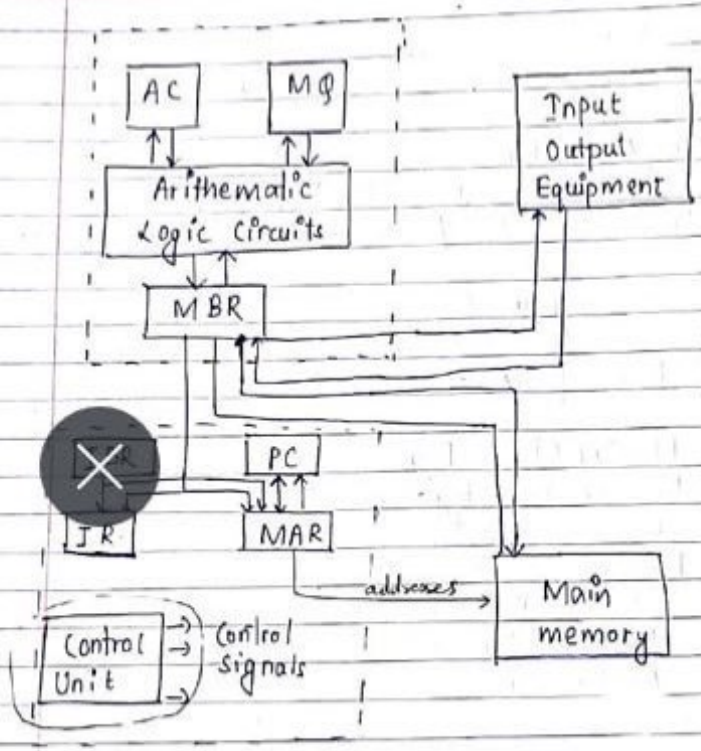
Computer Organizations
↳ how features are implemented
eg: control signals, interfaces, memory technology

Computer Architecture
↳ attributes that is visible to programmer
eg: instruction set, I/o mechanisms, addressing techniques

Evolution of Computer families

IAS (Institute for Advance Studies)

MAR → Memory Address Register
MBR → Memory Buffer Register
IR → Instruction register
PC → Program Counter
IBR → Instruction Buffer Register
MQ → Multiplier Quotient
Ac → Accumulator



Addressing modes

1. Immediate
2. Direct
3. Indirect
4. Register
5. Register indirect
6. Displacement

4. Stack

1. Immediate

Opcode	Operand

→ operand is present in instruction.

Adv.

→ fast

→ no memory reference to fetch data.

disad.

→ limited range.

2. Direct

Opcode	Address (A)

operand = (A) contents of A → operand

EA = A (effective address)

memory

adv

→ single memory reference to fetch data

→ no additional calculations to workout effective addres

dis

→ limited address space

3. Indirect

Memory

Opcode	Address

→ pointer to operand

→ operand

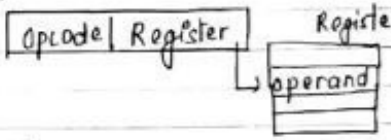
operand = (A)

EA = (A)

Adv
- Large address space
- 2^n addressable cells where n is no. of bits in the memory cell

Disad
- Multiple access to memory to find operand
- slower.

iv) Register

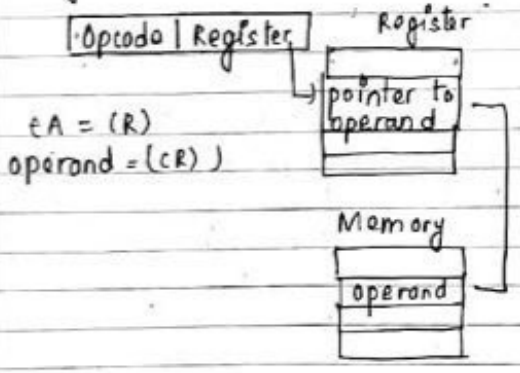


EA = R
Operand = (R)

adv
-) short instructions
→ fast instruction fetch
→ no main memory reference

dis
→ very limited address space
It is similar to direct addressing mode but working on register requires low clock cycles.

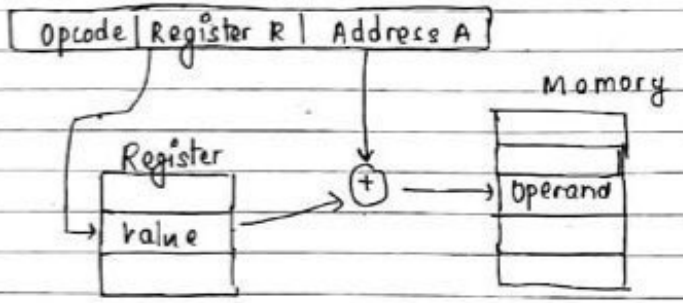
v) Register indirect



EA = (R)
operand = ((R))

adv
- faster, uses memory only once.

vi) Displacement



EA = A + (R)
Operand = (A + (R))

→ address field contain two values register name R and one is value A

→ Three types of register are used hence three uses:
(i) Relative addressing (here R=PC)
(ii) Base addressing (R holds pointer to base address)
(iii) Indexed addressing (R holds the index i.e can access all array cells in sequence)

4. Stack
   - top of stack

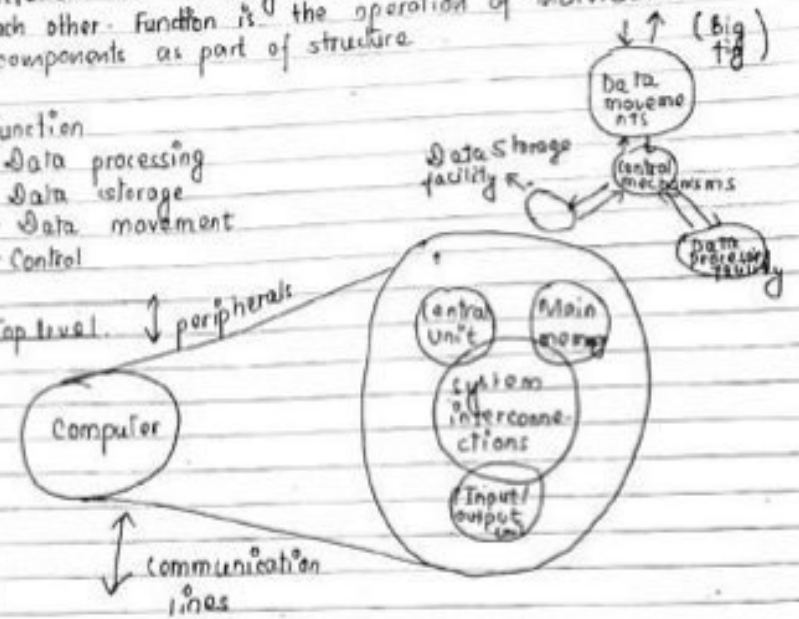
   eg: S.e Add (popping items from stack and add)

→ operand is on the top of the stack.

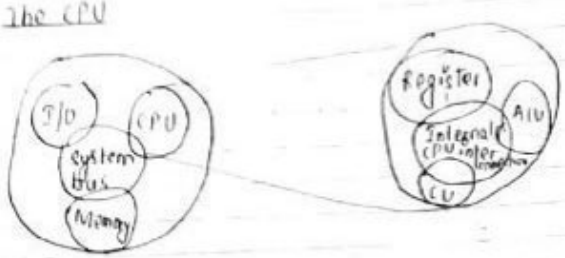
* Structure of Computer

Structure is the way in which components relate to each other. Function is the operation of individual components as part of structure.

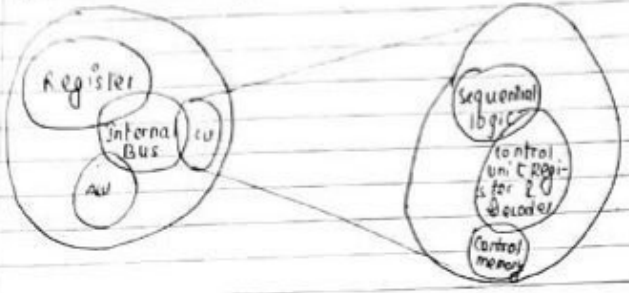
Function
- Data processing
- Data storage
- Data movement
- Control



* The CPU



* The control unit



Instruction Set & Format :-

Elements of instruction:
1. Operational code
2. Source code
3. Result operand reference
4. Next instruction reference.

The operation of CPU is determined by instructions known as machine instruction or computer inst. The collection of different instruction that CPU can execute is referred to as instruction set.

Instruction Representation
Instruction is represented by a sequence of bits. The instruction is divided into field corresponding to the consitituent element of instruction. It is difficult to differentiate by binary nos. Thus instruction are represented in symbolic form. opcodes are represented by mnemonics, that indicates operation., operands are also represented symbollically. example: Add R, Y adds the content of Y to reg. R. Thus, it is possible to write a machine language program in symbolic form. Each symbolic opcode has a fixed binary representation and the programmer specifies the location of each symbolic operand.

Instruction representation for 16-bits

opcode	operand reference	operand reference
4 bits	6 bits	6bits
16. bits		

Address format:

One of the traditional ways of describing processor architecture is in terms of no. of address contained in each instruction.
Let's think about how many no. of address might be needed in an instruction. Generally, arithematic and logical operations are either unary or binary. Thus, we need max. of two address to reference operand. To store the result, a third operand is needed & thus third address. After completion of an instruction, we need next instruction to be fetched and thus its address is needed. This says an instruction possibly requires 4 address reference: two operands, one result, next instruction address. Generally, four format is not used. one, two or three address are used with next instruction address obtained by from program counter.

eg: Y = (A - B) / (C + D * E)
Solve the equation using one, two and three address format.
Three address format :- Each instruction occupies two operand reference and one result operand reference.

eg: Instruction	Command
SUB Y, A, B	Y ← A - B
MPY T, D, E	T ← D * E
ADD T, T, C	T ← T + C
DIV Y, Y, T	Y ← Y / T

Two add

eg: Instruction	Command
SUB A, B	A ← A - B
MOV X, A	X ← A
MUL D, E	D ← D * E
MOV T, D	T ← D
ADD T, C	T ← T + C
DIV X, T	X ← X / T
MOV Y, X	Y ← X

All:

MOV Y, A	Y ← A
SUB Y, B	Y ← Y - B
MOV T, D	T ← D
MPY T, E	T ← T * E
ADD T, C	T ← T + C
DIV Y, T	Y ← Y / T

one add

eg. Instruction

SUB B
MOV AC, A
SUB B
MOV T, AC
MOV AC, D
MUL E
~~MOV T, AC~~
ADD C
MOV T

Command

AC ← A
AC ← AC - B
Y ← AC
AC ← D
AC ← AC * E
~~T ← AC~~
AC ← AC + C

Zero address format

One add

Instruction	Command
LOAD A	AC ← D
MUL E	AC ← AC*E
ADD C	AC ← AC+C
STOR Y	Y ← AC
LOAD A	AC ← A
SUB B	AC ← AC-B
DIV Y	AC ← AC*Y
STOR Y	Y ← AC

chapter-2

Register Transfer and micro- operations

A micro- operation specifies an operation whose result is stored typically in a register or memory location. The operation can be copying data from one reg to another or adding the contents of two reg and storing the third reg. When designing a sequential digital system, the designer can specify the behaviour of the system using micro-operations & then design hardware to match this specification.

Example:

Two one-bit register X and Y, The micro-operation is to copy the content of Y to X

eg: X ← Y

The micro-operation does not say how data are copied it just specifies the transfer, it could be direct connection or system bus.

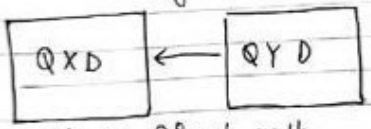


Fig (a) Direct path

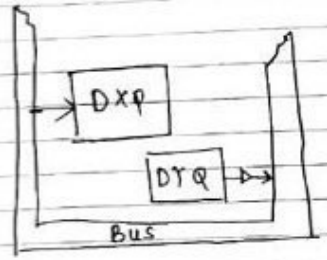


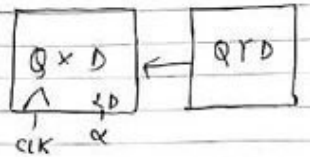
Fig: (b) Using Bus

Both are valid but system designer should take the best implementation.

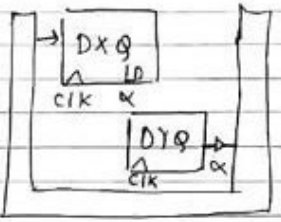
If α then X ← Y (is used in the form).

Conditions: α - operation

α: X ← Y when left side of colon is specified, data transfer takes place.



ⓐ Direct path



ⓑ Using Bus

* Indirect path α is used to load register X. In bus bo based tristate buffer is enabled by α to place the contents of Y on bus.

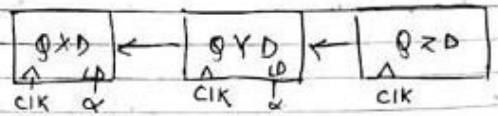
To improve system performance two or more micro-operation should be performed simultaneously. In this notation micro-operations are seperated by comma and return concurrently.

$X \leftarrow Y$ and $Y \leftarrow Z$

$X \leftarrow Y$, $Y \leftarrow Z$

if condition :

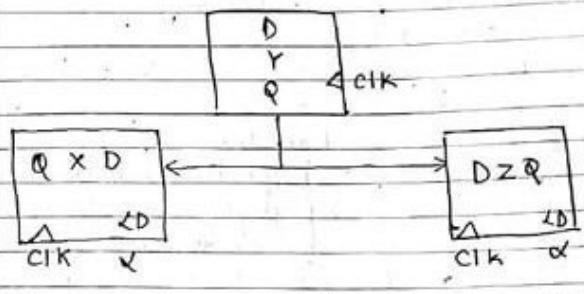
$\alpha : x \leftarrow y, Y \leftarrow z$



ⓐ Direct path

Also, data can be transfer in multiple location at a time.

i.e $\alpha : x \leftarrow Y$, $z \leftarrow Y$



But more than one value cannot be copied in same location.

* To transfer constant data into a register

$\alpha : X \leftarrow 0$

$\beta : X \leftarrow 1$

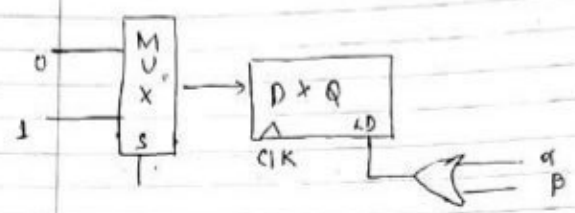


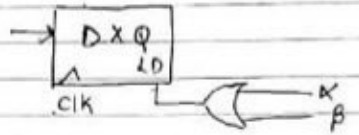
Fig: Implementation of data transfer

α : X ← 0

β : X ← 1

here,
X is set loading data for both transfer. Either α or β can assure assert the registers load signal. And MUX is used to ensure the proper data transfer.
If α = 0 and β = 0, X is 0. Since, data value is low reg. X ignores this data & retains its value.

Without MUX:



here, data load is exactly same as above but data is generated by β. Here, if α = 1 and β = 0, register load X as 0. When β = 1 and α = 0, X is loaded as 1.
if α = 0 and β = 0, the input data value is ignored by register as its load signal is not asserted.
This sceneria ignores the case when α and β = 1. In this situation, the register will try to clear its value and load 1 simultaneously which is not acceptable. Thus, this condition should be resolved.
In first case, we can assure that α and β is never 1 at some time or they should be modified as mutually exclusive.
For multi-bit register data transfer takes place directly, it is not necessary to specific individual bit transfer.
Example: X and Y are 4 bit register, α : X ← Y

4-bit X: 1111

4-bit Y: 1011

$X_3 \leftarrow Y_3$, $X_2 \leftarrow Y_2 \leftarrow X_1 \leftarrow Y_1$, $X_0 \leftarrow Y_0$

$X \leftarrow Y$

for individual bits, group of bits can be referred to as ranges in RTL (Register transfer language)
eg:
X_3, X_2, X_1 can be written as X(3-1) or X(3:1).
The individual bits are then used as follows:

α : X(3-1) ← Y(2-0)

β : $X_3 \leftarrow X_2$

T : X(3-0) ← X(2-0), X_3

Micro-operations for arithematic, logical and shift operation

There are 4 basic type of operation:
1. Linear shift
2. Circular shift
3. Arithematic shift
4. Decimal shift

1. Shift takes place one position to the left or one position to the right. The end bit is discarded & 0 takes the vacant position.
Eg: 1101
0110 right
1010 left

2. It works similar as linear shift but the difference is in the bit that is discarded in linear shift takes the vacant place.
Eg: 1101
1011 left
1110 right

3. This ~~way~~ was developed for signed no. 1 notation. here, left most bit is signed bit & remains unchanged by shift operation.
Eg: 1011
1101 left
1010 right
} change linear and keep the MSB as it is

4. It was developed for BCD representation, it acts like linear shift except its shifts one digit one bit or 4 bit
Eg: 2 7
0010 0111
~~10~~ 0111 0000 left
02 0000 0010 right

shift micro-operation

Operation	Notation
linear shift left	shl(x)
linear shift right	shr(x)
circular shift left	cil(x)
circular shift right	cir(x)
Arithematic " left	ashl(x)
" " right	ashr(x)
Decimal " left	dshl(x)
" " right	dshr(x)

Arithematic & logical operations

Operation	Eg:
Add	$X \leftarrow X+Y$
Subtract	$X \leftarrow X-Y$ or $X \leftarrow X+Y'$
Increment	$X \leftarrow X+1$
Decrement	$X \leftarrow X-1$
AND	$Y \leftarrow X \wedge Y$ or $X \leftarrow XY$
OR	$X \leftarrow X \vee Y$

XOR $x \leftarrow x \oplus y$
NOT $x \leftarrow /x$ or $x \leftarrow x'$

For data-transfer betⁿ register & memory, it is necessary to specify memory & its address.

eg: M[55] ← AC
AC ← M[55]

HDL (Hardware Description Language)
VHDL (Very high speed integrated circuit hardware description language)

Three sections:
1. Library Section
2. Entity Section
3. Architecture Section

It was developed to provide a standard for designing digital system. VHDL signifies a formal syntax. The designer creates design files using that syntax and then synthesis it. It also checks for error in syntax, but not in logic. VHDL was developed to meet the need for a standard hardware description language. It used to design custom ICs as well as application specific ICs as well as to fit designs into existing programming local devices.

Advantages
1) Portability
2) Device independent
3) Can be stimulated before implementation in hardware

VHDL design
1) Library section
2) Entity section
3) Architecture section

Library declaration consists of statements that specify libraries to be accessed and modules of these libraries. Commonly used library is IEEE.
use IEEE.std_logic_1164.all;
⇓
for input output logic

In entity section, the designer specifies the name of design and its input & output.

```
entity module-name is
port(input : in std_logic;
     output : out std_logic;);
end module-name;
```

The final section of VHDL design is architecture section. This section specifies, the behaviour and internal logic of system under design.

architecture arch-name of module-name is type and additional signal declarations;
begin
statements defining behaviour/logic;
end arch-name;

type and → name of architecture and entity to which it belongs
additional → Any new types and new signals local to this architecture

chapter-4

Computer Arithmetic.

Number Representation
→ Integer number
→ Floating-point number

Integer Number Representation
↳ Unsigned representation / notation
↳ Signed representation / notation

Unsigned representation
- This notation doesn't have a separate bit to represent the sign of number. Number can be either positive or negative in some unsigned notation. Two commonly used unsigned notation are:
  1) non-negative notation
  2) Two's complement notation.

1) Non-negatives
It treats every no. as either 0 or oppositive value ranging from 0 to 2^{n-1}
In general for n-bits nos a sequence of $a_{n-1}, a_{n-2}, \ldots a_0$ can be represented as unsigned no. A in the form.

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

* Integer Arithematic.

→ Basically, done for two's complement notation.

1. Negation:

For sign bit representation invert the sign bit
For two's complement representation take bitwise complement of each bit and add to the complement.

eg: +18 00010010

To negate take bitwise complement

11101101

Add 1 (2's comp) 1

11101101 1

11101111 ⇒ -18

Now, try negating -18

11101110

00010001 1

00010010 → +18

To represent in n-bit sequence of binary digits $a_{n-1}, a_{n-2}, \ldots a_0$ as a two's complement notation.

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

To negate

$\bar{a}_{n-1}, \bar{a}_{n-2} \ldots \bar{a}_0 + 1$

$$B = -2^{n-1} \bar{a}_{n-1} + \sum_{i=0}^{n-2} 2^i \bar{a}_i + 1$$

$$A + B = \begin{cases} 0 \\ \end{cases}$$

$$\Rightarrow -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i - 2^{n-1} \bar{a}_{n-1} + \sum_{i=0}^{n-2} 2^i \bar{a}_i + 1 = 0$$

$$\Rightarrow -2^{n-1}(a_{n-1} + \bar{a}_{n-1}) + 1 + \sum_{i=0}^{n-2} 2^i (a_i + \bar{a}_i) = 0$$

$$\Rightarrow -2^{n-1} + 1 + \sum_{i=0}^{n-2} 2^i$$

$$= -2^{n-1} + 1 + 2^{n-1} - 1$$

$$= 0$$

$\therefore A = -B$

Complement of A

this definition assume that we can first treat bit pattern as an unsigned integer. For the purpose of adding 1 and treat the result as 2's complement integer. There are 2 special cases to consider:

1. when A = 0, the result 0 negation of 0 is 0.
2. the second case is if we take the negation of bit pattern of 1 followed by n-1 0's then for 8 bit word.

