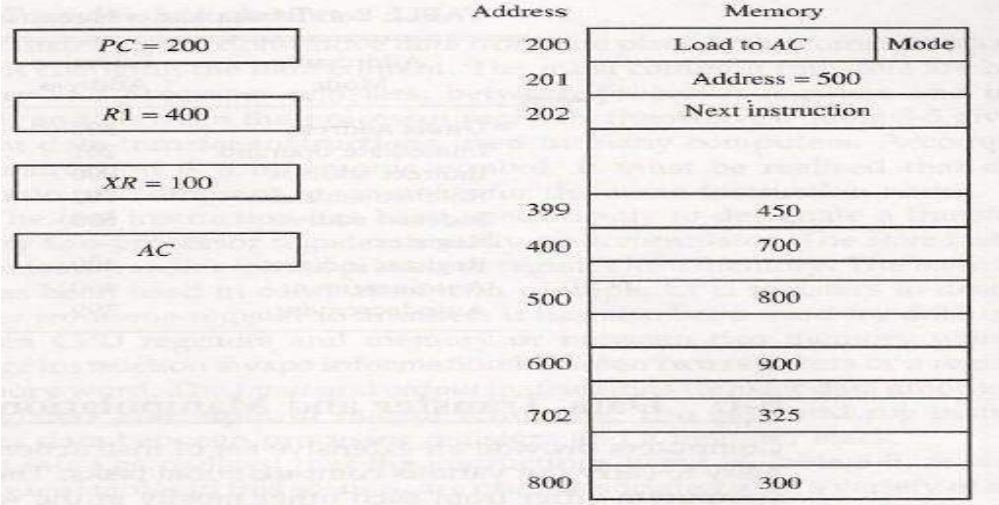


Figure 8-7 Numerical example for addressing modes.

- 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 code and mode, and the second word specifies the address part.
- *PC* has the value 200 for fetching this instruction. The content of processor register R1 is 400, and the content of an index register XR is100.
- AC receives the operand after the instruction isexecuted.
- In the **direct address mode** the effective address is the address part of the instruction 500 and the operand to be loaded into *AC* is500.
- 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.
  - In the **indirect mode** the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is300.
  - In the **relative mode** the effective address is 500 + 202 =702 and the operand is 325. (the value in PC after the fetch phase and during the execute phase is202.)
- In the **index mode** the effective address is XR+ 500 = 100 + 500 = 600 and the operand is900.
- In the **register mode** the operand is in R1 and 400 is loaded into*AC.*
- In the **register indirect mode** the effective address is 400, equal to the content of R1 and the operand loaded into AC is700.
- The **auto-increment mode** is the same as the register indirect mode except that R1 is incremented to 401 after the execution of theinstruction.
- The **auto-decrement mode** decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now450.

Table 8-4 lists the values of the effective address and the operand loaded into AC for the nine 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

Data Transfer andManipulation:

Most computer instructions can be classified into threecategories:

- *Data transferinstructions*
- *Data manipulationinstructions*
- *Program controlinstructions*

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 registersthemselves.

Table 8-5 gives a list of eight data transfer instructions used in manycomputers.

TABLE 8-5 Typical Data Transfer Instructions

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

- The ***load*** instruction has been used mostly to designate a transfer from memory to a processor register, usually anaccumulator.
- The ***store*** instruction designates a transfer from a processor register intomemory.
- The ***move*** instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another and also between CPU registers and memory or between two memorywords.

- The ***exchange*** instruction swaps information between two registers or a register and a memoryword.
- The ***input*** and ***output*** instructions transfer data among processor registers and input or outputterminals.
- The ***push*** and ***pop*** instructions transfer data between processor registers and a memorystack.
- Different computers use different mnemonics symbols for differentiate the addressingmodes.
- As an example, consider the *load to accumulator* instruction when used with eight different addressingmodes.
- Table 8-6 shows the recommended assembly language convention and actual transfer accomplished in each case

TABLE 8-6 Eight Addressing Modes for the Load Instruction

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)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

- *ADR* stands for anaddress.
- *NBA* a number oroperand.
- X is an indexregister.
- The @ character symbolizes an indirectaddressing.
- R1 is a processorregister.
- AC is the accumulatorregister.
- The $ character before an address makes the address relative to the program counter*PC*.
- The # character precedes the operand in an immediate-modeinstruction.
- An indexed mode instruction is recognized by a register that placed in parentheses after the symbolicaddress.
- The register mode is symbolized by giving the name of a processorregister.
- 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 indirect mode by placing a plus after the parenthesized register. The auto-decrement mode would use a minusinstead.

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 basictypes:
    - Arithmeticinstructions
    - Logical and bit manipulationinstructions
    - Shiftinstructions

Arithmetic instructions

- The four basic arithmetic operations are addition, subtraction, multiplication anddivision.
- Most computers provide instructions for all fouroperations.
- Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by mean softwaresubroutines.
- A list of typical arithmetic instructions is given in Table8-7.

TABLE 8-7 Typical Arithmetic Instructions

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

- The increment instruction adds 1 to the value stored in a register or memoryword.
- A number with all 1's, when incremented, produces a number with all0's.
- The decrement instruction subtracts 1 from a value stored in a register or memoryword.
- A number with all 0's, when decremented, produces number with all1's.
- The add, subtract, multiply, and divide instructions may be use different types ofdata.
- The data type assumed to be in processor register during the execution of these arithmetic operations is defined by an operationcode.
- An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precisiondata.
- The mnemonics for three add instructions that specify different data types are shown below. ADDI Add two binary integernumbers
- ADDF Add two floating-point numbers ADDD Add two decimal numbers in BCD
- A special carry flip-flop is used to store the carry from anoperation.

- The instruction "add carry" performs the addition on two operands plus the value of the carry the previous computation.
- Similarly, the "subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtractoperation.
- The negate instruction forms the 2's complement number, effectively reversing the sign of an integer when represented it signed-2's complementform.
- Logical and bit manipulationinstructions

Logical instructions perform binary operations on strings of bits store,registers.

- They are useful for manipulating individual bits or a group of that represent binary-codedinformation.
- The logical instructions consider each bit of the operand separately and treat it as a Booleanvariable.
- By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in register memorywords.
- Some typical logical and bit manipulation instructions are listed in Table8-8.

TABLE 8-8 Typical Logical and Bit Manipulation Instructions

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

- The clear instruction causes the specified operand to be replaced by0’s.
- The complement instruction produces the 1's complement by inverting all bits of theoperand.
- The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands.
- The logical instructions can also be used to performing bit manipulationoperations.
- There are three bit manipulation operations possible: a selected bit can cleared to 0, or can be set to 1, or can be complemented.
- The AND instruction is used to clear a bit or a selected group of bits of anoperand.
- The OR instruction is used to set a bit or a selected group of bits of anoperand.
- Similarly, the XOR instruction is used to selectively complement bits of anoperand.
- Other bit manipulations instructions are included in above table perform the operations on individual bits such as a carry can be cleared, set, orcomplemented.

- Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulationinstructions.

ShiftInstructions:

- Shifts are operations in which the bits of a word are moved to the left orright.
- The bit shifted in at the end of the word determines the type of shiftused.
- Shift instructions may specify logical shifts, arithmetic shifts, or rotate-typeoperations.
- In either case the shift may be to the right or to theleft.

Table 8-9 lists four types of shiftinstructions.

TABLE 8-9 Typical Shift Instructions

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

- The logical shift inset to the end bitposition.
- The end position is the leftmost bit position for shift rights the rightmost bit position for the shiftleft.
- Arithmetic shifts usually conform to the rules for signed-2's complementnumbers.
- The arithmetic shift-right instruction must preserve the sign bit in the leftmostposition.
- 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 thesame.
- The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logicalshift-instruction.
- The rotate instructions produce a circular shift. Bits shifted out at one of the word are not lost as in a logical shift but are circulated back into the otherend.
- 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, shift the entire register to the left.

ProgramControl:

- Program control instructions specify conditions for altering the content of the programcounter.
- The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instructionexecution.

- Thisinstructionprovidescontroloverthefl owofprogramexecutionandacapabilityforbranchingtodifferent programsegments.
- Some typical program control instructions are listed in Table8.10.

TABLE 8-10 Typical Program Control Instructions

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

- Branch and jump instructions may be conditional orunconditional.
- An unconditional branch instruction causes a branch to the specified address without anyconditions.
- The conditional branch instruction specifies a condition such as branch if positive or branch ifzero.
- The skip instruction does not need an address field and is therefore a zero-addressinstruction.

A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing programcounter. The call and return instructions are used in conjunction withsubroutines.
The compare instruction forms a subtraction between two operands, but the result of the operation not retained. However, certain status bit conditions are set as a result ofoperation. Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing theoperands.

Status Bit Conditions:

- The ALU circuit in the CPU have status register for storing the status bitconditions.
- Status bits are also called *condition-code* bits or *flag*bits.
- Figure 8-8 shows block diagram of an 8-bit ALU with a 4-bit statusregister.

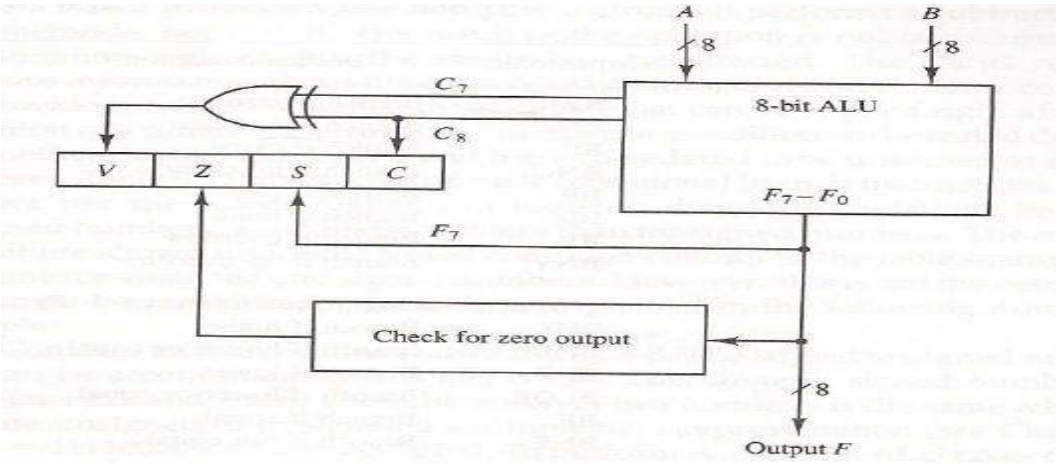


Figure 8-8 Status register bits.

- 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 theALU.
- Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is0.
- S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is0.
- Bit *Z* (zero) is set to 1 if the output of the ALU contains all 0's. It is clear to 0 otherwise. In other words, *Z* = 1 if the output is zero and *Z =0 if the output is notzero.*
- Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries equal to 1, and cleared to 0 otherwise.
- The above status bits are used in conditional jump and branchinstructions.

Subroutine Call and Return:

- A subroutine is self contained sequence of instructions that performs a given computationaltask.
- Themostcommonnamesusedarecallsubroutine,jumptosubroutine,branchtosubroutine,or branch and save returnaddress.
- A subroutine is executed by performing twooperations
- Theaddressofthenextinstructionavailableintheprogramcounter(thereturnaddress)isstore d in a temporary location so the subroutine knows where toreturn
- Control is transferred to the beginning of thesubroutine.
- The last instruction of every subroutine, commonly called *return from subroutine*, transfers the return address from the temporary location in the programcounter.
- Different computers use a different temporary location for storing the returnaddress.
- The most efficient way is to store the return address in a memorystack.
- The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into thestack.
- A subroutine call is implemented with the followingmicrooperations:

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

- The instruction that returns from the last subroutine is implemented by themicrooperations:

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

Program Interrupt:

- Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generatedrequest.
- The interrupt procedure is similar to a subroutine call except for threevariations:
- The interrupt is initiated by an internal or externalsignal.
- Address of the interrupt service program is determined by thehardware.
- An interrupt procedure usually stores all the information rather than storing only PCcontent.

