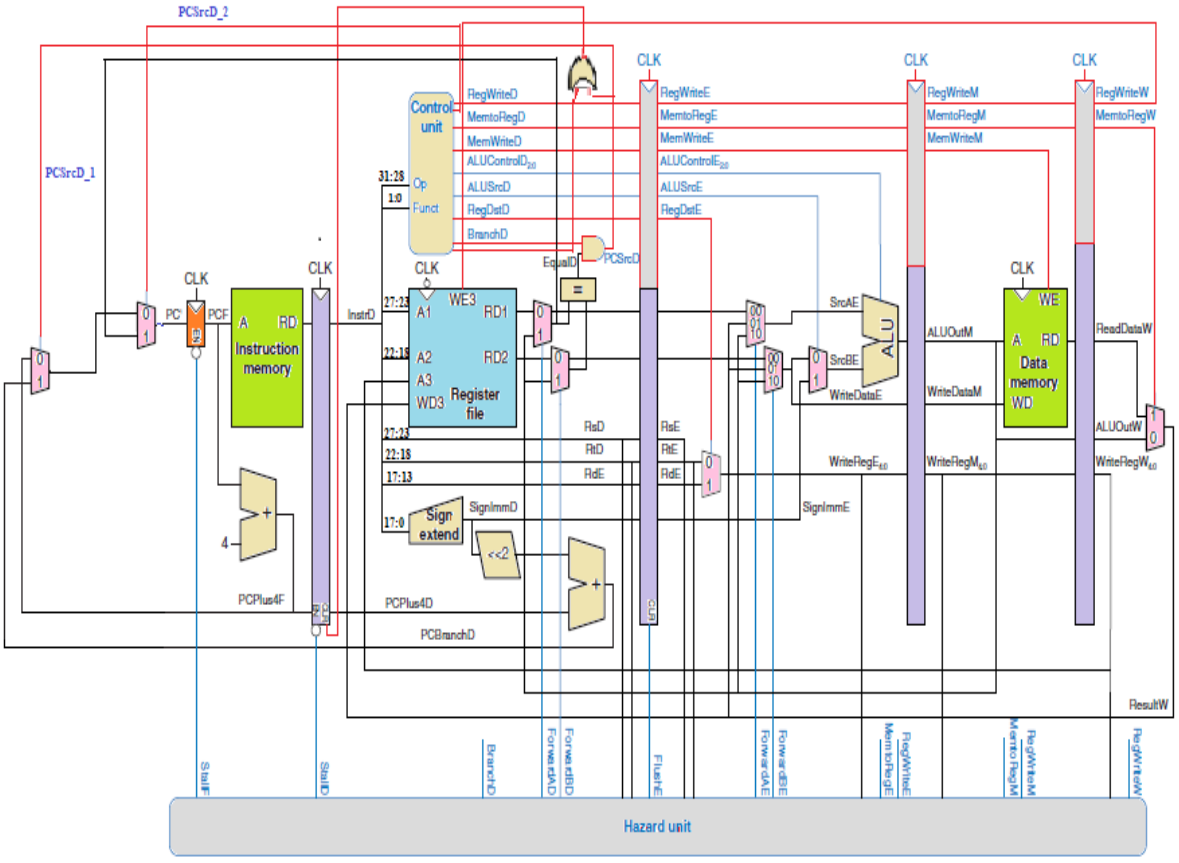
Concordia University

Engineering and Computer Science

Winter 2019

COEN 6741: Computer Architecture and Design
Dr. Abdelaziz Trabelsi

Project Report
Mini MIPS -32 Implementation



Team Members

Amulya Prabakhar (**40089026**)
Nishanth Tammalla (**40071525**)
Sibi Ravichandran (**40076654**)
Vivekchary Gandhari (**40079010**)

Form ENCS-001(09/03)

Report

Confirmation of Originality

Faculty of Engineering and Computer Science

Course Name & Number/Term: COEN 6741 (e.g., ENGR410/2) **Section:** ____ (e.g. M) **Instructor:** DR. ABDELAZIZ TRABELSI

Having researched and prepared this report for submission to the Faculty of Engineering & Computer Science, the undersigned certify that the following statements are to the best of my/our knowledge true:

1. The undersigned have written this report myself/themselves.
2. This report consists entirely of ideas, observations, references, information and conclusions composed or paraphrased by the undersigned, as the case may be, except for statements contained within quotation marks and attributed to the best of my/our knowledge to their proper source in footnotes or otherwise referenced.
3. With the exception of material in appendices, the undersigned have endeavored to ensure that direct quotations make up a very small proportion of the attached report, not exceeding 5% of the word count.
4. Each paragraph of this report that contains material which the undersigned have paraphrased from a source (print sources, multimedia sources, web-based sources, course notes or personal interviews, etc), has been identified by numerical reference citation.
5. All of the sources that the undersigned consulted and/or included in the report have been listed in the Reference section of the report.
6. All drawings, diagrams, photos, maps or other visual items derived from sources have been identified by numerical reference citations in the caption.
7. Each of the undersigned has revised, edited and proofread this report individually.
8. In preparing this report the undersigned have read and followed the guidelines found in Form and Style, by Patrick MacDonagh and Jack Borden (Fourth Edition: May 2000), available at http://www.encs.concordia.ca/scs/Forms/Form&Style.pdf.

Name: AMULYA PRABAKHAR (please print clearly) ID No: 40089026 Signature: Amulya.P Date: 29/03/2019

Name: NISHANTH TAMMALLA (please print clearly) ID No: 40071525 Signature: [signature] Date: 29/03/2019

Name: SIBI RAVICHANDRAN (please print clearly) ID No: 40076654 Signature: [signature] Date: 29/03/2019

Name: VIVEK CHARY GANDHARI (please print clearly) ID No: 40079010 Signature: [signature] Date: 29th Mar 201

Name: _____ (please print clearly) ID No: _____ Signature: _____ Date: _____

Name: _____ (please print clearly) ID No: _____ Signature: _____ Date: _____

Do Not Write in this Space – Reserved for Instructor

Table of Contents

1. INTRODUCTION..... 1
2. MINI MIPS ARCHITECTURE..... 1
2.1. Mini MIPS Instruction Set Architecture:..... 1
2.2. Pipelined Mini MIPS 2
2.3. Mini MIPS Micro architecture: 2
3. DATA PATH AND CONTROL UNIT IMPLEMENTATION 3
3.1. Data Path design:..... 3
3.1.1. Instruction Fetch:..... 3
3.1.2. Instruction Decode:..... 3
3.1.3. Execution stage: 4
3.1.4. Memory Access:..... 4
3.1.5. Write Back:..... 5
3.2. Control Unit Design:..... 5
4. HAZARD CONTROL..... 6
4.1. Hazard Free Design:..... 7
5. TEST BENCH AND PERFORMANCE CALCULATION. 9
5.1. Test Bench: 9
5.2. Performance Calculation: 10
6. SIMULATION RESULTS..... 10
7. FUTURE ENHANCEMENTS 11
8. CONCLUSION..... 11
RESPONSIBILITIES 11
REFERENCES..... 11

