Unit 3

Processor and Control unit

Basic MIPS Implementation – Building datapath – Control Implementation scheme – pipelining – pipelined datapath and control – Handling Data Hazard & Control Hazard – Exceptions.

~~Basic MIPS Implementation:-~~

Introduction to MIPS processor:-

MIPS stands for Microprocessor without Interlocked pipeline stages) or (Million Instruction per Second) is a RISC (Reduced Instruction set computer) developed by MIPS technologies. The MIPS ~~Acc~~ Instruction Set architecture (ISA) is a RISC based Microprocessor Architecture.

The early Architectures of MIPS were MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS 32 & MIPS 64. The current ~~Version~~ revisions are MIPS 32 (for 32 bit Implementation) and MIPS 64 (for 64 bit Implementation).

characteristics of MIPS:-

* Load and Store architecture
* General purpose registers (32 Registers)
* ALU operations have 3 operands (2 source + 1 destination)
* Simple Instruction set (RISC)
* Design for pipelining efficiency Including fixed Instruction set encoding.
* Simple branch operations.

Registers for MIPS :-

* MIPS 32, has 32bit, 32 General Purpose Registers (GPRs) named $R_0 \cdots R_{31}$
* General purpose Registers are also called Integer Registers.
* Additionally there are 32 floating point Registers (FPRs)
* Both single and double precision floating point operations are provided.

MIPS Addressing Modes :-

→ Immediate Addressing.

eg: addi $t0, $s0, 4.

→ Register Addressing

eg: add $s0, $s1, $s2

→ Base Addressing (Displacement Addressing)

eg: lw $s0, 16($s1).

→ Pc Relative Addressing

eg: bne $s0, $s1, target

→ Pseudo direct Addressing.

eg: j target Address

A Basic MIPS Implementation :- (First Write About MIPS & characteristics)

The basic MIPS implementation that includes a subset of the core MIPS instruction set.

* The Memory Reference Instructions Load Word (LW) and Store Word (SW).
* The Arithmetic - Logical Instructions add, sub, and, or, and slt.
* The instruct

This subset doesn't include all the integer Instructions for eg Shift, multiply and divide) & floating point instruction.

There are 5 steps for executing instructions

Step 1: Fetch Instruction
Step 2: Instruction Decode & Read Register.
Step 3: ALU operation, Branch Address computation
Step 4: LW/Store in Data Memory
Step 5: Register write.

A Overview of the MIPS Implementation:-

The core MIPS instruction include Integer arithmetic-logical Instruction, Memory-reference instructions, and the branch instructions. The need is to implement these instruction in the same and independent way.

For every instruction the following 2 steps are identical.

1. Sent the program counter (PC) to the memory that contains the code and fetch the instruction from that Memory.
2. Read one or two Registers, using fields of the instructions to select the registers to read.

For Load Word instruction we need to read only one register. But most instruction required 2 Registers.

After the above 2 steps the action to be completed depends on the instruction class.

Three commonly used instruction classes are

1. Memory Reference
2. Arithmetic - Logical
3. Branch.

Use of ALU

There are some similarities in different Instruction. For eg: All instruction class except jump, use arithmatic - Logic Unit (ALU) after Reading the Registers.

→ The Memory Reference instruction uses ALU for the address calculation. The branch instruction use ALU for comparison of the condition.

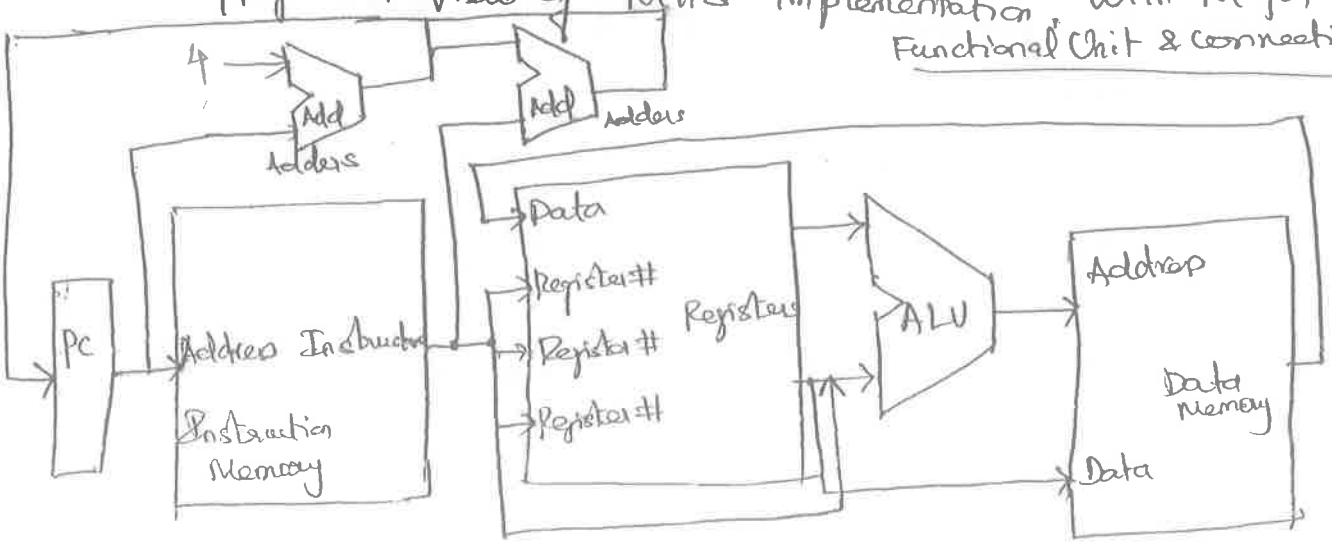
→ The Action Required to complete Various instructions

→ Memory Reference instruction need Access to Memory (Load Word) or Write data to Register for (Store Word)

→ ALU must write the data from ALU to Register.

→ For Branch we need to change the next instruction address based on comparison (or) PC must be increment to 4 to get address of next instruction.

High Level View of MIPS Implementation with Major Functional Unit & connections :-



Explanation:-

→ All instructions start by using program Counter to supply the instruction Address to the Instruction Memory.

→ After instruction is fetched the Register operands are read or fetched.

→ then operated on ALU for either Memory Address Calculation or Arithmetic operation or compare operation.

→ if Arithmetic Operation → result is written to the Register.

→ If Load and Store the ALU result is used as an address either for Loading data from memory or store data into memory.

→ Adders are used for either Calculating the branch address or increment the program counter.

In this diagram the data goes to a particular unit as coming from two different sources. for eg: the value written into the pc can come from one of two adders. In practice thes data lines cannot simply be wired together we must add a logic element that choose from among the multiple sources and chooses one of those sources to its destination.

This selection is commonly done with a device called a Multiplexor or a data Selector.

Logic Design Conventions :-

The datapath elements in MIPS Implementation consists of two types of logic elements.

1. elements that operates on Data Values eg: Combinational ~~Units~~ Elements
2. elements that contain State. eg: State elements ~~Sequential units~~.

Combinational elements :-

The output depends only on the current inputs. For a given input the Combinational element produces the same output. It doesnt have internal storage.

eg: ALU.

State ~~Sequential~~ elements :-

A State element has some internal ~~Stog~~ storage. It is called as State element because if we pull the power plug of the computer we can restart the computer by loading the State elements contained before.

* It is also called as sequential because the output depends on both current input and previous output.

eg: Memories, Registers, Flipflops.

There are two ways of indicating a signal :-

Asserted - Indicate a signal that is logically high.
deasserted - Indicate a signal that is logically Low.

Clocking Methodology :-

Clocking Methodology defines when a signal Can be read and when it Can be written.

The time for the read and write is important because, when a signal is reading and writing at same time then it reads the old value or new value (ie) It reads the wrong data.

Edge-triggered Clocking :- Any value stored in sequential logic element is updated only on a clock edge.

Any collection of combinational element (ALU) must have its input Coming from set of state elements (ie) (source Registers) and its output written into a set of state elements (ie destination Register) It is represented in the diagram below.

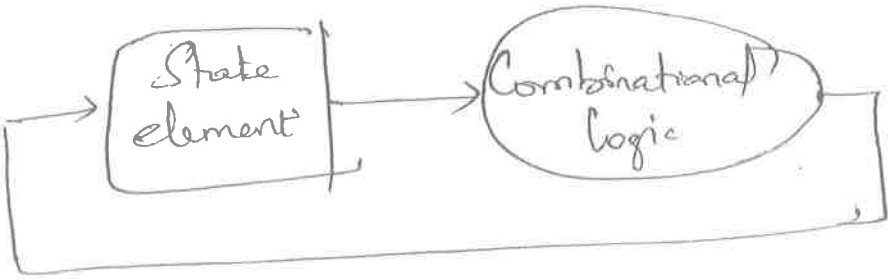
In this diagram the two state elements surrounding a block of combinational logic operates on single clock cycle.

All signal propagate from state element 1 through the combinational logic ~~&~~ to state element 2 in the time of 1 clock cycle.

If a state element is not updated on every clocks cycle then an explicit write control signal is required.

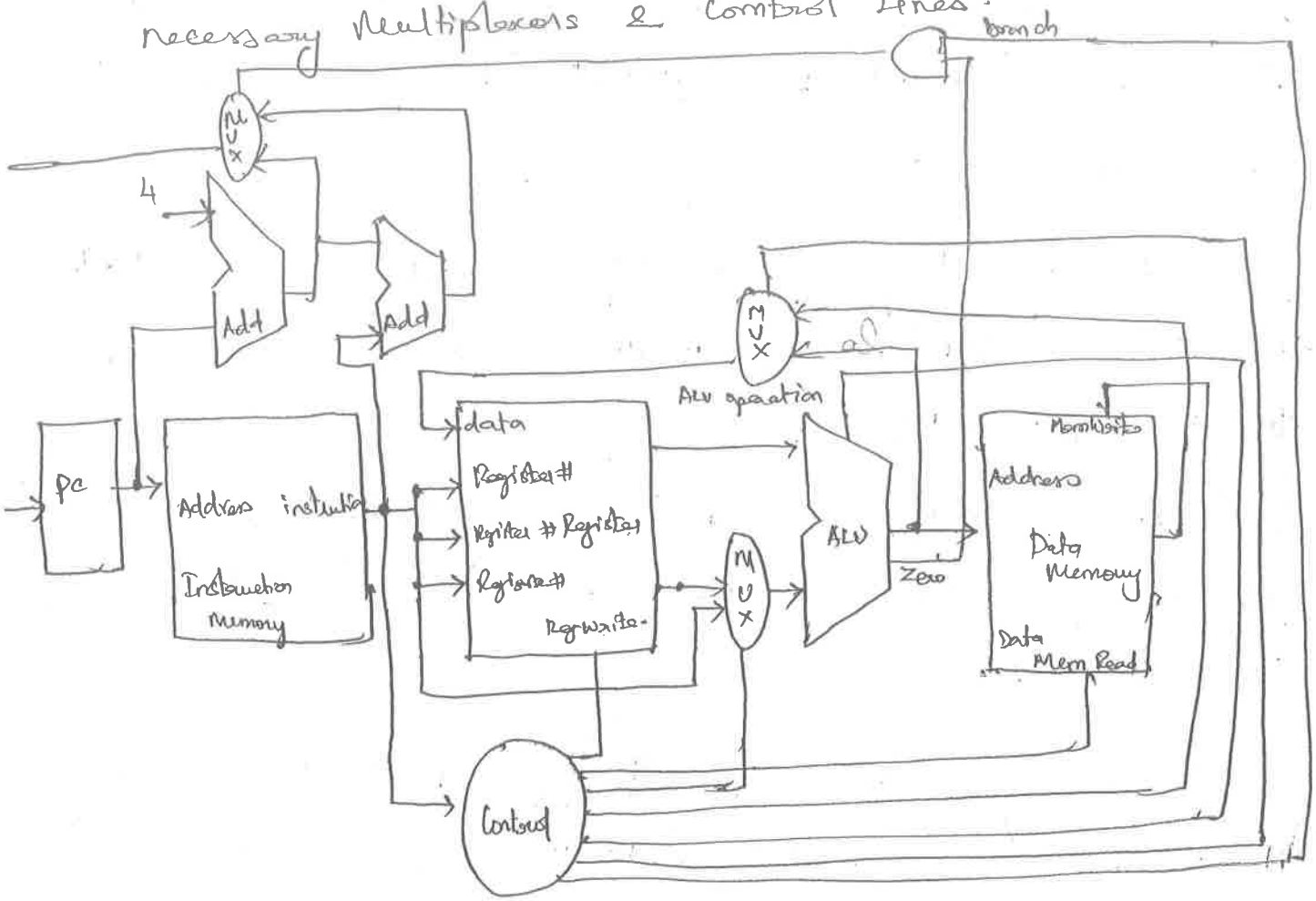
whereas in edge-triggered methodology it allows to read the content of Register and send a value through some combinational logic and write that Register in same clocks cycle

1) Edge-triggered Methodology



An edge triggered ~~Methodo~~ Methodology allows a state element to be read and write in same clock cycle. Feedback cannot occur within one clock cycle because Edge trigger updates state element

The Basic Mips Implementation including the necessary Multiplexors & Control Lines.



