UNIT I

COMPUTER ARCHITECTURE

BASIC STRUCTURE OF COMPUTER SYSTEM

1) Functional units (components of a computer system)

*) The computer architecture is composed of three main units:

i) The Central Processing Unit (CPU)
ii) The Memory unit and
iii) The Input/output (I/O unit)

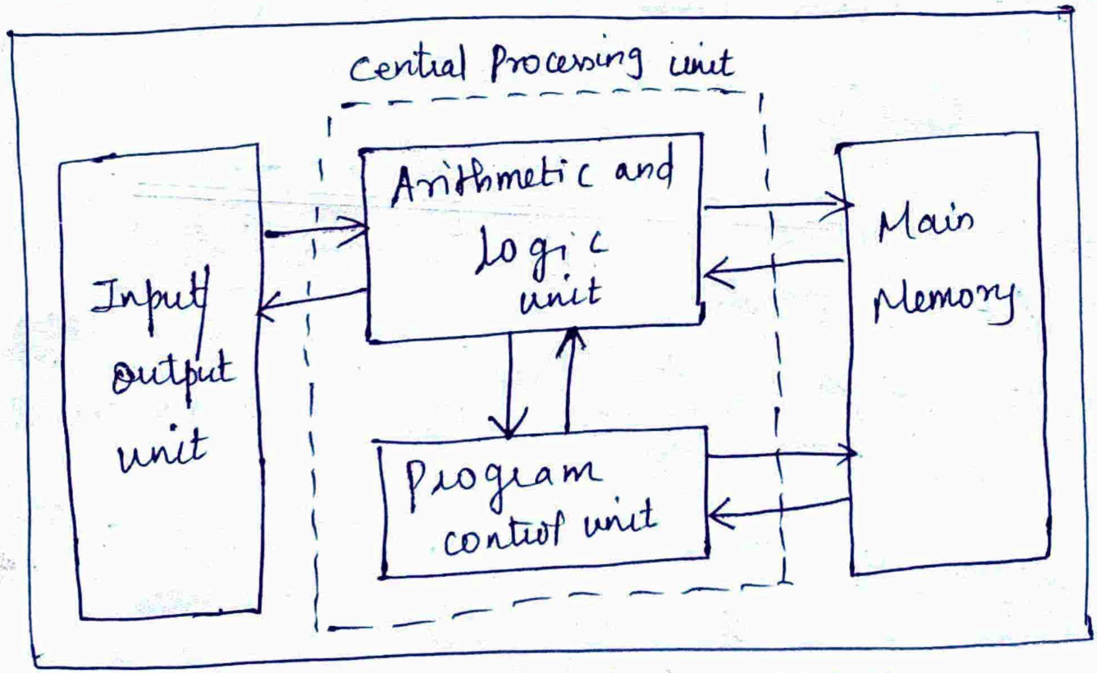


Fig. 1. Computer Architecture

i) Central Processing Unit (CPU):

*) CPU takes data and instructions from the memory unit and makes all sort of calculations based on the instructions given and the type of data provided.

*) The CPU, is the heart of the computing system, it includes three main components:

a) the control unit

b) one or more Arithmetic logic units (ALU) and

c) various registers.

a) control unit:

*) The control unit determines the order in which instructions should be executed and controls the retrieval of the proper operands.

*) It takes care of step by step processing of all operations inside the computer.

b) Arithmetic Logic unit (ALU):

*) ALU performs all arithmetic and logic operations.

*) It performs arithmetic functions like addition, subtraction, multiplication, division and also logic operations like greater than, less than and equal to etc.

c) Registers:

*) The registers are temporary storage locations to quickly store and transfer the data and instructions being used.

*) The registers have faster access time than memory.

ii) Memory Unit

*) The Memory unit is used for storing data and instructions before and after processing. Computer's memory can be classified into 2 types

a) Primary memory and
b) Secondary memory.

a) Primary Memory:

Primary memory can be further classified as → RAM and (Random Access Memory)
→ ROM (Read Only Memory)

→ RAM (Random Access Memory)

*) RAM is a read and write memory. Both read and write operations can be performed in RAM.

*) RAM is a volatile memory having a limited storage capacity

*) As RAM is a volatile memory, its contents can be accessed only when the computer is switched on. The contents of RAM are erased once the computer is switched off.

→ ROM (Read only Memory):

*) ROM is a special type of memory, in which, read operation alone can be performed.

*) ROM is a 'non-volatile' memory and contents are not lost even when the computer is switched off. ROM contains manufacturer's instructions, initial program called the 'boostrap loader' whose function is to start the operation of computer system once the power is turned on.

b) Secondary Memory:

*) Secondary Memory is used to store the content of programs and data permanentaly.

*) Secondary memory is a non-Volatile, permanent, low cost, Cheap memory.

*) Secondary storage devices are of two types < magnetic & optical

*) Magnetic devices include hard disks.
Optical devices are CD, DVD's, pendrive etc.

Characteristics of Secondary Memory:

→ These are magnetic and optical memories

→ It is known as backup memory

→ It is non-volatile memory - Data is permanently stored even if power is switched off.

→ computer can run without Secondary memory

→ slower than Primary memory.

2) Performance:

2.1) Measuring performance based on response time & throughput

*) If you were running a program on two different computers, the faster one is the one that does the job first. For the faster computer the response time is reduced and the throughput is increased.

*) Performance is measured by → response time &
→ throughput.

i) Response time (or) Execution time:

It is the time between the start and completion of a task. The total time required for the computer to complete a task, including disk access, memory access, I/o activities, operating system overhead, CPU execution time and so on.

ii) Throughput (or) Bandwidth:

It is the total amount of work done in a given time. It is the number of tasks completed per unit time.

*) Decreasing the response time → improves the throughpu
& → maximize the performance.

*) The performance and execution time for a computer X is given as

$$\text{performance}_X = \frac{1}{\text{Execution time}_X}$$

*) consider two computers X and Y, if the performance of X is greater than the performance of Y, then we have

$$\text{performance}_X > \text{performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

*) ie, the execution time of computer Y is longer than that of computer X, ie, computer X is faster than computer Y. ~~Thus~~ If X is n times faster than Y, then the execution time on Y is n times longer than it is on X.

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

(6)

INSTRUCTIONS
OPERATIONS OF COMPUTER HARDWARE: (6)

*) The ARM assembly language notation

ADD a, b, c

instructs a computer to add two variables b and c and put their sum in a.

*) Sequence of instruction that adds 4 variables are (a = b+c+d+e)

ADD a, b, c ; (sum of b and c placed in a)
ADD a, a, d ; (sum of b, c and and placed in a)
ADD a, a, e ; (sum of b, c, d and e and placed in a)

it takes 3 instruction to sum the four variables.

; → sharp symbol are comments.

ARM assembly language instruction:

Arithmetic instruction

i) add Instruction → add
Example → $\boxed{ADD\ r_1, r_2, r_3}$
meaning $r_1 = r_2 + r_3$

ii) Subtract
$\boxed{SUB\ r_1, r_2; r_3}$
$r_1 = r_2 - r_3$

Data Transfer Instruction

i) Load register
$\boxed{LDR\ r_1, [r_2, \#20]}$
r_1 = Memory $[r_2+20]$
Load word from memory to register.

ii) store register
$\boxed{STR\ r_1, [r_2, \#20]}$
memory $[r_2+20] = r_1$
Store word from register to memory

iii) Load register halfword

```
LDRH r1, [r2, #20]
```

r_1 = Memory[r_2+20]

load half word from memory to register

iv) Store register halfword

```
STRH r1, [r2, #20]
```

Memory [r_2+20] = r_1

Store half word from register to Memory

v) Load Register Byte

```
LDRB r1, [r2, #20]
```

r_1 = Memory [r_2+ 20]

Load Byte from memory to register

vi) store Register Byte

```
STRB r1, [r2, #20]
```

Memory [r_2+20] = r_1

Store Byte from register to Memory.

vii) Swap

```
SWP r1, [r2, #20]
```

r_1 = Memory [r_2+20],
Memory [r_2+20] = r_1

Swap register and memory.

viii) Mov

$\boxed{\text{MOV } r_1, r_2}$

$r_1 = r_2$

copy value of r_2 into r_1.

Logical Instruction:

i) and

$\boxed{\text{AND } r_1, r_2, r_3.}$

$r_1 = r_2 \& r_3$ Three reg. operands; bit-by-bit ANI

Perform bit-by-bit AND of r_2 and r_3 and save in r_3

ii) or

$\boxed{\text{ORR } r_1, r_2, r_3}$

$r_1 = r_2 | r_3$

Three reg. operands; bit-by-bit OR.

iii) not

$\boxed{\text{MVN } r_1, r_2}$

$r_1 = \sim r_2$

Two reg. operands; bit-by-bit NOT

iv) Logical shift left

LSL $r_1, r_2, \#10$

$\boxed{r_1 = r_2 << 10}$

Shift register r_2 by 10

v) Logical shift right

$\boxed{\text{LSR } r_1, r_2, \#10}$

$r_1 = r_2 >> 10$

shift right r_2 by 10

Conditional Branch Instruction

i) compare

$\boxed{\text{CMP } r_1, r_2}$

cond. flag = $r_1 - r_2$
Compare for conditional branch

ii) Branch on EQ, NE, LT, LE, GT, GE

$\boxed{\text{BEQ } 25}$

Branch if equal

Unconditional Branch instruction

i) Branch (always)

$\boxed{\text{B } 2500}$ Branch

goto pc +8 +10000

ii) Branch and link

$\boxed{\text{BL } 2500}$ for procedure call

r_{14} = pc+4, goto PC+8+10000.

Example: compiling C program into ARM

compile the statement
$f = (g+h) - (i+j)$

The statement is written as the following assembly language instruction

ADD t_0, g, h; → temporary variable to containe g + h
ADD t_1, i, j; → temporary variable t_1 contain i+j
SUB f, t_0, t_1; → f gets $t_0 - t_1$
ie, f = (g+h) − (i+j)

~~'DD t0, g, h ; temporary variable to contains g+h~~

~~ADD t1, i, j ; temporary variable t1 contains i+j~~

~~SUB f, t0, t1 ; f gets t0-t1 ie f = (g+h)-(i+j)~~

OPERANDS OF COMPUTER HARDWARE

7 *) The operands of arithmetic instructions are restricted and stored in a limited number of special locations called registers

*) The size of a register in ARM architecture is 32 bits. Word is a group of 32 bit

Word = 32 bit (or) 4 byte
(or) 2 half-word.

*) The number of registers in ARM architecture is 16 to 32 registers.

*) The reason for limited number of registers are,
i) smaller number of register is faster.
ii) Large number of register may increase clock cycle time

*) We use $r_0, r_1 \dots r_{15}$ to refer to registers 0,1,2 ... 15

compiling C assignment stmt using Registers.

$f = (g+h) - (i+j)$

The variables f, g, h i, j are assigned the registers r_0, r_1, r_2, r_3 & r_4.

```
ADD r5 , r0, r1    ; register r5 contain g+h
ADD r6, r2 , r3    ; register r6 contain i+j
SUB r4 , r5, r6    ; r4 gets r5 - r6.
```

Memory operand

*) The CPU can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence data structures (arrays and structures) are kept in memory.

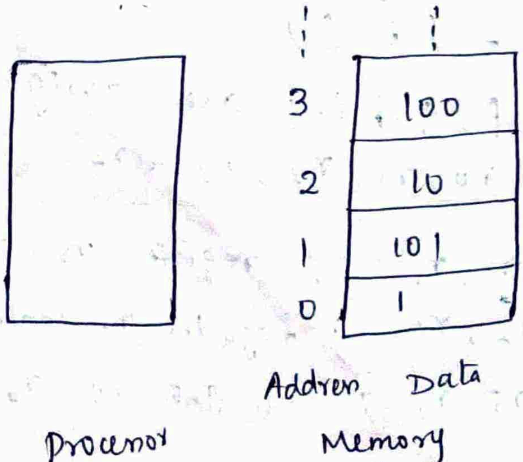
*) Arithmetic operations occurs only on registers in ARM instructions.

*) Instructions that transfer data between memory and registers. Such instructions are called data transfer instructions.

*) To access a word (32 bit) in memory, the instruction must supply memory address.

*) Memory is a large, single dimensional array, with the address acting as index to the array, starting at 0.

*) The figure 1 shows the memory address and the content of memory at those location



*). The address of the 3rd element is 2, and the value of Memory[2] is 10

Fig.1: Memory addresses and contents of memory

*) The data transfer Load instruction is used to copy data from memory to a register.

*) Format of load instruction is

name of operand	register to be loaded ,	constant and register used to access memory

compiling an assignment when a operand is in Memory

$g = h + A[8]$

Assume A is an array.
variable g and h are associated with registers r_1 and r_2. Temporary variable r5 is used. Base address of array is in r3.

→ $\boxed{\text{LDR r5, [r3, \#8]}}$

register r5 gets A[8]

The address of the array element is the sum of the base address of array [r3] and the number to select element 8 [r3+8]

$r5 \leftarrow r3+8$

→ $\boxed{\text{ADD } r_1, r_2, r5}$; $g = h + A[8]$

constant or Immediate operands

~~First Method:~~

*) Program will use a constant in an operation. Many ARM arithmetic instructions have a constant as an operand.

*) For example to add constant 4 to register r3,

LDR r5, [r_1, #Addr constant 4] ;
r5 = constant 4.

ADD r3, r3, r5 ; r3 = r3 + r5
r3 = r3 + 4

r_1 + Addr constant 4 is the memory address of the constant 4.

Alternate Method (2nd method)

If one operand of arithmetic instruction is a constant, it is called immediate constant.

*) To add 4 to register 3,

ADD r3, r3, #4 ; $r_3 = r_3 + 4$

*) The hash symbol(#) means the following number is a constant.

Index Register :

The register in the data transfer inst. holds an index of an array, with the offset used for the starting address of an array.

(8) REPRESENTING INSTRUCTION IN COMPUTER

*) Instructions are kept in computer as a series of high and low electronic signals and is represented as numbers.

*) Each piece of an instruction is considered as an individual number.

i) Arm : Instruction Fields:

cond	F	I	opcode	S	Rn	Rd	operand 2
4 bits	2 bits	1 bit	4 bits	1 bit	4 bit	4 bits	12 bits

i) cond : condition → this field is related to conditional branch instructions

ii) F : Instruction Format → ARM uses different instruction format when needed.

iii) I : Immediate → If I = 0, 2nd operand is register
If I = 1, 2nd operand is 12 bit Immediate (constant)

iv) op code : Basic operation of the instruction. This field denotes the operation and format of an instruction.

v) S : set condition code : This field is related to conditional branch instruction

vi) Rn : The first source operand

vii) Rd : The register destination operand.

viii) operand 2 : The second source operand

ii) ADD instruction format

cond	F	I	op code	S	Rn	Rd	operand 2
4 bit	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits

Translating ARM assembly instructio into Machine instruction

eg ADD r_5, r_1, r_2

The decimal representation is

field 1	2	3	4	5	6	7	8
14	0	0	4	0	1	5	2

$r_1 = 1$ $r_5 = 5$ $r_2 = 2$

*) Each of these segments of an instruction is field

*) The fourth field (4 in this case) tells the ARM computer that this instruction perform addition.

*) The sixth field → gives the number of the register ($r_1 = 1$) that is the first source operand of addition operation

*) Eight field → gives the number of the register ($r_2 = 2$) of the second operand of addition operation

*) Seventh field → gives the number of the register ($r_5 = 5$) that receives the sum.

add → field 4 → 4

1st operand → reg. r_1 → field 6 → 1

2nd operand → reg r_2 → field 8 → 2

sum → reg r_5 → field 7 → 5

*) The instruction can be represented as fields of binary numbers as:

1110	00	0	0100	0	0001	0101	000000000010
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits

This layout of instruction is called instruction format.

*) A form of representation of an instruction composed of fields of binary numbers is known as instruction format.

iii) Load and store Instruction:

The instruction format is

Cond	F	opcode	Rn	Rd	offset 12
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

eg

LDR $r5$, $[r_3, \#32]$

Temporary reg r_5 gets A[8]

14	1	24	3	5	32
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

*) F field is 1 → meaning that this is data transfer instruction

*) opcode field is 24 → this instruction does load word

Rn field → r_3 register

Rd field → r_5 register [destination reg]

offset 12 field → 32 as offset to add to base register

* The below figure shows the numbers used in each field for an instruction

Instruction	format	cond	F	I	op	S	Rn	Rd	Operand 2
ADD	DP	14	0	0	4_{ten}	0	reg	reg	reg
SUB	DP	14	0	0	25_{ten}	0	reg	reg	reg
LDR (load word)	DT	14	1	n.a	24_{ten}	n.a	reg	reg	address
STR (Store word	DT	14	1	n.a	25_{ten}	n.a	reg	reg	address

Fig: Numbers used in each field for an instruction

n.a → not applicable [this field doesnot appear in this format]

op → opcode, reg → register (number b/w 0 – 15)

constant → 12 bit constant

address → 12 bit address

DP → Data processing (DP) instruction format

DT → Data Transfer (DT) instruction format.

Stored program concept

computers are built on 2 key principles

1. Instructions are represented as numbers
2. Programs are stored in memory to be read or written, just as numbers.

This principle is called stored pgm concept

iv) Example for translating assembly instruction into Machine langua

A[30] = h + A[30]

is compiled into

LDR r5, [r3, #120] → [Temp. reg r5 gets A[30]]

ADD r5, r2, r5 → Temp. reg r5 gets h + A[30]

STR r5, [r3, #120] → stores h + A[30] back into A[30]

*) The decimal equivalent for 3 instruction is

cond	F	opcode			Rn	Rd	offset 12
		I	opcode	S			operand 2
14	1	24			3	5	120
14	0	0	4	0	2	5	5
14	1	25			3	5	120

*) The binary equivalent of decimal form is :

cond	F	opcode			Rn	Rd	off set 12
		I	opcode	S			operand 12
1110	1	11000			0011	0101	0000 1111 0000
1110	0	0	100	0	0010	0101	0000 0000 0101
1110	1	11001			0011	0101	0000 1111 0000

(9) (9) **ARM logical operation**

The figure shows logical operations in C, Java and ARM instruction

logical operations	C	Java	ARM instruction
Bit-by-bit AND	&	&	AND
Bit-by-bit OR	|	|	ORD
Bit-by-bit NOT	~	~	MVN
shift left	<<	<<	LSL
shift right	>>	>>>	LSR

i) **AND operation:**

example

AND r5, r1, r2 [reg r5 = reg r1 & reg r2]

register r2 contains

0000 0000 0000 0000 0000 1101 1100 0000_{two}

register r1 contains

0000 0000 0000 0000 0011 1100 0000 0000_{two}

After executing an ARM instruction, reg r5 would be

0000 0000 0000 0000 0000 1100 0000 0000_{two}

ii) **ORR operation:**

It is a bit by bit operation that places 1 in the result if either operand bit is 1.

example

ORR r5, r1, r2 [reg r5 = reg r1 | reg r2

register r2 contains

0000 0000 0000 0000 0000 1101 1100 0000_{two}

register r1 contains

0000 0000 0000 0000 0011 1100 0000 0000_{two}

After executing OR instruction reg r5 would be

0000 0000 0000 0000 0011 1101 1100 0000_{two}

iii) NOT operation :

example :

MVN - Move Not

MVN r_5, r_1 [reg r_5 = ~reg r1]

r_1 = 0000 0000 0000 0000 0011 1100 0000 0000_{two}

$\sim r_1$ = 1111 1111 1111 1111 1100 0011 1111 1111_{two}

∴ The register r_5 value is,

1111 1111 1111 1111 1100 0011 1111 1111_{two}

MOV - Move instruction.

MOV - simply copies one register to another operation without changing it.

Example :

Mov r_6, r_5 [reg r_6 = reg r_5]

r_5 = 0000 0000 0000 0000 0000 0000 0000 0111

∴ The register r_6 value is also

0000 0000 0000 0000 0000 0000 0000 0111

Here the register r_6 has the value of r_5.

iv) SHIFTS operation

Another class of such operations is called shifts. They move all the bits in a word to the left or right.

Example: if register r_0 contains

0000 0000 0000 0000 0000 0000 0000 1001_{two} = 9_{ten}

after Shift left instruction, the new value would be,

0000 0000 0000 0000 0000 0000 1001 0000_{two} = 144_{ten}

The two ARM shift operations are called. 13

(i) Logical Shift Left (LSL)

(ii) Logical Shift Right (LSR)

a) LSL (Logical Shift Left)

example

ADD r_5, r_1, r_2, LSL # 2 $[r_5 = r_1 + (r_2 << 2)]$

b) LSR: (Logical shift Right)

In order to shift 4bits to the right plac

To shift register r_5 right by 4 bits and place the result in r_6, you could do that with a move instruction:

MOV r_6, r_5, LSR #4 $[r_6 = r_5 >> 4]$

The following instructions shifts register r_5 right by the amound in register r3 and Places the result in r_6.

MOV r_6, r_5, LSR r3 $[r6 = r5 >> r3]$

decimal equivalent of 3 instructio

	cond	F	I	opcode	S	Rn	Rd	shift_imm		Shift		Rm
								Rs	0	Shift		Rm
ADD	14	0	0	4	0	2	5	2		0	0	5
MOV	4	0	0	13	0	0	6	4		1	0	5
MOV	14	0	0	13	0	0	6	3	0	1	1	5

11	8	7	6	5	4	3		0
Shift_imm			Shift		0	Rm		
Rs	0		Shift		1	Rm		

(10) Control Operations / Decision Making

1) Conditional branches
2) Loops
3) Comparison Instructions
4) Case switch statements
5) Branch instruction.

(1) Conditional branches / if - then-else

An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

i) BNE stands for branch if not equal.

ii) BEQ stands for branch if equal.

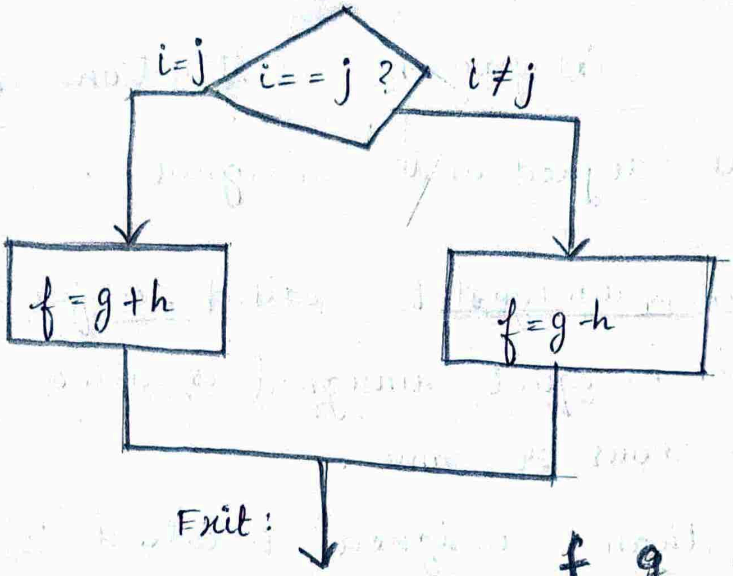
BNE and BEQ are called ∴ Conditional branches.

conditional branches:

(eg) BEQ r_1, r_2, L_1

BNE r_1, r_2, L_1

```
if (i==j)   f=g+h   else   f=g-h
```
← Stmt in



f	g	h	i	j
s_0	s_1	s_2	s_3	s_4

```
BNE  $s3 , $s4 , else
 ADD    $s0 , $s, $s2
else :  SUB  $s0 ,$s1 , $s2
```
→ Instruction

(2) (2) **Loops : / while**

```
While (save [i] == k)
  i=i+1;
```
← Loop in C

```
LDR    r0 , [r12 ,#0]

CMP    r0 , r5

BNE    Exit

ADD     r3 , r3 #1
  :
 Exit
```
← Instruction

* The first step is to load save [i] into a temporary register.
* Before we save we need to have its address.
* We must multiply the index i by 4 due to the byte addressing problem.
* We need to add the label loop to it so that we can branch back to that instruction at the end of the loop.

③ Comparison instructions :

Comparison instructions must deal with signed and unsigned numbers.

→ Less than unsigned is called LO for lower.

→ Less than or equal unsigned is called LS for lower or same.

→ Greater than unsigned is called HI for higher.

→ Greater or equal unsigned is called HS for higher or same.

Example:

r_0 = 1111 1111 1111 1111 1111 1111 1111 1111_{two}

r_1 = 0000 0000 0000 0000 0000 0000 0000 0001_{two}

and the following instruction is executed.

```
    CMP  r0, r1
BLO      L1 ;[unsigned branch]
BLT      L2; [signed branch]
```

The value in r_0 represents -1_{ten} if it is an integer and $4,294,967,295_{ten}$ is an unsigned integer. The value in register r_1 represents 1_{ten} is either case. The branch on lower unsigned instruction (BLO) is not taken to L1, since $4,294,967,295_{ten} > 1_{ten}$. However, the branch on less than instruction (BLT) is taken to L2, since $-1_{ten} > 1_{ten}$

4) Switch / case statement :

Most programming languages have a case or switch statement that allows the programmer to select one of many alternatives depending on a single value.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences called jump address table or jump table.

→ jump address table is also called as jump table. A table of instruction of alternative sequences.

There is a need for holding the address of the current instruction to be executed in a register. The register is called the Program counter.

→ Program counter : The register containing the address of the instruction in the program being executed.

5) Branch instruction :

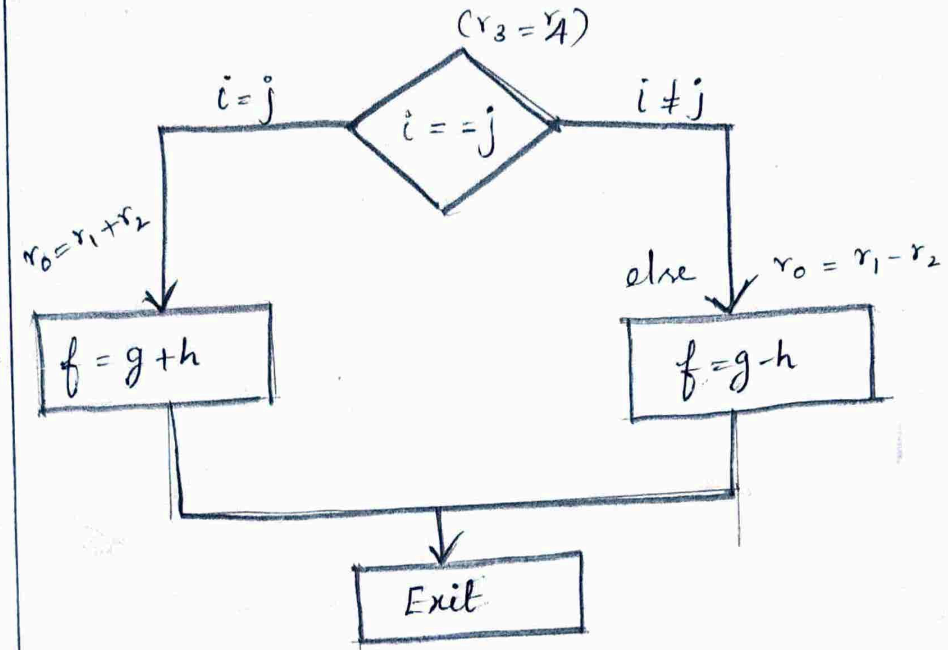
cond	opcode	offset
4 bits	4 bits	24 bits

Value	Meaning	Value	Meaning
0	EQ(Equal)	8.	HI (unsigned Higher)
1	NE (Not Equal)	9	LS (unsigned Lower or same)
2	HS (unsigned higher or same)	10	GE (signed greater than or Equal)
3.	LO (unsigned Lower)	11	LT (signed less than)
4.	MI (Minus , <0)	12	GT (signed greater than)
5.	PL - (PLUS , >=0)	13	LE (signed less than or equal)
6.	VS (overflow set, overflow)	14.	AL (Always)
7	VC (overflow clear, no overflow	15	NV(reserved)

$\boxed{\text{Program counter = Register + Branch address}}$.

→ PC-relative addressing is an regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

```
ADDEQ        r0, r1, r2
SUBNE        r0, r1, r2
```



```
CMP r3, r4
ADDEQ     r0, r1, r2     ;[f=g+h (skipped if  i≠j)
SUBNE     r0, r1, r2     [f=g-h (skipped if  i=j)

CMP r3, r4
BNE      Else            [go to Else  if i≠j]
ADD     r0, r1, r2       [ f = g+h (skipped if i≠j]
 B  exit                 [go to exit]
Else : SUB r0, r1, r2    [f=g-h (skipped if  i= j]
  exit:
```

Conditional execution provides a technique to execute instructions instructions depending on a test without using conditional branch instructions.

UNIT - 2

MIPS Addressing

ADDRESS AND ADDRESSING MODES

1. **Immediate:-**

   * Operand is Constant

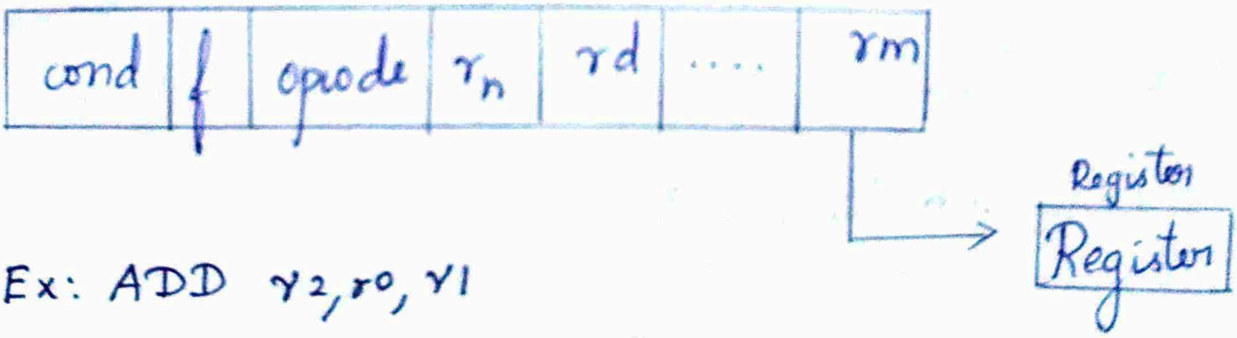
   * Within Instruction Itself

cond	f	opcode	rn	rd	Immediate

   Ex: ADD r_2, r0, #5

2. **Register:-**

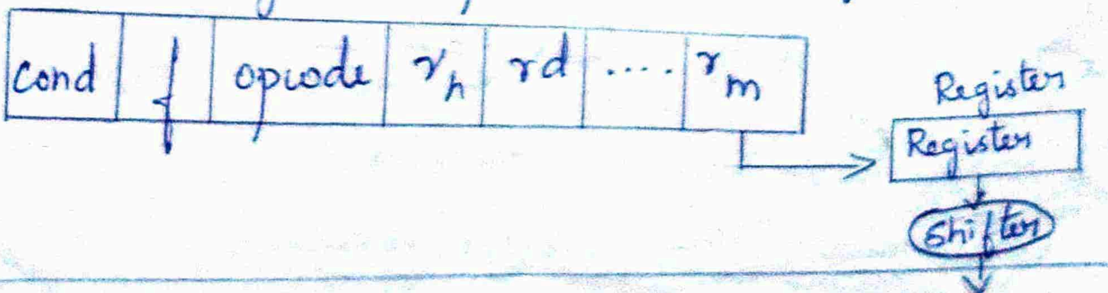
   * Operand is Register



   Ex: ADD r2, r0, r1

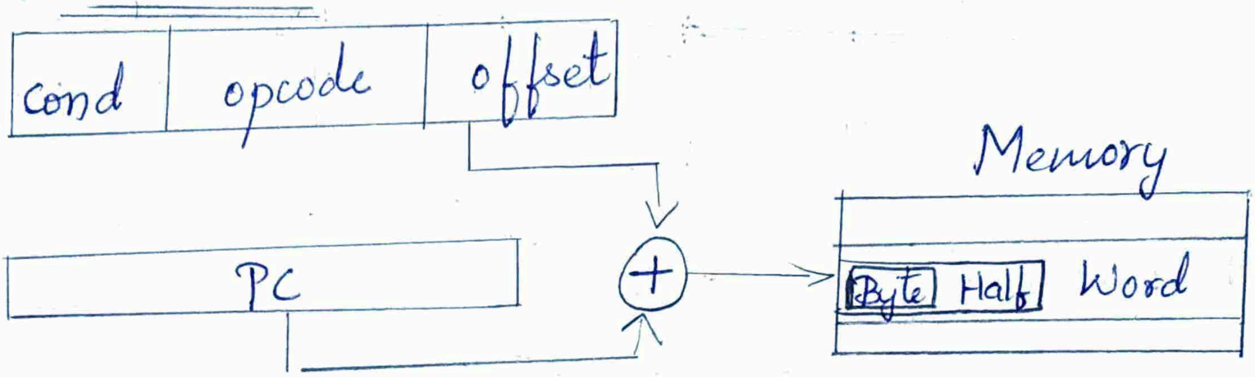
3. **Scaled register:**

   * Register operand is shifted



Ex: ADD r2, r0, r1, LSL#2.

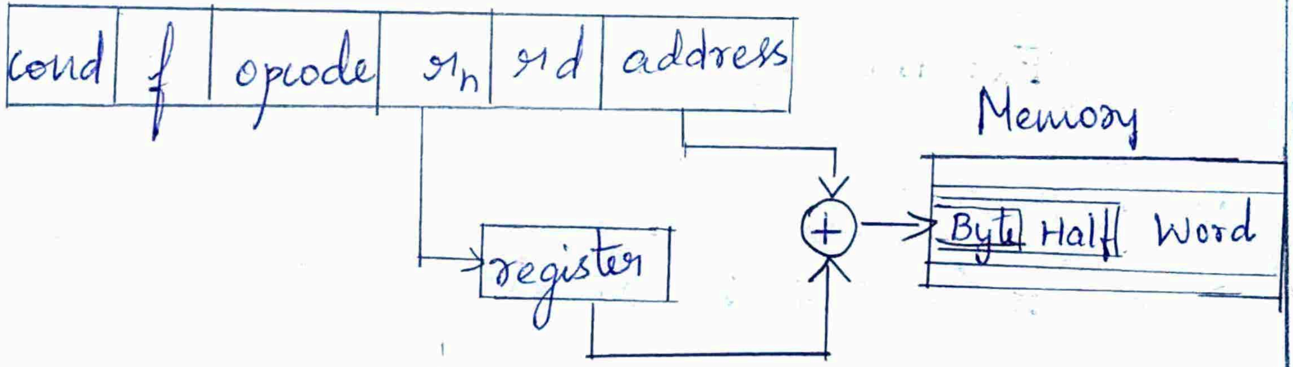
4. PC-relative :-



* Branch address = Sum of PC + constant

Ex: BEQ 1000

5. Immediate offset:



* Constant address is added to base register

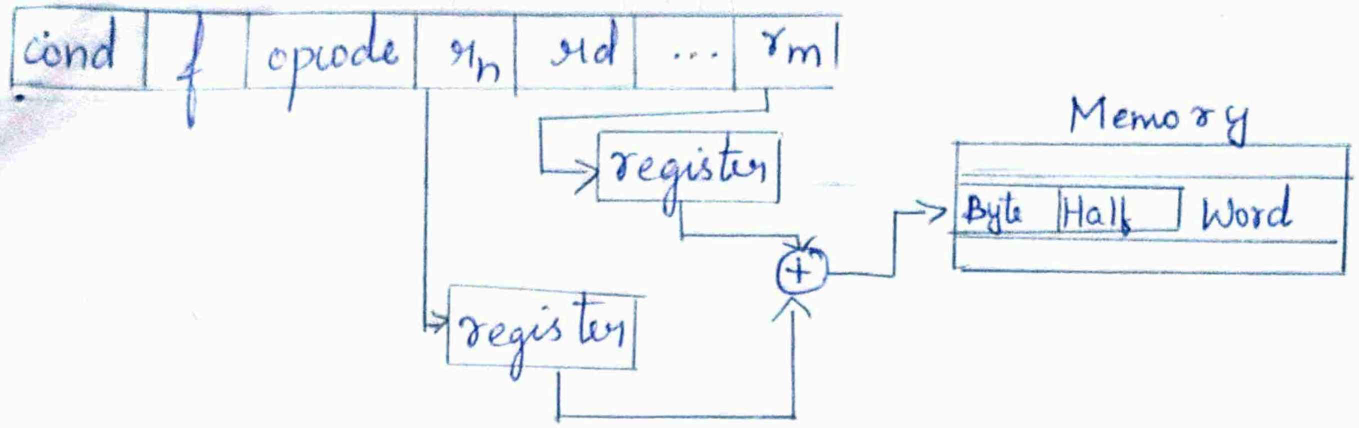
Ex: LDR r2, [r0,#8]

6. Register offset:-

* Another register is added to base register

* Mode can help Index → Array

* Array Index → One register
Base of array → another register

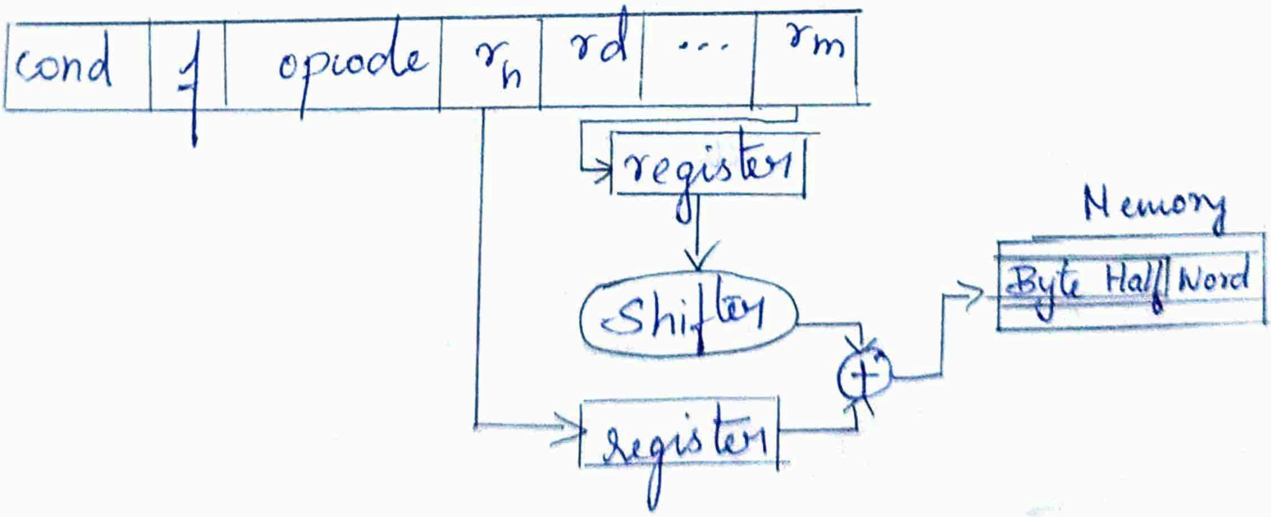


Ex: LDR r2, [r0, r1]

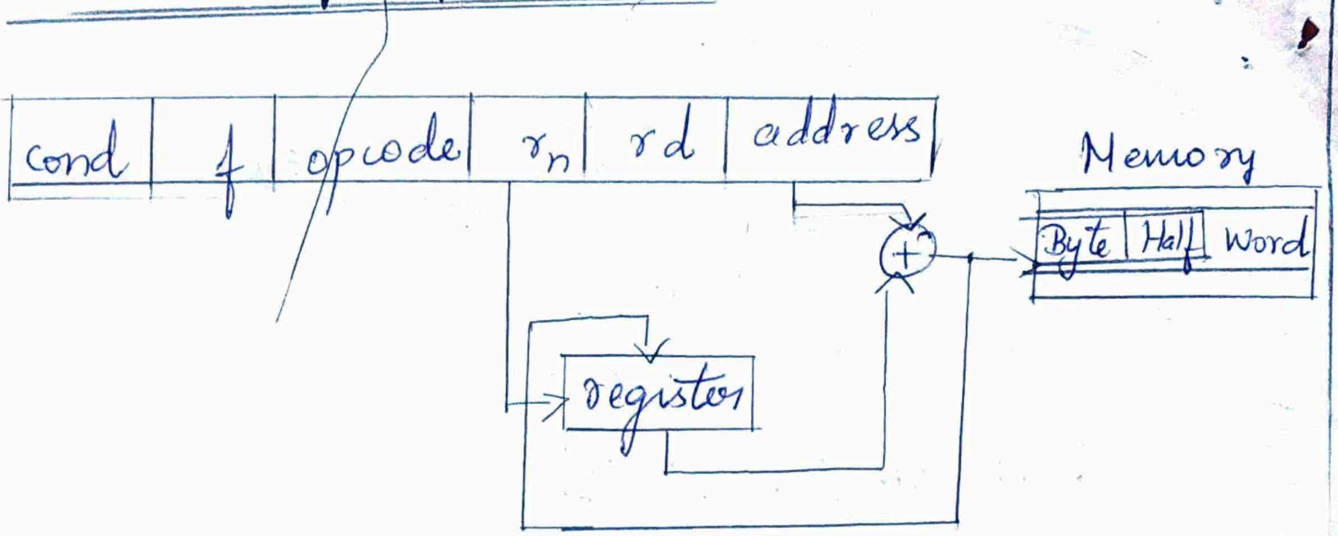
7. Scaled register offset:-

- * Second operand is shifted left or right
- * Register is shifted
- * Useful to turn array index into byte address
- * Shift left by 2 bits

Ex: LDR r2, [r0, r1, LSL#2]



8. Immediate offset pre-indexed: LDR r_2, [r0,#4]



* Update base register with new address

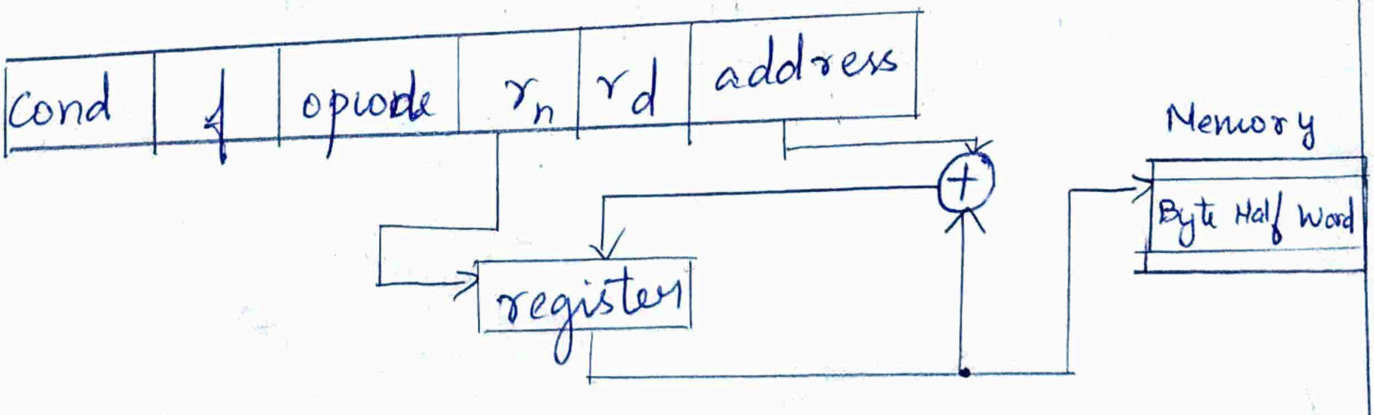
* the destination register changes based on the value fetched from memory

* the base register changes to the address that was used to access memory

* one, where the address is added to the base

* another, where the address is subtracted from the base.