Table of figures

Figure Number	Figure Name	Page Number
Figure 1	Pipelined Execution of Instructions	2
Figure 2	Basic Micro architecture of a Mini MIPS-32	2
Figure 3	RTL diagram of the Instruction fetch stage	3
Figure 4	RTL diagram of Instruction Decode Stage	4
Figure 5	RTL Diagram of Execution Stage	4
Figure 6	RTL diagram of Memory access Stage	4
Figure 7	RTL Diagram of Write back stage	6
Figure 8	RTL diagram of Control unit	6
Figure 9	Illustration and solution RAW using Forwarding	7
Figure 10	Illustration and solution of RAW using stalling.	7
Figure 11	Illustration and resolution of Branch hazards.	7
Figure 12	Forwarding Algorithm	8
Figure 13	Stall and Flush Logic	8
Figure 14	Forwarding and Branch Stall Logic	8
Figure 15	Pipelined processor with full hazard handling	9
Figure 16	Simulation Result showing pipelined execution of BEQ instruction	10
Figure 17	Simulation Result showing Pipelined Execution of Jump Instruction	10
Figure 18	Simulation Results showing pipelined execution of Load Instructions	10

List of Tables

Table number	Table Name	Page number
Table 1	Instruction types of a Mini MIPS	1
Table 2	Encoding of the Instruction Set	5
Table 3	Signal Values generated by control unit for different instructions	6
Table 4	Control Signals and their functions	6
Table 5	Summary of hazards.	6
Table 6	Table illustrates the CPI calculation	10

1. INTRODUCTION

The worldwide development of high end, sophisticated digital design system created a huge demand for high speed and low power general purpose processor. Different processor architecture have developed and optimized to achieve better performance. The MIPS is known as one of the best RISC processors ever designed. MIPS allow the instruction to execute in fewer clock cycles. The aim of the project is to develop the hazard free Mini MIPS pipelined architecture, which is a 32-bit microprocessor, designed to support a limited subset of the MIPS instruction set. We will divide our micro architectures into two interacting parts: the data path and the control unit. Data path contains structures such as memories, registers, ALUs, and multiplexers, whereas the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the data path. The implementation is divided into five stages namely,

1. Instruction Fetch
2. Instruction Decode
3. Execution
4. Memory Access
5. Write-back

Mini-MIPS uses the same 3 instruction formats of MIPS (R, I and J-types) to implement the following 10 instructions: AND, SUB, XOR, ANDI, SUBI, ADD, BEQ, LW, SW, and JR. It will be assumed that the memory can be accessed in one clock cycle and works synchronously with the CPU. This project is designed in VHDL and we are using Model-Sim to simulate our code and also using Xilinx software to produce the RTL design blocks.

2. MINI MIPS ARCHITECTURE

2.1. Mini MIPS Instruction Set Architecture:

MIPS design was intended to simplify processor design by eliminating hardware interlocks between the pipeline stages. MIPS are a load/store architecture, which means that only load and store instructions access memory. In our design, the MIPS instructions are 32-bit long and can be broken into three classes: the memory-reference instructions, the arithmetic- logical instructions, and the branch instructions. Also, there are three different instructions formats in MIPS architecture: R-Type instructions, I-Type instructions, and J-Type instructions. Since, we have designed a mini MIPS which performs 10 instructions, we are selecting 4 bits for opcode. However, for our future consideration we can also use 3 bits for the opcode since there are 4 R-Type instructions which will use the same op-code. Also, our architecture contains 32 32-bit Registers (R0 to R31), which can be indicated using 5 bits. To avoid the garbage value at the ALU output, R0 will always hold the value "0" by default or while flushing and clearing the buffers. Refer Instruction types of a Mini MIPS in Table 1 given below:

INSTRUCTIONTYPE	BITS (31 to 0)					
R TYPE	Opcode (4)	Rs (5)	Rt (5)	Rd (5)	Shift Amount (11)	Function (2)
I TYPE	Opcode (4)	Rs (5)	Rt (5)	Immediate (18)		
J TYPE	Opcode (4)	Address (28)				

Table 1. Instruction types of a Mini MIPS

Description of the fields in the table is as follows:

1) ***Opcode (31-28)*:** The opcode is of 4 bits only which means that there only 16 possible instructions, where we are using only 10 of them. Opcode is same for all the R-type instructions. Opcode in I, J-type or Opcode along with the ***Function (1-0)*** field in R-type specifies the operation to be performed.
2) ***Rs (27-23*):** This 5-bit field in R, I,- type indicates the first source register. It contains the address of the register where the operand is stored.
3) ***Rt (22-18):*** This 5-bit field in R-type indicates the second source register. It contains the address of the register where the operand is stored. Whereas, in I-type it indicates address of the target where the result is to be stored.
4) ***Rd (17-13):*** This 5-bit filed in R–type indicates the destination register. It contains the address of the register where the result is to be stored.
5) ***Shift amount (12-2*):** This field in R-type defines the number of bits to be shifted. It is used for shift instructions only. For future use we can reduce the number of bits in this and can extend the function field for more operation.
6) ***Immediate (17-0):*** This field in I-type defines immediate value which is sign extended and is second operand for the arithmetic instructions.

7) ***Address (27-0):*** This 28-bit field in J-type indicates the offset that is to be added to the current PC to calculate the destination for the jump.

Description of types of instructions used is as follows:

1. **ALU Instructions:** The ALU unit performs operations either on two registers or on one operand from the register and other is the sign-extended immediate value. MIPS perform operations on both signed and unsigned vales. Basic ALU operations are: AND, OR, ADD, SUB, ANDI and SUBI.
2. **Load and Store Instructions:** The load instruction loads the data from the memory using the effective address. The store instruction uses the effective address to write the data onto the memory.
3. **Control Instructions:** There are two control transfer instructions: Branches and Jumps. Branches are the conditional transfer instructions. The conditions used in MIPS are the comparison between the registers (BEQ). Jumps are the unconditional transfers to the destination address specified in the instruction.

2.2. Pipelined Mini MIPS

The pipelined micro architecture applies pipelining to the single-cycle micro architecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires no architectural pipeline registers. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today. In the pipelined MIPS, a new instruction is fetched in every clock cycle. During each clock cycle some part of each instruction is executed. Pipeline registers are edge triggered, so at the rising edge of the clock the contents of pipeline registers are instantaneously changed. It carries both the control signal and data from one stage to another. The results of one stage are stored in these registers at the end of clock cycle and on the next rising edge values are passed to the successive stages.

