II yr - CSE
IV - Sem.

COMPUTER ARCHITECTURE

UNIT - III

Processor And Control Unit.

① Explain basic MIPS implementation with Mux and control lines:

(OR)

Explain the basic MIPS implementation of instruction set.

Ans:- MIPS [Microprocessor without Interlocked pipeline Stages] is a reduced Instruction set Architecture (RISA) simplifies the processor design by eliminating h/w interlocks in the five pipeline stages.

A basic MIPS implementation consists of the subset core MIPS instruction set the includes the following,

(a) Memory-reference instruction.
(b) Arithmetic - logical " &
(c) Branch instructions.

⇒ In order to implement any of the above instruction the following two steps must be followed.

(1.) Initially, the program counter (PC) is sent to the memory containing code and instructions is fetched from it (memory).

(2.) Perform 'Read' operation on either one or two registers.

↳ If load word instruction is used, then only one reg is read,

↳ for all other instr such as add, sub, beq etc two reg is read.

⇒ After completion of these two steps the process of execution instr classes (ie memory-reference, ALU & branch) starts.

=> The memory references uses ALU to calculate the address of memory,

↳ the ALU instruction uses ALU for executing the operation and.

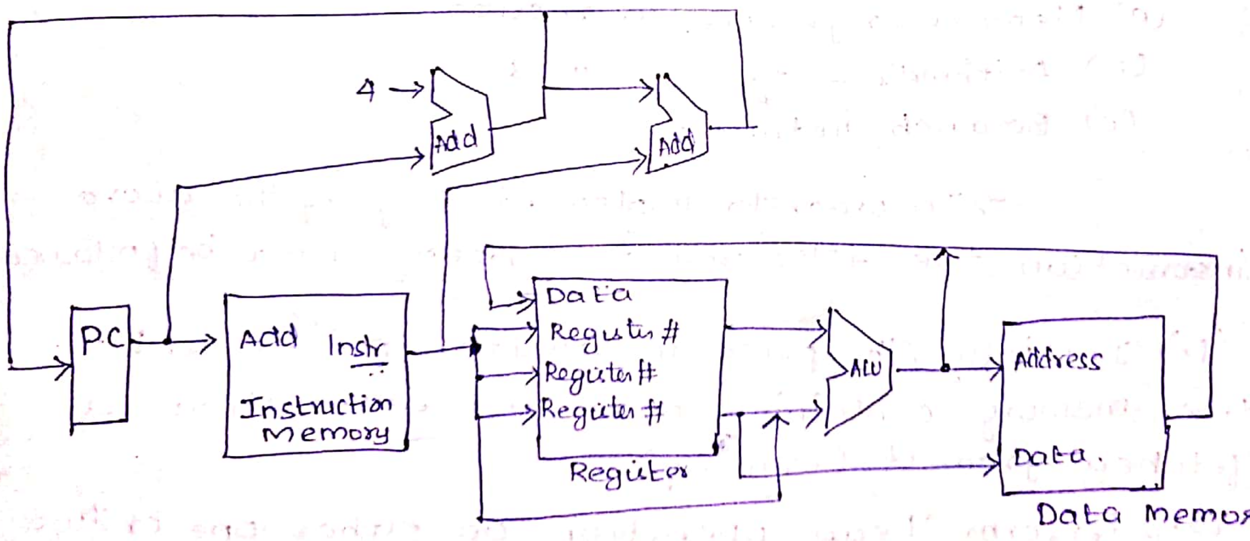
↳ branch instr uses ALU to compare the registers.

=> That is, memory reference instr is responsible for reading and writing the data for load and store instruction.

↳ The ALU instr is responsible for writing the data from the ALU or memory back into a reg &

↳ branch instr is responsible for modifying/changing the addr of next instr based on the comparison of reg.

=> The addr of next instr is obtained by incrementing PC by 4 (ie PC=PC+4).



a) High level Basic MIPS Implementation.

-> In above fig(a) every functional unit has an i/p that comes from two different sources.

↳ PC takes the i/p data from any of the two address,

↳ the register takes the i/p data from ALU or Data memory.

→ The ALU takes i/p from a register or the next instruction field.

⇒ Inorder to select appropriate sources from 2 different sources a device called Multiplexer is used.

In abv fig ; the data memory perform read operation when load instr is used and write operation when store instr is used.

Similarly register file performs write operations only when load instr or ALU instr is used.

⇒ These operation of each function units can be controlled by device called control lines.

⇒ In the above fig (a) neither mux nor control lines are used.

⇒ The Multiplexers and control lines are used only in the basic MIPS implementation.

well as control lines for the major functional units.

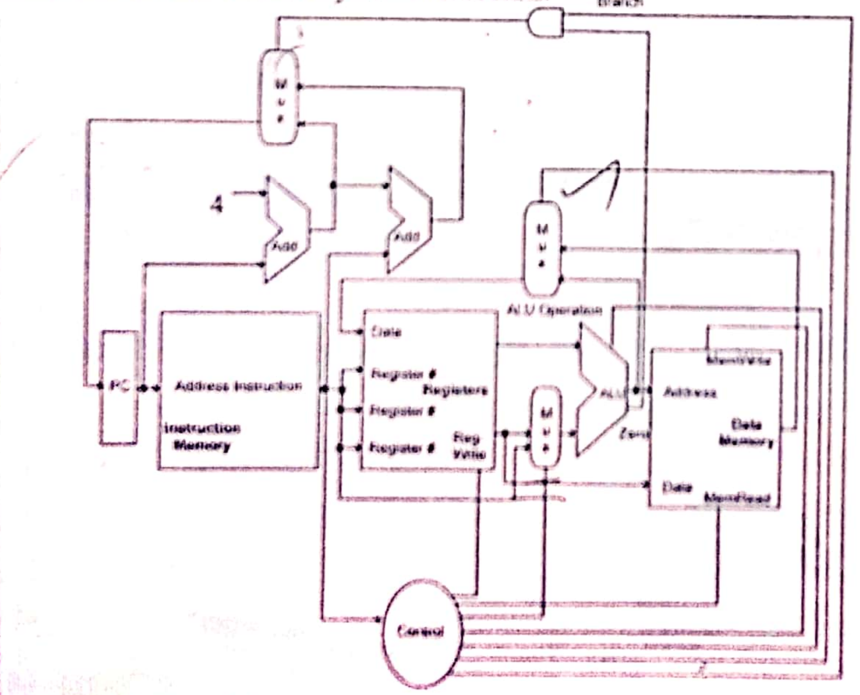


FIGURE 3.2 : The basic implementation of the MIPS subset including the necessary multiplexers and control lines

The basic MIPS implementation architecture Consists of 3 Mux and control lines used for functional units.

⇒ The control lines units specify the method for arranging the control lines for the functional units and two multiplexors.

⇒ The third multiplexors checks whether the program counter is incremented by 4. (or) whether the branch destination is written in the PC.

↳ It is set depending on the o/p of the ALU zero.

↳ The simple decoding process can determine the processing of setting the control lines.

X — X — X

② Explain in detail the operation of the datapath.

Ans:-

Datapath → Datapath is collective name for the registers, the ALU & interconnecting bus.

↳ During the progress of instr execution, transfer of data takes place from one reg to other.

↳ The data often ~~carried~~ passes via ALU to carryout some ALU operation if needed.

↳ The instr loaded in IR registers specifies the actions to be implemented.

↳ Thus, the instr decoder & control logic unit perform implementation of these action.

↳ The instr decoder selects the required register by means of generating the control signals & directs the transfer of data.

⇒ The operations performed by the data path shows how the instruction classes [i.e R-format or ALU, memory-reference instr, branch instr] uses the datapath effectively.

(i) Datapath operations for R-type Instruction:-

⇒ It requires 4 steps to execute the instruction.

eg:- Consider R-type instr as,

$\boxed{\text{Add \$t1, \$t2, \$t3.}}$

To execute above instr follow the step

Step 1 :- Initially the given instruction is fetched from the instr memory and the value of PC is incremented

Step 2 : The register \$t2 and \$t3 are read from the register file and the control lines setting is computed using the main control unit.

Step 3 :- The ALU performs the data read operations from the register file by making use of the function code [5-0] which generate the ALU function.

Step 4 :- The output of ALU is written in the register file and the destination reg \$t1 is selected.