The top MUX controls what value replaces the PC (PC+4 or the branch destination address) The Multiplexor is controlled by the gate that "ANDs" the zero output of ALU and a control signal that indicates that the instruction is a branch.

The middle multiplexor whose output returns to the register file, it is used to steer the output of the ALU.

The bottommost multiplexor is used to determine whether the second ALU input is from the Register or from the offset field of the load & store instruction.

```
L1: Bne $s1, 2 : L2      #aa| = 2 } branch 1
    add $s1, $s0, $s0    #aa = 0 }
L2: Bne $s2, 2 : L3      #bb| = 2 } branch 2
    add $s2, $s0, $s0    #bb = 0 }
L3: Bne $s1 $s2 L4       #aa| = bb -> branch 3
```

Here the branch 3 is depend on the values of the previous branchs, this type of branches can be predicted using <u>correlating branch prediction</u>.

Tournament predictor (or) Multilevel Predictor :-

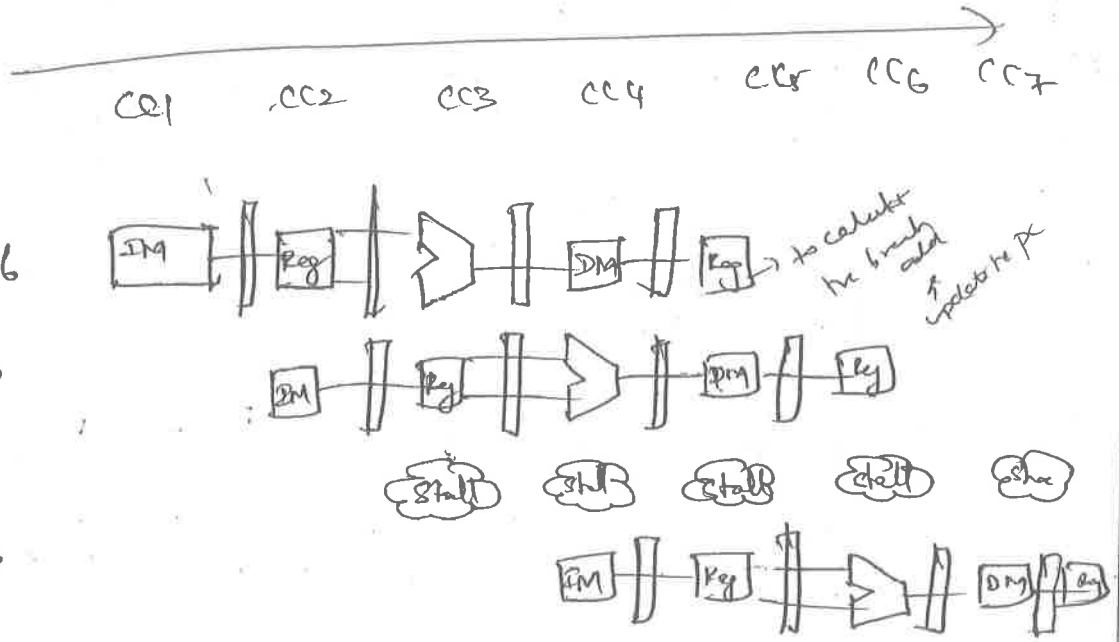
It uses multiple predictors for each branch. A tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behaviour. A selector would choose which predictor to use for any given prediction. The selector can can operate similarly to a 1 or 2 bit predictor, favoring which of the two predictors has been more accurate.

Delayed Branch :-

A delayed branch always executes the following instruction, but the second instruction following the branch will be affected by the branch.

Stalling:

Control Hazard can be overcome by stalling. This stalling can be done immediately after we fetch a branch waiting until the pipeline to determine the outcome of the branch.



Here we assume that we put an extra hardware so the we can test Registers to calculate branch address and update the program counter during the second stage.

If we cannot resolve the branch in the second stage (ie) in the case of longer pipeline so even longer stalls will be required.

Here we look at two schemes for resolving control Hazard and one optimization to improve this scheme.

1 Assume Branch Not Taken:-

As we saw before stalling until the branch is complete, is too slow. One common improvement over branch stalling is to Predict that the branch will not be taken and thus continue execution down the sequential Instruction stream.

If the branch is taken then the instruction that are being fetched and decoded must be discarded.

fetched decoded

(To discard the instruction, we merely change the original control values to 0s.) (i)

(We must change the three instruction in the IF, ID and EX stages when the branch reaches the MEM stage. [for Load-use stall]) (ii)

(We just change control to '0' in ID stage and let them to percolate through the pipeline.

Discarding instruct means 'flushing' the instruction in IF, ID, EX stages of pipeline.

2. Reducing the Delay of Branches :-

One way to improve branch performance is to reduce the cost of the taken branch. So far we assumed that next PC for branch is selected in MEM stage, but if we move the branch execution earlier in pipeline then fewer instruction need to be flushed.

Moving the branch decision up requires two action to occur earlier.

1. Computing the branch target Address
2. Evaluating the branch decision.

It is easy to compute the branch target Address, because we have the PC Value and immediate field in the IF/ID pipeline Register so we just move the branch adder from EX stage to the ID stage, so the branch target address Calculation will be performed for all instructions.

The Hardest part is evaluating the branch Decision. Here we are Moving the branch test to the ID stage which requires additional

forwarding unit and Hazard detection hardware. Equality test (ie) 'beq' can be tested by EXORing the bits and then ORing the result. If in case of branch on equal and branch on not equal the Register Value is depend on the instruction prior to it then Forwarding Unit and Hazard detection harware is needed.

BY bits OR

There are two complicating factors:-

1) During ID, we must decode the instruction, decide whether a bypass to the equality unit is needed. The bypass of source operands of a branch can come from either the ALU / MEM or MEM / WB pipeline Latches.

2) The values in branch comparison is needed during ID but it will produced by the previous instruction at later in time so it is possible that 'data Hazard' occur and a stall will be needed.

ALU + Branch → operand

If ALU instruction proceeds the branch instruction produces one of the operand, a stall will b required.

If ~~the~~ Load instruction precedes the branch instruction produces ~~the~~ one the operand two stall cycles will be needed because the operand is available only after MEM.

ID BR Despite these difficulties, Moving the branch execution to the ID Stage is an improvement because it reduces the penalty of a branch to only one instruction if the branch is taken.

To flush the instruction in the IF Stage we add a control line called IF.Flush that zero the instruction field of the IF/ID pipeline Register.

Dynamic Branch Prediction :-

In case of Static branch prediction (ie) prediction during the Compile time it wastes too much of performance because if the prediction to be taken is wrong then it has to flush all the instruction which started execution.

To overcome this Dynamic Branch prediction was found (ie) Predicting the branch behaviour during Execution time or runtime.

Dynamic branch prediction :-

Dynamic branch prediction is also called as "run time prediction". In this prediction the address of the instruction is looked up to see whether the branch was taken. If the branch was taken begin to fetch new instructions from the same place as the last time.

Branch prediction buffer or Branch History Table :-

The Implementation for the Dynamic branch prediction is achieved by using Branch History table.

The branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction.

(The memory contains a bit that says whether the branch was recently taken or not.)

Based on the bit it is classified into two types,

1. One-bit prediction
2. two-bit prediction.

One-bit prediction :-

(The Memory contains one bit which says whether the branch was recently taken or not)

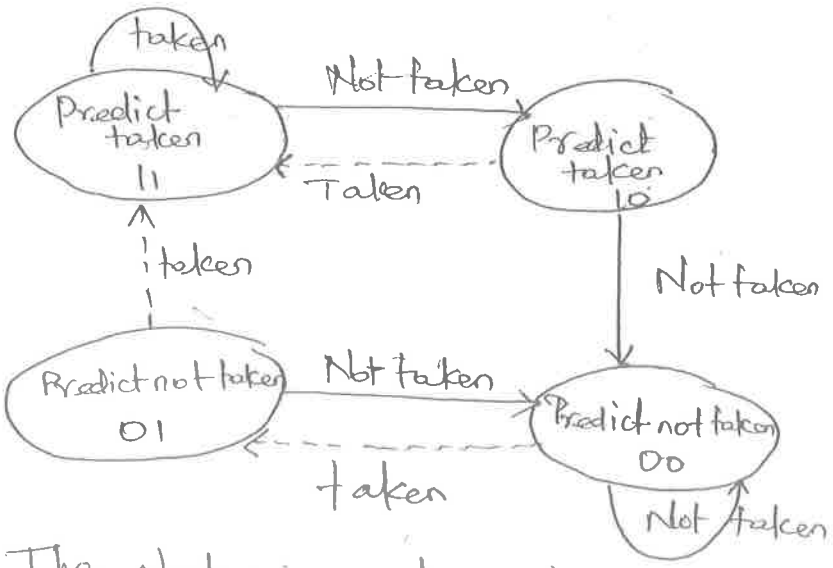
The prediction is a hint that is assumed to be correct,

at and fetching begins in the predicted direction.. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

The one-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrect twice, rather than once, when it is not taken.

two-bit prediction:

The remedy to one-bit prediction is two-bit prediction scheme is used. In a two-bit scheme, a prediction must miss twice before it is change.



The states in a two-bit prediction scheme.

In the diagram the two bits are used to encode the four states in the system.

The counter In this counter implementation the counters are incremented when a branch is taken,

and decremented when it is not taken. The counters saturate at 00 or 11.

One complication of the two-bit scheme is that it updates the prediction bit more often than a one bit predictor.)

There are also n-bit predictors, with n-bit counter is used. The counter can take values between 0 to $2^n - 1$.

When compared to n-bit predictor, 2 bit predictor do almost well.

Correlating branch predictors :- (or) Two level predictor :-

These two-bit predictor schemes use only the recent behavior of a single branch to predict the future behaviour of the branch. In correlation Branch prediction we also look at the recent behaviour of others branches rather than just the branch we are trying to predict.

Consider the code segment

```
if (aa == 2)
aa = 0;
if (bb == 2)
bb = 0;
if (aa != bb) {
```

```
L1: Bne $s1, 2 : L2      # aa! = 2 } branch 1
    add $s1, $s0, $s0    # aa = 0 }

L2: Bne $s2, 2 : L3      # bb! = 2 } branch 2
    add $s2, $s0, $s0    # bb = 0 }

L3: Bne $s1 $s2 L4       # aa! = bb -> branch 3
```

$s1, 2 : L2$
$\$s1, \$s0, \$s0$
$Bne\ \$s2, 2 : L3$
$add\ \$s2, \$s0, \$s0$

Here the branch 3 is depend on the values of the previous branchs, this type of branches can be predicted using <u>correlating branch prediction</u>.

Tournament predictor (or) Multilevel Predictor :-

It uses multiple predictors for each branch. A tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behaviour.

A selector would choose which predictor to use for any given prediction. The selector can can operate similarly to a 1 or 2bit predictor, favoring which of the two predictors has been more accurate.

Delayed Branch :-

A delayed branch always executes the following instruction, but the second instruction following the branch will be affected by the branch.

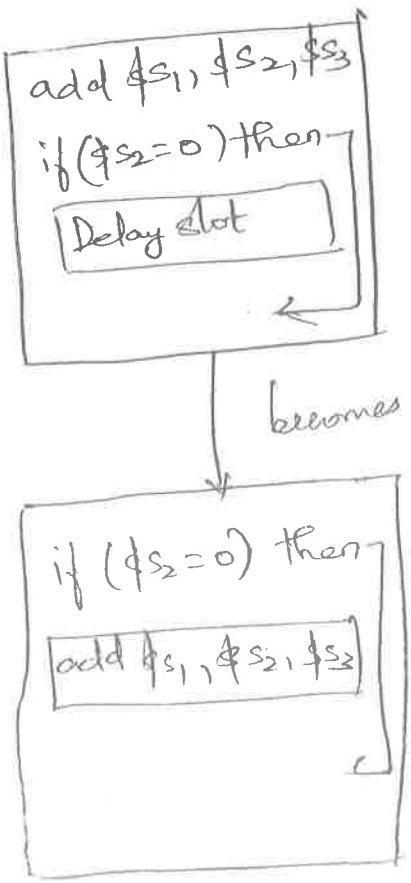
Compilers and assemblers try to place an instruction that always execute after the branch. (ie) The slot directly after a delayed branch instruction which is filled by an instruction that does not affect the branch is called "branch delay slot

The limitation on delayed branch scheduling arise from 1) the restriction on the instruction that are scheduled into the delay slot.