-128 => 10000000

01111111

10000000 => -128

Thus, there is representation of -2^n but not for $+2^n$.

2. Addⁿ and Subtraction

For addⁿ and subtraction positive result is obtained in ordinary binary representation and for negative result is obtain in 2's complement form.

Eg:

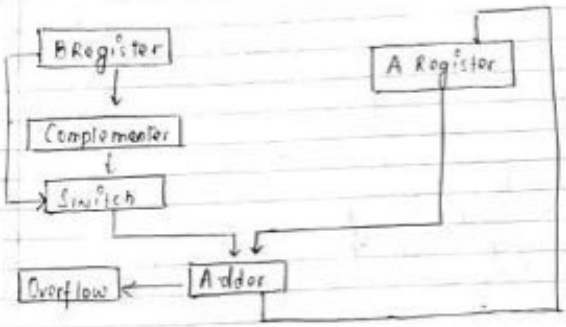
+2	0010	+2	0010
+5	0101	+7	0111
+7	0111		1001 => overflow

↓ overflow bit is set

+2	0010
-5	1011
-3	1101

In addⁿ result can be higher than word size being used. This condition is overflow. ALU should signal the overflow, so the result cannot be used. Overflow occurs if both nos and +ve or -ve. To subtract take 2's complement of subtrand and add it to minuend

Hardware Implementation



Switch => to select addition & subtraction

3. Multiplication

1) For non-negative nos

11 → 1011
13 → 1101

```
    1011
    0000 X
  1 0 1 1 V x
1 0 1 1 x x x
1 0 0 0 1 1 1 1   => 143
```

a) Algorithm type
The multiplier and multiplicant are loaded in 2 reg Q and M. A third reg 'A' is needed which is set initially to 0. A 1-bit carry register 'C' is initialized to 0 that holds carry resulting from addition

1) If $Q_0 = 1$, multiplicant is added to A reg
2) All bits of C and Q are shifted right one, so that C bit goes to A_{n-1}, A_0 goes to Q_{n-1} and Q_0 is lost.
3) If $Q_0 = 0$, no addition, only shifting is done.

C	A	Q	M
0	0000	1101	1011
	1101	1011	A←A+M

C	A	Q	M	
0	0000	1101	1011	
0	1011	1101	1011	A ← A+M
0	0101	1110	1011	shift right
0	0010	1111	1011	shift right
0	1101	1111	1011	A ← A+M
0	0110	1111	1011	shift right
1	0001	1111	1011	A ← A+M
0	1000	1111	1011	shift right

A & M

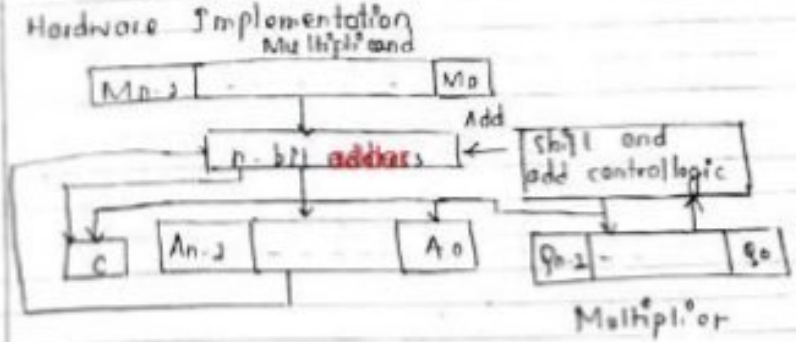
15 ← 8

15 → 1111

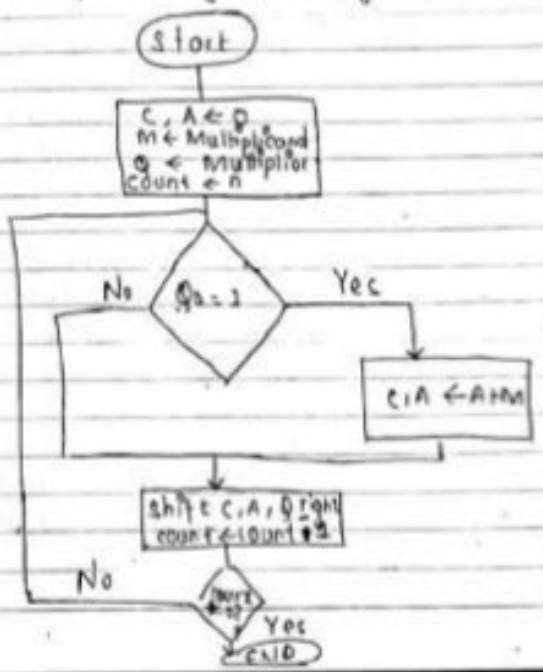
8 → 1000

C	A	Q	M	
0	0000	1000	1111	
0	0000	0100	1111	shift right
0	0000	0010	1111	shift right
0	0000	0001	1111	shift right
0	1111	0001	1111	A ← A+M
0	0111	1000	1111	shift right
0	0011	1100	1111	shift right
0	0001	1110	1111	
0	0000	1111	1111	

Hardware Implementation



Flowchart for Unsigned Binary multiplication



2's complement

$$\begin{array}{r} -5 \\ \times\ -3 \\ \hline 15 \end{array}$$

```
      1011
      1101
     -----
      1011
     0000x
    1011xx
   1011xxx
  ---------
  10000111   => 143
```

Thus for -ve nos or 2's complement no, ordinary multiplication is not possible. Multiplication is performed using Booth's algorithm.

- Multiplier is Q reg, multiplicand in M. Q-1 is one bit reg to right of Q0. Result is stored in A & Q. Initially A & Q-1 is zero. Now, bits are checked in Q-1 and Q0.
- If 11 or 00 occur then bits of A, Q and Q-1 is shifted to 1 bit right
- If 10 or 01 then multiplicand is added or subtracted from A depending upon whether the bits are 01 or 10.

Following addition right shift occurs

- shifting is such that An-1 goes to An-2 and remains in An-1 in both cases.

Booth's Algorithm:

1-1 } shift right
0-0 }

0-1 → A ← A+M } shifting
1-0 ⇒ A ← A-M }

7X3

A	Q0	Q-1	M	
0000	0111	0	0011	
1101	0111	0	0011	A ← A } 1st
1110	1011	1	0011	Shifting right } cycle
1111	0101	1	0011	Shifting right 2nd
1111	1010	1	0011	3rd
0010	1010	1	0011	A ← A+M }
0001	0101	0	0011	Shifting right }

-7 x 3

-7 ⇒ 1001
3 ⇒ 0011

A	Q	Q-1	M	
0000	1001	0	0011	
1101	1001	0	0011	A ← A - M
1110	1100	1	0011	shifting right
0001	1100	1	0011	A ← A + M
0000	1110	0	0011	shifting right
1101	1111	0	0011	shifting right
1110	1111	1	0011	shifting right

A	Q	Q-1	M	
0000	1001	0	0011	
1101	1001	0	0011	A ← A - M
1110	1100	1	0011	shifting right
				A ← A + M

A	Q	Q-1	M	
0000	0011	0	1001	
0111	0011	0	1001	A ← A - M
0011	1001	1	1001	shift right
0001	1100	1	1001	shift right
1010	1100	1	1001	A ← A + M
1101	0110	0	1001	shift right
1110	1011	0	1001	shift right

-> 11101011
2's -> 00010101

8 x -5
8 -> 01000
-5 -> 01011

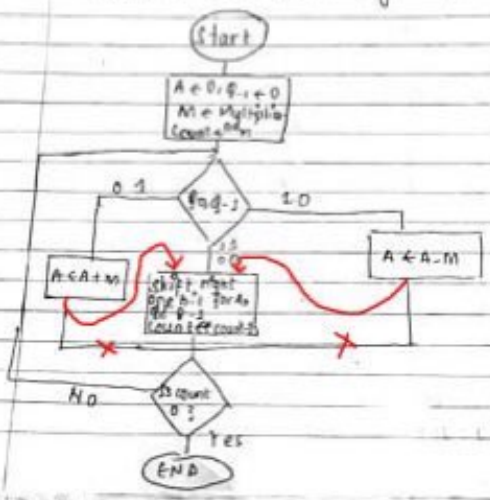
A	Q	Q-1	M	
00000	01000	0	10101	
00000	00100	0	10101	shift right
00000	00010	0	10101	shift right
00000	00001	0	10101	shift right
10101	00001	0	10101	A ← A - M
11010	10000	1	10101	shift right
10101	10000	1	11011	A ← A + M
11010	11000	0	11011	shift right

~~00101 01000~~
101000

A	Q	Q-1	M	
0000	01000	0	11011	
00000	00100	0	11011	Shift right
00000	00010	0	11011	"
00000	00001	0	11011	"
00101	00001	0	11011	A ← A-M
00010	10000	1	11011	shift right
11101	10000	1	11011	A ← A+M
11110	11000	0	11011	shift right

2's → 00000 10000
⇒ 40

7. Flowchart for Booth's Algorithm



4. Division:
For unsigned binary integer: Bits of dividend are examined from left to right until the set of bits examined represents a number greater than or equal to divisor. This refers division divides the dividend.
2. Until this 0's are placed in quotient from left to right. When event occur i.e greater no. one is placed in quotient and divisor is subtracted from partial dividend. The result is partial remained.
Now, division follows in cyclic pattern.
3. At each circle additional bits from the dividend are added to partial remainder until the result is greater than or equals to the divisor. This process continues until all the bits of dividend are examined.

eg: Divisor: 1011 (11)
Dividend: 1001 0011 (147)

```
          0000 1001
     1011 ) 1011  0011
            101   1
            0111  0
          - 1011  1
            00111 2
          -  101  1
            0100  → Remainder
```

2's complement Division

1. Divisor is loaded in M dividend in A and Q register. Dividend must be expressed as twice n bit 2's complement number.
2. Shift A, Q left 1 bit position. If M and A has same sign performed.
   A ← A−M otherwise
   A ← A+M
3. Above expression operation is successful if the sign of A is same before & after operation.
   a) If operation is successful then, set $Q_0 = 1$
   b) If operation is unsuccessful then, $Q_0 = 0$ and restore the previous value of A
4. Repeat above step for the no. of bits in Q, remainder is in A if the sign of divisor & dividend were same then, quotient is in Q, otherwise the correct quotient is 2's complement of Q.

Perform
$-7/3$

0011 ⇒ 3
−7 ⇒ 1111 1001

A

	A	Q	M	
	1111	1001	0011	
1	1111	0010	0011	→ shift left
2	0010	0010	0011	A ← A+M
	1111	0010	0011	restoring A, $Q_0 = 0$
	1110	0100	0011	shift left
	0001	0100	0011	A ← A+M
	1110	0100	0011	restoring A, $Q_0 = 0$
3	1100	1000	0011	shift left
4	1111	1000	0011	A ← A+M
	1111	1001	0011	set $Q_0 = 1$ (2nd)
	1111	0010	0011	shift left
	0010	0010	0011	A ← A+M
	1111	0010	0011	$Q_0 = 0$, Restore A

Q = 2
R = 1111
= 0001

Result is 2's complement of 0010 = 1110

Perform 7/3

7 → 0000 0111

3 → 0011

A	Q	M	
0000	0111	0011	
0000	1110	0011	shift left
1101	1110	0011	A ← A - M
0000	1110	0011	Restore, $Q_0 = 0$
0001	1100	0011	shift left
1110	1100	0011	A ← A - M
0001	1100	0011	Restore A, $Q_0 = 0$
0011	1000	0011	shift left
0000	1000	0011	A ← A - M
0000	1000	0011	Restore A, $Q_0 = 0$
0001	0000	0011	shift left
0010 1101	0000 0010	0011	A ← A + M
0001 0001	0001 0010	0011	A, $Q_0 = 1$

0011 → 3

Remainder → 0001

Remainder = 1

Quotient = 2

0001 / 1101 = 0010

1111 / 0011 / 1101 / 0000

0111 / 1101 / 0011

-8/-5

-8 → 1111 1000

-5 → 1111 1011

0000 1000

0000 0101

1111

1111

+0101

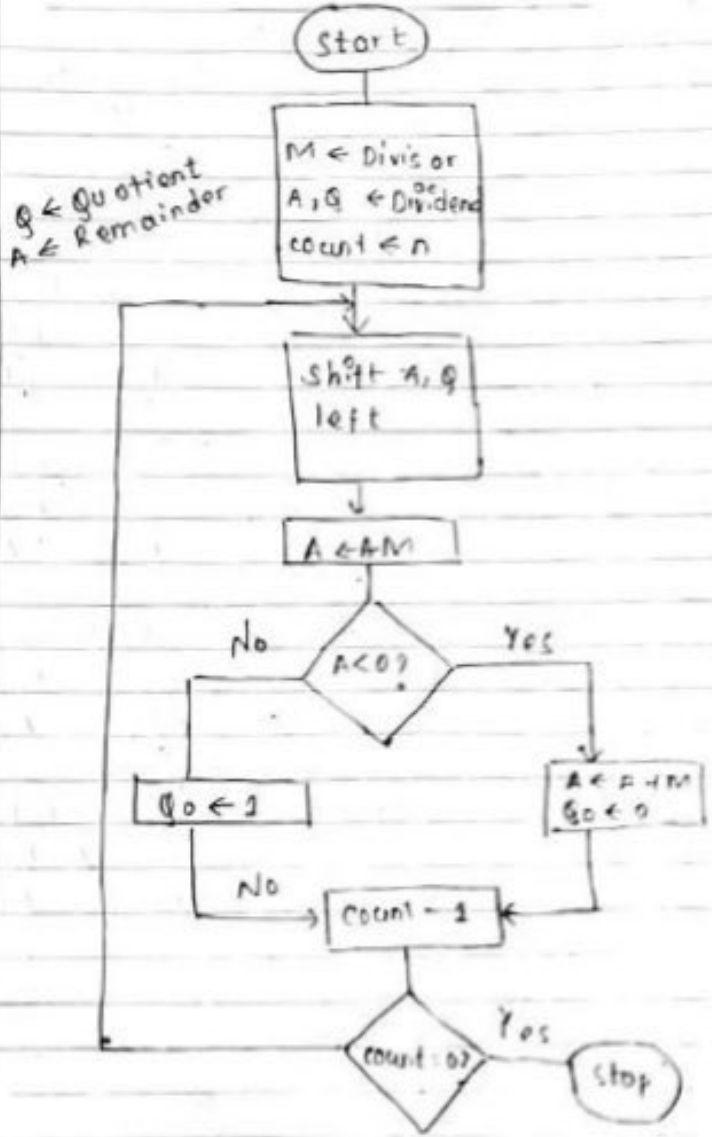
10100

A	Q	M	
1111	1000	1011	
1111	0000	1011	shift left
0100	0000	1011	A ← A - M
1100 11	0000	1011	Restore, $Q_0 = 0$
1110	1110	0000	1011 shift left
0101	0011	0000	1011 A ← A - M
1001 1	1110	0000	1011 Restore A $Q_0 = 0$
1100	1100	0000	1011 shift left
0101	0001	0000	1011 A ← A - M
1000 1	1100	0000	1011 Restore A, $Q_0 = 0$
1000	1000	0000	1011 shift left
0101	1001	0000	1011 A ← A - M
1101	1001	0000	1011 Restore A, $Q_0 = 1$

Rem → 1101 → 0000 1101 → 3

Quotient → 1

Flow Chart



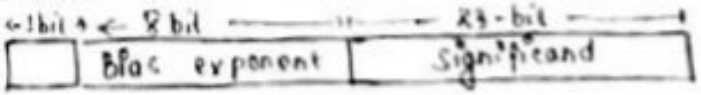
Floating-point Representation

In fixed point notation, it is possible to represent a range of positive and negative integers centered on 0. For very large or fractional number or very small, integer point representation is not suitable. Thus, we use exponent representation for decimal numbers. This allows range of very large and very small no. represented with few digit. This can be applicable for binary nos. also for that we store the value in 3 fields : sign (+ or -), significand, exponent, mantissa

$$\pm S \times B^{\pm E}$$

Sign → (+ or -)
S → Significand
E → exponent

The base B is implicit and need not to be stored because it is same for all numbers. Radix point is at right of left mat or most significant bit i.e there is one-bit to the left of radix point. For example: for 32-bit floating point format



→ left mat-bit is sign-bit which is either +ve or -ve
→ 8-bit gives the Bias exponent
→ 23-bit gives significand

This is bias (exponent) representation. (The exponent is represented in bias rep). A fixed value called bias is subtracted from the field to get true exponent. The bias equals $2^{k-1} - 1$ where k is the no. of exponent bits. For example:
for 8 bit $2^8 - 1$
$\Rightarrow$ 127
i.e. 127 is subtracted from bias exponent to get true exponent value. Here, base is assumed to be 2. The rest 23-bit is significant also known as ~~mate~~ 'mantisa'
To perform operation of floating point no, the no. must be normalized. A normalized no. is one in which most significant digit of significand is non-zero.
for base 2 representation: A normalized no is one in which MSB is '1'.
eg:
$1.bbbbb \pm 2^{\pm e}$
where b is either 0 or 1
and e $\rightarrow$ bias exponent.
Since MSB is always 1, 23-bit representation can be used to represent 24-bit no.

eg: $1.1010001 * 2^{20}$ $\left(\begin{smallmatrix}127\\+20\end{smallmatrix}\right)$
can be represented as

0 1001 0011 10101000100000000000000

Five regions on the no. line are not included in this range.
1. Negative no. less than $(-2 \rightarrow 2^{-28}) * 2^{128}$ called negative over-flow
2. Negative no. greater than 2^{-127} called negative under-flow.
3. Zero
4. Positive no. less than 2^{-127} called positive under-flow.
5. Positive no. greater than $(2 \rightarrow 2^{-23}) * 2^{128}$ called positive over-flow.

Floating point Arithemathe

For addition & subtraction, it is necessary to ensure that both operands have same exponent value. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication & division are straight forward. A floating point operation may produce one of these conditions:
1) Exponent overflow:
A positive exponent value exceeds the max. possible exponent value.
2) Exponent underflow:
A negative exponent is less than the min. possible value i.e the no. is too small to be represented.
3) Significand overflow:
Addition of 2 significands of same sign may result in a carry of MSB.

4) Significand underflow:
digit may flow off right-end of significand.

Addition & subtraction

① Check for zeroes
② Alignment of significand
③ Addition or subtraction of significand
④ Normalize the result.

1. Check for zeros:
Addition & subtraction are identical except for sign change. Thus, process begins by changing the sign of subtrahend if it is subtract operation. If either operand is zero, the other is reported as result.

2. Significand alignment:
Here, no should be manipulated so that exponents are equal. Alignment is achieved by shifting either the smaller no. to right or larger no. to left. Shifting is such that the magnitude portion of significand is shifted right and one is increased in exponent until both exponents are same.

3. Addition
The two significands are added together considering # to their signs. The signs may differ & result is zero. significand may also overflow by 1 digit if so, significand is shifted right and exponent is increased by 1. When exponent overflow occurs, operation is haulted.

4. Normalization
Normalization consist of shifting significant digit to left until MSB is non-zero

eg:
$X = X_s * B^{\pm X_e}$
$Y = Y_s * B^{\pm Y_e}$

$X+Y = (X_s * B^{X_e - Y_e} + Y_s) * B^{Y_e}$ } $X_e \leq Y_e$
$X-Y = (X_s * B^{X_e - Y_e} - Y_s) * B^{Y_e}$ }

Eg: $X = 0.3 * 10^2$
$Y = 0.23 * 10^3$

Multiplication & Division

Multiplication:
1. If operand is zero, result is reported as 0.
2. add in exponent if exponent are stored in bias form, the exponent sum would have doubled the bias. Thus, bias value is subtracted from sum. The result is either exponent overflow or underflow. If exponents are in proper range, the next step is multiplying the significants considering the signs. The case is applicable for both sign and 2's complement notation. The product will be double the length of multiplier & multiplicant. extra bits are lost

during rounding. The result is then normalized.

Division:

- if divisor is 0, an error is issued or the result is set to infinity. if dividend is zero, result is zero. Next,
- Divisor exponent is subtracted from dividend exponent & the significant is divided. exponent overflow or ~~or~~ underflow is checked.

$X = X_s + B^{X_E}$

$Y = Y_s * B^{Y_E}$

$\Rightarrow X * Y = (X_s * Y_s) * B^{X_E + Y_E}$

$X/Y = (X_s / Y_s) * B^{X_E - Y_E}$

eg: $X = 0.3 * 10^2$

$Y = 0.2 * 10^3$

$X * y = (0.3 * 0.2) * 10^{2+3}$

$= (0.06) * 10^5$

$X/Y = \left(\frac{0.3}{0.2}\right) * 10^{2-3}$

$= 1.5 * 10^{-2}$

$= 0.15$

BCD Adder:

The most common format used to represent decimal data is called BCD. In BCD notation, every 4-bit represent 1 decimal digit. (0000(0) to 1001(9))
for values from 0 to 9 representation are same as in binary. Bits value above 9 are not used in BCD i.e 1010 to 1111.
Multiple decimal numbers are stored as multiple groups of 4-bits per digit. BCD is a signed notation i.e it can be -ve or +ve. Here, no. are stored as absolute value and 1-bit is require to store sign bit.

Eg:

+27	0	0010	0111
-27	1	0010	0111

BCD Adders
BCD Adder is similar to signed magnitude notation as long as the sum of 2-digits do not exceed 9. There are 2 situation in which error occurs:

(i) When the result is not a valid BCD digit.