Types of interrupts:
There are three major types of interrupts that cause a break in the normal execution of aprogram.
They can be classifiedas

- Externalinterrupts:

These comefrominput—output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other externalsource.
Ex: I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or powerfailure.

- Internalinterrupts:

These arise from illegal or erroneous use of an instruction ordata.
Internal interrupts are also called*traps.*
Ex: interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protectionviolation.
Internal and external interrupts are initiated form signals that occur in hardware ofCPU.

- Softwareinterrupts

A software interrupt is initiated by executing aninstruction.
Software interrupt is a special call instruction that behaves like an interrupt rather than a

subroutine call.

Reduced Instruction SetComputer:

A computer with large number instructions is classified as a *complex instruction set computer,* abbreviated as CISC.
The computer which having the fewer instructions is classified as a *reduced instruction setcomputer,*
abbreviated as RISC.

CISC Characteristics:

- A large number of instructions--typically from 100 to 250instructions.
- Some instructions that perform specialized tasks and are usedinfrequently.
- A large variety of addressing modes—typically from 5 to 20 differmodes.
- Variable-length instructionformats
- Instructions that manipulate operands inmemory

RISC Characteristics:

- Relatively fewinstructions
- Relatively few addressingmodes
- Memory access limited to load and storeinstructions
- All operations done within the registers of theCPU
- Fixed-length, easily decoded instructionformat
- Single-cycle instructionexecution
- Hardwired rather than microprogrammedcontrol
- A relatively large number of registers in the processorunit
- Efficient instructionpipeline

RISC Pipelines

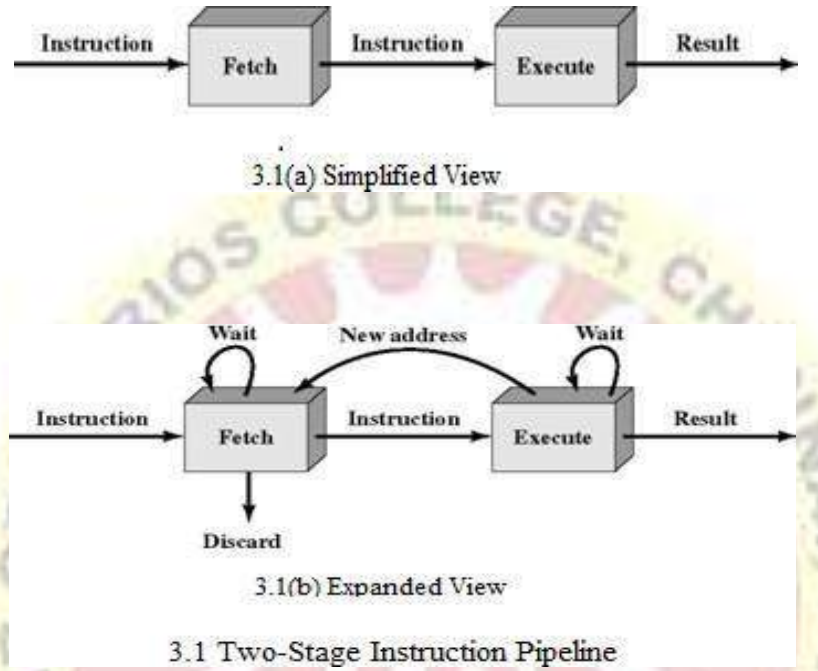
A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different numbers of steps, they are basically variations of these five, used in the MIPS R3000 processor:

- fetch instructions from memory
- read registers and decode the instruction
- execute the instruction or calculate an address
- access an operand in data memory
- write the result into a register

If you glance back at the diagram of the laundry pipeline, you'll notice that although the washer finishes in half an hour, the dryer takes an extra ten minutes, and thus the wet clothes must wait ten minutes for the dryer to free up. Thus, the length of the pipeline is dependent on the length of the longest step. Because RISC instructions are simpler than those used in pre-RISC processors (now called CISC, or Complex Instruction Set Computer), they are more conducive to pipelining. While CISC instructions varied in length, RISC instructions are all the same length and can be fetched in a single operation. Ideally, each of the stages in a RISC processor pipeline should take 1 clock cycle so that the processor finishes an instruction each clock cycle and averages one cycle per instruction (CPI).

INSTRUCTIONPIPELINING

As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry, use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach is instruction pipelining in which new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.



3.1(a) Simplified View

3.1(b) Expanded View

3.1 Two-Stage Instruction Pipeline

Figure 3.1a depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called *instruction prefetch*or *fetch overlap*.

This process will speed up instruction execution only if the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 3.1b), we will see that this doubling of execution rate is unlikely for 3 reasons:

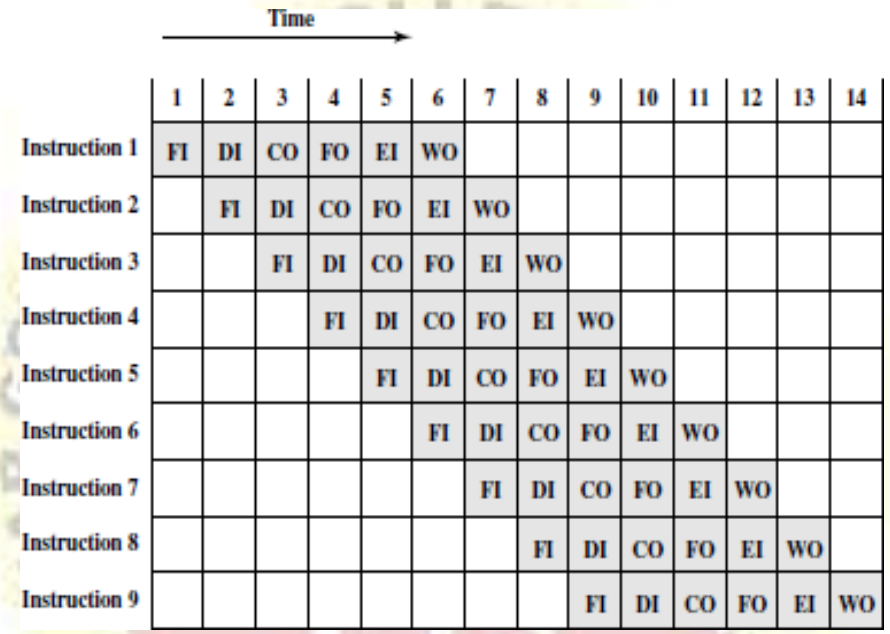
The execution time will generally be longer than the fetch time. Thus, the fetch stage may have to wait for some time before it can empty itsbuffer.

A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction isfetched.

When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost .If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

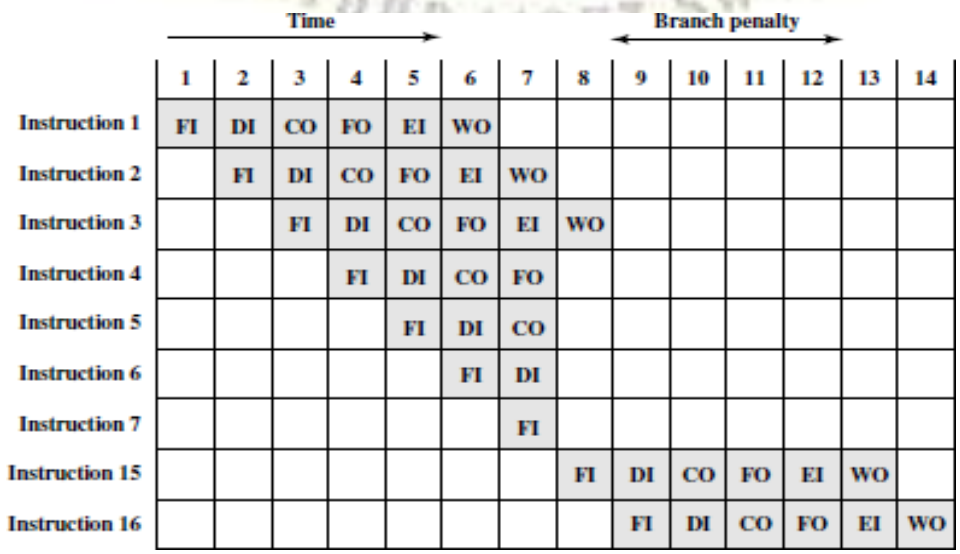
To gain further speedup, the pipeline must have more stages. Let us consider the following

decomposition of the instruction processing.
Fetch instruction (FI): Read the next expected instruction into abuffer.
Decode instruction (DI): Determine the opcodeand the operandspecifiers.
Calculate operands (CO): Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of addresscalculation.
Fetch operands (FO): Fetch each operand frommemory.
Execute instruction (EI): Perform the indicated operation and store the result, if any, in the specified destination operandlocation.
Write operand (WO): Store the result inmemory.
Figure 3.2 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.



Timing Diagram for Instruction PipelineOperation

FO and WO stages involve a memory access. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable

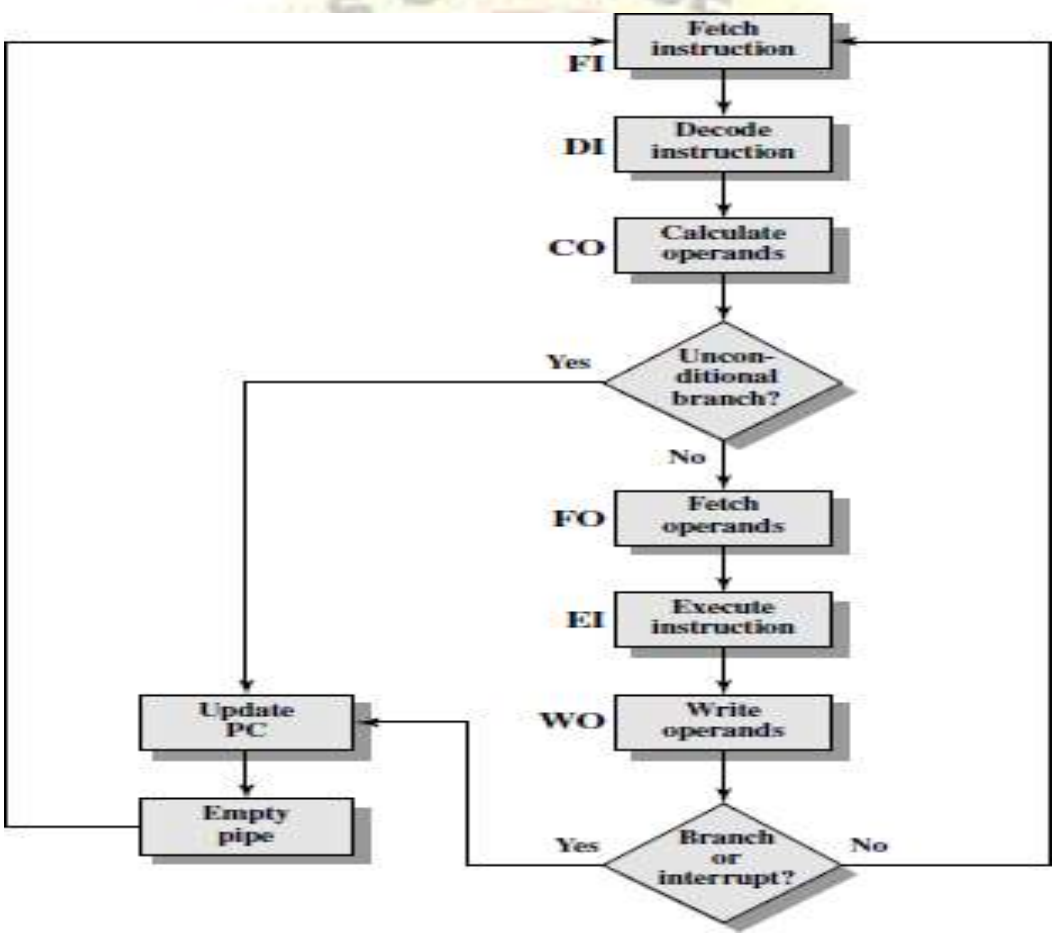


event is an interrupt.