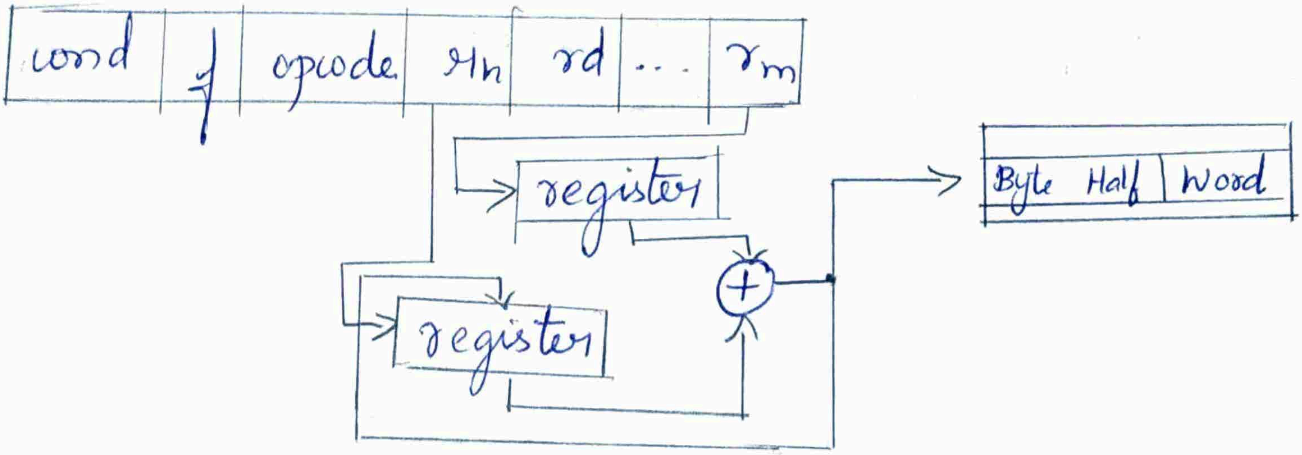
9. Immediate offset post-indexed:



Ex: LDR r_2, [r0], #4

- * Base register is used to access memory first
- * Constant is added or subtracted.
- * Depending on your program set-up, pre-indexed or post-indexed is desired.

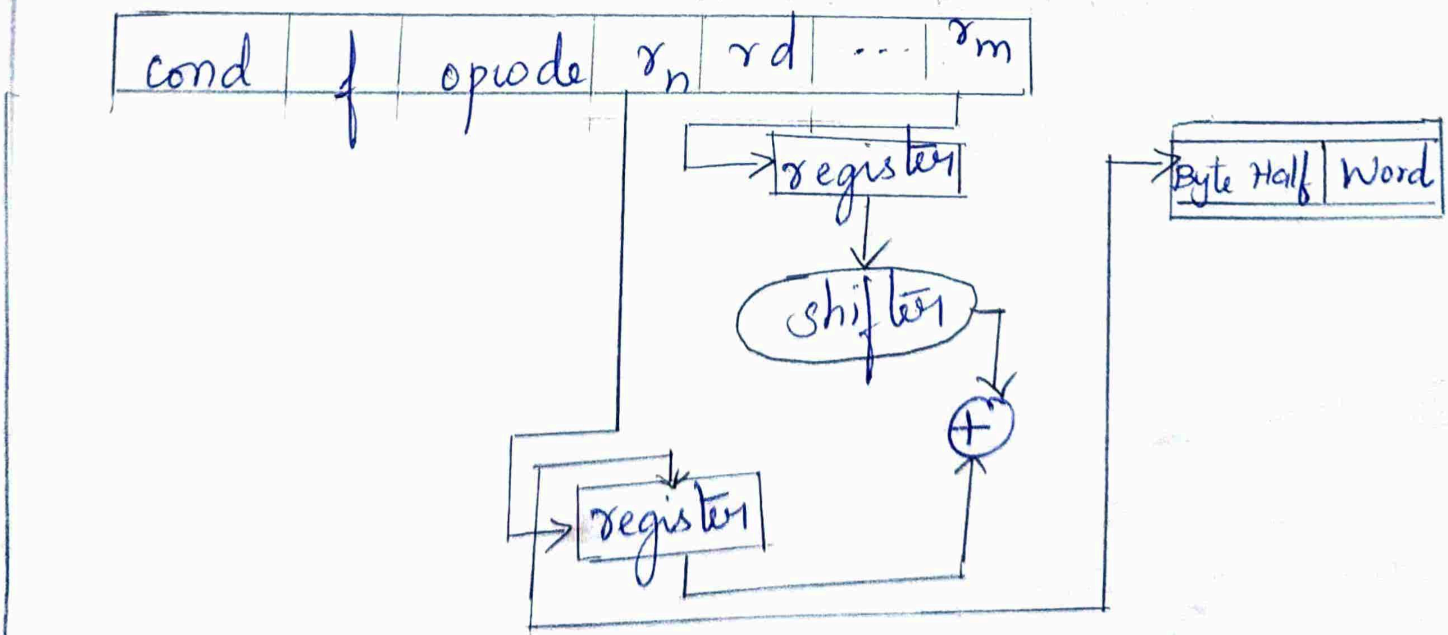
10. Register offset pre-indexed:



- * It is same like Immediate Pre-Indexed
- * Except add or subtract a register instead of a constant

Ex:- LDR r_2, [r0, r1]!

11. Scaled register offset pre-indexed:-

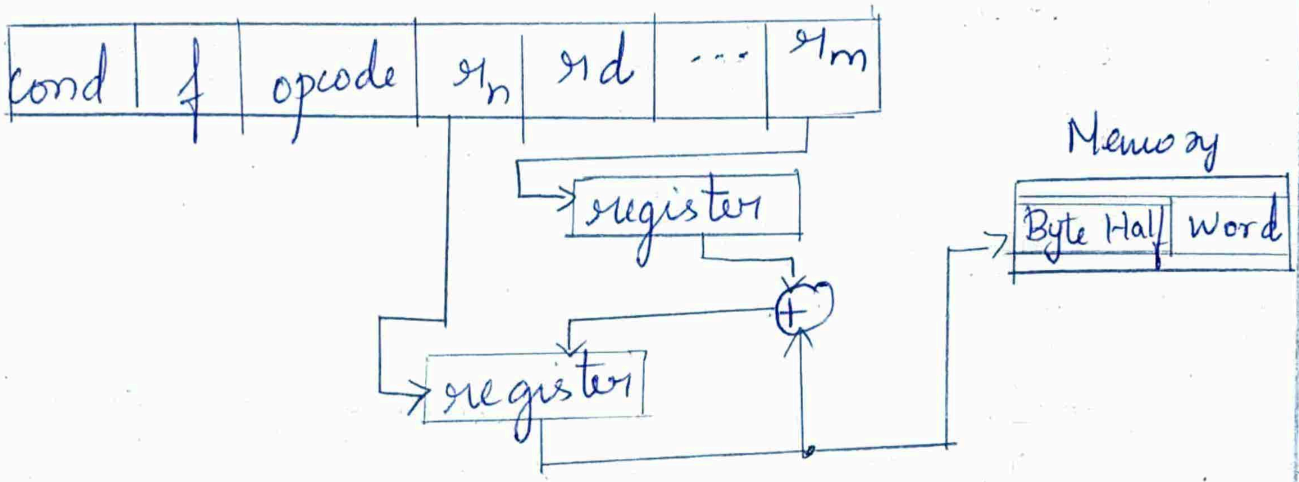


* It is same like Register Pre-Indexed.

* Except shifting the Register before adding or subtracting it.

Ex : LDR r2 , [r0, r1 , LSL #2]!

12. Register offset post-indexed :



* It is same like Immediate Post-Indexed.

* Except adding or subtracting a register instead of constant.

Ex : LDR r2 , [r0] , r1

UNIT – II

Arithmelic for compulers

I ADDITION AND SUBTRACTION :-

* Addition and subtraction is the basic operation performed by the computer.

* First we shall discuss about addition

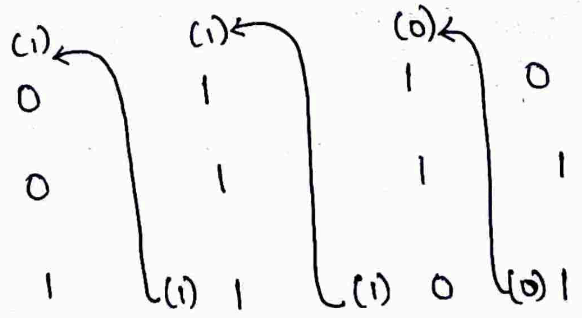
1 ADDITION :-

* In addition, digits are added bit by bit from right to left, if the carry is produced then it is passed to the next digit to the left.

* For example, adding 6_{ten} to 7_{ten} in binary

```
0000  0000  0000  0000  0000  0000  0000  0110 = 6ten
0000  0000  0000  0000  0000  0000  0000  0111 = 7ten
-------------------------------------------------------
0000  0000  0000  0000  0000  0000  0000  1101 = 13te
-------------------------------------------------------
```

Fig. 1 Binary addition, showing Carries from right to left



The above figure shows the clear view of addition operation. Here, the carry is enclosed by parenthesis.

The arrow shows that how carry is

added to the next digit to the left. (2)

* The right most bit 0 is added with 1 and result is 1, carry is 0.

* The carry 0 is added to second digit to the right and perform addition, carry is added to third digit.

* Like this addition is performed till 32 bits.

Overflow in Addition:.

* Overflow occurs when the result of addition cannot be represented with the available hardware.

* Here first operand is a 32 bit word and second operand is a 32 bit word. so the result also must be 32 bit word.

* If the result exceeds more than 32 bit word then it causes overflow.

* In addition operation, adding operands with different signs cannot cause overflow.

* Overflow occurs only by adding two positive numbers or two negative numbers.

* Addition can be performed by using both signed and unsigned binary numbers.

* Positive numbers can be represented as unsigned numbers.

* Sign bit 0 represent positive number and 1 represent negative numbers.

* Before performing addition user must identify whether the number is signed or unsigned.

* If binary number is signed, then the left most bit represent the sign and the rest of the bit represent the number.

2 SUBTRACTION:

* Subtraction is the inverse operation of addition. Subtraction operation cannot be performed directly.

* Using addition we can obtain the result of subtraction process.

Subtracting 6_{ten} from 7_{ten}

```
0000 0000 0000 0000 0000 0000 0000 0111   – 7ten
0000 0000 0000 0000 0000 0000 0000 0110   – 6ten
-----------------------------------------------
0000 0000 0000 0000 0000 0000 0000 0001   – 1ten
-----------------------------------------------
```

or we can do by using two's complement representation of -6

```
0000 0000 0000 0000 0000 0000 0000 0111   = 7ten
1111 1111 1111 1111 1111 1111 1111 1010   = -6ten
-----------------------------------------------
0000 0000 0000 0000 0000 0000 0000 0001   = 1ten
-----------------------------------------------
```

Subtraction using 1's complement:.

Rules:

* Determine the 1's complement of the smaller number

* Add 1's complement to the larger number

* If carry exist add with the result.

Eg. Subtract 11001 from 11100 using 1's complement.

```
   1 1 1 0 0
   0 0 1 1 0    – 1's complement of 11001
  -----------
 ① 0 0 0 1 0
 +         1
  -----------
   0 0 0 1 1
```

Subtraction using 2's complement:.

Rules: 2's complement = 1 + 1's complement

* Determine 2's complement of larger number.

* Add the 2's complement to the larger number.

* If carry exists discard it.

Eg: Subtract 11001 from 11100 using 2's complement

```
 11100
 00111   -> 2's complement of 11001
-------
[1]00011
```

Discard the carry

Result = 00011

Overflow In Subtraction:.

* If we subtract a negative number from a positive number and the result is negative, the overflow will occur.

Eg: Consider $x = +15$, $y = -12$

$x = 15 = 01111$

$y = -12 = 11100$.

```
  0 1 1 1 1   —   +15
  1 1 1 0 0   —   -12
 ___________     _____
  1 0 0 1 1       -3
```

Here result is negative. overflow occurs

If we subtract a positive number from a negative number and the result is positive, then overflow will occur.

Operation	Operand A	Operand B	Result indicating overflow.
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A−B	≥ 0	< 0	< 0
A−B	< 0.	≥ 0	≥ 0

Fig: Overflow conditions for addition and subtraction

3

Multiplication

Multiplying 1000_{ten} by 1001_{ten}

```
Multiplicand     1000
                     ten
Multiplier       1001 ten
               ---------
                 1000
                0000-
               0000--
              1000---
             ---------
              1001000    product
                     ten
```

* The first operand is called multiplicand and second is multiplier. The final result is the product.

* The first observation is that number of digits in product is considerably larger than the number in either the multiplicand or the multiplier.

* Hence like add, multiply must cope with overflow because we frequently want a 32-bit product as a result of multiplying two 32-bit numbers.

* The decimal digit is restricted to 0 and 1. With only two choices, each step of the multiplication is simple:

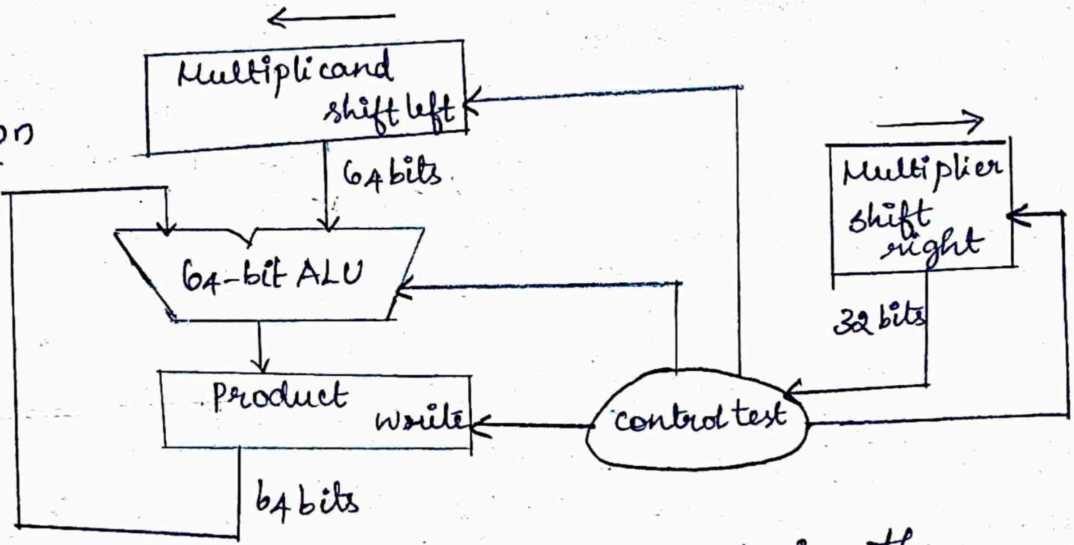
(i) Just place a copy of the multiplicand (1 x multiplicand) in the proper place if the multiplier digit is 1 (or)

(ii) Place 0 (0 x multiplicand) in the proper place if the digit is 0.

* The multiplication of binary numbers must always use '0' and '1'.

3.1) Sequential version of the multiplication algorithm and hardware :

Fig: First version of multiplication Hardware



* Let us assume that the multiplier is in the 32 bit Multipier register and 64 bit product register is initialized to zero.
* From the first step to the last 32 steps, 32 multiplicand would move 32 bits to the left.
* Hence we need 64-bit multiplicand register initialized with 32 - multiplicand in the right half and zero in the left half.
* This register is then shifted 1 step to assign the multiplicand with the sum being accumulated int the 64-product register.

The below diagram gives the three basic steps for each bit.

(i) The least significant bit of the multiplier determines whether the multiplicand is added to the product register.

(ii) Shift left : The effect of moving the intermediate operands to the left

(iii) Shift right : The shift right gives us the next bit of the multiplier.

These theree steps are repeated 32 times to obtain the Product.

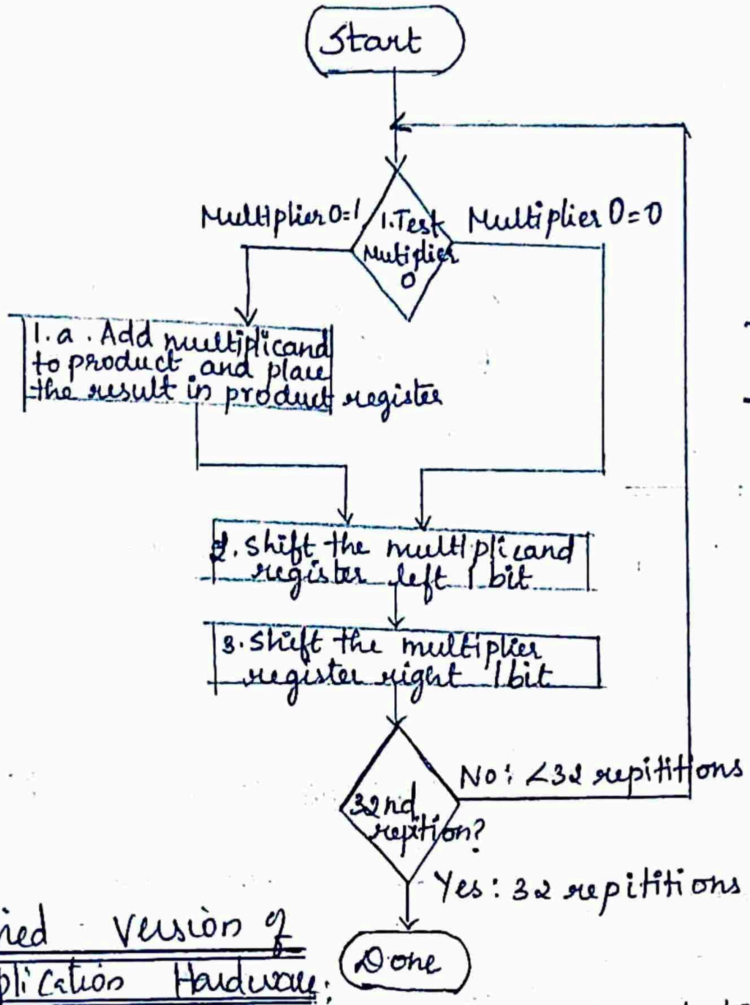
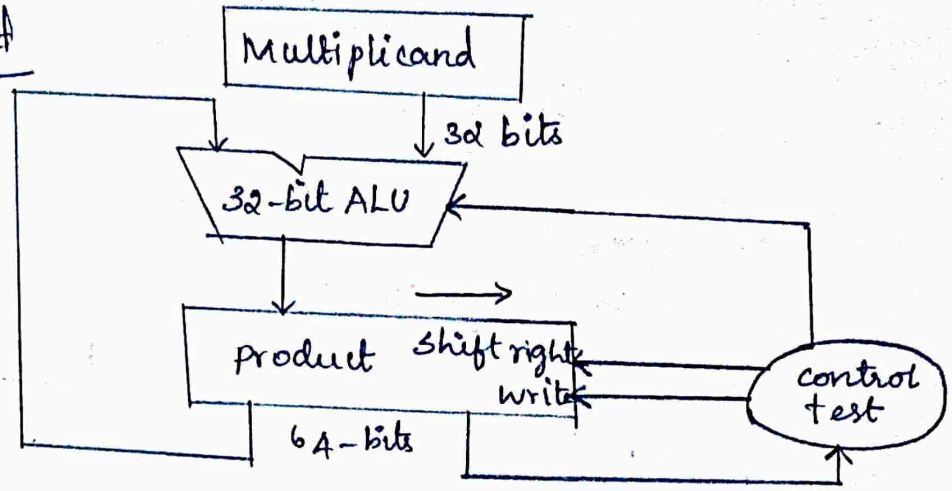

Start
Multiplier 0=1
1. Test Multiplier 0
Multiplier 0=0
1.a. Add multiplicand to product and place the result in product register
2. Shift the multiplicand register left 1 bit
3. Shift the multiplier register right 1bit
32nd repition?
No: <32 repititions
Yes: 32 repititions
Done

Fig: First Multiplication algorithm.

3.2 Refined Version of Multiplication Hardware:

* This algorithm requires almost 100 clock cycles to multiply two 32-bit numbers.
* This algorithm and hardware are easily refined to take 1 clock cycle per step. The speed up comes from performing the operations in parallel.
* The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand.
* The hardware also optimize the adder and register to halve, the width of the adder and register is reduced by noticing whether there is any unused portions of registers and address.

Fig: Refined version of Multiplication Hardware



3.3 Signed Multiplication:

When the algorithm completes, the lower word would have the 32-bit product.

0010 two × 0011 two

Iteration	Step	Multiplier	Multiplicand	Product
0	intial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Product = Prod + Mcand	0011	0000 0010	0000 0010
	2: shift left Multiplicand	0011	0000 0100	0000 0010
	3: shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: shift left multiplicand	0001	0000 1000	0000 0110
	3: shift right multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: shift left multiplicand	0000	0001 0000	0000 0110
	3: shift right multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: shift left multiplicand	0000	0010 0000	0000 0110
	3: shift right multiplier	0000	0010 0000	0000 0110

Faster Multiplication:

3.4

- * Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input in the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
- * Multiply can go ever faster than five add times because of the use of carry save adders.

Multiply in ARM:

- * ARM provides multiply instruction that puts the lower 32 bits of the product into the destination register.
- * Since it doesn't offer the upper 32 bits, there is no difference between signed and unsigned multiplication.

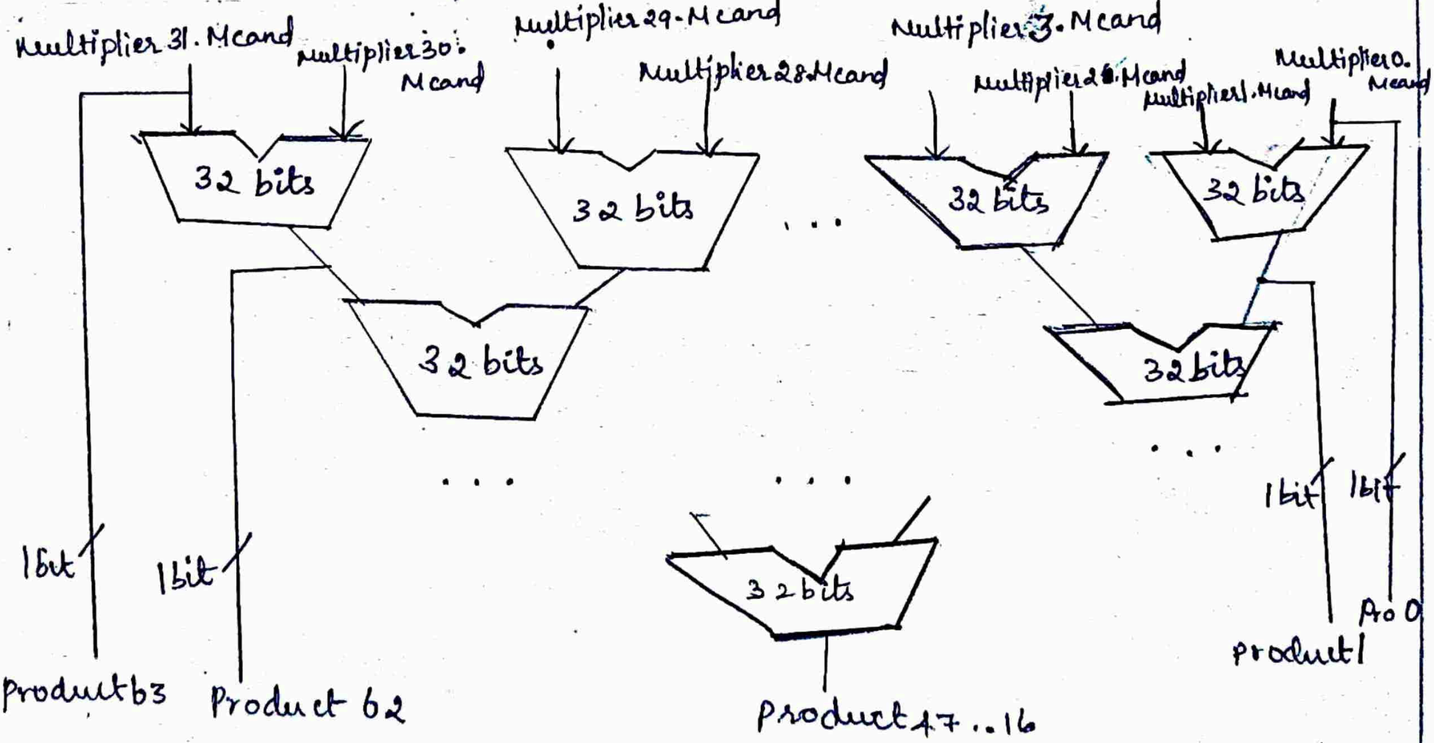


Fig : Fast Multiplication Hardware

4) Booth's Multiplication Algorithm:

*) Booth's algorithm is a multiplication algorithm that multiplies two signed binary number in 2's compliment notation

Algorithm

Step 1 : Load $A = 0$, $Q_{-1} = 0$
M = Multiplicand
Q = Multiplier
n = no. of bits

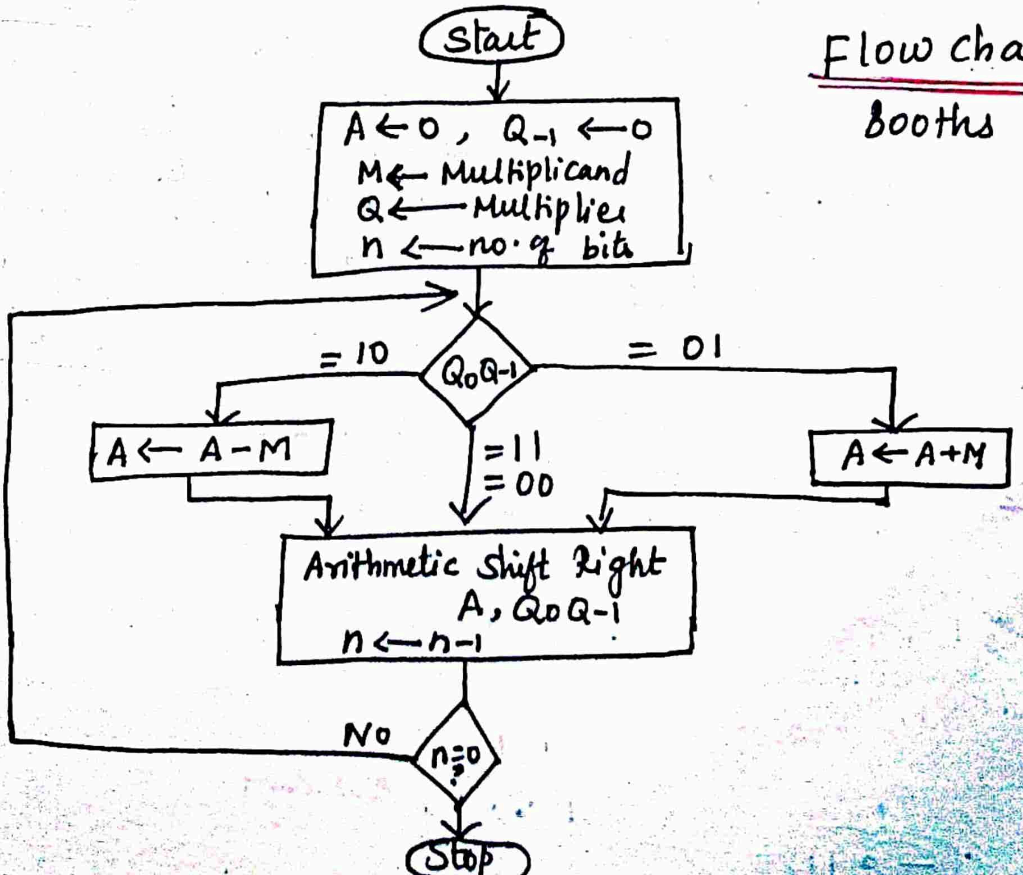
Step 2 : check the status of $Q_0 Q_{-1}$

if $Q_0 Q_{-1} = 10$ perform $A \leftarrow A - M$
if $Q_0 Q_{-1} = 01$ perform $A \leftarrow A + M$

Step 3 : Arithmetic Shift Right $A, Q_0 Q_{-1}$

Step 4 : Decrement n, if not zero, repeat step 2 through 4

Step 5 : Stop



Flow chart of Booths Algorithm

Multiply (−7) and (3) using booth's multiplication

$M = -7$ $\qquad$ $Q = 3$

$= -(0111)$ $\qquad$ $= 0011$

$= 2\text{'s com }(0111)$

1's = 1000
+ 1
2's 1001

$\boxed{M = -7 = 1001}$

$\boxed{-M = 0111}$

Tracing table

M = 1001 $\qquad$ Q = 0011

n	A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
4	0	0	0	0	0	0	1	1	0	Initial
3	0	1	1	1	0	0	1	1	0	$Q_0Q_{-1} = 10$ ∴ $A = A - M$
	↓ 0	0	1	1	1	0	0	1	1	ASR A, Q_0Q_{-1}
2	0	0	0	1	1	1	0	0	1	$Q_0Q_{-1} = 11$ ASR A, Q_0Q_{-1}
1	1	0	1	0	1	1	0	0	1	$Q_0Q_{-1} = 01$ ∴ $A \leftarrow A + M$
	↓ 1	1	0	1	0	1	1	0	0	ASR A, Q_0Q_{-1}
0	↓ 1	1	1	0	1	0	1	1	0	$Q_0Q_{-1} = 00$ ASR A, Q_0Q_{-1}

n = 3

$Q_0Q_{-1} = 10$

so, $A = A - M$

M = 1001

0110
+1
(−M) 0111

0000
0111
A ← 0111

n = 1

$Q_0Q_{-1} = 01$

∴ $A = A + M$

0001
1001
1010

Result = 11101011 = −21

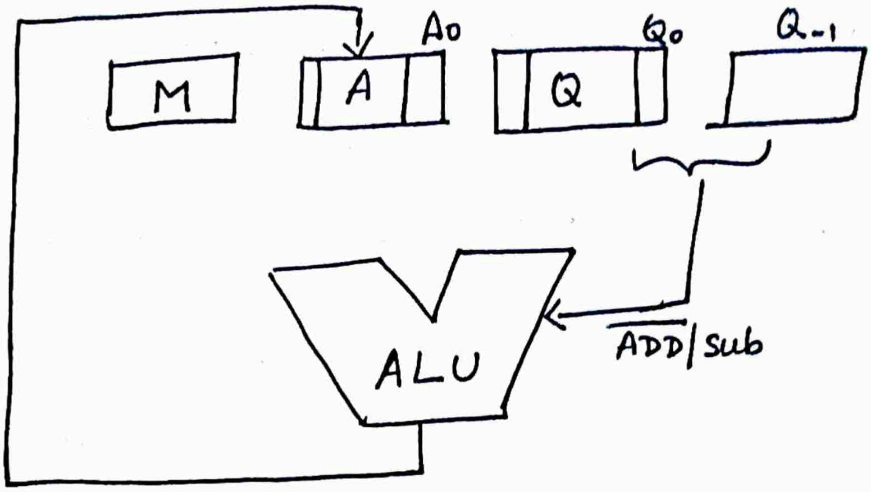
Check ①1101011

sign bit

1's comp = 0010100
+1
0010101 2's com

−21

Hardware structure



*) If the two bits of $\underline{Q_0 Q_{-1}}$ are $\underline{0-0}$ (or) $\overline{1-1}$, then all the bits of A, $Q_0 Q_{-1}$ are shifted to right 1-bit without addition or subtraction.

*) If the two bits of $\underline{Q_0 Q_{-1}}$ are $\underline{0-1}$, then the multiplicand (M) is added ie, $\underline{A = A+M}$, then all the bits of A, $Q_0 Q_{-1}$ are shifted to right.

*) If the two bits of $\underline{Q_0 Q_{-1}}$ are $\underline{1-0}$, then the multiplicand (M) is subtracted ie, $\underline{A = A-M}$, then all the bits of A, $Q_0 Q_{-1}$ are shifted to right.

*) The no. of bits (n) is decremented by 1.
The above process is repeated until n becomes 0.

*) In case of <u>addition</u>,
$\overline{\text{Add}}$ /sub line = 0
In case of <u>subtraction</u>,
$\overline{\text{Add}}$ /sub line = 1

Q_0	Q_{-1}	$\overline{\text{Add}}$/sub	Shif
0	0	x	1
0	1	0	1
1	0	1	1
1	1	x	1

<u>Truth table</u>

(5) Modified Booth's Algorithm / Bit-pair Recording:

*) To speed up the multiplication process in Booth's algorithm a technique called <u>bit-pair recording</u> is used. It is also called <u>modified booth's algorithm</u>.

*) It halves the no. of iterations.

<u>Example:</u>

13 × -6

A = 13 (Multiplicand) 0 1 1 0 1

B = -6 (Multiplier)

= 2's com (6)

= 1 1 0 1 0

+6 → 0 0 1 1 0

1's com → 1 1 0 0 1

+ 1

2's comp → 1 1 0 1 0

<u>Booth's scheme,</u>

0 0	0
1 1	0
1 0	-1
0 1	+1

2's com 6 = 1 1 0 1 0 0

so(-6) → 0 -1 +1 -1

```
          0   1   1   0   1   —Multiplicand (13)
          0  -1  +1  -1   0   → Multiplier (-6)
    ------------------------
   (1)(1)(1)(1)(1) (1)(1)
(0)→ 0  0  0  0  0  0  0  0  0  0
(-1)→1  1  1  1  1  0  0  1  1       → 2's complement of Multiplicand
(+1)→0  0  0  0  1  1  0  1          → Multiplier
(-1)→1  1  1  0  0  1  1             → 2's complement of Multiplicand
(0)→ 0  0  0  0  0  0
    ------------------------------
     1  1  1  0  1  1  0  0  1  0  [-78]
                                  (Product)
sign bit
```

2's complement of Multiplicand:

13 → 0 1 1 0 1

1's → 1 0 0 1 0

+ 1

2's 1 0 0 1 1

Multiplicand selection decision table

Multiplier bit			Multiplicand	Steps	
i+1	i	i−1			
0	0	0	0×M		Shift Right 2bit AQ
0	0	1	+1×M	A = A+M,	Shift right 2bit AQ
0	1	0	+1×M	A = A+M,	Shift right 2bit AQ
0	1	1	+2×M	Shift left M, A = A+M,	shift right AQ, 2bit
1	0	0	−2×M	Shift left M, A = A−M,	Shift right AQ 2 bit
1	0	1	−1×M	A = A−M,	shift right AQ 2bit
1	1	0	−1×M	A = A−M,	shift right AQ 2bit
1	1	1	0×M		shift right AQ 2bit

Tracing Table : 13 × (−6)

Multiplicand 13 (M) → 01101 → 001101

6 → 0110

Multiplier (Q)(−6) → 1010 → 111010

6 → 0110
1's = 1001
+1
1010

Iteration	Steps	A	Q	Q_{-1}
0	Initial	000 000	1110 $\boxed{10}$	$\boxed{0}$
1	100 (−2×M) Shift Left M A = A−M	100110	111010	0
	Shift right 2 bit	111 001	1011 $\boxed{10}$	$\boxed{1}$
2	101 A = A−M	101100	101110	1
	Shift right 2 bit	111011	0010 $\boxed{11}$	$\boxed{1}$
3	111 shift right 2 bit	111110	110010	1

No. of iteration = no. of bits / 2 = 6/2 = 3 iteration

sign bit

Product = 11110110010 (−78)

Step1: (−2×M) 100 → shift left M, A = A−M

M → 001101

shift left (M) → 011010

(−M) = 1's (M) = 100101

+1

(−M) 100110

A = A−M

A → 000000

(−M) → 100110

(A−M) → 100110

Step 2 101 −1×M → A = A−M

1 1 1

A → 111001

−M 110011

(A−M) → 101100

M → 001101

(−M) → 110010

+1

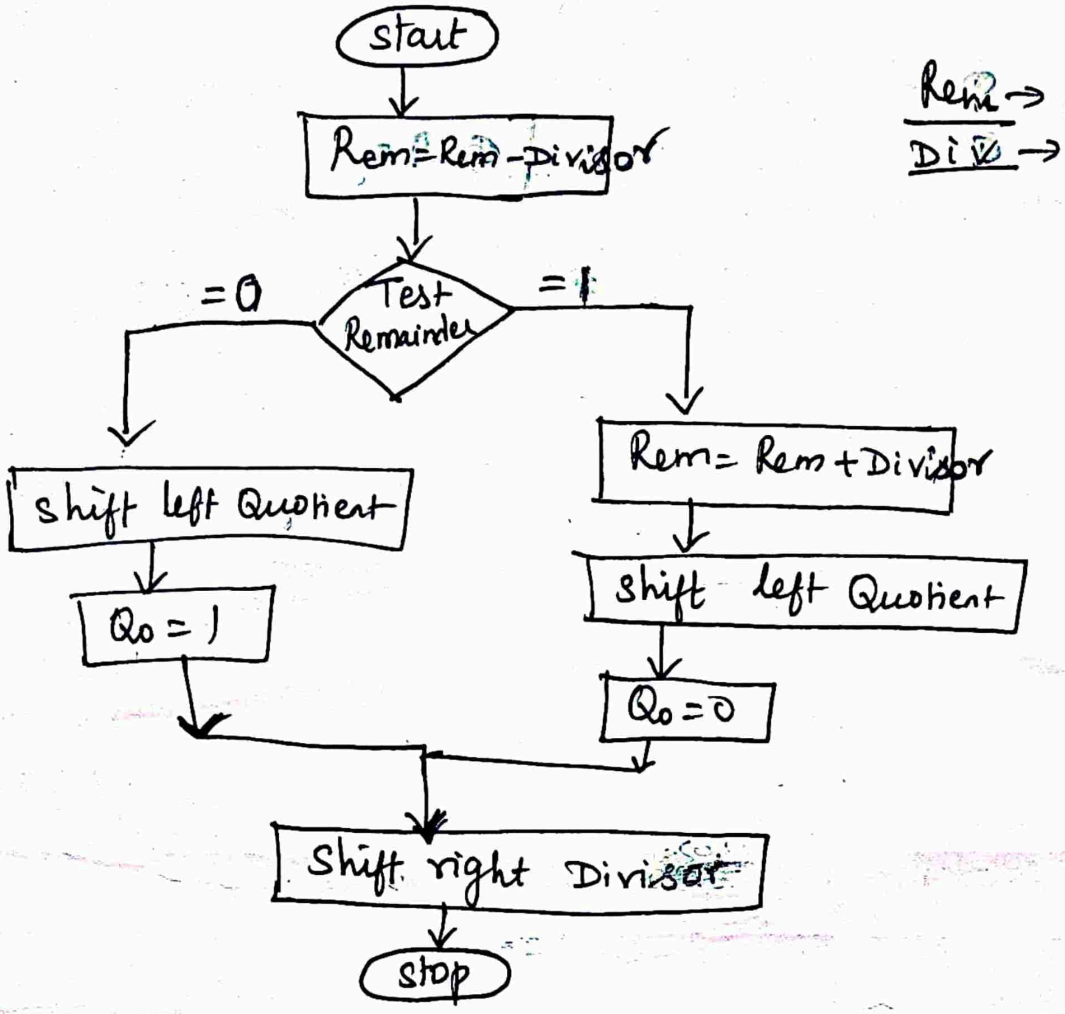
110011

Step 3 111 → 0×M

5) DIVISION

Flow chart for Division:

Rem → Remainder
Div → Divisor



Algorithm: (3 steps)

step 1: 1. Reminder (R) = Reminder − Divisor

step 2: 2. i) If R = 0
Reminder = Reminder + Divisor
Shift left Quotient
$Q_0 = 0$

ii) If R = 1
Shift left Quotient
$Q_0 = 1$

step 3: 3. Shift right Divisor

7 → Divident (Remainder) = 0000 0111

2 → Divisor = 0010 0000

Division Example using the algorithm :

Tracing Table

Iteration	Step	Quotient	(2) Divisor (D)	(Divident) (7) Remainder
0	Initial Values	Q_0 0000	0010 0000	0000 0111
1	1: Rem = Rem − Div	0000	0010 0000	$\boxed{1}$110 0111
	R=1 2b: Rem = Rem + Divisor, Shift left the Quotient, $Q_0 = 0$	0000	0010 0000	0000 0111 .
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem − Div	0000	0001 0000	$\boxed{1}$111 0111
	R=1 2b: Rem = Rem + Div, Shift left the Quotient, $Q_0 = 0$.	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	$\boxed{1}$111 1111
	R=1 2b : Rem = Rem + Div, shift left the Quotient, $Q_0 = 0$.	0000	0000 1000	0000 0111
	3 : Shift Div right	0000	0000 0100	0000 0111
4	1 : Rem = Rem − Div	0000	0000 0100	$\boxed{0}$000 0011
	R=0 2a: Shift left the Quotient , $Q_0 = 1$	0001	0000 0100	0000 0011
	3 : Shift Div right	0001	0000 0010	0000 0011
5	1 : Rem = Rem - Div	0001	0000 0010	$\boxed{0}$000 0001
	R=0 2a : Shift left the Quotient , $Q_0 = 1$	0011	0000 0010	0000 0001
	3 : Shift Div right	0011	0000 0001	0000 0001

step 1

R = R - D

R → 0000 0111

D → 0010 0000

(-D) → 1's → 1101 1111

+ 1

11100000

⇒ R → 0000 01111

+ (-D) → 11 10 0000

$\boxed{1}$110 0111

R = 1

R = R + D

R → 1110 0111

D → 0010 0000

0000 0111

step 2

R = R - D

D → 0001 0000

(-D) 1's 1110 1111

+ 1

11110000

R → 0000 0111

(-D) → 1111 0000

$\boxed{1}$111 0111

R = 1

R = R + D

R → 1111 0111

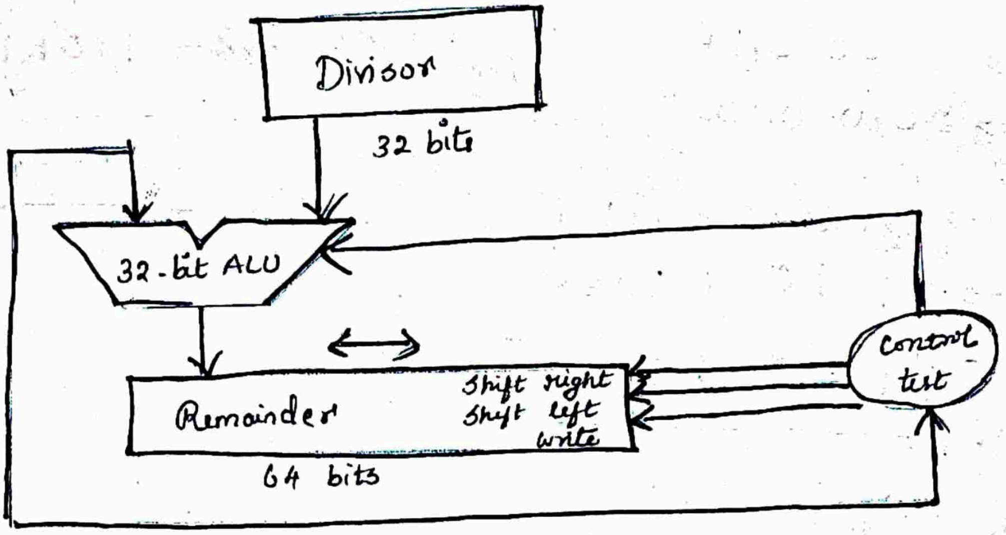
D → 0001 0000

0000 0111

step 3, 4, 5

as above.