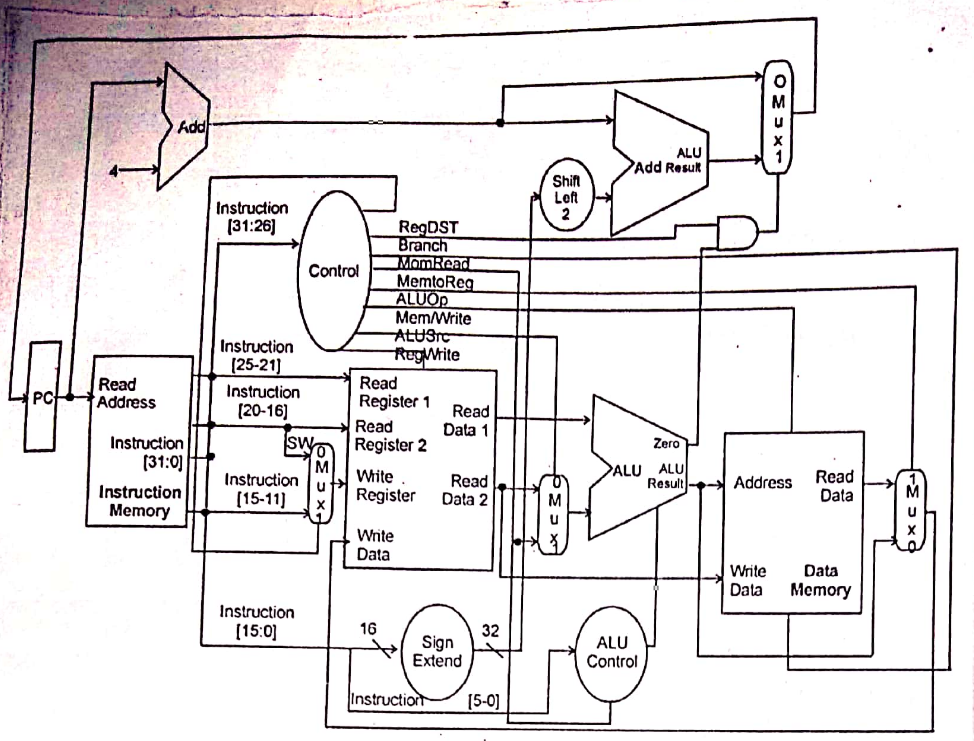


FIGURE 3.17 The datapath in operation for an R-type instruction such as add $t1,$t2,$t3.

(ii) DATA PATH OPERATION FOR LOAD INSTRUCTION (MEMORY REFERENCE INSTRUCTION).

=> data path for load word instr requires 5 steps to execute.

=> Consider the load word instr as,

$\boxed{\text{lw \$t1, offset (\$t2).}}$

To execute above instr, follow 5 steps

Step 1:

Initially the given instr is fetched from the instr memory and the value of PC is incremented.

Step 2:

The register $t2 which resides in the register file is read.

Step 3:-

ALU performs the sum operations on the data read from the register file and it also compute the sign-extended bit (ie offset) which is 16 bit long.

Step 4:- The data (sum of the value) in the ALU is used as the data memory address.

Step 5:- The data of the memory unit is written in the register file and finally the destination register $t1 is given as o/p (ie bits 20-16).

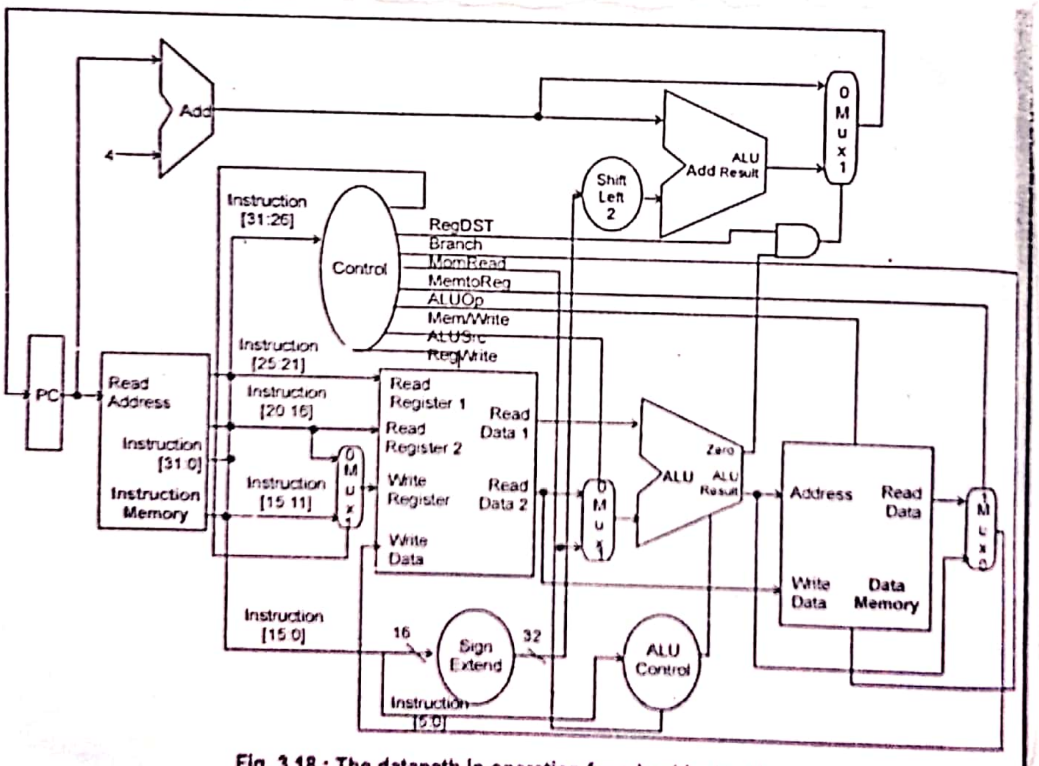


Fig. 3.18 : The datapath in operation for a load instruction

(iii) DATA PATH OPERATION FOR BRANCH-ON-EQU INSTRUCTION.

=> It requires 4 steps to execute the instr. consider beq instr.

$$\boxed{\text{beq \$t1, \$t2, offset}}$$

Step 1:- Initially the given instr is fetched from the instr memory and the value of PC is incremented.

Step 2: The ~~co~~ register contents of \$t1 & \$t2 are read from the register file.

Step 3:- ALU perform the subtract operation on the data which is read from the register file.

→ The incremented value of PC (i.e, PC=+4) is added to sign extend unit and the offset which is 16 bit long is shifted left by 2-bits.

→ The result obtained is used as the branch target address.

Step 4:- If the zero(0) o/p of the ALU is used in determining the adder result that is to be stored in PC.

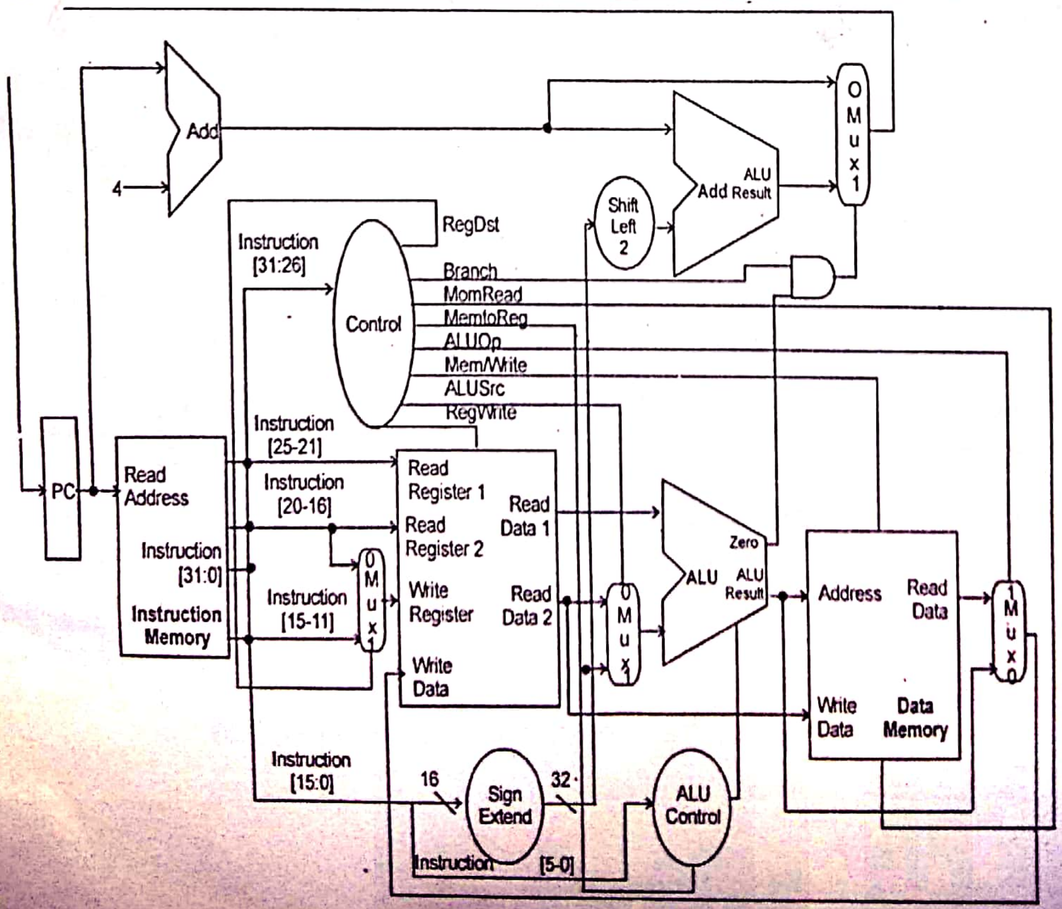


Fig. 3.19 : The datapath in operation for a branch on equal instruction

X—X—X

③ What is pipelining? Discuss about pipelined data path & control.