Eg:

	0100
6 +	0110
10	1010 → invalid

(ii) when the result is valid BCD-digit

⇒ 9 : 1001
+ 8 : 1000
: 10001 : invalid digit

In either case adding 6 to the result generated by a binary adder produces the correct result.

Note : 6 is binary pattern that is unused in BCD representation.

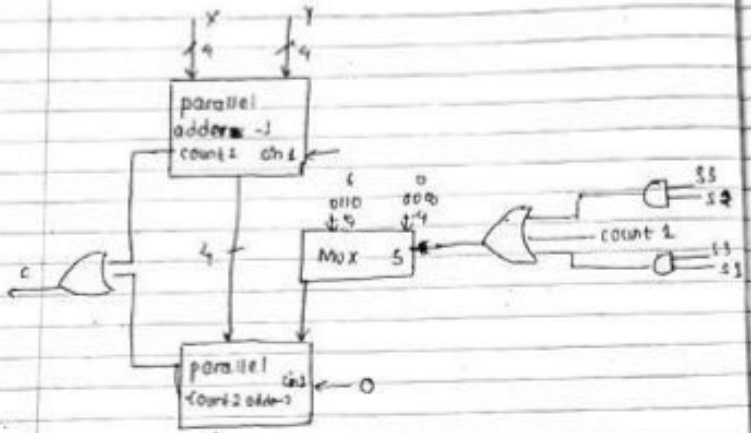


Fig : BCD Adder

In figure, 2 digits X and Y are first added together then, this result is either added to 0 or to 6 to produce the correct BCD sum. If the result is not a valid BCD digit then, either S_3 & S_2 or S_3 & S_1 will produce 1. In this case, MUX control bit is set to 1 which causes 6 to be added the result. The MUX select bit is also set to 1 if the add of X & Y generates a carry out as in the case of X = 8 and Y = 9. This causes to add 6 to the original sum again producing the correct value. If either parallel adder generates 1 the carry out of BCD adder should also be 1. Multiple copies of this hardware can be cascaded by connecting their carry out / carry in signals to form a multi-digit BCD adder.

Chapter - 5

Control unit Design

Micro-operations :
Micro-bytes name gives that operation are small and simple. Micro-operations are smaller series of steps. each of which involves the processor register. Instruction cycle is made up of number of smaller units. One sub division is fetch, indirect, execute and interrupt. Execution of a program consists of sequential execution of instructions. Instruction is executed during an instruction cycle made up of shorter sub-cycles. The performance of each sub-cycle involves micro-operation.

* Sub-cycle of Instruction cycle

1) Fetch cycle → It fetches the instruction from memory.
Register involved are:
Memory Address Register (MAR)
↳ connected with memory address line for read/write operation

Memory Buffer Register (MBR)
↳ connected with data lines to store value in memory or store data from memory.

Program counter (PC)
↳ holds the address of next instruction to be fetched

Instruction Register (IR)
↳ holds the last instruction fetched

t1: MAR ← (PC)
t2: MBR ← memory
t3: PC ← PC + I
t4: IR ← (MBR)
→ 4 micro-operations are performed in fetch cycle.

The grouping of micro-operation must follow 2 simple rules:
1. The proper sequence of events must be followed.
2. Conflicts must be avoided. Read & write should not be done from same register in one time unit.

2) Indirect cycle:-
once a instruction is fetched, the next step is to fetch source operand. let us assume one address instruction format with direct and indirect addressing. If instruction specifies indirect address then, indirect cycle must precede execute cycle. This includes following micro-operation:
t1: (IR address)
t2: MBR ← Memory
t3: IR(address) ← (MBR(address))

3. Interrupt cycle:-

After execution cycle, test is performed for existence of any interrupt. If there is interrupt, interrupt cycle occurs. This include following micro-operations:-

t1: MBR ← (PC)
t2: MAR ← save_address
PC ← routine_address
t3: Memory ← (MBR)

In 1st step, content of PC are transferred to MBR so that they can be saved for return from interrupt. MAR is loaded with address at which PC contents are to be saved and PC with address of interrupt. In 3rd step, MBR content is store in memory, processor then begins with next instruction cycle.

4. Execute cycle

eg:- For execute cycle, it is difficult to predict the micro-operations. For a machine with 'N' different opcode, there are 'N' different sequences of micro-operation.

Eg: Add R1, X
t1: MAR ← (IR address)
t2: MBR ← memory
t3: R1 ← (R1) + (MBR)

eg:
ISZ X ⇒ Instruction is incremented and skip if zero
t1: MAR ← (IR address)
t2: MBR ← memory
t3: MBR ← (MBR) + 1
t4: Memory ← (MBR)
If ((MBR) = 0) then (PC ← (PC) + 1)

Instruction cycle
Instruction cycle code given 2-bit
P Q
0 0 : Fetch
0 1 : Indirect
1 0 : Execute
1 1 : Interrupt

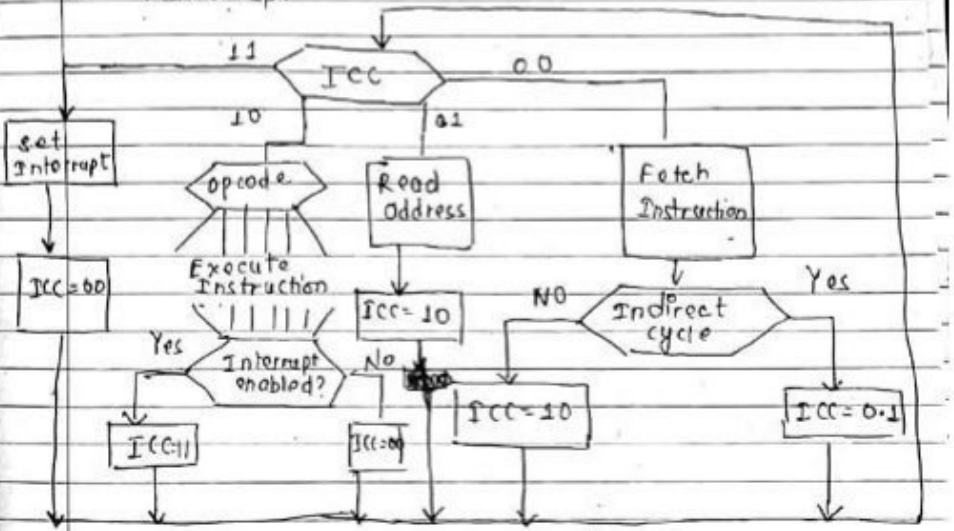


Fig: Flow Chart for Instruction cycle

Fetch Indirect and Interrupt cycle has one sequence of micro-operation but for execute cycle there is another sequence of micro-operation depending upon opcode. All these micro-operation are bring together to complete an instruction cycle. For this, we have one 2-bit register called instruction cycle code (Icc). At the end of 4-cycles, the Icc is set appropriately. Indirect cycle is always followed by execute cycle and interrupt cycle is followed by fetch cycle. For both execute & fetch cycle, the next cycle depends on state of system.

Control of processor

Functional Requirements:
(ALU, Register interconnection, Internal data path, control unit)

1. Define the basic elements of processor.
2. Describe the micro-operations that processor does.
3. Determine the micro-operations that control unit must perform to cause the micro-operations to be done. → sequencing & execution

2.1. data transfer in b/w reg.
2. data transfer to & from memory reg.
3. ALU operation

control signals are input & output to control unit.

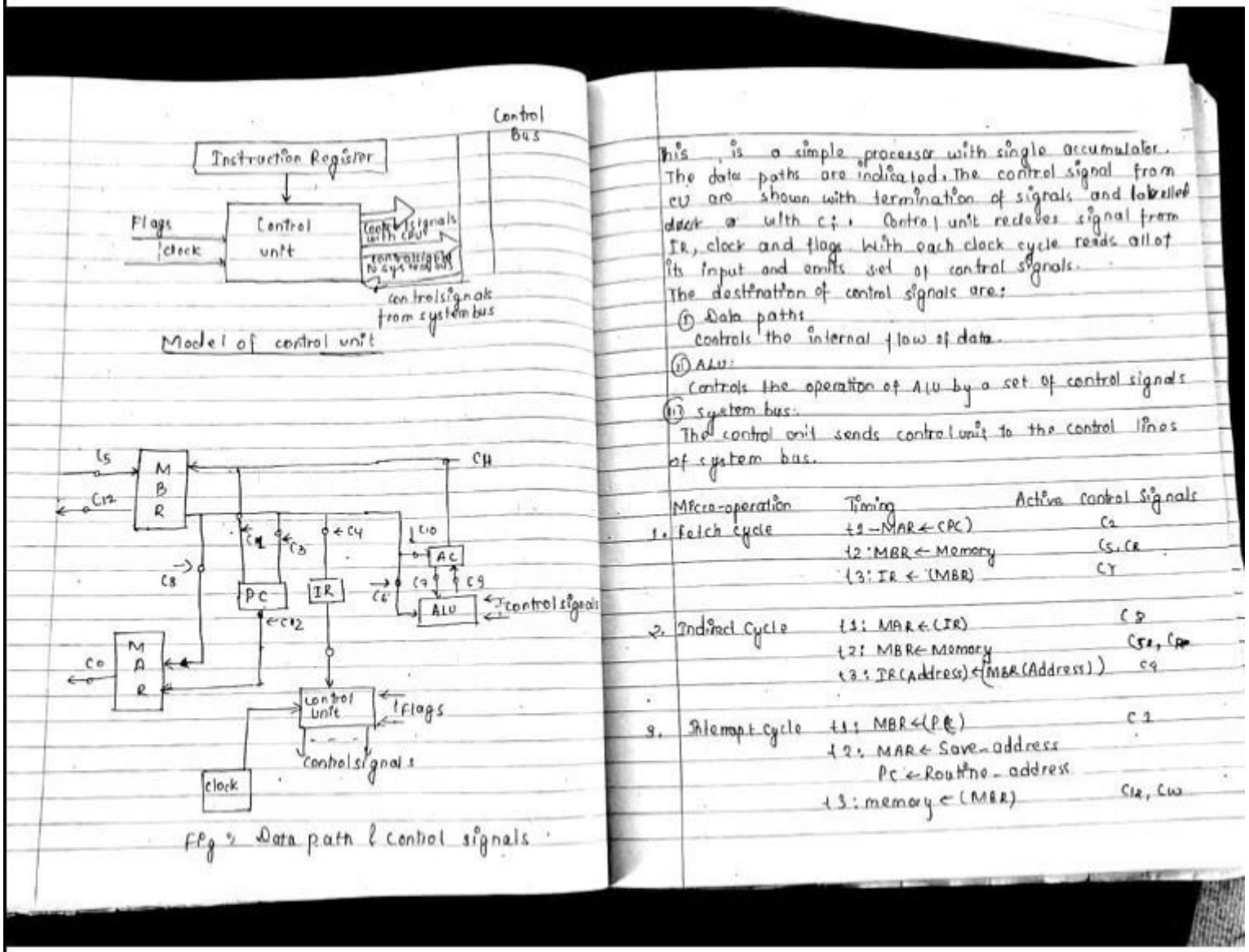
Inputs

1. clock:
It is the time of control unit. All micro-operation should be executed in a signal clock pulse.
2. IR:
opcode of current instruction determines the operation to be performed during execution cycle.
3. Flags:
gives the status of processor and outcome of previous ALU.
4. Control signals from control bus:
The control bus portion of system bus provides signals to control unit such as interrupt & acknowledgement.

outputs:

1. Control signals within processor:
1. Signal that move data from one register to another.
2. control signals that activate ALU fxn.

2. Control signals to control bus:
1. Signals to memory
2. Signals to IO modules



This is a simple processor with single accumulator. The data paths are indicated. The control signal from CU are shown with termination of signals and labelled as with C1. Control unit receives signal from IR, clock and flags. With each clock cycle reads all of its input and emits set of control signals.
The destination of control signals are:

(i) Data paths
Controls the internal flow of data.

(ii) ALU:
Controls the operation of ALU by a set of control signals

(iii) System bus:
The control unit sends control info to the control lines of system bus.

Micro-operation	Timing	Active control signals
1. Fetch cycle	t1: MAR ← (PC)	C2
	t2: MBR ← Memory	C5, C8
	t3: IR ← (MBR)	C7
2. Indirect Cycle	t1: MAR ← (IR)	C8
	t2: MBR ← Memory	C5, C9
	t3: IR(Address) ← (MBR(Address))	C9
3. Interrupt cycle	t1: MBR ← (PC)	C1
	t2: MAR ← Save-address PC ← Routine-address	
	t3: memory ← (MBR)	C12, C10

CR← Read signal to system bus
CW← write signal to system bus

4. Execute
Add
t1 : MAR ← (IR) — C8
t2 : MBR ← memory — C5, CR
t3 : R1 ← (R1) + (MBR) — C6, C7

* Hardwired Control Unit:
In hardwired implementation, control unit is implementation comb^n of ckt. Its input signals are transferred into set of output signals which are control signals.

* Control Unit Inputs:
The inputs of control signals are IR, clock, flags & control bus signals. Individual bits of flags & control bus signals have some meaning. IR and clock are not directly used for control unit. Control units makes use of opcode & perform different action for different instruction. There is a unique logic input for each opcode. Each fxns can be performed by a decoder which takes an encoded input & produce a single output. A decoder has n-binary input and 2^n-binary output.
Example:
4 → input
2^4 → 16 bit

I_3	I_2	I_1	I_0	O_{15} O_{14} O_{13} O_{12} O_{11} O_{10} O_9 O_8 O_7 O_6 O_5 O_4 O_3 O_2 O_1 O_0
0	0	0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0	0	0	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0	0	1	0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0	0	1	1	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0	1	0	0	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0	1	0	1	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0	1	1	0	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0	1	1	1	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
1	0	0	0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
1	0	0	1	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
1	0	1	0	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1	0	1	1	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1	1	0	0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
1	1	0	1	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1	1	1	0	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1	1	1	1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

The clock portion of control unit issues a repetative sequence of pulses, this is useful for measuring the duration of micro-operation. The control unit emits different control signals as different time units within a signal single instruction cycle. A counter is needed as input for different control signal which can be initialized after the end of the instruction cycle.

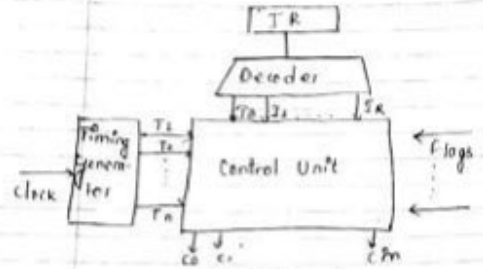


Fig: Control Unit with Decoded input

$C_5 \Rightarrow$ T2 : Fetch cycle + T2 : Indirect + T2 : Execute
$C_5 \Rightarrow P'Q'T_2 + P'QT_2$

Let us discuss internal logic of control unit that produces output control signals as a fxn of input signals. A boolean expression is derived for each inputs. Let us consider a single control signal C_5 from micro-operation of instruction cycle. C_5 causes data to be read from memory into MBR. Consider new control signals P and Q with following interpretations:

~~$C_5 \Rightarrow$ T2 : Fetch cycle + T2 : Indirect + T2 : Execute~~
~~$C_5 \Rightarrow P'Q'T_2 + P'QT_2 + PQ'T_2$ (Add)~~

P	Q	
0	0	Fetch
0	1	Indirect
1	0	Execute
1	1	Interrupt

then for C_5 :
$C_5 \Rightarrow P'Q'T_2 + P'QT_2 + PQ'T_2$ (add)

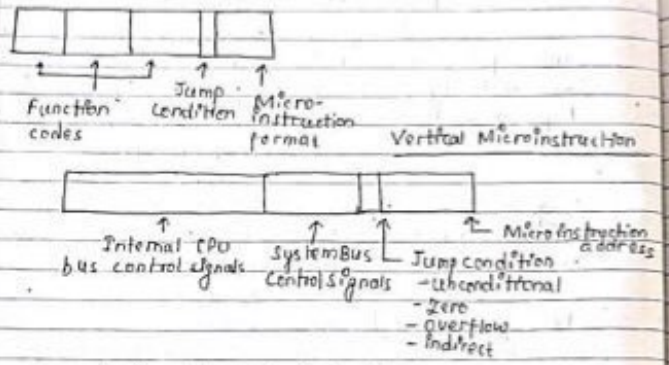
This process is repeated for every control signal generated by processor. The boolean eq^n defines the logic of control unit. These equations needs a no. of comb^n ckt. Thus, micro-programming approach is ~~unit~~ used.

Assignments:
2019 2b (Fall) 2014 (Fall) 2b 2016 (Spring) 1a
2013 Fall (8a) 2015 (Fall) 2b, 3b

2. Microprogrammed Control Unit

Micro-instruction: It is set of micro-operation occuring at one time. A sequence of micro-instruction is known as micro-program or firmware. For any micro-operations, control line from ~~comp~~ control unit is either on or off. This can be represented by binary digit for each control-line. Thus, each micro-operation is represented by bit-patterns of one and zero in control-word.
All the control-words are sequence of micro-operations and we bring it together. Since, micro-operations are

not fixed we put the control-words in memory. with unique address. An address-field is added in An control-word for next control-word to be executed for certain true condition, the true condition is also specified by few bits. This format of micro-instruction is known as horizontal micro-instruction.



Horizontal Microinstruction

horizontal:
The format of micro-instruction or control-word is a 1-bit for each internal processor control-line and 1-bit for each system-bus control lines, a conditional-field that indicates condition for branching, a field with address of micro-instruction to be executed when branch is taken.

Such micro-instructions can be interpreted as:
1. To execute micro-instruction turn on all the control lines indicated by 1, off all the control lines indicated by 0.
2. If condition indicated by condition-bit is false the next micro-instruction is executed i.e in sequence.
3. If condition-bit is true, the next micro-instruction is executed indicated in address-field.

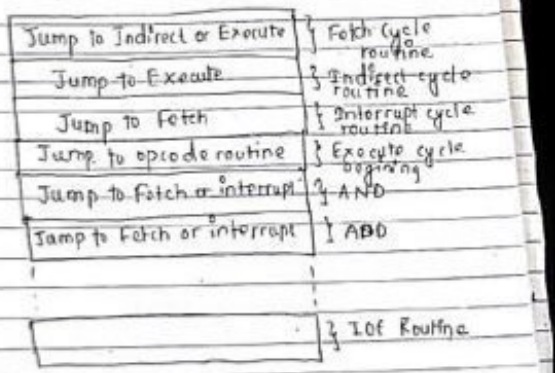


Fig: Organization of Control Memory

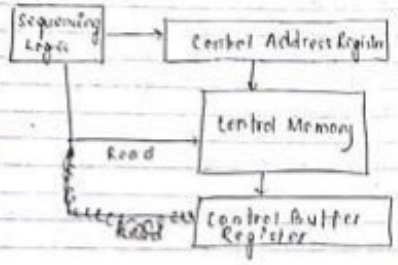


Fig: Control-unit Micro-Architecture

Here, the set of micro-instruction is stored in computer memory. The control-address register contains the address of next-instruction to be read. When a micro-instruction is read from memory, it is transferred to control buffer reg. The sequencing unit loads the control address reg. and issues the read command.

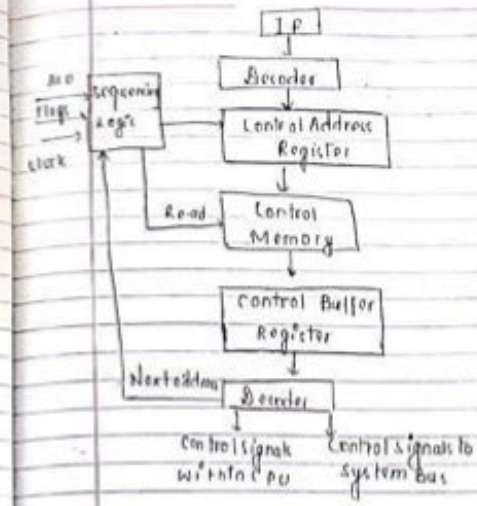


Fig: Functioning of Micro-programmed Control Unit

1. To execute an instruction, the sequencing logic unit issues a read command to control memory.
2. The word whose address is specified in control address register is read in control-buffer register.
3. The content of control-buffer register generates signals and next address-information for sequencing logic.
4. The sequencing logic unit loads new address in to control address-register based on CBR and ALU flags.

* All these happens in 1 clock pulse. Depending upon CAR & ALU flags one of three decision is made:
1. Get the next instruction
2. Jump to new routine based on a jump micro-instruction
3. Jumps to machine-instruction routine.

Micro-Instruction sequencing:
Micro-programmed control unit perform two tasks:
1. Micro-Instruction Sequencing → Get next micro-instruction from control memory
2. Micro-Instruction Execution → Generate control signals needed to execute micro-instruction

imp
3 types:
1. Two-address format
2. Single-address
3. Variable-address format

Sequencing Techniques
control memory generates next micro-instruction. The next address is generated on the basis of format of address in micro instruction.
① Two address format
② Single address format
③ Variable format

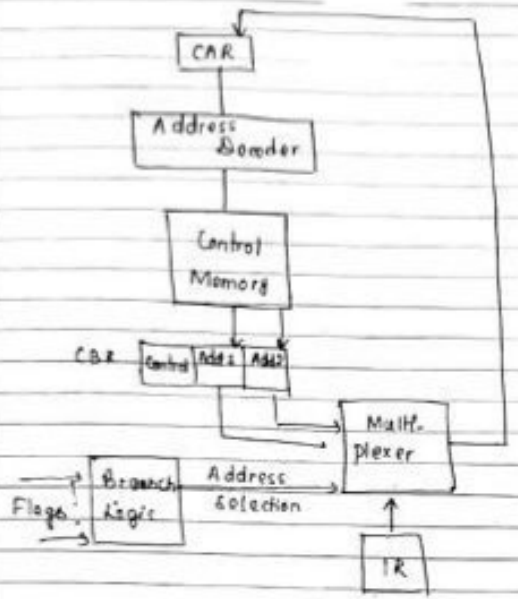


Fig: Two address field

This is the simplest approach, there is 2 address field in each micro-instruction. A multi-plexer is provided that serves as a destination for address-field and instruction reg. Based on address selection, input, MUX transmits either Opcode or one of 2 address to control address register. CAR is decoded

to produce next micro-instruction address. Address selection signals are provided by branch logic module whose inputs are from flags and control buffer register.