Timing Diagram for Instruction Pipeline Operation withinterrupts

Figure 3.3 illustrates the effects of the conditional branch, using the same program as Figure 3.2. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds.

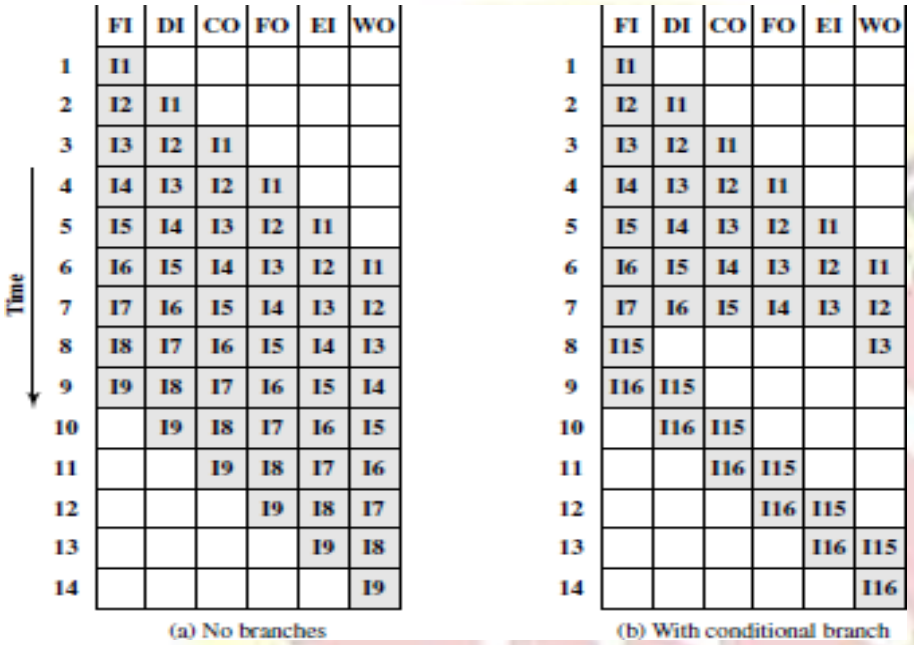
In Figure 3.2, the branch is not taken. In Figure 3.3, the branch is taken. This is not determined until the end of time unit 7.At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch. Figure 3.4 indicates the logic needed for pipelining to account for branches and interrupts.



Six-stage CPU InstructionPipeline

Figure 3.5 shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time. In Figure 3.5a (which corresponds to Figure 3.2), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 3.5b, (which corresponds to Figure 3.3), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

For high-performance in pipelining designer must still consider about :
At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a singleinstruction.
The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the logic controlling the gating between stages is more complex than the stages being controlled.
Latching delay: It takes time for pipeline buffers to operate and this adds to instructioncycle time.



(a) No branches (b) With conditional branch

Vector Processors
Vector processors are co-processor to general-purpose microprocessor. Vector processors are generally register-register or memory-memory. A vector instruction is fetched and decoded and then a certain operation is performed for each element of the operand vectors, whereas in a normal processor a vector operation needs a loop structure in the code. To make it more efficient, vector processors chain several vector operations together, i.e., the result from one vector operation are forwarded to another as operand.
Characteristics of Vector processing
A vector is an ordered set of elements. A vector operand contains an ordered set of n elements, where n is called the length of the vector. Each element in a vector is a scalar quantity, which may be floating point number, an integer, a logical value, or a character (byte).
In **vector processing**, two successive pairs of elements are processed each clock period. In dual vector pipes and dual sets of vector functional units allow two pairs of elements to be processed during the same clock period. As each pair of operations is completed, the results are delivered to the appropriate elements of the result register. The operation continues until the number of elements processed is equal to the count specified by the vector length register.
For example: C (1:50) = A (1:50) + B (1:50)

This vector instruction includes the initial addresses of the two source operands, one destination operand, the length of the vectors and the operation to be performed.
Vector instructions are classified into for basic types:

F1: V = V f2: V = S

F3: V * V = V f4: V*S = V

Where V indicates vector operand and S indicates scalar operand. The operations f1 and f2 are unary operations such as vector square root, vector sine, vector complement, vector summation and so on. On the other hand, operations f3 and f4 are binary operations such as vector add, vector multiply, vector scalar adds and so on.
In **vector processing**, identical processes are repeatedly invoked many times, each of which can be subdivided into subprocesses.
In **vector processing**, successive operands are fed through the pipeline segments and require as few buffers and local controls as possible. This parallel vector processing allows the generation of more than two results per clock period. The parallel vector operations are automatically initiated either when successive vector instructions use different functional units and different vector registers, or when successive vector instructions use the result stream from one vector register as the operand of another operation using different functional units. This process is known as chaining.
Because of the startup delay in a pipeline, a vector processor performs better with longer vectors.
Vector processing is usually faster and more efficient than scalar processing because it reduces the overhead associated with maintenance of the loop control variables.
Vector Instruction Fields
Vector instructions are usually specified by the following fields:
Opcode (operation code):
This field is used to select the functional unit or to reconfigure a multifunctional unit to perform the specified operation.
Base addresses:
In case of memory reference instruction, this field specifies the base addresses needed for source operands and result vectors. If the operands and results are located in the vector register file, the designated vector registers must be specified.
Address increment:
This field specifies the space between the two elements in the main memory. Usually, the elements are consecutively stored thus the increment is 1. However, with variable increment higher flexibility can be offered in the applications.
Address offset:
This field specifies the offset to the base address. Using the base address and the offset, the effective memory address can be calculated. The offset can be either positive or negative.
Vector length:
this field determines the termination of a vector instruction. Vector length affects the processing efficiency because the additional subdividing is required for long vectors.
Array processor
A computer/processor that has an architecture especially designed for processing arrays (e.g. matrices) of numbers. The architecture includes a number of processors (say 64 by 64) working simultaneously, each handling one element of the array, so that a single operation can apply to all elements of the array in parallel. To obtain the same effect in a conventional processor, the operation must be applied to each element of the array sequentially, and so

consequently much more slowly.

An array processor may be built as a self-contained unit attached to a main computer via an I/O port or internal bus; alternatively, it may be a *distributed array processor* where the processing elements are distributed throughout, and closely linked to, a section of the computer's memory.

Array processors are very powerful tools for handling problems with a high degree of parallelism. They do however demand a modified approach to programming. The conversion of conventional (sequential) programs to serve array processors is not a trivial task, and it is sometimes necessary to select different (parallel) algorithms to suit the parallel approach.

UNIT - III

Microprocessor Architecture and its Operations - 8085 MPU - 8085 Instruction Set and Classifications. Programming in 8085: Code conversion - BCD to Binary and Binary to BCD conversions - ASCII to BCD and BCD to ASCII conversions - Binary to ASCII and ASCII to Binary conversions.

What is the 8085 Microprocessor?

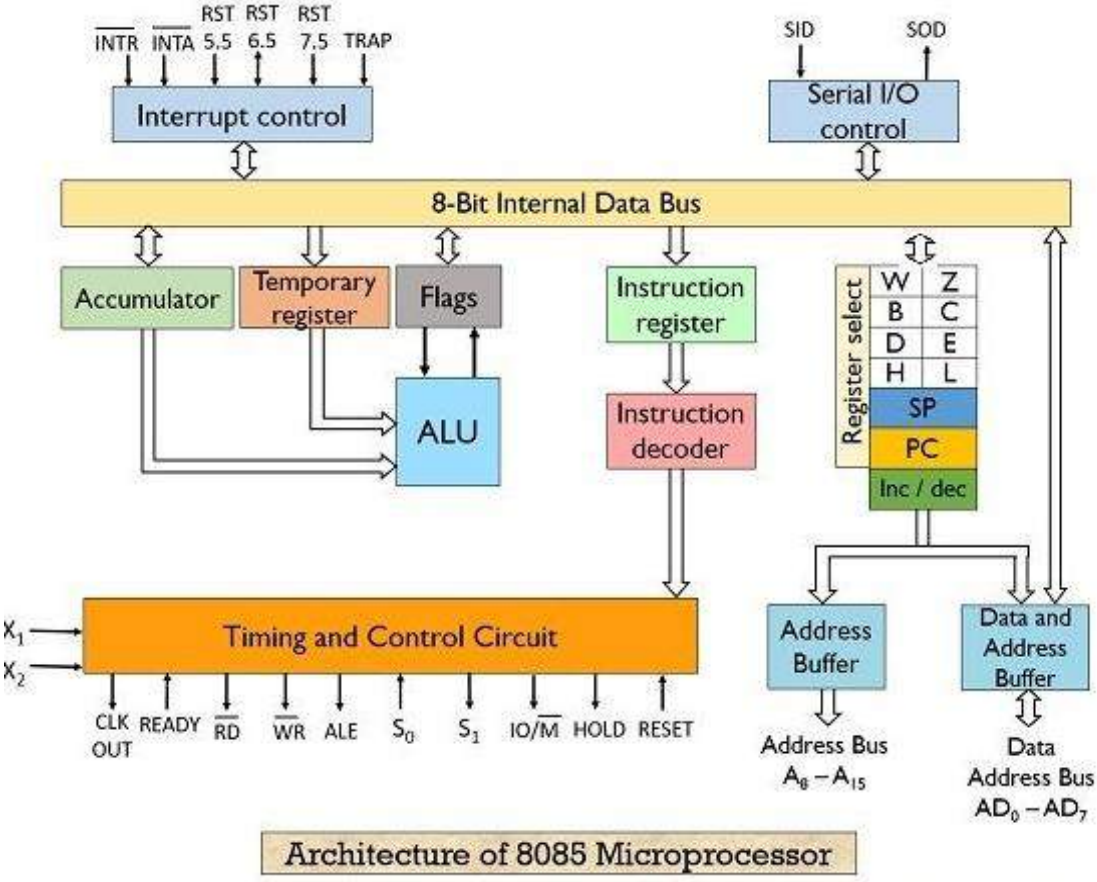
The 8085 is an 8-bit microprocessor, and it was launched by the Intel team in the year of 1976 with the help of NMOS technology. This processor is the updated version of the microprocessor. The configurations of 8085 microprocessor mainly include data bus-8-bit, address bus-16 bit, **program counter**-16-bit, stack pointer-16 bit, registers 8-bit, +5V voltage supply, and operates at 3.2 MHz single segment CLK. The applications of 8085 microprocessor are involved in microwave ovens, washing machines, gadgets, etc. The **features of the 8085 microprocessor** are as below:

This microprocessor is an 8-bit device that receives, operates, or outputs 8-bit information in a simultaneous approach.