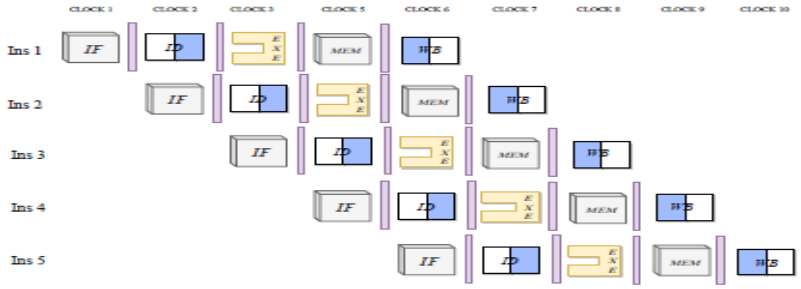


Figure 1. Pipelined Execution of Instructions

2.3. Mini MIPS Micro architecture:

We design a multi cycle processor as follows, first, we construct a data path by connecting the architectural state elements and memories with combinational logic, and we also add non architectural state elements to hold intermediate results between the steps. Then we design the controller. The controller produces different signals on different steps during execution of a single instruction, so it is now a finite state machine rather than combinational logic. We again examine how to add new instructions to the processor. Finally, we analyze the performance of the multi cycle processor against hazards. The basic micro architecture of a Mini MIPS without considering hazards is given in Figure 2

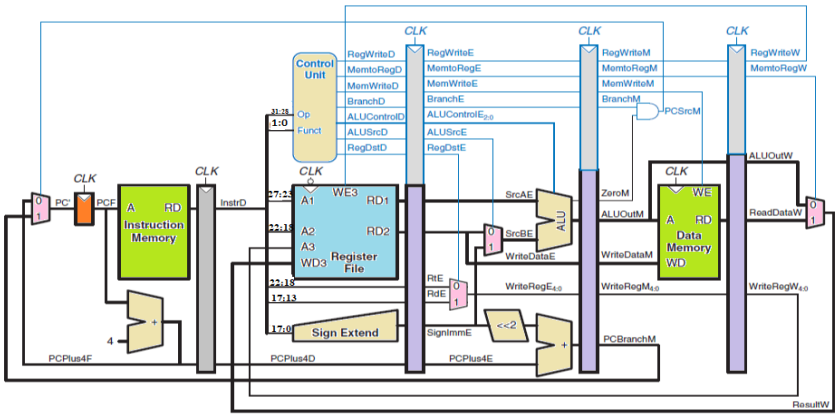


Figure 2. Basic Micro architecture of a Mini MIPS-32

3. DATA PATH AND CONTROL UNIT IMPLEMENTATION

3.1. Data Path design:

Data path designed to support data transfers required by instructions. It will have an instruction memory unit and a data memory unit. For 32-bit architecture, the data path will be 32 bits. Execution of instructions in data path is controlled by the control unit that monitors the flow of instructions to prevent any potential hazards. Data path contains 5 execution stages namely, Instruction Fetch, Instruction Decode, Execution, Memory Access and Write back.

3.1.1. Instruction Fetch:

The Instruction Fetch stage is where a program counter will pull the next instruction from the correct location in program memory. In addition the program counter was updated with either the next instruction location sequentially, or the instruction location as determined by a branch. The simulation tool (Modelsim) will throw error that the item to be accessed exceeds the defined range when we try to address the instruction and data memory using more than 10 bits.

Hardware	Description
Program Counter	The program counter (PC) register contains the address of the instruction to execute
Instruction Memory	It is a ROM with 1024 Rows * 8 Bits = 8192 Bits (i.e 8 kB). Can Store 256 Instructions. Each instruction is of 4 rows. For example, the first instruction starts at '0', so the second instruction starts from '4' and so on. Since there are only 1024 rows (0 to 1023) (Binary: 100 0000 0000), Maximum number of bits can be used to address the instruction memory should be 10 bits, if exceeds, then our simulator will throws an exception.
PC Mux	It is used to select the next PC from PC+4, Branch target and Jump target. The control unit sends a control signal to the multiplexer to select the next PC.
IF to ID Buffer	Buffer gets the data during falling edge of the clock

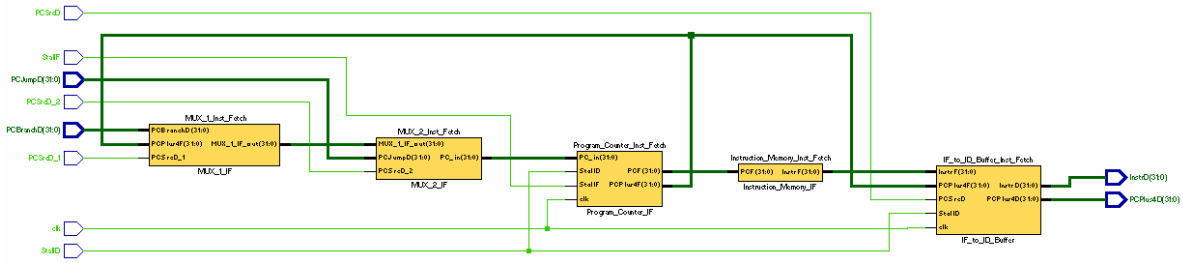


Figure 3. RTL diagram of the Instruction fetch stage

3.1.2. Instruction Decode:

The Instruction Decode stage is where the control unit determines what values the control lines must be set to depending on the instruction. In addition, hazard detection is implemented in this stage, and all necessary values are fetched from the register banks. This stage requires Register file, Sign Extension unit and control unit. RTL block of the Instruction Decode stage is given in Figure 4

Hardware	Description
Register file	• Contains 32,32-bit registers with register 0 being defined as always zero. • Control unit send a register write enable control signal to indicate whether the result is to be written onto the register file or not. Registers Rs and Rt are fed to the register file
SignExtension Unit	18 bits extended to 32 bits based on MSB bit, for positive value append 0's to the left and for negative value append 1's to the right
Control Unit	ALU opcode and function codes are sent to the control unit, which generates ALU control ID and few control signals to decide the operation to be executed in the ALU
ID to EX Buffer	Buffer gets the data during falling edge of the clock

.

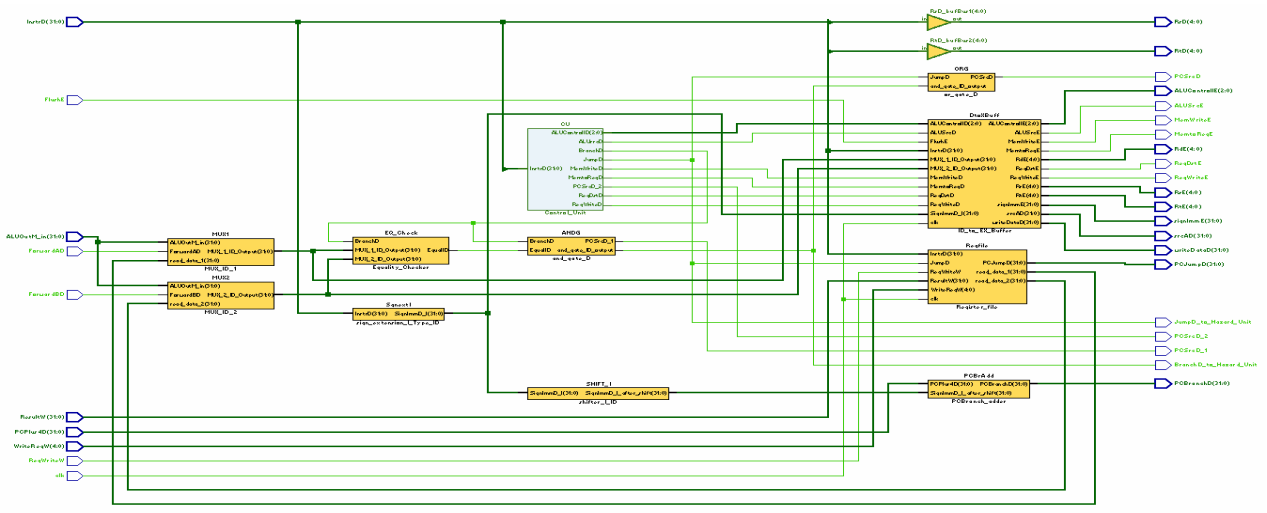


Figure 4. RTL diagram of Instruction Decode Stage

3.1.3. Execution stage:

The execution stage is responsible for taking the data and actually performing the specified operation on it. The execute stage consists of an ALU, multiplexers, Ex to Memory Buffer. RTL Block diagram of the execution stage is given in Figure 5