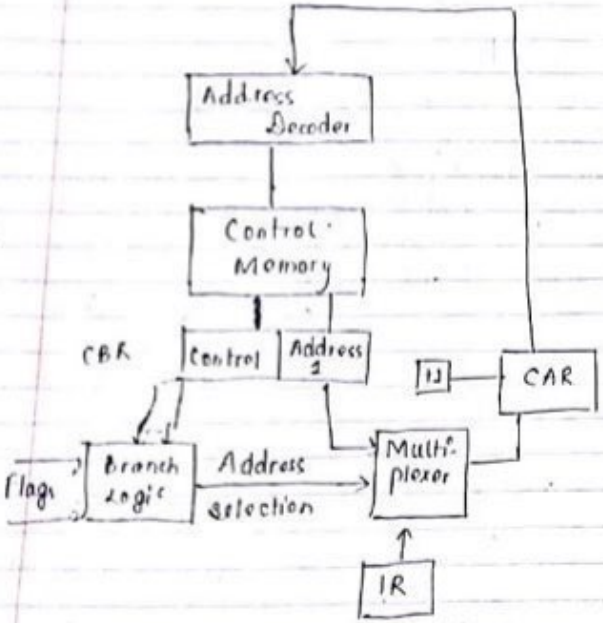


Fig: single address field

Next sequential address is generated from either address field, instruction register code or next sequential address from CAR.

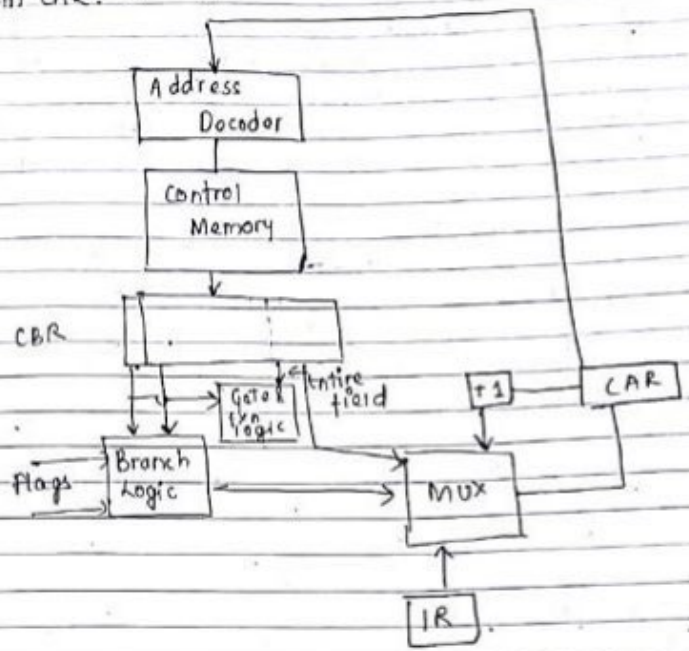


Fig: variable address format

Another approach is to provide to different micro-instruction format. Here, 1-bit decides which format is being used. In one format remaining bits are used to activate control signals. In other format,

Some bits provides the branch logic & remaining bits provides the address. In 1st format, next address is either next sequential address or an address by IR. In 2nd format, either a conditional or unconditional branch is specified. In this format one entire cycle is consumed with each branch micro-instruction. In 1st format address generation is part of same cycle.

Address Generation

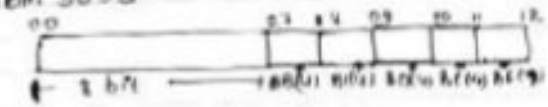
1. Explicit → Next sequential micro-instruction is present in microinstruction.
2. Implicit → Additional Logic is required to generate next address.
   - (i) Mapping
   - (ii) Addition
   - (iii) Residual

2. Implict:

Mapping:

Mapping is required with all designs. The opcode portion of a machine instruction must be mapped into micro-instruction address. This occurs once per instruction cycle. Another technique is combining or adding two portion of an address to form complete address.

eg: IBM 3033



13-bit CAR

Here, control address register is 13-bit long, the higher 8 bits normally do not change and are copied from directly from 8-bit field of an micro-instruction and remaining 5-bits are set to specify the specific address of next-microinstruction to be fetched.
The final technique is residual control, this involves the use of micro-instruction address that has been previously saved in temporary storage within control unit.
Ex: LSI-11
Next instruction is generated by one of these 5 ways:

(i) Next sequential address
In absence of other instruction, CAR is incremented by 1.

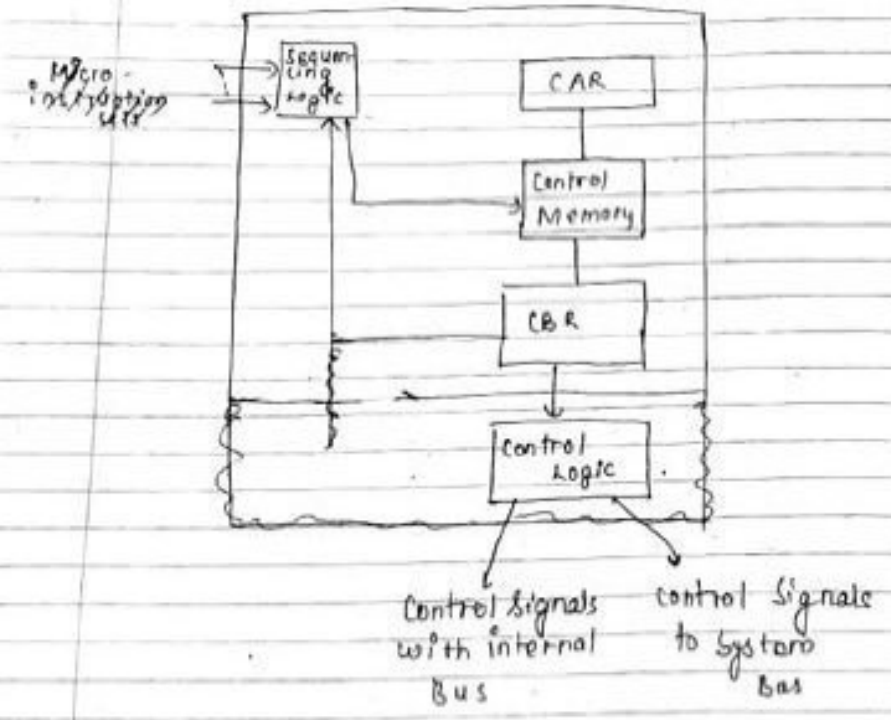
(ii) Opcode mapping
At the begining of each instruction cycle, the next micro-instruction is determined by opcode.

(iii) Interrupt testing
If interrupt occurs, this determines next micro-instruction address.

(iv) Branch
Conditional & Non-conditional Branch Instruction are used.

(v) Sub-routine facility
One-level subroutine facility is provided. In this,

One-bit in every micro-instruction is dedicated to do this task. When the bit is set, 11-bit return register is loaded with the updated contents of CAR. A sub-sequent micro-instruction that specifies or return will cause the CAR to be loaded from return-register.



Classification of Microinstruction

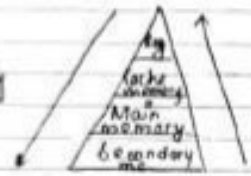
1. Horizontal / Vertical
2. Packed / Unpacked
3. Hard / Soft
4. Direct / Indirect

Application of microprogramming

1. Emulation
2. Operating system support
3. Realization of computer
4. Microdiagnostic
5. User tailoring

Chapter - 6

Memory Organization



Memory unit is an essential component of computer as it is used for storing programs and data. Memory unit that communicates directly with CPU is main memory.
Memory device that provides memory backup is known as auxillary.

Main memory :

Main memory is the central storage unit based on semiconductor ICs. RAM and ROM are semiconductor ICs. RAM is available in two types Static and Dynamic.
Static RAM consists of flipflops and store binary information. Dynamic RAM stores information in the form of electric charge applied for capacitors. DRAM offers reduced power consumption and larger storage capacity in a single memory chip.

ROM :

ROM is used for storing programs that permanently resides in computer. ROM is used for storing an initial program called boot-strap loader. Boot-strap loader is a program whose fxn is to start computer software operating when power is turn on.

+ Types of Rom

RAM and ROM chips :-

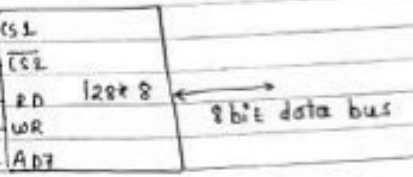


Fig : Block diagram of RAM chip

CS1	TS2	RD	WR	Memory fxn	State of data b
0	0	X	X	inhibit	High impedan
0	1	X	X	inhibit	High impedan
1	0	X	X	inhibit	High impedan
1	0	0	1	write	Input data to
1	0	1	X	read	output data fro
1	1	X	X	inhibit	High Impedan

Fig : Functiontable

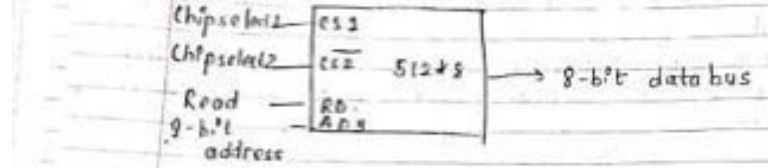


Fig: Block diagram of ROM chip

Auxillary Memory

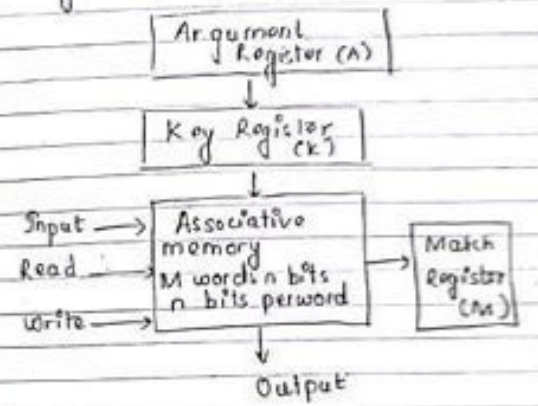
Devices that provides backup storage.
Features
Access time
storage

Types
Magnetic Disk
Magnetic tapes.

Associative Memory

Content Address Memory (CAM)

Associative memory unit access the content of memory by data rather than address. It is also known as CAM. This type of memory is access simultaneously and in parallel by the data content when a word is retain in an associative memory, no address is given. The memory finds an empty unused location to store the word. To read from associative memory, the content or part of word is specified. The memory locates the word which matches the content & they are read. An associative memory is expensive than RAM as each cell must have storage capacity as well as logic circuits for matching its content.



Associative memory consists of a memory array & logic for m-words with n-bits per word. The argument register (A) and Key Register (K), each contains. The match register M has m-bits, one for each memory words. Each word is memory is compared

in parallel with the content of argument register. The words that match bits of argument register set a corresponding bits in ~~argument~~ match reg. after matching, reading is accomplished by a sequential access to memory for those words whose corresponding bits in match register have been set.

eg:

A 101 110011

K 111 000000

CAM			
word 1	110	00000	Match not found
word 2	100	11001	Match not found
word 3	101	11001	Match found.

The relⁿ betⁿ memory array & external register in an associative memory is shown in figure. The cells are marked by 'c' which two subscripts. First give the word no and second specify the bit position in the word. Thus, cell 'cij' is the cell for bit j in word i. Aj in the argument reg is compared with all bits in column j of array provided that Kj = 1. This is done for all column, if a match occurs betⁿ arguments and bits in word i, the corresponding Mi is set to 1. If one or more unmasked bits of argument and the word do not match Mi is cleared to 0.

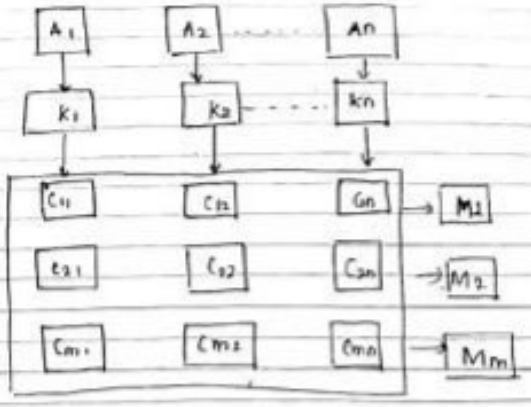


Fig: Associative Memory of m-words n bit per word.

Match Logic:

Match Logic for each word can be derived from the algorithm for 2 binary nos. Let us ignore key bit and compare argument in A with bits in cells of word. Word i is equal to argument in A if Aj = Fij for j = 1 to n. Two bits are equal if both are 1 or 0. This can be interpreted as $x_{ij} = A_j F_{ij} + A_j' F_{ij}'$

where $x_{ij} = 1$ if bits in position are equal otherwise 0.

For i^{th} word to be equal to argument A, we must have all x_j variable equal to 1. Thus, M_i can be set 1.
Boolean fxn for this condition is $M_i = x_1 x_2 x_3 \ldots x_n$ and AND operation.

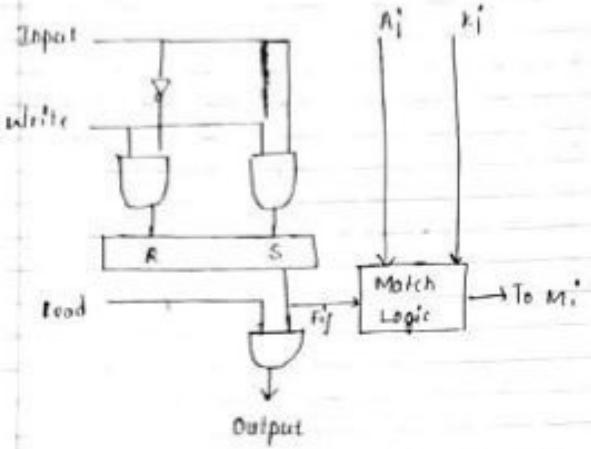


Fig: One cell of associative Memory

Now, let us include key bit K_j in comparision logic. The requirement is that if $K_j = 0$, the corresponding bits should not be compared only when $k_j = 1$, they must be compared. Thus,

$x_j + k_j' : x_j$ when $k_j = 1$
1 when $k_j = 0$

$x_j + k_j' \Rightarrow x_j$
when $k_j = 1$
$x_j + k_j' \Rightarrow x_j + 0$
when $k_j = 0 \Rightarrow x_j$
$x_j + k_j' \Rightarrow x_j + 1$
$\Rightarrow 1$

A term $x_j + k_j'$ will be in one state if the pair of bit is not compared. This is necessary so that ~~and~~ AND operation has no effect for output 1. Now, it can be expressed as :-

$M_i = (x_1 + K_1')(x_2 + K_2') \ldots (x_j + x_j')$
$M_i = (A_1 F_{i1} + A_1' F_{i1}' + k_1')(A_2 F_{i2} + A_2' F_{i2}' + k_2') + \ldots$
$(A_n F_{in} + A_n' F_{in}' + K_j')$
$M_i = \prod_{j=1}^{n} (A_j F_{ij}' + A_j' F_{ij}' + k_j')$

$\prod \rightarrow$ used for AND operation.

Read Operation.

If more than one word in memory matches the argument field, all the match word will have 1 in corresponding bits position of match reg. The bits of match reg is scanned one at a time. The match words are read in sequence by applying read signal to each word whose bit is 1. In most cases, associative memory stores table with no identical items. Thus, no read signal is needed. output is connected directly to M_i.

Write operation:

An associative memory must have a right capability for storing information to be searched. If entire memory is loaded with new information at once prior to search operation then, writing can be done in sequence. This will make RAM for writing & CAM for reading.

For unwanted words to be deleted and new words to be inserted, a special register is required to distinguish betn active and inactive words. This register also known as tag register would have as many bits as their as words in memory. All the active words are set to bit 1. Word is clear in memory by clearing the tag bit to 0. Word is deleted by scanning the tag register until first 0 is encounter. This is the position to delete & insert new word.

Cache Memory

It is the fast small memory where active portion of program & data are store, so that average access time is reduced. Thus, it reduces the total execution time of program. Frequently used data are stored in cache memory. The cache memory access time is upto 5 to 10 times faster than main memory. Average access time of memory can be approach to access time of cache by placing frequently used instruction into cache. The basic operation of cache is as follows:-

- When CPU needs to access memory the cache is examined. If the word is found in cache it is read from there. If the word is not found in cache, main memory is access to read the word. A block of word is then transferred to cache memory from main memory.
- Performance of cache memory is measured in terms of a quantity called hit ratio. When the CPU refers to memory & finds it in cache, it is said to produce a hit. If the word is not found in cache, it is said to miss. The ratio of no. of hits divided by total CPU references to memory is the hit ratio. If the hit ratio is high so that the most of CPU accesses is in cache then, main memory average access time is closer to access time of cache.

Transformation of main memory data into cache memory is mapping.

i) Direct Mapping
ii) Associative Mapping
iii) Set-Associative Mapping

Direct Mapping :

Eg :

Let us consider cache can hold 64kB.

→ Data is transferred b/w main memory and cache in blocks of 4 bytes each i.e cache is organized in 16K = 2^14 lines of 4 bytes each.

→ Main memory consists of 16M bytes with each byte directly addressable by 24-bit address (2^{24}=16M). That, for mapping purpose, we can consider main memory to consist 4M blocks of 4 bytes each.

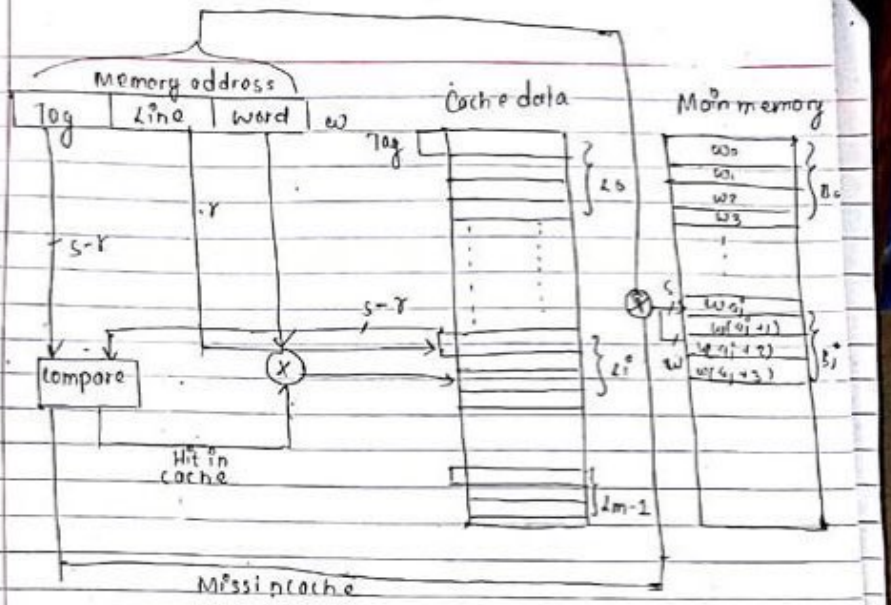
1. Direct Mapping:

It is the simplest technique, it maps each block of main memory into only one possible cacheline. Mapping is expressed as :

$i = j$ modulo m

where i => cacheline no.
j => main memory block no.
m => no. of lines in cache.

Mapping function is implemented using address. Main memory address can be viewed consisting of three fields. The least significant w bits identify word within block of main memory. The remaining s bits specify one of 2^s blocks of main memory. The cache logic interprets these s bits as a tag of (s-r) bits and a line field of r bits. This fields identifies one of $m = 2^r$ lines of cache.



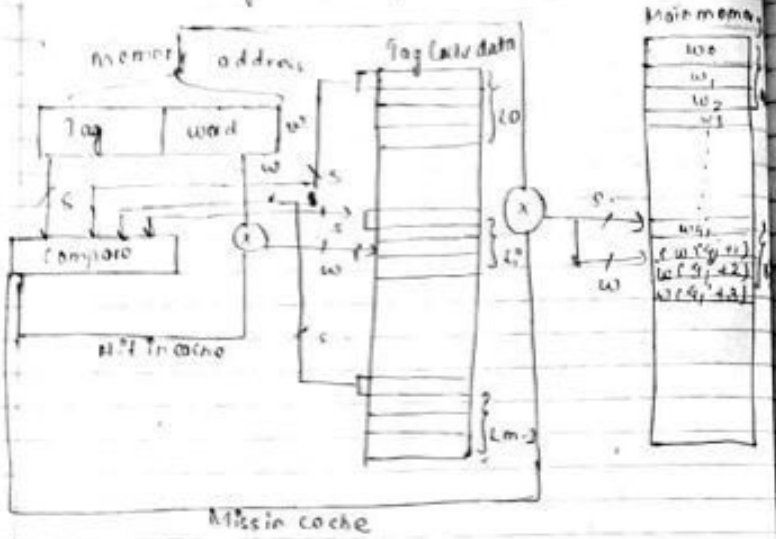
Disadvantages:

There is a fix cache location for any given block. Thus, if a program refrence from 2 different blocks that map into same line, then the block will swap in cache continuously and hit will be low.