- The processor consists of 16-bit and 8-bit address and data lines and so the capacity of the device is 2^{16} which is 64KB of memory.
- This is constructed of a single NMOS chip device and has 6200 transistors
- A total of 246 operational codes and 80 instructions are present
- As the 8085 microprocessor has 8-bit input/output address lines, it has the ability to address 2^8 = 256 input and output ports.
- This microprocessor is available in a DIP package of 40 pins
- In order to transfer huge information from I/O to memory and from memory to I/O, the processor shares its bus with the DMA controller.
- It has an approach where it can enhance the interrupt handling mechanism
- An 8085 processor can even be operated as a three-chip microcomputer using the support of IC 8355 and IC 8155 circuits.
- It has an internal clock generator
- It functions on a clock cycle having a duty cycle of 50%

The 8085 Microprocessor Architecture

The architecture of the 8085 microprocessor mainly includes the timing & control unit, Arithmetic and logic unit, decoder, instruction register, interrupt control, a register array, serial input/output control. The most important part of the microprocessor is the central processing unit.



Architecture of 8085 Microprocessor

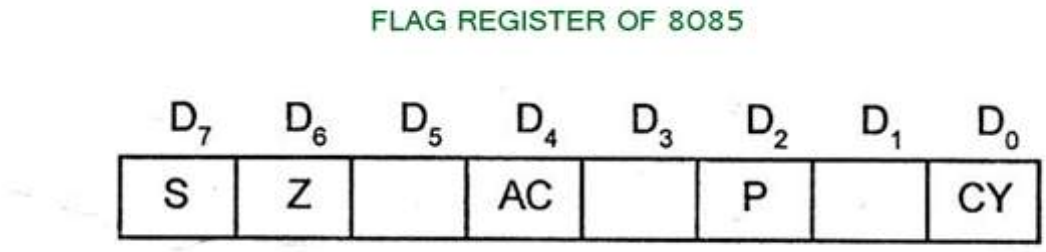
Operations of the 8085 Microprocessor

The main operation of ALU is arithmetic as well as logical which includes addition, increment, subtraction, decrement, logical operations like AND, OR, Ex-OR, complement, evaluation, left shift or right shift. Both the temporary registers as well as accumulators are utilized for holding the information throughout the operations then the outcome will be stored within the accumulator. The different flags are arranged or rearrange based on the outcome of the operation.

Flag Registers

The flag registers of **microprocessor 8085** are classified into five types namely sign, zero, auxiliary carry, parity and carry. The positions of bit set aside for these types of flags. After the operation of an ALU, when the result of the most significant bit (D7) is one, then the sign

flag will be arranged. When the operation of the ALU outcome is zero then the zero flags will be set. When the outcome is not zero then the zero flags will be reset.



Flag is an 8-bit register containing 5 1-bit flags:
Sign - set if the most significant bit of the result is set.
Zero - set if the result is zero.
Auxiliary carry - set if there was a carry out from bit 3 to bit 4 of the result.
Parity - set if the parity (the number of set bits in the result) is even.
Carry - set if there was a carry during addition, or borrow during subtraction/comparison.

8085 Microprocessor Flag Registers

In an arithmetic process, whenever a carry is produced with the lesser nibble, then an auxiliary type carry flag will be set. After an ALU operation, when the outcome has an even number then the parity flag will be set, or else it is reset. When an arithmetic process outcome in a carry, then carry flag will be set or else it will be reset. Between the five types of flags, the AC type flag is employed on the inside intended for BCD arithmetic as well as remaining four flags are used with the developer to make sure the conditions of the outcome of a process.

Control and Timing Unit

The control and timing unit coordinates with all the actions of the microprocessor by the clock and gives the control signals which are required for communication among the microprocessor as well as peripherals.

Decoder and Instruction Register

As an order is obtained from memory after that it is located in the instruction register, and encoded & decoded into different device cycles.

Register Array

The general purpose programmable registers are classified into several types apart from the accumulator such as B, C, D, E, H, & L. These are utilized as 8-bit registers otherwise coupled to stock up the l6 bit of data. The permitted couples are BC, DE & HL, and the short term W & Z registers are used in the processor & it cannot be utilized with the developer.

Special Purpose Registers

These registers are classified into four types namely program counter, stack pointer, increment or decrement register, address buffer, or data buffer.

Program Counter

This is the first type of special-purpose register and considers that the instruction is being performed by the microprocessor. When the ALU completed performing the instruction, then the microprocessor searches for other instructions to be performed. Thus, there will be a requirement of holding the next instruction address to be performed in order to conserve time. Microprocessor increases the program when an instruction is being performed, therefore that the program counter-position to the next instruction memory address is going to be performed…

Stack Pointer in 8085

The SP or stack pointer is a 16-bit register and functions similar to a stack, which is constantly increased or decreased with two throughout the push and pop processes.

Increment or Decrement Register

The 8-bit register contents or else a memory position can be increased or decreased with one. The 16-bit register is useful for incrementing or decrementing program counters as well as stack pointer register content with one. This operation can be performed on any memory position or any kind of register.

Address-Buffer & Address-Data-Buffer

Address buffer stores the copied information from the memory for the execution. The memory & I/O chips are associated with these buses; then the CPU can replace the preferred data by I/O chips and the memory.

Address Bus and Data Bus

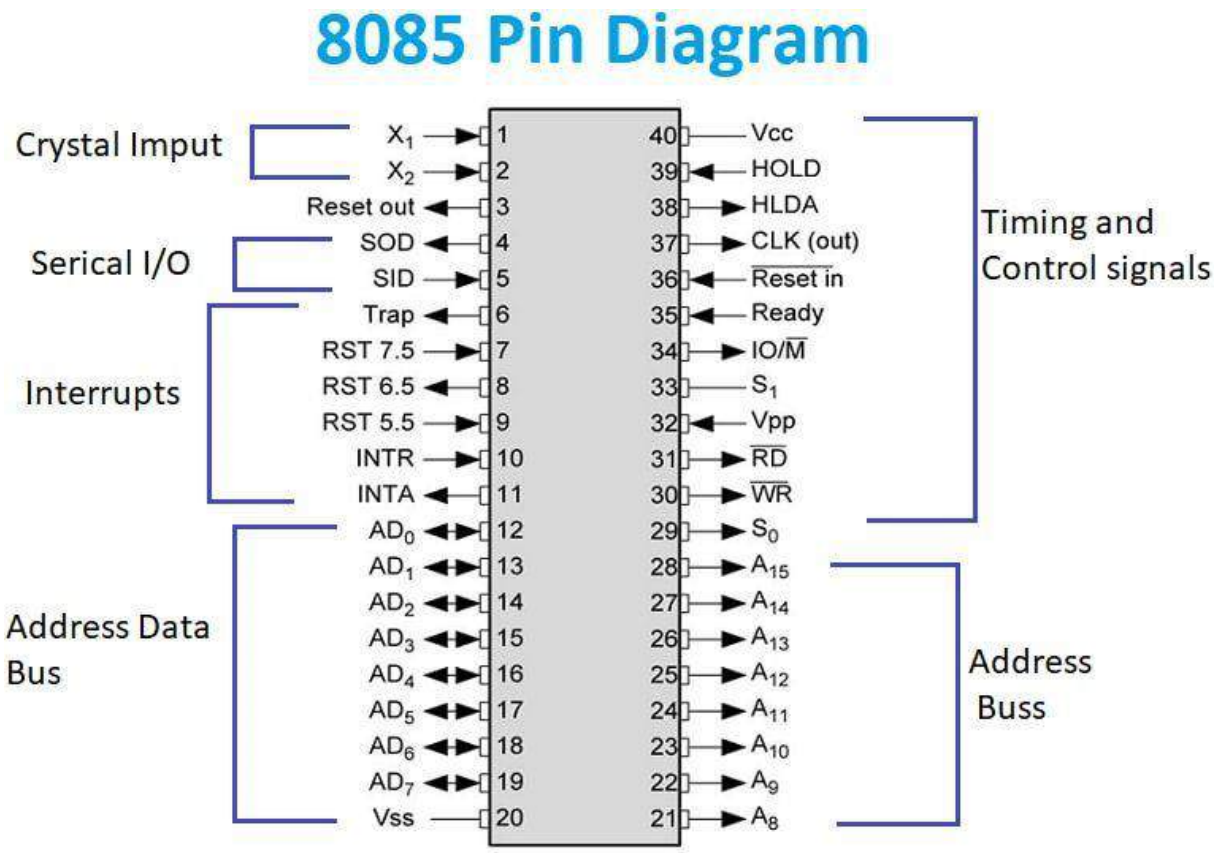
The data bus is useful in carrying the related information that is to be stock up. It is bi-directional, but the address bus indicates the position as to where it must be stored & it is uni-directional, useful for transmitting the information as well as address input/output devices.

Timing & Control Unit

The timing & control unit can be used to supply the signal to the 8085 microprocessor architecture for achieving the particular processes. The timing and control units are used to control the internal as well as external circuits. These are classified into four types namely control units like RD’ ALE, READY, WR’, status units like S0, S1, and IO/M’, DM like HLDA, and HOLD unit, RESET units like RST-IN and RST-OUT.

Pin Diagram

This 8085 is a 40-pin microprocessor where these are categorized into seven groups. With the below 8085 microprocessor pin diagram, the functionality and purpose can be known easily.



Data Bus

The pins from 12 to 17 are the data bus pins which are AD_0 – AD_7, this carries the minimal considerable 8-bit data and address bus.

Address Bus

The pins from 21 to 28 are the data bus pins which are A_8 – A_{15}, this carries the most considerable 8-bit data and address bus.

Status and the Control Signals

In order to find out the behavior of the operation, these signals are mainly considered. In the 8085 devices, there are 3 each the control and status signals.

RD – This is the signal used for the regulation of READ operation. When the pin moves into low, it signifies that the chosen memory is read.

WR – This is the signal used for the regulation of WRITE operation. When the pin moves into low, it signifies that the data bus information is written to the chosen memory location.

ALE – ALE corresponds to Address Latch Enable signal. The ALE signal is high at the time of the machine's initial clock cycle and this enables the last 8 bits of the address to get latched with the memory or external latch.
IO/M – This is the status signal that recognizes whether the address to be allotted for I/O or for memory devices.
READY – This pin is used to specify whether the peripheral is able to transfer information or not. When this pin is high, it transfers data and if this is low, the microprocessor device needs to wait until the pin goes to a high state.
S_0 and S_1 pins – These pins are the status signals which defines the below operations and those are:

S0	**S1**	**Functionality**
0	0	Halt
1	0	Write
0	1	Read
1	1	Fetch

Clock Signals

CLK – This is the output signal which is pin 37. This is utilized even in other digital integrated circuits. The frequency of the clock signal is similar to the processor frequency.
X1 and X2 – These are the input signals at pins 1 and 2. These pins have connections with the external oscillator that operates the device's internal circuitry system. These pins are used for the generation of the clock that is required for the microprocessor functionality.
Reset Signals

There are two reset pins which are Reset In and Reset Out at pins 3 and 36.

RESET IN – This pin signifies resetting the program counter to zero. Also, this pin resets the HLDA flip-flops and IE pins. The control processing unit will be in a reset state till RESET is not triggered.
RESET OUT – This pin signifies that the CPU is in reset condition.

Serial Input/Output Signals

SID – This is the serial input data line signal. The information that is on this dateline is taken into the 7^{th} bit of the ACC when the RIM functionality is performed.
SOD – This is the serial output data line signal. The ACC's 7^{th} bit is the output on the SOD data line when the SIIM functionality is performed.

Externally Initiated and Interrupts Signals

HLDA – This is the signal for HOLD acknowledgment that signifies the received signal of HOLD request. When the request is removed, the pin goes to a low state. This is the output pin.
HOLD – This pin indicates that the other device is in the need to utilize data and address buses. This is the input pin.
INTA – This pin is the interrupt acknowledgment that is directed by the microprocessor device after the receival of the INTR pin. This is the output pin.
INTR – This is the interrupt request signal. It has minimal priority when compared with other interrupt signals.