AN IMPROVED VERSION OF THE DIVISION HARDWARE.



Faster Division :

*) This technique produce more than one bit of the portion quotient per step

*) The SRT division technique tries to guess several quotient bits per step based on the upper bits of the dividend and remainders.

*) A typical value today is 4 bits.

*) Use 6 bits from the remainder & 4 bits from the divisor.

(7) Non- Restoring Division :

*) To avoid the need for restoring A after an unsuccessful subtraction.

8 by 3

Algorithm :

Step 1: Assign A = 0
signbit of A = 0
→ left shift A, Q
→ A = A−M , set q_0

step 2 : Sign bit of A = 1
→ left shift A, Q
→ A = A+M , set q_0

step 3 : To calculate remainder
if sign bit of A = 1 , Rem = A+M
if sign bit of A = 0 , Rem = A

Q → Dividend
M → Divisor

$\frac{(Q)\ 8}{(M)\ 3}$ → Dividend / Divisor

Q (8) → 1000
M (3) → 00011
−M = 2's(M)

```
     11100
        +1
    ------
-M → 11101
```

Tracing Table

Iteration	Steps	A	Q
0	Initial	$\boxed{0}$0000	1000
1	~~A=0~~ Shift left A Q	00001	000□
	A=A−M → set q_0	$\boxed{1}$1110	000$\boxed{0}$
2	A=1 Shift left A Q	11100	000□
	A=A+M → set q_0	$\boxed{1}$1111	000$\boxed{0}$
3.	A=1 Shift left A Q	11110	000□
	A=A+M → set q_0	$\boxed{0}$0001	000$\boxed{1}$
4.	A=0 Shift left A Q →	00010	001□
	A=A−M → set q_0	$\boxed{1}$1111	001$\boxed{0}$

Quotient

step 1

A = A−M

```
 A → 00001
−M → 11101
     -----
     11110
```

step 2

```
 A → 11100
 M → 00011
     -----
     00011
```

step 3

```
 A → 11110
 M → 00011
     -----
     00001
```

step 4

```
 A → 00010
−M → 11101
     -----
     11111
```

steps

Restore Remainder:

Reminder = A + M

A → 1 1 1 1 1

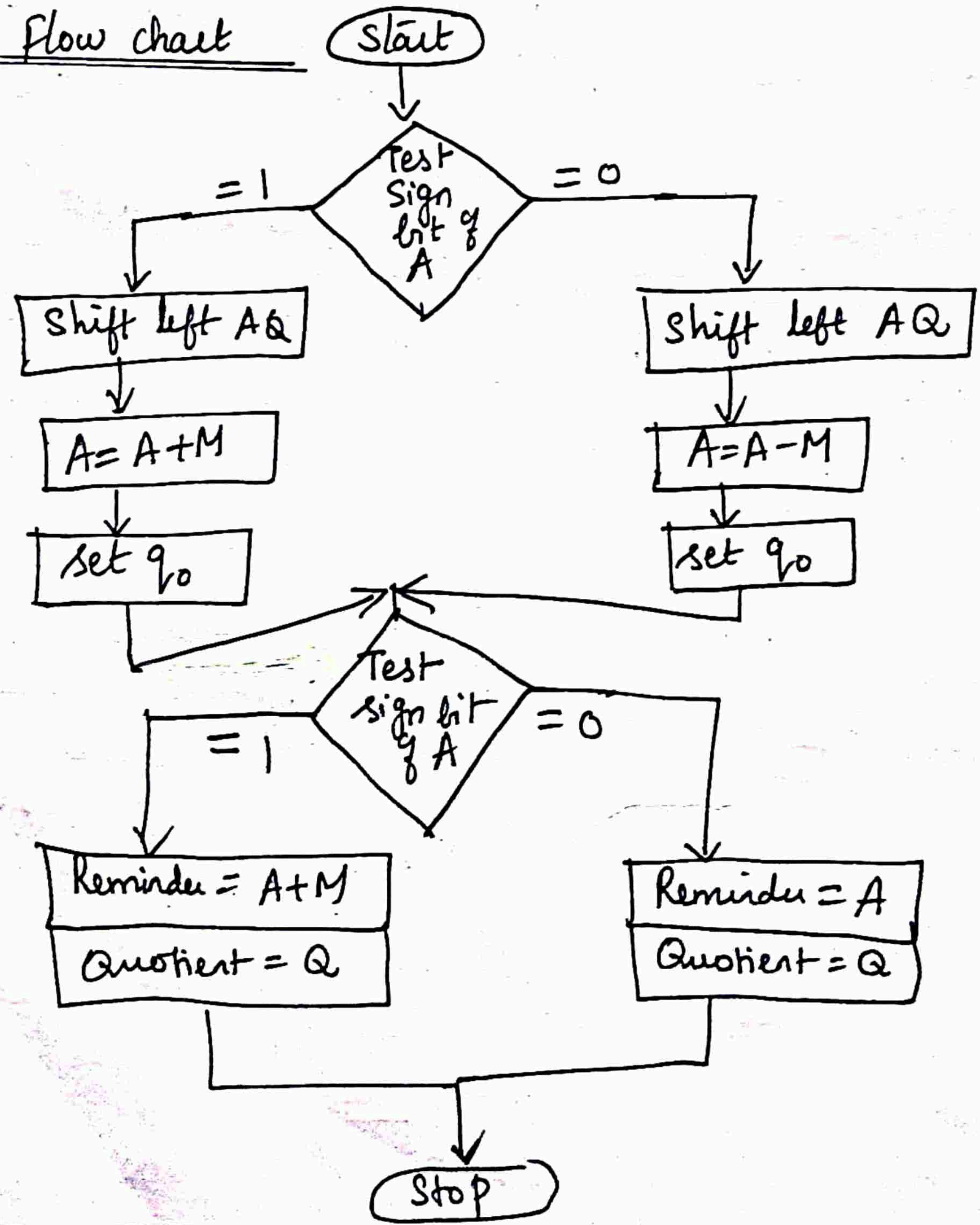
M → 0 0 0 1 1

Remainder 0 0 0 1 0

If A = 1
Remainder = A+M
If A = 0
Remainder = A

Quotient: 0010 (2)   **Remainder** : 00010 (2)

Flow chart



8) Floating point representation:

The numeric value of a finite number is represented by four integer components

i) sign (S) ii) base (b) iii) Significant (or) Fraction (F)

iv) Exponent (E)

Floating point number representation:

$$\boxed{(-1)^S \times F \times 2^E}$$

S → sign
F → fraction
E → Exponent

eg: 6.345×10^{23}

-7.525×10^{-12}

Fraction is represented as

$i_m \, i_{m-1} \, ---- \, i_1 \, i_0 \cdot F_1 \, F_2 \, ---- \, F_{n-1} \, F_n$

where, i → integer part
F → fraction part

Over Flow:-

If the positive exponent becomes too large to fit in the exponent field (ie, the exponent is greater than 38), then it is called overflow.

Under flow:-

If the negative exponent becomes too large to fit into the exponent field (ie, exponent is larger than -38), then it is called underflow.

Normalized scientific notation:-

The number which has a single digit to the left of the decimal number is called normalized scientific notation

eg. 1.0×10^{-9}

Single precision Floating point representation :-

31	30 29 28 27 26 25 24 23	22 - - - - - - 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Sign	Exponent	Fraction
1 bit	8 bit	23 bits

1 bit	8 bit	23 bit
S	E	F

If the floating point value is represented in a single 32-bit word, then it is called single precision.

Sign bit → 1 bit
Exponent → 8 bit
Fraction → 23 bits

Double precision Floating point representation :-

31	30 29 28 27 26 25 24 23 22 21 20	---------- 8 7 6 5 4 3 2 1 0
Sign	Exponent	Fraction
1 bit	11 bit	20 bits

63 62 61 60 59 58 57 56 - - - - - - - - - - - - 37 36 35 34 33 32
Fraction (continued)
32 bits

1 bit	11 bit	52 bit
S	E	F

If the floating point value is represented in two 32-bit (64-bit) word, then it is called double precision.

Sign bit → 1 bit
Exponent → 11 bits
Fraction → 23 bits

For example, 1.0×2^{-1} is represented as

Exponent

$-1 + \underbrace{127}_{\text{bias}} = 126$

$= 01111110$

Bias is a constant added to the exponent to make the range of exponent non-negative

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1.0×2^{1} is represented as

$1 + \underbrace{127}_{\text{bias}} = 128 = 1\,0000000$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IEEE 754 Floating point representation :-

$-0.75 \Rightarrow 0.11 \times 2^{0}$

0.75×2	1.5	1
0.5×2	1.0	1
0×2	–	

Scientific notation : -0.11×2^{0}

normalized scientific notation -1.1×2^{-1}

right -ive (-1)
Left +ive

exponent, $-1 + 127 = 126 = 01111110$

Single precision representation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bit 23 bit

Double precision representation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bit 20 bit

0 0 0 0 0 0 0000000 0000 0000 0000 0000000 0 0 0 0 0

32 bit

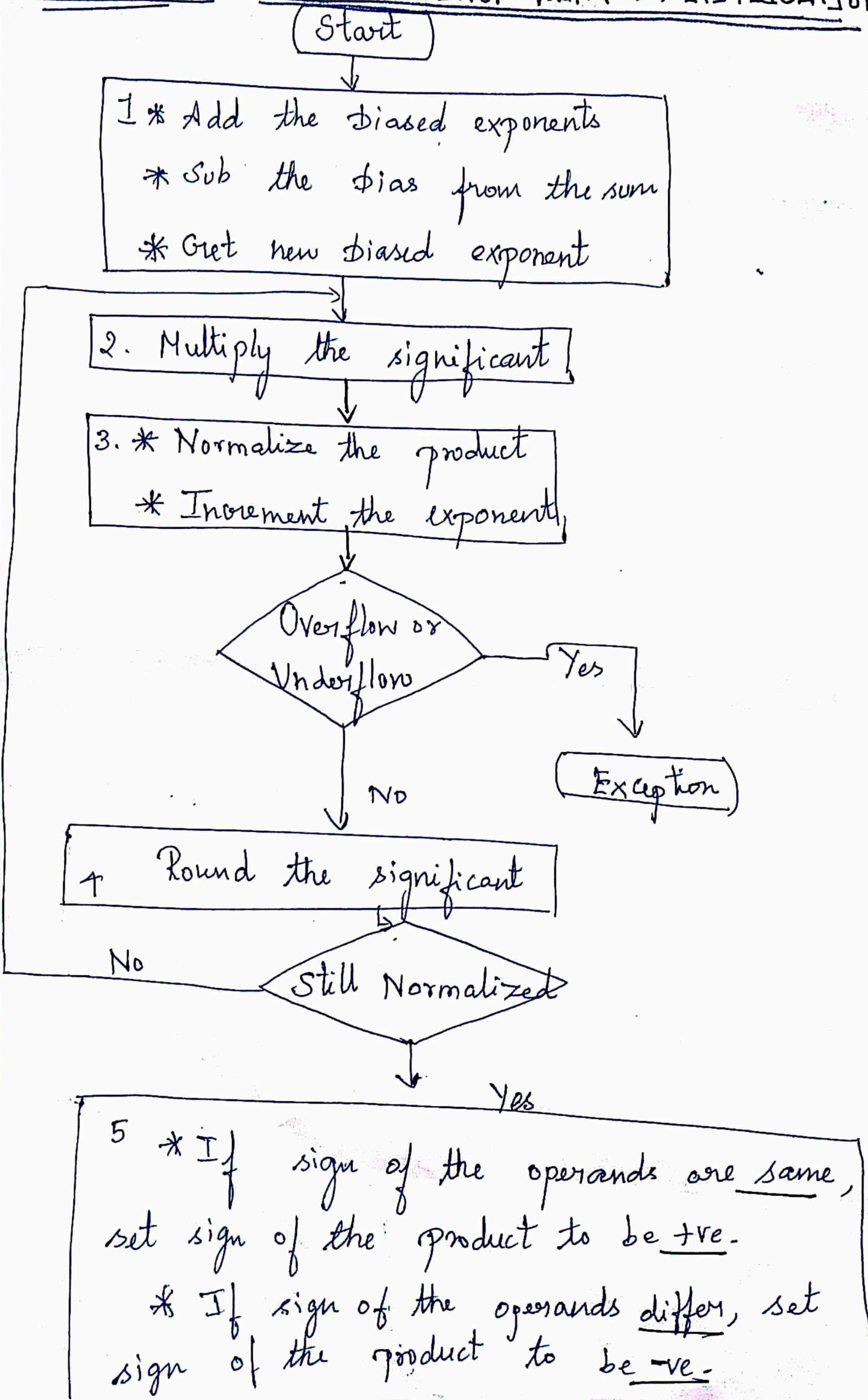
11

FLOATING POINT MULTIPLICATION

* Floating point is a standard way for representing "real" numbers...

* As with addition, the process of multiplication and the process of rounding can produce a non-normalized number. which in turn can result in over/under flow on re-normalization

FLOW CHART FOR FLOATING POINT MULTIPLICATION:-


Start
1 * Add the biased exponents
* Sub the bias from the sum
* Get new biased exponent
2. Multiply the significant
3. * Normalize the product
* Increment the exponent
Overflow or Underflow
Yes
Exception
No
4 Round the significant
No
Still Normalized
Yes
5 * If sign of the operands are same, set sign of the product to be +ve.
* If sign of the operands differ, set sign of the product to be -ve.

↓

Done

i) Example for Binary floating point Multiplication:.

$\boxed{0.5_{ten} \quad -0.4375_{ten}}$

step I:

$0{\cdot}5 = 0{\cdot}100 \times 2^0$

$0{\cdot}4375_{ten} = 0{\cdot}0111 \times 2^0$

0·4375×2	0·8750	0	↓
0·8750×2	1·750	1	
0·750 ×2	1·500	1	
0·500 ×2	1·0	1	

0·5×2	1·0	1	↓

Adding the exponent:-

We calulate the exponent of the product by simply adding the exponents of the operands together.

$$-1 + (-2) = -3$$

(or)

With the biased exponents as well to make sure; we obtain the same result.

$$(-1+127) + (-2+127) - 127 = (-1-2) + (127+127-127)$$

$$= \boxed{-3} + 127 = 124$$

step II :- Multiply the significands:-

Three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significand.

```
        1.000
        1.110
      -------
         0000
        1000
       1000
      1000
  -----------
      1110000two
```

$= 1.110000_{two} \times 2^{-3}$

step III: Scientific normalisation:-

After the multiplication, the product can be shifted right one digit to put in normalized form, adding 1 to the exponent.

$$1.110000_{two} \times 2^{-3}$$

step IV :- Round off the product:

Significand is only four digits long, so we must round the numbers.

step V :-

Since sign of the operands differ, the product is -ive ie, (0.5 & -0.4375)

$$\therefore \boxed{-1.110_{two} \times 2^{-3}}.$$

ii) Decimal Floating point Multiplication:-

$\boxed{\text{Ex}: (1.110_{10} \times 10^{10}) \times (9.200 \times 10^{-5})}$

step I: Adding the exponent :-

We calculate the exponent of the product by simply adding the exponents of the operands together

$-\cancel{1}+(-\cancel{2})=-3$

$10 + (-5) = 5$

step II :- Multiply the significant :-

Three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significand

$= 10.21200 \times 10^5$

step III :- Scientific normalization

After the multiplication, the product can be shifted right one digit to put in normalized form, adding 1 to the exponent

1.021200×10^6

step IV :- Round off the product :

Significand is only four digits long, so we must round the number.

1.021×10^6

step V :-

Since the sign of the operands same, the product is +ve.

$\therefore \boxed{1.021 \times 10^6}$

Subword parallelism:

*) Many graphics systems originally used 8 bi to represent each of the three primary colors plus 8 bits for a location of a pixel.

*) By partitioning the carry chains within a 128 bit adder, a processor could use parallelism to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16- bit operands, four 32 bit operands or two 64-bit operands.

*) parallelism is of two types
i) sub word parallelism
ii) data level parallelism

*) With the appropriate subword boundaries this technique results in parallel processing of subwords.

*) This is a form of SIMD (single instruction Multiple Data processing

*) Graphics and audio applications can take advantage of performing simultaneous operations on short vectors.

*) Architects recognized that many graphics and audio applications would perform

the same operation on vectors of this data.

Example:

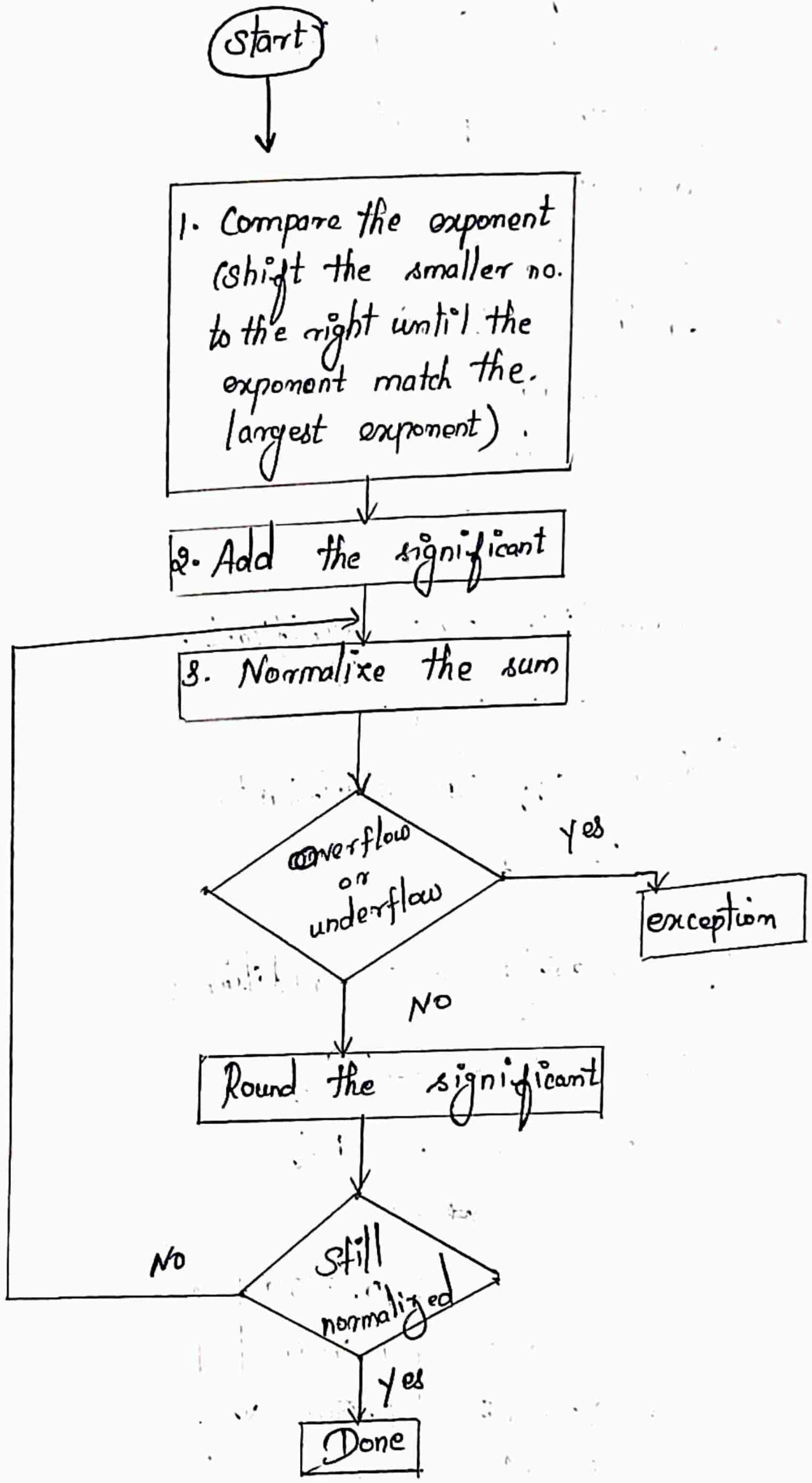
*) If word size is 64 bits and subwords sizes are 8, 16 and 32 bits. Hence an instruction operates on eight 8 bit subwords, four 16 bit subwords, two 32 bit subwords or one 64 bit subword in parallel.

128 bit adder:

→ sixteen 8 bit adds
→ Eight 16 bit adds
→ Four 32 bit adds.

9/5/22

Floating point addition / Subtraction



Algorithm (Addition / Subtraction):

Step 1: Compare the exponent. (shift the smallest number to match the largest no.)

Step 2: Add the significant.

Step 3: Normalize the sum

Step 4: Round the significant.

Decimal floating pt Addition

9.999×10^{1} 1.610×10^{-1}

① compare the exponent.

1.610×10^{-1}

$= 0.01610 \times 10^{1}$

② Add the significant

$$\begin{array}{r} 9.999 \\ 0.01610 \\ \hline 10.01510 \times 10^{1} \\ \hline \end{array}$$

③ normalize the significant.

1.00150×10^{0}

④ Round off the significant.

1.002×10^{2}.

Binary floating point addition:

0.5_{ten} & -0.4375_{ten} :

$0.5 \times 2 = 1.0 \mid 1 \downarrow$

$0.5 = 0.1 \times 2^{0}$.

0.4375×2	0.8450	0
0.8750×2	1.750	1
0.750×2	1.500	1
0.500×2	1.0	1

$\boxed{0.0111 \times 2^{0}}$

① Compare the exponent

$0.1 \times 2^{0} \Rightarrow 1.0 \times 2^{-1}$

$-0.0111 \times 2^{0} \Rightarrow 1.11 \times 2^{-2} \Rightarrow 0.111 \times 2^{-1}$

② Add the significant.

$1.0 + (-0.111) \times 2^{-1}$

$$
\begin{array}{r}
1.000 \\
+\;1.001 \\
\hline
\cancel{1}0.001 \times 2^{-1}
\end{array}
\qquad\qquad
\begin{array}{r}
1.000 \\
+\;1 \\
\hline
1.001 \\
\hline
\end{array}
$$

2's comp

(3). normalize the significant.

$\cancel{1}0.\cancel{1}001 \times 2^{-1} \Rightarrow 1.00000 \times 2^{-4}$.

(4) Round off the significant.

1.000×2^{-4}

Decimal floating point subtraction :

$9.999 \times 10^{1} \qquad - 1.610 \times 10^{-1}$

step1 : compare the exponent.

$-1.610 \times 10^{-1} = 0.01610 \times 10^{1}$

step2 : Subtract the significant.

$$
\begin{array}{r}
9.99900 \times 10^{1} \\
0.01610 \times 10^{1} \\
\hline
9.98290 \times 10^{1} \\
\hline
\end{array}
$$

step3 : normalize the significant

9.98290×10^{1}

step4 : Round off the significant.

9.983×10^{1}.

Binary floating point Subtraction:

$(0.5_{ten}) - (0.4375_{ten})$

$0.5 \times 2 = 1.0 \mid 1 \downarrow$

$\boxed{0.5 = 0.1 \times 2^{0}}$

0.4375×2	0.8750	0	
0.8750×2	1.750	1	$\downarrow$
0.750×2	1.500	1	$\boxed{0.0111 \times 2^{0}}$
0.500×2	1.0	1	

Floating point multiplication:

Single precision - 32bit

Double precision - 64 bit.

* Overflow (+)
* Underflow (-)

Single precision:

1bit	8bit	23 bit
S	E	F

Double precision:

1bit	11bit	52bit
S	E	F

Floating point multiplication:

① **Binary floating point:**

X. 0.5_{ten} & -0.4375_{ten}

Step1: Adding the exponent.

$0.5 = 0.1 \times 2^0$

$0.4375 \rightarrow 0.011 \times 2^0$

normalize

1.0×2^{-1}

1.11×2^{-2}

$(-1) + (-2) = -3$

Step2. Multiply the significant

$$\begin{array}{r} 1.0 \\ \times\ 1.11 \\ \hline 10 \\ 10 \\ 10 \\ \hline 1.110 \\ \hline \end{array}$$

1.110×2^{-3}

$$\begin{array}{r} 2 \underline{|5} \\ 2 \underline{|2} - 1 \\ 1 - 0 \end{array}$$

$0.5 \times 2 \,|\, 1.0 \,|\, 1$

0.4375×2	0.8750	0
0.8750×2	1.750	1
$0.~~75~~0 \times 2$	1.500	1
0.500×2	1.00	1

$$\begin{array}{r} 1.0 \\ 1.11 \\ \hline 10 \\ 10 \\ 100 \\ \hline 1.1\ 1\ 0 \\ \hline 1.110 \times 2^{-3} \end{array}$$

eg

$11.10 \times 2^{-}$

1.110×2

Step 3: normalize

1.110×2^{-3}

Step 4: Round off product.

(only 4 digit) significand.

1.110×2^{-3}

Step 5: Sign bit

-1.110×2^{-3}

eg

1.11111×2^{-5}

1.1111×2^{-5}

(2) Decimal floating point

$1.110_{10} \times 10^{10}$

9.200×10^{-5}

Step 1: exponent addition

$10 + (-5) = 5$

Step 2: Multiply the significant.

$$\begin{array}{r} 1.110 \\ \times\ 9.200 \\ \hline 10.212000 \\ \hline \end{array}$$

10.212000×10^{5}

Step 3: normalize

1.0212000×10^{6}

Step 4: Round off product

1.021×10^{6}

Step 5: Sign bit

1.021×10^{6}

① **Compare the exponent:**

$0.1 \times 2^{0} \Rightarrow 1.0 \times 2^{-1}$

$0.0111 \times 2^{0} \Rightarrow 1.11 \times 2^{-2} \Rightarrow 0.111 \times 2^{-1}$

② **Subtract the significant:**

$1.0 - (0.111) \times 2^{-1} \rightarrow$

$$\begin{array}{r} 1.000 \\ +\;\;\;\;\;\;\;1 \\ \hline 1.001 \\ \hline \end{array}$$

$$\begin{array}{rl} & 1.000 \\ + & 1.001 \\ \hline & 0.001 \times 2^{-1} \\ \hline \end{array}$$

③ **Normalize the significant:**

1.0000×2^{-4}

④ **Round off the significant:**

1.000×2^{-4}

UNIT III

Processor and control Unit

(1)

(1)

BUILDING DATA PATH:

* The datapath is the pathway that the data takes through the CPU.

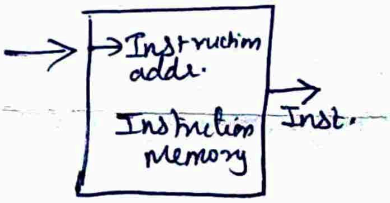
*) The datapath consists of functional units that perform addition, subtraction, logical AND, OR, shifting, etc.

Data path Elements:

The elements are

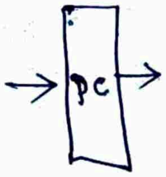
i) Instruction Memory
ii) Register File
iii) Arithmetic logic unit (ALU)
iv) Data Memory
v) Adders

i) Instruction Memory:



→ Instruction memory is a memory unit that is used to store the instruction of a program.

ii) Program counter (PC)



→ PC is used to store the address of the instruction being executed.

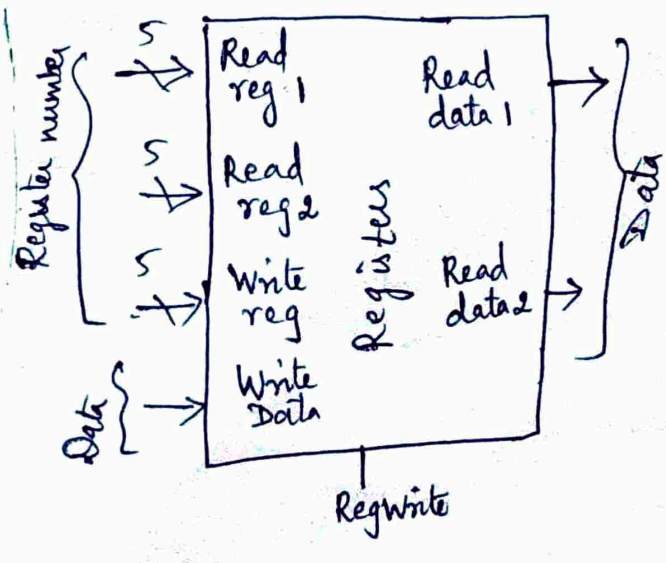
→ It is a 32-bit register

iii) Registers

→ registers are storage location available in CPU.

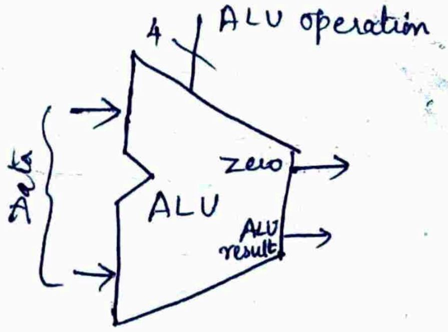
→ Registers has 4 inputs
- two read port
- one write port
- one Write Data

→ Registers has 2 outputs
- Two read data



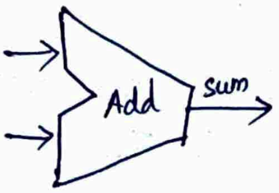
iv) ALU:

- It takes two 32-bit inputs
- Produces a 32-bit result, as well as 1-bit signal if the result is 0.
- 4 bit wide



v) Adder:

→ Adders are used to perform addition of two 32-bit inputs and places the result on its output



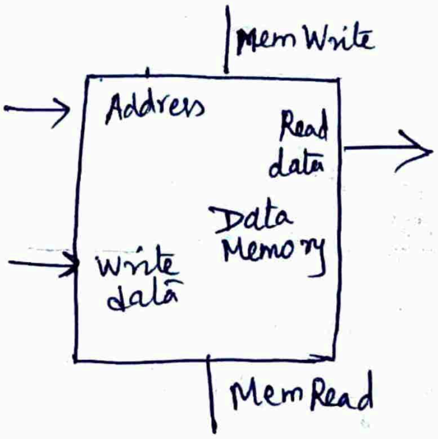
→ Adders increment PC → to point to next instruction.

vi) Data Memory unit

→ It has two input
- Address
- Write Data

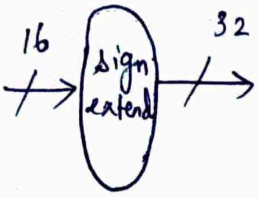
→ It has one output
- Read data

→ contains seperate memory read & write controls



vii) Sign extension Unit:

→ sign extend is used to increase the size of a data item.

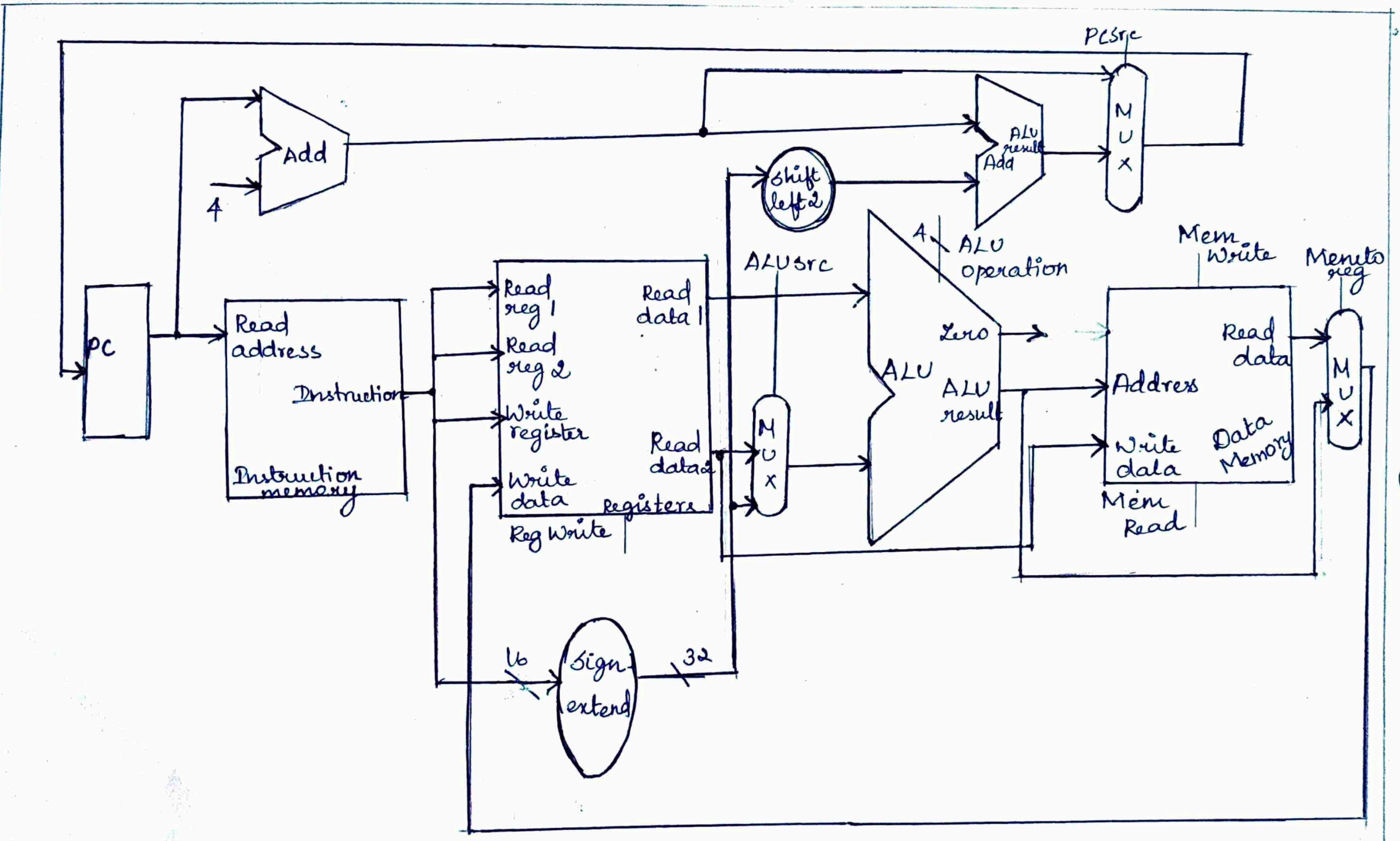


viii) Mux / Multiplexer:

→ It is also called as data selector.

→ It allows multiple input and produces one output.

Figure 4-17



4-17 Simple data path for the MIPS architecture.

③ The ALU Control Implementation :

The MIPS ALU defines 6 following Combinations of 4 Control inputs .

ALU Control lines	Function
0000	AND
0001	OR
0010	add
0110	Subtract
0111	Set on less than
1100	NOR .

*) ALU will need to perform one of these functions .

*) For : load word and store word instructions : , we use the ALU to compute memory address by addition .

*) For the R- type instructions , ALU needs to perform one of the five actions (AND, OR, add, subtract , or set on less than) .

*) For branch equal , ALU performs subtraction .

ALU op : The 4 bit ALU control input uses small Control unit that has the function field of instruction and a 2- bit control field , which is called as ALU op .

*) ALU op indicates whether the operations ~~to~~ to be performed should be add for loads and stores (or) subtract for beq .

*) The output of ALU control unit is 4 bit signal that Controls the ALU by generating one of the 4- bit combinations.

Implementation technique:

*) The main control unit generates the ALUOP bits.

*) The ALU control generates the signals to control the ALU unit.

*) Here ALU op bits are used as inputs.

*) Using Multiple level of control reduces the size of the main control unit.

*) Using several smaller control units increases the speed of the control unit.

*) The speed of the control unit is critical to clock cycle time.

Designing the main control unit:

In order to design the main control unit, three instruction classes are used. They are

1. R- type instructions
2. Branch instructions
3. Load - store instructions.

1. R- type instructions :

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0 .

*) Opcode → 0

*) These instructions have three registe operands → rs, rt, rd .

*) rs, rt → Sources

*) rd → destination .

*) The ALU function is in the funct field .

*) The shamt field is used only for shifts .

2. Branch instructions :

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

*) Opcode → 4 .

*) The registers rs and rt are source registers that are compared for equality .

*) The 16 bit address field is sign extended, shifted and added to the PC to compute the branch target address .

3. Load or Store instruction :

Field	35 or 43	rs	rt	address
Bit positions	31 : 26	25 : 21	20 : 16	15 : 0

*) Opcode → 35_{ten} (load).

*) Opcode → 43_{ten} (store).

*) rs → base register.

*) It is added to 16 bit address field to form the memory address.

*) For loads, rt → destination register.

*) For stores, rt → source register.

Major observations about these instructions :-

*). Opfield is also called as opcode (contained in 31 : 26 bits).

Opcode : The field that denotes the operation and format of an instruction.

*) The registers to be read are always rs and rt fields at 25 : 21 and 20 : 16.

*) The base register for load and store, always in the position 25 : 21 (rs).

*) Destination register is in one of the 2 places.

1) For load (position 20 : 16 (rt))

2) For R-type Instruction (position (15 : 11) rd)

The Simple datapath with the Control unit :

*) The input to the control unit is the 6-bit opcode field from the instruction.