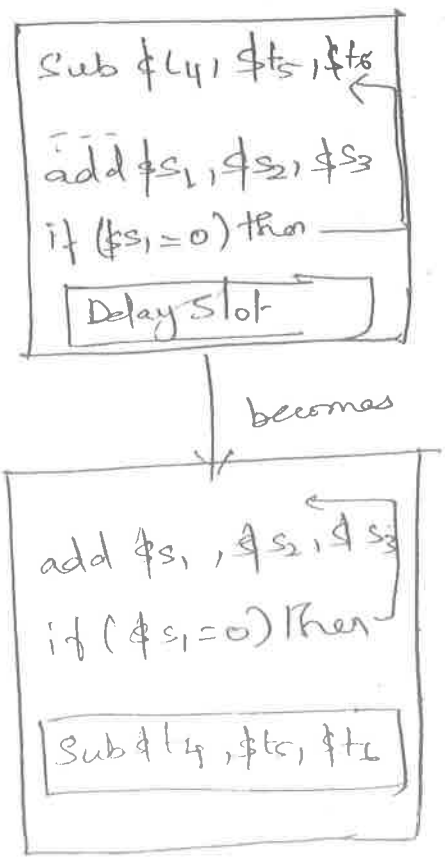
2) our ability to predict at compile time whether a branch is likely to be taken or not.

Scheduling the branch delay slot

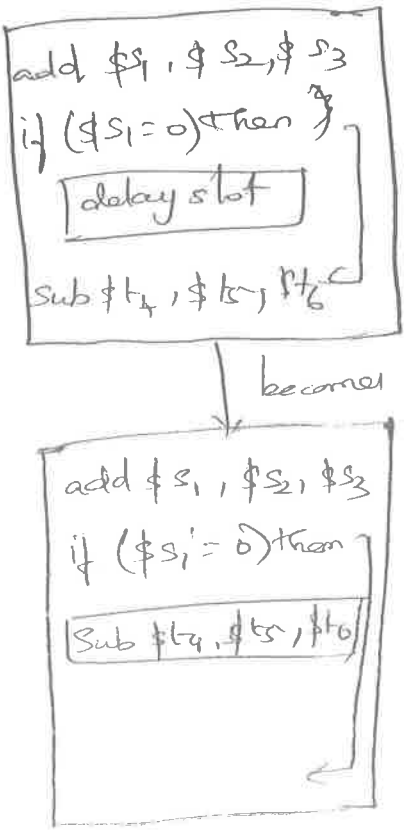
a) from before



b) from target



c) from the fall through



Exceptions

Exceptions :- (Internal Events)

→ It is also called as interrupt, An Unscheduled event that disrupts the program execution.

→ It is used for detecting overflow.

→ It is hard to implement Exceptions and Interrupts.

Exceptions are classified into two types:
- Imprecise Exception : These exception are not associated with the exact Instruction
- Precise Exception : These exceptions are associated with the exact Instruction

Interrupts :- (External Events).

These are the exceptions that comes from outside of the processor. They are also called as external events.

Following are the example that distinguishes the interrupts & exceptions

Type of Event	from where ?	MIPS terminology
1) I/O Device Request	External	Interrup
2) Invoke operating system from user program	Internal	Exception
3) Using an Undefined instruction.	Internal	Exception
4) Hardware Malfunction	Internal / External	Interrupt / Exception

Without proper attention to exception during design of the control unit attempts to add exceptions to a complicated implementation which reduces the performance.

How Exceptions are Handle in MIPS Architecture:

The two type of exceptions that the MIPS implementation generates are ① Execution of Undefined instruction ② Arithmetic Overflow.

① Undefined Instruction - Use of an Undefined Instruction (ie. Using Instruction that is not supported by the Architecture)

② Arithmetic Overflow - If the result of an Arithmetic operation is to large to occupy the Register.

Actions to be performed by a processor when a Exception Occur:

Consider an arithmetic overflow in Instruction add $1,$2,$3 then the processor perform the following tasks.

i) Every Instruction will be having some address or each instruction will be stored in memory in a different

is to save the address of the offending instruction in the "Exception Program Counter" (EPC).

2) Transfer the control to the Operating System.

3) The operating system then handles the exception by

(a) taking appropriate action like providing some i) service to user program, ii) taking some predefined Action in response to an overflow, iii) Stopping the execution of the program and iv) reporting error.

4) After performing the action the operating system can either

1) Terminate the program

2) Continue its execution by restarting the execution from the address present in the EPC (Exception program Counter).

Handling of Exception by Operating System:-

The Operating System must know the reason or cause of the exception in order to handle the exception eg: for reporting an error

There are two Main Methods used :-

→ Include a status Register (Cause Register) which holds the reason for the exception

→ Use Vectored interrupt.

~~In Vectored Interrupt the Address to which the Operating system control is transferred is determined by the Cause of exception~~

~~the~~

In Vectored interrupts the address to which the operating systems control to handle the exception is determined by the cause of exception.

Exception type	Exception Vector Address (in Hex)
Undefined Instruction	8000 0000 hex
Arithmetic overflow	8000 0180 hex

when the exception is not Vectored, the operating system decodes the status register to find the Cause.

MIPS Implementation:-

We need to add two additional registers to the MIPS Implementation

EPC (Exception Program counter) :- A 32 bit reg used to hold the address of instr.

A 32 bit Register Used to hold the address of the affected instruction

01010
01100

Cause Register (cause) :-

It is a Register Used to record the Cause of the exception. This Register is 32 bit. Some bits in the 32 bit are unused.

Assume there is a 5bit field that encodes the two possible exception sources mentioned,

5bit

01010 ⇒ 10 → represents Undefined Instruction

01100 ⇒ 12 → represents Arithmetic Overflow.

Exceptions in pipelined Implementation :-

A pipelined implementation treats exception as another form of Control Hazard

Suppose an arithmetic Overflow in add instruction takes place, we must flush (stall) the instructions that follows the add instruction from the pipeline and begin fetching instruction from the new address. ((ie) Saving Address in EPC and Moving control to Operating System).

EPC → OS

Change in the Pipeline Hardware :-

ID-stg - flush

→ To flush the instruction in the ID stage we use the Multiplexor in the ID stage to '0' control signal for stall.

ID flush - OR-d.

→ A new control signal called ID. Flush is ORed with stall signal from the Hazard detection Unit to flush during ID.

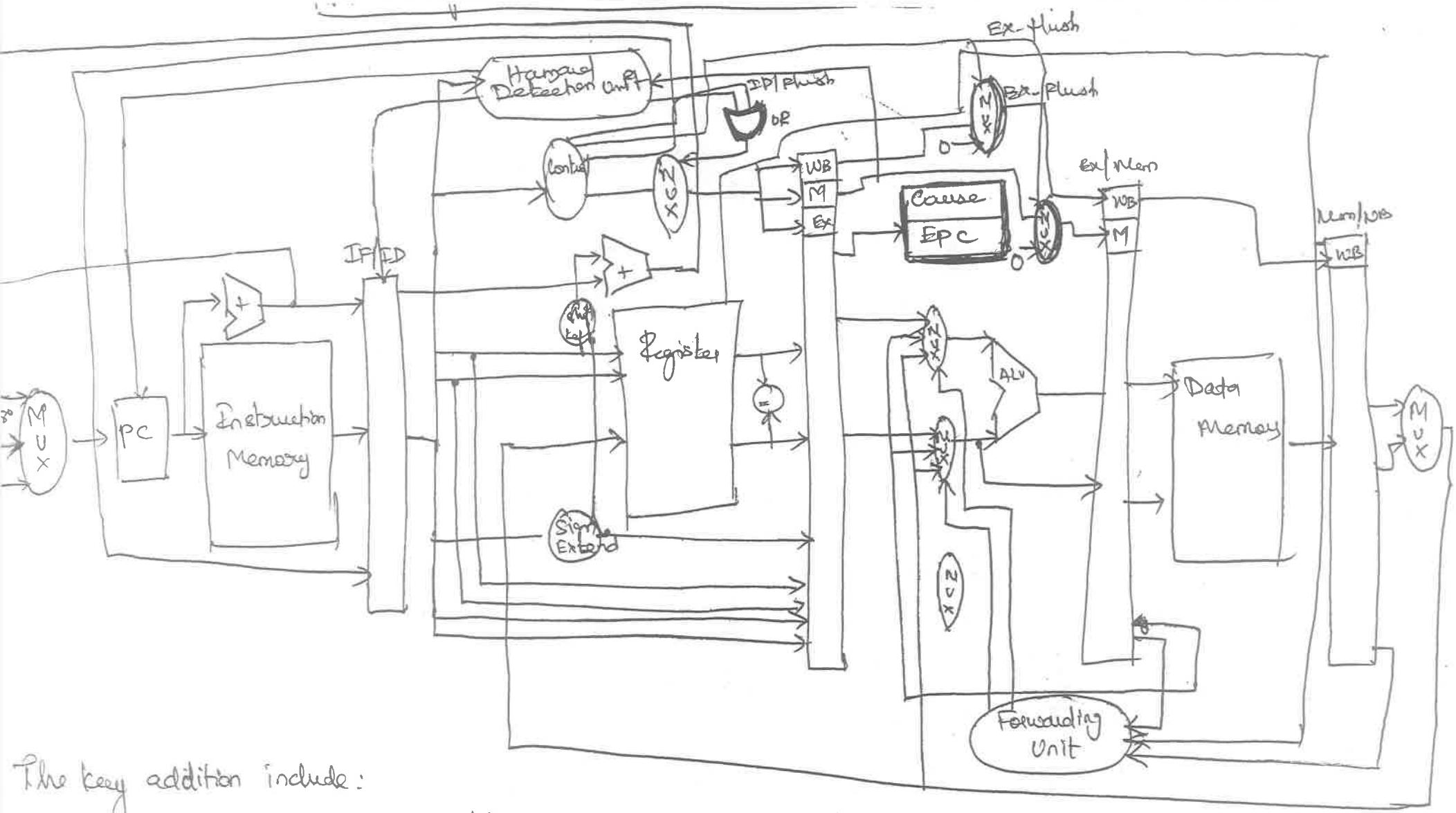
EX-phase

→ To flush the instruction in the EX - phase we use new signal called EX. Flush to cause new multiplexor to zero the control line.

EX-flush - mux D.

→ To start fetching the instruction from location $8000\ 0180_{hex}$ which is MIPS exception address, 8000 0180 hex. So we simply add an additional input to the + PC Multiplexor that sends 8000 0180 here to the PC.

→ Save the address in the EPC (Exception program counter). We save as the 'address + 4' so exception handling routine must subtract 4 from the saved value for restarting the execution.



The key addition include:

→ Cause Register to record the cause

→ New Input with value 8000 0180 hex

→ EPC Register to save the address.

↳ Refer the diagram In book Pg No: 373

Fig: 4.60

eg:-

40_{hex} Sub \$11 \$2 \$4

44_{hex} Add \$1 \$2 \$3 (Consider Arithmetic Overflow) in Add Instruction

48_{hex} and \$4 \$5 \$6 } flushed out.

Instruction Invoked when an exception occurs:-

8000 0180_{hex} Sw \$26, 1000(\$0) ← } Exception Handling Instruction

8000 0184_{hex} Sw \$27, 1000(\$0)

Explanation:-

while executing the instructions, A exception occurs in Add \$1 \$2 \$3 instruction so $44_{hex} + 4_{hex}$ = 48_{hex} in stored in PC ← 48_{hex} then executes the instruction starting at 8000 0180_{hex}.

After completing the Exception Handling Instruction then the execution is restarted at location in EPC which is substracted by 4.