(or)

Discuss about data and control path methods in pipelining

(or)

Discuss the modified datapath to accomodate pipelined execution with a diagram.

Ans:-

Pipelining ⇒ Pipelining is an important implemen-tation technique in which multiple instruction are overlapped in execution.

(or)

Pipelining can be defined as an effective technique of organizing simultaneous tasks by dividing it into the number of subtasks.

↳ This increases the NO of tasks performed per second, thereby enhancing the speed of execution of instructions.

Structure of pipeline:-

⇒ A pipeline carries several segments each of which carries two important components.

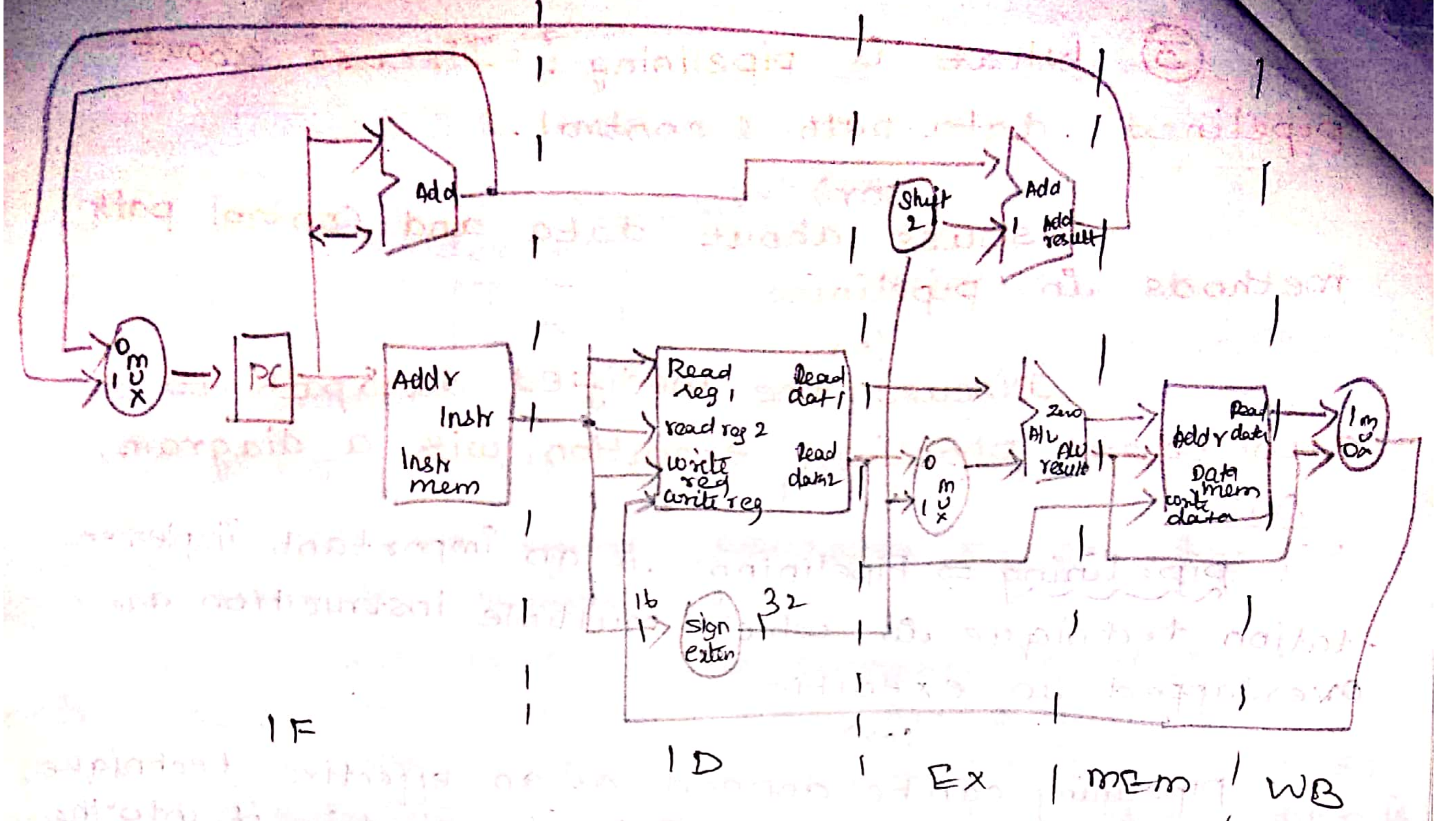
(i) An input register
(ii) A computational circuit.

⇒ An i/p reg contains data
⇒ A computation circuit will perform the sub operations associated with that seg & its o/p acts as a i/p to next segment reg.

Pipelined Data path :-

⇒ It consists of 5 different stages. It also indicates 5 instr will be under execution within a single clock cycle.

① Instruction fetch (IF)
② Instruction decode and register file read (ID)
③ Execution or address calculation (EX)
④ Data Memory access (DMA)
⑤ Write back (WB).

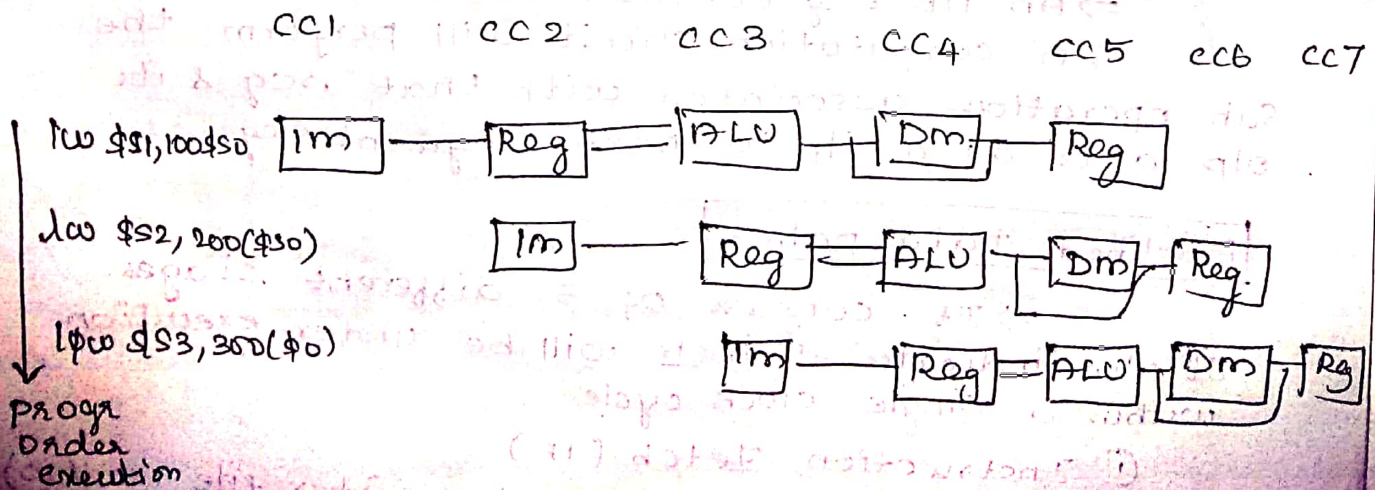


⇒ The data in the datapath flows from left to right.

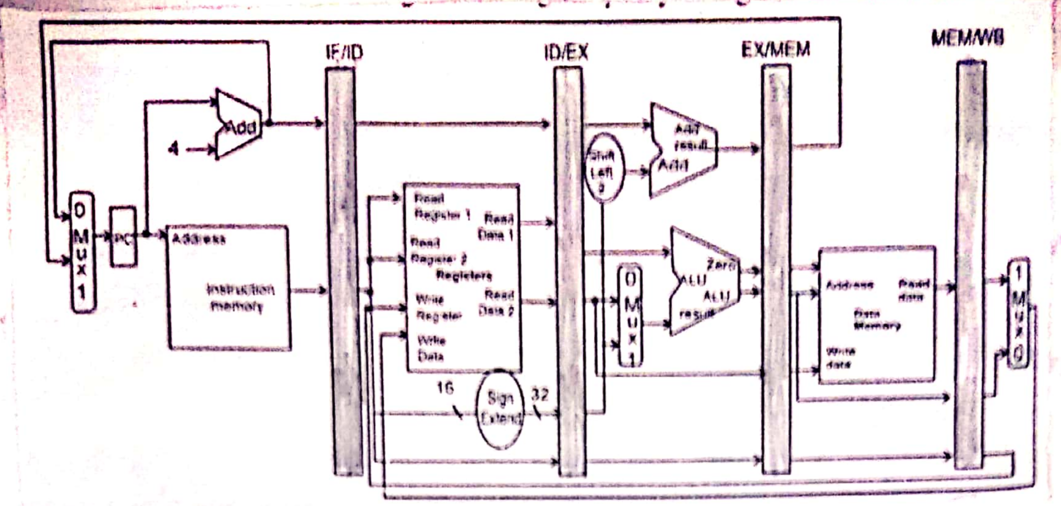
→ If the data flows from right to left, then it leads to hazards in the pipeline. [first right-to left leads to data hazards & second right to left leads to control hazard].