2. Associative Mapping

Associative Mapping removes the drawback of direct Mapping by providing main memory block to be loaded into any line of cache. Memory address logic intreprets a tag and a word field. To whether

the block is in cache, the cache control logic examines every lines tag from a match.



Advantage

Fast due to associative memory. Flexible to replace block when new block is read into cache.

Disadvantage

Need of complex circuit to examine the tags of all cache lines in parallel

3. Set Associative

It exhibits the strength of both direct & associative mapping. Cache is divided into v-set each of which consists of k-lines. The relations are

$m = v * k$

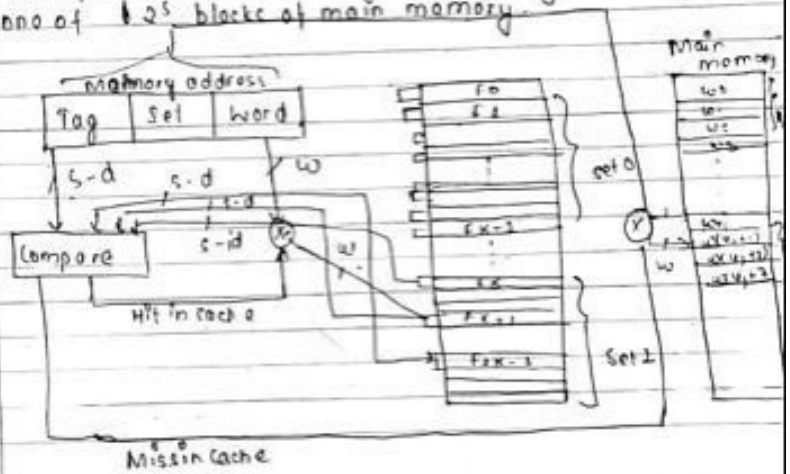
$i = j \bmod v$ where

i = cache set no

j = main memory block

m = no. of lines in cache.

This is known as k-way set associative mapping. With set associative, a block B_j can be mapped into any line of set i. Here, cache control logic interprets memory address in three fields tag, set, word. The d set bits specify one of $v = 2^d$ sets. s-bits of tag specify one of 2^s blocks of main memory.



chapter - 7

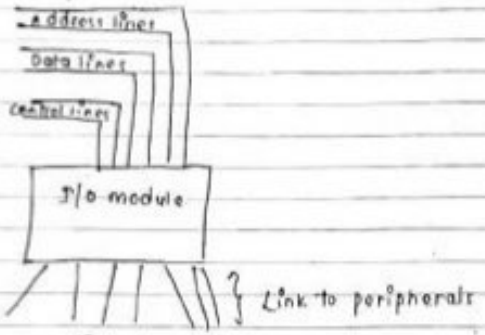
I/O Organization

I/O Organizations

I/O module :

* I/O module is the key element for computer system. I/O module controls the peripheral device. I/O module has two measure functions:

1. Interface to processor & memory through system bus or central switch
2. Interface to one or more peripheral devices by tailored data links.



Block diagram

* External device:

provides a means of exchange of data betⁿ external environment & the computer. An external device is attached to computer by a link to an I/O module. The link is used to exchange control, status, data betⁿ the I/O module and external device. External device is referred to as peripheral device or peripherals. It is classified in three categories:

1. Human readable : Suitable for communicating with computer user
   Eg : VDU
2. Machine readable : Suitable for communicating with equipments
   Eg : Magnetic disk
3. Communication : Suitable for communicating with remote devices
   Eg : Routers, VDU, Human Readable machine or another computers.

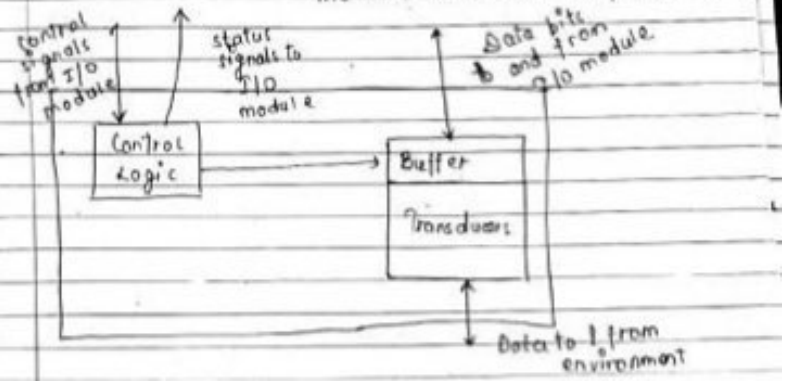


Fig : Block diagram of External device.

The interface to I/O module is in the form of control data & status signals. Control signals determine the function that the device will perform such as send data to I/O module, or accept data from I/O module, report status, etc. Data are in the form of bits which are send or recieved by I/O module. Status signal indicates the state of device. Control logic controls the device operation. Transducer convert the electrical energy in other form of energy. Buffer holds the data temporarily.

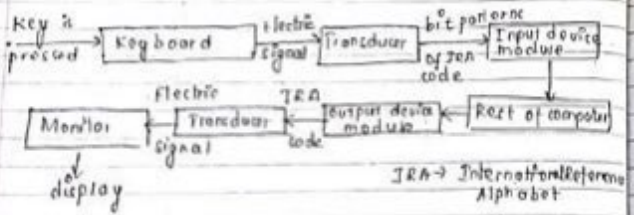


Fig: Working of keyboard monitor.

* Disk drives:
Contains electronic for exchanging data, control and status signal with I/O module. It also controls read write mechanism. Two types of disk drives:
1. fixed head
2. Movable head

Fixed head disk transducer is capable of converting magnetic patterns on moving disk and bits in device buffer. Moving head disk also cause the disk arm to move in and out across disk surface.

Functions of I/O modules:
1. Control and Timing
2. Processor Communication
3. Device Communication
4. Data buffering
5. Error detection.

1. Control and Timing
In any period of time processor may communicate with one or more external device depending upon need of program. The internal resources must be shared among no of activities including data I/O. Thus, control & timing is required by I/O module for the coordination bet internal resources & external device. Required for
Example:
The control of data transfer from an external device to processor involves:
i) the processor integrates the I/O module to check the status of attached device.
ii) the I/O module returns the device status.
iii) if the device is operational & ready to transmit processor request for transfer of data.
iv) I/O module recieves the data from external device.
v) the data are transferred from I/O module to processor.

Processor Communication

Processor involves: command decoding

1. Command decoding: I/O module accepts command from processor sent as control signals. Examples: Read/Write Command for disk.
2. Data: Data exchange takes place between I/O modules and processor through data bus.
3. Status reporting: Peripherals are slow in operation. Thus, status of I/O module should be reported. Status signals are busy and ready.
4. Address recognition: I/O module must recognize the unique address of each peripherals.

Device communication

Device comm^n involves commands, status, information and data exchange.

Data Buffering:

Data transfer rate of main memory and processor is high relative to external device. Data from main memory are send to I/O Module and are buffered to sent them in peripheral at its rate. I/O module must be able to operate at both device & memory speeds.

Error Detection:

I/O module is also responsible for error detection. It also should report the error to the processor. Error includes mechinal & electrical malfunction reported by device.

Examples paper jam.
Another error consists changes in bit pattern as it is transmitted from device to I/O modules. Error detecting code also deals with transmission error.

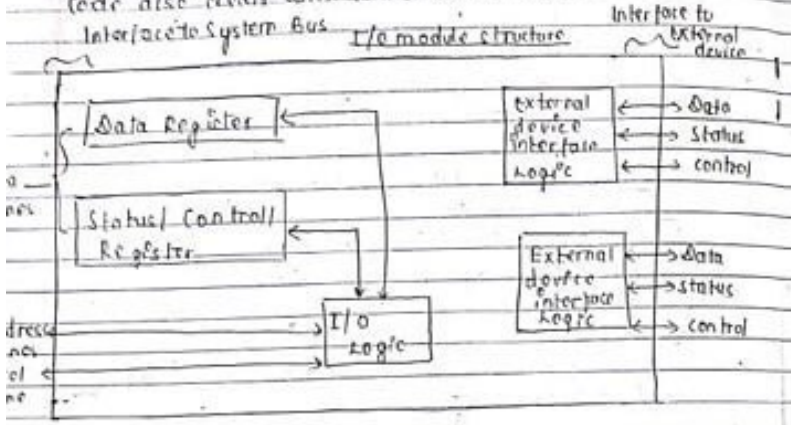


Fig: Block Diagram of I/O module.

Module is connected to rest of the computer through set of signals lines. Data transfer two to and from are buffered in data register. Status register provides the status information. Status register also functions as control register to accept control information from register. Logic interacts with processor through set of control lines. Control lines are used to command to I/O module. The module must also recognize the unique address associated with device it controls.

I/O module allow processor to connected with wide range of devices. I/O module hides the details of peripheral so that processer can function in simple read/write mechanism.
I/O module provides a high level interface to processer. This is also reffered as I/O channel or I/O processor. An I/O module i.e primitive and requires detailed control is referred as I/O controller or device controller. I/O controller and seen on micro controller & I/O channels in mainframe.

* Types of I/O:

1. Programmed I/O

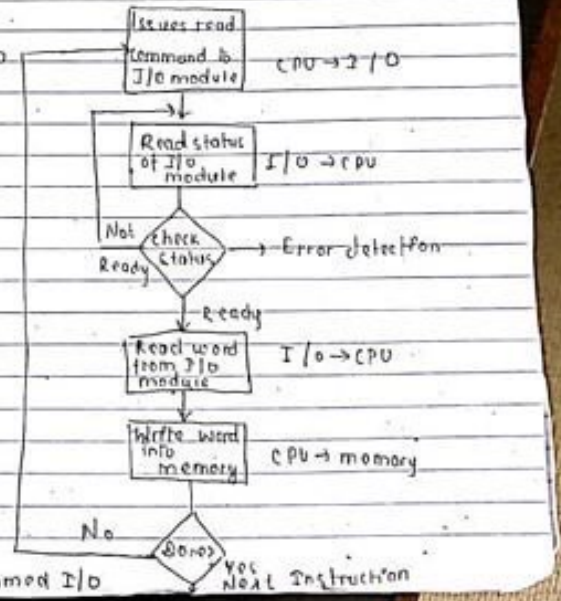


Fig: Programmed I/O

Programmed I/O transfers data between processor & I/O module. When processor issues a command to I/O module, it must wait until the time of operation of I/O. If processor is faster than I/O module, then processor time is wasteful on encounter of an instruction related to I/O module processor issues an I/O command. Programmed I/O performs the requested action and set bits in I/O status register. I/O module doesn't interrupt processor thus, processor needs to check the status of I/O module until it finds the operation is complete. For I/O operation, processor issues an address specifying I/O module, external device and I/O command.

Processor issues four types of I/O commands:-

1) Control
2) Test
3) Read
4) Write

Control: It activates peripheral to perform required action. For examples rewind or forward in an magnetic disk.

Test: Tests the various status conditions associative with I/O modules and its peripherals. Processor checks for the available peripherals. It also checks the most recent operation is completed or not.

Read: I/O module reads data from peripherals and place it into an internal buffer. The processor then obtains data by requesting I/O module to place it on data bus.

Write: I/O module takes data from databus & transmits the data to peripherals.

Programmed I/O reads data from peripherals into memory, in above programmed I/O data are read ie 1 word at a time. For each word ie read, the processor must check status that the word is available in I/O module. Thus, th it makes the processor busy all the time.

I/O instructions:

In programmed I/O, there is the related instruction that processor fetches from memory and I/O commands that processor issues to an I/O module to execute the instruction. Instructions are mapped into I/O command i.e they have one to one relationship.

There is many devices connected through I/O module to system, each device has its own unique address. When processor issues commands, the command contains the address of desired device. I/O module must interpret the address.

When processor, memory and I/O shares a common bus two modes of addressing are possible:

1) Memory mapped

2) Isolated

Memory mapped I/O has a single address space for memory location of I/O devices. The processor treats the status and data register of I/O module as memory location and uses same instruction to access both memory & I/O devices.
Example for 16 bit address lines, a combined total of 2^{16} = 1024 memory location and I/O address can be supported in any combination. Memory mapped I/O has a single read line and a single write line on the bus. The bus may be equipped with memory read & write plus input/output commands.

Isolated I/O supports both I/O address space & memory address space i.e for 10 address line, it has 1024 memory locations & 1024 memory & I/O addresses as I/O address & memory is I. Isolated, it is isolated I/O. Memory Memory map can give large set of instructions. Isolated I/O can give few I/O instructions. For example: for 16 bits address, 512 bit is used for memory & 512 bit is used for I/O.

3) Interrupt Driven I/O

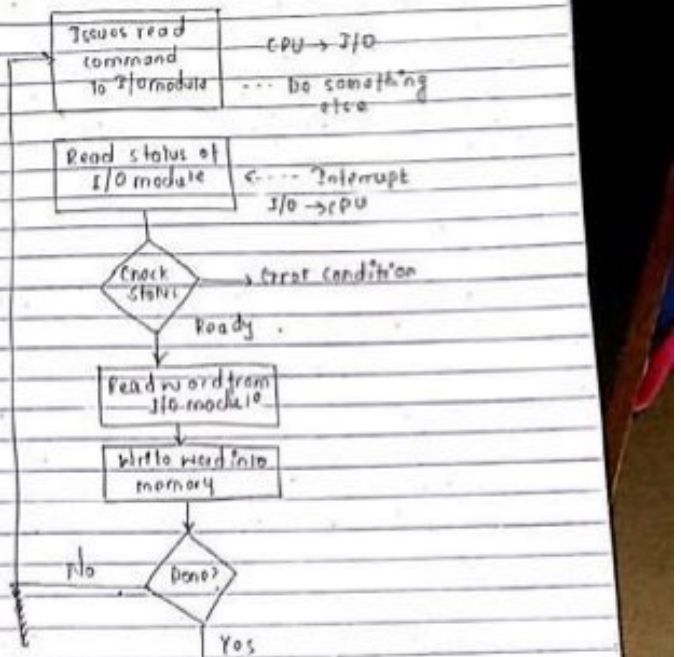


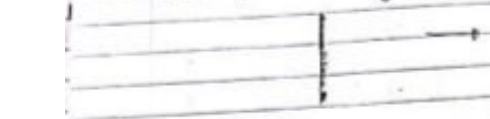
Fig: Interrupt Driven I/O

To overcome the problem with programmed I/O, interrupt driven I/O is used. For this processor issues and I/O command to module and go to do other work. When I/O module is ready, it issues an interrupt to processor. The processor then executes the transfer exchange data. Then processor then executes the transfer & then resume it former processing.

Let us illustrate how it works: I/O module recieves a read command from processor. The module then communicate with peripheral when the data is in data register of I/O module, it issues an interrupt to processor. When the processor request, the module places the data in data bus & it is ready for next I/O operation.

Processor checks for interrupt at every end of instruction cycle. When it recieves the interrupt, processor saves the content, processes the interrupt and then resume the former processing.

Interrupt Processing



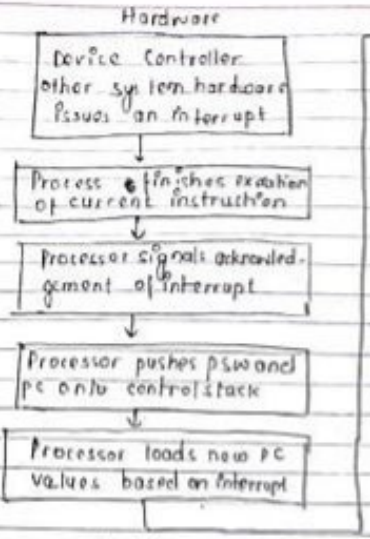


Fig 1 Simple Interrupt Processing

The occurence of interrupt triggers a number of events in processor hardware and software. Figure above shows the sequence:-

i) Device issues and interrupt signal to the processor.
ii) Processor first finishes the current execution and then responds to the interrupt.
iii) Processor tests for interrupt, if there is 1, it sends an acknowledgment signal to device. This allows the device

to remove its interrupt signal

iv) The processor may processes the interrupt routine. For this purpose, it saves the information of current program. Status of processor is saved in PSW and the location of next instruction in PC. These are then pushed to system control stack.

v) Processor now loads the PC with entry location of interrupt handling program. If there is more than one interrupt handling routine, processor must decide which one to invoke.

Once, PC is loaded, processor proceeds to the next instruction cycle which begins with instruction fetch. Instruction fetch is determined by PC, thus control is transferred to interrupt handler program. The execution results in following operations.

vi) Here, the program counter & PSW relating to the interrupt program are saved to system stack. The content of the processor needs to be saved as it may be used by interrupt handler. Interrupt handler will begin by saving the contents of all of the registers on the stack.

vii) The interrupt handler next processes the interrupt. This includes an examination of status information relating to the I/O operation or other event that causes interrupt. ~~When interrupt p~~

viii) When interrupt processor is complete, the saved register value are retrieve from stack and restored to registers.

ix) Finally PSW and PC values are restored.

Design Issue for Interrupt Driven I/O

There are two design issues while implementing interrupt driven I/O:

i) For multiple I/O modules, how does processor determines which device issued the interrupt?

ii) For multiple interrupt which one to process first?

for device identification there is four techniques.

1. Multiple Interrupt lines
2. Software Polling
3. Daisy chain
4. Bus Arbitration.

1. For multiple interrupt lines, devices are connected in multiple lines. Multiple interrupt lines can be issued in both processor & I/O modules.

When the processor detects an interrupt, it branches to interrupt service routine whose job is to poll each I/O module which device cause the interrupt. The pole could be in the form of command, example: test I/O. Processor issues Test I/O and places the address of I/O module on address line. The I/O module response if it set the interrupt. I/O module contains an addressable status register. The processor reads the status of I/O module. On Identification of correct module, the processor branches to device service routine of specific

device. Its main disadvantage is: it is time consuming.

A different technique is daisy chaining. It provides hardware poll. For interrupt, all I/O module shares a common interrupt request line. The interrupt acknowledgement line is daisy chained through the modules. When processor senses an interrupt, it sends out an interrupt acknowledgement. The signal propagates through series of modules until it gets the requesting module. The requesting module responds by placing words on data lines. Word is a vector that may be I/O module address or other unique identifier. This is also known as vectored interrupt.

With bus arbitration, I/O module first gain the control of bus before it raise interrupt request line. Thus, only one module can raise interrupt at a time. When the processor detect interrupt, it responds on interrupt acknowledgement line.

2. With multiple interrupt line the processor just picks the interrupt line with highest priority. With Software polling, the order in which modules are polled determines their priority. The order of modules on a daisy chain determines their priority.

Direct Memory Access (DMA)

DMA is used to overcome the drawbacks of programmed I/O and interrupt driven I/O. These two I/O suffer from two major drawbacks:

(i) The I/O transfer rate is limited by the speed with which the processor can test and service a device.

(ii) The processor is tied up managing an I/O transfer, a number of instruction must be executed for each I/O transfer.

For large volume of data to be moved DMA is to be used. DMA involves an additional module on the system bus. DMA takes control of system from processor. When processor wishes to read and write a block of data it issues a command to DMA module by sending following information:-

i) Read/Write Request

ii) The address of I/O device involved, communicated on the data lines.

iii) The starting location in memory to read from or write to. These are communicated on data lines and is stored in address register of DMA.

iv) The number of words to be read or written.

The processor than continues with other work. Now the transfer takes place between I/O DMA and memory without the involvement of processor. When the entire block of data is transferred, it sends an interrupt signal to processor. Thus, processor involvement is only at the beginning and end of transfer.

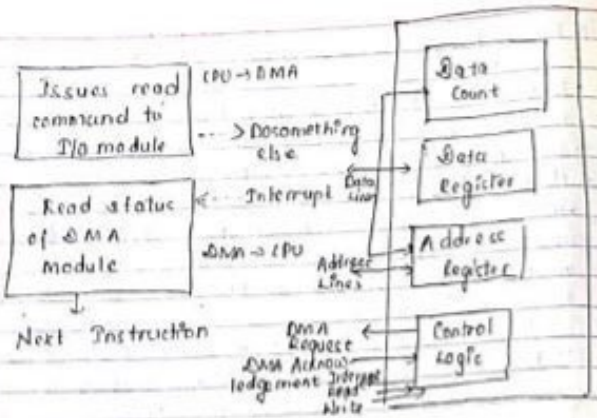


Fig: Direct Memory Issues

Fig: DMA Block Diagram

DMA modes of data transfer

1. Burst mode
2. Cycle stealing mode
3. Transparent mode.

1. CPU gives the initialization of no. of bits to DMA as instruction. DMA takes control of system bus. With burst mode, data transfer takes place continuously till all the data are transferred. This mode is needed for fast device such as magnetic disk where data transmission cannot be stopped or slowed down. It makes CPU ideal for long period.

2. Cycle stealing:
CPU and DMA controller shares the clock cycle i.e first cycle by CPU and next by DMA controller & process is repeated i.e DMA steals the cycle of CPU.

3. Transparent mode:
Most efficient in terms of time processing. DMA controller takes control of system bus only when CPU is not using it. Thus, CPU is not ideal in this case. But if CPU is using bus all the time, DMA couldn't take the control of system bus. This mode requires hardware to detect the state of system bus.