*) The output of the control unit consists of 8 signals.

1) Three 1 bit signals → used to control multiplexors.

2) Three signals (Reg write , MemRead and Memwrite) → controlling reads and writes in the register and data memory.

3) 1- bit Signal (Branch) → used to determine whether to possibly branch.

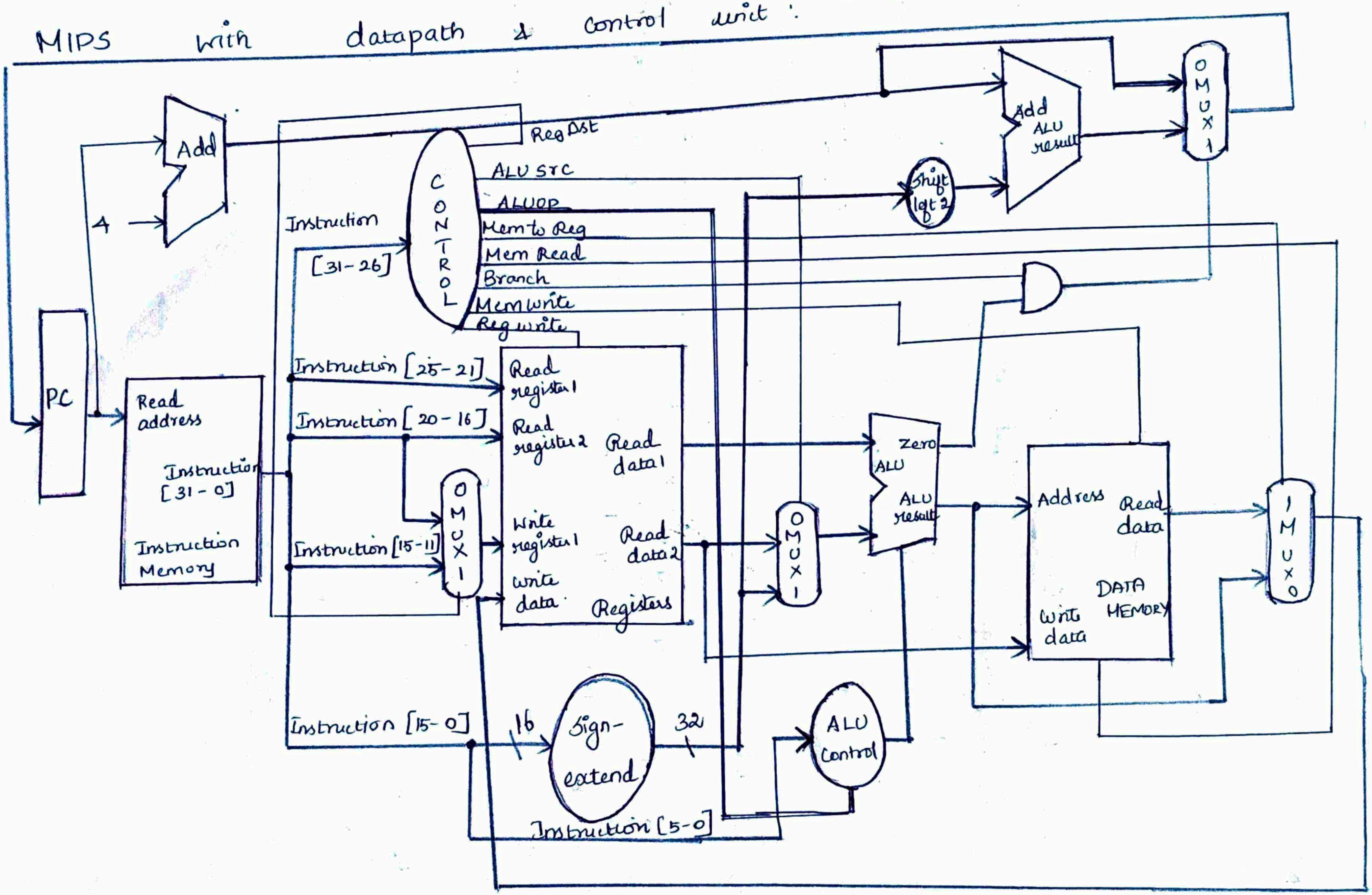
4) 2 - bit Control signal (ALUOp) → used for ALU.

*) ~~An~~ AND gate is used to combine the branch control signal and the Zero output from the ALU.

*) the AND gate output controls the selection of the next PC.

Figure 4-17 (Book)

MIPS with datapath & control unit :



MIPS Data path & control path

4 Pipelining:

* Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

* It is a most important technique used to increase the speed and performance of the processor.

* Pipeline approach take less time

* To execute MIPS instruction using pipeline it has five steps. They are.

⇒ **Fetching** - fetch instruction from memory.

⇒ **Decoding** - read registers while decoding the instruction

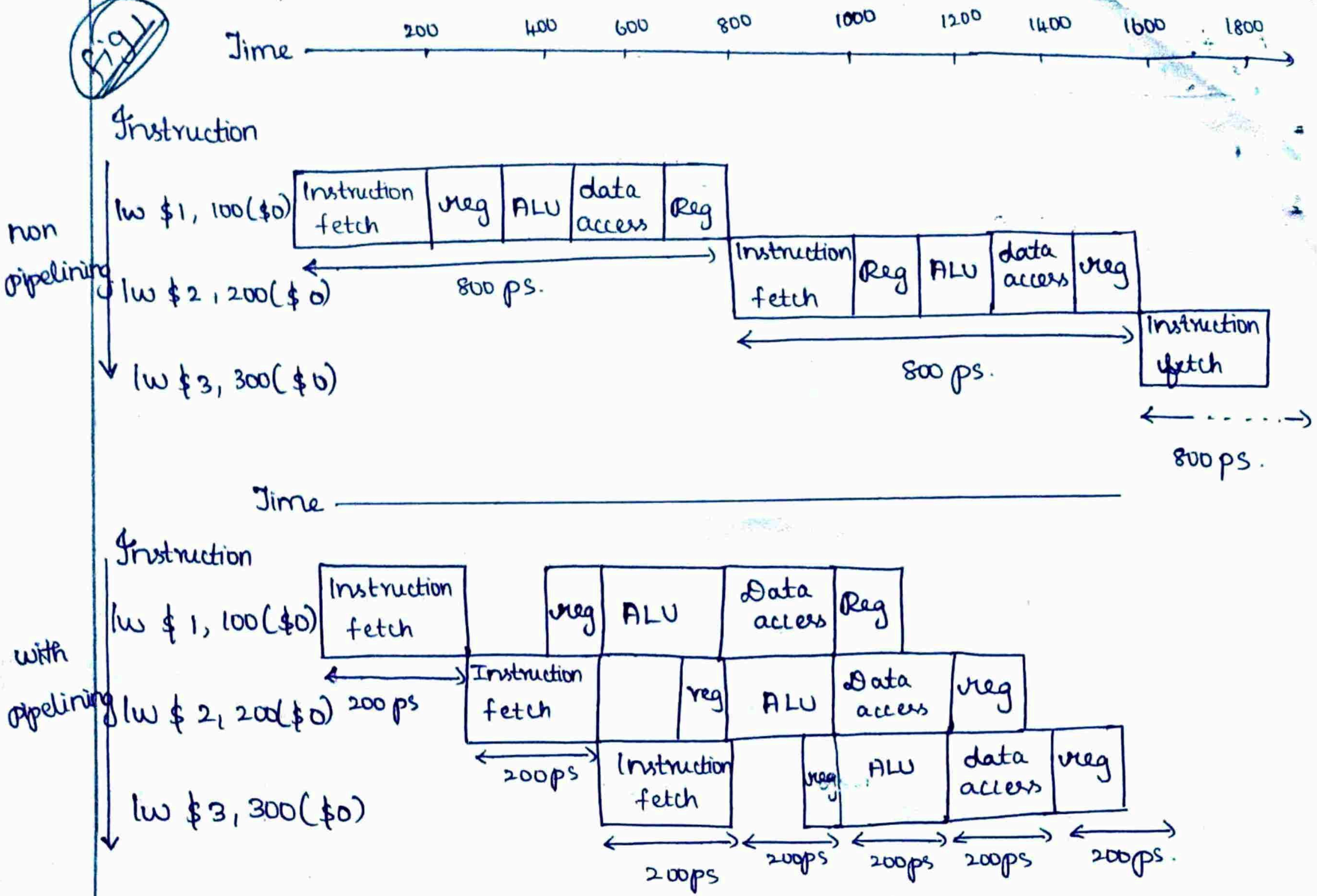
⇒ **Execution** - execute the operation.

⇒ **Memory access** - access an operand in data memory.

⇒ **Write back** - write the result into the register

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipe-n}}}{\text{number of pipe stages.}}$$

Fig 1



Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction.

Pipeline Hazards:

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards. There are three different types of hazard. They are.

1) Structural hazard
2) Data hazard
3) Control hazard.

1) **Structural Hazards:-**

Structural hazards are the hazards in which the hardware does not support the combination of instructions that are set to execute.

2) **Data hazard:-**

Data hazard also called pipeline data hazard.

Data hazard occurs when the data that is needed to execute the instruction is not available.

Eg: ADD s0, t0, t1

SUB t2, s0, t3.

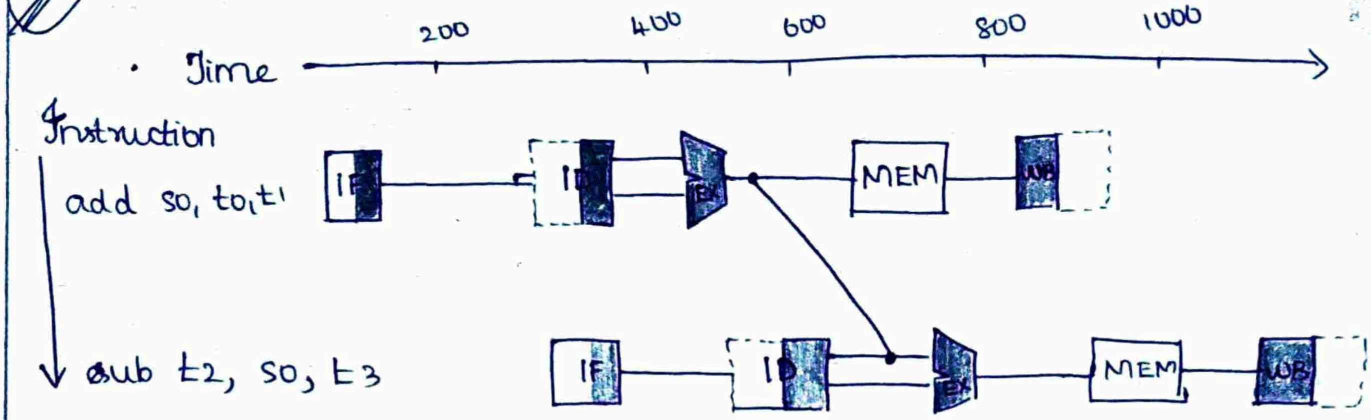
The solution for data hazard is forwarding

a) **Forwarding:-**

Also called bypassing.

It is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible memory.

fig2 Diagram for forwarding.



b) Load ~~use~~ use data hazard:-

It is a specific form of data hazard in which data being loaded by a load instruction has not yet become available when it is needed by another instruction.

c) Pipeline stall:-

Also called bubble

A stall initiated in order to resolve a hazard.
→ Draw Fig 3 here also.

3) Control Hazards:-

The third type of hazard is called control hazard.

Control hazard is also called branch hazard

→ Draw Fig 3.

Control hazard occurs when the flow of instruction addresses is not what the pipeline expected.

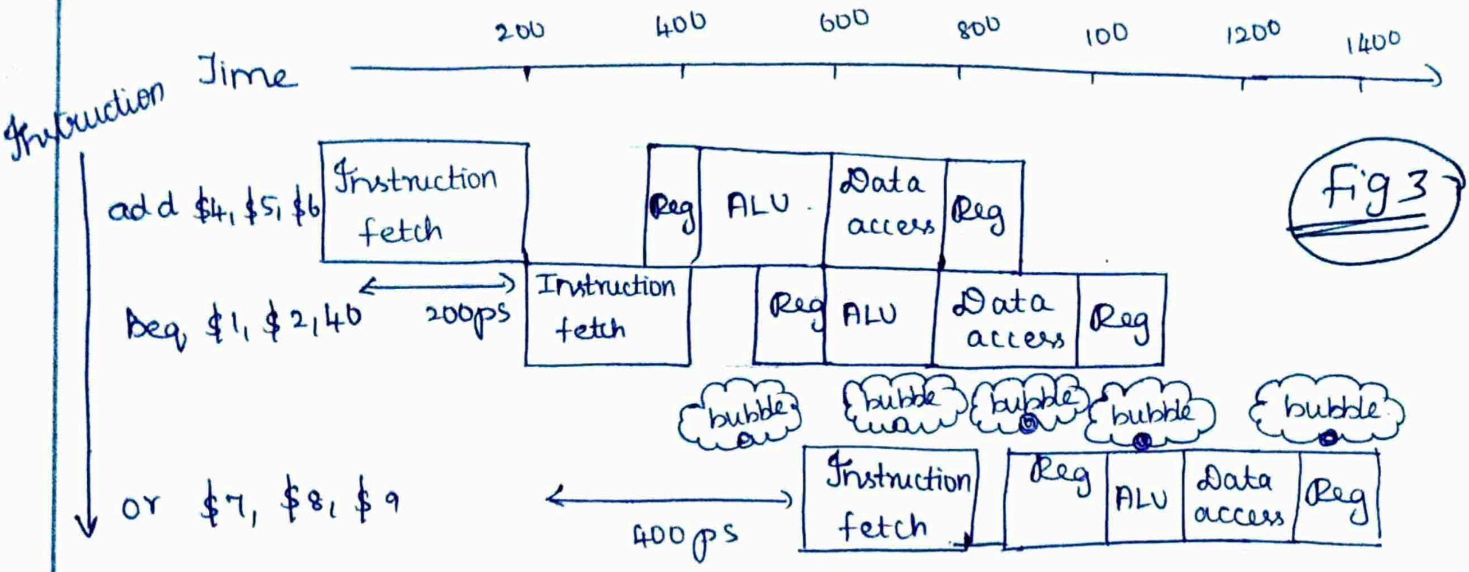


Fig 3

Branch prediction:

A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

(5) Pipelined Datapath

Pipelining has five different stages. They are

- * Instruction Fetch (IF)
- * Instruction decoding (ID)
- * Execution (EX)
- * Memory access (MEM)
- * Write back (WB)

1) Instruction Fetch:.

⇒ The top portion is the instruction fetch which shows the instruction being read from memory using address in the PC and then being placed in the IF/ID pipeline register.

⇒ The PC address is incremented by 4 and then written back into the PC to be ready for next clock cycle.

2) Instruction decoding:.

Instruction decoding shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two register.

3) Execution:.

It shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using ALU.

4) Memory access:-

It is the top portion shows the load instruction reading the data memory using the address from EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

5) Write back:.

It is the bottom portion which shows the final step: reading the data from MEM/WB pipeline register and writing it into the register file in the middle ~~of the figure~~.

Types of Instruction:-

There are three basic types of instruction. They are

1) R-type instruction
2) Load type instruction
3) Store type instruction

① Load Instruction and Store Instruction:-

⇒ The ~~dath~~ datapath of load word and store word instructions are created by executing five stages of pipelined process.

⇒ Load word instruction is active in all five stages.

⇒ Left half of the register is used for written and right half of the register used for read.

⇒ Now we shall discuss about the load word and store word in detail.

i) Load word:-

Instruction Fetch:-

⇒ In IF, the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register.

⇒ The PC address is incremented by 4 and then written back into the PC to be ready for next clock cycle.

⇒ This incremented address is also saved in the IF/ID pipeline register in case it is needed for later instruction.

ii) Instruction decoding (ID):.

⇒ The instruction portion of IF/ID pipeline register supplys 16 bit immediate field.

⇒ It is sign extended to 32 bits and the register numbers to read the two registers.

⇒ All the three values are stored in the ID/EX pipeline register along with the incremented PC address.

iii) Execution:.

⇒ Load instruction reads the contents of register 1 and the sign extended immediate from the ID/EX pipeline register and adds them using ALU

⇒ That sum is placed in EX/MEM pipeline register.

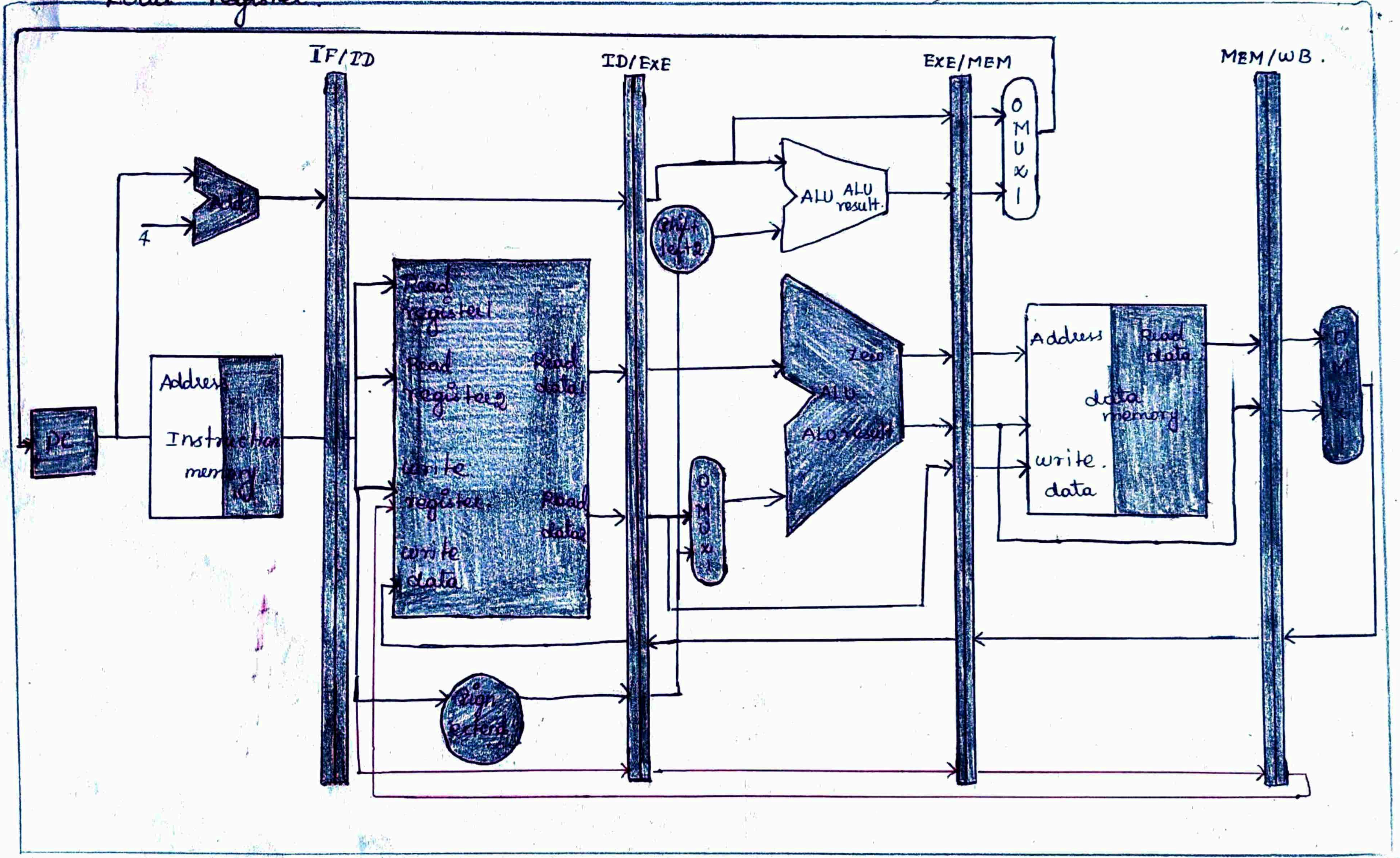
iv) Memory Access:.

⇒ Load Instruction read the data memory using the address from EX/MEM pipeline register and loading the data into MEM/WB pipeline register

Write Back:. Reading the data from the MEM/WB pipeline register and writing it into register file in the middle.

Load ~~register~~:

LOAD Instruction (pipelining)



2) Store word:-

i) Instruction Fetch:-

⇒ In IF, the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register.

⇒ The PC address is incremented by 4 and then written back into the PC to be ready for next clock cycle.

⇒ This incremented address is also saved in the IF/ID pipeline register in case it is needed for later instruction.

ii) Instruction decoding (ID):-

⇒ The instruction portion of IF/ID pipeline register supplys 16 bit immediate field.

⇒ It is sign extended to 32 bits and the register numbers to read the two register.

⇒ All the three values are stored in the ID/EX pipeline register along with the incremented PC address.

iii) Execution:-

⇒ ~~Load~~ Store instruction reads the contents of register1 and the sign extended immediate from the ID/EX pipeline register and adds them using ALU

⇒ That sum is placed in EX/MEM pipeline register.

iv) Memory Access:-

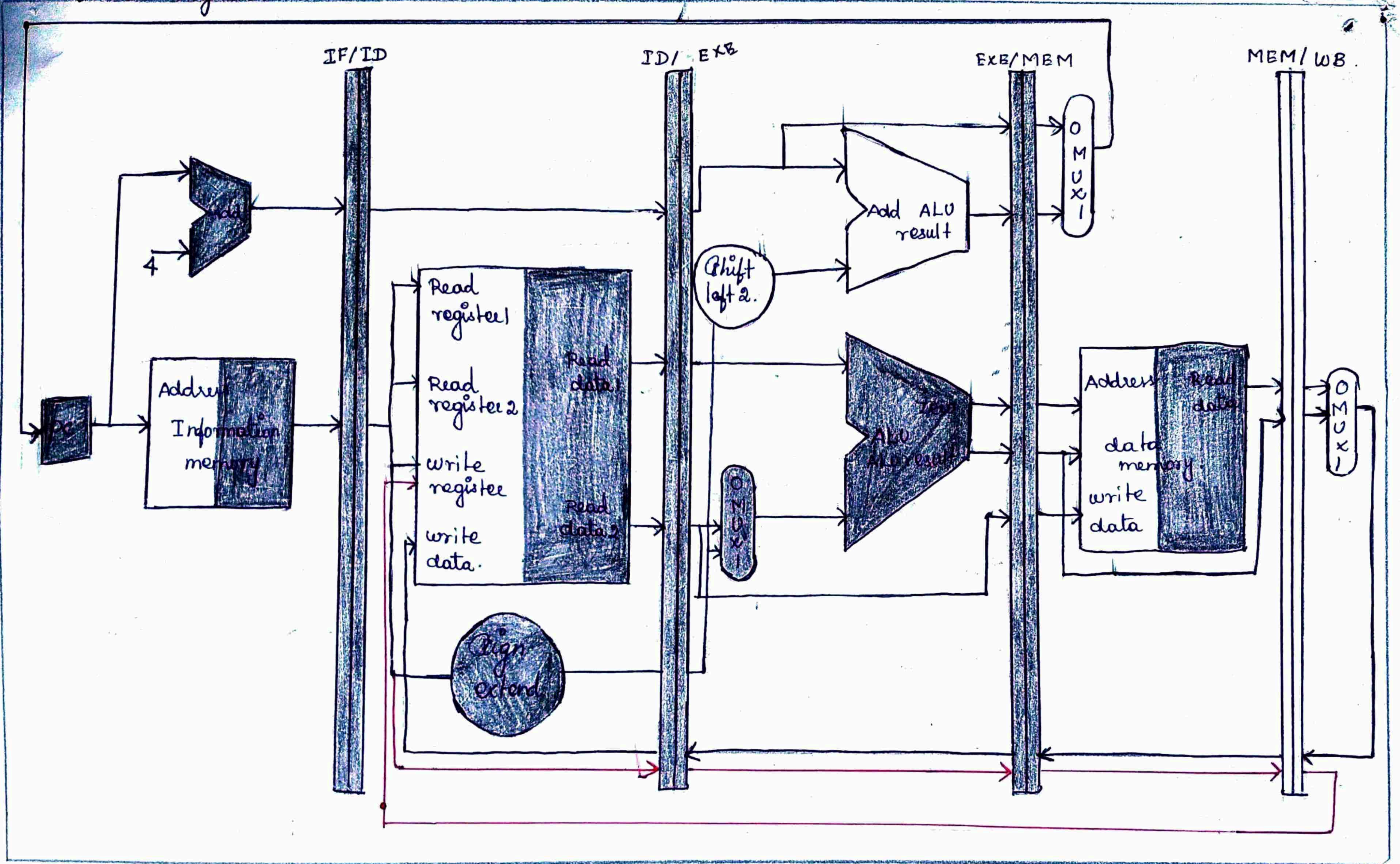
Store instruction read the data memory using the address from EX|MEM pipeline register and storing the data into MEM |WB pipeline register.

v) Write Back:.

No process is done

Store Instruction (pipelining)

Store ~~register~~:



(3) R- type Instruction:-

⇒ Instruction fetch

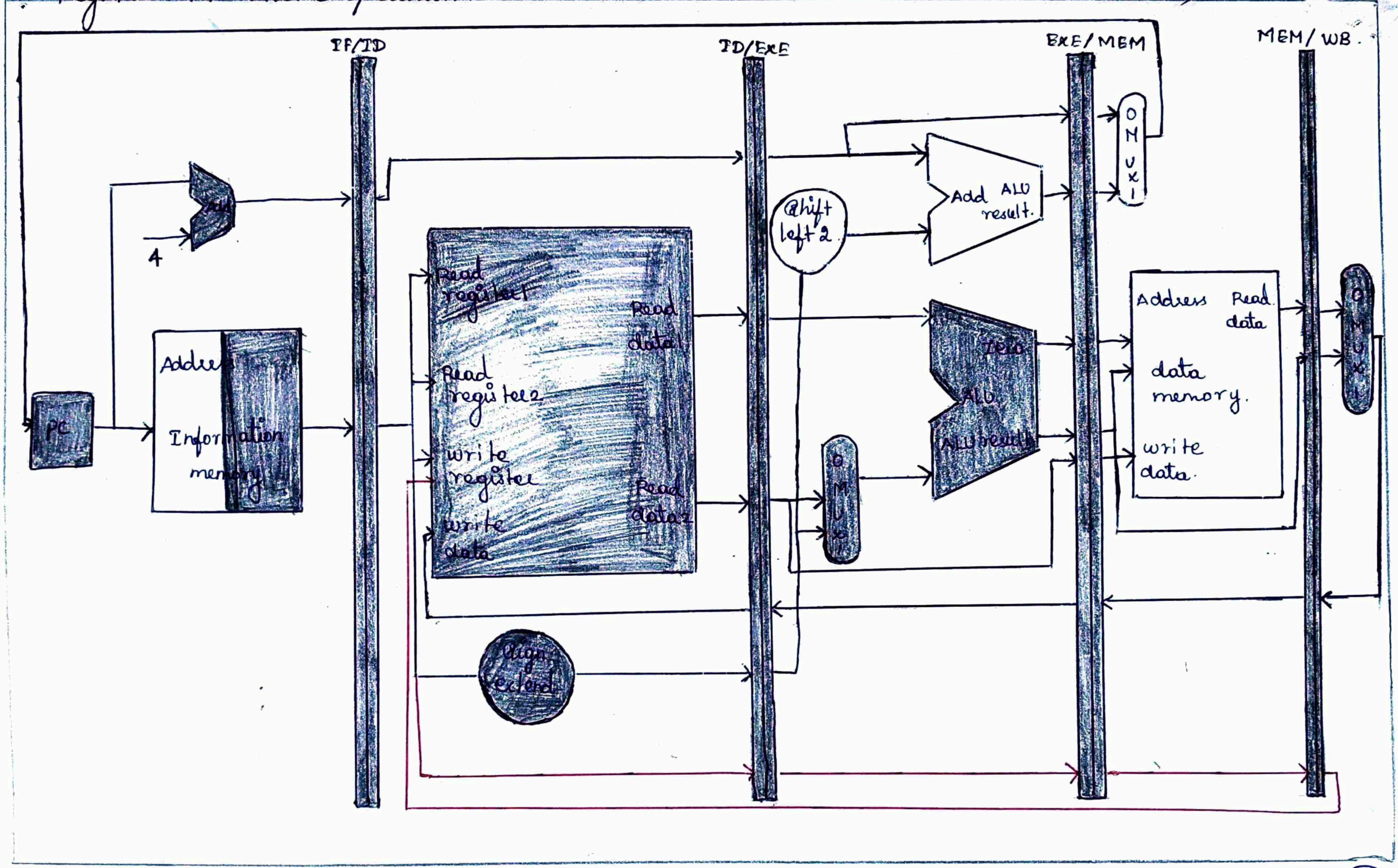
⇒ Instruction decode and register file read

⇒ Execute and address calculation - second register value is stloaded into the EX/MEM pipeline register to be used in the next stage

⇒ Memory access - no process is done.

Register Type Instruction (pipelining)

Register Arithmetic operation



(6) Pipelined control :.

Pipeline can be implemented in two ways :-

1. Single cycle implementation.
2. Multiple cycle implementation.

*. As we have already discussed the single cycle datapath, now we are going to discuss about how to provide control to the pipelined datapath.

*. To provide control to the pipelined datapath, the following task has to be performed.

Step 1 :- Label the control lines on the existing datapath.

Step 2 :- Borrow the control from the simple datpa

Step 3 : Use same ALU control logic, branch logic, destination regdister number multiplexer and control lines used in simple datapath.

Pipeline control for single cycle implementation.

*. In case of single cycle implementation, assume that the PC is written on each clock cycle.

*. So no need to provide separate write signal for the PC.

*. Suppose, if we are using pipline control in single cycle implementation for that also no need to separate write signals.

*. Because pipeline registers also written during each clock cycle.

*. To specify the control for the pipeline we have to set the control values during each pipeline stage.

*. Because each control line is associated with a component active in ohly a single pipeline stage.

We have five stages for pipeline such as.

*. Instruction fetch.

* Instruction decode/register file read.

* Execution/address calculation.

* Memory access.

*. Write back.

Instruction fetch:

The control signals to read instruction memory and to write the PC always asserted. So in the instruction fetch there is nothing ~~to~~ special to control in this pipeline stage. For instruction fetch stage we don't want special control line.

Instruction decode / register file ~~&~~ read:-

As happened in the Instruction fetch, here also the same thing happens in the every clock cycle. So in this stage also we no need to set any control lines.

Execution / address calculation:

In this stage we have to use some control lines for example such as RegDst, ALUOP, and ALU src. These control lines are used to select the result register, ALU operation and either the read data or sign extended immediate for the ALU.

Memory Access:

In this stage the following control lines are used.

are used such ~~B~~ Branch, MemRead, MEMWrite

The branch equal, load and store instructions are used above instruction lines.

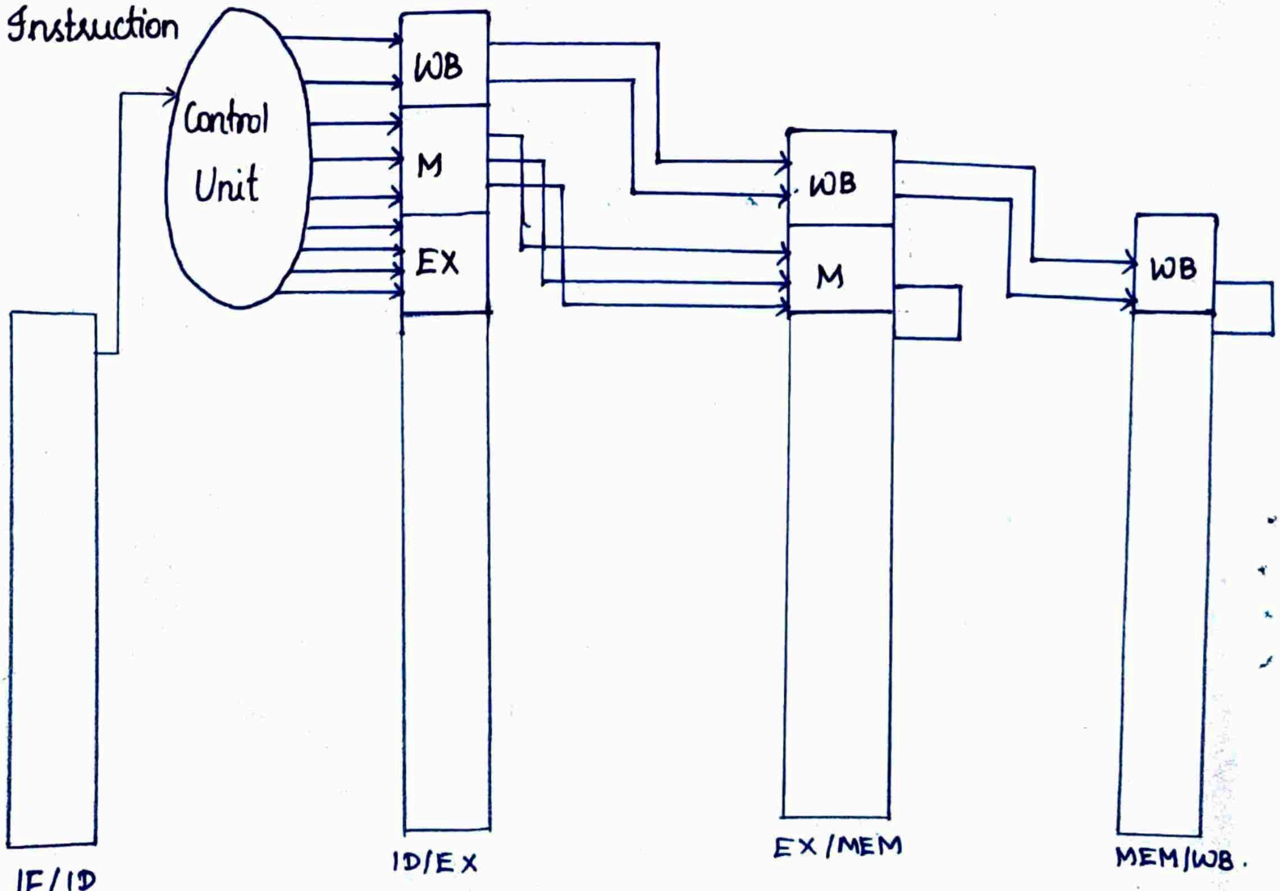
Write back::

In this stage two control lines are used such as MemtoReg and Regwrite.

MemtoReg:: which decides between the ALU result or the memory value to the register file.

Regwrite:: writes the chooren value.

For pipeline control we have seven control line are used in five stages of pipeline process.



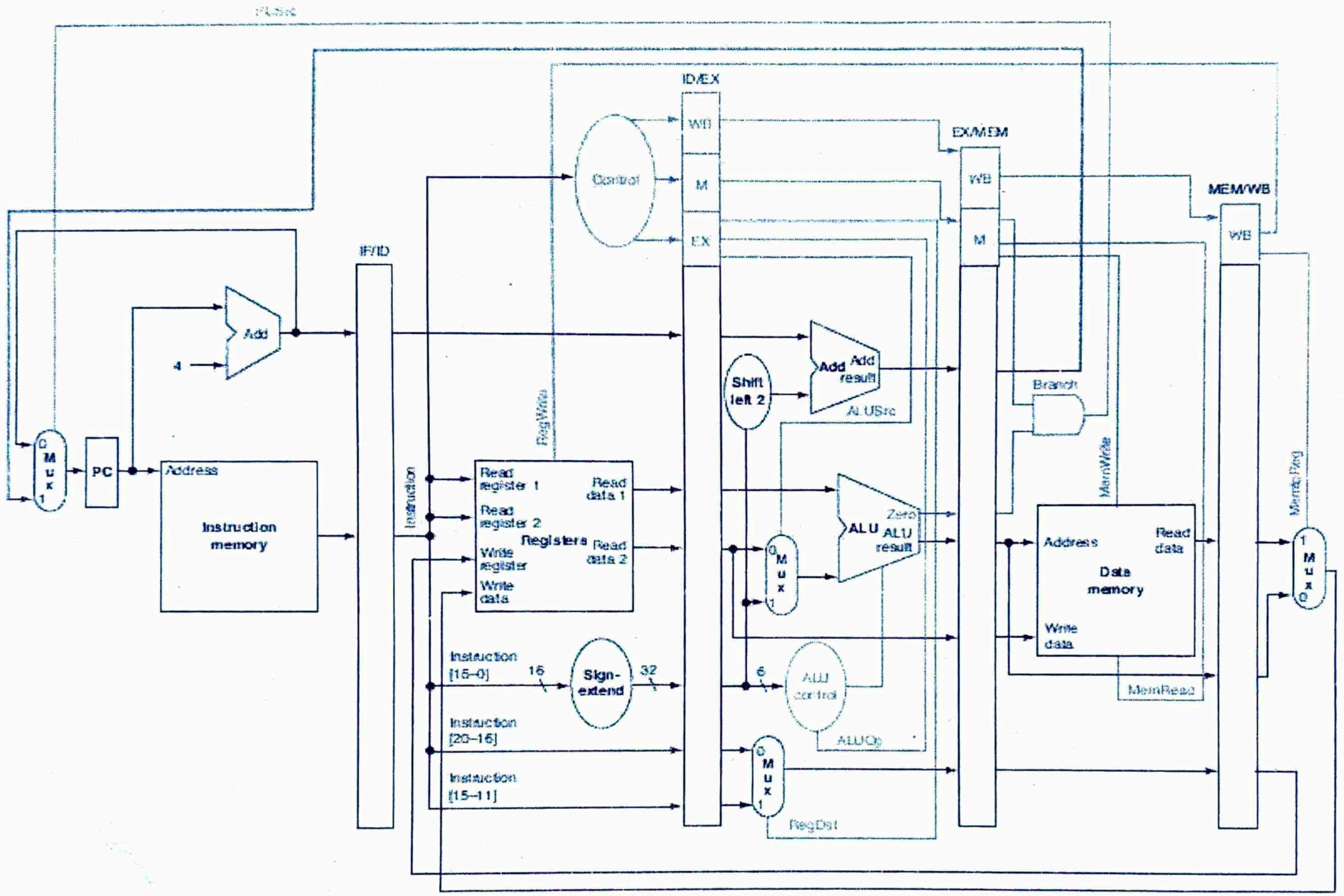


Figure 4-51 : Pipelining (Pipelined Datapath and Control Signals)

7 DATA HAZARDS: Forwarding Versus Stalling

Data Hazards:

Also called a pipeline data hazard. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instructions is not yet available.

Data Hazards are obstacles to pipelined execution

```
sub $2, $1, $3      # Register $2 written by sub
and $12, $2, $5     # 1st operand ($2) depends on sub
or  $13, $6, $2     # 2nd operand ($2) depends on sub
add $14, $2, $2     # 1st ($2) + 2nd ($2) depend on sub
sw  $15, 100($2)    # Base ($2) depends on sub
```

The last four instructions are all dependent on the result in register $2 of the first instruction. If register $2 had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following register that refer to register $2.

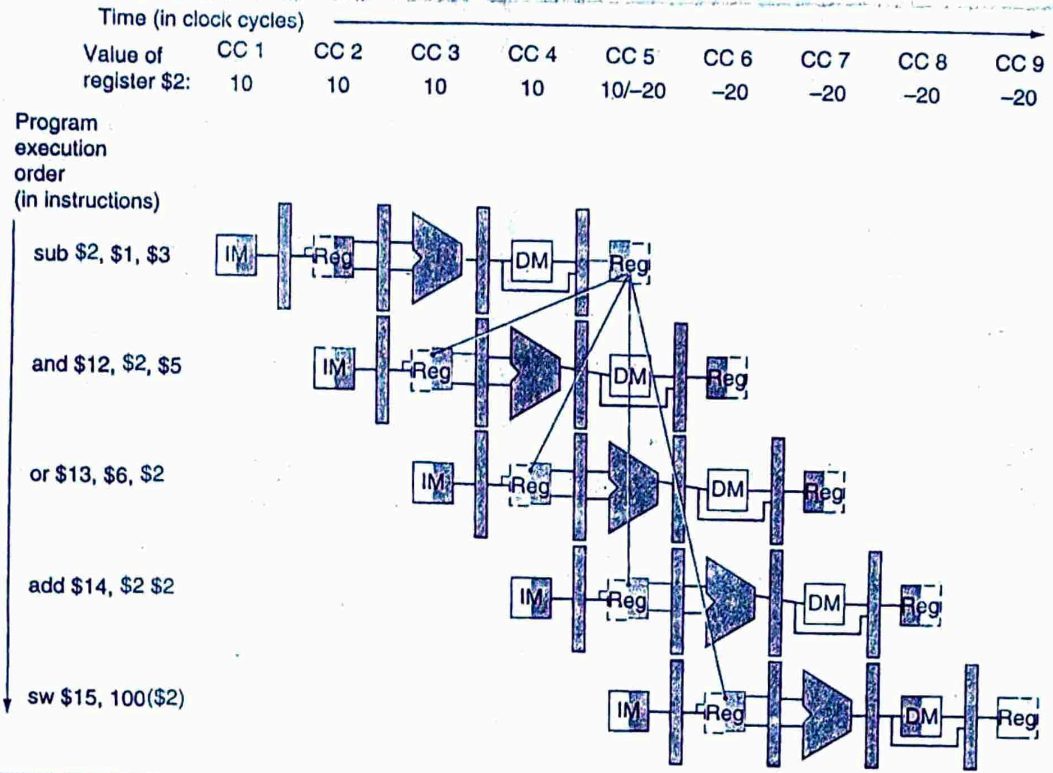


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into $2, and all the following instructions read $2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards.*

Fig 1:- ~~Pipep~~ Pipelined dependencies in a five-instruction sequence using simplified datapaths to show the dependencies.