Hardware	Description
Arithmetic and Logic Unit	This unit performs the execution of Arithmetic and Logical Operations
Ex to memory Buffer	Buffer gets the data during falling edge of the clock

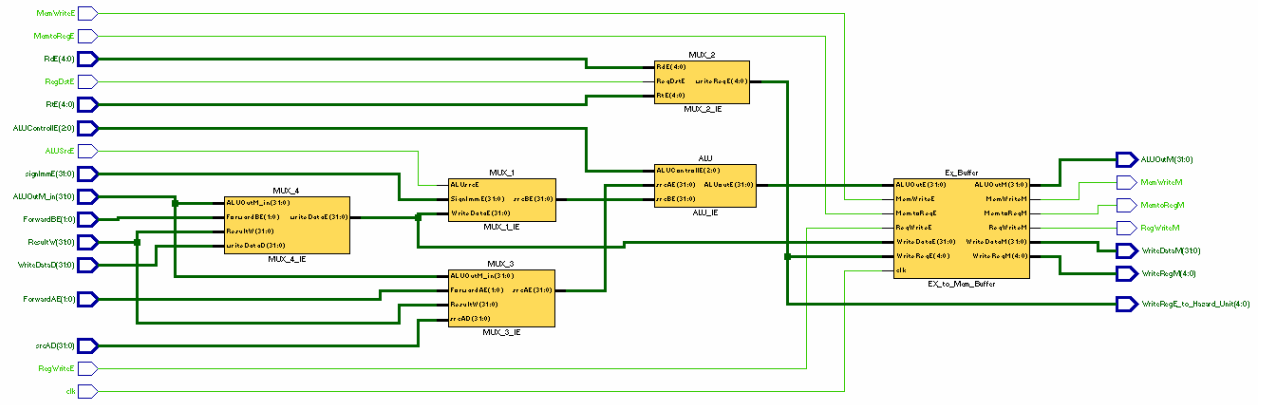


Figure 5. RTL Diagram of Execution Stage

3.1.4. Memory Access:

The load and store instructions execute at this stage; the other instructions will remain idle during this stage. Since these instructions have a unique step, we need this extra stage to account for them. So as a result, this stage is expected to be fast. This stage contains only Data Memory block and a memory to write back buffer. Refer to RTL block shown in Figure 6.

Data Memory: It is RAM with 1024 Rows * 32 Bits = 32768 Bits (i.e 32kB) = 1024 data can be stored. Since there are only 1024 rows (0 to 1023) (Binary: 100 0000 0000) in Data Memory- address can be indicated only by 10 Bits. if exceeds, the simulator will throw error.

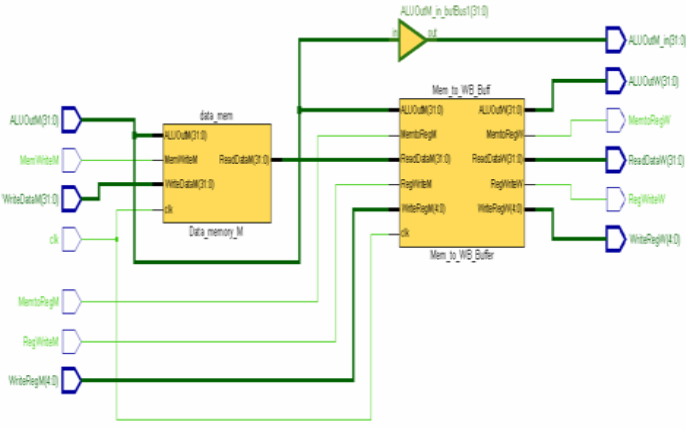


Figure 6. RTL diagram of Memory access Stage

3.1.5. Write Back:

The write back stage is responsible for writing the calculated value back to the proper register. It has input control lines that tell it whether this instruction writes back or not, and whether it writes back ALU output or Data memory output. It then chooses one of these outputs and feeds it to the register file based on these control lines. It uses 2:1 mux unit for the selection purpose. Refer to block diagram shown in Figure 7

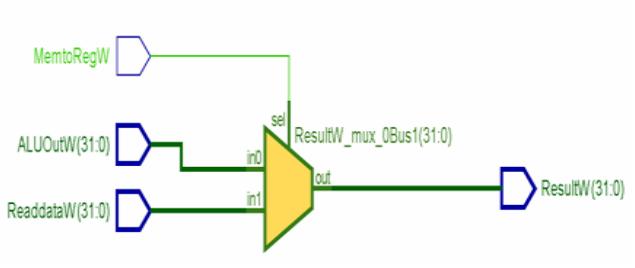


Figure 7. RTL Diagram of Write back stage

3.2. Control Unit Design:

The control unit (CU) is a component of a computer's CPU; it is contained within the processor that coordinates the sequence of data movements into, out of, and between a processor's many subunits. So every stage knows what to do and what to operate. The control unit is implemented in the Decode stage. The operation code and function field are extracted from the instruction set and are transferred to the control unit where various control signals are generated for the proper execution and completion of instruction. These signals include Set mux's to correct input; Operation code to ALU; Read and write to register file; Read and write to memory (load/store); branches; Branch target address computation etc. Control unit RTL block is shown as per the Figure 8