⇒ Every instr in pipeline has its own data path. consider 3 instr

lw $s1, 100($s0)
lw $s2, 200($0)
lw $s3, 300($s0).



The pipelined version of datapath



Pipelined Datapath for load Instruction:-

⇒ load instr which passes through all five ~~instr~~ stages of pipelined execution.

⇒ load instr is active when passed through all five stages.

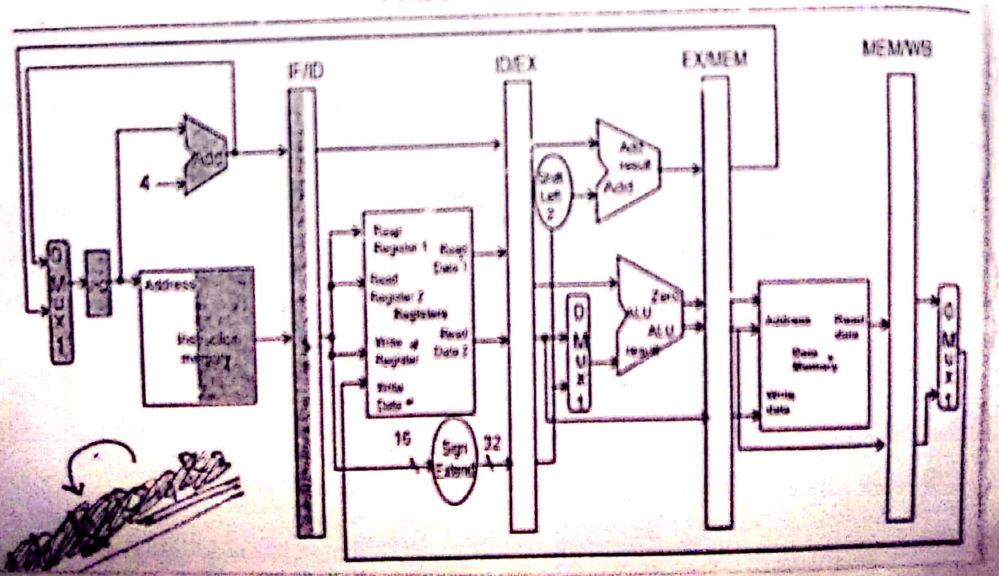
(1) IF :-

↳ Initially instruction is fetched from the instr memory and value of PC is incremented by 4.

↳ The incremented value is stored/saved in the IF/ID pipeline register,

↳ So that if any other instr in the sequence require this value, then it can be refer IF/ID pipeline register.

Instruction Fetch (IF) for load Instr



(2) Instruction decode & Register File Read.

↳ This stage supplies 16-bit immediate field which is extended to 32 bits long ex Sign-extended unit.

↳ It also have register members that reads the 2 register.

⇒ All these value stored in ID/EX pipeline register including the incremented PC addr.

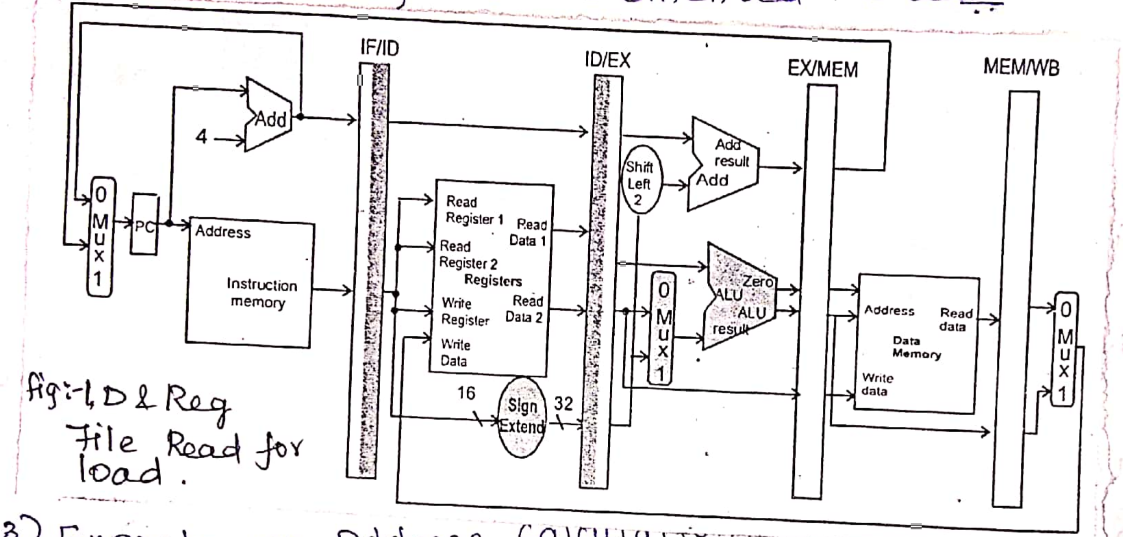


fig:- ID & Reg File Read for load.

(3) Execute or Address Calcula

→ In this stage, the data in the register 1 is read by the load instr

↳ and also reads the sign extend bits and performs addition operations on them using ALU.

→ This Sum is Stored in EX/MEM pipeline register.

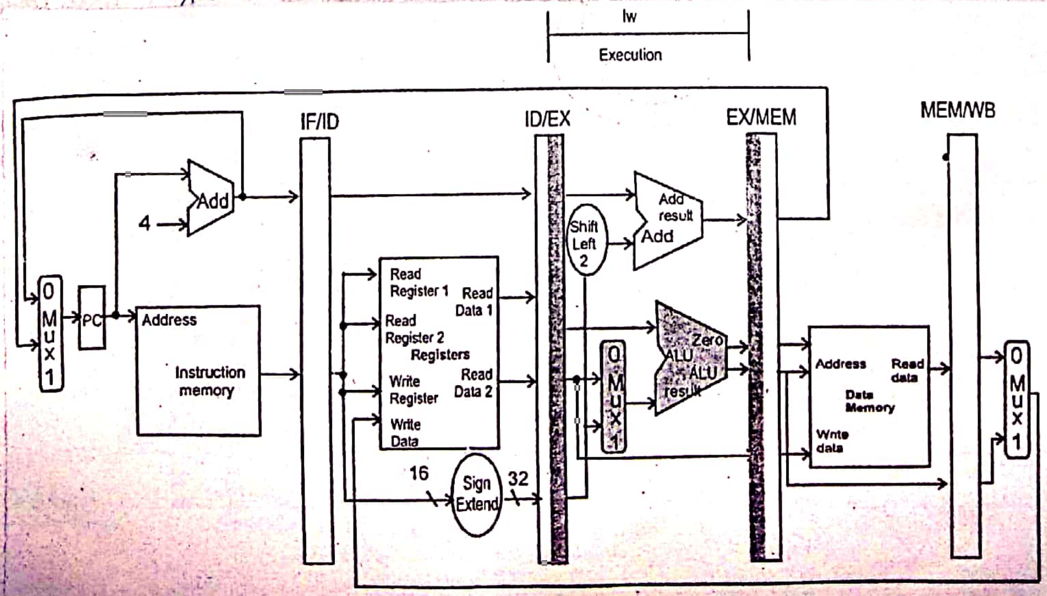


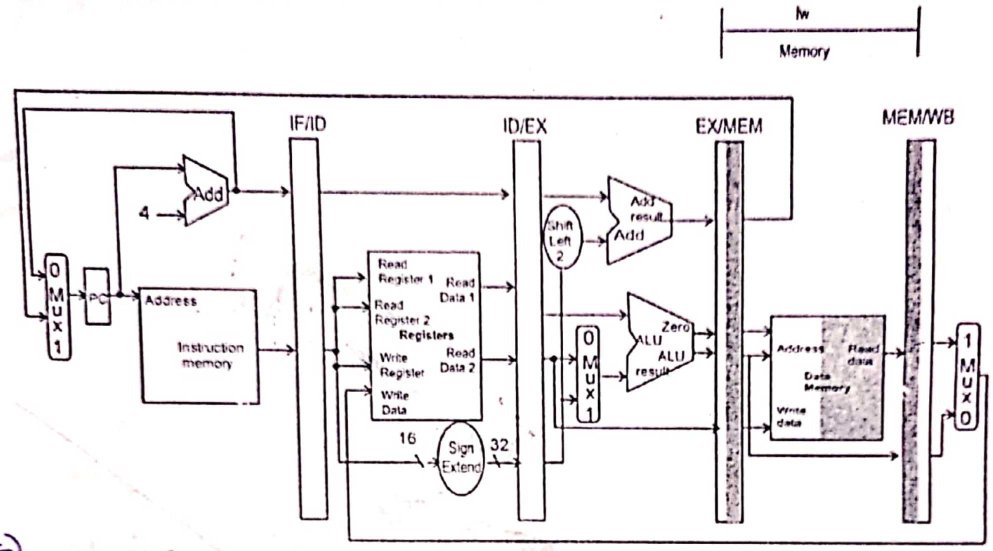
FIGURE 3.30 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 3.28 used in this pipe stage

[4] Memory Access :-

→ In this stage, the load instr reads the data memory with the help of the address present in the Ex/mem pipeline register

↳ and the data is then loaded in the mem/WB pipeline register.