Interrupt Signal	**Next instruction location**
TRAP	0024
RST 7.5	003C
RST 6.5	0034
RST 5.5	002C

TRAP, RST 5.5, 6.5, 7.5 – These all are the input interrupt pins. When any one of the interrupt pins are recognized, then the next signal has functioned from the constant position in the memory based on the below table:
The priority list of these interrupt signals is

TRAP – Highest

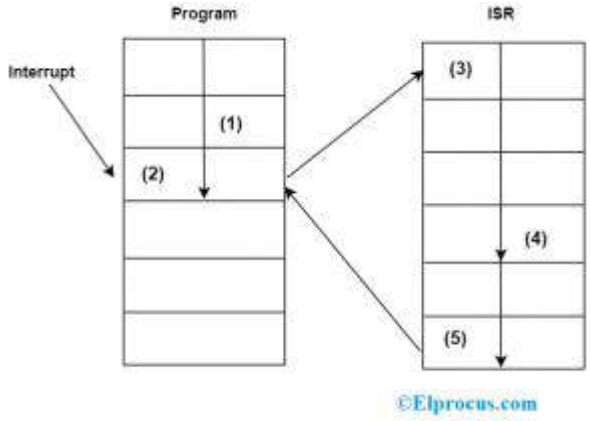
RST 7.5 – High

RST 6.5 – Medium

RST 5.5 – Low

INTR – Lowest

The power supply signals are **Vcc** and **Vss** which are +5V and ground pins.



8085 Microprocessor Interrupt

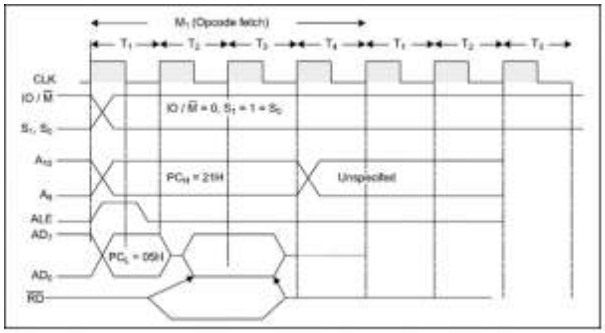
Timing Diagram of 8085 Microprocessor

To clearly understand the operation and performance of the microprocessor, the timing diagram is the most suitable approach. Using the timing diagram, it is easy to know the system functionality, detailed functionality of every instruction and the execution, and others. The timing diagram is the graphical portray of instructions is steps corresponding to time. This signifies the clock cycle, time period, data bus, operation type such as RD/WR/Status, and clock cycle.

In the 8085 microprocessor architecture, here we will look into the timing diagrams of I/O RD, I/O WR, memory RD, memory WR, and opcode fetch.

Opcode Fetch

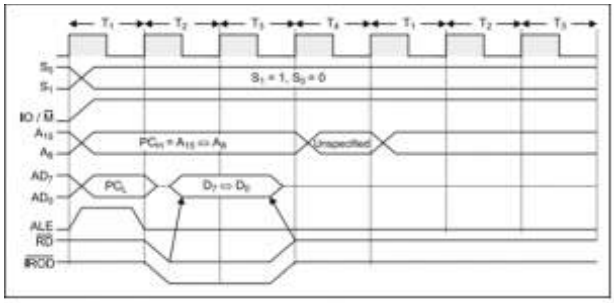
The timing diagram is:



Opcode Fetch in 8085 Microprocessor

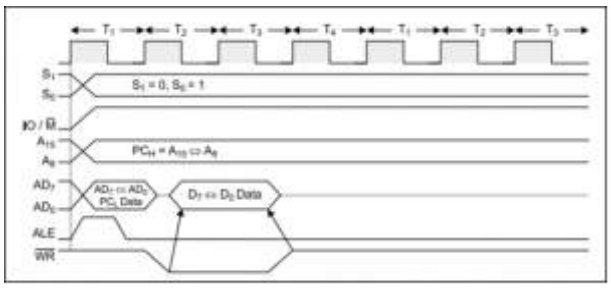
I/O Read

The timing diagram is:



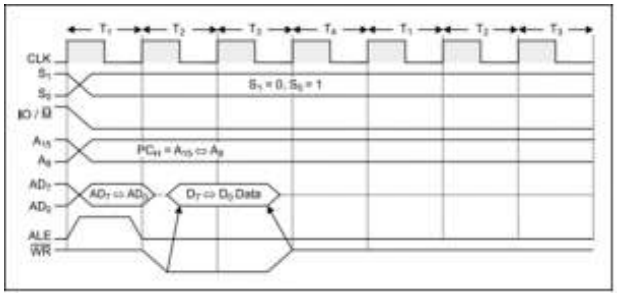
Input Read

I/O Write
The timing diagram is:

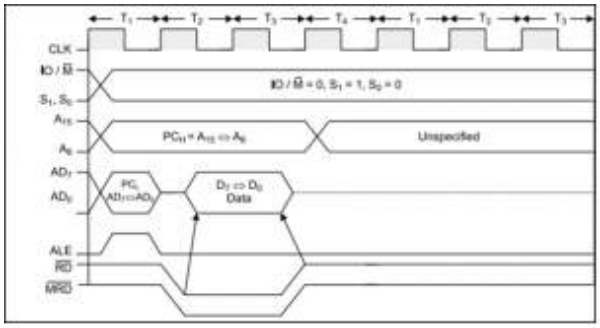


Input Write

Memory Read
The timing diagram is:



Memory Write
The timing diagram is:



Memory Write in 8085 Microprocessor

For all these timing diagrams, the commonly used terms are:

RD – When it is high, this means the microprocessor reads no data, or when it is low, this means the microprocessor reads data.
WR – When it is high, this means the microprocessor writes no data, or when it is low, this means the microprocessor writes data.
IO/M – When it is high, this means the device performs I/O operation, or when it is low, this means the microprocessor performs memory operation.
ALE – This signal implies valid address availability. When the signal is high, it performs as an address bus, or when it is low, it performs as a data bus.
S0 and S1 – Signifies the kind of machine cycle that is in progress.
Consider the below table:

	Status Signals			**Control Signals**		
Machine Cycle	IO/M’	S1	S0	RD’	WR’	INTA’
Opcode fetch	0	1	1	0	1	1
Memory Read	0	1	0	0	1	1
Memory Write	0	0	1	1	0	1
Input Read	1	1	0	0	1	1
Input Write	1	0	1	1	0	1

8085 Microprocessor Instruction Set

The **instruction set of 8085** microprocessor architecture is nothing but instruction codes used to achieve an exact task, and instruction sets are categorized into various types namely control, logical, branching, arithmetic, and data transfer instructions.

8085 Program to convert a two-digit BCD to binary

In this program we will see how to convert BCD numbers to binary equivalent.

Problem Statement

A BCD number is stored at location 802BH. Convert the number into its binary equivalent andstore it to the memory location 802CH.

Discussion

In this problem we are taking a BCD number from the memory and converting it to its binaryequivalent. At first we are cutting each nibble of the input. So ifthe input is 52 (0101 0010) then we can simply cut it by masking the number by 0FH and F0H. When the Higher order nibble is cut, thenrotate it to the left four times to transfer it to lower nibble.

Now simply multiply the numbers by using decimal adjust method to get final decimal result.

Input

Address	Data
. . .	. . .
802B	52
. . .	. . .

Program

Address	HEX Codes	Labels	Mnemonics	Comments
8000	31, FF, 80		LXI SP,80FFH	Initialize stack pointer
8003	21, 2B, 80		LXI H, 802BH	Pointer to the IN-BUFFER
8006	01, 2C, 80		LXI B, 802CH	Pointer to the OUT-BUFFER
8009	7E		MOV A, M	Move the contents of 802BH to A

Address	HEX Codes	Labels	Mnemonics	Comments
800A	CD, 0F, 80		CALL BCDBIN	Subroutine to convert a BCD number to HEX
800D	02		STAX B	Store Acc to memory location pointed by BC
800E	76		HLT	Terminate the program
800F	C5	BCDBIN	PUSH B	Saving B
8010	47		MOV B, A	Copy A to B
8011	E6, 0F		ANI 0FH	Mask of the most significant four bits
8013	4F		MOV C, A	Copy A to C
8014	78		MOV A, B	Copy B to A
8015	E6, F0		ANI F0H	Mask of the least significant four bits
8017	0F		RRC	Rotate accumulator right 4 times
8018	0F		RRC	
8019	0F		RRC	
801A	0F		RRC	
801B	57		MOV D, A	Load the count value to the Reg. D

Address	HEX Codes	Labels	Mnemonics	Comments
801C	AF		XRA A	Clear the contents of the accumulator
801D	1E, 0A		MVI E, 0AH	Initialize Reg. E with 0AH
801F	83	SUM	ADD E	Add the contents of Reg. E to A
8020	15		DCR D	Decrement the count by 1 until 0 is reached
8021	C2, 1F, 80		JNZ SUM	
8024	81		ADD C	Add the contents of Reg. C to A
8025	C1		POP B	Restoring B
8026	C9		RET	Returning control to the calling program

Output

Address	Data
. . .	. . .
802C	34
. . .	. . .

8085 Program to convert an 8-bit binary to BCD

In this program we will see how to convert binary numbers to its BCD equivalent.

Problem Statement

A binary number is store dat location 800H. Convert the number into its BCD equivalent and store it to the memory location 8050H.

Discussion

Here we are taking a number from the memory, and initializing it as a counter. Now in each step of this counter we are incrementing the number by 1, and adjust the decimal value. By this process we are finding the BCD value of binary number or hexadecimal number.

We can use INR instruction to increment the counter in this case but this instruction will not affect carry flag, so for that reason we have used ADI 10H

Input

Address	Data
. . .	. . .
8000	34
. . .	. . .

Program

Address	HEX Codes	Labels	Mnemonics	Comments
F000	21, 00, 80		LXI H,8000H	Initialize memory pointer
F003	16, 00		MVI D,00H	Clear D- reg for Most significant Byte

Address	HEX Codes	Labels	Mnemonics	Comments
F005	AF		XRA A	Clear Accumulator
F006	4E		MOV C, M	Get HEX data
F007	C6, 01	LOOP	ADI 01H	Count the number one by one
F009	27		DAA	Adjust for BCD count
F00A	D2, 0E, F0		JNC SKIP	Jump to SKIP
F00D	14		INR D	Increase D
F00E	0D	SKIP	DCR C	Decrease C register
F00F	C2, 07, F0		JNZ LOOP	Jump to LOOP
F012	6F		MOV L, A	Load the Least Significant Byte
F013	62		MOV H, D	Load the Most Significant Byte
F014	22, 50, 80		SHLD 8050H	Store the BCD
F017	76		HLT	Terminate the program

Output

Address	Data

Address	Data
. . .	. . .
8050	52
. . .	. . .

8085 code to convert binary number to ASCII code

Problem – Assembly level program in 8085 which converts a binary number into ASCII number.

Program –

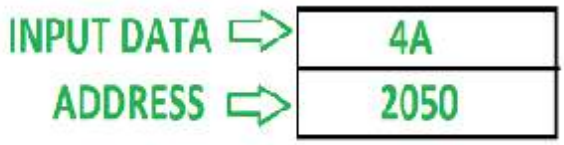
Main routine:

ADDRESS	MNEMONICS	COMMENTS
2000	LDA 2050	A<-[2050]
2003	CALL 2500	go to address 2500
2006	STA 3050	A->[3050]
2009	LDA 2050	A<-[2050]
200C	RLC	Rotate the number by one bit to left without carry
200D	RLC	Rotate the number by one bit to left without carry
200E	RLC	Rotate the number by one bit to left without carry
200F	RLC	Rotate the number by one bit to left without carry

ADDRESS	MNEMONICS	COMMENTS
2010	CALL 2500	go to address 2500
2013	STA 3051	A->[3051]
2016	HLT	Terminates the program

Sub routine:

ADDRESS	MNEMONICS	COMMENTS
2500	ANI 0F	A<-[A] AND 0F
2502	CPI 0A	[A]-0A
2504	JNC 250A	Jump to [250A] if carryflag is 0
2507	ADI 30	A<-[A]+30
2509	RET	Return to the next instruction from where subroutine address was called in main routine
250A	ADI 37	A<-[A]+37
250C	RET	Return to the next instruction from where subroutine address was called in main routine



8085 program to convert 8 bit BCD number into ASCII Code

Now let us see a program of Intel 8085 Microprocessor. This program will convert 8-bit BCDnumbers to two digit ASCII values.