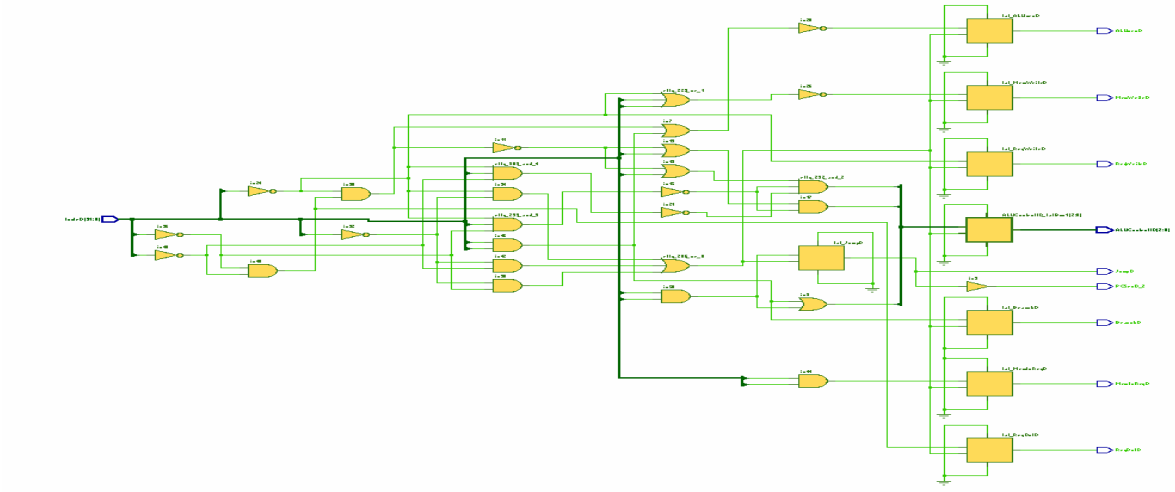


Figure 8. RTL diagram of Control unit

Encoding of the operations is given in the table follows; the table gives the values of opcode and function bits for each operation and the ALU control ID generated by the Control Unit based on the received opcode and function bits. For BEQ and JR operations the ALU control ID will be generated as "111" which means a no operation by ALU. Because we are performing these operations at decode stage to avoid the Branch or Control Hazard. If opcode exceeds the value in this table then the control unit will return NULL value.

S.No	FUNCTION	INSTRUCTION TYPE	OPCODE	FUNCTION	ALU CONTROL ID
1	AND	R-TYPE	0000	00	000
2	SUB	R-TYPE	0000	01	001
3	XOR	R-TYPE	0000	10	010
4	ANDI	I-TYPE	0001	N/A	000
5	SUBI	I-TYPE	0010	N/A	001
6	ADD	R-TYPE	0000	11	011
7	LW	I-TYPE	0011	N/A	011
8	SW	I-TYPE	0100	N/A	011
9	BEQ	I-TYPE	0101	N/A	111
10	JR	R-TYPE	0110	N/A	111

Table 2. Encoding of the Instruction Set

Signals generated from the Control Unit are binary values which hold either 0 or 1, the table below represents the value generated by Control Unit for different Instructions.

S.No	Signal Name	Value Generated By Control Unit Based on the Instruction Type									
		AND	SUB	XOR	ANDI	SUBI	ADD	LW	SW	BEQ	JR
1	RegWriteD	1	1	1	1	1	1	1	0	0	0
2	MemtoRegD	0	0	0	0	0	0	1	x	x	x
3	MemWriteD	0	0	0	0	0	0	0	1	0	0
4	BranchD	0	0	0	0	0	0	0	0	1	0
5	ALUsrcD	0	0	0	1	1	0	1	1	0	1
6	RegDstD	1	1	1	0	0	1	0	x	x	x
7	JumpD	0	0	0	0	0	0	0	0	0	1
8	PCSrcD_2	0	0	0	0	0	0	0	0	0	1
Note: x represents don't care											

Table 3. Signal Values generated by control unit for different instructions

Description of control signals from the control unit:

Signal Name	Function
RegWriteD	Indicates whether the operation will write the output into the register.
MemtoRegD	Indicates whether the operation will load the value from memory to register. (Load Word)
MemWriteD	Indicates whether the operation will store the value from the register to memory. (SW)
BranchD	Indicates whether the instruction is BEQ.
JumpD	Indicates whether the instruction is JR.
ALUSrcD	Indicates whether the second input to the ALU has to come from register or Immediate value.
RegDstD	Indicates which register (Rt or Rd) will act as the destination register.
PCSrcD_2	Indicates whether the instruction is JR, it acts as a selection line for mux_2 in IF.

Table 4. Control Signals and their functions

4. HAZARD CONTROL

In a pipelined system, multiple instructions are handled concurrently. When one instruction is dependent on the results of another that has not yet completed a hazard occurs. There are 3 types of hazards, Structural hazard, Data Hazard and Control hazards.

SL.NO.	TYPE	DESCRIPTION	SOLUTION
1	**STRUCTURAL HAZARD**	Same hardware used for two different things at the same time.	Using two different memories for Instruction and Data.
2	**DATA HAZARD**	Assume InstrI followed by InstrJ,	
	Read After Write	InstrJ tries to read operand before InstrI writes it	Forwarding, Stalling. (Ref Figure 9 and 10)
	Write After Read	InstrJ tries to write operand before InstrI writes it	Reads in stage 2 and writes in stage 5
	Write After Write	InstrJ tries to write operand before InstrI writes it	Reads in stage 2 and writes in stage 5
3	**CONTROL HAZARD**	when the decision of what instruction to fetch next has not been made by the time the fetch takes place	Branch mispredictionpenalty can be reduced if the branch decisioncould be made earlier.

Table 5. Summary of hazards.

Figure below illustrates hazards that occur when one instruction writes a register (r0) and subsequent instructions read this register. Some data hazards can be solved by forwarding (also called bypassing) a result from the Memory or Write back stage to a dependent instruction in the Execute stage. This requires adding

multiplexers in front of the ALU to select the operand from either the register file or the Memory or Write back stage.

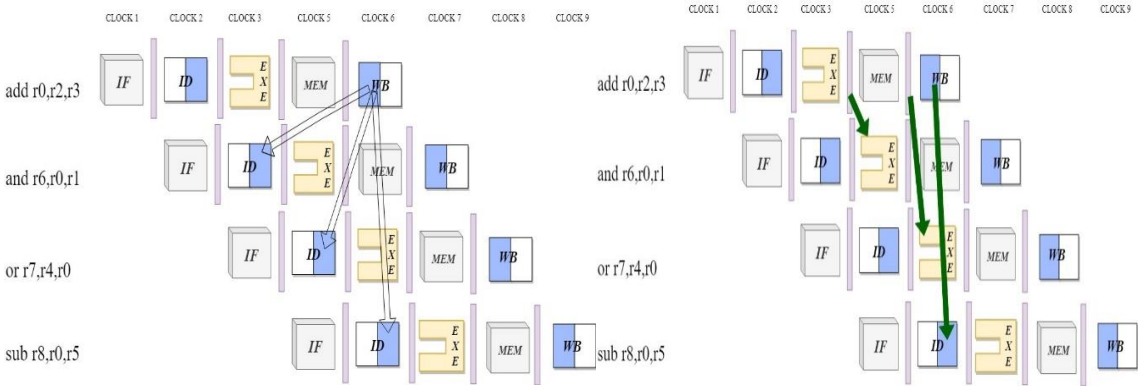


Figure 9. Illustration and solution RAW using Forwarding.

The lw instruction receives data from memory at the end of cycle 4. But the AND instruction needs that data as a source operand at the beginning of cycle 4. The alternative solution is to stall the pipeline, holding up operation until the data is available, Figure below illustrates this technique.