↳ and is written back into the register file present In ~~the~~ between the IF/ID & ID/Ex stage.



~~(5)~~

Figure 3.31(a) : MEM and WB stages of load instruction

(5) Write back :- This is the last stage. The data is read from the mem/wB pipeline reg and written back to reg file present in between IF/ID & ID/Ex stage.

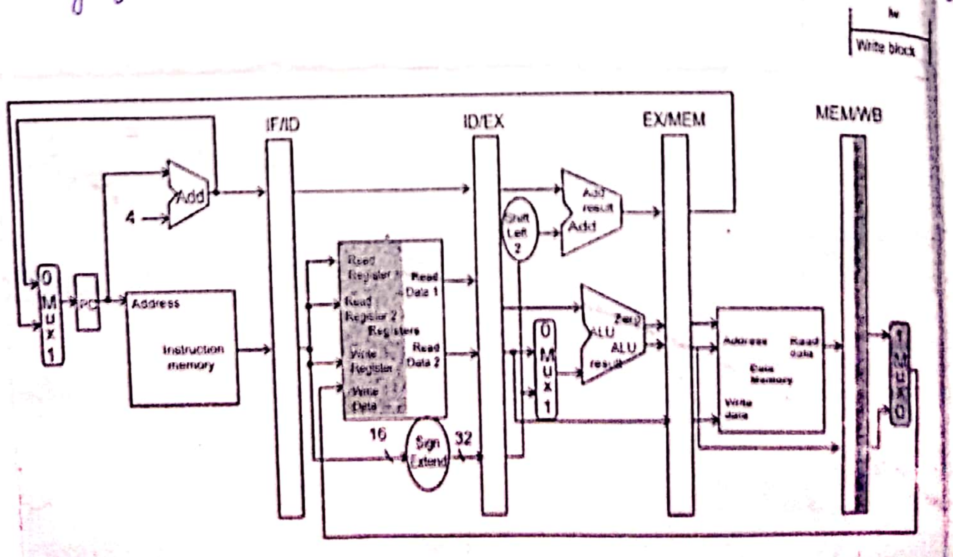


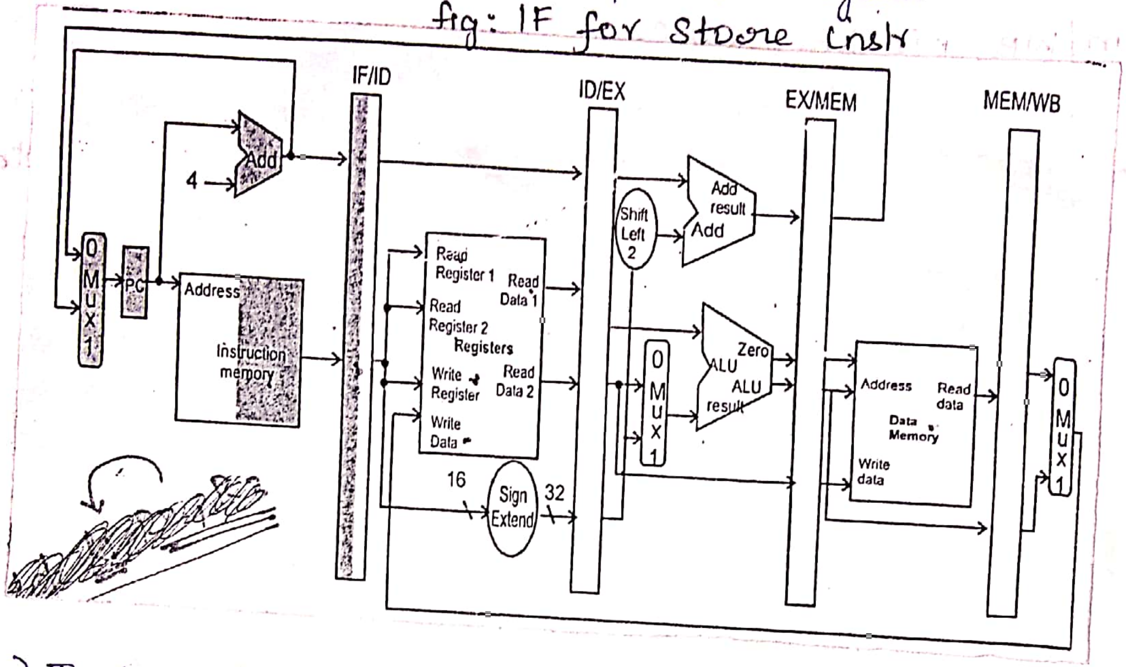
FIGURE 3.31 (b): MEM and WB: the fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 3.33 used in this pipe stage.

PIPELINE STAGES FOR STORE INSTRUCTION.

(1) Instruction Fetch:(IF)

→ In this stage, the instr is read from memory using the addr in PC & then is placed in the IF/ID pipeline register.

fig: IF for Store instr



(2) Instruction decode & register file read:-

→ The instr in the IF/ID pipeline reg supplies the reg no for reading 2 reg & extends the sign of the 16 bit immediate.

→ These ~~three~~ three 32-bit values are all stored in the ID/EX pipeline reg.

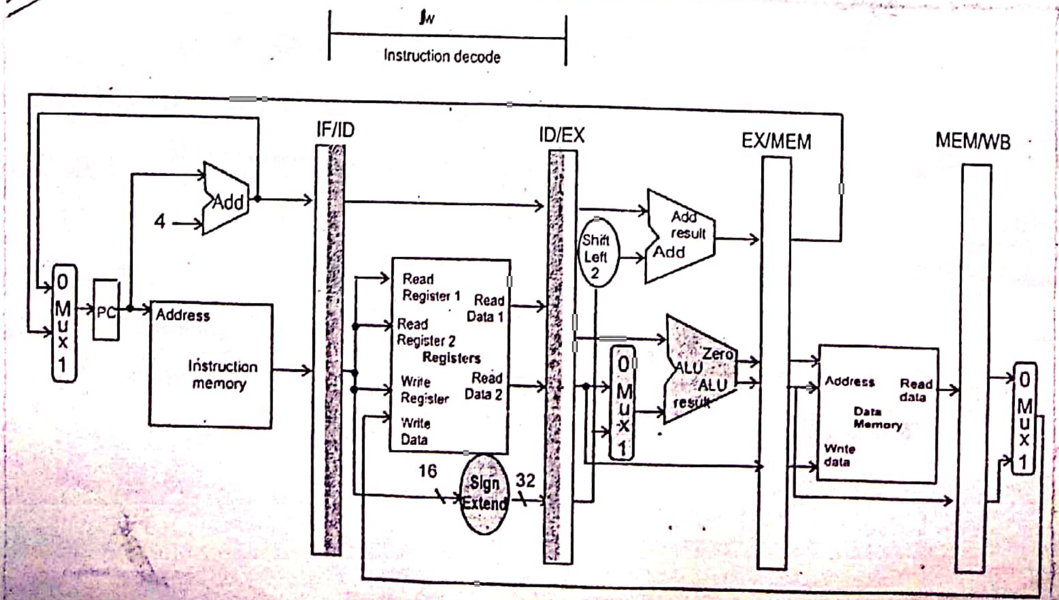
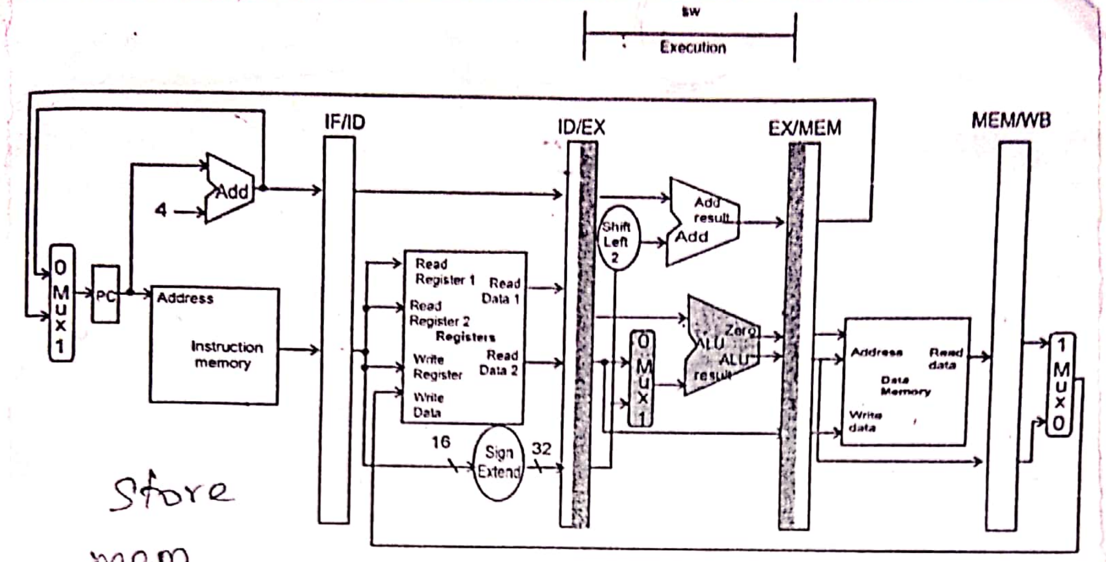


FIGURE 3.29 : IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 3.28 highlighted.