Problem Statement

Write 8085 Assembly language program where an 8-bit BCD number is stored in memory location 8050H. Separate each BCD digit and convert it to corresponding ASCII code and store it to the memory location 8060H and 8061H.

Discussion

In this problem we are using a subroutine to convert one BCD digit(nibble) to its equivalent ASCII values. As the 8-bit BCD number contains two nibbles, so we can execute this subroutine to find ASCIIvalues of them. We can get the lower nibble very easily by masking the upper nibble, and for the upper nibble, we have to mask the lower nibble at first, then rotate the register content dour times to the right to make, now we can change it to ASCII values.

Here we will put 26H as input, the program will return 32 and 36. These are the ASCII values of 2 and 6 respectively.

Note: This program can also take 8-bit binary number to ASCII values.

Input

Address	Data
. . .	. . .
8050	26
. . .	. . .

Program

Address	HEX Codes	Labels	Mnemonics	Comments
8000	31, 00, 81		LXI SP, 8100	Initialize SP
8003	21, 50, 80	START	LXI H, 8050H	Initialize pointer with the first location of IN-BUFFER

Address	HEX Codes	Labels	Mnemonics	Comments
8006	11, 60, 80		LXI D, 8060H	Initialize pointer with the first location of OUT-BUFFER
8009	7E		MOV A, M	Move the contents of 8050H to A
800A	47		MOV B, A	Copy A to B
800B	0F		RRC	Rotate accumulator right 4 times
800C	0F		RRC	
800D	0F		RRC	
800E	0F		RRC	
800F	CD, 1A, 80		CALL ASCII	This subroutine converts a binary no. toASCII
8012	12		STAX D	Store the contents of the accumulator specified the contents by DE register pair
8013	13		INX D	Go to next location
8014	78		MOV A, B	Copy B to A
8015	CD, 1A, 80		CALL ASCII	This subroutine converts a binary no. toASCII
8018	12		STAX D	Store the contents of the accumulator specified the contents by DE register pair

Address	HEX Codes	Labels	Mnemonics	Comments
8019	76		HLT	Terminate the program
801A	E6, 0F	ASCII	ANI 0FH	Converts a BCD number to its corresponding ASCII value + 48 0 To 9 ----------à48 To 57 + 55 A To F ----------à 65 To 70 + 48 +7 So +48 is common but if the hex digit is between A to F then +7 is additional.
801C	FE, 0A		CPI 0AH	
801E	DA, 23, 80		JC CODE	
8021	C6, 07		ADI 07H	
8023	C6, 30	CODE	ADI 30H	
8025	C9		RET	Returning control to the calling program

Output

Address	Data
. . .	. . .

Address	Data
8060	32
8061	36
. . .	. . .

8085 code to convert binary number to ASCII code

Now let us see a program of Intel 8085 Microprocessor. This program will convert binary or hexadecimal number to ASCII values.

Problem Statement

Write 8085 Assembly language program to convert binary or Hexadecimal characters to ASCII values.

Discussion

We know that the ASCII of number 00H is 30H (48D), and ASCII of 09H is39H (57D). So all other numbers are in the range 30H to 39H.TheASCII value of 0AH is 41H (65D) and ASCII of 0FH is 46H (70D), so all other alphabets (B, C, D, E, F) are in the range 41H to 46H.

Here we are providing hexadecimal digit at memory location 8000H, The ASCII equivalent is storing at location 8001H.

The logic behind HEX to ASCII conversion is very simple. We are just checking whether the number is in range 0 – 9 or not. When the number is in that range, then the hexadecimal digit is numeric, and we are just simply adding 30H with it to get the ASCII value. When the number is not in range 0 – 9, then the number is range A – F,so for that case, we are converting the number to 41H on wards.

In the program at first we are clearing the carry flag. Then subtracting 0AHfrom the given number. If the value is numeric, then after subtraction the result will be negative, so the carry flag will beset. Now by checking the carry status we can just add 30H with the value to get ASCII value.

In other hand when the result of subtraction is positive or 0, then we are adding 41H with the result of the subtraction.

Input

first input

Address	Data
. . .	. . .
8000	0A
. . .	. . .

second input

Address	Data
. . .	. . .
8000	05
. . .	. . .

third input

Address	Data
. . .	. . .
8000	0F
.	.

Address	Data
. .	. .

Program

Address	HEX Codes	Labels	Mnemonics	Comments
F000	21, 00, 80		LXI H, 8000H	Load address of the number
F003	7E		MOV A,M	Load Acc with the data from memory
F004	47		MOV B,A	Copy the number into B
F005	37		STC	Set CarryFlag
F006	3F		CMC	ComplementCarry Flag
F007	D6, 0A		SUI 0AH	Subtract 0AHfrom A
F009	DA, 11, F0		JC NUM	When carry is present, A is numeric
F00C	C6, 41		ADI 41H	Add 41H forAlphabet
F00E	C3, 14, F0		JMP STORE	Jump to store the value
F011	78	NUM	MOV A, B	Get back B toA
F012	C6		ADI 30H	Add 30H withA to get ASCII
F014	23	STORE	INX H	Point to next location to store address

Address	HEX Codes	Labels	Mnemonics	Comments
F015	77		MOV M,A	Store A to memory location pointed by HL pair
F016	76		HLT	Terminate the program

Output

first output

Address	Data
. . .	. . .
8001	41
. . .	. . .

second output

Address	Data
. . .	. . .
8001	35
. . .	. . .

third output

Address	Data
. . .	. . .
8001	46
. . .	. . .

Program to convert ASCII to binary in 8085 Microprocessor

Here we will see one 8085 program, the program will convert ASCII to binary values.

Problem Statement–

Write an 8085 Assembly level program to convert ASCII to binary or Hexadecimal character equivalent values.

Discussion–

The ASCII of number 00H is 30H (48D), and ASCII of 09H is 39H (57D). So all other numbers are in the range 30H to 39H. The ASCII value of 0AH is 41H (65D) and ASCII of 0FH is 46H (70D), so all other alphabets (B, C, D, E, F) are in the range 41H to 46H.

Here the logic is simple. We will check whether the ASCII value is less than 58H (ASCII of 9 + 1) When the number is less 58, then it is numeric value. So we simply subtract 30H from the ASCII value, and when it is greater than 58H, then it is alphabetical value. So for that we are subtracting 37H.

Input

first input

Address	Data
…	…
8000	41

Address	Data
…	…

second input

Address	Data
…	…
8000	35
…	…

third input

Address	Data
…	…
8000	46
…	…

Program

Address	HEX Codes	Labels	Mnemonics	Comments
F000	21, 00, 80		LXI H, 8000H	Load address of the number
F003	7E		MOV A,M	Load ASCII data to Acc from memory

Address	HEX Codes	Labels	Mnemonics	Comments
F004	FE, 58		CPI 58H	Compare with ASCII(9) + 1
F006	D2, 0E, F0		JNC NUM	The input is numeric
F009	D6, 37		SUI 37H	Subtract offset to get Alphabetic character
F00B	C3, 10, F0		JMP STORE	Store the result
F00E	D6, 30	NUM	SUI 30H	Subtract 30 to get numeric value
F010	23	STORE	INX H	Point to next location
F011	77		MOV M,A	Store Acc content to memory
F012	76		HLT	Terminate the program

Output

first output

Address	Data
…	…
8001	0A
…	…

Second Output

Address	Data
…	…
8001	05
…	…

third output

Address	Data
…	…
8001	0F
…	…

UNIT - IV

Programming in 8085:BCD Arithmetic - BCD addition and Subtraction - Multibyte Addition and Subtraction - Multiplication and Division. Interrupts: The 8085 Interrupt – 8085 Vectored Interrupts BCD Addition

In this program we will see how to add two 8-bit BCD numbers.

Problem Statement

Write 8085 Assembly language program to add two 8-bit BCD number stored in memory location 8000H – 8001H.

Discussion

This task is too simple. Here we are taking the numbers from memory and after adding we need to put DAA instruction to adjust the accumulator content to decimal form. The DAA will check the AC and CY flags to adjust a number to its decimal form.

Input

Address	Data
...	...
8000	99
8001	25
...	...

Program

Address	HEX Codes	Labels	Mnemonics	Comments
F000	21, 00, 80		LXI H,8000H	Point to first operand
F003	7E		MOV A, M	Load A with first operand
F004	23		INX H	Point to next operand
F005	86		ADD M	Add Acc and memory element
F006	27		DAA	Adjust decimal
F007	21, 50, 80		LXI H,8050H	Locate destination address
F00A	77		MOV M, A	Store the result into memory

Address	HEX Codes	Labels	Mnemonics	Comments
F00B	D2, 12, F0		JNC DONE	If CY = 0, jump to Done
F00E	3E, 01		MVI A, 01H	Load 01H into Acc
F010	23		INX H	Point to next location
F011	77		MOV M,A	Store the carry
F012	76	DONE	HLT	Terminate the program

Output

Address	Data
...	...
8050	25
8051	01
...	...

BCD subtractions

Here we will see how to perform BCD subtractions using 8085.

Problem Statement

Write 8085 Assembly language program to perform BCD subtractions of tow numbers stored at location 8001 and 8002. The result will be stored at 8050 and 8051.

Discussion

To subtract two BCD numbers, we are going to use the 10s complement method. Taking the first number and storing into B, Load 99 into A then subtract the number to get the 9's complement. After that add 1 with the result to get 10's complement. We cannot increase using INR instruction. This does not effect on CY flag. So we have to use ADI 01. Then DAA instruction will be used to adjust the decimal. Then if the result is negative we are storing FF as upper byte, otherwise 00 as upper byte.

Input

Address	Data
...	...
8000	01
8001	97
8002	88
...	...

Program

Address	HEX Codes	Labels	Mnemonics	Comments
F000	21, 01, 80		LXI H,8001H	Point to get the choice
F003	46		MOV B,M	Load operand to B
F004	3E, 99		MVI A,99H	Load A with 99H
F006	23		INX H	Point to next operand

Address	HEX Codes	Labels	Mnemonics	Comments
F007	96		SUB M	Subtract M from A
F008	C6, 01		ADI 01H	Add 01H to get 10's complement
F00A	80		ADD B	Add B with A
F00B	27		DAA	Adjust decimal
F00C	6F		MOV L,A	Store A to L
F00D	DA, 3A, F0		JC SKP2	If CY = 1, jump to SKP2
F010	26, FF		MVI H,FFH	Load H with FFH
F012	C3, 62, F0		JMP STORE	Store result
F015	26, 00	SKP2	MVI H,00H	Clear HL
F017	22, 50, 80	STORE	SHLD 8050H	Store result from HL
F01A	76		HLT	Terminate the program

Output

Address	Data
...	...

Address	Data
8050	09
8051	00
...	...

8085 Program to Add two multi-byte BCD numbers

Now let us see a program of Intel 8085 Microprocessor. This program is mainly for adding multi-digit BCD (Binary Coded Decimal) numbers.

Problem Statement

Write 8085 Assembly language program to add two multi-byte BCD (Binary Coded Decimal) numbers.

Discussion

We are using 4-byte BCD numbers. The numbers are stored into the memory at location 8501H and8505H. One additional information is stored at location 8500H. In this place, we are storing the byte count. The result is stored at location 85F0H.