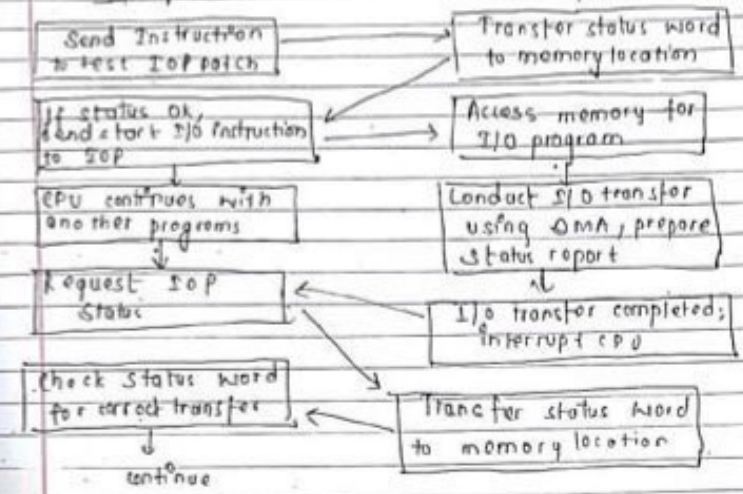


Fig: CPU-IOP communication

* I/O channels & I/O processor

I/O channels are the modules that behaves like processor with specialized instruction set, CPU directs the processor to execute a I/O program in memory. I/O processor fetches and executes instruction without any involvement of CPU. CPU is interrupted only when I/O activities are completed for I/O processor, module has its own local memory. With this large set of I/O devices can be controlled with minimal involvement of CPU. It controls the communication with interactive terminals.

*Characteristics of I/O channel:-

I/O channel executes I/O instructions that completely control I/O operation. CPU doesn't executes I/O instruction instead all instructions are stored in main memory that can be executed by I/O processor.
CPU only initiates I/O channels to perform transfer. The instruction specify device or devices, area and or areas of memory storage, priority and access to be taken.

Types of I/O channels:-
1. Selector
2. Multiplexer

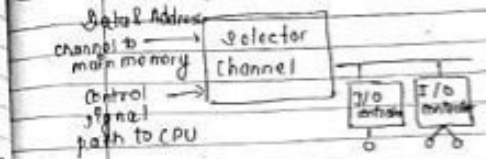


Fig : Selector

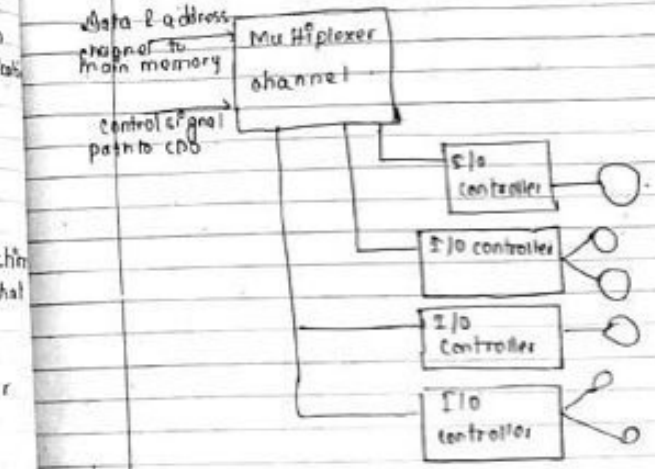


Fig : Multiplexer

A selector controller multiple high speed devices but one at a time. I/O channel selects one device and data-transfer takes place.
A multiplexer channel can handle multiple I/O devices at same time. For high speed device, a block multiplexer interleaves block of data from the several devices.

External Interface

1. Serial Interface
2. Parallel Interface.

Fig: Parallel interface.

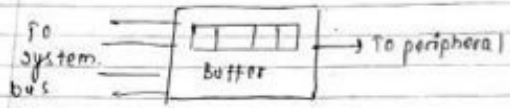


Fig: Serial Interface.

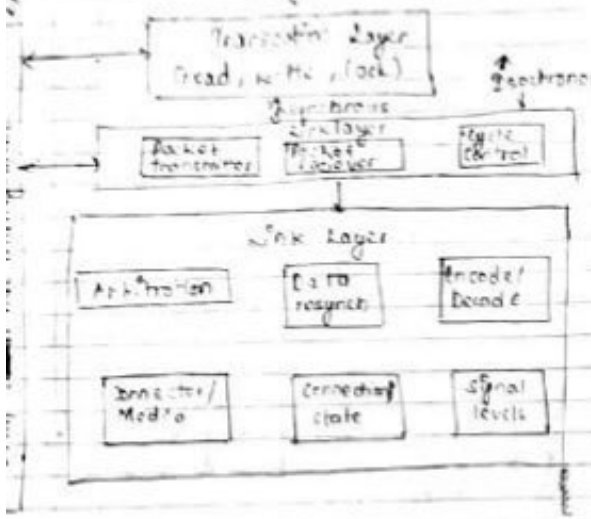
In serial interface, only one line is used to transmit data and bits must be transmitted one at a time. In parallel interface multiple lines connects I/O module and peripheral thus, multiple bits are transferred simultaneously. Parallel interface is used for high speed device like tape & disks and serial interface is used for printers & terminals.

Firewire and InfiniBand Architecture

Firewire

* Firewire interfaces has advantages over older I/O interfaces. It is very high speed low cost and easy to implement. Firewire supports other electronic devices also. It uses serial transmission rather than parallel. It provides a single I/O interface with the simple connector that can handle numerous devices through a single port so that mouse, printer, sound and land-lookup can be replaced with the single connector. It provides a 3-layer protocol:
  (i) Physical layer
  (ii) Link layer
  (iii) Transaction layer.

Fig: Firewire protocol stack



Physical layer specifies various transmission media and their character with different physical and data transmission properties. Data rates are 25 to 400 Mbps. The physical layer converts binary into digital signal. It also provides arbitration service that guarantees only one device at a time will transmit data.

2. Link layer: This layer defines the transmission of data in the form of ~~package~~ packets. Two types of transmission are supported: asynchronous & isochronous. In asynchronous packet data are transferred to an explicit address and acknowledgment is ~~written~~ return. In isochronous, transfer takes place in a sequence of fixed size packets transmitted at regular intervals, no acknowledgment is return.

3. Transcation layer: defines a request response protocol that hides the lower layer details.

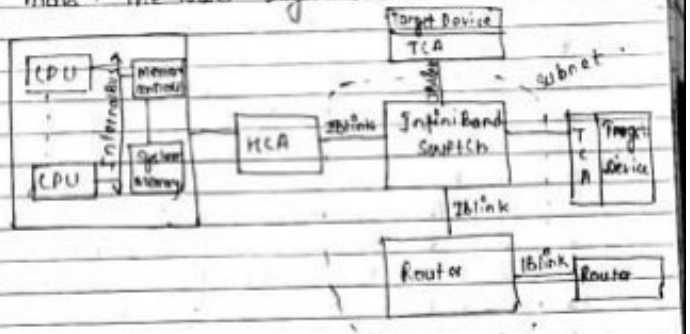


Fig: INFINIBAND switch

- Infiniband is generally designed for server. It enables servers, remote storage and other network devices to be attached in a central fabric of switch and links. Infiniband channel enables 64 devices to be placed

17 m away using cu wire, 300m using multimode optical fibre and upto 10km using single mode optical fibre. The key elements of this architecture are:

i) Host channel adapter: HCA links the ser server to an infini-band switch. HCA is attached to memory controller that has access on system bus and controls traffic between processor and memory and between HCA & memory. It uses DMA to read and write.

ii) Target channel adapter: TCA connects storage system routers and other peripheral devices to infini-band.

iii) Infini-band switch: A switch provides point to point physical connection to a variety of devices & switches traffic from one link to another. The switch is used for communication betn server & devices through adapters. They manages the linkage without interrupting servers operation.

iv) Links: Links betn switch and channel adapter or betn 2 switch.

v) Sub-net: consists of one or more interconnected switch plus links that connect other devices to switches. Sub-net allows administrators to confine broadcast & multicast transmission within the subnet.

vi) Router: connect: subnet or switch to a router network

Chapter- 8

Reduced Instruction set computers (RISC)

→ Pipelining
- Risc & Instruction pipelining
- pipelining Hazards
- Register windowing.

RISC:
A computer with fewer instruction with simple constructs for faster execution within CPU without having use of memory is classified as RISC.

Characteristics of RISC:
i) Relatively few instruction
ii) Relatively few addressing modes
iii) Memory access is limited to load & storage store instruction.
iv) All operations are done within the registers of CPU
v) Fixed length, easy decoded instruction format.
vi) Single cycle instruction execution
vii) Hardwire rather than microprogrammed controlled unit
viii) For load & store it uses register to register movement.

11 Parallel Processing

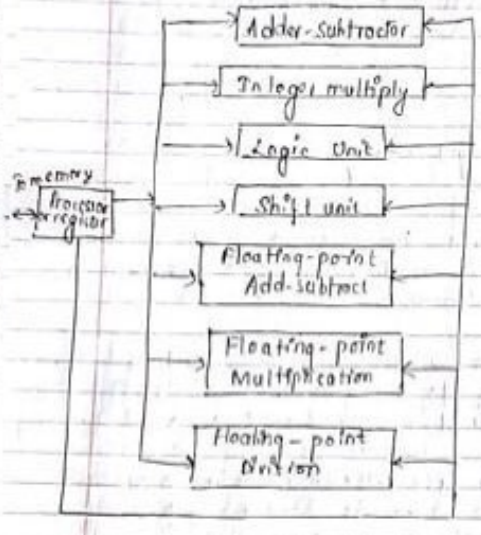


Fig: Processor with multiple functional units

Parallel processing is the processing of data concurrently to achieve the faster execution time or simultaneous data processing. for the purpose of increasing computational speed of a computer system. For example when an instruction is being executed in ALU, the next instruction can be read from memory. The purpose of parallel processing is to speedup the computer processing capability and increase its through put i.e the amount of processing that can be accomplished in given interval of time. Above figure gives one possible way of separating execution unit into 8-functional unit operating in parallel.

classification of Parallel Processing

Flynn's classified computer in four groups:-

1. Single Instruction Stream, single data stream (SISD)
2. Single Instruction Stream, Multiple data stream (SIMD)
3. Multiple Instruction Stream, Single data stream (MISD)
4. Multiple Instruction stream, Multiple data stream (MIMD)

This classification is preferred according to how data and instruction can be executed simultaneously. Instruction stream is the sequence of instruction read from memory. Data stream is the operation performed on data in the processor. Parallel processing occurs in instruction stream or data stream or in both. Flynn classified computer in above given 4 groups.

SISD represents the organization of a single computer containing a control unit, a processor and a memory unit. Instructions are executed simultaneously and the system may or may not have internal parallel processing. Parallel processing is through

multiple functional unit. SIMD ha:

SIMD has organization including many processing units under the supervision a common control unit. Receives same instruction but operates on different items of data.

MISD is only theoritical implementation. No pratical system has been constructed using this organization.

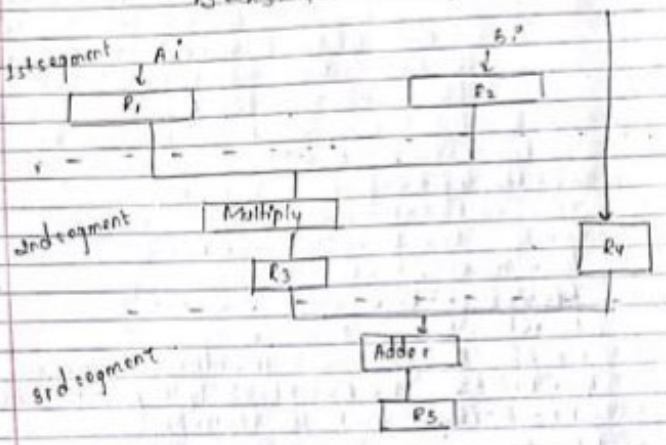
MIMD refers to computer capable of processing several programs at the same time. Most multi-processor & multiple computer system is classified in this category.

Pipelining

Pipelining is a technique of decomposing a sequential process into sub-operation, with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments. It is a collection of processing segments through which binary information flows. Each segments processes partially the partition task. The result from each segment is transferred to next segment in pipeline. The final result is obtained after the data have passed through all segment.

Example: $A_i * B_i + C_i$ for $i = 1, 2, 3, \ldots, 7$

Here, this example say to perform the combine multiply and add operation with the stream of number. Each sub operation is to be implemented in a segment within a pipeline. Each segment has 1 or 2 register and a combinational circuit. sub-operation in each segment are $R_1 \leftarrow A_i$, $R_2 \leftarrow B_i$ → input A_i & B_i
$R_3 \leftarrow R_1 * R_2$, $R_4 \leftarrow C_i$ → multiply R_1 & R_2 and input C_i
$R_5 \leftarrow R_3 + R_4$ → Add R_3 and R_4



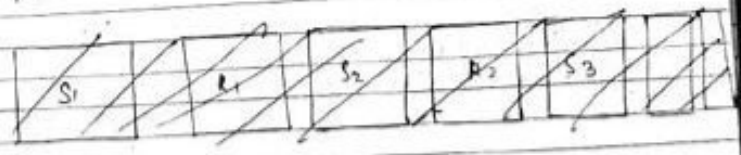
Clockcycle	1st segment		2nd segment		3rd segment
	$R_1 \to A_1$	$R_2 \to B_1$	$R_3 \to R_1 * R_2$	$R_4 \to C_1$	$R_5 \to R_3 + R_4$
1	A_1	B_1			
2	A_2	B_2	$A_1 * B_1$	C_1	
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8			$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9					$A_7 * B_7 + C_7$

F: $(A_i * B_i) / (C_i + D_i)$

~~$R_1 \leftarrow A_i$, $R_2 \leftarrow B_i$ → Input A_i, B_i~~
~~$R_3 \leftarrow A_i * B_i$, $R_4 \leftarrow C_i$~~
~~Add & Div $R_5 \leftarrow R_3 / R_4$~~
~~$R_5 \leftarrow D_i$, R_6~~
$R_1 \leftarrow A_i$, $R_2 \leftarrow B_i$, $R_3 \leftarrow C_i$, $R_4 \leftarrow D_i$ → Input A_i, B_i, C_i, D_i
$R_5 \leftarrow R_1 * R_2$, $R_6 \leftarrow R_3 + R_4$ → Multiply $R_1 * R_2$ and $R_3 + R_4$
$R_7 \leftarrow R_5 / R_6$ → Divide R_5 and R_6.

Speedup = $\frac{\text{non pipeline}}{\text{pipeline}}$
$= \frac{27}{9} = 3$.

General considerations:
Example: The general structure of 4 segment pipeline is illustrated in figure below. The operand pass through all 4 segments in a fixed sequence. Each segment consists of a combinational circuit Si that performs a sub-operation over the data stored following through pipe. The segments are separated by registers 'Ri' that holds intermediate results between the stages. Task is the total operation performed going through all segments in pipe.



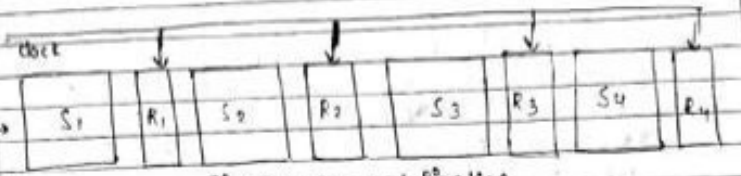


Fig: Four segment pipeline

Segment/ clockcycles	1	2	3	4	5	6	7	8	9
1	T_1	T_2	T_3	T_4	T_5	T_6			
2		T_1	T_2	T_3	T_4	T_5	T_6		
3			T_1	T_2	T_3	T_4	T_5	T_6	
4				T_1	T_2	T_3	T_4	T_5	T_6

Fig: Space Time diagram for no. of Task = 6

k = 4
n = 6
Clock = k + n - 1
= 4 + 6 - 1
= 9

Here, horizontal axis gives the clock cycle and vertical axis gives the segment number. Six task T_1 to T_6 in four segment. When the task T_1 is passed through all the segments, it is completed in 4th clock cycle. After the 1st task is completed, at every clock cycle next task is completed.
Now, k segment pipeline with time period tp to execute n task. First, task T_1 requires time = ktp where
k = no. of segment
tp = time period of segment
T_1 = k tp
The remaining (n-1) task emerge from one-pipe at 1 task per cycle. Therefore,
total time = (n-1) tp
For non-pipeline unit, performing the same operations takes time equal to tn to complete each task.
Therefore, total time is ntn.

* The speed up of pipeline over non pipeline is given by ratio
S = total time of non pipeline / total time of pipeline.

= ntn / (1+n-1) tp

As no. of task increases n becomes much larger than than k-1 and k+n-1 approaches to the value of n.

∴ S = ntn / ntp
= tn / tp.

For time taken same in pipeline and non-pipeline.
$t_n = k t_p$
S = k tp / tp
= k

∴ The maximum speedup that a pipeline can provide is k, where k is the no. of segment in pipeline.

Q1. tp: 20ns
k = 4
n: 100 tasks
Find the speedup that a pipeline can provide over nonpipeline.

Non pipeline
$S = nkt_p$
$= 100 \times 4 \times 20$
$= 8000$ ns

Pipeline
$t_p = 20$ ns
$= 4 \times 20$ ns
$= 80$ ns

total clock cycle for 100 task $= (k+n-1)$
$= 4 + 99$
$= 103$

$S = \frac{nkt_p}{(k+n-1)t_p}$
$= \frac{80 \times 100}{103 \times 20}$ ns
$= 3.8834$

Arithmetic Pipeline

Arithmetic pipeline unit are used in high speed computers. They are used to implement floating point operations, multiplication of fixed point notation. example: Floating point addition and subtraction

eg: $X = A \times 2^a$
$Y = B \times 2^b$

Here, addition and subtraction can be performed in four segments. The sub-operations are:-

1. Compare the exponent
2. Align Mantissa
3. Add or subtract
4. Normalize the result

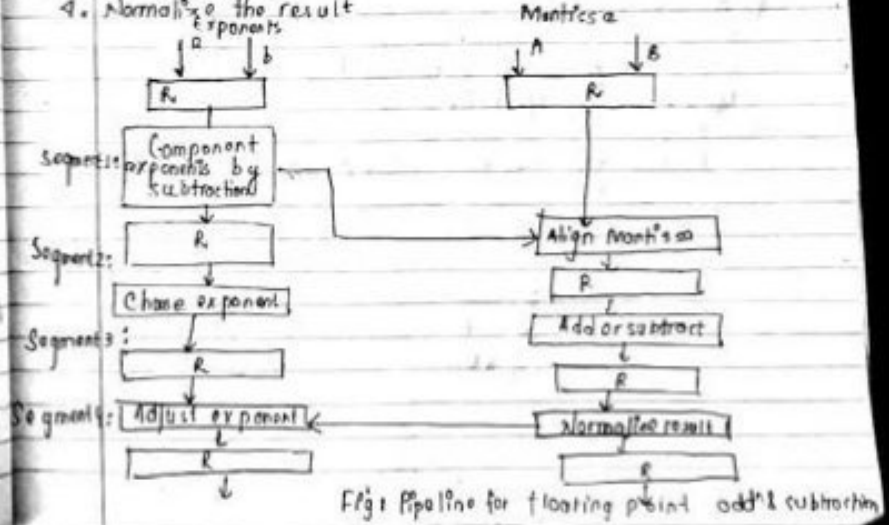


Fig: Pipeline for floating point add & subtraction

The exponent are compared by subtracting them to determine their difference. The largest exponent is chosen as the exponent of result. The exponent determines how many times the mantissa associated with smaller exponent must be shifted to right. This gives alignment of mantissa. Shift must be designed as a combinational circuit to reduce the shift time. Two mantissa's are added or subtracted in segment 3. The result is normalized in segment 4. For overflow the sum of difference of mantissa are shifted right and exponent is increased by 1. For underflow the number of leading 0 in mantissa determines the number of left shift and the number is subtracted from exponent.

Q. Suppose time delay's for 4 segment are $t_1 = 60ns$, $t_2 = 70ns$, $t_3 = 100ns$, $t_4 = 80ns$ and interface register $t_r = 10ns$. Find the speedup over of pipeline over nonpipeline.

→ $k = 4$

$t_n = 60 + 70 + 100 + 80 + 10$, $S = \frac{t_n}{t_p}$

$= 320$

$t_p = 100 + 10$

$= 110$

$= \frac{320}{110} = 2.84 \approx 2.90$

* Instruction pipeline

1. Fetch the instruction
2. Decode the instruction
3. Calculate effective address
4. Fetch the operand
5. Execute the instruction
6. Store the result

} 6 sub-operation

4 op.

Pipeline can also occurs in instruction string. An instruction pipeline reads consecutive instruction from memory while previous instruction are being executed in other segment. Here, instruction fetch and execute phases overlap and perform simultaneous operation.

Example: An instruction fetch unit and an instruction execut unit provide two segment pipeline. The instruction fetch segment can be implemented by First-In-First-Out Buffer. Here, when execute unit is not using memory, the control increases the program counter and uses its address value to read consecutive instruction from memory.

For complex instruction, computer requires add phases other then fetch and execute. Sequence of steps to process an instruction are :

1) Fetch the instruction
2) Decode the instruction
3) Calculate effective address
4) Fetch the operand
5) Execute the instruction

6). Store the result

There are difficulties that will prevent instruction pipeline to operate at its maximum rate. Different segments may take different time to operate on the incoming information. Some segments are skipped for certain operation. Thus to design an instruction pipeline in efficient way, instruction cycle is divided into segment of equal duration. The time that each step take to fulfil its function depends on the instruction & the way it is executed.