(3) Execute & Address Calculation:- →The effective addr is placed in the Ex/MEM pipeline register.



Store mem ↑ n

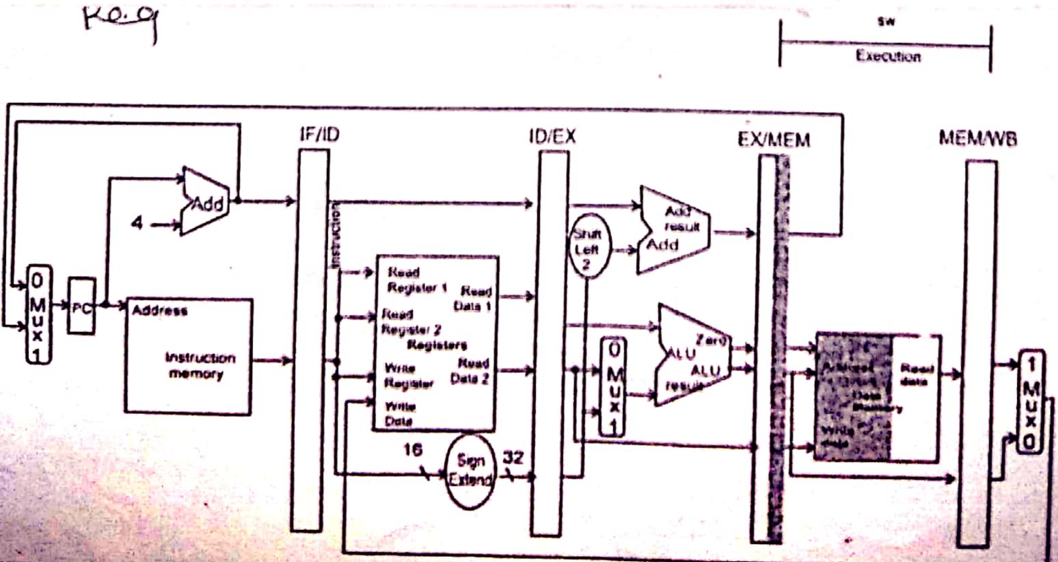
FIGURE 3.32 EX: the third pipe stage of a store instruction

(4) Memory Access:- The data being written to memory. Note that the register containing the data to be stored was read in an earlier stage & stored in ID/EX.

→The only way to make the data available during the mem stage is to place the data into Ex/mem pipeline register in the EX stage,

→just as we stored the effective addr into Ex/mem.

| Reg



(5) Write back:- For this instr, nothing happens in the write back stage.

→ Since every instr behind the store is already in progress, we have no way to accelerate those instr.

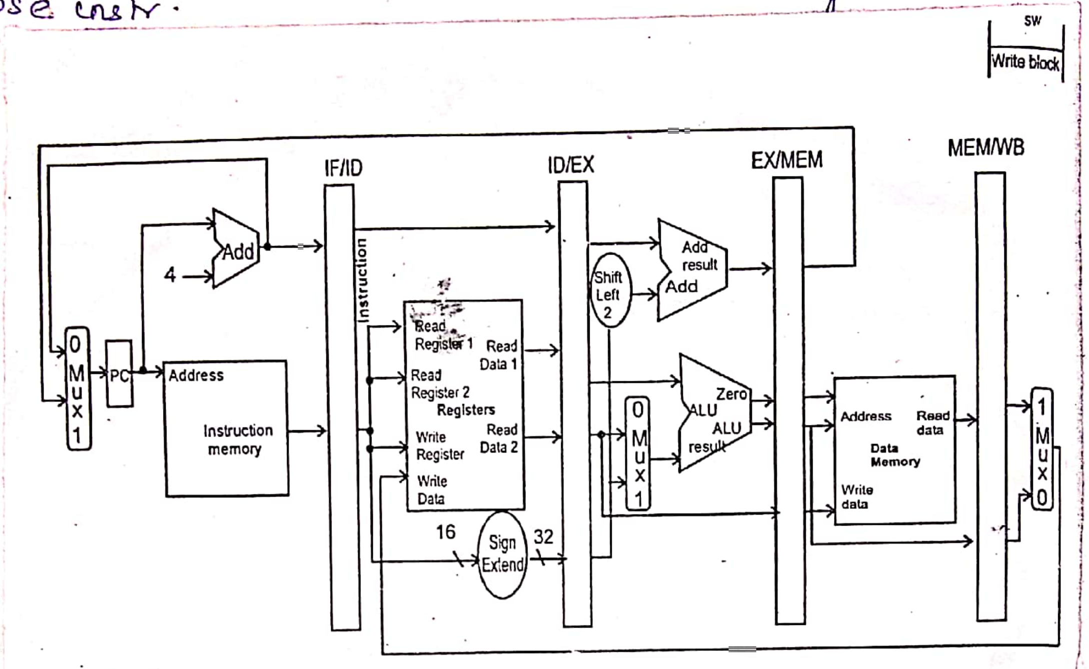


FIGURE 3.34 MEM and WB: the fourth and fifth pipe stage of a store instruction

(4) Explain how the instr pipeline works. What are the various situations where an instr pipeline can stall?

Ans :-

Instruction pipeline:-

An instruction pipeline is a sequence of independent steps with storage at the termination of each step.

The instr pipeline speeds the instr execution by overlapping the execution of the current instr with fetch, decode & operand fetch of subsequent instr.

This technique is also referred as "instruction look ahead".

Sol.

Four Segment Instruction pipeline :-

→ complete execution involves 6 steps.

steps	descriptions
1	I.F
2	I D
3	Effective addr calculation
4	operands fetch
5	Instr execution
6	Store the result.

=) For simplicity, above 6 steps are reduced to 4. This is done by combining steps 2 & 3 and step 5 & 6.

Steps	Description	Symbol.
Step 1	Instr Fetch	F
Step 2	Instr decode + Effective addr Calculation	IE
Step 3	operands fetch	O
Step 4	Instr execution + Store the result	IS.

Instruction pipeline can stall during two situations

→ data hazards

→ Control "

1. STALL DURING A DATA HAZARD :-

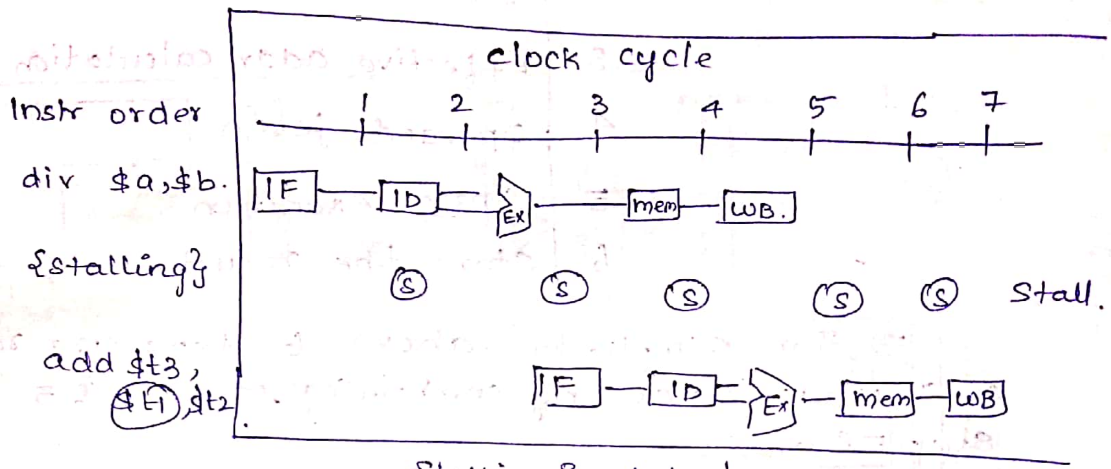
=) A stall occurs when processing of one instr in the pipeline depends on the processing of other instr.

↳ This situation is called data hazard where stalling occurs.

and stalling remains until that another instr completes its processing.

→ This stalling results in increase in clock cycles based on the time taken by the execution of another instr.

eg :- div $t1, $a, $b # t1 = a ÷ b
      add $t3, $t1, $t2 # t3 = t1 + t2.