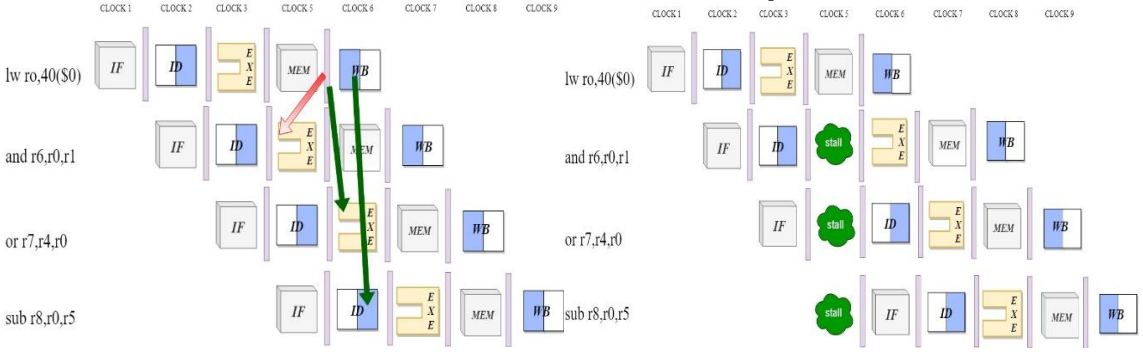


Figure 10. Illustration and solution of RAW using stalling.

Figure 11illustrates a branch hazard, in which a branch from address 0to address 16 is taken. The branch decision is not made until cycle 4. These instructions must be flushed, and the 'and' instruction is fetched from address 16 in cycle 5. These improvements can be done with the early branch decision being made in cycle 2. In cycle 3, the and instruction is flushed and the 'and' instruction is fetched.

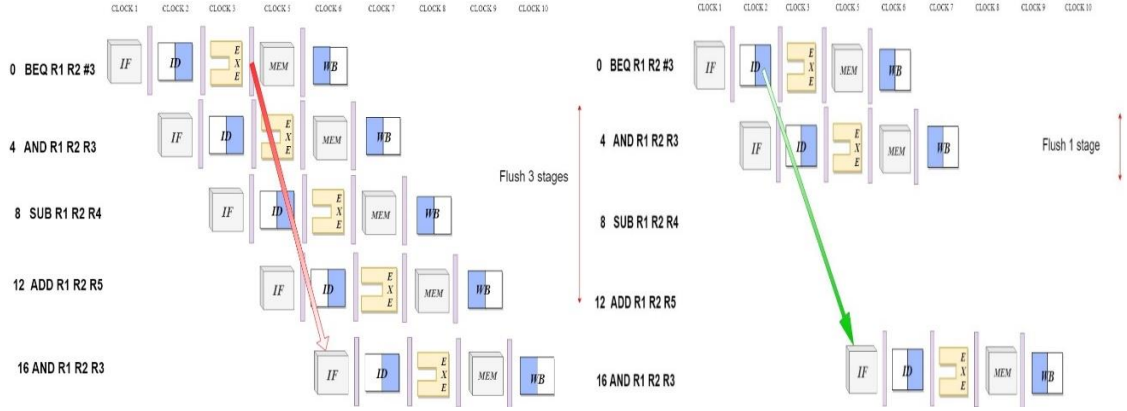


Figure 11. Illustration and resolution of Branch hazards.

4.1. Hazard Free Design:

Overcoming Structural hazards: Since we are using two different memories one for Instruction and the other for data. Our design is free from structural hazards.

Overcoming Data Hazard using Forwarding Technique: In order to get rid of Read After Write hazard, a hazard unit has been implemented with two forwarding multiplexers. The hazard unit receives the two source registers from the instruction in the Execute stage and the destination registers from the instructions in the Memory and Write back stages. It also receives the RegWrite signals from the Memory and Writeback stages to know whether the destination register will actually be written. The forwarding algorithm used to overcome data hazard is shown below

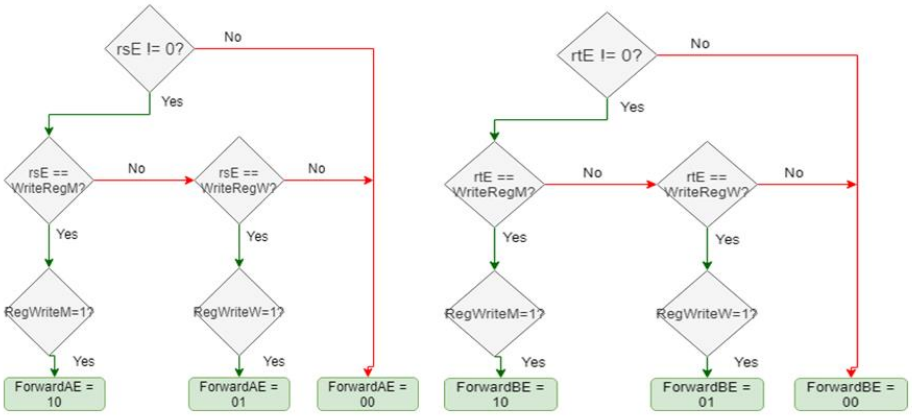


Figure 12. Forwarding Algorithm

Overcoming Data Hazard using Stalls: The hazard unit examines the instruction in the Execute stage. If it is lw and its destination register (rtE) matches either source operand of the instruction in the Decode stage (rsD or rtD), that instruction must be stalled in the Decode stage until the source operand is ready. Stalls are supported by adding enable inputs (EN) to the Fetch and Decode pipeline registers and a synchronous reset/clear (CLR) input to the Execute pipeline register. Figure shows the logic to stalls and flushes

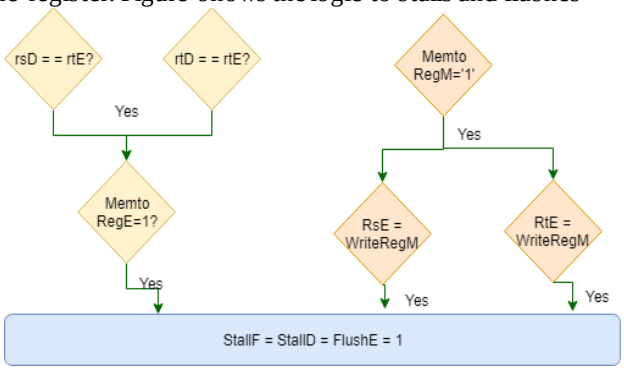


Figure 13. Stall and Flush Logic

Overcoming Control Hazard: To handle control hazard, branch decision is made earlier and hence moved PCSrc and gate to decode stage also added equality comparator. The PCBranch adder must also be moved into the Decode stage so that the destination address can be computed in time. The synchronous clear input (CLR) connected to PCSrcD is added to the Decode stage pipeline register so that the incorrectly fetched instruction can be flushed when a branch is taken. If the result of an ALU instruction is in the Execute stage or the result of a lw instruction is in the Memory stage, the pipeline must be stalled at the Decode stage until the result is ready. Decode Stage Forwarding Logic and Branch stall logic is given as,