The HL pair is storing the address of first operand bytes, the DE is storing the address of second operand bytes. C is holding the byte count. We are using the stack to store the intermediate bytes of the result. After completion of the addition operation, we are popping from the stack and storing into the destination.

Input

Address	Data
. . .	. . .
8500	04

Address	Data
8501	19
8502	68
8503	12
8504	85
8505	88
8506	25
8507	17
8508	20
. . .	. . .

Program

Address	HEX Codes	Labels	Mnemonics	Comments
F000	31,00, 20		LXI SP, 2000H	Initialize Stack Pointer
F003	21,00, 85		LXI H,8500H	load memory address to get byte count

Address	HEX Codes	Labels	Mnemonics	Comments
F006	4E		MOV C,M	load memory content into C register
F007	06,00		MVI B,00H	clear B register
F009	21,01, 85		LXI H, 8501H	load first argument address
F00C	11,05, 85		LXI D, 8505H	load second argument address
F00F	1A	LOOP	LDAX D	load DE with second operand address
F010	8E		ADC M	Add memory content and carry with Acc
F011	27		DAA	Decimal adjust the acc content
F012	F5		PUSH PSW	Store the accumulator content into the stack
F013	4		INR B	increase b after pushing into a stack
F014	23		INX H	Increase HL pair to point next address
F015	13		INX D	Increase DE pair to point next address
F016	0D		DCR C	Decrease c to while all bytes are not exhausted
F017	C2,0F, F0		JNZ LOOP	When bytes are not considered, loop again

Address	HEX Codes	Labels	Mnemonics	Comments
F01A	D2,21, F0		JNC SKIP	when carry = 0, jump to store
F01D	3E,01		MVIA, 01H	when carry = 1, push it into stack
F01F	F5		PUSH PSW	Store the accumulator content into the stack
F020	04		INR B	increase b after pushing into the stack
F021	21,F0, 85	SKIP	LXIH, 85F0H	load the destination pointer
F024	F1	L1	POP PSW	pop AF to get back bytes from the stack
F025	77		MOV M, A	store Acc data at the memory location pointed by HL
F026	23		INX H	Increase HL pair to point next address
F027	05		DCR B	Decrease B
F028	C2,24, F0		JNZ L1	Goto L1 to store stack contents
F02B	76		HLT	Terminate the program

Output

Address	Data

Address	Data
. . .	. . .
85F0	01
85F1	05
85F2	29
85F3	94
85F4	07
. . .	. . .

Program for subtraction of multi-byte BCD numbers in 8085 Microprocessor

Here we will see one program that can perform subtraction for multi-byte BCD numbers using 8085 microprocessor.

Problem Statement −

Write an 8085 Assembly language program to subtract two multi-byte BCD numbers.

Discussion −

The numbers are stored into memory, and one additional information is stored. It will show us the byte count of the multi-byte BCD number. Here we are choosing 3-byte BCD numbers. They are stored at location 8001H to 8003H, and another number is stored at location 8004H to 8006H. The location 8000H is holding the byte count. In this case the byte count is 03H.

For the subtraction we are using the 10’s complement method for subtraction.

In this case the numbers are: 672173 – 275188 = 376985

Input

Address	Data
…	…
8000	03
8001	73
8002	21
8003	67
8004	88
8005	51
8006	27
…	…

Program

Address	HEX Codes	Labels	Mnemonics	Comments
F000	21, 00, 80		LXI H,8000H	Point to get the count
F003	4E		MOV C,M	Get the count to C

Address	HEX Codes	Labels	Mnemonics	Comments
F004	11, 01, 80		LXI D,8001H	Point to first number
F007	21, 04, 80		LXI H,8004H	Point to second number
F00A	37		STC	Set the carry flag
F00B	3E, 99	LOOP	MVI A,99H	Load 99H into A
F00D	CE,00		ACI 00H	Add 00H and Carry with A
F00F	96		SUB M	Subtract M from A
F010	EB		XCHG	Exchange DE and HL
F011	86		ADD M	Add M to A
F012	27		DAA	Decimal adjust
F013	77		MOV M,A	Store A to memory
F014	EB		XCHG	Exchange DE and HL
F015	23		INX H	Point to next location by HL
F016	13		INX D	Point to next location by DE
F017	0D		DCR C	Decrease C by 1

Address	HEX Codes	Labels	Mnemonics	Comments
F018	C2, 0B, F0		JNZ LOOP	Jump to LOOP if Z = 0
F01B	76		HLT	Terminate the program

Output

Address	Data
…	…
8001	85
8002	69
8003	37
…	…

8085 Program to multiply two 2-digit BCD numbers

Now let us see a program of Intel 8085 Microprocessor. This program will find the multiplication result of two BCD numbers.

Problem Statement

Write 8085 Assembly language program to find two BCD number multiplication. The numbers are stored at location 8000H and 8001H.

Discussion

In this program the data are taken from 8000H and 8001H. The result is stored at location

8050H and 8051H.

As we know that 8085 has no multiply instruction so we have to use repetitive addition method. In this process after each addition we are adjusting the accumulator value to get decimal equivalent. When carry is present, we are incrementing the value of MS-Byte. We can use INR instruction for incrementing, but here ADI 01H is used. The INR instruction does not affect the CY flag so we need ADI instruction.

Input

first input

Address	Data
. . .	. . .
8000	12
8001	20
. . .	. . .

second input

Address	Data
. . .	. . .
8000	27
8001	03
.	.

Address	Data
. .	. .

Program

Address	HEX Codes	Labels	Mnemonics	Comments
F000	21, 00, 80		LXI H,8000H	Load first operand address
F003	46		MOV B, M	Store first operand to B
F004	23		INX H	Increase HLpair
F005	4E		MOV C, M	Store second operand to register C
F006	1E, 00		MVI E, 00H	Clear register E
F008	63		MOV H, E	Clear H register
F009	7B		MOV A, E	Clear A register
F00A	B9		CMP C	Compare C with A
F00B	CA, 23, F0		JZ DONE	When Z = 0,jump to DONE
F00E	80	LOOP	ADD B	Add B with A
F00F	27		DAA	Decimal Adjust

Address	HEX Codes	Labels	Mnemonics	Comments
F010	57		MOV D, A	Store A to D
F011	D2, 19, F0		JNC NINC	Jump tp NINC
F014	7C		MOV A, H	Store H to A
F015	C6, 01		ADI 01H	Increase A by1
F017	27		DAA	DecimalAdjust
F018	67		MOV H, A	Restore H from A
F019	7B	NINC	MOV A, E	Load E to A
F01A	C6, 01		ADI 01H	Increase A by1
F01C	27		DAA	Decimal adjust
F01D	5F		MOV E, A	Restore E from A
F01E	B9		CMP C	Compare C with A
F01F	7A		MOV A,D	Load D to A
F020	C2, 0E, F0		JNZ LOOP	Jump to LOOP
F023	6F	DONE	MOV L, A	Load A to L

Address	HEX Codes	Labels	Mnemonics	Comments
F024	22, 50, 80		SHLD 8050H	Store HL pair at location 8050 and 8051
F027	76		HLT	Terminate the program

Output

first output

Address	Data
. . .	. . .
8050	40
8051	02
. . .	. . .

second output

Address	Data
. . .	. . .
8050	81

Address	Data
8051	00
.	.
.	.
.	.

8085 Program to Divide two 8 Bit numbers

In this program, we will see how to divide two 8-bit numbers using 8085 microprocessor.

Problem Statement

Write 8085 Assembly language program to divide two 8-bit numbers and store the result at locations **8020H** and **8021H**.

Discussion

The 8085 has no division operation. To get the result of the division, we should use the repetitive subtraction method.

By using this program, we will get the quotient and the remainder. 8020H will hold the quotient, and 8021H will hold the remainder.

We are saving the data at location 8000H and 8001H. The result is storing at location 8050H and 8051H.

Input

The Dividend: 0EH

The Divisor 04H

The Quotient will be 3, and the remainder will be 2

Program

Address	HEX Codes	Labels	Mnemonics	Comments

Address	HEX Codes	Labels	Mnemonics	Comments
F000	21,0E, 00	START	LXIH,0CH	Load 8-bit dividend in HL register pair
F003	06,04		MVIB,04H	Load divisor in B to perform num1 / num2
F005	0E,08		MVIC, 08	Initialize the counter
F007	29	UP	DADH	Shifting left by 1 bit HL = HL + HL
F008	7C		MOVA, H	Load H in A
F009	90		SUB B	perform A = A – B
F00A	DA,0F, F0		JC DOWN	If MSB < divisor then shift to left
F00D	67		MOVH, A	If MSB > divisor, store the current value of A in H
F00E	2C		INR L	Tracking quotient
F00F	0D	DOWN	DCRC	Decrement the counter
F010	C2,07, F0		JNZ UP	If not exhausted then go again
F013	22,20, 80		SHLD 8020	Store the result at 8020 H
F016	76		HLT	Stop

Output

Address	Data
. . .	. . .
8020	03
8021	02
. . .	. . .

Interrupts in 8085 microprocessor and its type:

When microprocessor receives any interrupt signal from peripheral(s) which are requesting its services, it stops its current execution and program control is transferred to a sub-routine by generating CALL signal and after executing subroutine by generating RET signal again program control is transferred to main program from where it had stopped. When microprocessor receives interrupt signals (INTR), it sends an acknowledgement (INTA) to the peripheral which is requesting for its service.

Interrupts can be classified into various categories based on different parameters.

• Hardware and Software Interrupts –

When microprocessors receive interrupt signals through pins (hardware) of microprocessor, they are known as Hardware Interrupts. There are 5 Hardware Interrupts in 8085 microprocessor. They are – INTR, RST 7.5, RST 6.5, RST 5.5, TRAP.

Software Interrupts are those which are inserted in between the program which means these are mnemonics of microprocessor. There are 8 software interrupts in 8085 microprocessor. They are – RST 0, RST 1, RST 2, RST 3, RST 4, RST 5, RST 6, RST 7.

• Vectored and Non-Vectored Interrupts –

Vectored Interrupts are those which have fixed vector address (starting address of sub-routine) and after executing these, program control is transferred to that address.

Non-Vectored Interrupts (Scalar Interrupt) are those in which vector address is not predefined. The interrupting device gives the address of sub-routine for these interrupts. INTR is the only non-vectored interrupt in 8085 microprocessor.

• Maskable and Non-Maskable Interrupts –

Maskable Interrupts are those which can be disabled or ignored by the microprocessor. These

interrupts are either edge-triggered or level-triggered, so they can be disabled. INTR, RST 7.5, RST 6.5, RST 5.5 are maskable interrupts in 8085 microprocessor.
Non-Maskable Interrupts are those which cannot be disabled or ignored by microprocessor. TRAP is a non-maskable interrupt. It consists of both level as well as edge triggering and is used in critical power failure conditions.

UNIT - V

Direct Memory Access(DMA)and 8257 DMA controller - 8255A Programmable Peripheral Interface. Basic features of Advanced Microprocessors - Pentium - I3 , I5 and I7

Direct Memory Access (DMA) transfers the block of data between the *memory* and *peripheral devices* of the system, **without the participation** of the **processor**. The unit that controls the activity of accessing memory directly is called a **DMA controller**

What is DMA and Why it is used?

Direct memory access (DMA) is a **mode of data transfer** between the memory and I/O devices. This happens **without the involvement** of the processor. We have two other methods of data transfer, **programmed I/O** and **Interrupt driven I/O**. Let's revise each and get acknowledge with their drawbacks.

In **programmed I/O**, the processor keeps on scanning whether any device is ready for data transfer. If an I/O device is ready, the processor **fully dedicates** itself in transferring the data between I/O and memory. It transfers data at a **high rate, but it can't get involved in any other activity** during data transfer. This is the major **drawback** of programmed I/O.