EPC − 4_{hex}

$48_{hex} - 4_{hex}$

= 44_{hex} so execution restarts in 44_{hex} address

Pipeline Datapath & Control :-

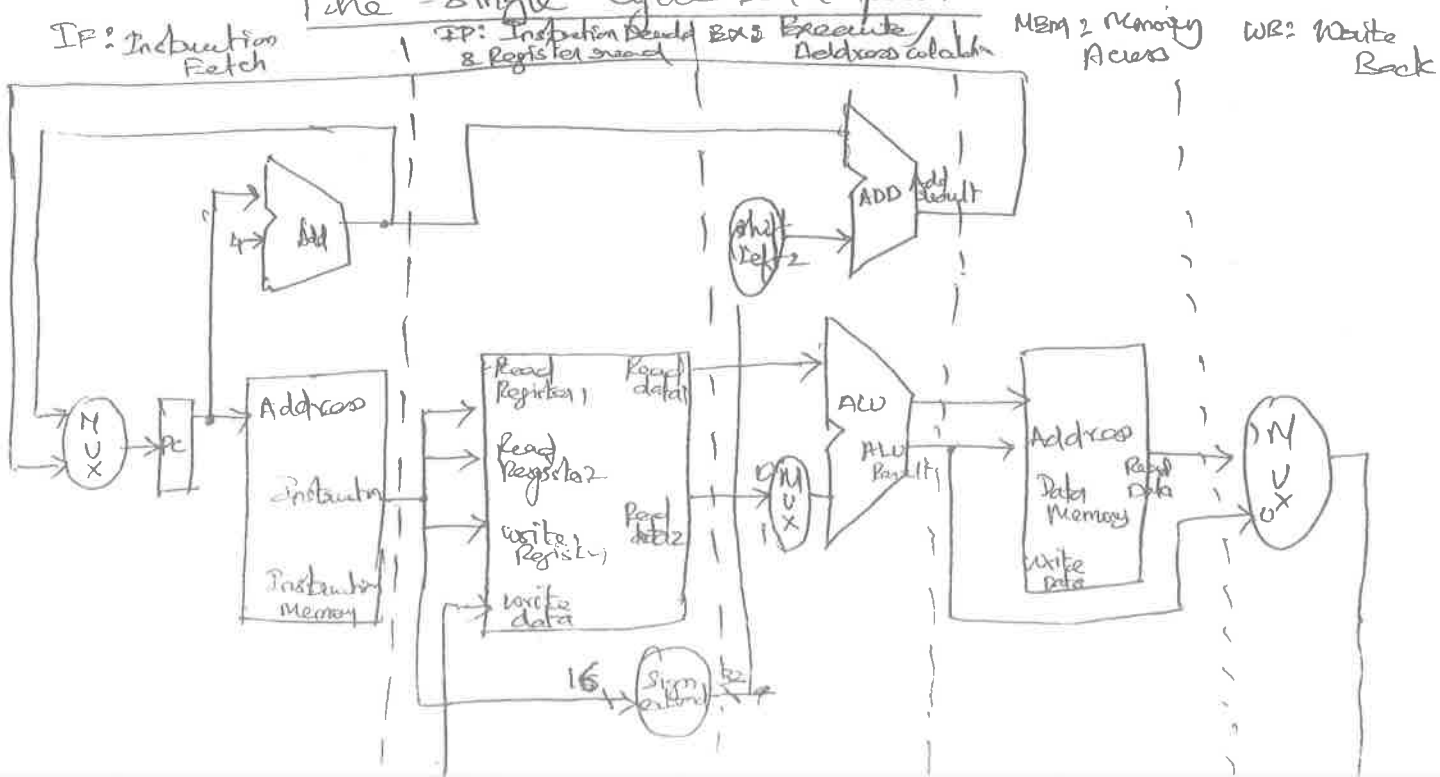
Datapath:- The components of the processor that performs arithmetic operation is referred as datapath.

In a pipeline the instruction execution is divided into 5 stages so we must seperate the datapath into five pieces where each piece is named corresponding to a stage of instruction execution.

The stages are

1. IF : Instruction Fetch
2. ID : Instruction Decode & Register file Read
3. Ex : Execution or address calculation
4. MEM : Data Memory Access
5. WB : Write Back

The Single cycle Data path.



→ The instruction and data move generally from left to right through the five stages as they complete execution.

→ There are two ~~execution~~ exceptions to this left-to-right flow of instruction.

① The write-back stage places the result back into the register file in the middle of the datapath.

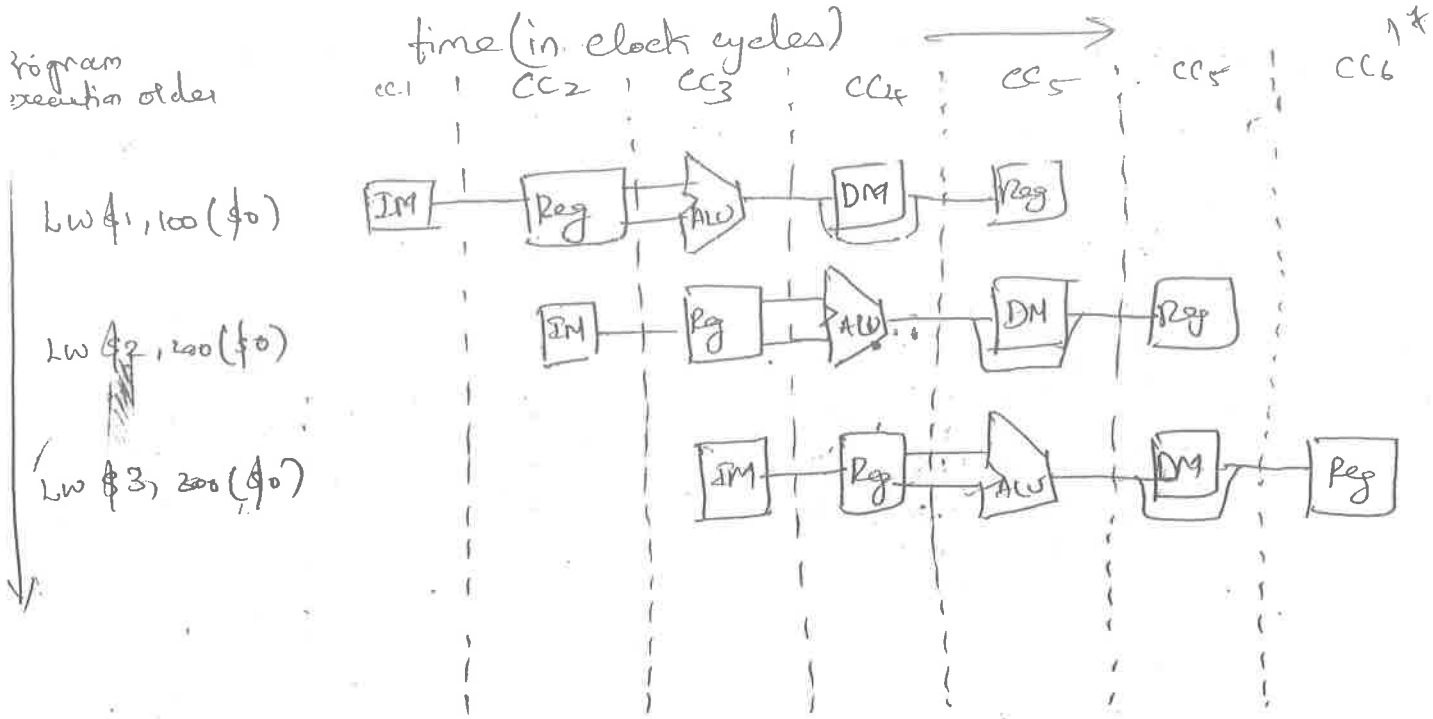
② The selection of next value of the PC chooses between the incremented PC and the branch address from the MEM stage.

→ These two are called exception because the data flow is backwards (ie) it is Right-to-left affects the execution.

→ The data flowing from right-to-left doesn't affect the current instruction, only the later instructions are influenced. Because it can lead to data Hazard or control Hazard.

Pipelined Version of Datapath :-

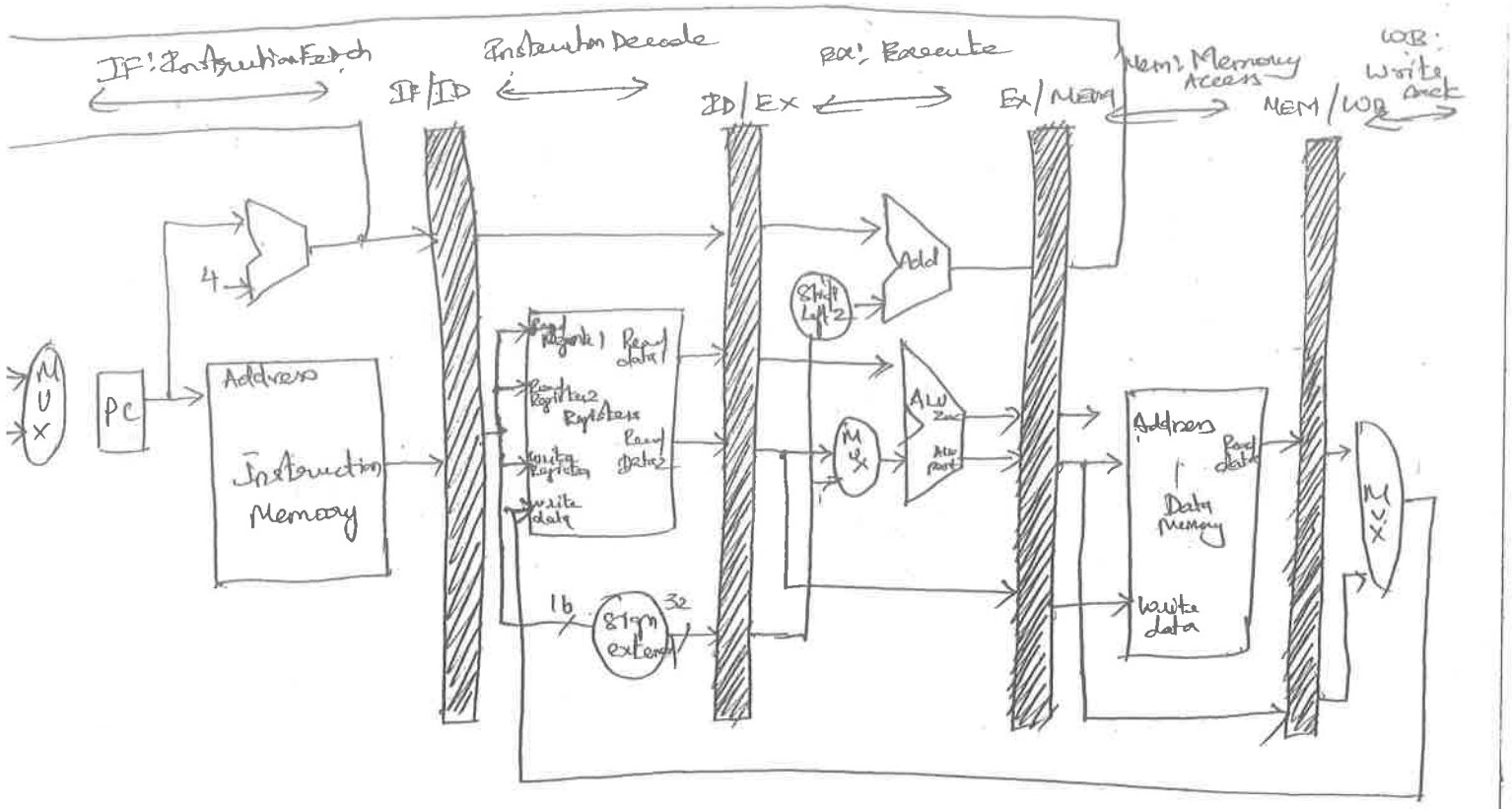
When instructions are executed, each instruction executes in seperate datapath, (ie) for eg three instruction need three datapath.



Instead of using 3 Datapath for executing 3 instructions, we adds registers to hold data so that portions of a single datapath can be shared during instruction execution.

To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similarly for all the stages we must place registers wherever there are dividing lines between stages.

All instruction advance during each clock cycle from one pipeline Register to the next. The registers are named for the two stages separated by that register. for eg. The pipeline register between the IF & ID stages is called IF/ID.



The pipelined Version of the datapath.

Pipeline Stages for the "Load Word" Instruction:-

1) Instruction fetch:- The instruction is being read from the memory using the address in the PC and the being placed in IF/ID pipeline register.
The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle.

2) Instruction Decode & Register File Read:-

IF/ID pipeline Register supplies the register Number to read the two register and a 16-bit immediate field which is sign extended to 32 bits.
All the three values are stored in the ID/EX pipeline

register along with the incremented PC address.

) Execute or Address Calculation :- The load instruction next reads the contents of register 1 and the sign-extended immediate from ID/EX pipeline Register and add them using ALU and sum is placed in EX/MEM pipeline Register.

) Memory Access : The Load instruction Reads the data from the data Memory using the address from the EX/MEM pipeline Register and Load data into MEM/WB pipeline Register.