Example 4 segment instruction pipeline. Here decode the instruction and calculate effective address is combined in one segment and execute the instruction & store the result in one segment.

Fetch instruction
from memory

→

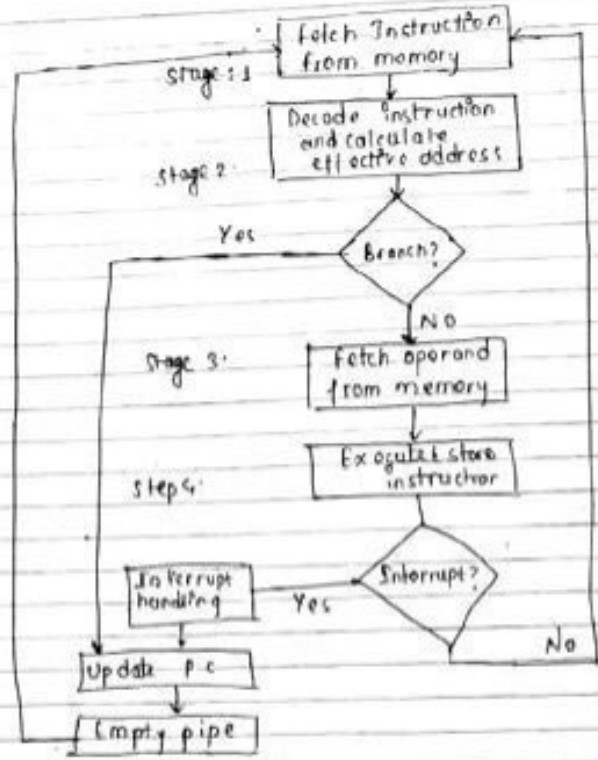
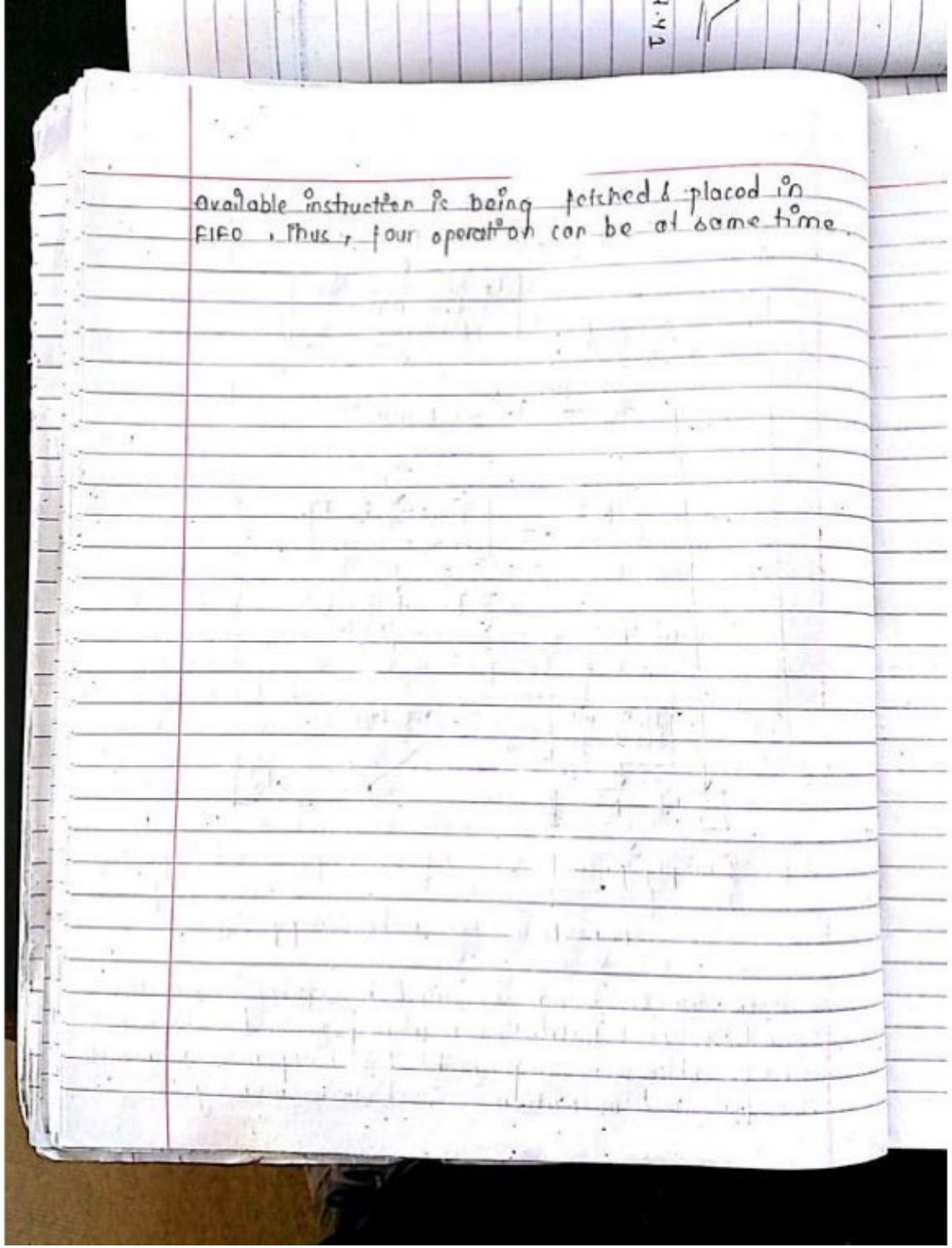


Fig: Four stage instruction pipeline

- An instruction is being executed in segment 4, in the same time next instruction is fetching operand in segment 3 effective address is being calculated in a seperate arithematic ckt for 3rd instruction and when memory is


available instruction is being fetched & placed in FIFO, thus, four operation can be at same time.

FI → Segment that fetch instruction.
DA → segment that decode instruction and calculate effective address.
FO → segment that fetches operand
EX → segment that executes & stores instruction

Step / Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
3(branch)			FI	DA	FO	EX							
4				FI	-	-	FI	DA	FO	EX			
5								FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

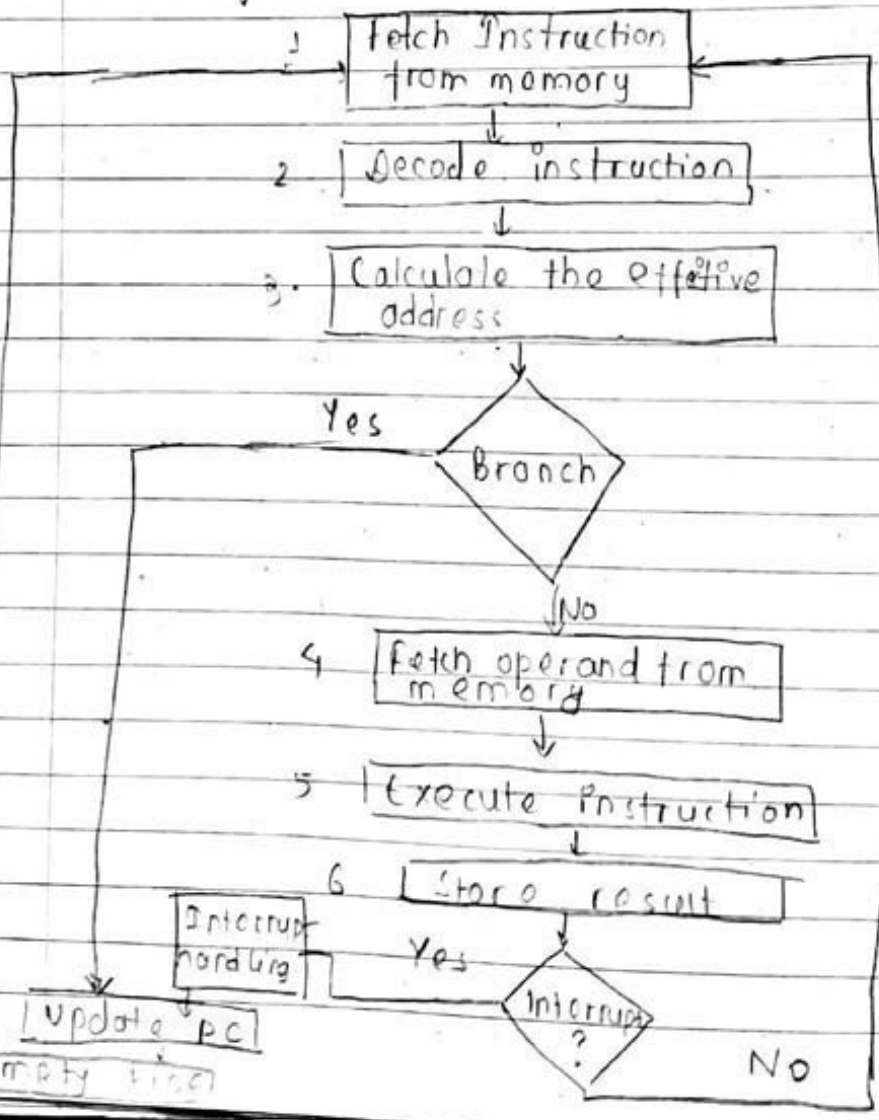
Assume that processor has separate instruction & data memories. Thus, FI & FO proceeds at same time. In absence of branch, each segment operates of different instruction. Thus, in step 4 instruction 1 is being executed in segment EX, the operand for instruction 2 is being fetched in segment FO, instruction 3 is decoded in segment DA and instruction 4 is being fetch from memory in segment FI.

For instruction 3, being a branch instruction. Here as soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA is haulted until the branch

instruction is executed. If the branch is taken, a new instruction is fetched in step 7. If branch is not taken instruction fetched previously in step 4 can be used.

Another delay may occur in the pipeline if the EX segment needs to store result of the operation in the data memory while FO needs to fetch an operand. In that case, FO must wait till EX finishes it execution.

Six stage instruction



Conflicts in pipeline

1. Resource conflict:
It is caused when two segment access memory at same time. These conflicts can be removed by using separate memory for data and instruction.

2. Data dependency
It occurs due to collision of data or address. A collision occurs when an instruction cannot proceed because previous instruction did not complete certain operation. Data dependency occurs when an instruction need data that are not yet available. Example:
Instruction I0 may need the operand i.e generated by Ex of previous instruction and that may be still executing. An address dependency occurs when an operand address cannot be calculated due to the information not being available needed by addressing mode. Examples An instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into registers. Thus, operand access to memory must be delayed till the required address is available.

Imp
Hazards

Dealing with the difficulty:
1. Hardware interlocks
2. Operand forwarding
3. Delayed load

Hardware interlock:
Hardware interlock can be inserted. Interlock is a circuit that detects the instruction whose source operands are the destination for other instruction in pipeline. After detecting the instruction, it is delayed by enough clock cycle to remove the conflicts.

Operand forwarding:
It needs a special hardware to detect the conflic & then, avoids it by routing the data through special path. Example: Instead of transferring ALU result to destination register. The hardware checks the operand & if it is required as source in next instraction it passes the result directly into input. This method requires additional hardware to detect the conflict.

Delayed load:
Here, compiler detects the conflicts & insert no operatio to avoid the conflict. The 'no-op' instruction can be done by re-ordering instruction that is necessary to delay.

Eg: Operations for four instructions

1. LOAD : $R_1 \leftarrow M[address]$
2. LOAD : $R_2 \leftarrow M[address]$
3. ADD : $R_3 \leftarrow R_1 + R_2$
4. STORE : $M[address] \leftarrow R_3$

Let us Assume the instruction proceeds in three segment pipeline.

I : Instruction fetch

A : ALU operation (Instruction decoding, Address calculation, operand fetch and ALU operation)

E : Execute operation

Step / Instruction	1	2	3	4	5	6	7
	I	A	E				
LOAD: R_1			I	A	E		
LOAD: R_2				I	A	E	
ADD: R_1+R_2					I	A	E
STORE: R_3							

Here conflict occcur in instruction 3 because the operand R_2 is not available in segment A. A is using the data from R2 in same clock cycle. Re is not correct value since, it has not been transferred from memory. Now, compiler must make sure that instruction following load instruction

uses the data fetch from memory. If the compiler cannot find useful instruction to put after load it inserts no operation instruction i.e it has no operation & wastage clock cycle. This is known as delayed load.

Advantage is: it requires no hardware to remove the conflict.

Step / Instruction	1	2	3	4	5	6	7
LOAD : R1	I	A	E				
LOAD : R2		I	A	E			
No operation			I	A	E		
ADD: R1 + R2				I	A	E	
Store: R3					I	A	E

Fig: Timing diagram without data dependency.

3) Branch conflict:

A branch instruction can be conditional or unconditional, and an uncondition branch always alters the sequential flow by loading the PC with target address. In a conditional branch the control select the target instruction if the condition is satisfy or next sequential instruction if the condition is not satisfy. The branch instruction breaks the normal sequence.

* Handling of conditional Brach
1. Prefetch target Instruction
2. Branch Target Buffer
3. Loop Buffer
4. Branch prediction
5. Delayed Branch

1. Prefet

1. Here, the target instruction is prefetched along the instruction following the branch. Both are saved till the branch is executed. If the branch condition is successful, the pineline continues from the branch-target instruction.

2. Branch Target Buffer:
It is an associative memory included in fetch segment of the pipeline. Each entry in BTB consists of address of a previously executed branch instruction and target instruction for the branch. It also stores the next few instruction after the branch target instruction. When pipeline decodes the branch instruction, it searches in BTB for the address of the instruction. If it in BTB, the instruction is available and is prefetched directly. If not present in BTB, the pipeline shift towards new instruction & stores the target instruction in BTB.

3. Loop Buffer:
Loop Buffer is a small high speed registor maintained by fetched segment in pipeline. When a program loop is detected, it is storaid in loop buffer with all branches. Now, it can be executed directly without having access to memory till loop mode is removed by final branching out

4. Branch prediction
A pipeline with branch prediction uses some logic to guess outcome of a conditional branch instruction before it is executed. The pipeline then, prefetches the instruction string from predicted path.

5. Delayed Branch
Here, compiler detects the branch instruction & rearrange instruction for operation of pipeline without interrupt. No operation instruction is inserted after a branch instruction. This causes the computer to fetch the target instruction during no operation.

eg

Load from memory to R1
Increment R2
ADD R3 to R4
Subtract R5 from R6
Branch to address X.
Instruction in X

Let, us consider 3-stage instruction as in the case of delayed load.

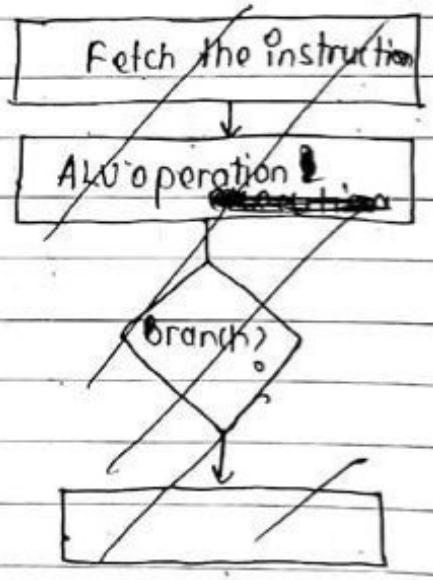
Clock cycles	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. subtract				I	A	E				
5. Branch to x					I	A	E			
6. No-op						I	A	E		
7. No-op							I	A	E	
8. Instruction in x								I	A	E

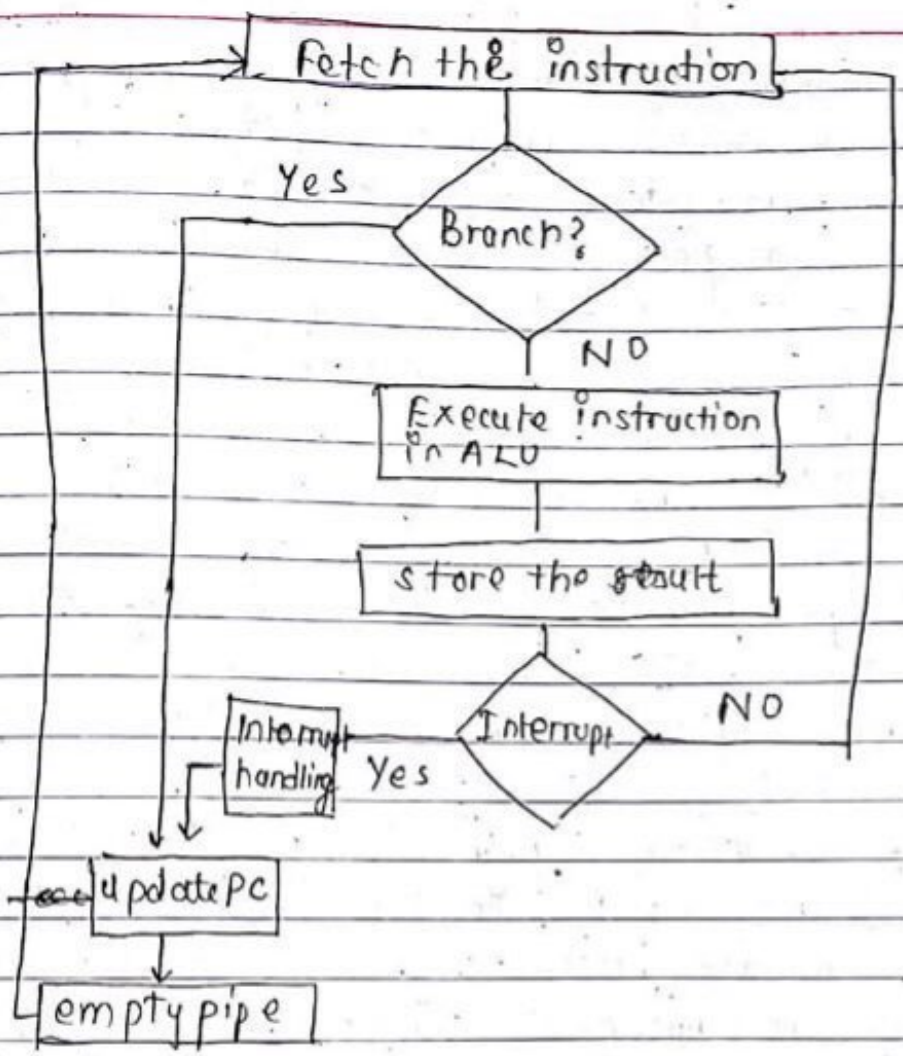
Rearranging the Instruction

Clock cycle	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to x			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in x						I	A	E

* RISC Pipelining

- effective way of using instruction pipelining.





RISC is the ability to use an efficient instruction pipeline. It uses instruction pipeline using a small no of sub-operation with each being executed in one clock cycle. Because of fixed length instruction format, the decoding of the operation can occur at the same time as the register selection. All operands are present in registers. thus, there is no need to calculate effective address or fetching operands from memory. Thus, instruction pipeline can be implemented with 3 segments :-

1. Fetch the instruction
2. Execution in ALU
3. Store the result.

RISC machines use two memories for storing instruction and data. These reduces conflicts between a memory access to fetch an instruction and to load or store an operand. RISC executes instruction at the rate of one per clock cycle. It is not possible to expect that every instruction be fetched from memory and executed in one clock cycle. What is done is to start each instruction with each clock-cycle and to pipeline the processor to achieve the goal of single clock cycle instruction execution. Advantages of RISC over CISC is it uses pipeline segment that requires only one clock cycle.

RISC compiler translates high level program to machine language program and instead of having additional hardware to handle branch & data conflicts, it relies on minimizing this effect by compiler by inserting no operation instruction or rearranging instruction.

overlapping windows in RISC

Imp

* Register window

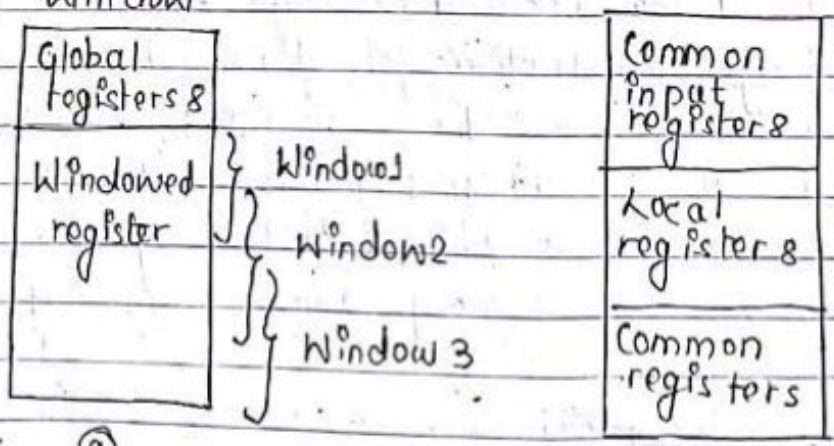


Fig: Register windowing in SPARC Process.