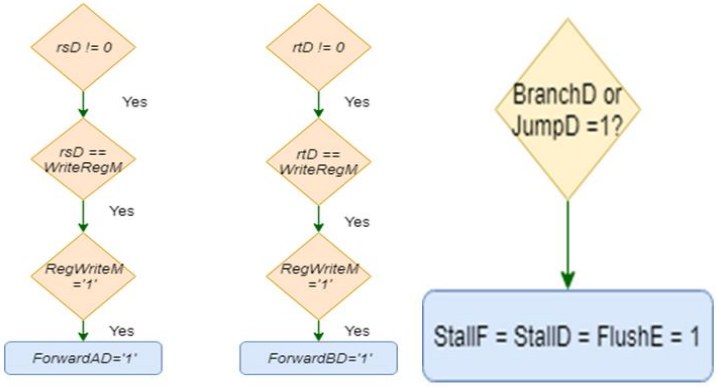


Figure 14. Forwarding and Branch Stall Logic

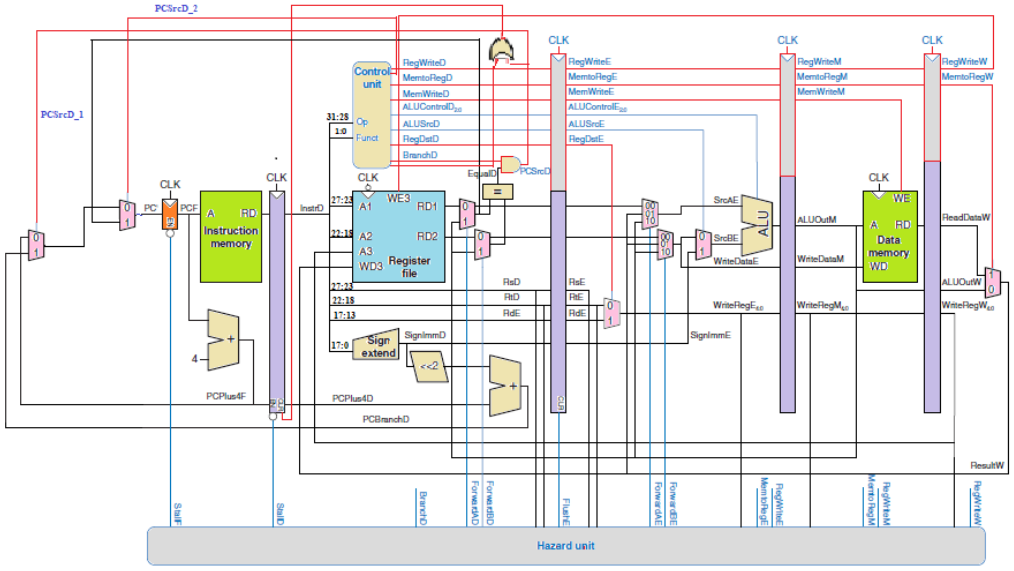


Figure 15. Pipelined processor with full hazard handling

5. TEST BENCH AND PERFORMANCE CALCULATION.

5.1. Test Bench:

To ensure the proper performance of the processor we have created a test bench. Default values loaded into Registers by default: R(0)= x"00000000" R(1)=x"00000000F" R(2)= x"0000000F", R(3)= x"00000034". Values Loaded into Data Memory by default: at location x"0000" = x"00000032". The above values are loaded by default just because to reduce the timing of simulation during demo.

Instruction Location	ASSEMBLY	INSTRUCTION IN HEX	HAZARD &SOLUTION			RESULT (values - Hex)
			TYPE	REGISTER	SOLUTION	
0	AND R1 R15 R3	0784 6000	No Hazard			R1=F R15=34 R3=4
4	BEQ R1 R2 #3	5088 0003	Branch	N/A	Branch Stall	Since R1=R2, PC moves to 20.
8	AND R1 R2 R3	0088 6000	**NOT EXECUTED DUE TO BRANCH TAKEN**			
12	SUB R1 R2 R4	0088 8001				
16	ADD R1 R2 R5	0088 A003				
20	AND R1 R2 R3	0088 6000	No Hazard			R1=F R2=F R3=F
24	SWR0 R3 #1	400C 0001	Data	R3	Forwarding	The value 'F' is stored in Data Memory at location x"1".
28	ADD R3 R2 R4	0188 8003	Data	R3	Forwarding	R3=F R2=F R4=1E
32	SUB R4 R1 R5	0204 A001	Data	R4	Forwarding	R4=1E R1=F R5=F
36	XOR R5 R4 R6	0290 C002	Data	R5, R4	Forwarding	R5=F R4=1E R6=F
40	JR R15 0 0	6780 0000	Jump	N/A	Jump Stall	Jump to the address stored in R15: 52
44	ANDI R1 R7 #25	109C 0019	**NOT EXECUTED DUE TO JUMP**			
48	AND R1 R2 R3	0088 6000				
52	LW R0 R7 #0	301C 0000	No Hazard			R7 is loaded with the value x"32"
56	ANDI R7 R1 #23	1384 0017	Load	R7	Load Stall	R7=32 Imm=17 R1=2B
60	SUBI R7 R3 #7	238C 0007	Load	R7	Load Stall	R7=32 Imm=7 R3=2B

5.2. Performance Calculation:

The following table shows the CPI Calculation for our MIPS Processor.

Total Number of instructions	**16**
# of stalls with Hazard techniques	**4**
# of clock cycles taken to complete execution	**21**
CPI= (CPU Clock cycles for the program)/ (Instruction count + Stalls)	**1.05**

Table 6. Table illustrates the CPI calculation

6. SIMULATION RESULTS

In this section, we have attached the experimental results obtained in the lab after simulating the project in ModelSim.