) Write-back :- Reading the data from MEM/WB pipeline Register and writing it into the Register file.

ipeline Stages for the ~~load~~ Store instruction :-

1) Instruction fetch: The instruction is read from memory using the address in the PC and then It is placed in IF/ID pipeline Register. Similarly PC is incremented and placed in IF/ID.

2) Instruction Decode & Register File Read :- The instruction in IF/ID pipeline Register supplies the Register number for Reading two register and 16 bit immediated filed is sign extended and stored in ID/EX.

3) ~~The effective~~ Execute and address calculation :- the effective address is calculated & placed in EX/MEM pipeline Register.

4) Memory Access :- Now the data in the Register which is stored in the ID/EX is just placed in the EX/MEM pipeline register & Moved to Data Memory.

5) Write Back :- For store instruction nothing happen in this phase.

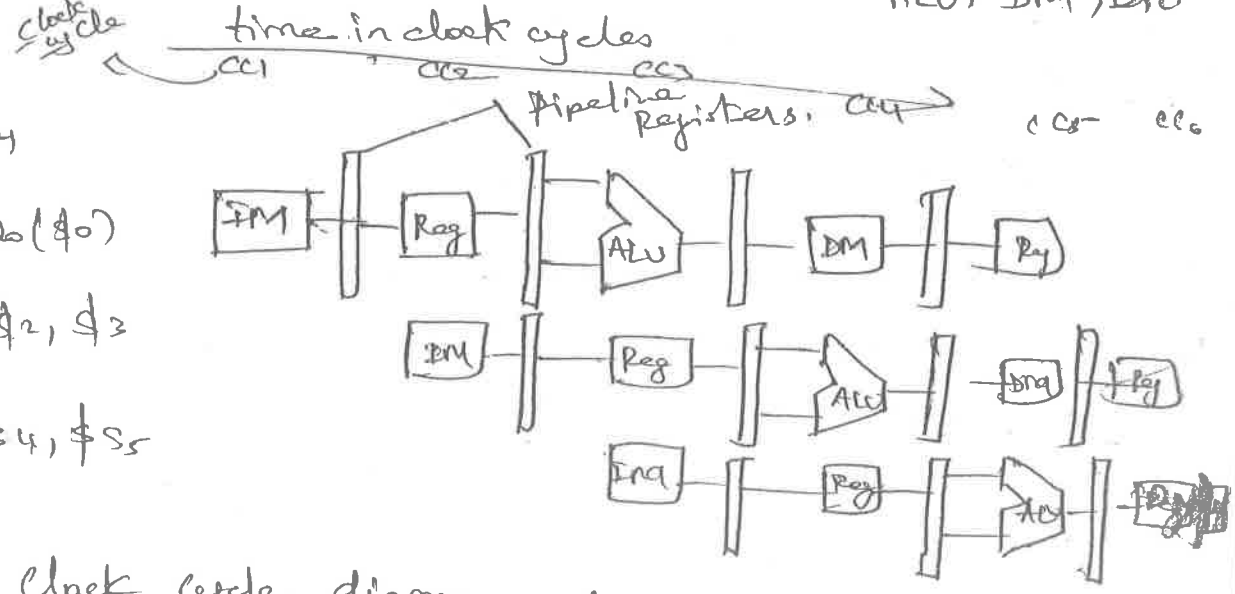
Graphically Representing Pipelines :-

~~The~~ There are two basic styles of pipeline figures,

1. Multiple-clock-cycle pipeline diagram
2. Single-Clock-cycle diagrams.

Multiple clock cycle pipeline diagram:-

In this diagram time advanced from the left to right and instruction advance from the top to bottom. pipeline Registers are placed inbetween the stages. A rectangle is used for representing each stage. It is simpler but do not contain all the detail. It gives a overview of pipeline situations. It uses Physical Resources like IM, Reg, ALU, DM, etc



Multiple Clock cycle diagram for 3 instruction.

Traditional Multiple-clock cycle pipline Diagram:-

This diagram uses the name of the each stage like Instruction fetch, Instruction Decode, Execution, Data access, Writeback.

time in clock cycle

Program execution order	CC1	CC2	CC3	CC4	CC5	CC6	CC7
Lw $10, 20($0)	Instruction fetch	Instruction Decode	Execution	Data Access	Write Back		
Add $1, $2, $3		Instruction fetch	Instruction Decode	Execution	Data Access	Write Back	
Sub $2, $4, $5			Instruction fetch	Instruction Decode	Execution	Data	Write

Single-clock cycle pipeline diagram:-

This diagram shows the state of the entire datapath during a single clock cycle.

The ~~three~~ instruction in the pipeline are identified by the label above their respective stages.

It is more detailed view and it takes more space to show the no of clock cycles.

Pipelined Control:-

Inorder to co-ordinate and control the flow of execution control is needed. The various control lue use are ALU control logic, branch logic, destination register number multiplexer and control lines.

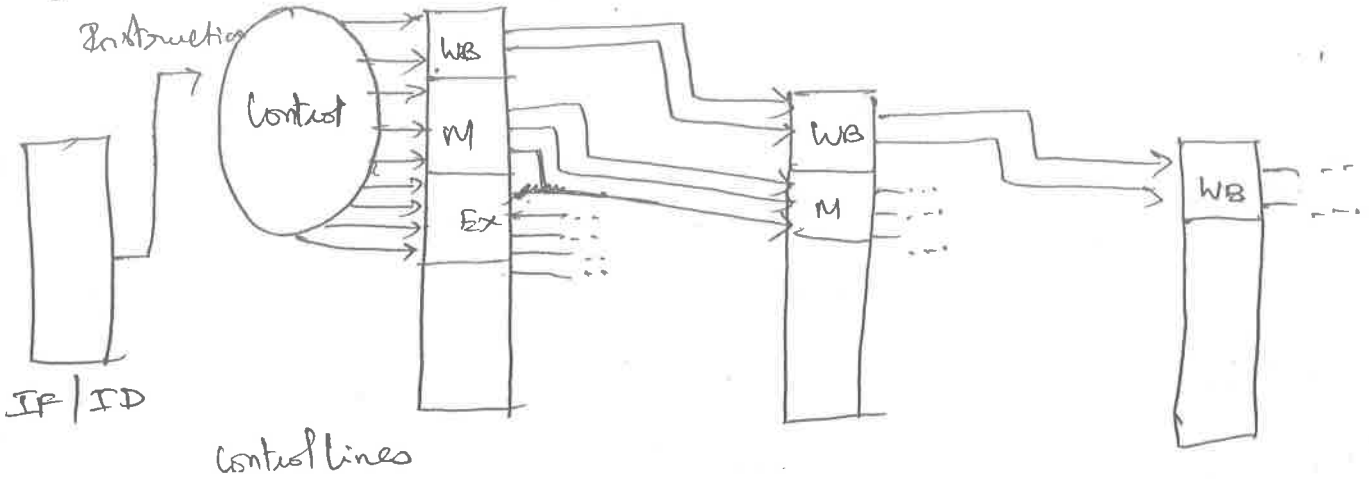
Setting up of ALU control bit depending on the ALU op control bit and function codes for R-type instruction

Instruction opcode	ALU op	Instruction operation	Function Code	Desired ALU action	ALU control Input
LW	00	Load word	xxx xxx	add	0010
SW	00	store word	xxx xxx	add	0010
Branch equal	01	branch equal	xxx xxx	subtract	0110
R-type	10	add	(32) 100000	add	0010
R-type	10	subtract	(34) 100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10		100101	OR	0001

The function of each 7 control signal is given in the table below.

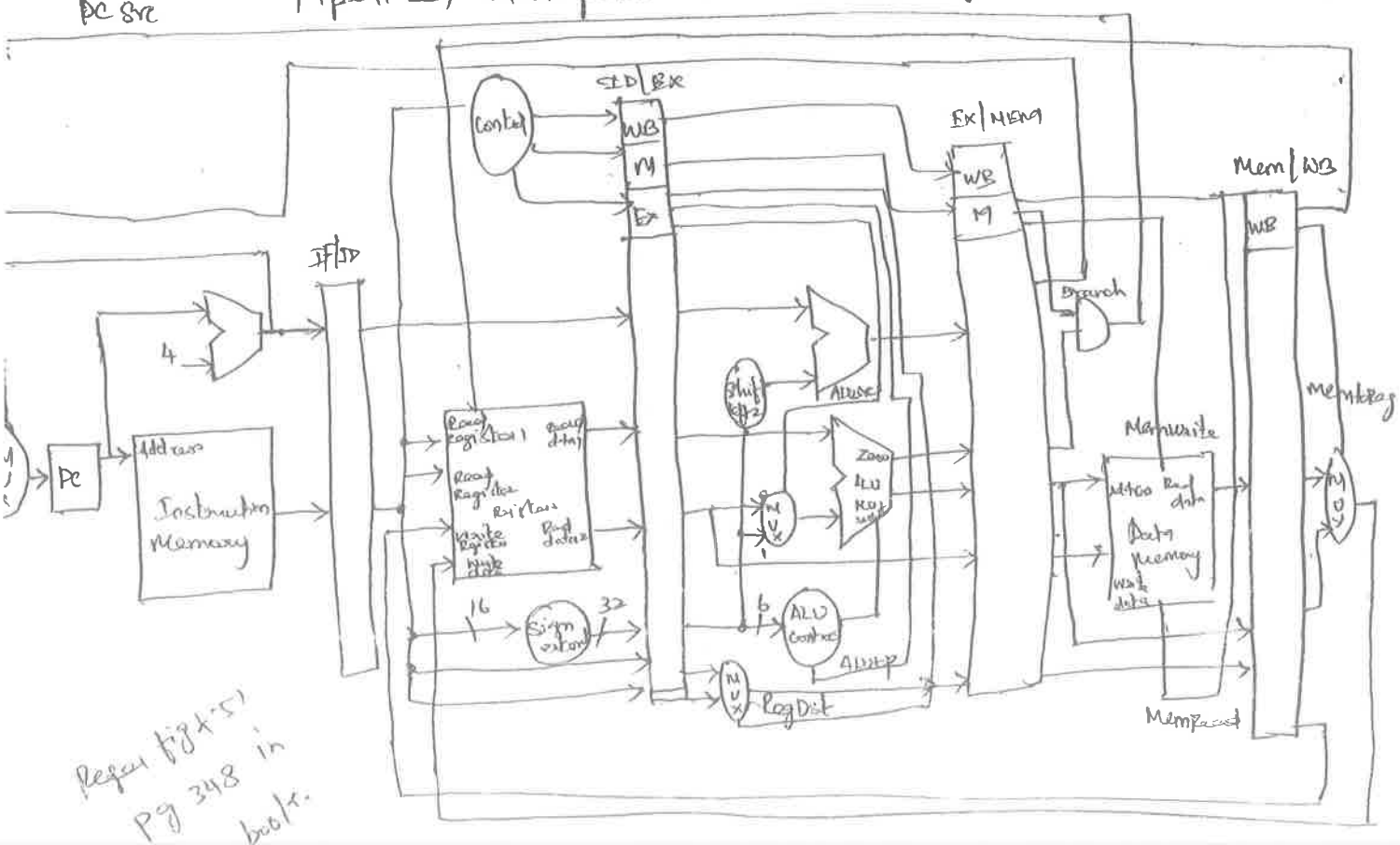
Signal Name	Effect when deasserted (0)	Effect when asserted (1)
Reg Dst ~~RegWrite~~	The register destination number for the write register comes from the rt field (bits 20:16)	The register destination number for the write Register comes from the rd field (bits 15:11)
RegWrite	None	The register on the write register input is written with the value on the write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC+4.	The pc is replaced by the output of the adder that computes branch target.
MemRead	None	Data Memory contents designated by the address input are put on the Read data output
MemWrite	None	Data Memory contents designated by the address input are replaced by the value on the write data input.
emtoReg	The value fed to the Register data input comes from the ALU.	The value fed to the Register write data input comes from the data Memory.

∴ the first two stages have no control needed, the control lines for the final three stages are given below.



Out of 9 four control lines are used in the Ex phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines, three are used during the MEM stage, and the last two are passed to MEM/WB for use in WB stage.

Pipelined datapath with control signal.



Refer fig 4.51 pg 348 in book.

Each Control line is associated with a component active in only a single pipline stage.
We can divide the control lines into 5 groups according to pipeline stages.

1) Instruction fetch: The control signal to read instruction Memory & to write pc are always asserted so no control is needed.

2) Instruction decode / Register file Read: No optional control line is needed to set.