* The values read for register \$2 would not be the result of the sub instruction unless the read occured during clock cycle 5 or later.

* The lines from the top datapath to the lower ones show the dependencies.

* Fig 2 shows the dependencies between the pipeline registers and the inputs to the ALU for the same code sequence as in Fig 1.

* The change is that the dependencies begin from a pipeline register, rather than waiting for the WB stage to write the register file.

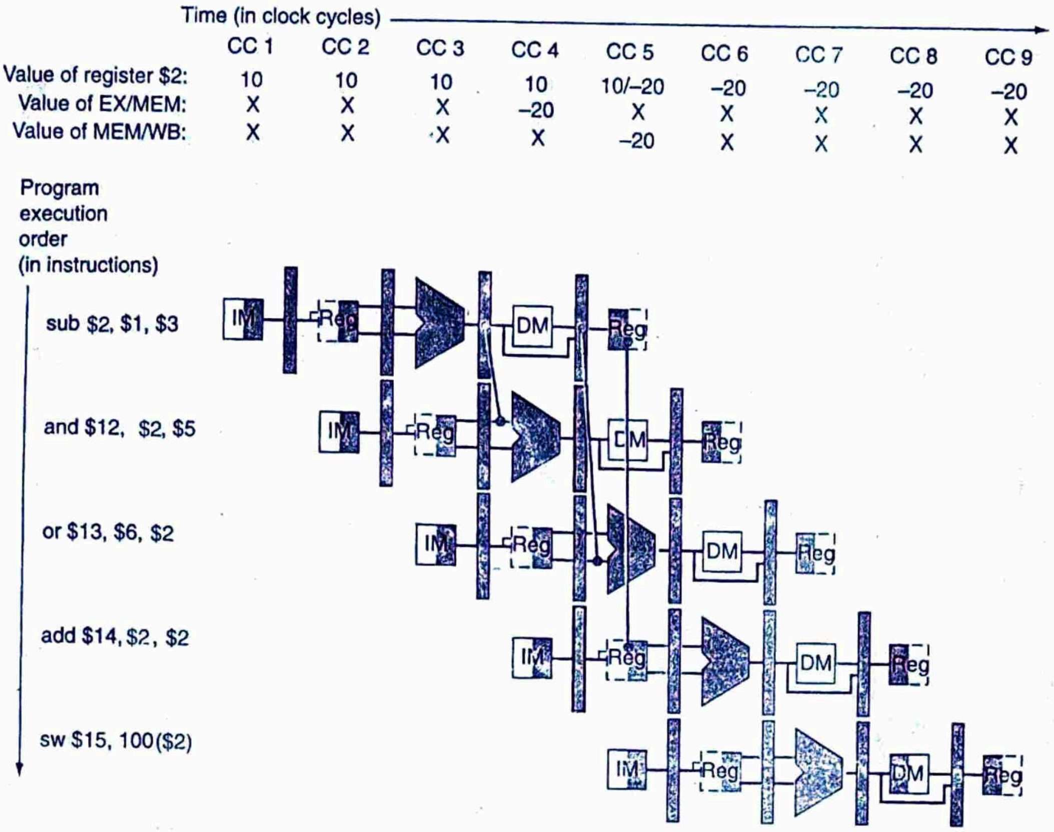
* Thus, the required data exists in time for later instructions, with the pipeline registers holding the data to be forwarded.

The conditions for detecting hazards and the control signals to resolve them.

1] EX Hazard:-

```
    if (EX/MEM.RegWrite
   and (EX/MEM.RegisterRd ≠0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA=10

    if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd ≠0)
 and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB =10
```



Time (in clock cycles)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

FIGURE 4.53 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the AND instruction and OR instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register. Register file "forwarding"—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register $2 having the value 10 at the beginning and –20 at the end of the clock cycle. As in the rest of this section, we handle all forwarding except for the value to be stored by a store instruction.

Fig 2

2] MEM hazard:

if (MEM/WB. RegWrite
and (MEM/WB. RegisterRd ≠ 0)
and (MEM/WB. RegisterRd = ID/EX. RegisterRs)) ForwardA = 01

if (MEM/WB. RegWrite
and (MEM/WB. RegisterRd ≠ 0)
and (MEM/WB. RegisterRd = ID/EX. RegisterRt))
ForwardB = 01

Data Hazards and Stalls :-

```
if (ID/EX . Mem Read and
((ID/EX . Register Rt = IF/ID. Register Rs) or
(ID/EX . Register Rt = IF/ID . Register Rt)))
Stall the pipeline.
```

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline
```

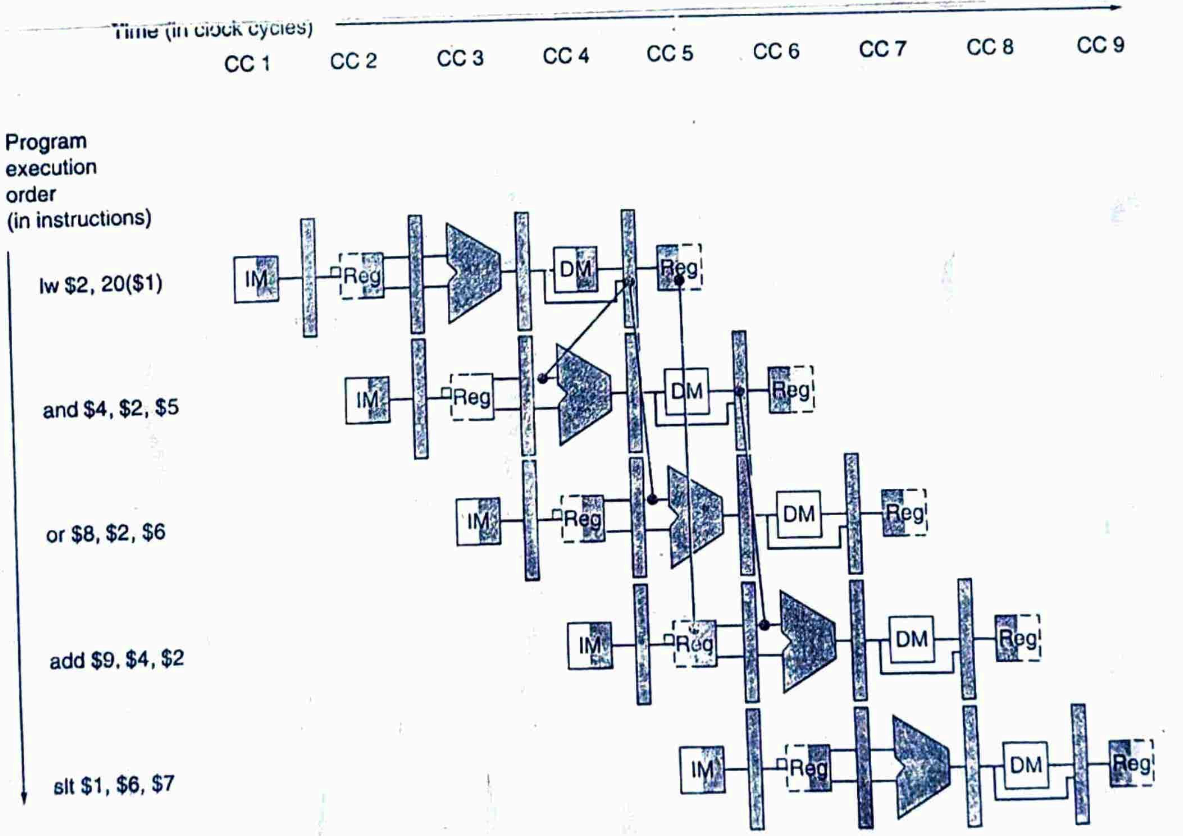


FIGURE 4.58 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

Data Hazards and Stalls:

when an instruction tries to read a register following a load instruction that writes the same register.

Nop: An instruction that does no operation to change state.

Fig: 3 A Pipelined Sequence of Instructions where stalls are inserted

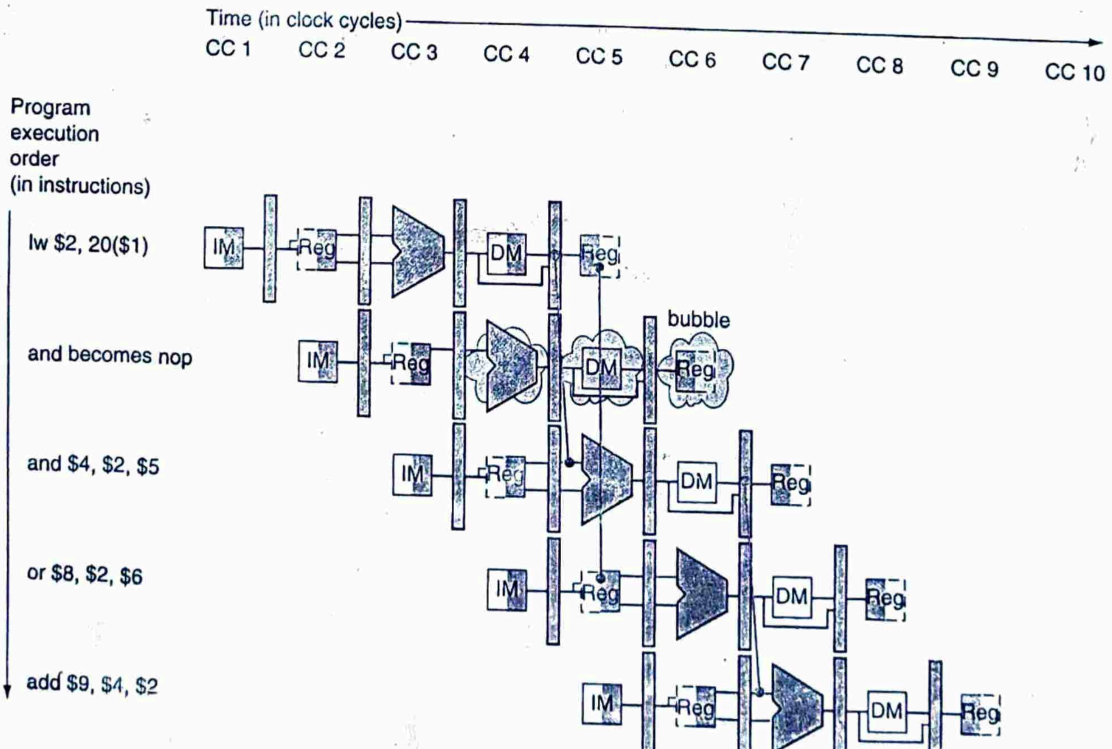


FIGURE 4.59 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the OR instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

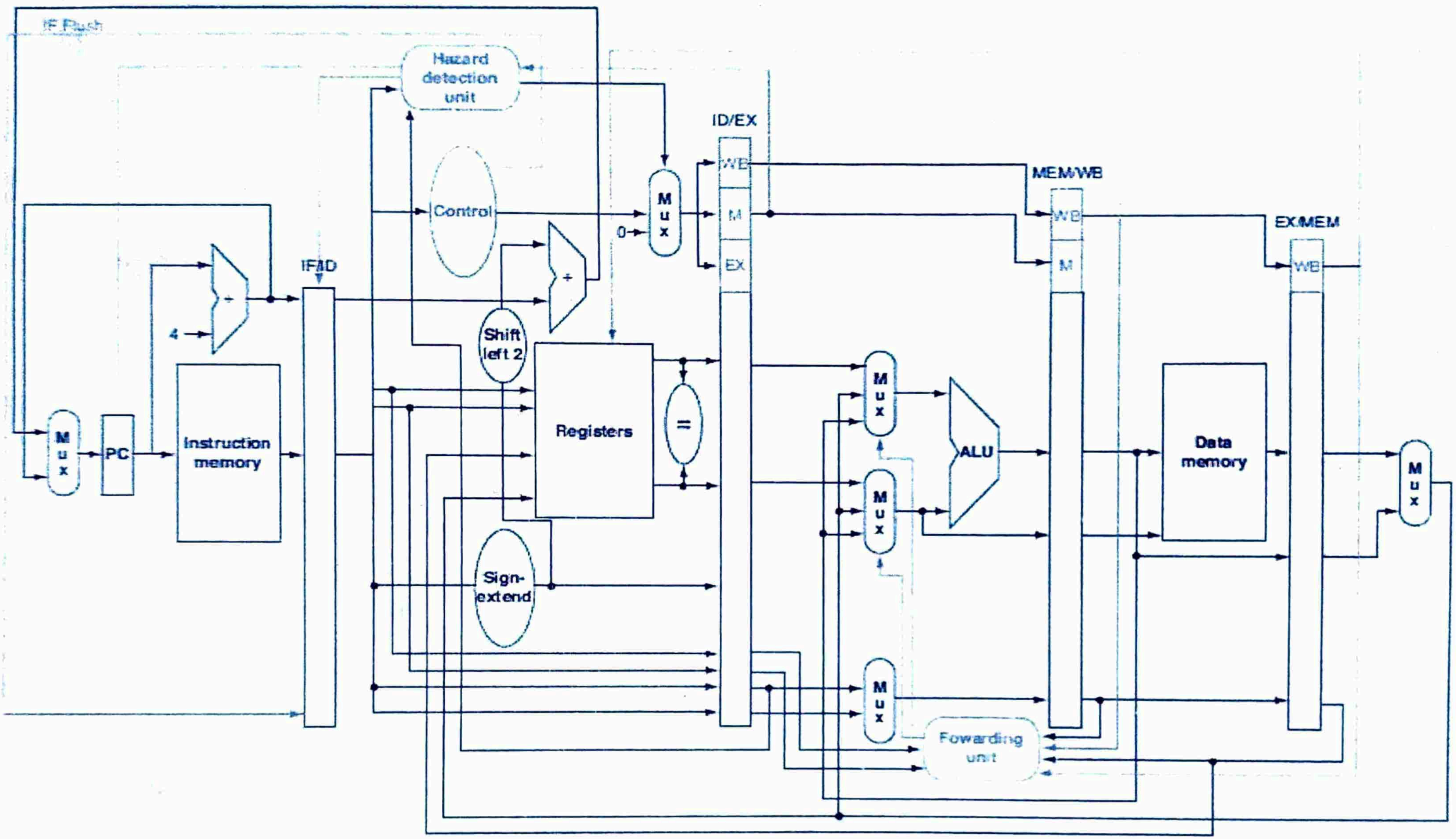


Figure 4-65 : Hazards (Data path and control)

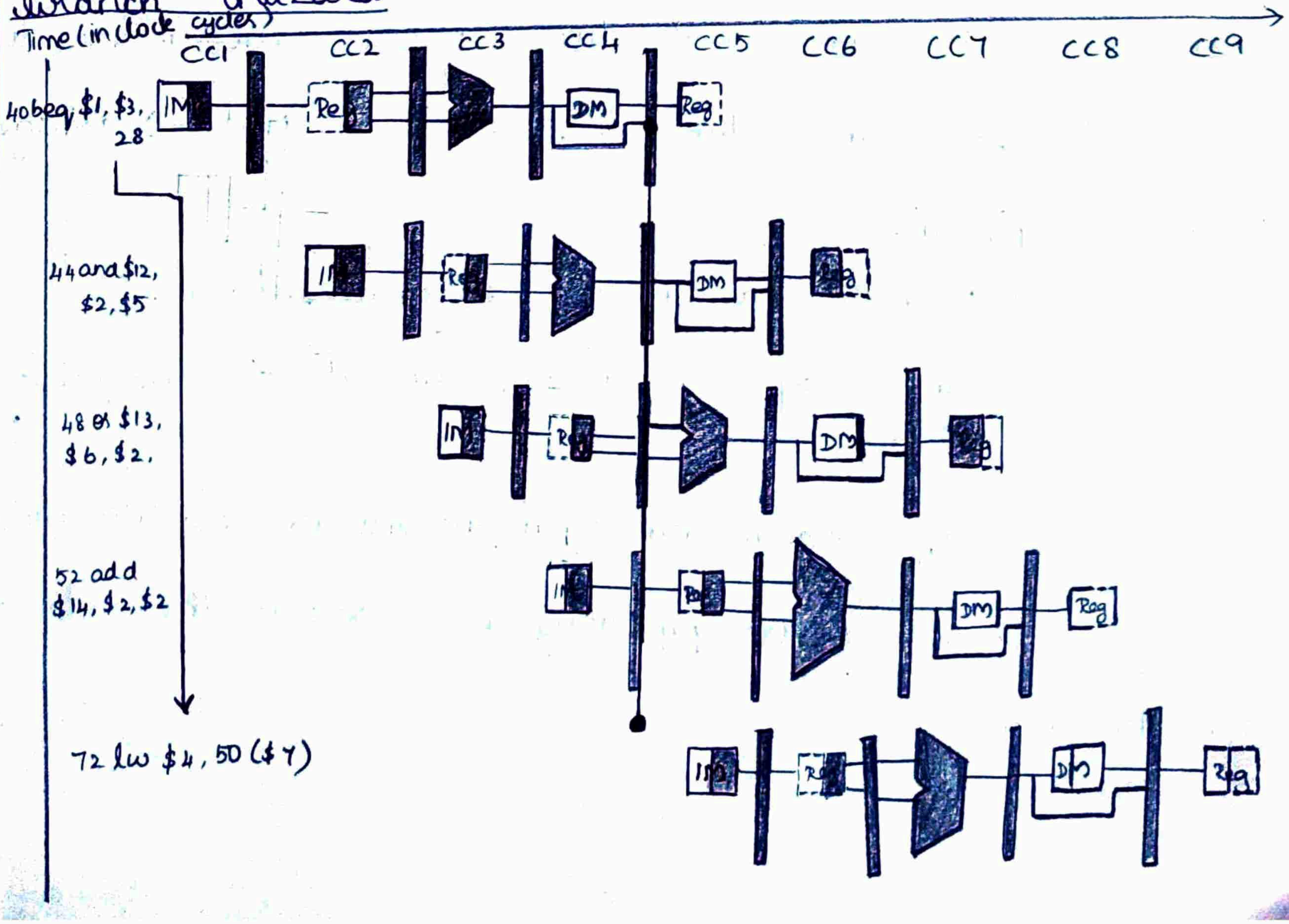
8 Control Hazards:-

★ Control hazards occurs when we execute the branch instruction in pipeline process'

★ Control hazards also called branch hazards when the proper instruction cannot execute in the proper clock cycle

★ An instruction must be fetched at every clock cycle to sustain the pipeline.

★ The delay in determining the proper instruction to fetch is called control hazard or branch hazard.



Methods to resolve control hazards.

The methods to resolve control hazards are

① * Branch Not Taken
② * Reducing the delay of branches
③ * Dynamic Branch Prediction.

① Branch Not Taken:

* If the branch is taken then the instructions that are being fetched and decoded must be discarded

* After discarding the execution continues at the branch target.

* Discarding instructions means we must be able to flush instructions in the IF, ID and EX stages of the pipeline.

* Flush is a method used to discard instructions in a pipeline usually due to an unexpected error.

* If branch not taken means no need to discard any instructions and pipelining will execute the instructions continuously.

* Branches are taken then only pipeline has stalled so by reducing the delay of branches we can improve the performance of pipelining process in this condition.

② Reducing the delay of Branches:.

* One way to improve branch performance is to reduce the cost of the taken branch.

* The next PC for a branch is selected in the MEM stage only but if we move the branch execution earlier in the pipeline then fewer instructions need to be flushed.

* Moving branch execution earlier in the pipeline will increase the speed of performance.

* When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is required - a situation that is similar to the use of condition codes for branches.

* Moving the branch decision up requires two actions to occur earlier. They are

→ computing the branch target address

→ Evaluating the branch decision.

* The easy part of the change is to move up the branch address calculation.

* Now, just move the branch adder from the EX stage to ID stage.

* The harder part is the branch decision itself. For branch equal, we would compare two registers read during ID stage to see if they are equal.

* Equality is tested by using exclusive ORing.

* For example, to implement branch on equal, we will need to forward results to the equality test logic that operates during ID.

There are two complicating factors.

1) During ID, we must decode the instruction, decide whether bypass to the equality unit is needed.

Forwarding the operands 29 of branches are handled by ALU forwarding logic unit.

Note that bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.

2) The values in a branch comparison are needed during ID stage but may be produced later in time, it is possible to occur data hazard and stall will be needed.

To overcome these difficulties we can move the branch EX to the ID stage, because it reduces the penalty of a branch to only one instruction if the branch is taken.

(3) Dynamic Branch Prediction.

★ Dynamic branch prediction is the predictior of branches at runtime using runtime information.

★ One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time. This technique is called dynamic branch prediction.

★ One implementation of that approach is a branch prediction buffer or branch history table

★ A branch prediction buffer is a small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not

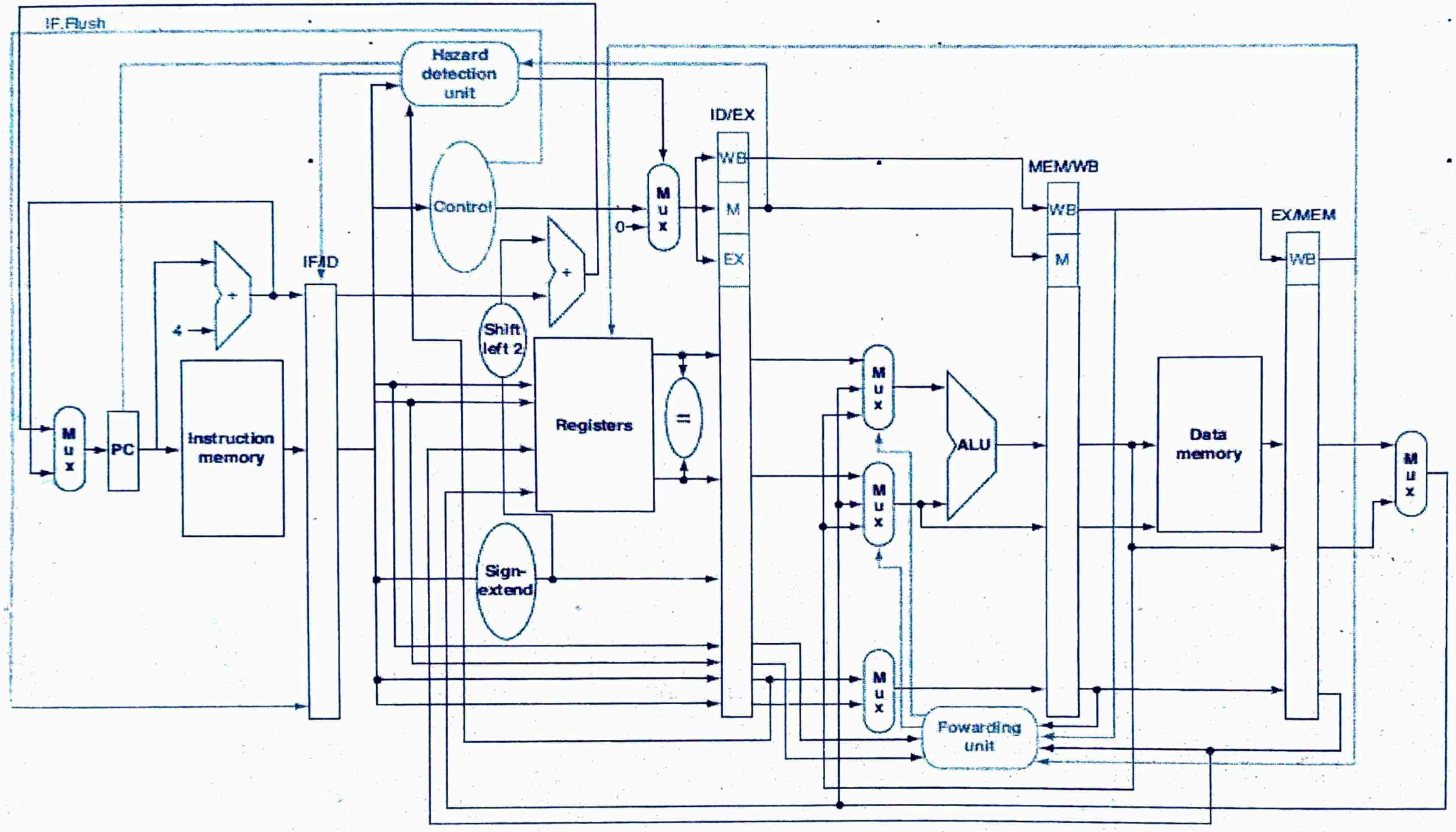


Figure 4-65 : Hazards **(Data path and control)**

9

EXCEPTIONS

Control is the most challanging aspect of processor design. One of the hardest parts of control is implementing :

* exceptions and
* interrupts.

Exceptions : Exceptions are also called interrupt.

* An unscheduled program event that distrupts the execution of the program is exception. [Internal or external distrubance]

Interrupts : An exception that comes from outside of the processor is interrupts. [external distrubance]

They were initially created to handle unexpected events from within the processor, like arithmetic overflow.

→ Exception refer to any unexpected change in control flow without distinguishing whether the cause is internal or external.

→ Interrupt refer to only the event which is caused externally.

Type of event	From where	MIPS terminology.
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception.
Using an undefined instruction	Internal	Exception.
Hardware malfunctions	Either	Exception or interrupt

* Many of the requirements to support exceptions come from the specific situation that causes exception to occur.

* Without proper attention to exceptions during design of the control unit, complicate the task of getting the design correct.

How exceptions are handled in the MIPS Architecture :

* The two types of exceptions that occur ~~can generate~~ are ~~the~~ i) undefined instruction and an ii) arithmetic overflow.

a) Exception Program counter (EPC)

* The basic action that the processor must perform when an exception occurs is to save the address of the offending instruction in the Exception Program Counter (EPC) and then transfer control to the operating system at some specified address.

The operating system can take the appropriate action, which may involves the following,

* Providing some service to the user program.
* taking some predefined action in response to a overflow.
* reporting an error.

* After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its executingon using EPC.

The method used is the MIPS Architekture is to include Cause register which holds a field that indicates the reason for the exception.

b) Vector Interrupts

A second method, vector interrupts.

Vector interrupt: An interrupt for which the address to which the control is transferred is determined by the cause of the exception.

Exception type	Exception vector address (in hex)
Undefined instruction	8000 0000$_{hex}$
Arithmetic overflow	8000 0180$_{hex}$.

* The address are separated by 32 bits.

* We can perform the processing required required for exceptions by adding a few extra registers and control signals.

EPC:- A 32-bit register used to hold the address of the affected instruction.

Cause register: A register used to record the cause of the exception. In the MIPS architecture this register is 32 bits, although some bits are currently unused.

Exceptions in Pipelined Implementation.

A pipelined Implementation treats exceptions as another form of control hazards.

→ * To flush the instruction in IF stage nop is used.

→ * To flush the instruction in ID stage multiplexor is used.

→ * To flush the instruction in EX phase a new signal called EX is used.

* Flush causes new multiplexors to zero the control lines.

* Many exceptions require that we eventually complete the instruction that caused the exception as if it executed normally.

* The easiest way to do this is to flush the instruction and restart it from the begining after the exception is handled.

* The EPC → capture the address of the interrupted instructions and the MIPS cause register records all possible exceptions in a clock cycle.

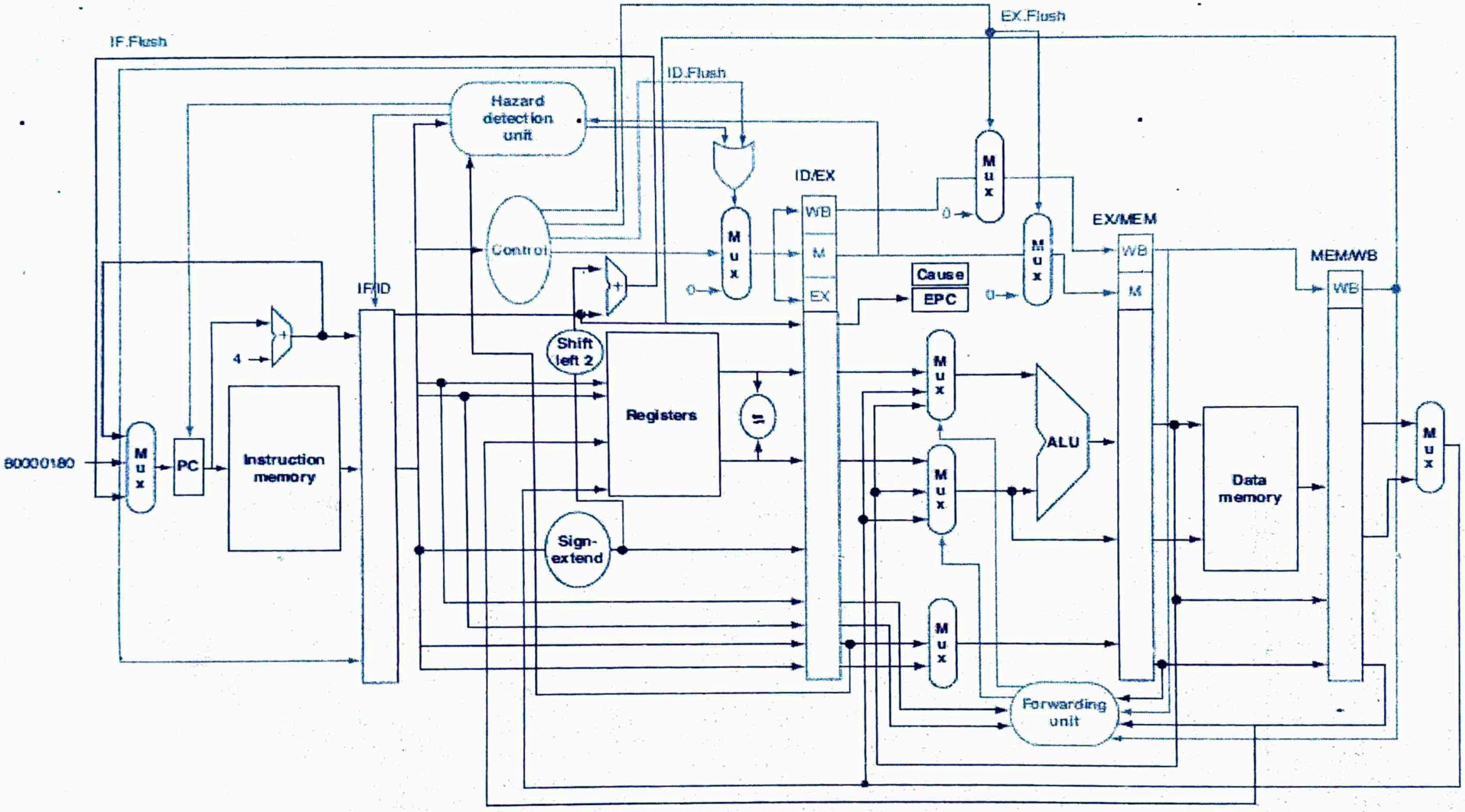


Figure 4-66 : **Exception Handling** (Data path with Control to handle Exception)

MIPS Data path with control path UNIT-III

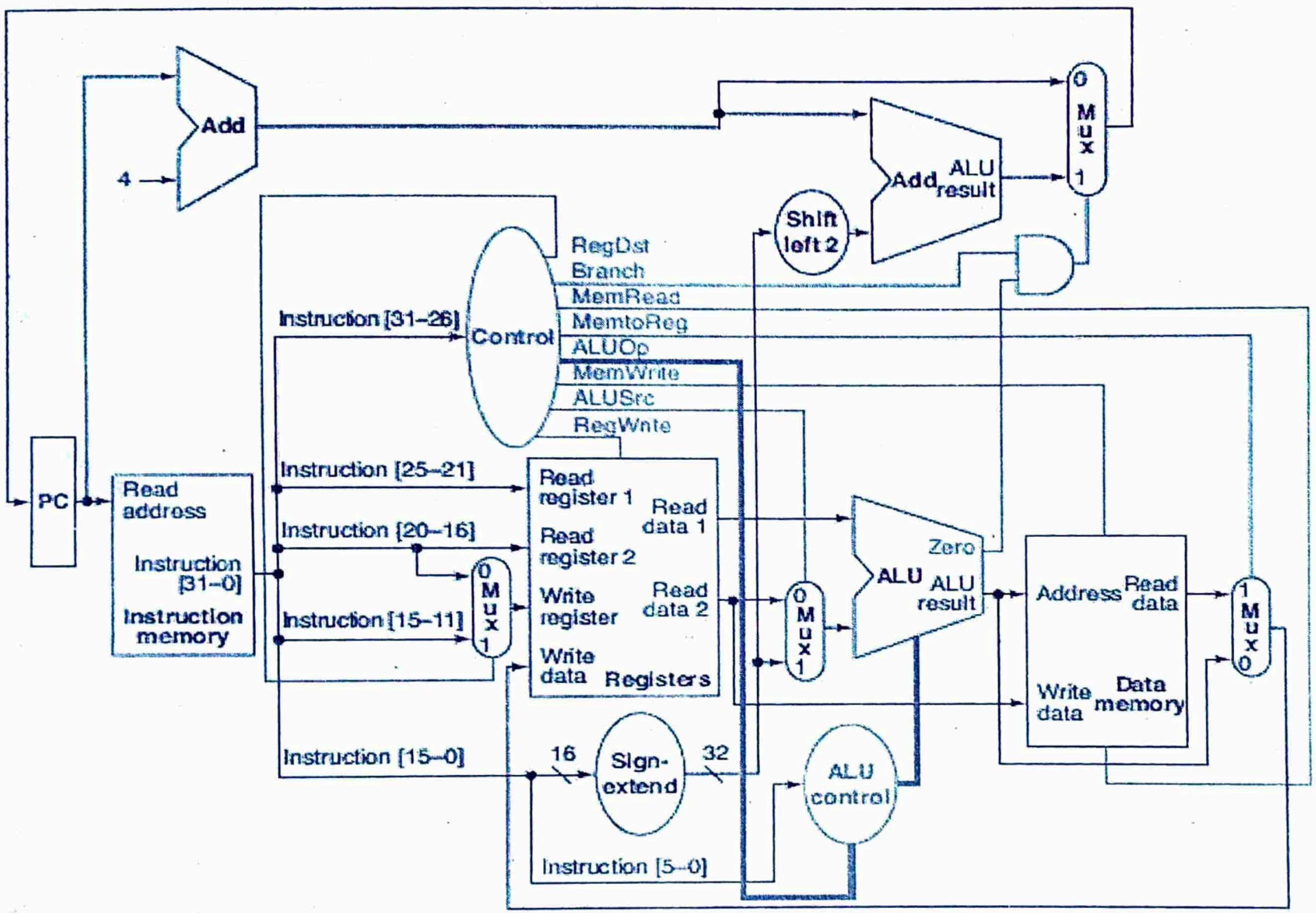


FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

Pipelined (datapath + control path)

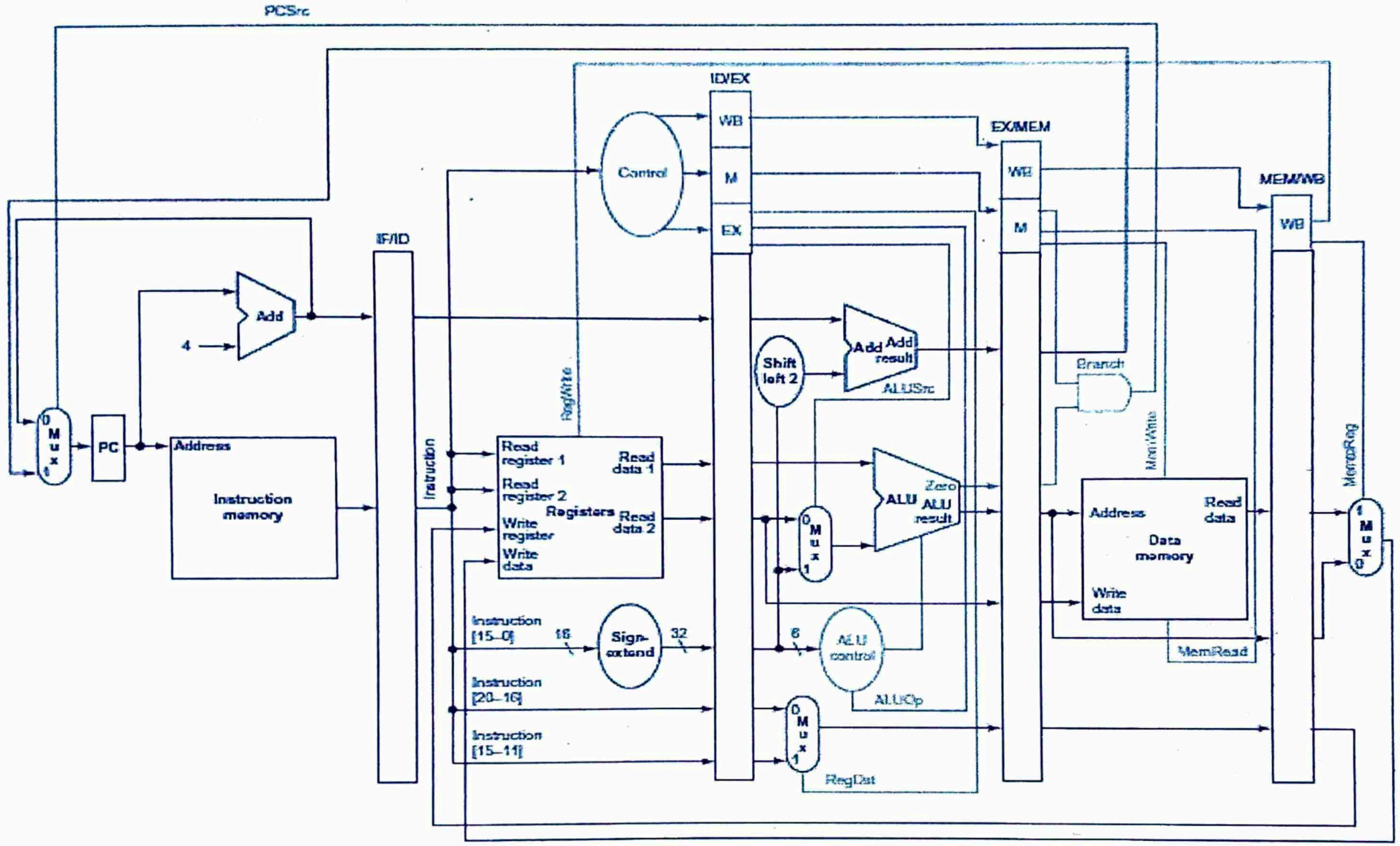


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Exception (data path + control) (37)

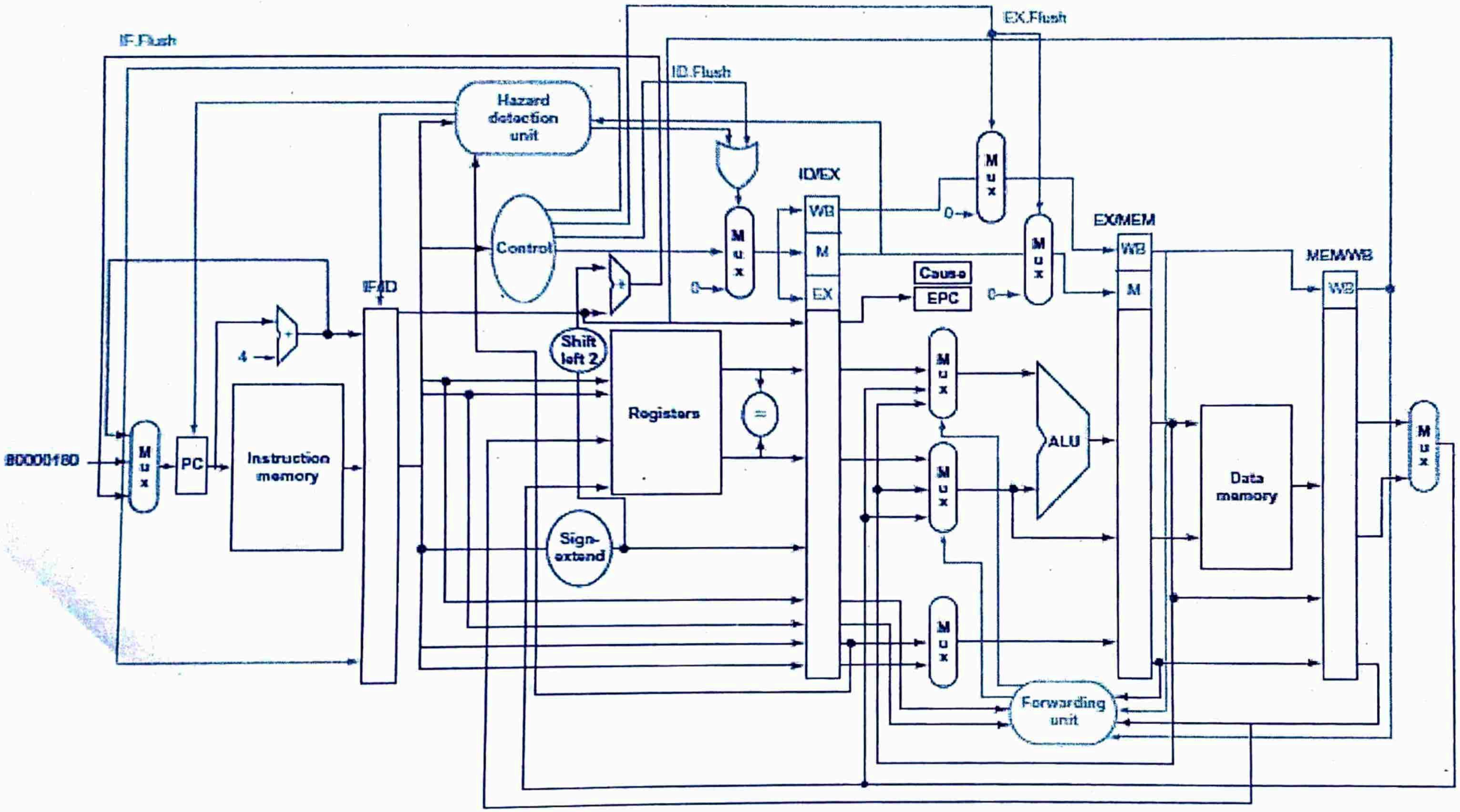


FIGURE 4.66 The datapath with controls to handle exceptions. The key additions include a new input with the value 8000 0180$_{\text{hex}}$ in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the instruction that caused the exception. The 8000 0180$_{\text{hex}}$ input to the multiplexor is the initial address to begin fetching instructions in the event of an exception. Although not shown, the ALU overflow signal is an input to the control unit.

Hazards (data Hazard (38) & control Hazard)

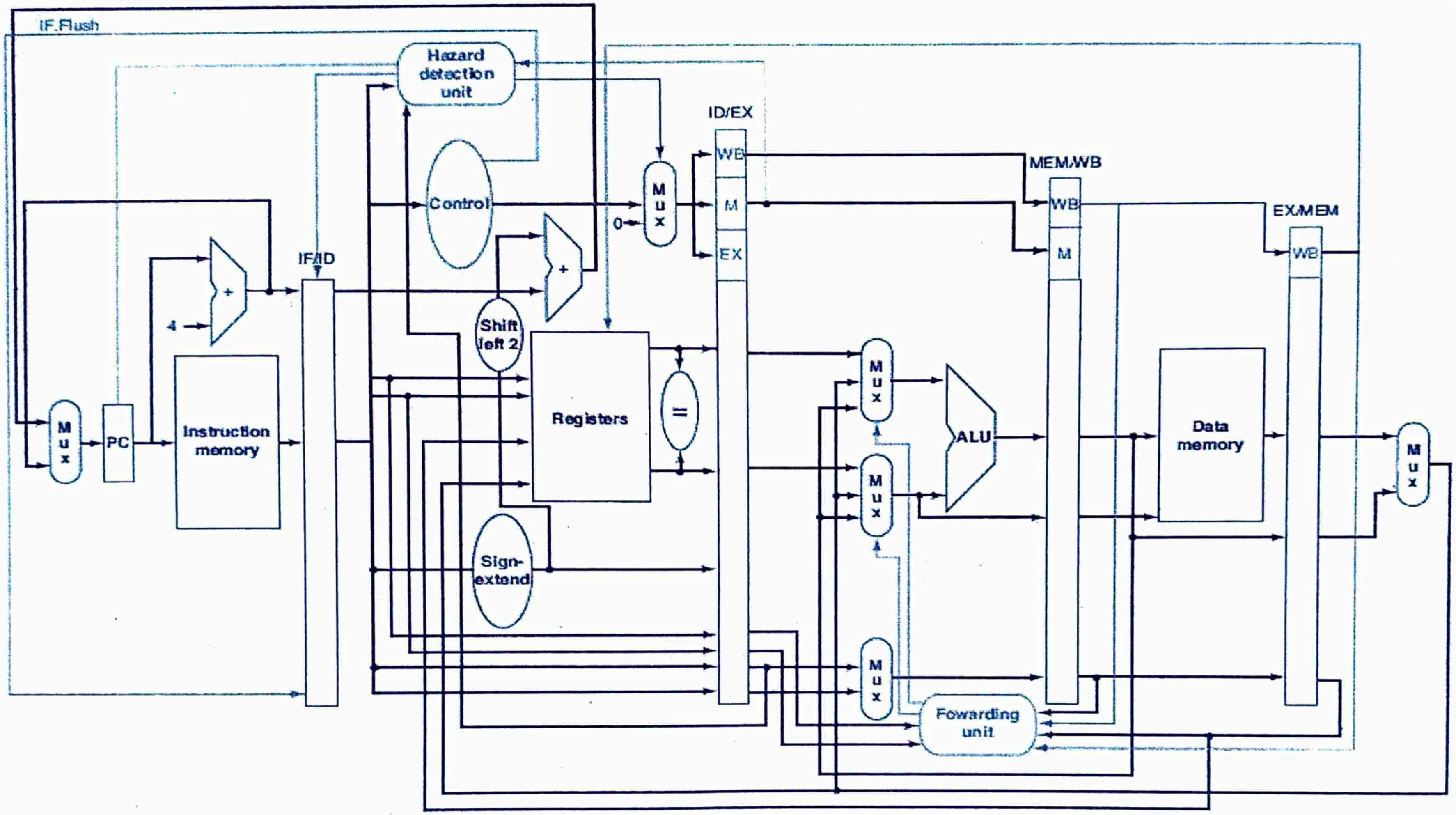


FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

Hazards (data Hazard & control Hazard)

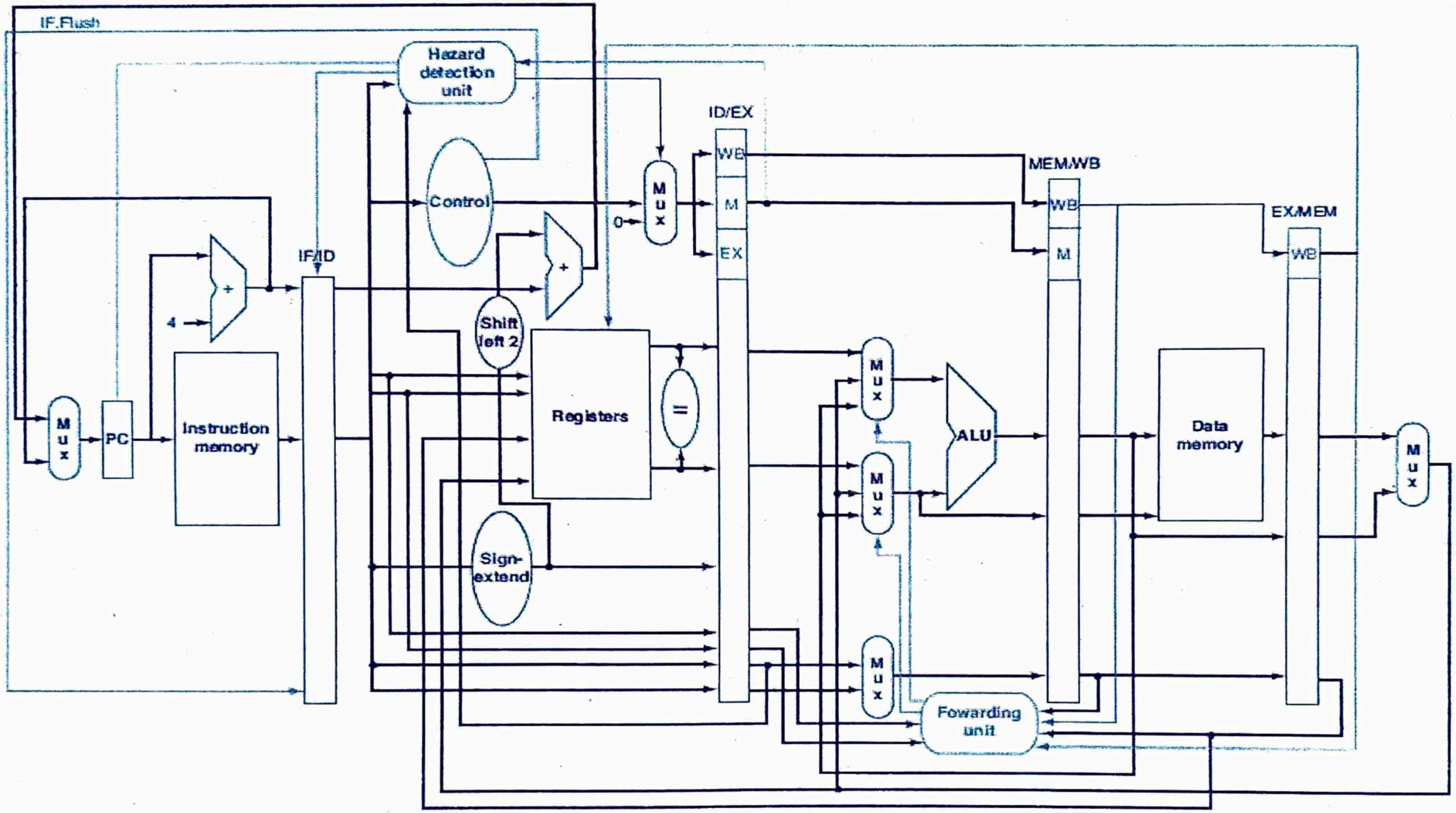


FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

UNIT IV

Parallelism

1) Parallel Processing challenges

Goals of parallelism:

*) parallelism speeds up the computer processing capability.

*) It increases throughput by making two or more ALU in CPU.

*) It improves the performance of the computer for a given clock speed.

Types of parallelism:

i) Instruction level parallelism
ii) Task level parallelism
iii) Bit-level parallelism
iv) Data level parallelism
v) Transaction level Parallelism

Approaches to exploit ILP:

2 approaches to exploit ILP are

i) Dynamic hardware intensive approach

example

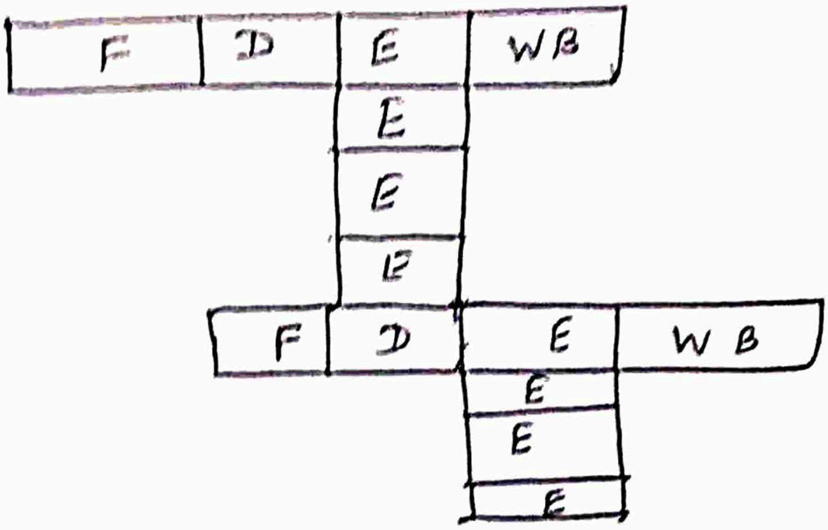
→ Athlon
→ Pentium III and 4
→ Alpha 21264

ii) Static compiler intensive approach:

Example:

→ IA-64 architecture

→ Intel's Itanium



Dynamic Multiple issue - processor:

→ Hardware decides which instruction execute together

→ complex hardware

→ Simple compiler

→ Dynamic multiple-issue processors are also known as super scalar processors

Simple Example to avoid data Hazard:

```
lw     $t0, 20($s2)
addu   $t1, $t0, $t2
 sub   $s4, $s4, $t3
slti   $t5, $s4, 20
```

Dynamic pipeline scheduling :

*) It is divided into 3 major units

i) Instruction fetch and issue unit
ii) Multiple functional units
iii) Commit unit

*) Each functional unit has buffers called reservation stations, which hold the operands and the operation.

*) As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated.

Loop Unrolling :

```
for (i = 1 ; i <= 1000 ; i = i+4)
    {
      x[i] = x[i] + y[i] ;
      x[i+1] = x[i+1]+ y[i+1];
      x[i+2] = x[i+2] + y(i+2];
      x[i+3] = x[i+3] + y[i+3];
    }
```

*) Such techniques work by unrolling the loop either

→ Statically by the compiler
→ Dynamically by the hardware.

2) PARALLEL PROCESSING CHALLENGES:

*) There are three fundamental ways of improving the performance of the application using concurrency:

i) Reduce latency
ii) Hide Latency
iii) Increase throughput

i) Data Distribution:

*) Data locality may occur

→ due to each processor having its own distinct local memory.

→ due to processor-specific caches as in a shared memory system.

ii) Inter-process communication

Inter process communication is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in a operating system

Factors to consider IPC:

i) cost of communications
ii) Latency Vs Bandwidth
iii) visibility of communication
iv) Synchronous Vs. asynchronous communication

iii) Load balancing:

*) Load balancing refers to the process of distributi approximately equal amount of work among tasks so that all tasks are kept busy all the time.

Steps for Achieving load balancing:

i) Equally partition the work each task receives

ii) Use dynamic work assignment.

iv) Speed up challenge

$$\left.\begin{array}{c}\text{Execution time}\\ \text{after}\\ \text{improvement}\end{array}\right\} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \begin{array}{c}\text{Execution}\\ \text{time}\\ \text{unaffected.}\end{array}$$

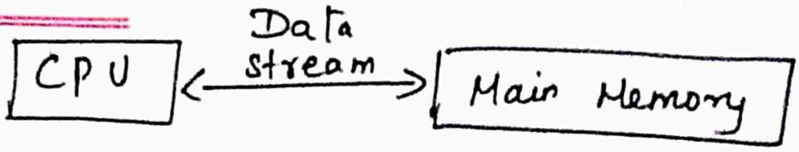
This is amdahl's ~~way~~ Law

Modified Amdahl's law:

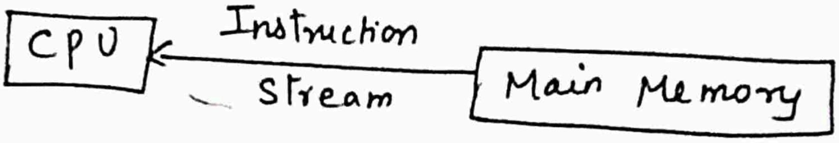
$$\left.\begin{array}{c}\text{Speed}\\ \text{up}\end{array}\right\} = \frac{\text{Execution time before}}{\left(\begin{array}{c}\text{execution time}\\ \text{before}\end{array} - \begin{array}{c}\text{Execution time}\\ \text{affected}\end{array}\right) + \frac{\text{execution time affected}}{100}}$$

3) FLYNN'S CLASSIFICATION:

Data stream:



Instruction stream:



Categories of Flynn's classification:

i) SISD – Single Instruction Single Data Stream

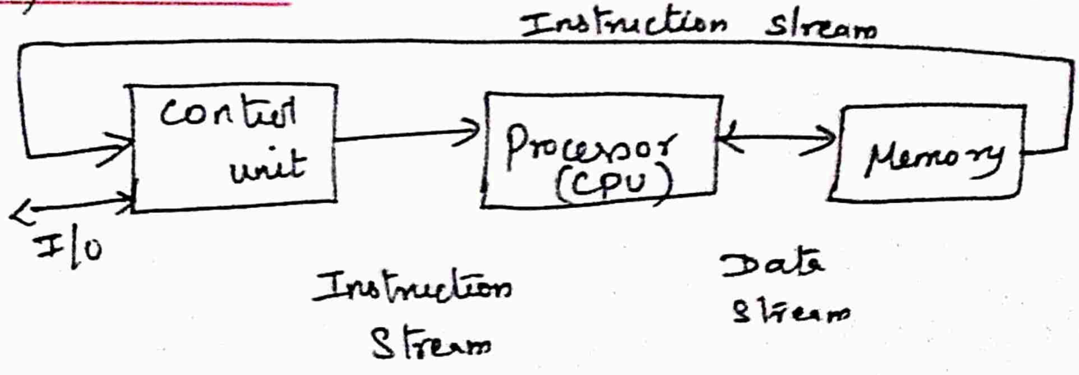
ii) SIMD – Single Instruction Multiple Data Stream

iii) MISD – Multiple Instruction Single Data Stream.

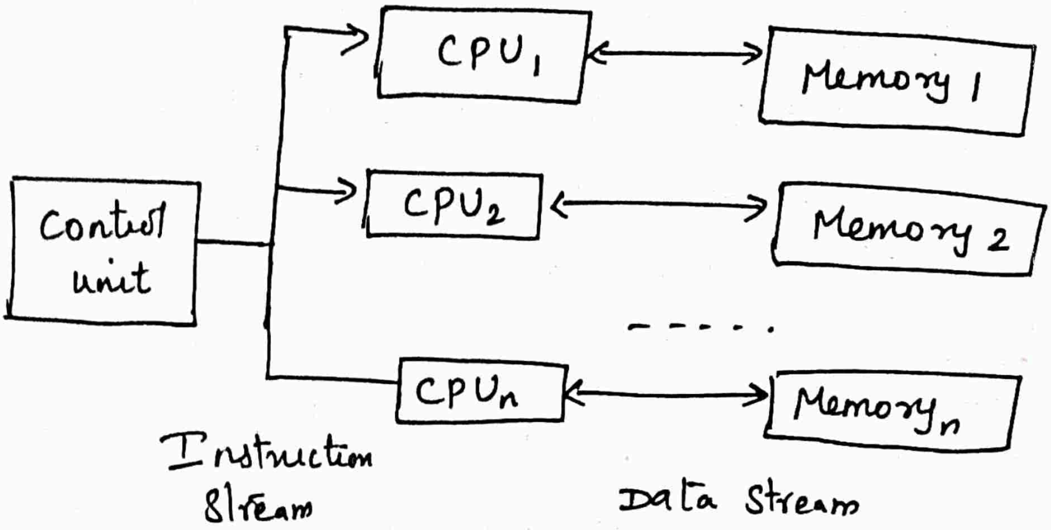
iv) MIMD – Multiple Instruction Multiple Data stream.

		Instruction stream	
Data stream		Single Instruction	Multiple Instruction
	Single Data	SISD	MISD
	Multiple Data	SIMD	MIMD

i) SISD:



ii) SIMD :



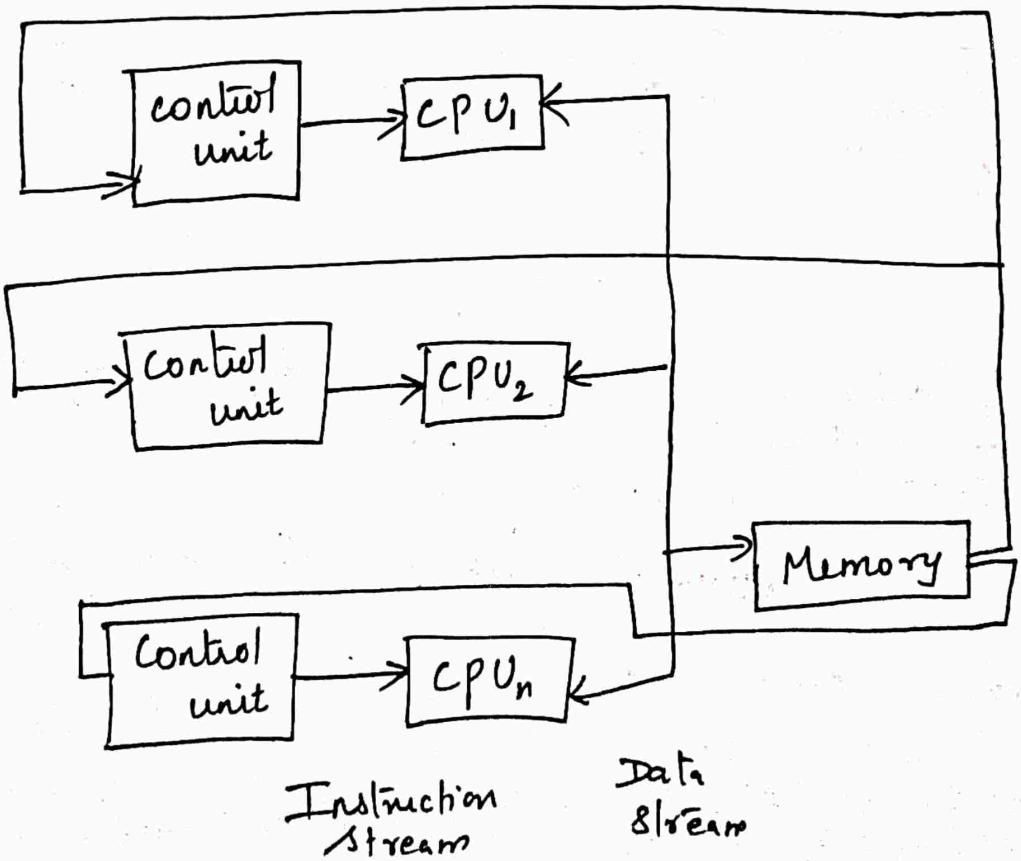
Advantages of SIMD system:

→ Reduces the cost of control unit.

→ Needs only one copy of code.

→ Used for multimedia extension

iii) MIMD :



Example:

→ Pipeline Architecture

Limitation :

→ Low level of parallelism
→ High Bandwidth required
→ High complexity

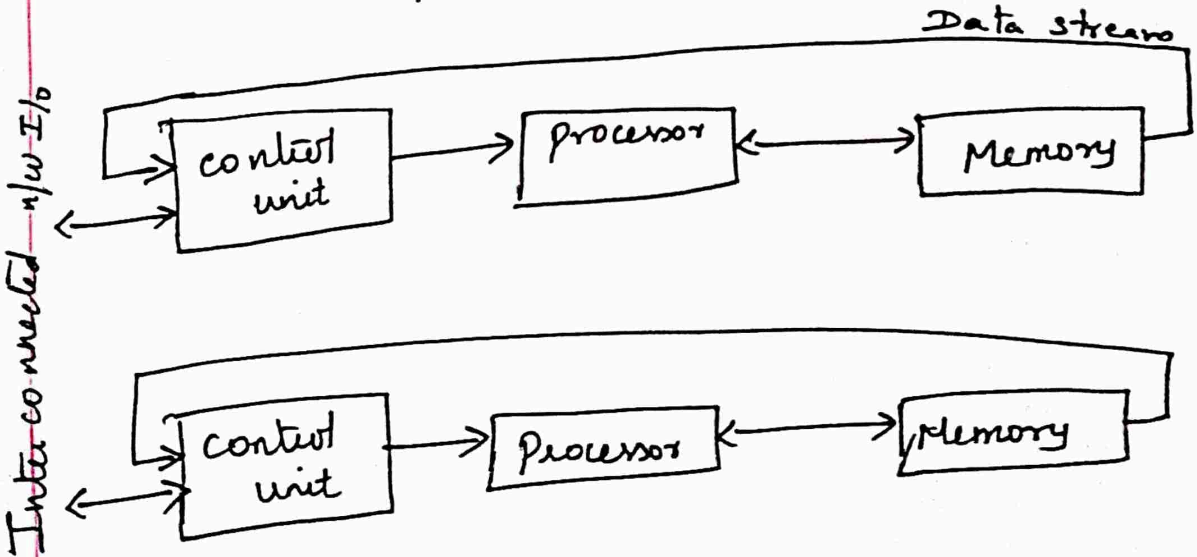
iv) MIMD :

characteristics of MIMD system:

→ Each instruction operates on different data.
→ Each processing elements are associated data memory.

Example:

→ Distributed computing system
→ IBM 370/168M
→ CRAY XMP



*) The processors work on their own data with their own instructions.

4) Vector Architecture

*) An older interpretation of SIMD is called a vector architecture which has been closely identified with cray computers.

*) It is again a great match to problems with lots of data-level parallelism

*) Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors, the vector architectures pipelined the ALU to get good performance at lower cost.

Properties of vector instructions:

→ A single vector instruction is equivalent to executing an entire loop.

→ In a vector instruction, the computation of each result in a vector is independent of the computation of other results in the same vector.

→ vector architectures and compilers help to write efficient applications.

→ The cost of the latency to main memory is seen only once for the entire vector.

→ As an entire loop is replaced by a vector instruction whose behavior is predeterme

→ control Hazards would normally arise from the loop branch are non-existent.

5) HARDWARE MULTITHREADING:

*) Allows multiple threads to share the functional units of a single processor in an overlapped fashion

Use of hardware multithreading:

→ Promote utilization of existing hardware resources.

Need of hardware multithreading:

→ To tolerate latency

→ To improve system throughput

→ To reduce context switch property

Three implementations

→ coarse grained Multi threading

→ fine geained Multi threading

→ Simultaneous Multi threading

i)Coarse grained Multi threading :

Advantages :

- → reduces the penalty of cost high stalls.
- → Less likely slow down.
- → Relieves need to have very fast thread-switching.

Dis advantages

- → Difficult to overcome throughout losses due to shorter stalls.
- → when stall occurs, pipeline should be emptied.

ii) Fine - grained Multi threading :

*) switches between thread after every instruction. such that no two instructions from the thread are in the pipe line concurrently

Advantage :

- → No need for dependency
- → No need for branch prediction logic.
- → Bubble cycles are used for executing useful instruction
- → Improved system throughput
- → It hides throughput losses due to short and long stalls.

Disadvantages:

→ Extra hardware complexity

→ Reduced single thread performance.

→ Resourse contention between threads in caches and memory.

→ dependency checking logic between threads remains (load/store).

iii) Simultaneous Multi threading (SMT)

*) It exploits the following features of modern processors:

→ Multiple functional units

→ Register renaming and dynamic scheduling

Advantage of SMT:

→ More functional unit, parallelism available than single threading

→ Register renaming + dynamic scheduling are used.

→ Multiple instructions from independent threads can be issued without regards to dependences among them.

→ SMT is a variation on multithreading that uses resources of a multiple-issue dynamically scheduled processors.

Design challenges :

* The design challenges of SMT processor include the following:

→ Larger register file needed to hold multiple contexts.

→ Not affecting clock cycle time

→ instruction issue

→ instruction completion

→ Ensuring that cache and TLB conflicts generated by SMT do not degrade performance.

Comparison of the SMT processor to the base super scalar processor:

→ Utilization of functional units

→ Utilization of fetch units

→ Accuracy of batch predictor

→ Hit rates of primary caches

→ Hit rates of secondary caches

Performance Improvement:

*) The key to maximize SMT performance is to share the following

→ Issue slots

→ Functional units

→ Renaming registers.

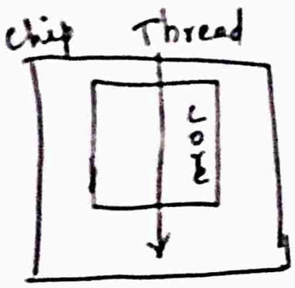
Superscalar processor

→ A superscalar processor with no multithreading support.

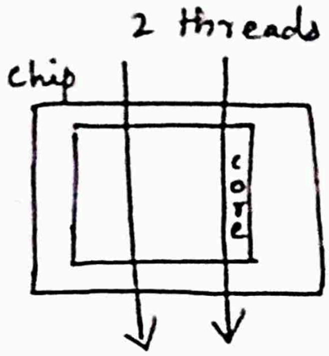
→ A superscalar processor with coarse-gain multithreading.

6) Multicore processors and other shared Memory multiprocessors:

MULTICORE PROCESSORS:



one chip,
one core,
One Executing thread



processors with hyper-threading technology, one chip, one core, two executing threads.

The issue slot usage is limited by the following factors:

→ imbalance in the resources needs.

→ resource availability over multiple thread.

→ Number of active threads considered

→ Finite limitations of buffer.

→ Ability to fetch enough instructions

Multi-core processors have multiple execution cores on a single chip:

Multi core processors may have

→ Two core

→*) Dual-core cpu.

→ Three core

*) Tri-core CPU

*) Example AMD phenom II X_3

→ Four core

*) Quad-core CPU

*) Example AMD phenom II X_4

→ six core

*) Hexa-core CPU

*) intel i5 and i7 processors

→ Eight core

*) octo-core CPU

*) AMD FX-8350

→ Ten core

*) Example intel Xeon E7-2850

*) In these designs, each core has its own execution pipeline. And each core has the resources required to run without blocking resources needed by other S/w threads.

Applications:

*) Multi-core processors are widely used across many application domain including

→ General-purpose

→ Embedded

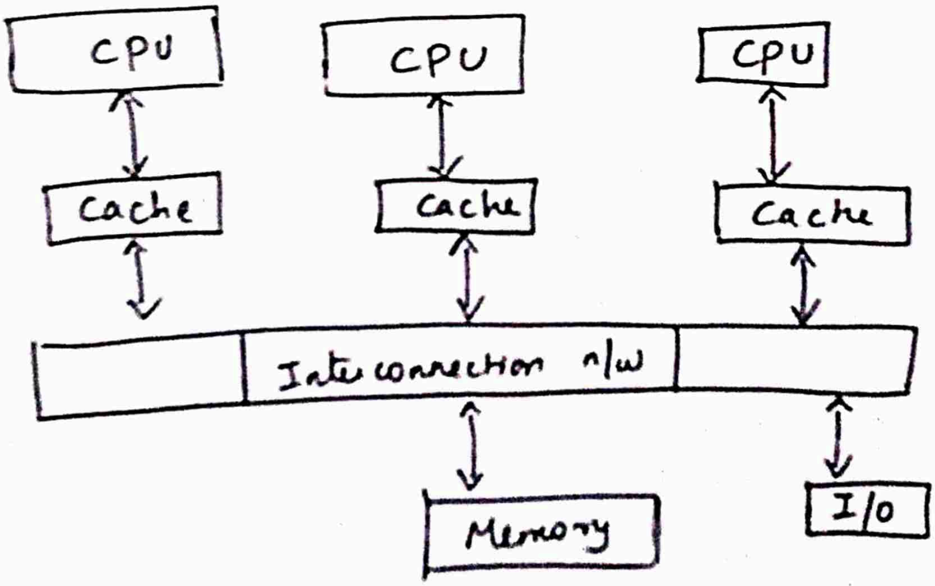
→ Network

→ DSP

→ Graphics.

7) Shared Memory Multi processor

i) Symmetric shared memory inter core Communication method:



*) Multi core processor implements multiprocessing in a single physical package that can be implemented by

→ Distributed shared memory

→ Symmetric shared memory

Types of memory access for single address space:

Types of Memory access:

i) Uniform Memory Access

ii) Non uniform Memory Access

ii) Distributed shared memory inter core

Message passing Method:

*) Communication between multiple processors is done by explicitly sending and receiving information.

Routines for communication:

→ Send message routine

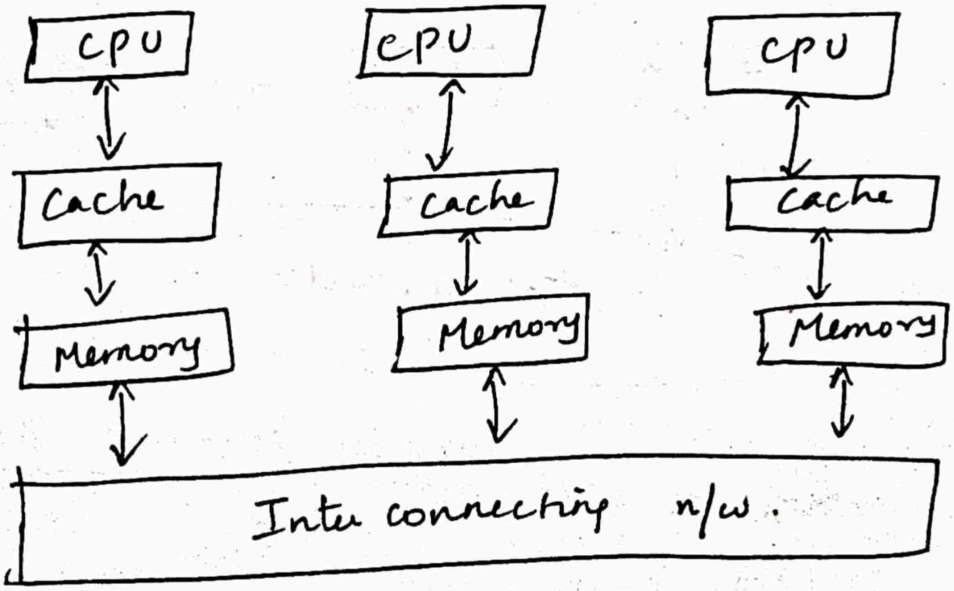
→ Receive Message routine

Advantage:

*) Lower latency which leads to better communication

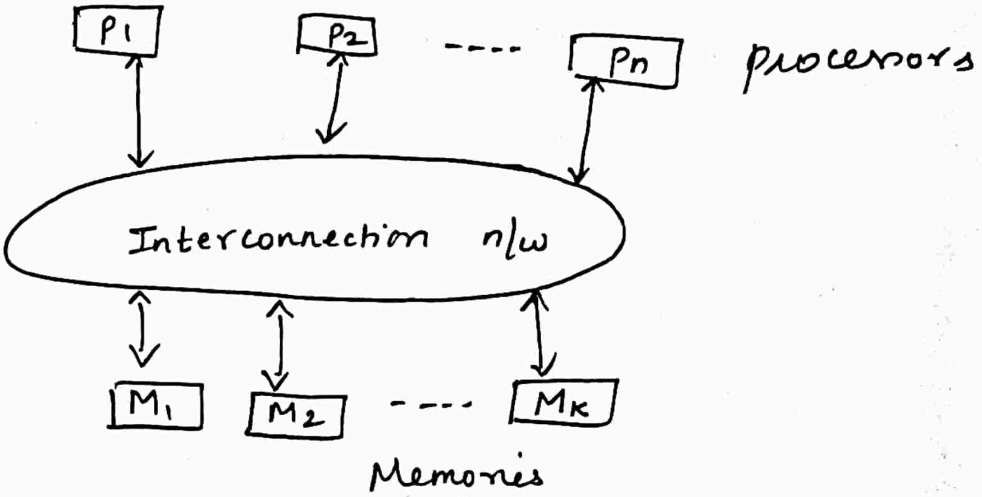
*) More efficiency

*) Enhanced performance.



Uniform memory Accen (UMA) Multiprocessor

*) A system which has the same network latency for all accesses from the processors to the memory modules is called a Uniform Memory Accen (UMA) multiprocessor



Non- Uniform Memory Accen (NUMA) Multiprocessors:

*) The network latency is avoided when a Processor makes a request to access its local memory.

*) However, a request to access a remote memory module must pass through the network.

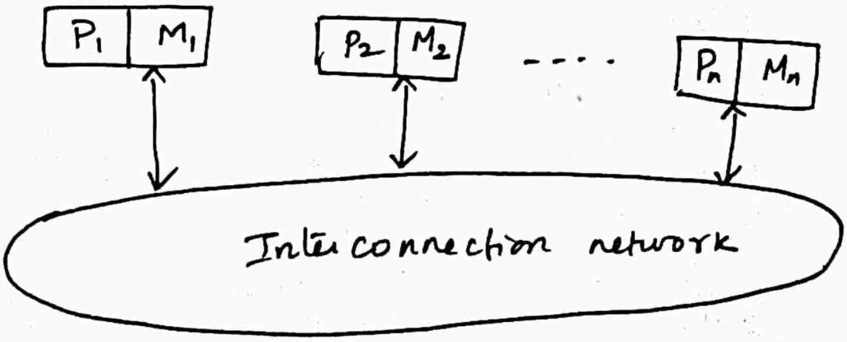


Fig: Non - uniform Memory Accen (NUMA)

Interconnection Network:

*) Interconnection network must allow information transfer between any pair of nodes in the system.

Ring:

*) A ring network is formed with point-to point connections between nodes.

→ Single ring

→ Hierarchy of rings

Hierarchy of rings:

*) It is a two-level hierarchy

*) The upper level ring connects the lower - level ring

*) The average latency for communication b/w any two nodes on lower level rings is reduced with this arrangement.

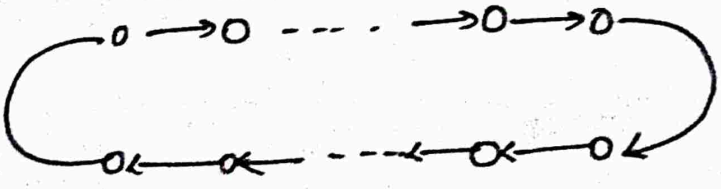


fig: Single ring

The upper level ring may become a bottleneck when many nodes on diff lower level rings communicate with each other frequently.

Hierarchy of rings

*) A ring network is formed with point to point connection between nodes.

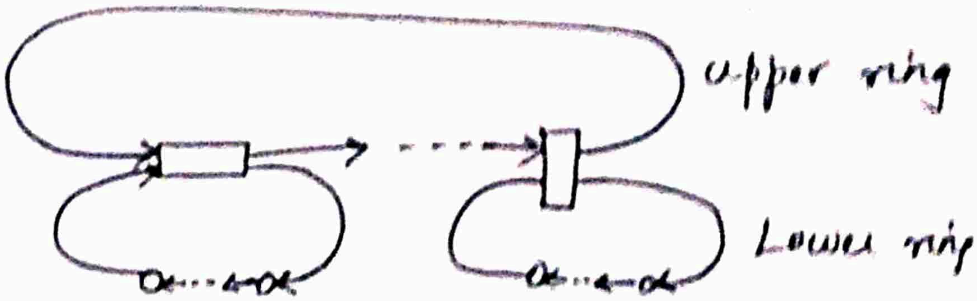


Fig: hierarchy of rings

Cross Bar:

*) A crossbar is a network that provides a direct link between any pair of units connected to the network.

*) For example, we can implement the structure using a crossbar that comprises a collection of swithches for n processors and k memories nxK switches are needed

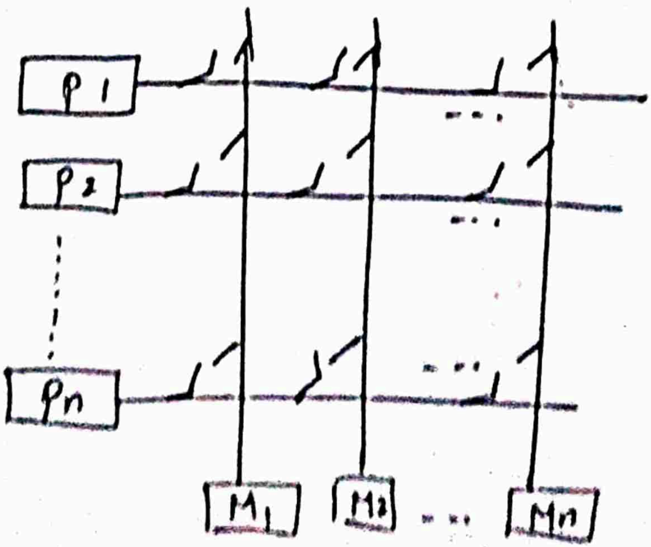


Fig: cross bar inter connection network.

8) Introduction to Graphics processing Unit

*) The increasing demands of processing for computer graphics has led to the development of specialized chip called graphics processing units (GPU).

*) The primary purpose of GPU is to accelerate the large number of floating point calculations needed in high-resolution 3D graphics, such as in video games.

*) A separate controlling program runs in the general purpose processor of the host computer and invokes the GPU program when necessary.

*) Before initiating the GPU computation, the pgm in the host computer must first transfer the data needed by the GPU program from the main memory into the dedicated GPU memory.

*) The processing cores in a GPU chip have a specialized instruction set and hardware architecture, which are different from those used in general purpose processor.

GPU vary from CPU:

*) Allow them to dedicate all their resources to graphics.

*) Some tasks poorly or not at all.

*) The GPU problems sizes are typically hundred of megabyte to gigabyte, but not hundreds of gigabytes to terabytes.

Different style of architecture:

*) Perhaps the biggest difference is that GPU do not rely on multilevel caches to overcome the long latency to memory.

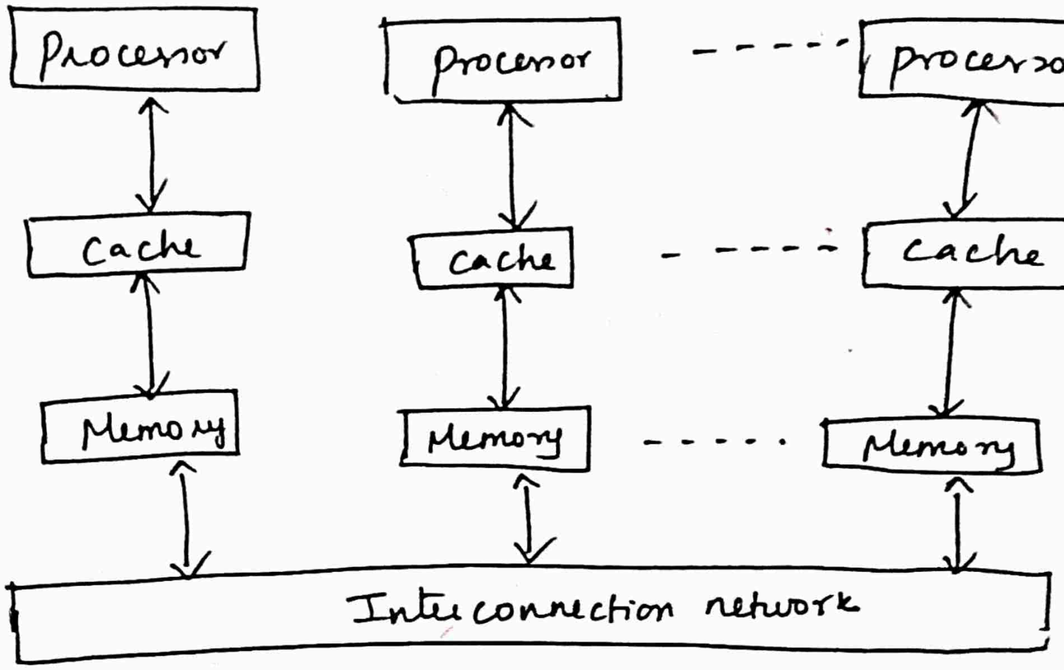
*) GPU rely on hardware multithreading to hide the latency to memory.

*) That is, between the time of a memory request and the time that data arrives, the GPU executes hundreds or thousands of threads that are independent of that request.

*) GPU typically have 4 to 6 GIB or less, while CPU have 32 to 256 GIB. Finally keep in mind that for general purpose computation, you must include the time to transfer the data b/w CPU memory and GPU memory.

9) Clusters :

*) The alternative approach to sharing an address space is for the processors to each have their own private physical address space.



Message passing Multiprocessor :

*) The classic organization of a multiprocessor with multiple private address space are called as Message-passing multiprocessor.

*) This alternative multiprocessor must communicate via explicit message passing, which traditionally is the name of such style of computers.

10) Warehouse - Scale Computers.

*) Internet services required the construction of new buildings to house, power, and cool 100,000 servers.

*) They act as one giant computer and cost on the order of $150M for the building, the electrical and cooling infrastructure, the servers and the networking equipments that connects and houses 50,000 to 1,00,000 servers. consider them a new class of computer called as warehouse - Scale computers (WSC).

*) The most popular framework for batch processing in a WSC is MapReduce and its open- source twin Hadoop-

*) Map runs on thousands of servers to produce an intermediate result of Key-value pairs.

*) Reduce collects the o/p of those distributed tasks and collapses them using another programmer - defined function.

*) One Map program calculates the number of occurances of every English word in a

Program wise large collection of document.