In **Interrupt driven I/O**, whenever the device is ready for data transfer, then it raises an **interrupt to processor**. Processor completes executing its ongoing instruction and saves its current state. It then switches to data transfer which causes a **delay**. Here, the processor doesn't keep scanning for peripherals ready for data transfer. But, it is **fully involved** in the data transfer process. So, it is also not an effective way of data transfer.

The above two modes of data transfer are not useful for transferring a large block of data. But, the DMA controller completes this task at a faster rate and is also effective for transfer of large data block.

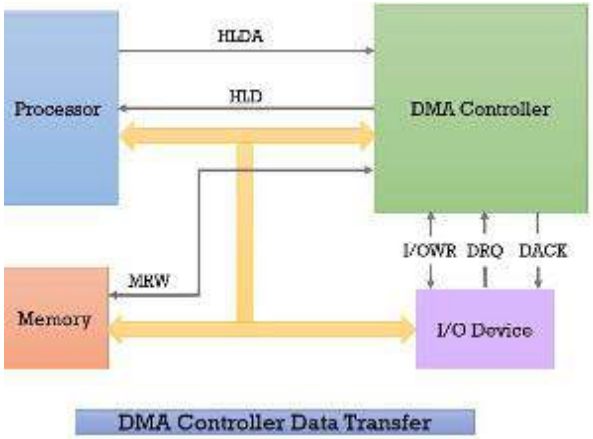
The DMA controller transfers the data in three modes:

1. **Burst Mode:** Here, once the DMA controller gains the charge of the system bus, then it releases the system bus only after **completion** of data transfer. Till then the CPU has to wait for the system buses.
2. **Cycle Stealing Mode:** In this mode, the DMA controller **forces** the CPU to stop its operation and **relinquish the control over the bus** for a **short term** to DMA controller. After the **transfer of every byte**, the DMA controller **releases** the **bus** and then again requests for the system bus. In this way, the DMA controller steals the clock cycle for transferring every byte.

3. **Transparent Mode:** Here, the DMA controller takes the charge of system bus only if the **processor does not require the system bus**.

Direct Memory Access Controller & it's Working

DMA controller is a **hardware unit** that allows I/O devices to access memory directly without the participation of the processor. Here, we will discuss the working of the DMA controller. Below we have the diagram of DMA controller that explains its working:

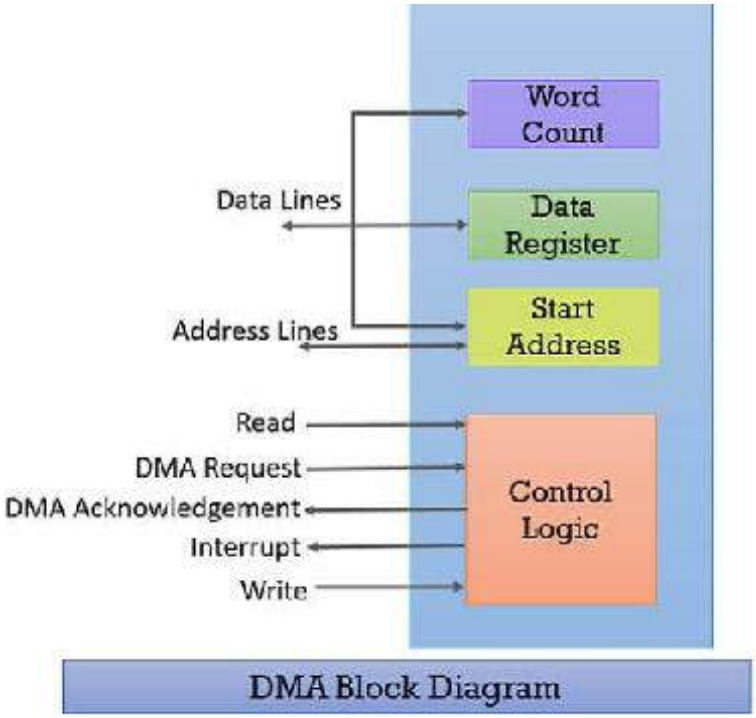


1. Whenever an I/O device wants to transfer the data to or from memory, it sends the DMA request (**DRQ**) to the DMA controller. DMA controller accepts this DRQ and asks the CPU to hold for a few clock cycles by sending it the Hold request (**HLD**).
2. CPU receives the Hold request (HLD) from DMA controller and relinquishes the bus and sends the Hold acknowledgement (**HLDA**) to DMA controller.
3. After receiving the Hold acknowledgement (HLDA), DMA controller acknowledges I/O device **(DACK)** that the data transfer can be performed and DMA controller takes the charge of the system bus and transfers the data to or from memory.
4. When the data transfer is accomplished, the DMA raise an **interrupt** to let know the processor that the task of data transfer is finished and the processor can take control over the bus again and start processing where it has left.

Now the DMA controller can be a separate unit that is shared by various I/O devices, or it can also be a part of the I/O device interface.

Direct Memory Access Diagram

After exploring the working of DMA controller, let us discuss the block diagram of the DMA controller. Below we have a block diagram of DMA controller.



Whenever a processor is requested to read or write a block of data, i.e. transfer a block of data, it instructs the DMA controller by sending the following information.

1. The first information is whether the data has to be read from memory or the data has to be written to the memory. It passes this information via **read or write control lines** that is between the processor and DMA controllers **control logic unit**.
2. The processor also provides the **starting address** of/ for the data block in the memory, from where the data block in memory has to be read or where the data block has to be written in memory. DMA controller stores this in its **address register**. It is also called the **starting address register**.
3. The processor also sends the **word count**, i.e. how many words are to be read or written. It stores this information in the **data count** or the **word count** register.
4. The most important is the **address of I/O device** that wants to read or write data. This information is stored in the **data register.**

Direct Memory Access Advantages and Disadvantages

Advantages:

1. Transferring the data without the involvement of the processor will **speed up** the read-write task.
2. DMA **reduces the clock cycle** requires to read or write a block of data.
3. Implementing DMA also **reduces the overhead** of the processor.

Disadvantages

1. As it is a hardware unit, it would **cost** to implement a DMA controller in the system.
2. Cache **coherence** problem can occur while using DMA controller.

8255A PROGRAMMABLE PERIPHERAL INTERFACE

The 8255A is a general purpose programmable I/O device designed to transfer the data from I/O to interrupt I/O under certain conditions as required. It can be used with almost any microprocessor.

It consists of three 8-bit bidirectional I/O ports (24I/O lines) which can be configured as per the requirement.

Ports of 8255A

8255A has three ports, i.e., PORT A, PORT B, and PORT C.

- **Port A** contains one 8-bit output latch/buffer and one 8-bit input buffer.
- **Port B** is similar to PORT A.
- **Port C** can be split into two parts, i.e. PORT C lower (PC0-PC3) and PORT C upper (PC7-PC4) by the control word.

These three ports are further divided into two groups, i.e. Group A includes PORT A and upper PORT C. Group B includes PORT B and lower PORT C. These two groups can be programmed in three different modes, i.e. the first mode is named as mode 0, the second mode is named as Mode 1 and the third mode is named as Mode 2.

Operating Modes

8255A has three different operating modes −

- **Mode 0** − In this mode, Port A and B is used as two 8-bit ports and Port C as two 4-bit ports. Each port can be programmed in either input mode or output mode where outputs are latched and inputs are not latched. Ports do not have interrupt capability.
- **Mode 1** − In this mode, Port A and B is used as 8-bit I/O ports. They can be configured as either input or output ports. Each port uses three lines from port C as handshake signals. Inputs and outputs are latched.
- **Mode 2** − In this mode, Port A can be configured as the bidirectional port and Port B either in Mode 0 or Mode 1. Port A uses five signals from Port C as handshake signals for data transfer. The remaining three signals from Port C can be used either as simple I/O or as handshake for port B.

Features of 8255A

The prominent features of 8255A are as follows −

- It consists of 3 8-bit IO ports i.e. PA, PB, and PC.
- Address/data bus must be externally demux'd.

- It is TTL compatible.
- It has improved DC driving capability.

Features of Pentium Processors:

• It is a highly integrated device containing about 1.2 million transistors.
•Wider Data Bus Width: The Pentium processors have a wider data bus width. The data bus width has been increased from 32-bit to 64-bit to improve the data transfer rate.
•Faster Floating Point Unit: Faster algorithm provides up to ten times speed-up for common operations including add, multiply and load.
• Improved Cache Structure: Pentium processors include separate code and data caches integrated on-chip to meet performance goals.
•Dual Integer Processor: Pentium processor has integer processor. It allows execution of two instructions per clock.
•Branch Prediction Logic: The Pentium uses the technique called branch prediction to check whether a branch will be valid or invalid.
•Data Integrity and Error Detection: The Pentium processors have added significant data integrity and error detection capability.
•Super Scalar Processor: Processors capable to parallel instruction execution of multiple instructions are known as super scalar processors.

The Pentium III

The Pentium III The Pentium III microprocessor is an improved version of the Pentium II microprocessor. Even though it is newer than the Pentium II, it is still based on the Pentium Pro architecture.
The salient architectural features are:
1. P-III CPU has been developed using 0.25 micron technology and includes over 9.5 million transistors. It has three versions operating at 450 MHz, 500 MHz and 550 MHz which are commercially available.
2. P-III incorporates multiple branch prediction algorithms.
3. Seventy new instructions have been added to Pentium III. These instructions are useful in advanced imaging, speech processing and multimedia applications.
4. Dual independent bus architecture increases bandwidth.
5. P-III employs dynamic execution technology.
6. A 512Kbyte unified, non-blocking level 2 cache has been used.
7. Eight 64-bit wide Intel MMX registers along with a set of 57 instructions for multimedia applications are available.

Features of Core i5 Processor

Here, are important features of Core i5 Processor:

- i5 processors offer an ability to work with integrated Memory, which helps to hence the performance of the applications.
- It increases the Memory speed up to 1333 MHz
- i5 processors have a rapid performance rate. So, it can perform at the maximum CPU rate of 3.6 GHz
- Turbo technology present in the i5 Processor helps you to boost up the working speed of the computational systems.
- I5 processer uses 64-bit architecture for the users for reliable working.

Advantages of i5 processors

Here, are pros/benefits of using i5 processors:

- It has a high-speed performing rate so that system are able to perform at the maximum CPU rate of 3.6 GHz
- Turbo technology is present in the device which helps you to boost up the working speed
- It offers 64-bit architecture to get reliable working.

Disadvantages of i5 processors

Here, cons/drawback of Cori i5 Processor

- Not support high data visualization technology for users to view high-quality images and video graphics.
- Power consumption of core-i7 is not better compared to the core-2 duo processor type.
- It demands newer motherboards.
- i5 Processor is sensitive to higher voltages.

Features of Core i7 Processor

Here, are essential features of Core i7 Processor:

- Supports 64-bit execution
- Front Side bus Speed include 2GH
- High speed working with the multitasking feature
- i7 offers a feature of hyper-threading technology
- Support DDR3 main memory

Advantages of Core i7 Processor

Here, are pros/benefits of Core i7 Processor

- Very fast processing speed
- Offer highly reliable cooling system
- Four cores allow for handling software that requires lots of computations.
- Provide high data visualization to users that help them to get high-quality images and video graphics.
- The ideal Processor for gaming enthusiasts and digital artists

Disadvantages of Core i7 Processors

Here, are cons/drawbacks of using i7 Processors

- Relatively costly Processor
- Power consumption is high compared to other processors.
- i7 processors work only with DDR3 Memory, which means that users upgrading from DDR2 will require to have a new motherboard.
- Not many software needed for multithreading, which means the average users do not get much performance gain.

*****