3) Execution / Address calculation: The signal to be set are RegDst, ALUOp and ALUSrc. The signal select the result Register the ALU operation and either Read data 2 or a sign-extended immediate for the ALU.

4) Memory Access: The control line set in this stage are Branch, MemRead and MemWrite. These signal are set by the branch equal load & store instructions.

5) Write back: The two control lines are MemtoReg which decides between sending the ALU result or memory value to Register file. & RegWrite which writes the choosen value.

Building a Datapath:

Datapath elements:

A unit that is used to operate on or hold data within a processor. In the MIPS the datapath elements are
1. Memories
2. Register file
3. The ALU
4. Adders

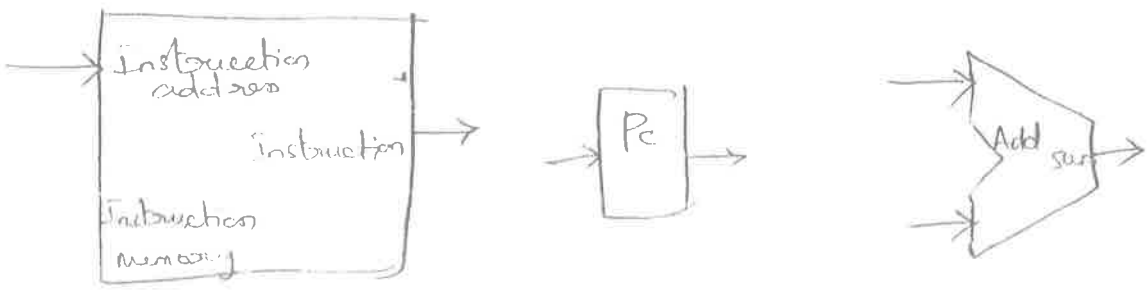
Program Counter:

It is register containing the address of the instruction currently being executed. Always the PC is incremented 4 inorder to hold the address of the next instruction to be fetched.

This increment to the PC is achieved by using an adder

The adder is a combinational logic that is build from the ALU by writing the control line so that it performs add operation.

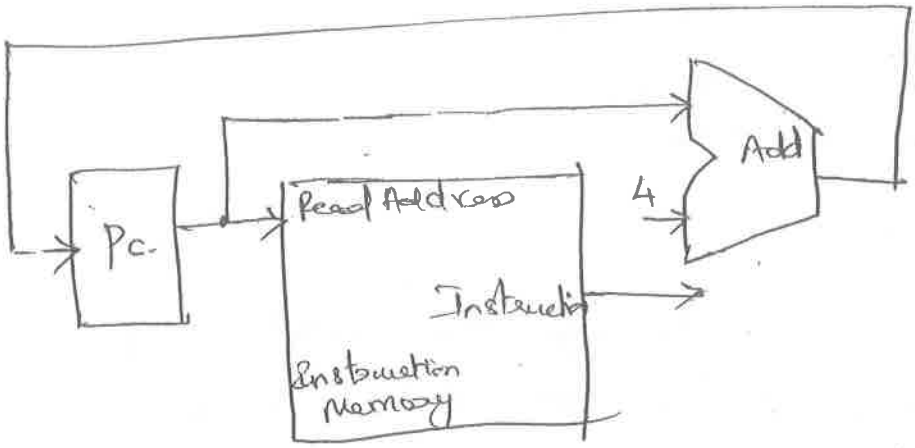
Fig1: two state element: PC & Instruction Memory (Store & Access)
Combinational element: Adder. (Compute Next instruction address)



> To execute an instruction, we must start by fetching the instruction from memory.

> Increment the PC so that next instruction to be executed is selected.

fig²: A datapath that fetches instruction and increment the program counter.



R-format Instruction:-

The R-format instruction read two registers, perform ALU operation on the contents of the register and write the result back to a Register.

This instruction class includes add, sub, AND, OR and slt.

eg: add $t1, $t2, $t3 [reads two Register $t2 & $t3 and add then write back to $t1]

The processor's 32 general purpose registers are stored in a structure called a register file.

Register file: A state element consists of a set of register that can be read and written supplying a register

The input to the register file specifies the Register number to be read and an output from register file that will carry the value that has been read from the registers.

To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register.

So therefore we need a total of four inputs (three for Register numbers and one for data) two output (both for data)

The register inputs are 5 bit wide to speify on of 32 register ($2^5 = 32$). Data input & two data output ~~data~~ buses are each 32 bit wide.

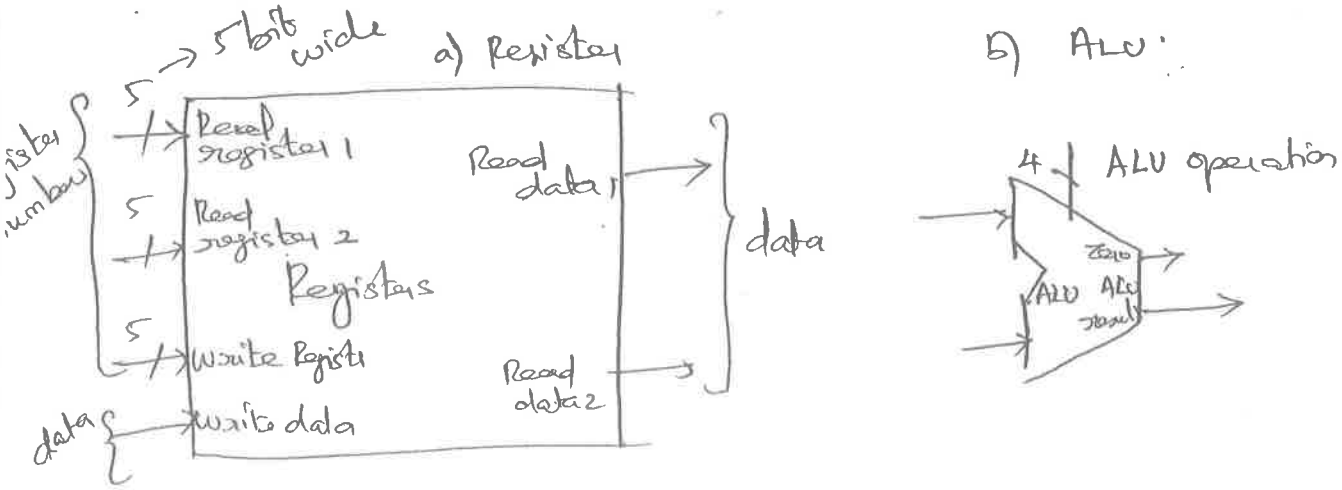


fig 8: Two element needed to implement R-format ALU operation

MIPS Load Word & Store Word Instruction:

The general form of load word & store

sw $t1, offset_value ($t2). These instruction compute memory address by adding base register $t2 to 16 bit signed offset field contained in the instruction.

eg: lw $t1, 20($t2)

→ base Register

→ offset_value.

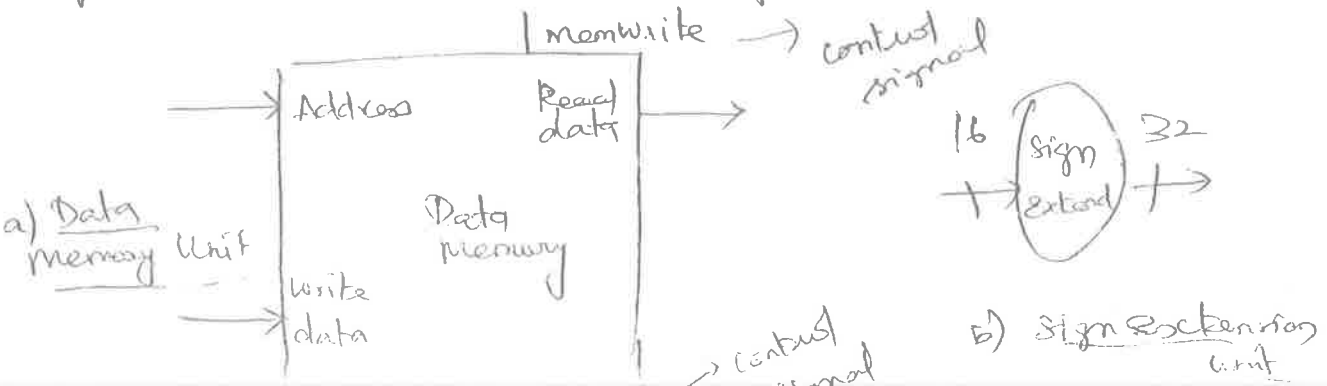
which the memory address can be calculated as $\boxed{\$t1 = \text{Memory}(\$t2 + 20)}$ and stored in $t1

In addition to Register file and ALU we need one additional unit called Sign Extend to extend the 16 bit offset_field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. Data Memory has read & write control signal, an address input, and an input for the data to be written into memory.

Sign Extend :-

To increase the size of a data item by replicating the higher-order sign bits of the original data item in the higher-order bits of larger, destination data item.

fig 4: two units needed to implement load & store.



Branch Instruction :-

The branch instruction has three operands, two registers that are compared for equality and a 16 bit offset field used to compute branch target address relative to the branch instruction address.

eg : beq $t1 , $t2 , 200
↳ 16 bit offset

To implement this instruction, we must compute the branch target address by adding the sign extended offset field of the instruction to the PC.

two things are need to be considered,

* The instruction set architecture specified that the base for the branch address calculation is the address of the instruction following the branch (PC+4). This value can also be used for calculation branch target Address.

* The offset field is shifted 2 bit so that it is a word offset (which is equivalent to multiplying by 4).

Branch target Address :-

The address specified in a branch, which becomes the new program counter (PC) if the branch is taken.

branch taken :-

A branch where the branch condition is satisfied and the program Counter (PC) becomes the branch target.

branch Not taken :-

A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

The branch datapath Must do two operations :-

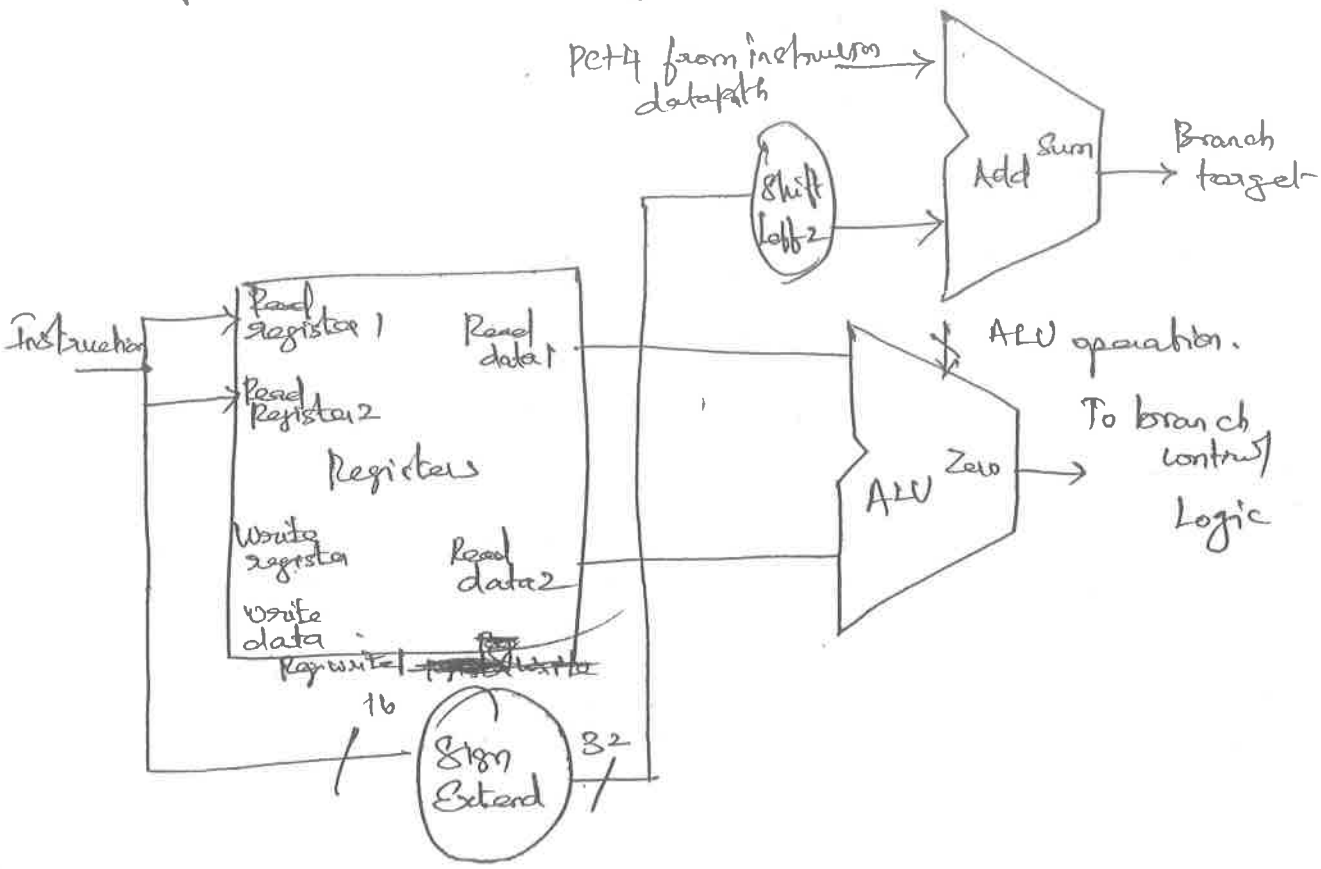
1. Compute the branch target Address
2. Compare the register Contents.