```
map ( string key, String value):
    // Key : document name
    // value : document contents for each word W
                                   in value:
  EmitIntermediate (W, "I");
    // Key : a word
    // values : a list of count int result =0;
      for each v in values:
      result + = ParseInt (v);
  Emit (AsString (result));
```

*) The function EmitIntermediate used in the Map function emits each word in the document and the value one.

*) Then the Reduce function sums all the values per word for each document using Pause Int () to get the number of occurances per word in all documents.

*) This time the target is providing Information technology for the world instead of high performance.

WSC have three major distinction :

i) Ample , easy parallelism :

*) A WSC architect has whether the applications in the targeted market place have enough parallelism to justify the amount of parallel hardware and whether the cost is too high for sufficient communication hardware to exploit this parallelism.

*) First, batch applications like MapReduce benefit from the large number of independent data sets that need independent processing, such as billions of web pages from a web crawl.

*) Second, interactive Internet service applications, also known as software as a service (saas), can benefit from millions of independent users of interactive internet services.

*) Reads and writes are rarely dependent is Saas, so saas rarely needs to synchronize. For example, search uses a read-only index and email is normally reading and

writing information.

*) We call this type of easy parallelism Reques-level parallelism, as many independent efforts can produce in parallel naturally with little need for communication or synchronization.

ii) operational costs count:

*) Traditionally, server architects design their systems for peak performance within a cost budget and worry about energy only to make sure they dont exceed the cooling capacity of their enclosure

*) They usually ignored operational costs of a server, assuming that they pale in comparison to produce costs.

*) WSC have longer life time.

*) The building and electrical and cooling infra-structure are often amortized over 10 or more years.

*) The operational costs adds up

→ energy

→ power distribution

→ cost of a WSC over 10 years.

iii) Scale and the opportunities:

*) To construct a single WSC, you must purchase 100,000 servers along with the supporting infrastructure, which means volume discounts.

*) Hence, WSC. are so massive internally that you get economy of scale even if there are not many WSC.

*) The flip side of the economic opportunity of scale is the need to cope with the failure frequency of scale.

UNIT V

MEMORY AND I/O SYSTEMS

1) Memory Hierarchy:

*) Memories are made up of registers in the memory in one storage location are called memory location.

$$\boxed{1 \text{ byte} = 8 \text{ bits}}$$

Key operations:

→ Read

*) Data is retrieved from the memory

*) Also called as fetch operation.

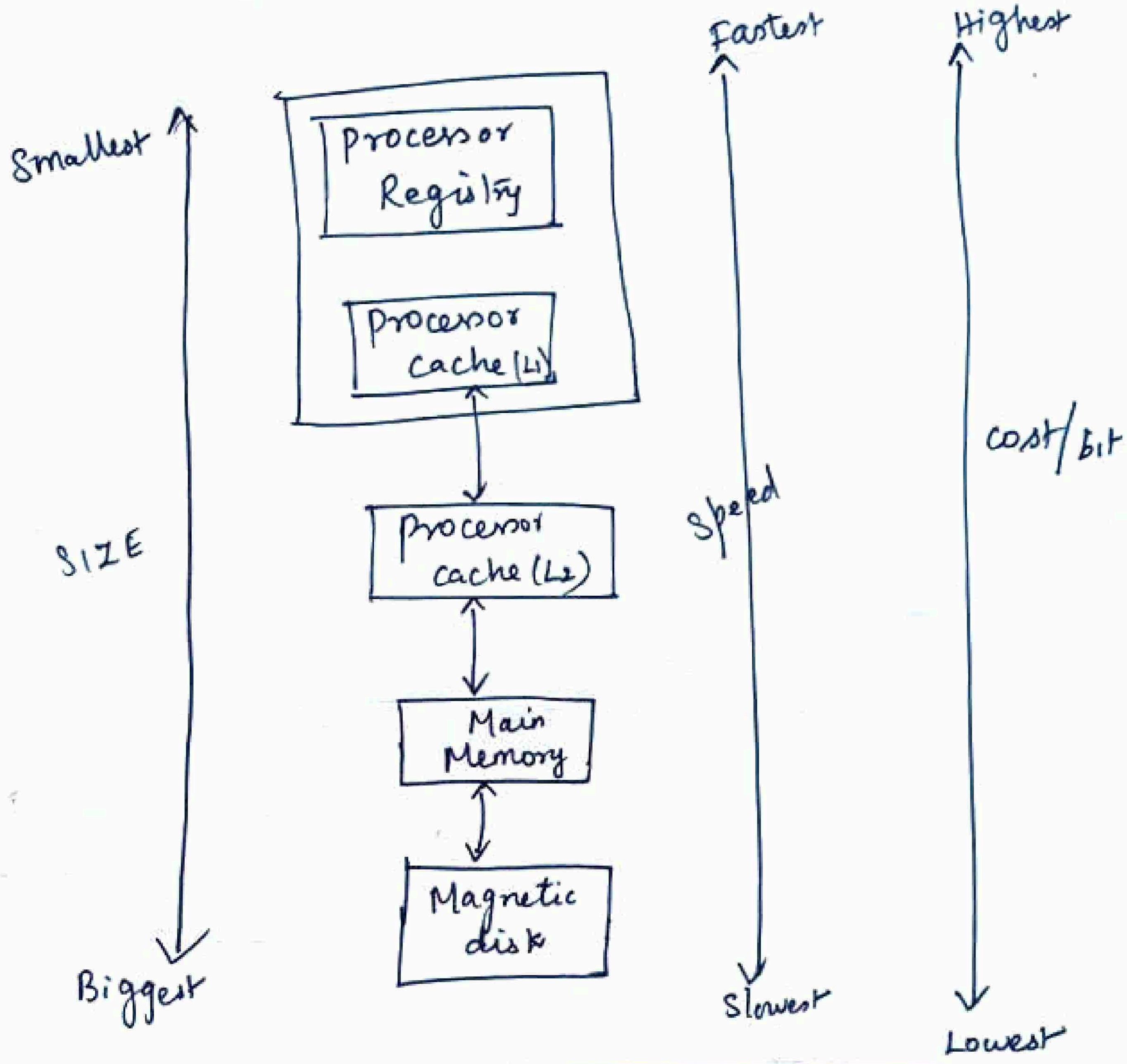
→ Write

*) Data is stored into the memory

*) Also called as store operation.

Hierarchy:

1. Processor Register
2. Primary cache
3. Secondary cache
4. Main Memory
   → Single in-line memory Module
   → Dual in-line memory module
   → Rambus in-line Memory Module
5. Secondary Memory
   → Magnetic disk



	Speed	Size	cost
Primary cache	High	lower	Low
Secondary cache	Low	Low	Low
Main Memory	Lower than Secondary	High	High
Secondary Memory	very Low	very high	very high

2) **Memory Technologies:**

There are 4 primary technologies

i) Static Random Accen Memory

ii) Dynamic Random Accen Memory

iii) ROM and Flash memory

iv) Magnetic disk.

i) static Random Accen Memory:

→ It is part of RAM.

→ Located Very close to cpu.

→ They are simply integrated circuits

→ single accen port

→ Have fixed accen time to any data

→ don't need to refresh.

→ use 6 to 8 Transistor.

→ Need Minimal Power.

ii) Dynamic Random Accen Memory:

→ In DRAM, the value is kept is a cell

→ Stored as a charge is a capacitor

→ single transistor used to accen this stored charge, either to read the value or to overcome the charge stored here.

→ Asynchronous DRAM

→ Synchronous DRAM

→ Rambus Memory

iii) ROM and Flash Memory:

*) There are diffeuent variations in non-volatile memory. They are

- *) Read only Memory (ROM)
- *) programmable ROM (PROM)
- *) Erasable PROM (EPROM)
- *) Electrically EPROM (EEPROM)
- *) Flash Memory

ROM:

→ Inbuilt in the computer.

→ non-volatile memory.

PROM:

→ Allow user to load programs & data.

→ It is more flexible and convenient.

→ PROM are faster

→ Less expensive.

EPROM:

→ Stored data can be erased.

→ New data can be loaded

→ while erasing, entire EPROM chip content is erased

→ If chip must be physically removed from the chip for re-programming the entire content is erased.

3) Cache Memory

*) The speed of main memory is very low when compared to the speed of modern processor.

*) A scheme that reduces the time needed to access the information from main memory is required. An efficient is to use a fast cache Memory.

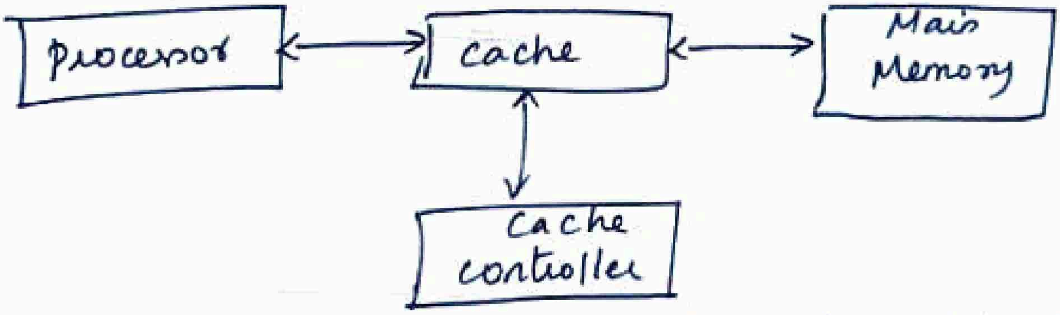
*) To execute a program in a computer system

→ pgm is loaded into main Memory

→ pgm inst is fetched by The processor

→ processor fetches the data from MM

→ Execute the program.



*) If the processor requests the data in cache, which is not available in cache it is referred as cache Miss.

*) cache controller decides which memory block should be moved in or out of cache and main memory.

Types of cache Memory:

i) Primary cache

→ referred as L1 cache.

⇛ Located in processor chip

ii) Secondary cache

→ referred as L2 cache

→ placed b/w L1 cache & MM

Advantages of cache:

→ Faster than Main Memory

→ Less access time

→ can be executed in short time

→ store data for temporary use.

Disadvantages of cache:

→ Limited capacity

→ very expensive

Block Replacement:

*) The fraction of memory accesses found in a level of the memory hierarchy is called as hit ratio or hit rate.

Write operations:

There are 2 ways for write operation

i) Write through

ii) Write back

iii) Write buffer.

Read Miss :

*) When the addressed word in a read operation is not in the cache, it is called as Read miss.

Write Miss :

*) During write operation, word is not in the cache, it is called as Write miss.

$$\boxed{\text{Miss rate} = 1 - \text{Hit rate}}$$

Average Memory Access Time (AMAT) :

$$\boxed{\text{AMAT} = \text{Time for BIT} + \text{Miss Rate} * \text{Miss Penalty}}$$

*) It is the average time to access memory consider both hits and misses and the frequenck of different accesses.

4) Measuring and improving cache performance:

*) Two techniques used to improve cache Performance

→Reducing miss rate

→ Reducing miss penality

CPU Execution time:

$$\left.\text{CPU execution time}\right\} = \text{CPU clock cycles} + \text{Memory stall clock cycle}$$

+ clock cycle time

Memory stall clock cycle :

*) The number of cycles during which the CPU is stalled waiting for a memory accen is called memory stall cycle.

*) Memory stall clock cycles is defined as the sum of read-stall cycle and write-stall cycle.

i) $$\left.\text{Memory stall clock cycles}\right\} = \text{Read - stallcycle} + \text{Write - stall cycles.}$$

ii). $$\left.\text{Read-stall cycles}\right\} = \frac{\text{Read}}{\text{program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

iii) $$\left.\text{Write -stall cycles}\right\} = \left(\frac{\text{Writes}}{\text{program}} \times \text{Write miss rate} \times \text{Write miss Penality}\right) + \text{write buffer stall.}$$

iv) $$\left.\begin{matrix}\text{Memory stall}\\\text{clock cycle}\end{matrix}\right\} = \frac{\text{Memory access}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

v) $$\left.\begin{matrix}\text{Memory stall}\\\text{clock cycles}\end{matrix}\right\} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Miss}}{\text{Inst.}} \times \text{Miss penalty}$$

Mapping Function:

*) There are three types of mapping functions

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

i) Direct Mapping:

*) It is the simplest method to determine cache location to store memory blocks.

*) The position of a memory block is

(BLOCK ADDRESS) MODULO (NUMBER OF BLOCKS IN THE CACHE)

*) To implement direct mapping the 16-bit memory address is divided into 3-fields

TAG (5)	BLOCK (7)	WORD (4)
MSB	16 bits	LSB

Example

$$\boxed{j = i \text{ modulo } m}$$

i → main memory block number

j → Cache block number

m → Number of blocks in cache.

ii) Associative mapping:

*) A cache structure in which a block can be placed in any location in the cache is called fully associative cache.

TAG (12)	WORD(4)
MSB	LSB

*) 12-tag bits are required to identify a memory block.

*) 4-word bits are used to select one of the 16-bits in a block.

iii) Set Associative Mapping:

*) A cache that has fixed number of location, where each block can be placed is called set-associative mapping.

*) Blocks of cache are grouped into sets.

In set-associative cache, the set containing a mag memory blocks is given by

$$\boxed{\text{(BLOCKNUMBER) MODULO (NUMBER OF SETS IN THE CACHE)}}$$

5) VIRTUAL MEMORY

*) Technique that automatically move program and data blocks into the physical main memory when they are required for execution are called virtual memory technique

*) The main memory can act as a "cache" for the secondary storage.

*) MMU - The memory management unit controls this virtual memory. It translates virtual addr into physical addr.

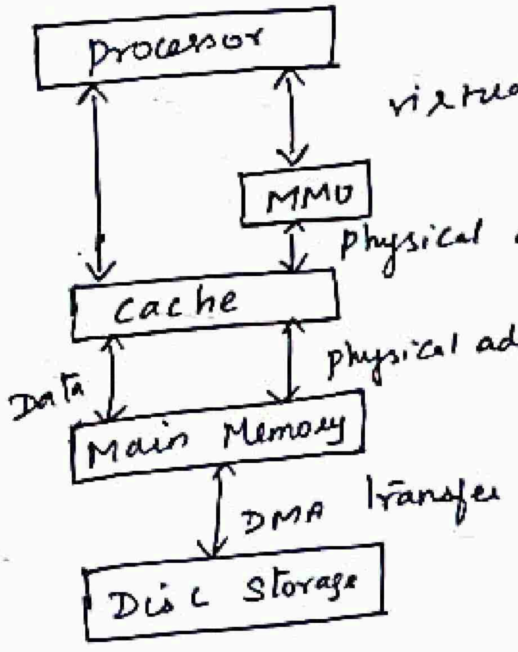


Fig: virtual Memory

Address Translation:

*) A virtual memory address translation method based on the concept of fixed-length.

*) Each virtual address has 2 fields

i) virtual page number

ii) offset

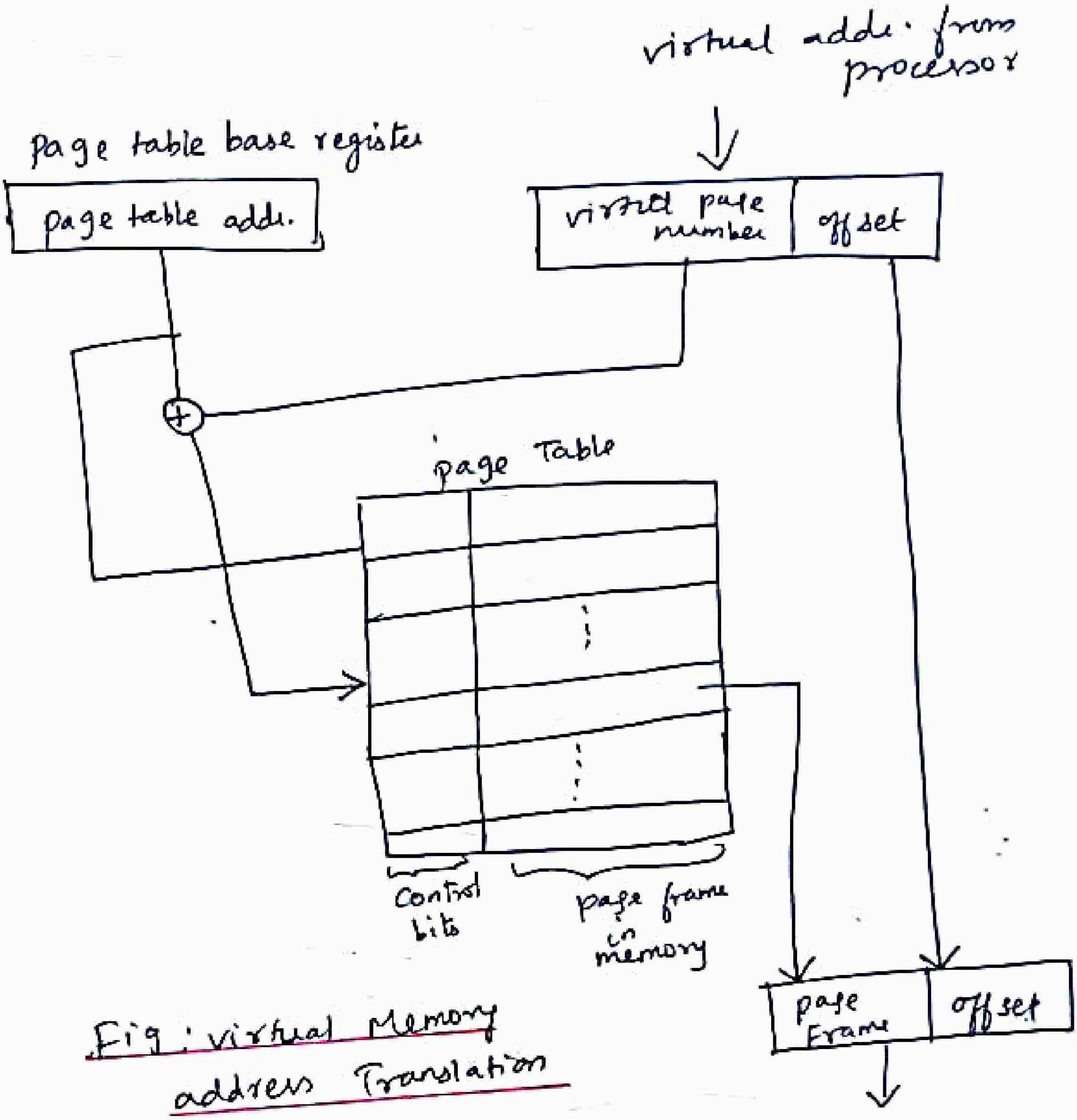


Fig : virtual Memory address Translation

Page Table :

*) The page table contains information about the main memory location of each page.

*) This information includes the main memory address where the page is stored and current status of the page.

*) Two important control bits are

i) valid bit

ii) Motify bit

6) Translation Look aside buffer (TLB)

*) virtual memory processor has to access page table is kept in the memory. To reduce the access time and degradation of performance, a small portion of page table is accomodated in MMU. This small cache (portion) is called Translation look aside buffer (TLB).

*) TLB contains virtual address of entry, in addition to the page table entry.

Following steps are used in address Translation process with TLB:

i) Processor generates the virtual address

ii) MMU looks in the TLB for the referenced page.

iii) If the page table entry. for this page is found in the TLB, the physical address is obtained immediately.

iv) If the page table entry is not found, then the corresponding page entry is obtained from the page table in the main memory.

v) TLB is updated accordingly,

→ page fault
→ page replacement
→ write operation

Page Fault:

*) When a program generates an address request to a page that is not available in the main memory, then a page fault is occurred.

*) If a valid bit for a virtual page is off, it causes a page fault. If page fault occurs then the control has given to operating system.

*) When the MMU detects a page fault it asks the OS to handle the situation by raising a exception.

*) The operating system copies the contents of the page from the disk into the main memory and transfers the control back to the interrupted task.

page replacement:

*) When a new page is brought from the disk if the memory is full, a page must be replaced from the main memory.

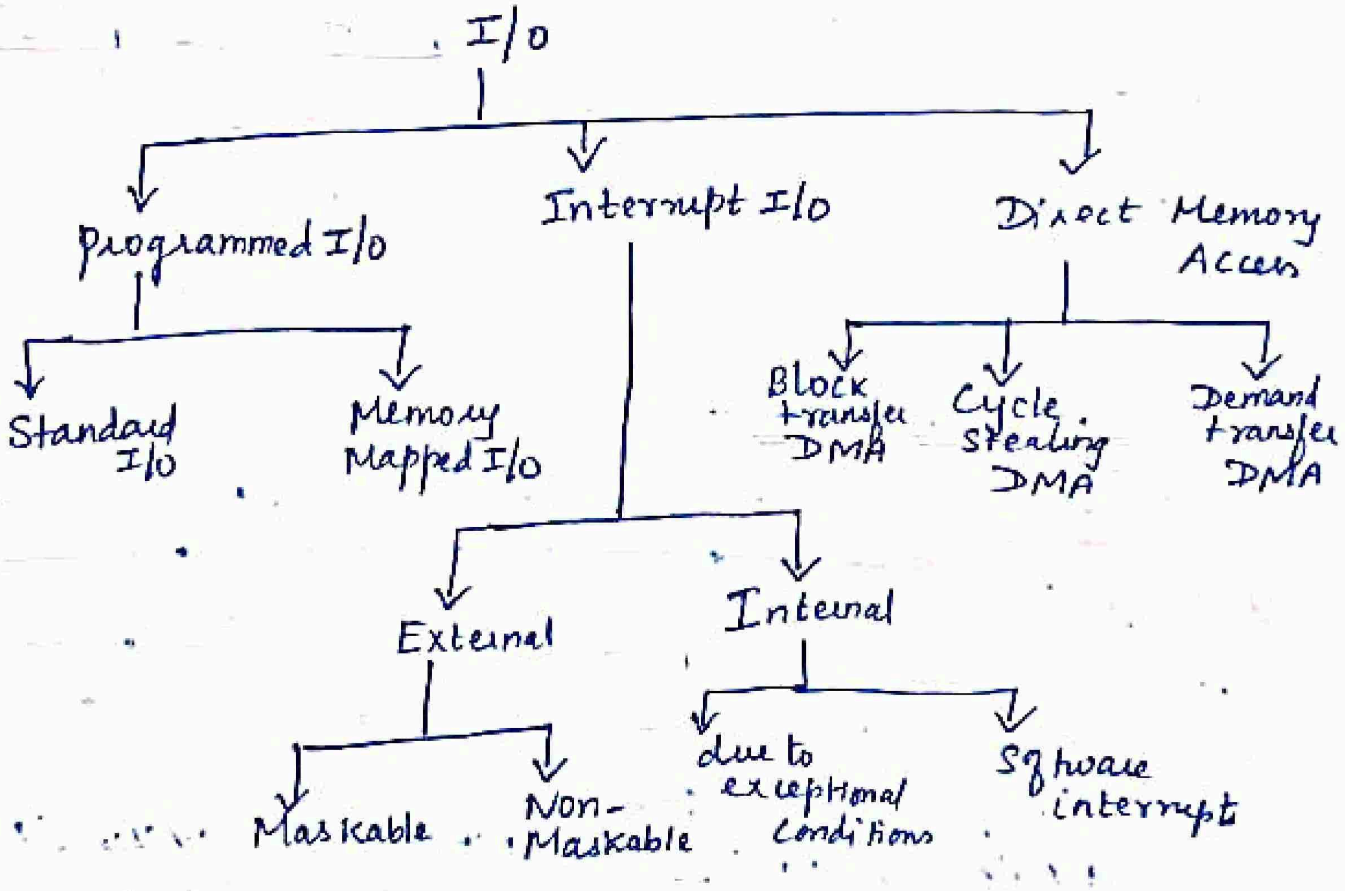
*) LRU replacement algorithms can be used for page replacement.

*) A modified page has to be written back to the disk before it is removed from the main memory.

7) Input / Output System:

*) The important component of any computer system are CPU, memory and IO devices.

*) The CPU fetches instructions from memory, processes them and stores results in memory. The other components of the computer system may be loosely called the Input / Output System



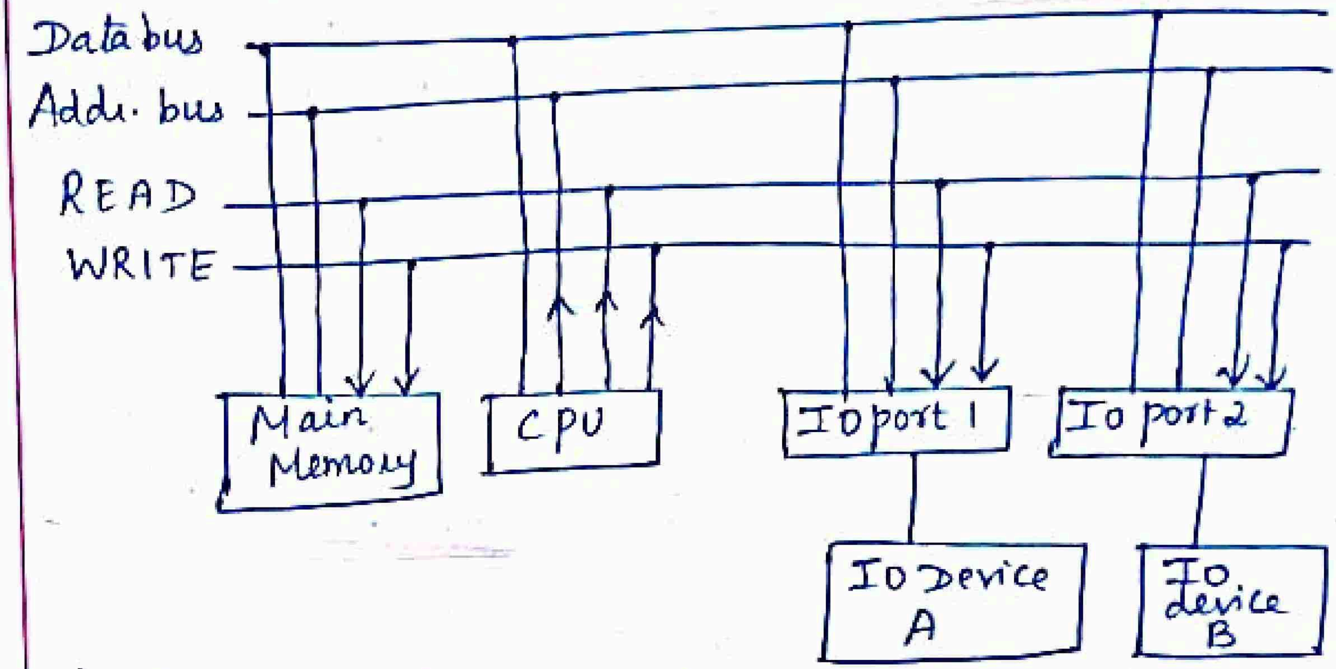
Programmed IO:

*) IO operations are distinguished by the extent to which the CPU is involved in their execution. If IO operations is said to be using Programmed IO.

→ Memory - Mapped IO

→ IO - Mapped IO.

i) Memory - Mapped IO :



*) The control lines READ & WRITE are activated by the CPU for processing memory reference instructions and Io instructions.

ii) Io - Mapped Io :.

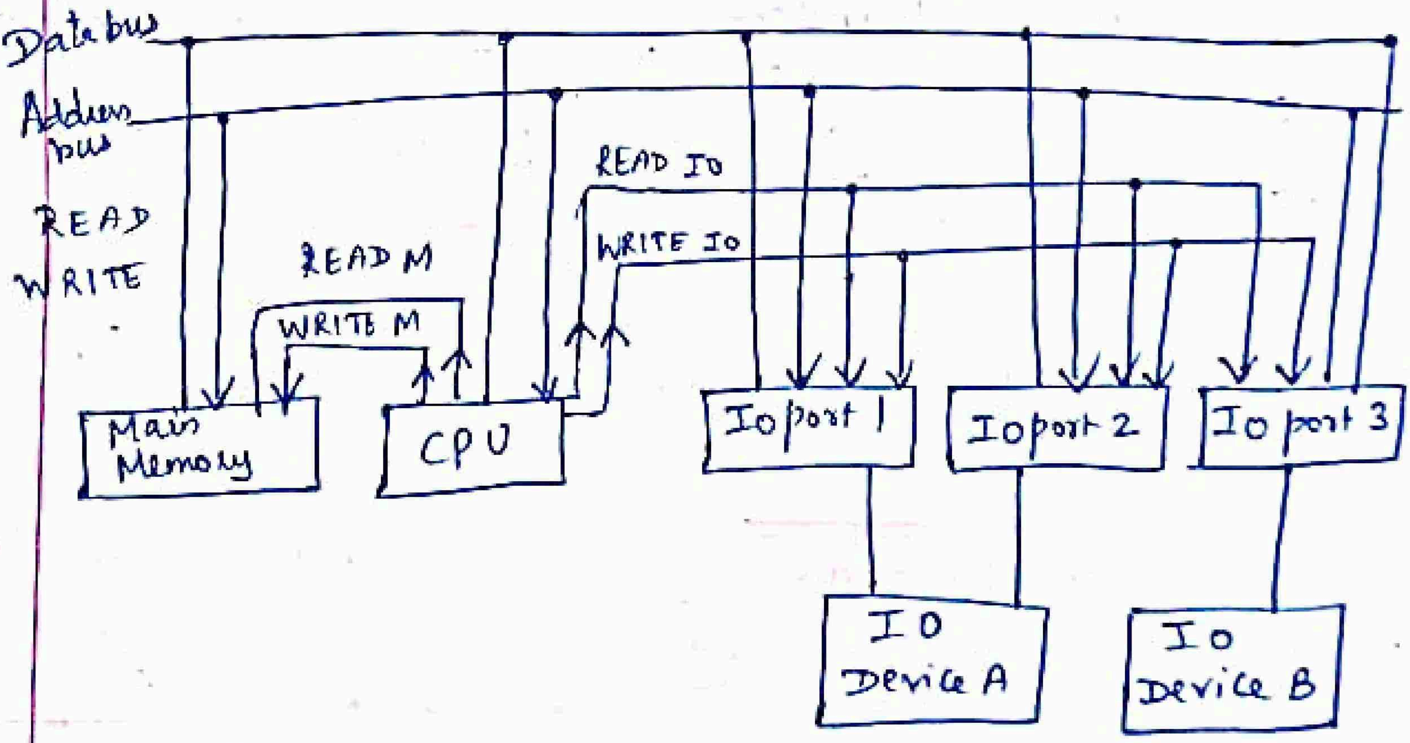


figure: Programmed I/o with I/o- mapped I/o.

*) In IO-mapped IO system, the memory and IO address space are separate. Similarly, the control lines used for activating memory.

*) Two sets of control lines:

→ READ M

→ WRITE IO

IO Instructions:

*) Two IO instructions are used to implement programmed IO. Two instructions are IN & OUT.