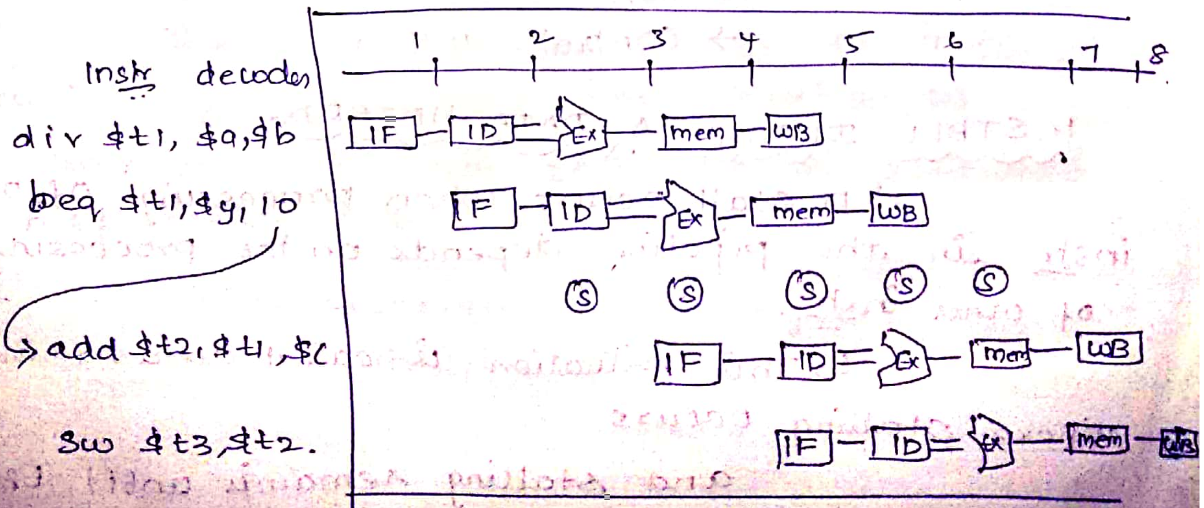


Stalling in data hazard.

(2) Stall During a control Hazard :-

↳ It is occur in pipeline when processing a branch instr and
↳ control cannot decide which instr to fetch next.
↳ This situation is known as control hazard where stalling occurs and remains until the branch instr completes its processing.

eg :- div $t1, $a, $b : t1 = a ÷ b
      beq $t1, $y, 10 : if t1 = 0
      add $t2, $t1, $c : t2 = t1 + c
      sw $t3, $t2 : t3 = t2.



The stall occur after branch instr. The control stop processing next instr until add branch instr are executed.

X — X — X

(5) Explain pipeline hazards in detail.

Pipeline Hazard

A pipeline hazard is a situation where a part of pipeline or whole pipeline is blocked or idle. In this blocked situation operation cannot be performed.

A pipeline hazard is also known as pipeline bubble.

There are 3 types of pipeline hazards/ pipeline dependencies

(1) Resource / structural hazard.
(2) control hazard
(3) Data hazard.

Stall :- A stall is a cycle in the pipeline without new i/p.

Structural dependency / Hazard :-

→ This araises due to the resource conflict in the pipeline.

→ A resource conflict is a situation when more than one instr tries to access the same resource in the same cycle.

→ A resource can be a register, memory or ALU.

Eg :-

Instr/ cycle	1	2	3	4	5
I_1	IF(mem)	ID	Ex	mem	
I_2		IF(mem	ID	Ex	
I_3		✗	IF(mem)	ID	Ex
I_4.				IF(mem)	ID.

→ In the above scenario, in cycle 4, Instr I₄ are trying to access same resource (memory) which introduces a resource conflict.

↳ To avoid this problem, we have to keep the instr on wait until the required resources (memory in our case) becomes available.

→ This wait will introduce stalls in the pipeline as shown below:-

Inst/ cycle	1	2	3	4	5	6	7	8
I_1	IF (mem)	ID	Ex	Mem	WB			
I_2		IF (mem)	ID	Ex	Mem	WB		
I_3			IF (mem)	ID	Ex	Mem	WB	
I_4				-	-	-	IF (mem)	ID

Solution for structural dependency/hazard:

↳ To minimize structural hazard stalls in the pipeline, we use a hardware mechanism called Renaming.

Renaming:-

⇒ According to renaming, we divide the memory into two independent modules used to store the instr and data seperately called code Memory (CM) & data memory (DM) respectively.

⇒ CM will contain all the instr

⇒ DM will contain all the operands that are required for the instruction.

Instr/ cycle	1	2	3	4	5	6	7
I_1	IF(cm)	ID	Ex	Dm	WB		
I_2		IF (cm)	ID	Ex	DM	WB	
I_3.			IF (cm)	ID	Ex	Dm	WB
I_4				IF (cm)	ID	Ex	Dm
I_5					IF (cm)	ID	Ex.
I_6						IF (cm)	ID
I_7							IF (cm)

II. CONTROL HAZARD (BRANCH HAZARD)

→ This type of dependency / hazard occurs during the transfer of control instruction such as BRANCH, CALL, JMP etc.

→ On many instr architectures, the processor will not know the target address of these instr when it needs to insert the new instr to the pipeline.

→ due to this, unwanted instr are fed in the pipeline.

Eg:- Consider the instr.

100 : I_1

101 : I_2 (JMP 250)

102 : I_3

250 : BI_1

Expected o/p:

$I_1 \rightarrow I_2 \rightarrow BI_1$

I_1	IF	ID	Ex	Mem	WB	
I_2		IF	ID (PC:250)	Ex	Mem	WB.
I_3			IF	ID	Ex	mem
I_4				IF	ID	Ex

<u>olp sequence</u> :- $I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow BI_1$

⇒ So, the olp sequence is not equal to the expected olp, that means the pipeline is not implemented correctly.

⇒ To correct the above problem we need to stop the instruction fetch until we get the target addr of branch instr.

⇒ This can be implemented by introducing delay slot until we get the target address.

Instr/cycle	1	2	3	4	5	6
I_1	IF	ID	Ex	Mem	WB	
I_2		IF	ID (PC: 250)	Ex	mem	WB
delay.	–	–	–	–	–	–
BI_1				IF	ID	Ex

<u>olp</u> sequence : $I_1 \rightarrow I_2 \rightarrow$ delay (stall) $\rightarrow BI_1$

are executed. (18) X — ^ — ^

As the delay slot performs no operations, this o/p sequence is equal to the expected o/p Sequence.

→ But this slot introduces stall in the pipeline

Branch penalty ⇒ The no of Stalls introduced during the branch operations in the pipelined processors is known as branch penalty.

Total no of Stalls in pipeline due to branch instr } = Branch frequency × branch penalty.

iii) DATA HAZARD:-

If in a program, 2 instr need same memory & they should be executed sequentially then data hazard will occur.

→ If one instr depends on output of other (may get old value) ⇒ data hazard.

eg:- I_1: Add R1, R2, R3

I_2: Sub R4, R1, R2.

⇒ when abv instr is executed ~~which~~ data dependency will occur
↳ which means Instr I_2 tries to read the data before I_1 writes it. ∴ I_2 gets old value [of R_1] from I_1.

Instr | cycle	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID(old value)	EX

⇒ To minimize data dependency stalls in the pipeline operand forwarding is used.

⑥

Solving data Hazard :-

Operand forwarding (bypassing)

→ In operand forwarding, we use the interface register present between the stages to hold intermediate o/p, so that dependent instr can access new value from the interface register directly.

Eg :- I_1 : Add R1, R2, R3

I_2 : Sub R4, R1, R2.

Instr/cycle	1	2	3	4
I_1	IF	ID	Ex	Dm.
I_2		IF	ID	Ex.

Types of Data Hazards :-

3 types :-

① RAW [Read After Write] [flow-data dependency]

RAW hazard occurs when instr J tries to read data before instr I writes it.

eg:- I : $R_2 \leftarrow R_1 + R_3$

J : $R_4 \leftarrow R_2 + R_3$.

② WAR [write After Read] (Anti-data dependency.

WAR hazard occurs when instr J tries to write data before instr I reads it.

eg :- I : $R_2 \leftarrow R_1 + R_3$

J : $R_3 \leftarrow R_4 + R_5$.

③ WAW (write after write) [output dependency]

WAW hazard occurs when instr J tries to write o/p before instr I writes it

eg :- I : $R_2 \leftarrow R_1 + R_3$

J : $R_2 \leftarrow R_4 + R_5$.

WAR & WAW hazards occur during the out-of-order execution of instr.

X—X—X

⑥ How to handle Control Hazard.

=> There are three methods for dealing with control hazards.

=> Control hazards occurs when the control has to decide on the instr that is fetched next; while a branch instr is executing.

To ~~co~~ handle/deal control hazard.

(1) stall pipeline

(2) Branch prediction

(3) Dynamic branch prediction.

(1) stall pipeline

↳ In this method, the pipeline processor has to stop or stall the fetch next instr until the branch instr completes its execution.