To compute branch target address the branch datapath includes a sign Extension Unit and an adder.

To compare two register we need to use two register file to supply two register operands. This Comparison is done using ALU.

We send to two register operands to the ALU with the control set to subtract. If Zero signal out, then ALU unit is asserted therefore two values are equal.

fig: 5 The datapath for a branch uses the ALU to evaluate the branch condition & seperate adder to compute branch target address.



Jump Instruction:-

The Jump instruction operates by replacing the lower 28 bits of the PC with lower 26 bits of the instruction shifted left by 2 bits..

eg: J 2000

↳ 26 bit offset

which points to 2000 × 4 ⇒ (8000 location)

requires shift left by 2 // equivalent to multiplying by 4 //

Delayed Branch:-

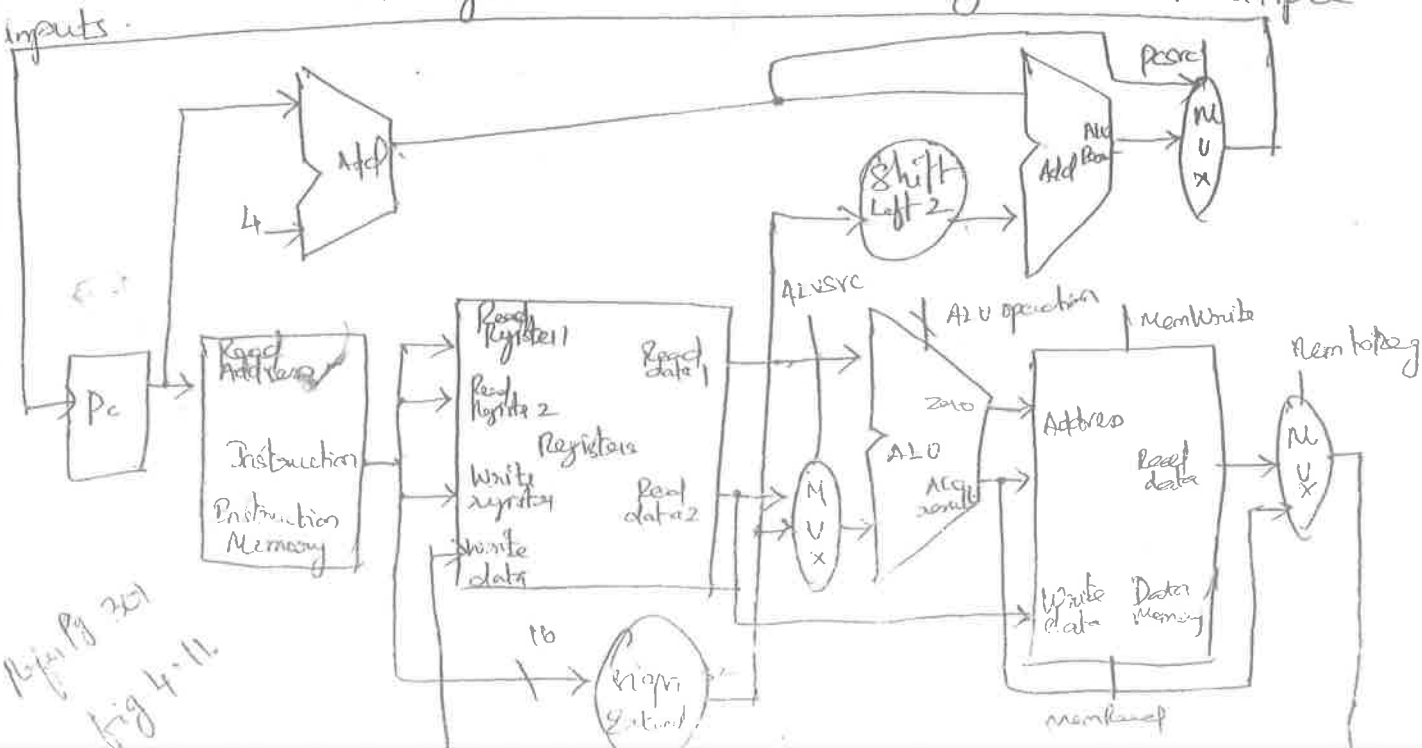
A type of Branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

Creating a single Datapath:-

Now we can combine the data path components needed for the individual instruction classes and add the control to complete the implementation.
This simplest data path will attempt to execute all instructions in one clock cycle.

No datapath resource can be used more than once per instruction.

To share datapath elements between two different instruction classes we may need to allow multiple connections to the input of an element using a multiplexer and control signal to select among the multiple inputs.



Hennessy Pg 301
Fig 4.11

Unit - III

Data Hazard.

Data Hazard:

It is also called as pipeline data Hazard. When a planned instruction cannot execute in the proper clock cycle because data needed is not yet available.

Program Execution Order

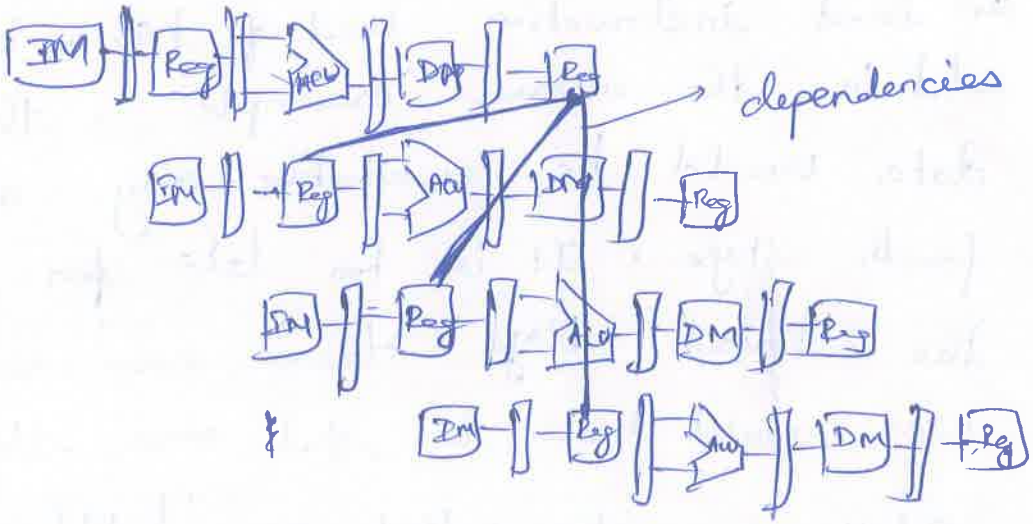
Time (clock cycle)

Sub $2 $1, $2

and $12, $2, $5

Or $13, $6, $2

add $14, $2, $2



Data Hazard can be overcome in two ways

1. Forwarding or bypassing
2. Stalling.

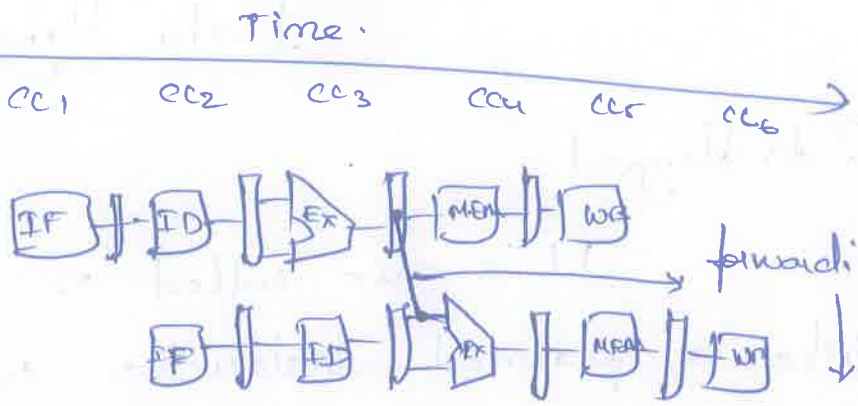
Forwarding or Bypassing:-

A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive.

Program execution order

add $s0, $t0, $t1

Sub $t2 $s0 $s3



(the data of $s0 is forwarded to Execution stage of 2nd instruction)

Stalling :- (Stall) :- [Waiting Until the Hazard is resolved is called stalling]

If the first instruction was an load instruction load of $s0, instead of add in the above example, the desired data would be available only after the fourth stage. It is too late for the input of the third stage. Hence even with forwarding we would have to stall one stage.

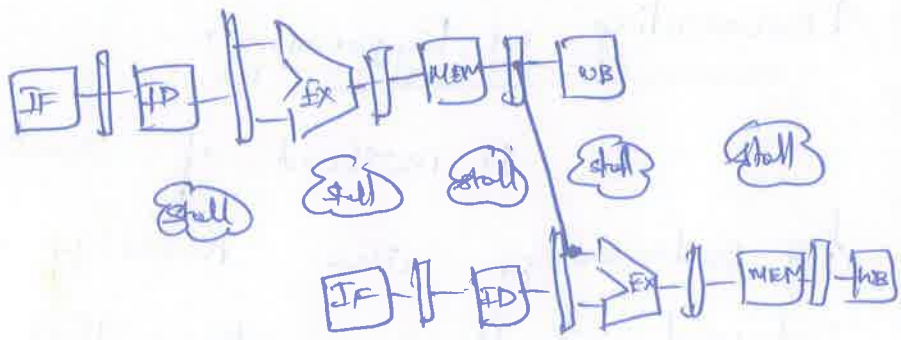
Stalls are also called as "bubble".

eg: stalling with forwarding for Load instruction.

Program execution order

lw $s0, 8($s1)

Sub $t2, $s0, $s3



The More precise notation for the two pairs of hazard conditions are.

1.a EX/MEM . Register Rd = ID/EX . Register Rs
1.b EX/MEM . Register Rd = ID/EX . Register Rt
2.a MEM/WB . Register Rd = ID/EX . Register Rs
2.b MEM/WB . Register Rd = ID/EX . Register Rt

eg:

```
add $s0, $t0, $t1
sub $t2 $s0 $s3
```

here the when the ~~and~~ sub instruction in EX stage, and the prior 'add' is in MEM stage so this is hazard 1a.