IN: The instruction IN x causes a word to be transferred from IO port x to the accumulator register A.

OUT: The instruction OUT x transfers a word from the accumulator register A to the port X.

Limitations of programmed I/O

i) The speed of the CPU is reduced due to low speed IO devices

ii) Most of the CPU time is wasted.

IO Data transfer:

1. Read the IO device's status bit.
2. Test the status bit to determine if the device is ready to transfer data.
3. If ready, proceed with data transfer, otherwise go to step 1.

8) Direct Memory Accen (DMA)

A special contol unit is provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the procenor. This approach is called DMA.

DMA Hardware:

*) All the accen to main memory is done via a shared system bus. The IO device is connected to the system bus via a DMA controller

*) DMA controller maintains 3 registers. The registers are:

i) Addren Register (IOAR):

It is used to hold the addren of the next word to be transferred. It automatically inc or dec after each word transfer.

ii) Data counter (DC)

It is used to store the number of words that remains to be transferred. It is decremented automatically after each transfer and tested for zero. When the data count reaches zero, the DMA transfer halts.

iii) Data Register (IODR):

It is used to store the data to be transferred. When the data transfer over, the DMA controller sends an interrupt signal to the CPU to notify the end of IO data transfer.

DMA Data Transfer:

The following steps are used in DMA data transfer:

1. The CPU executes two IO instructions to initialize the DMA register.

a) IOAR is initialized with the base address of the memory block to be used in data transfer.

b) DC is initialized with number of words to be transferred to or from that memory region.

2. When the DMA controller is ready to transmit or receive data, it activates.

Transparent DMA:

*) A variation of cycle stealing mode is called transparent DMA. The DMA operation does not affect the CPU operation.

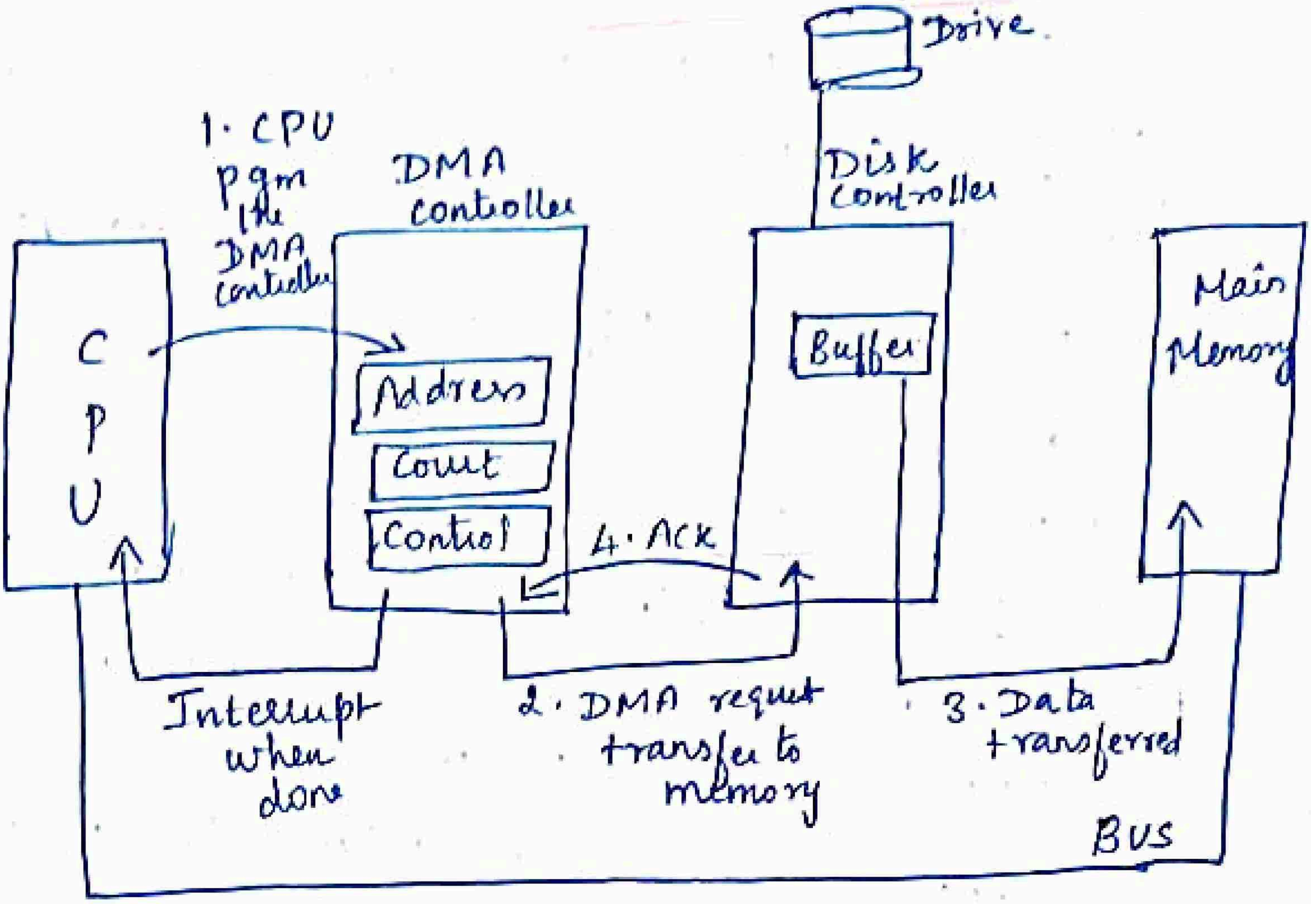


Fig: Interaction of CPU & DMA

DMA Bus arbitration:

There are two approaches to bus arbitration

i) Centralized bus arbitration

ii) Distributed bus arbitration

Centralized bus arbitration

i) Daisy chaining

ii) polling Method

iii) Independent request

9) Interrupts:

*) Interrupts is a signal which has highest priority from hardware or software which processor should process its signal immediately.

*) The following steps are taken while handling the interrupt:

i) Io device enables the INT REQ control line.

ii) Interrupt indicator is enabled in the CPU register.

iii) The CPU identifies the source of the interrupt.

iv) The CPU obtains the memory address of the required interrupt handler.

Interrupt Selection:

1. Single Line interrupt system
2. Multi Line interrupt System
3. Vectored Interrupt.

i) Single Line Interrupt System.

*) It is the simplest of all and uses less hardware.

*) All Io ports share a common single INT REQ line.

*) An INT ACK line is activated by the CPU.

*) The CPU uses polling method to identify the sources of interrupt.

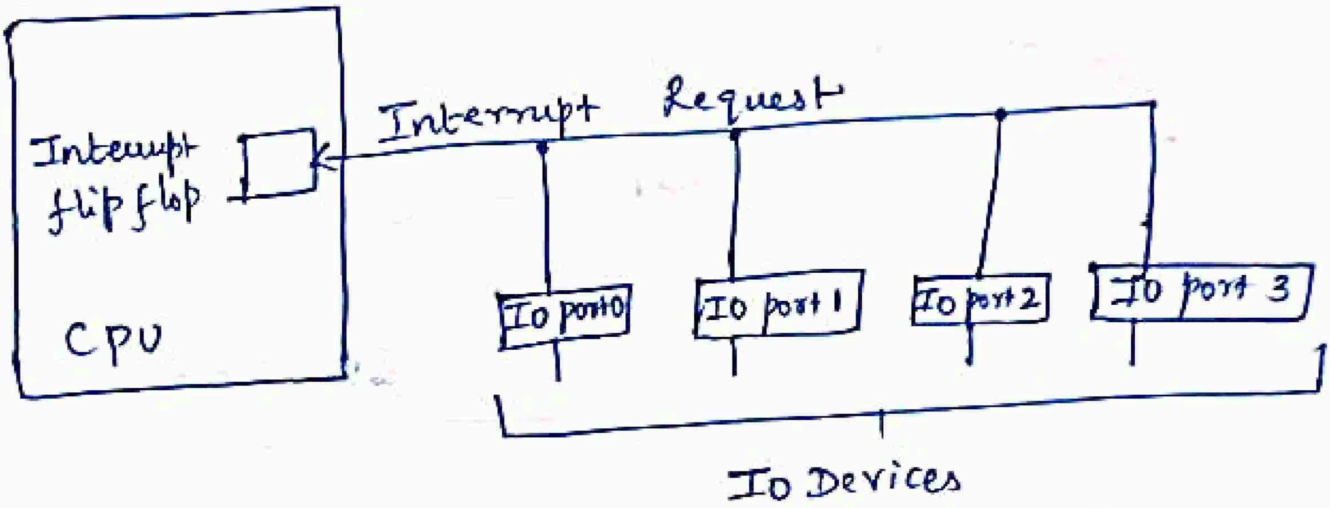
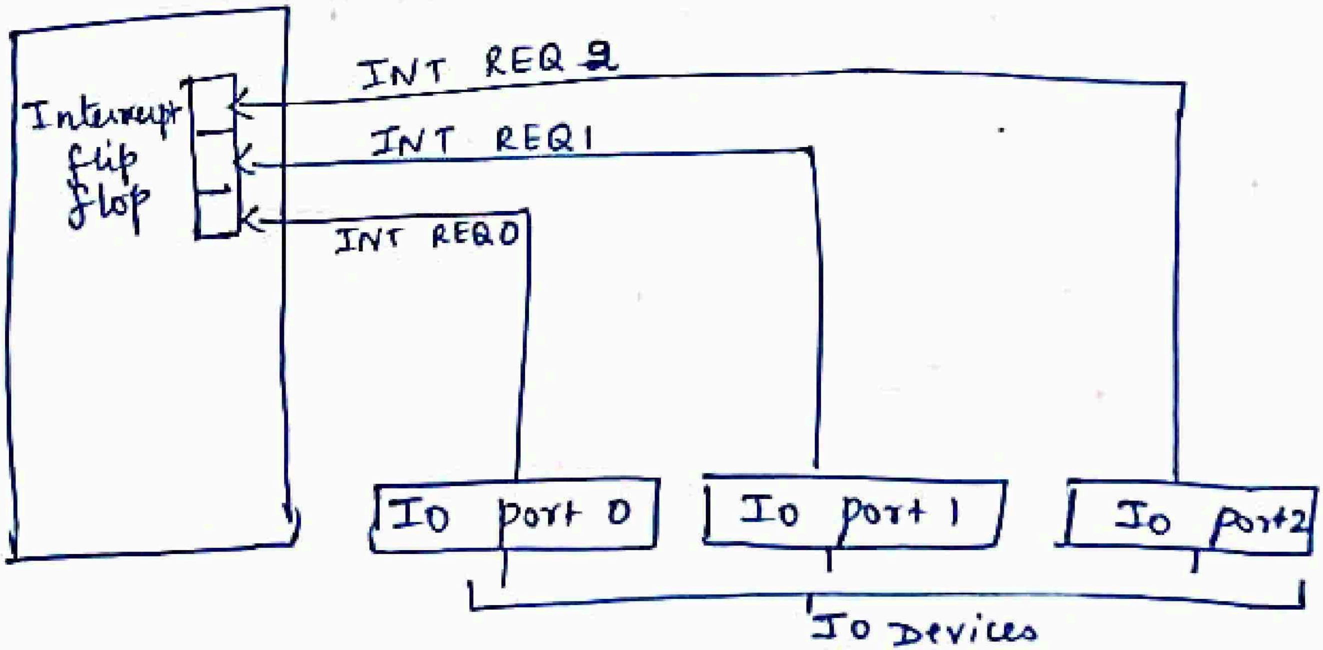


Fig: Single Interrupt System

ii) Multi - Line Interrupt System:

*) In Multi-line interrupt system - each system can send interrupt request immeditely.

*) The processor has multiple interrupt lines. Each line is assigned a unique priority



iii) Vectored Interrupt:

*) In vectored interrupt, the interrupting device must supply the CPU with the starting address or interrupt vector of the program.

*) This method is most flexible in handling interrupts.

IO processors (IOP):

*) The DMA mode of data transfer reduces CPU's overhead in handling IO operations. It also allows parallelism in CPU and IO operations.

IO instruction Executed by CPU:

i) START IO → initiates the IO operation.

ii) HALT IO → it causes the IOP to terminate IO program execution.

iii) TEST IO → It is used to determine the status of the named IO device and IOP.

IO Instruction Executed by IOP:

i) Data transfer Instruction.

ii) Branch Instruction.

iii) IO device control Instruction

↳ rewind, seek address, print pg

→ next CCW

→ read, write, sense.

10) Universal Serial Bus (USB):

*) A Universal Serial Bus (USB) is a common interface that enables communication between devices and a host controller such as personal computer (PC).

*) The commercial success of the USB is due to its simplicity and low cost.

USB Architecture:

*) The USB uses point-to-point connections and a serial transmission format.

*) When multiple devices are connected, they are arranged in tree structure.

*) If I/O devices are allowed to send messages at any time, two messages may reach the hub at the same time and interfere with each other. For this reason, the USB operates strictly on the basis of polling.

*) Periodically, the host polls each hub collect status information and learn about new devices that may have been added or disconnected.

*) When a device is first connected to a hub, or when it is powered on, it has the address 0.

Universal Serial Bus tree structure:

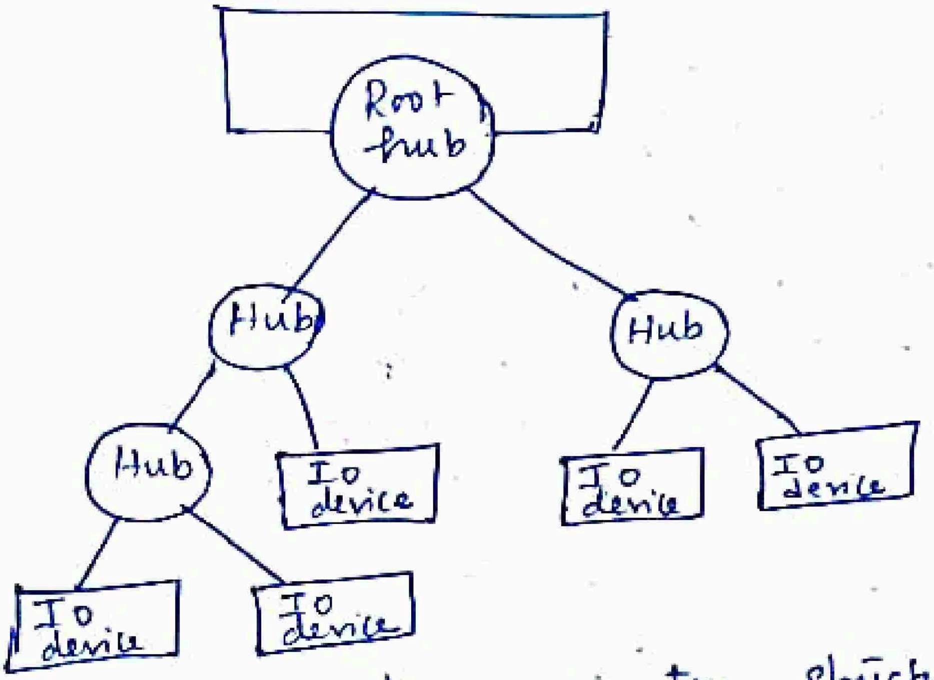
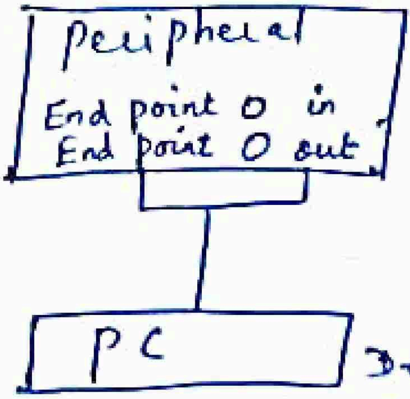


Fig: USB tree Structure

*) The USB is contolled by a host, there are multiple peripherals but only one host per bus. The host can be taken as master and peripheral as slaves, where by the former is responsible for managing the connection, transactions and scheduling bandwidth.

*) USB system uses tiered star topology.

*) In order to synchronize the host and receiver clocks, sync field is used.

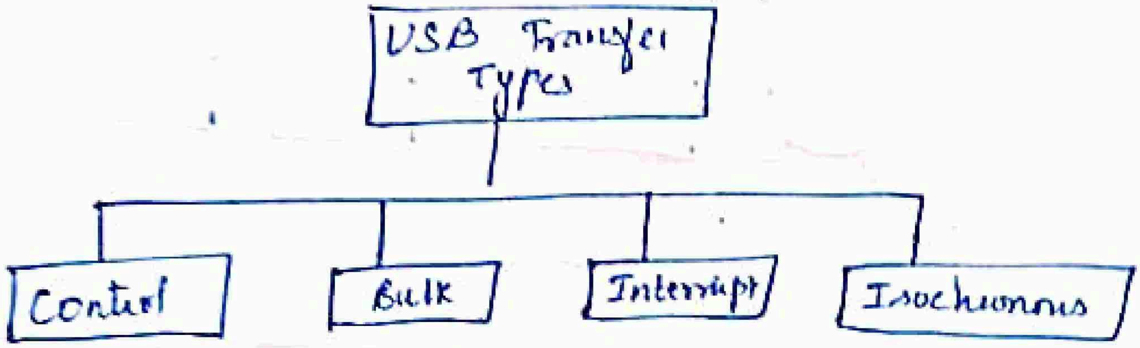


Default End points

Transfer Types:

There are four types of transfer modes which can be used for communication:

i) Control Transfer.
ii) Bulk Transfer
iii) Interrupt Transfer
iv) Isochronous Transfer



Transaction:

A single transaction contains transmission of up to three packets. These packets are:

i) Tocket packet
ii) Data packet
iii) Handshake packet

Handshaking:

•) Handshaking is a mechanism to check the success/failure of a request or to check the delivery of a packet.

•) Terms related to handshaking:

i) ACK → Acknowledgment
ii) NAK → Negative Acknowledgment
iii) STALL → Request not supported
iv) ERR → Split Transaction Error

11) Bus Structure:

*) The bus interconnection scheme supports following types of data transfer

i) Memory to processor → Memory read operation
ii) Processor to Memory → Memory write operation
iii) Processor to I/O → I/O write operation
iv) I/O to processor → I/O read operation
v) I/O to or from memory → DMA operation

Two Types of bus structure:

i) Single bus structure
ii) Multiple bus structure
a) Traditional hierarchial bus architecture
b) High- performance hierarchial bus architecture

i) Single Bus Structure:

In single bus structure, address bus, data bus and control bus are shown by single bus called system bus.

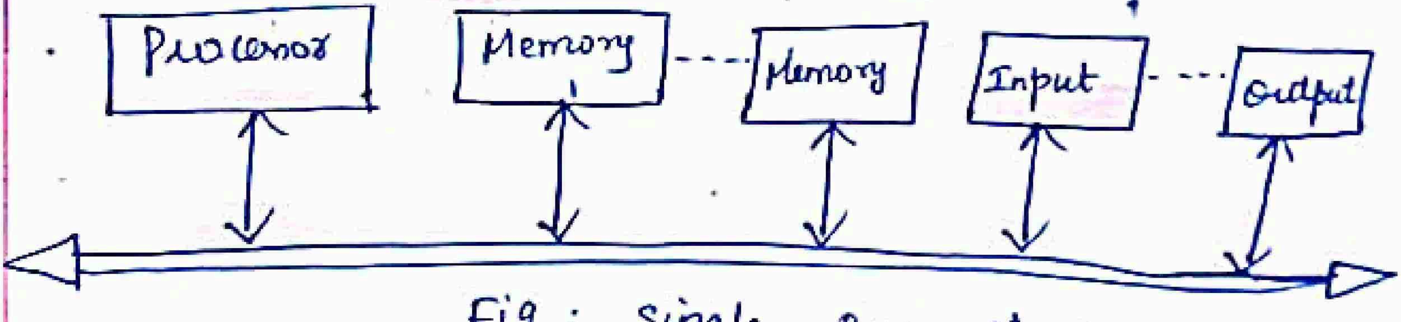


Fig : Single Bus Structure

*) In single bus, only two units can communicate with each other at a time.

*) The main advantage

i) Low cost

ii) flexibility

ii) Multiple Bus Structure:

*) Now-a-days the data transfer rates for video controllers and network interfaces are growing rapidly. To meet this demand, most computer systems use multiple buses.

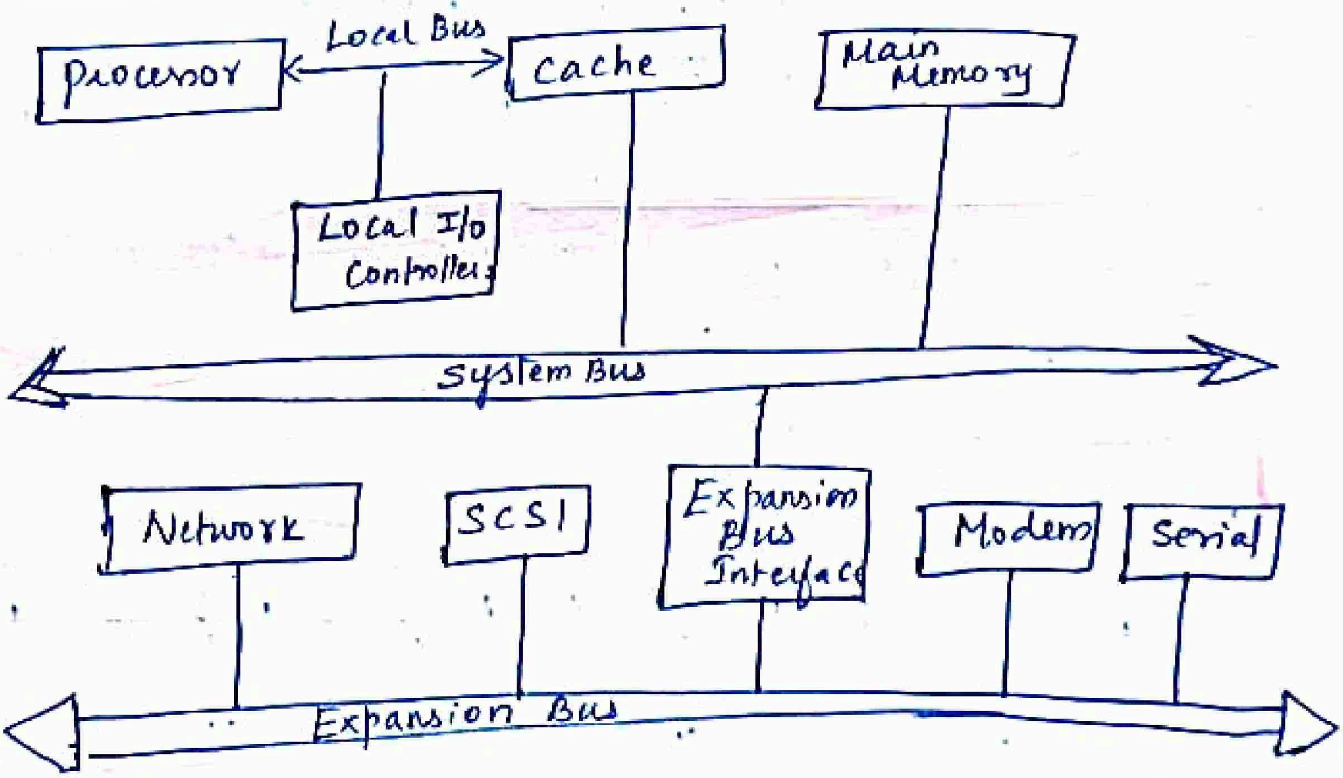
*) These multiple buses have hierarchial structure.

*) Two types of multiple bus structure

a) Traditional hierarchial bus structure

b) High performance hierarchial bus structure.

a) Traditional hierarchial bus structure:



*) Traditional Bus connection uses 3 buses

i) Local Bus

ii) System Bus

iii) Expanded Bus

*) Main memory can be moved off local bus to a system bus.

*) Expansion Bus Interface

→ Buffers data transfer between System bus and system bus.

b) High performance hierarchial bus Structure:

*) Specifically designed to support high-capacity I/o devices.

*) It uses
i) Local bus
ii) System bus
iii) Expanded bus
iv) High speed bus.

*) Cache controller is connected to high-speed Bus

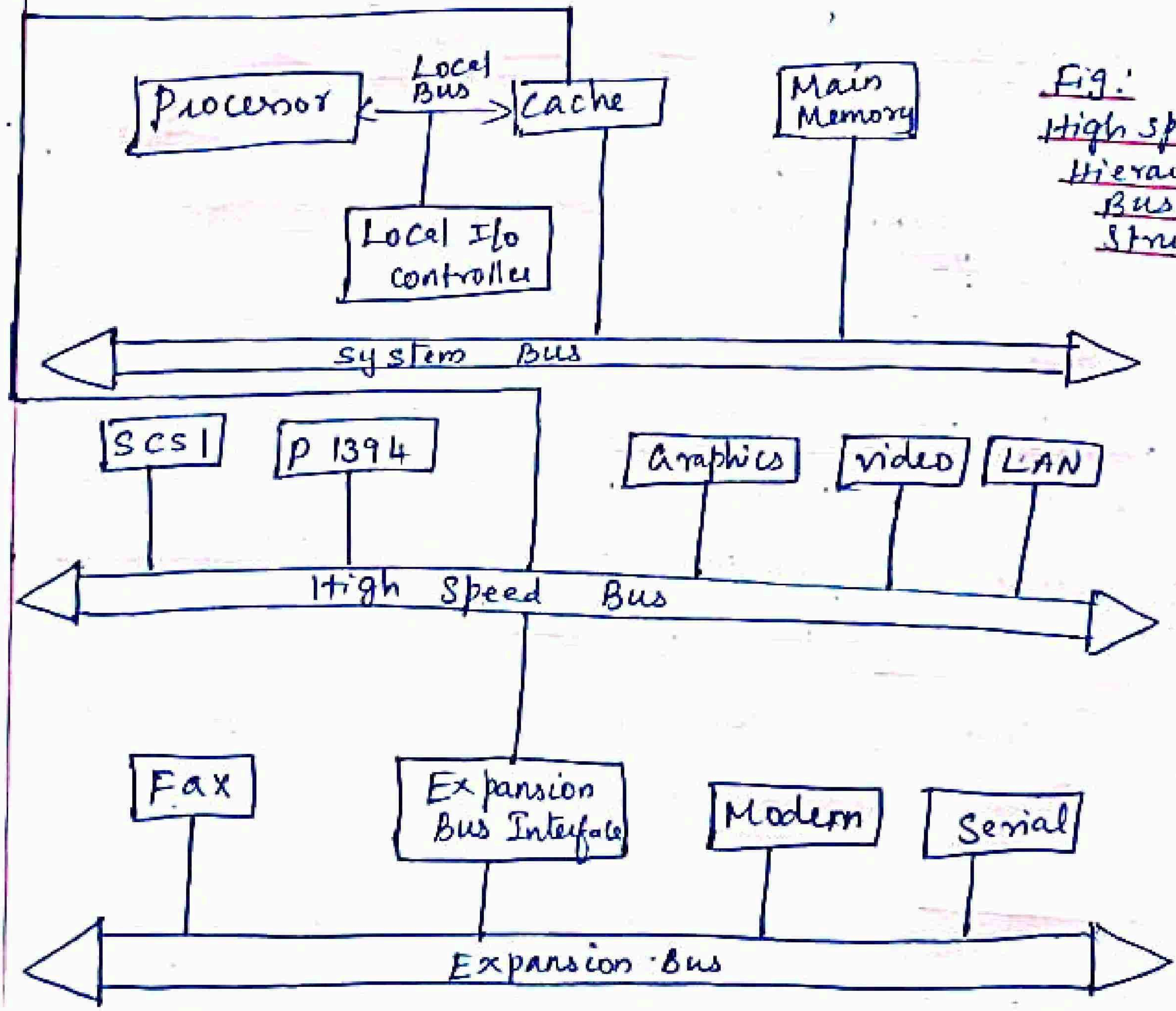


Fig: High speed Hierarchial Bus Structure.

*) This bus supports high speed LAN, such as Fiber Distributed Data Interface (FDDI), video and graphics workstation controllers. ~~and~~

2) Bus operation:

*) The processor, main memory and I/o devices are interconnected by means of a common bus.

*) When processor, memory and I/o devices are connected in the system, one particular unit acts as the bus master → supervises the use of bus by other unit others bus slaves.

*) Mostly, CPU is the bus master
Memory + I/o are bus slaves.

*) Data Transfer is carried out in 3 ways

i) Synchronous
ii) Asynchronous

i) Synchronous Bus:

*) In synchronous bus, all devices derive timing information from a common clock signal.

a) Synchronous Input Operation:

At time t0,

The processor place the device address on the address bus.

At time t1,

The device places its data on the data bus

At time t_2,

the processor reads the data lines and loads the data from data bus into input buffer

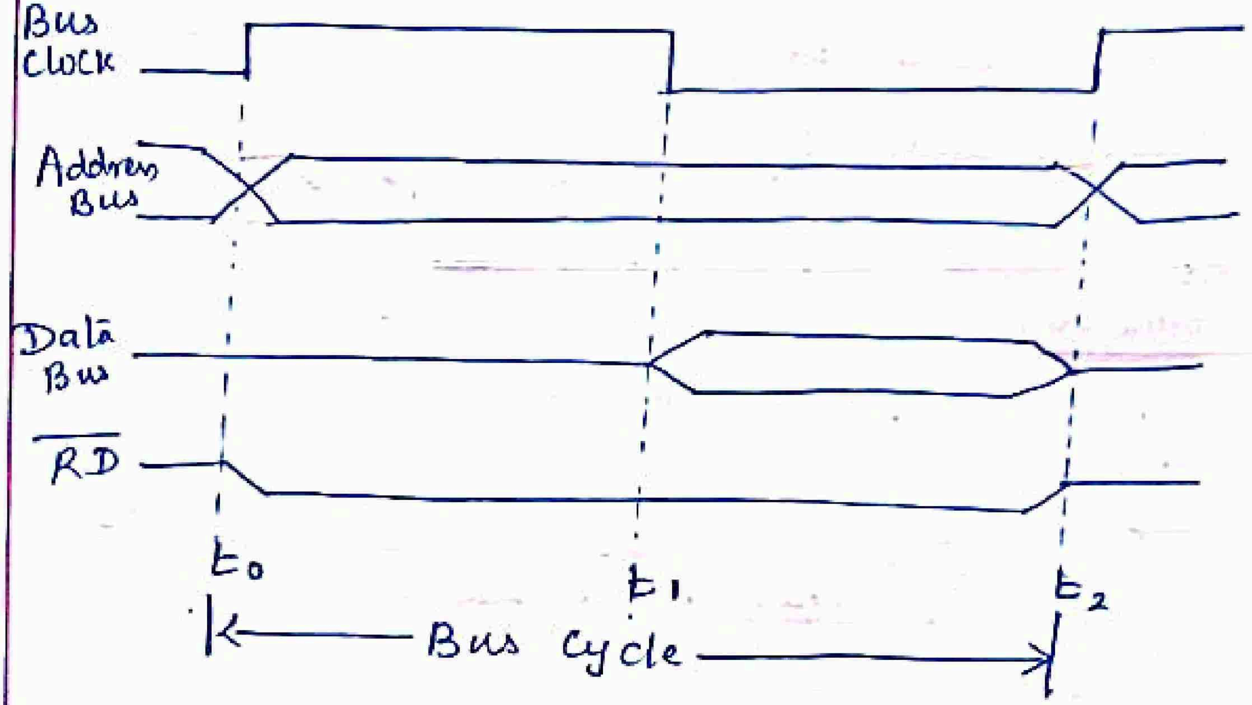


Fig : Timing diagram for synchronous Input transfer

b) Synchronous output Operation

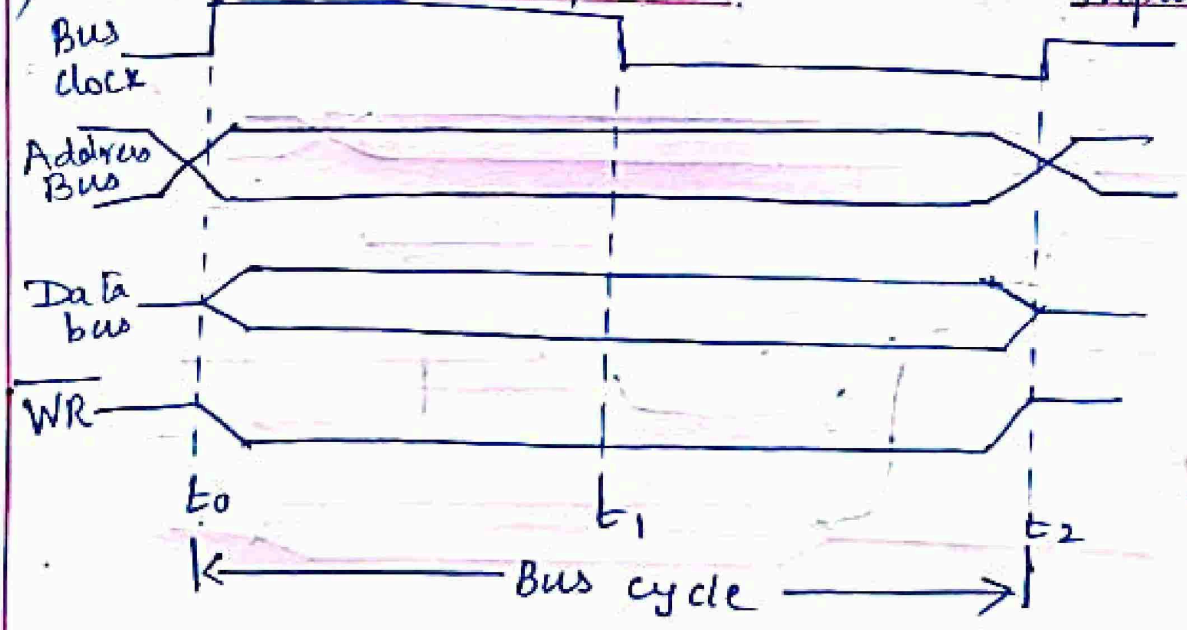


Fig : Timing diagram for Synchronous Output transfer

At time t_0, → processor places o/p data on data lines.

At time t_1, → the device loads the data into data buffer

At time t_2, → the o/p data has been received by the I/O device and the cycle ends

ii) Asynchronous Bus:

*) In asynchronous bus, the common clock is eliminated

*) The data transfer on the system bus is achieved by the use of a hand shake between the processor and the device being addressed.

*) Two control signals → Ready
→ Accept

a) Asynchronous Input operation:

At time t_0,

→ processor places the address on address bus

At time t_1,

→ the device is ready to accept data by activating the Ready Line.

At time t_2,

→ the device receives the ready signal,

→ places the data on data bus & then informs the processor that it has done (by activating Accept Line).

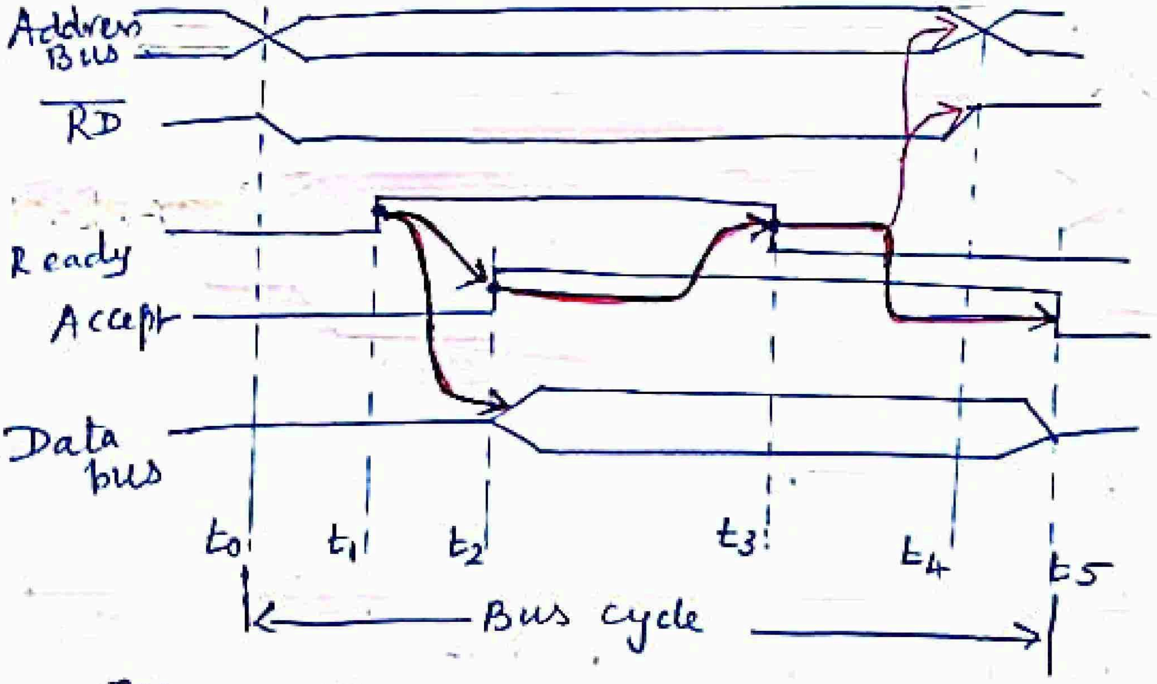


Fig: Timing diagram for asynchronous Input transfer

b) Asynchronous output operation:

*) At time t0, For output operation, the processor places the address, data on the address & data bus

At time t1,

→ device send data by activating Ready signal

At time t2,

→ device indicates that it has done by activating Accept signal.

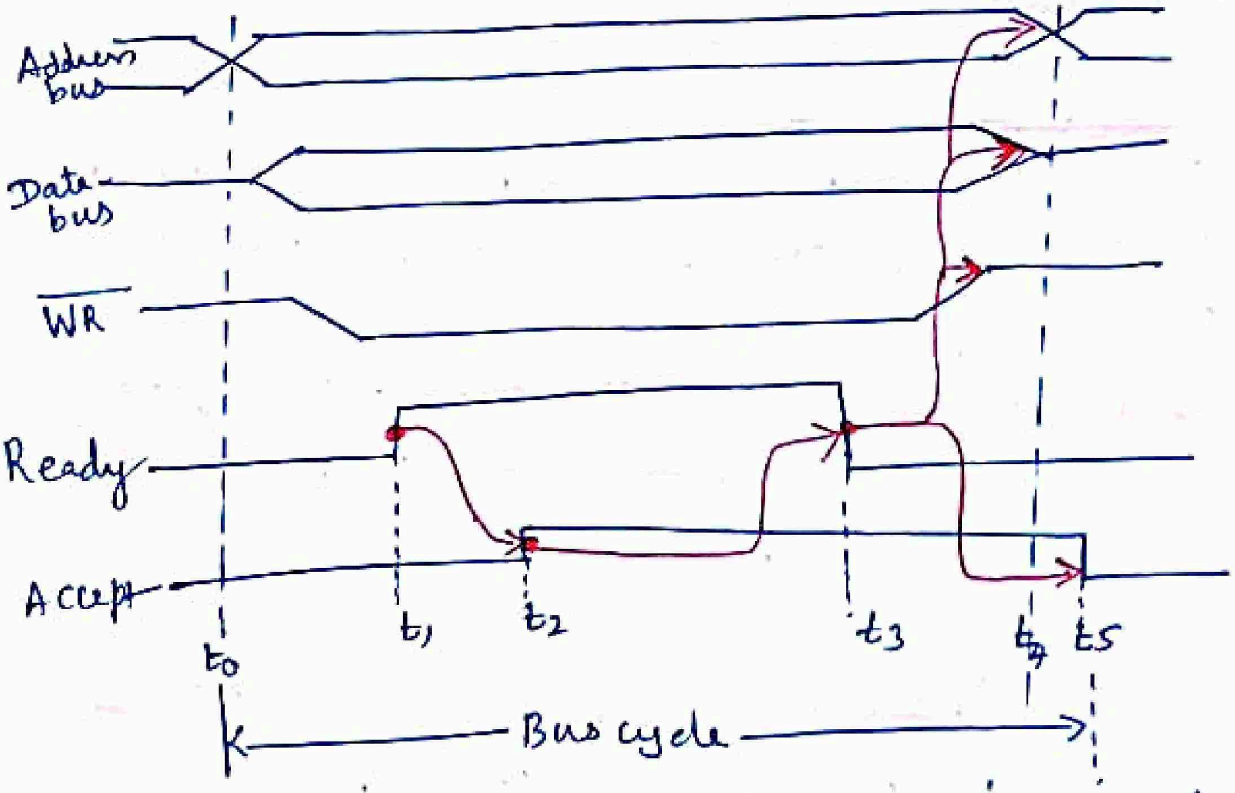


Fig: Timing diagram for asynchronous output transfer

13) Bus Arbitration

*) The device that is allowed to initiate data transfer on the bus is called bus master.

*) Bus arbitration is the process by which the next device to become the bus master is selected and bus mastership is transferred to it.
The selection of bus master is based on priority basis.

Approaches to bus arbitration:

i) Centralized bus arbitration → Single bus master

ii) Distributed bus arbitration → All devices participate in the selection of next bus master

i) Centralized bus Arbitration Scheme:

There are 3 arbitration scheme:

a) Daisy chaining

b) polling method

c) Independent Request.

a) Daisy chaining:

*) It is simple and cheaper method. All masters make use of the same line for bus request.

*) In response to bus request, the controller sends a bus grant if the bus is free. The controller activates the busy line and gains control of the bus

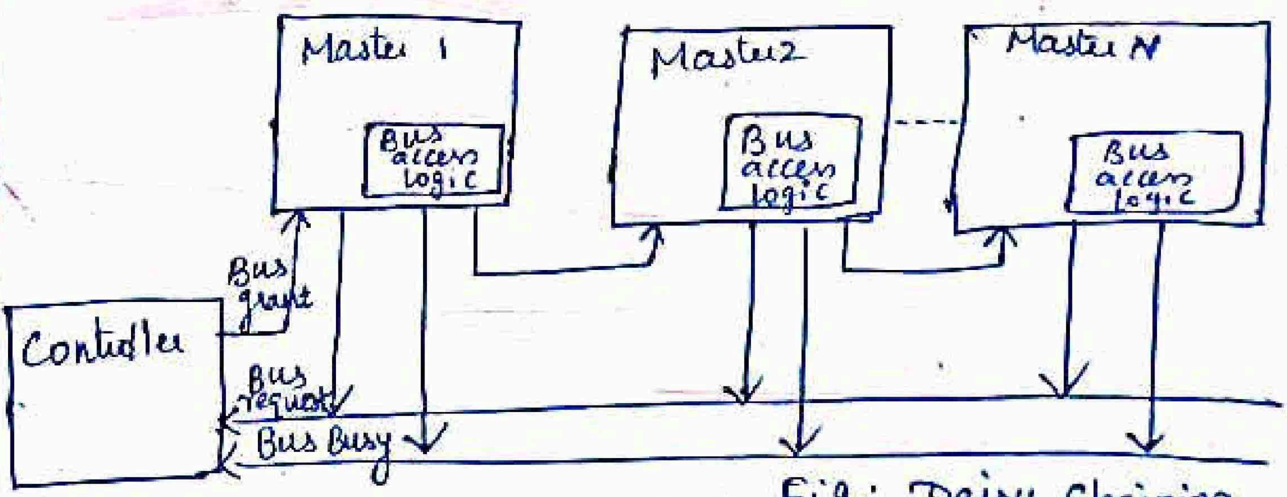


Fig: Daisy Chaining Method

Advantage:

→ simple & cheaper

→ Requires Least no. of lines.

Disadvantage:

→ Failure of any one master cause the whole system to fail

→ priority of master is fixed by physical location.

b) Polling Method:

*) The number of address lines required depends on the number of masters connected in the system.

*) In response to a bus request, controller generates a sequence of master addresses. When the requesting master recognizes its address, it activates the busy line and begins to use the bus

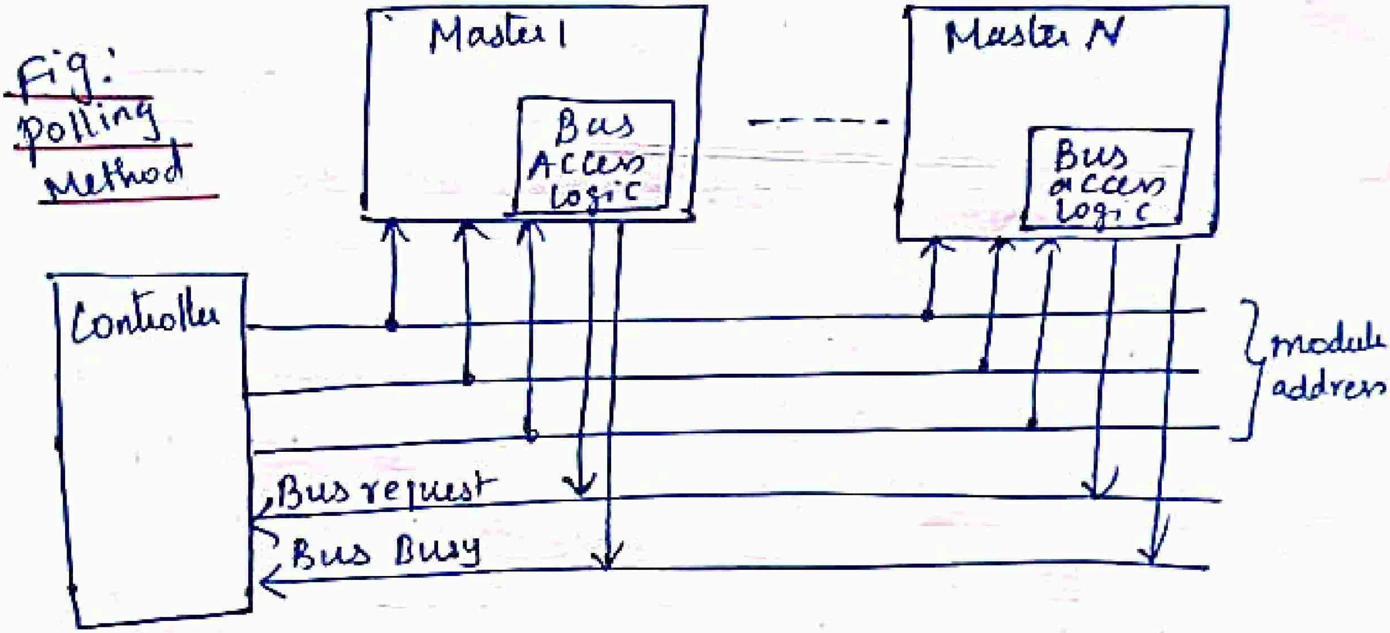


Fig: Polling Method

Advantage:

i) Priority can be changed by altering the polling sequence.
ii) If one module fails entire system does not fail

c) Independent Request:

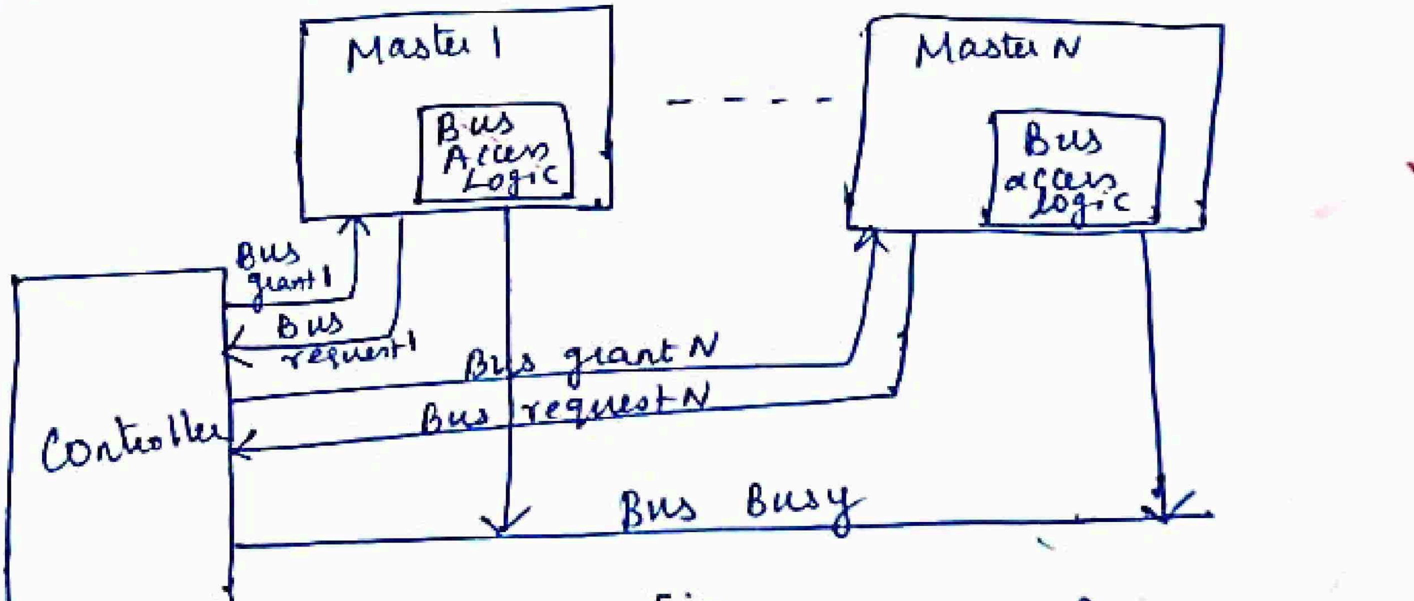


Fig: Independent Request

*) In this scheme, each master has a separate pair of bus request and bus grant lines and each pair has a priority signed to it.

*) Controller selects the highest priority request

Disadvantage :

i) It requires more bus request and grant signals

Advantage :

i) Arbitiation is fast & independent of the number of masters in the system.

ii) Distributed Arbitration:

*) In distributed arbitration, all devices participate in the selection of the next bus master.

*) In this scheme, each device on the bus is assigned a 4-bit identification number. The number of bits used for identification number depends on the number of devices connected on the bus.

*) In this scheme, the device having highest ID number has highest priority.