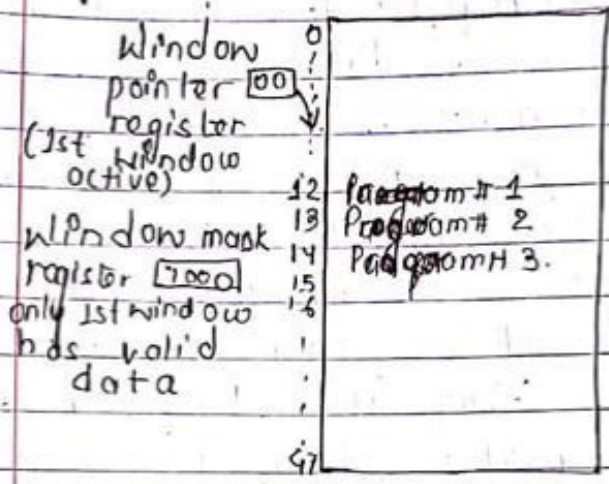
Similarly, (d^3y)

The reduced hardware of RISC processor leaves addⁿ hardware space available on chip. RISC CPU's use this space to include a large number of register. CPU access data more quickly in register than in memory. Thus, having more register makes data available, fast and also reduces the memory refrences.
Although RISC processor has large number of register, it may not be able to access all of them at any given time. Global registers are always available. other register are windowed i.e only a subset of register is accessible at any given time. Figure above explains windowing of registers. Here, this processor can access any of 32 different registers at a given time. Of these 32 register, 8 are available as global registers and remaining register are in register window. Register window overlap each other. In SPARC processor, last 8 register of 1st window are also the first 8 reg of window 2. and last 8 reg. of window 2 is 1st 8 reg. of window 3. The middle 8 reg. of window 2 are local.

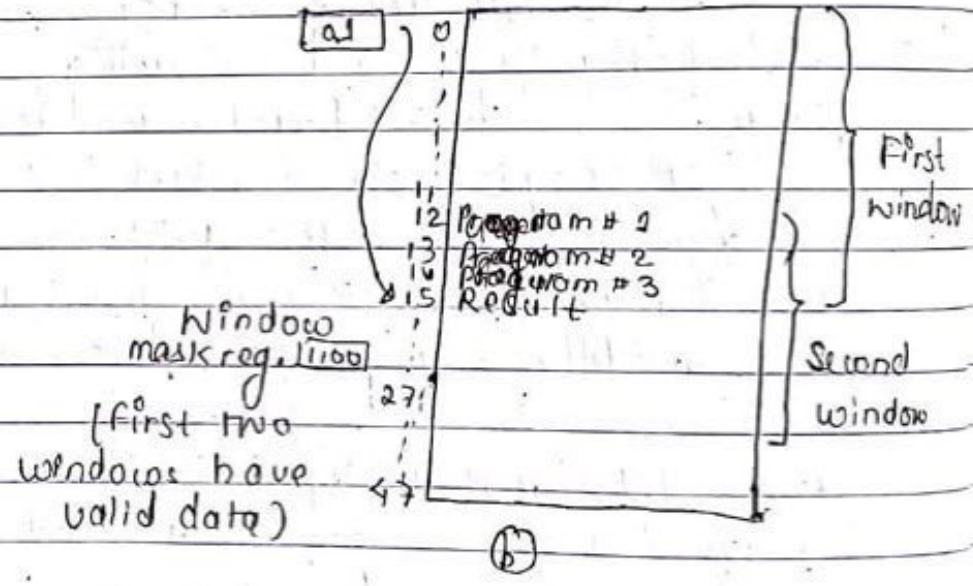
RISC CPU must keep the track of which window is active & which contains the valid data. For this, window pointer reg. is used to point the active window. A 1-bit window mask register is used that denotes valid windows.

RISC Register window is benefitial when CPU calls sub-routine. When calling process takes place, processor activates register window. Example:

let us consider a CPU with 48 reg with three windows with 16 reg each and overlap of 4 register bet^n window.



(a)



(b)

Initially CPU is running on the 1st window as in fig (a), Here subroutine is called and 3 parameters are to be passed. CPU stores this parameters in 3 of overlapping registers and calls the sub-routine.

Subroutine access the parameter, calculates and result to be return to the main program is again stored in the overlapping register. Here, 1st and 2nd window is active as in fig (b). Register windowing is not linear but circular. The last window overlaps ~~with 1~~ window 1.

Register renaming is used to add flexibility to the idea of register windowing. A process that uses register renaming can use any register to comprise its working reg. window. The CPU uses pointer to keep track of active register and which physical register corresponds to which logical register. Register renaming allows any group of physical register to be active.

dy/dp = dy_m + 2r

Chapter - 9

Introduction to parallel processing :-

Characteristics

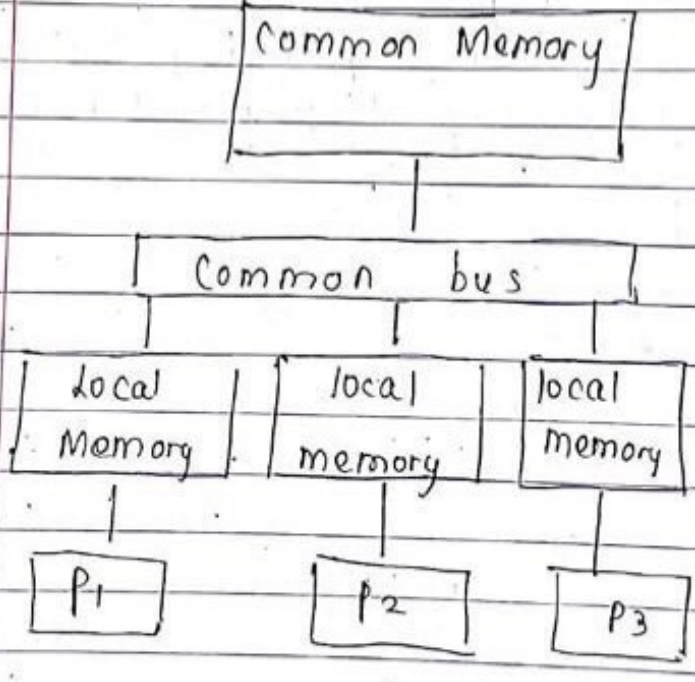
Multiprocessor

Flynn's classification

Enslow classification

↳ Tightly coupled

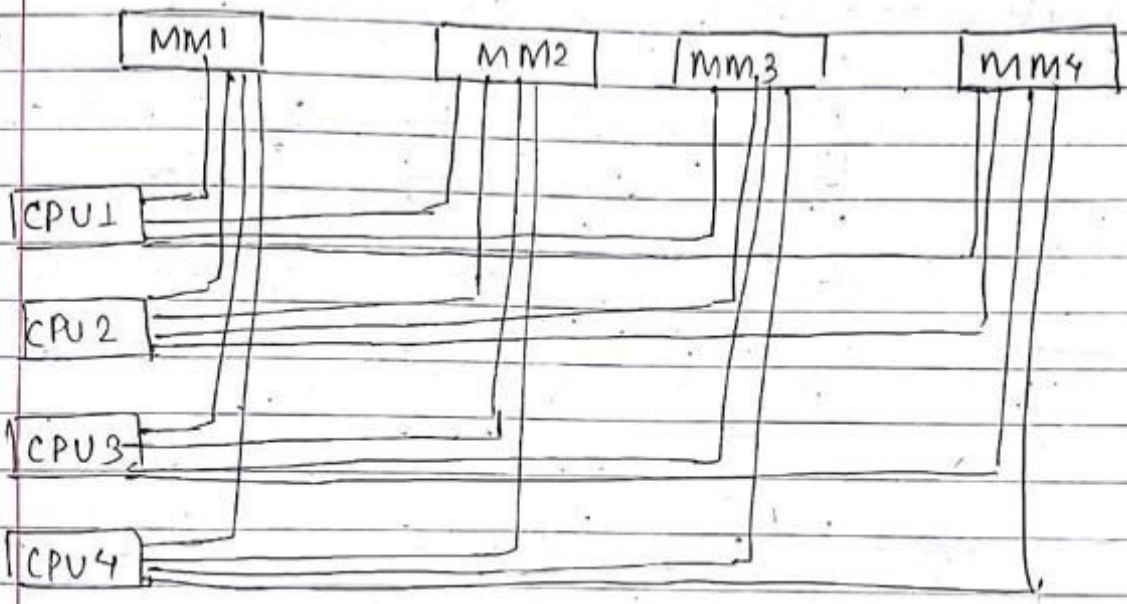
↳ loosely coupled.



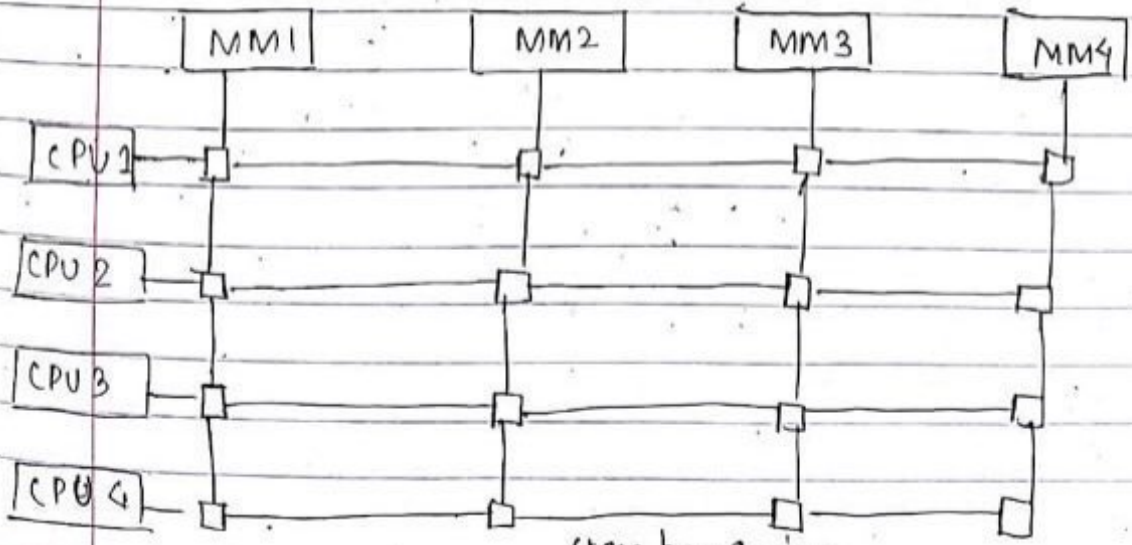
loosely coupled.

8 marks Interconnection Structure

1. Time-shared common bus
2. Multi-port memory
3. Crossbar Switch
4. Multistage switch
5. Hypercube.



Multiport



Cross bar Switch

Herentiating w.r.t. p, we get

$\frac{dy}{dp} = Dy_{n+2}$

4. Multistage Switching N/w

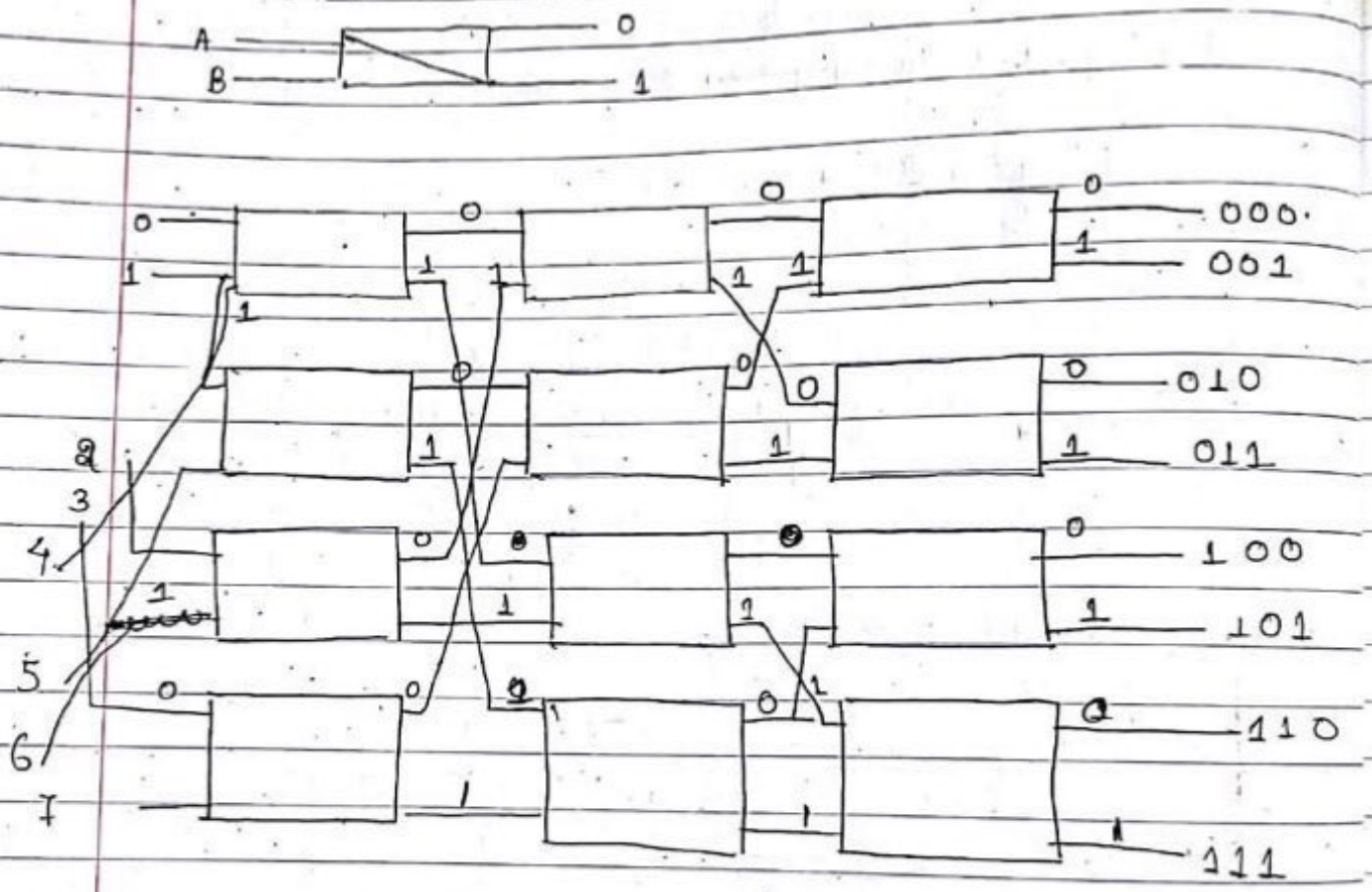
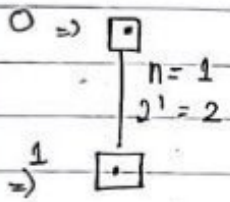


Fig: Omega switching N/w

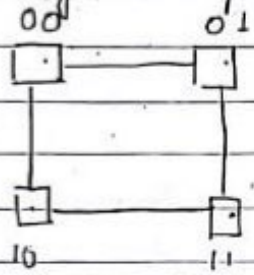
5. Hypercube Inter connection

$N = 2^n$ processor connected in n-dimensional cube

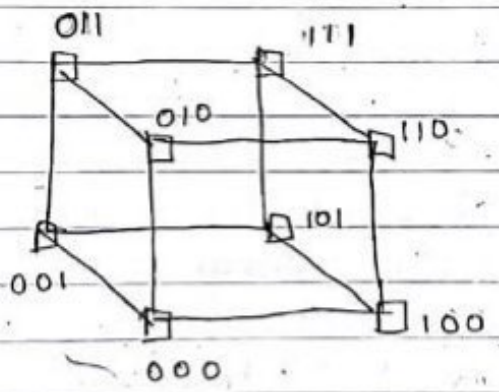
n-bit address can be assigned to processor.



One cube



Two cube



Three-cube

ating w.r.t. p we get

$\frac{dy}{dp} = D y_n + 2r$

8 marks

- Cache Coherence (Write polices)

- Extra hardware
- Snoopy controller

* Vector processing

Used for
→ Long range weather forecasting
→ Petroleum explorations
- Seismic data analysis
- Medical diagonosis
→ AI & expert systems.
→ Image Processing

Operates on array of data.
i.e C(I) = A(I) + B(I) for I = 1 to 100
contains three address format.

Operational code	Base address source 1	Base address 2 source 2	Base address destination	vector Length

C[100]
C[1] = A[1] + B[1]
!
C[100] = A[100] + B[100]

* Memory Interleaving

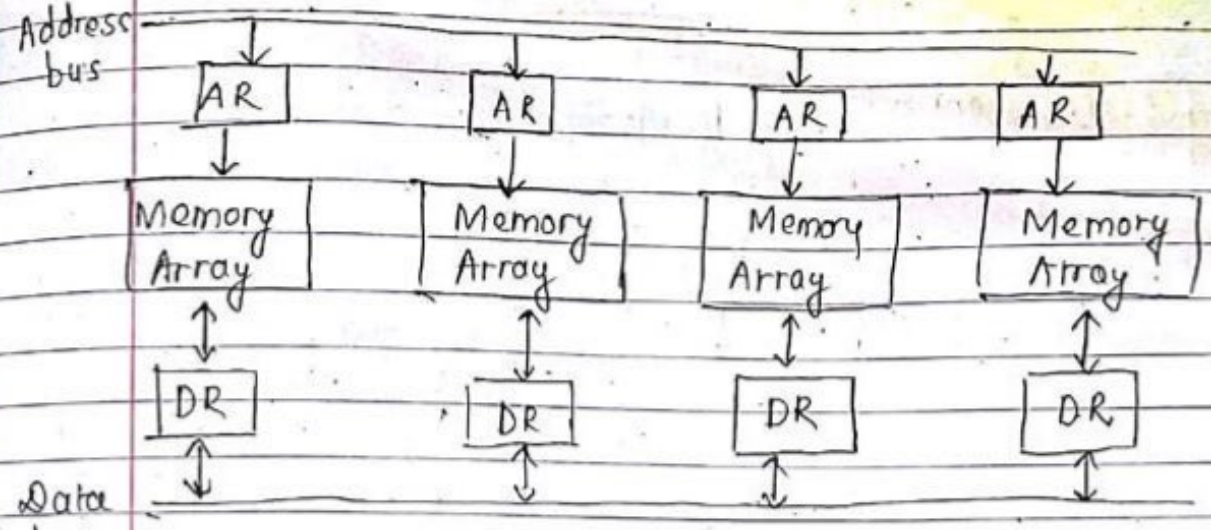
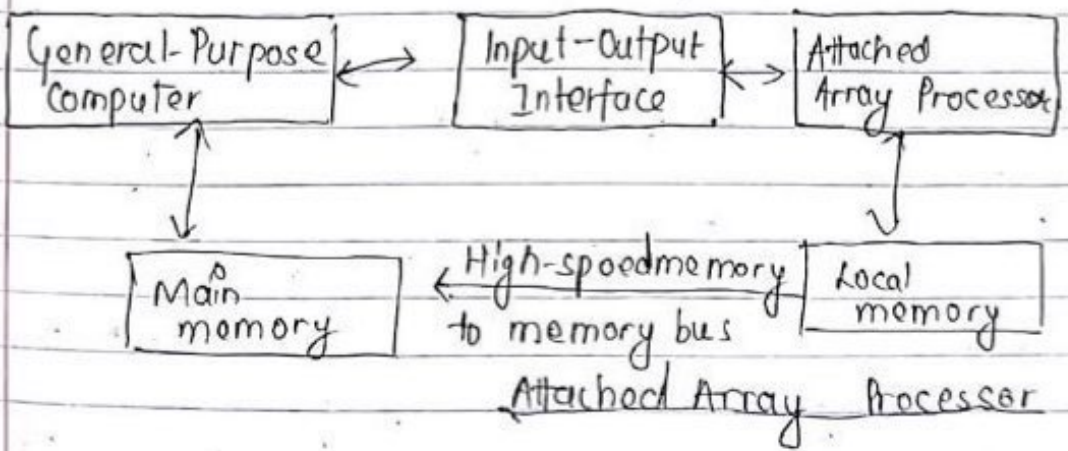


Fig: Multiple module memory organization

* Array Processors

→ performs computation of large arrays of data
→ Two types:
① Attached Array Processor
② SIMD Array



Attached Array Processor

dy/dp = Dy_n + 2 - ...ating w.r.t. p, we get

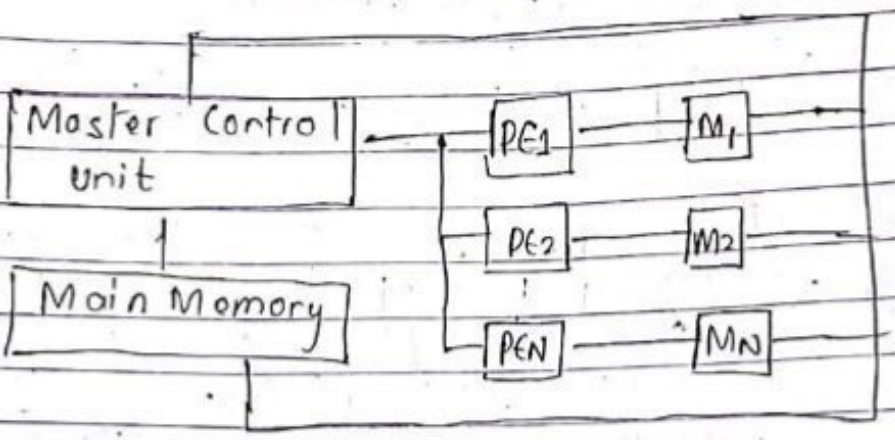


Fig : SIMD processor organization

* Introduction to Multi-thread.

1. Interleaved multi-threading → fine-grained multi-threading
2. Blocked multi-threading → coarse-grained multi-threading
3. Simultaneous multi-threading → multiple processor superscalar
4. Chip multi-processing → multicore processor.

Chapter. 10
Multicore Computers:

- Hardware Performance Issues
  - Increase in Parallelism
    - Pipelining
    - SMT (Simultaneous multithreading)
    - Superscalar
  - Alternative chip organization
  - Power consumption

- Software performance issues.

Power consumption
↳ Logic → more power, more heat
memory → less power, less heat

Short notes { Dual, Quad