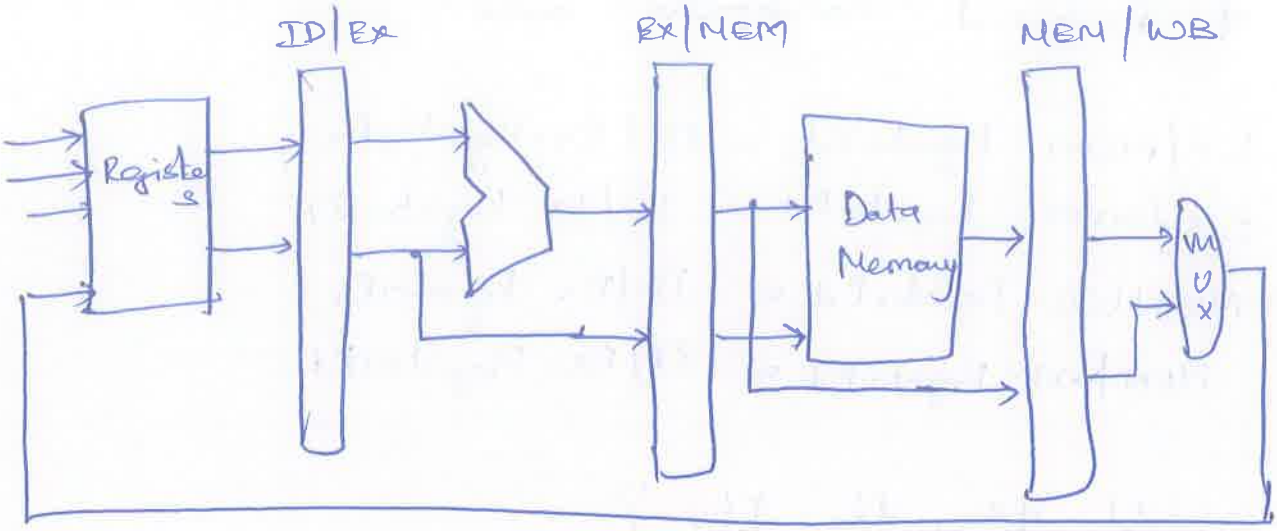
$$\boxed{\text{EX/MEM . Register Rd} = \text{ID/EX . Register Rs} = \$s_0}$$

add instruction — Sub instruction. — dependent Register

Data Hazard and Forwarding :-

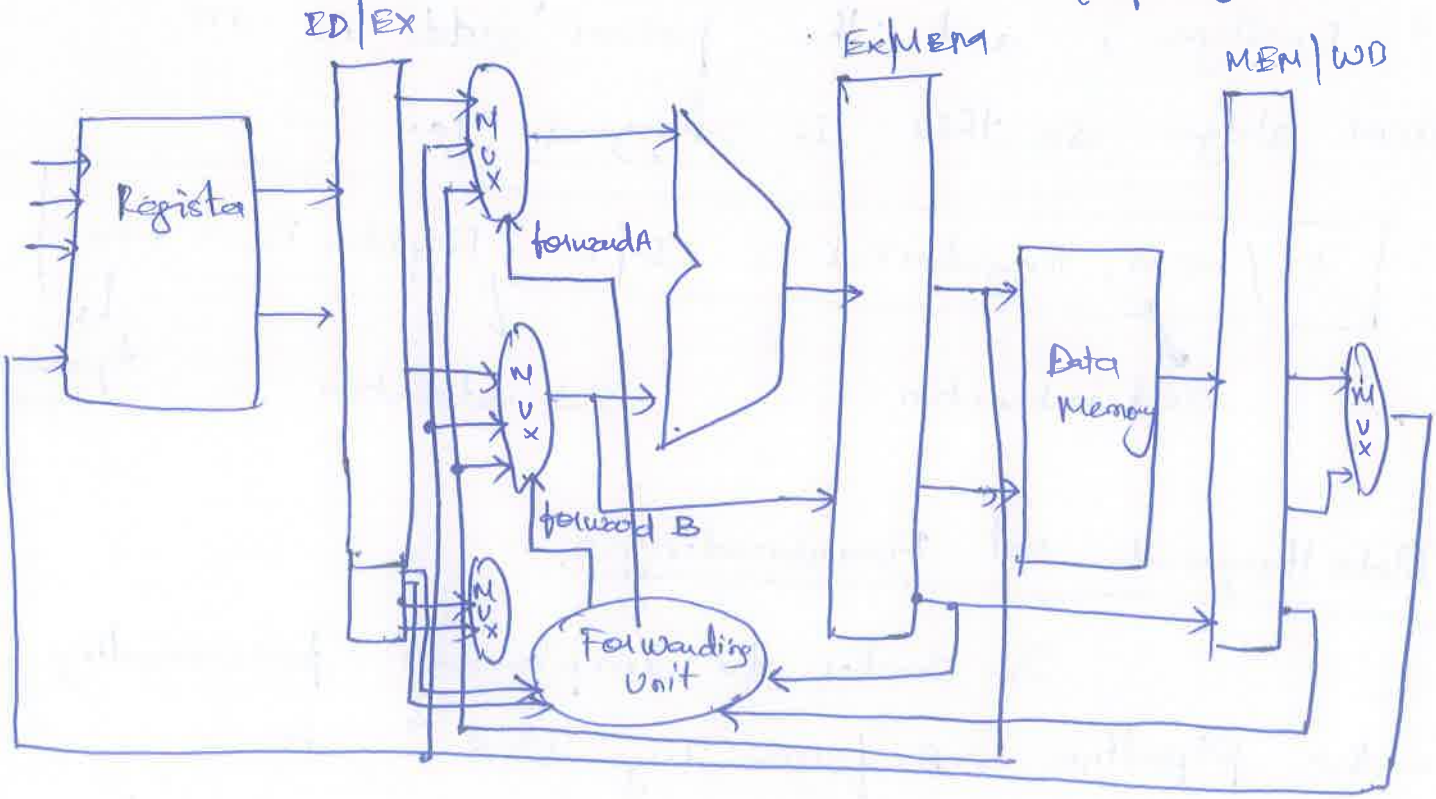
In order to implement forwarding in the pipeline A "forwarding Unit" is added, which inturn controls the two input to the ALU. One input from EX/MEM pipeline Register and another one from MEM/WB Register.

ALU & Pipeline Register Befor Forwarding



ALU & Pipeline Register With forwarding

(Refer Pg 354)



The control values for the forwarding multiplexer.

MUX control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from Register File
Forward A = 10	EX/MEM	The First ALU operand is forwarded from ~~prior~~ prior ALU result
forward A = 01	MEM/WB	The first ALU operand is forwarded from data Memory or an earlier ALU result
ForwardB = 00	ID/Ex	The Second ALU operand comes from the Register file
Forward B = 10	Ex/MEM	The second ALU operand is forwarded from the prior ALU result
Forward B = 01	MEM/WB	The second ALU operand is forwarded from data Memory or earlier ALU result

Consider the conditions for Detecting Hazard and the Control signal to resolve them

eg:-

if (EX/MEM . RegWrite)
and (EX/MEM . RegisterRd ≠ 0)
and (EX/MEM . RegisterRd = ID/EX . Register Rs))

hazard condition

ForwardA = 10

Control for MUX.

Which means the second instruction is in Execution stage, first instruction is in Mem stage then the data can be forwanded (ForwardA = 10) from prior ALU result.

The dependencies between the pipeline Registers moves forward in time.

time (in clock cycles) →

Program Execution order

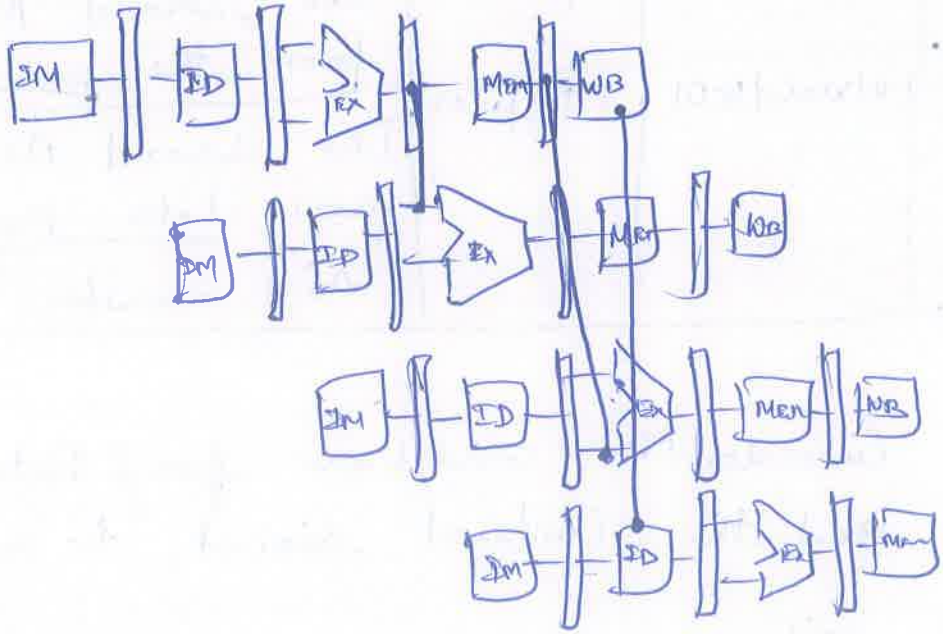
(Refer Pg 358)

Sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14 $2, $2



Data Hazards and Stalls:

In addition to Forwarding unit we need "hazard detection Unit". It operates during the ID stage so that it can insert stall between

the load and its use.

The condition for the Hazard for load instructions are

```
if ( ID/Ex . MemRead and
     (( ID/Ex. RegisterRt = IF/ID. RegisterRs) or
      (ID/Ex . RegisterRt = IF/ID. RegisterRt )))
     :
     Stall the pipeline.
```

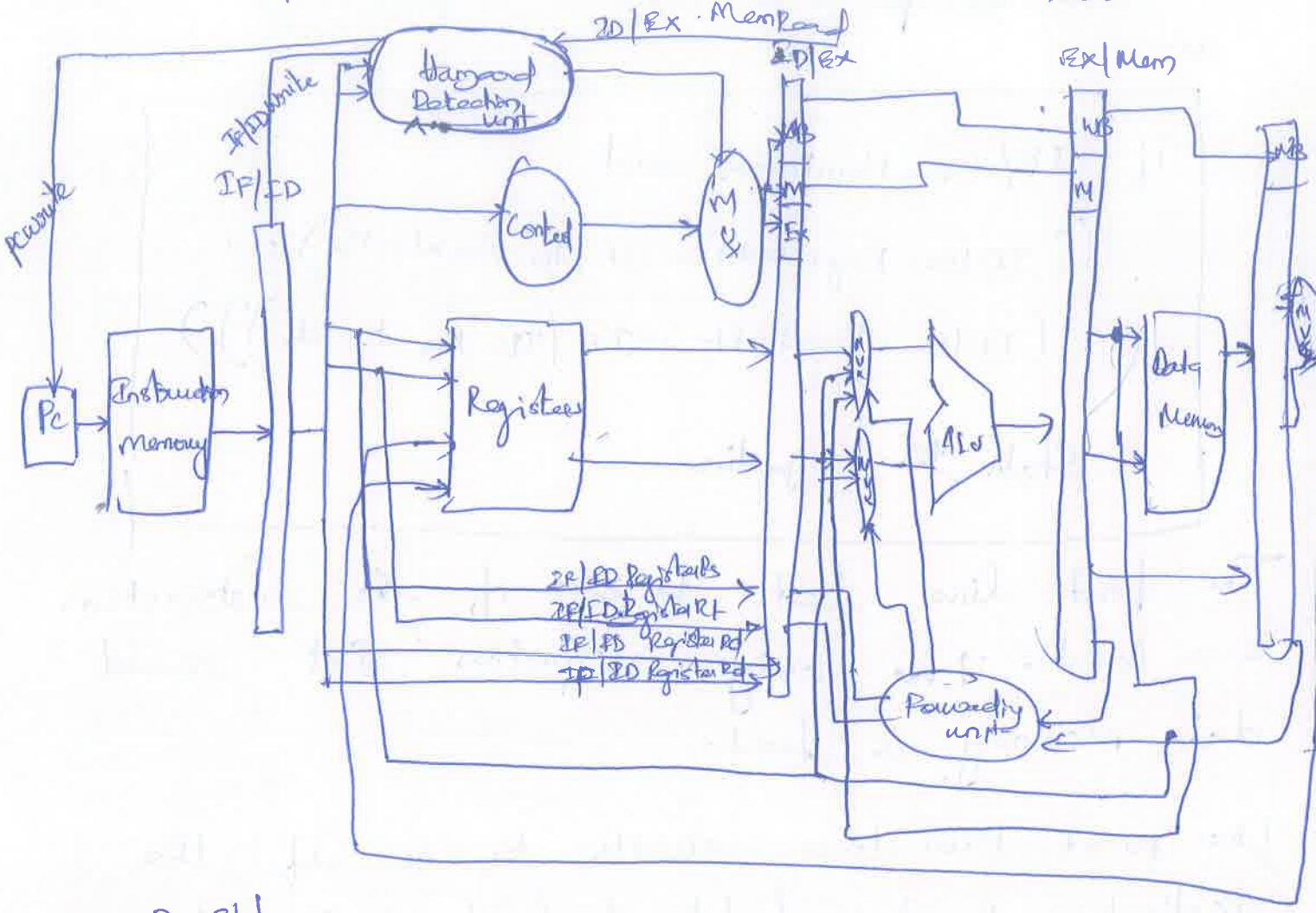
The first line test to see if the instruction is a load. The only instruction that read data memory is load.

The Next two lines check to see if the destination Register field of load in Ex stage Matches either source register of the instruction in the ID stage.

If the condition holds, the instruction stalls one clock cycle. Called as 'nop' - An instruction that does no operation to change state.

nops - which acts like bubble can be inserted into the pipeline by deasserting all nine Control signals (setting them 0) in Ex, MEM,

WB stages. No registers or Memories are written if the control values are all zero.



Refer Pg 361
fig 4.60