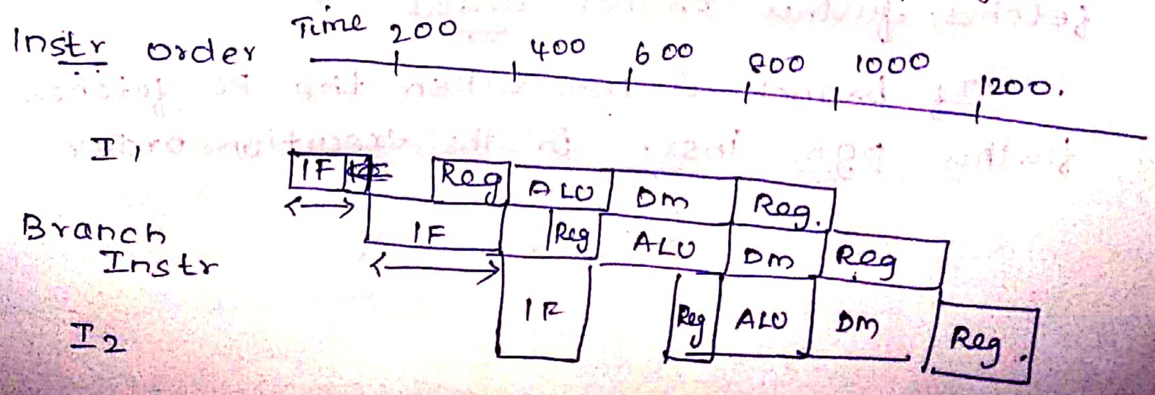
↳ Eventhough this method eliminates control hazard, this is a very ~~p~~ low process which increases the clock cycle.

(2) Branch prediction :- branch taken / branch not taken

=> This method eliminates control hazard by assuming that the conditional "branch instr will not be accepted or taken" into consideration of the control for its execution.

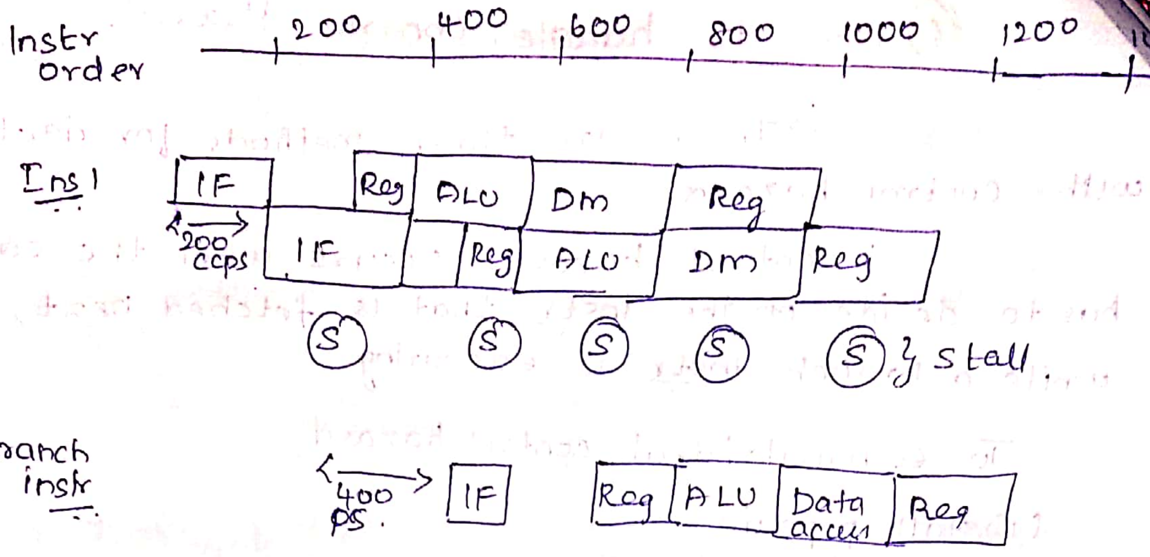
=> In some cases, if a branch instr does get taken, it means that the prediction is wrong.

↳ Then the next instr is executed in the order will be executed only after branch instr are completely executed.



(12) Pipeline Processor when "Branch NOT

Pipeline processor when "branch is Taken"



(3) Dynamic Branch prediction :-

=> Normal branch prediction technique is efficient for smaller pipelines, but with deeper pipelines, when a branch gets taken the clock cycle increases drastically.

↳ To overcome this issue, a check must be made before executing any branch instr to see whether the branch was taken or not when the last time was exeuted.

For this purpose, a special buffer called branch instr buffer or branch history table is maintained for each instr.

↳ This table helps the control to check for whether the branch is taken.

=> After this following any one task is peform.

→ If branch is taken the pgm counter (PC) fetches further branch instr

→ If branch is not taken the PC fetches further pgm instr in the execution order.

x —x— x

7) Explain in detail how exceptions are handled in MIPS architecture?

Ans:- An Exceptions are events that alter or interrupt the normal flow of execution of any program.

→ There are 2 types of exceptions that an instr can produce, in MIPS architecture

They are,

1. Execution of a undefined instr
2. " " an arithmetic overflow.

=> MIPS archi identifies which instr is causing the exception, & the reason for causing that exception.

For this purpose, this archit uses 2 special registers.

They are

(a) EPC [Exception Program Counter] → It is a 32-bit register that keep track of the address of exception causing instr in a pgm

(b) Cause Register:-

↳ It stores the reason for causing an exception in a 32-bit reg.

↳ Another method to ~~identi~~ identify the reason for causing is "Vectored Interrupts".

These vector contains a list of addr for each type of exception in hexadecimal form.

Consider below exception

(a) undefined instr → 8000 0000
(b) Arithmetic Overflow → 8000 0180.

⇒ The OS determines the cause of ~~exp~~ exception & using the addr where the exception occurred.

↳ After identifying addr & cause of exception the control is transferred to OS.

↳ The OS then uses the addr of the exception in vector or in cause reg to perform any of following action

1) Provide an appropriate exception handling service to the corresponding pgm.

2) Directly handle the exception without interfering the pgm

3) Stop the pgm execution & report an error.

Exception Handling Pipeline Implementation :-

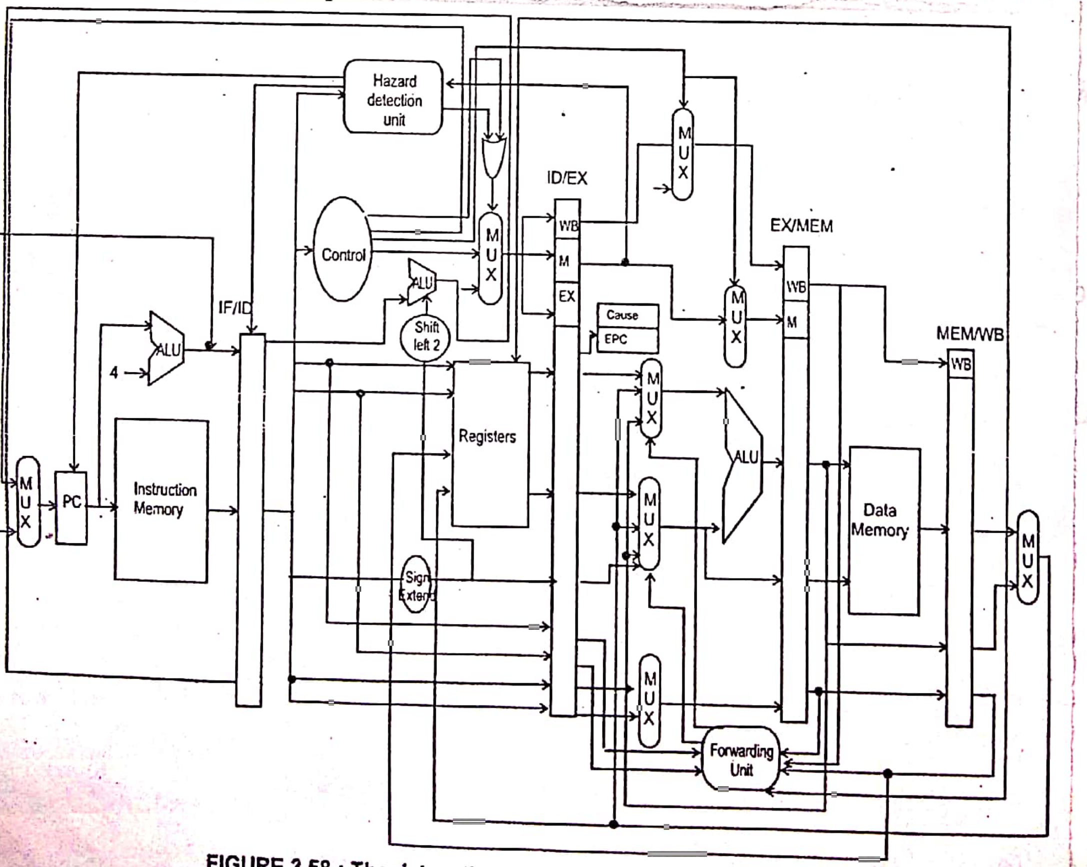


FIGURE 3.58 : The datapath with controls to handle exceptions

=> The abv datapath is a pipeline implementation of an instr execution.

=> The hexadec addr of the exception is provided to mux

=> A mux job is to select the control lines on Data path based on the instr being executed or provided.

Additional control signals at the time of exception. They are

(1) ID. Flush

↳ This control signal is used to flush away the instr in PC at ID stage.

(2) Ex. Flush

↳ This control signal is used to flush away the instr in PC at Ex stage (Exception.

↳ It is invoked when "~~Arth~~ "Arithmetic overflow" exception has occurred.

X — X — X.