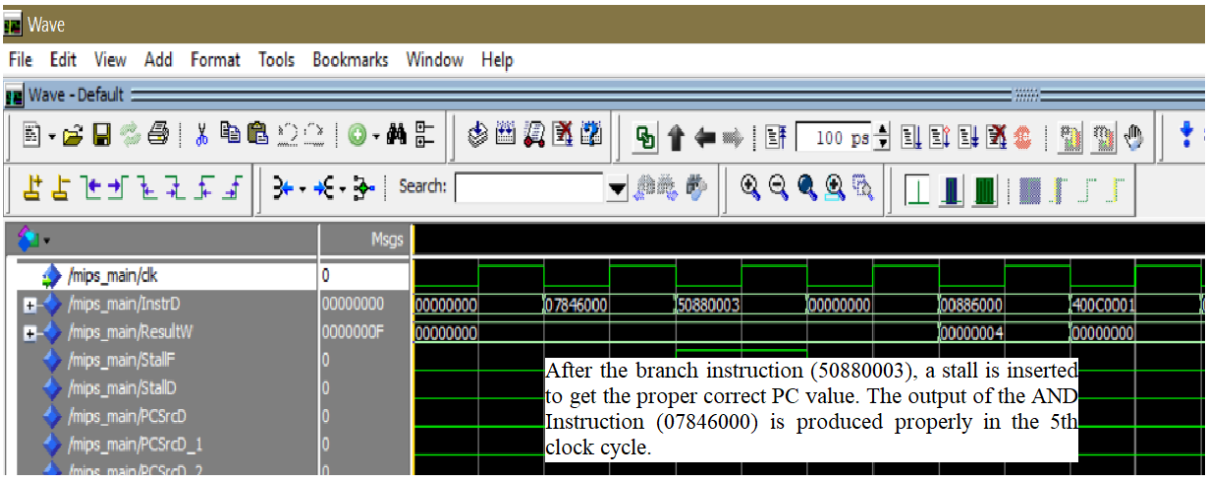


Figure 16. **Simulation Result showing pipelined execution of BEQ instruction**

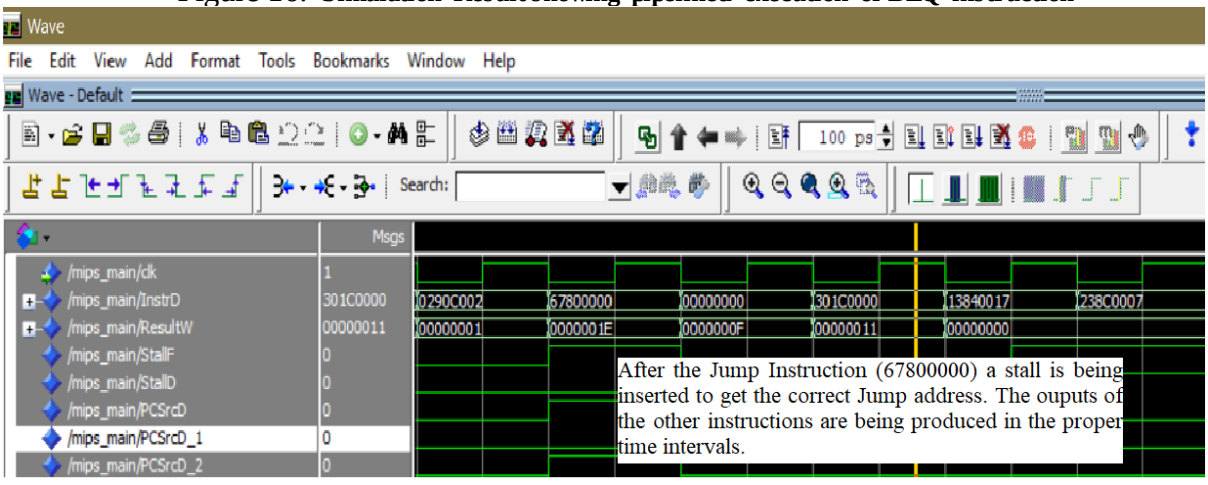


Figure 17. **Simulation Result showing Pipelined Execution of Jump Instruction**

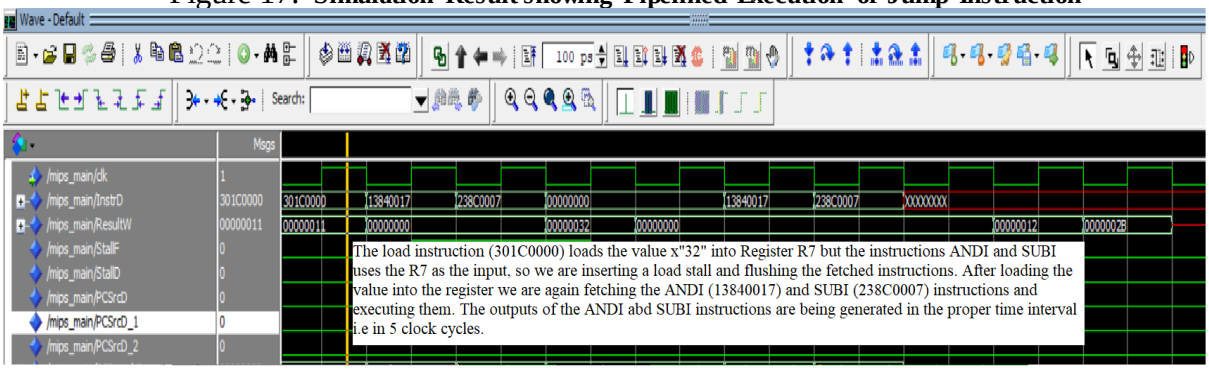


Figure 18. **Simulation Results showing pipelined execution of Load Instructions**

Phases and Challenges Faced during execution of Project:

Phase 1: We designed a Mini MIPS-32 without hazard unit in this phase.We faced issues for jump and branch instructions. To overcome this issue, we came up with 2 different PCSrc signals for Jump and BEQ instructions instead of one.
Phase 2: In this phase we developed the hazard unit; we faced issues to resolve hazards when they occur. We did some structural changes to our architecture and we were able to overcome these hazards.

7. FUTURE ENHANCEMENTS

1. We can reduce 11 bits for the shift in the R type instruction format and assign the 6 bits to the opcode thus increasing the more number of instructions to perform by MIPS.
2. We can add cache memory to the processor to improve the memory performance.
3. The Instruction memory which we use is of type ROM and it has some predefined values for the instruction opcode. we can change the ROM type of memory to more user friendly to define own values

8. CONCLUSION

The objective of this project was to design, simulate, and implement a simple 32-bit microprocessor with an instruction set that is similar to a MIPS, here we succeeded in building a 32-bit Mini MIPS which can execute 10 instructions without any hazards, The five stage 32-bit MIPS architecture has been implemented using MODELSIM by writing a VHDL code. The data hazards have been dealt by forwarding and stalling whenever necessary. The structural hazards have been taken care of by dividing the clock for read and write operations and using separate memory. To reduce the stalls that occur due to control hazards, the branch detection unit has been moved to the Instruction Decode (ID) stage which decreased the branch penalty to 1 clock cycle. We achieved executing 16 instructions (with 4 stalls) with a CPI of 1.05 that is 21 clock cycles. We also synthesized our design to get RTL blocks using Precision RTL tool.

RESPONSIBILITIES

Team Member	Roles and Responsibilities				
	Team Leader	Code Development and Testing	Report	Poster	RTL Implementaion
Amulya		✓	✓		✓
Nishanth		✓		✓	✓
Sibi	✓	✓	✓		✓
Vivek		✓		✓	✓

S.No	File Name	Author
1	Instruction Fetch Stage	NishanthTamalla
2	Instruction Decode Stage & Hazard Unit	SibiRavichandran
3	Instruction Execution Stage	Amulya
4	Data Memory And Write Back Stage:	Vivekchary

REFERENCES

1. Harris, D., & Harris, S. *Digital design and computer architecture*.
2. Hennessy, J., & Patterson, D. Computer architecture (5th ed.).
3. Patterson, D., Hennessy, J., & Alexander, P. Computer organization and design (5th ed.).
4. Concordia Course Web Sites: Log in to the site. (2019). Retrieved from https://moodle.concordia.ca/moodle/pluginfile.php/3463809/mod_resource/content/36/3Appendix%20C.pdf
5. https://www.researchgate.net/profile/Kirat_Singh3/publication/286836815_Vhdl_Implementation_of_A_Mips-32_Pipeline_Processor/links/566e58ba08ae1a797e40623b/Vhdl-Implementation-of-A-Mips-32-Pipeline-Processor.pdf
6. Diary RawoofSulaiman, "Using Clock gating Technique for Energy Reduction in Portable Computers" Proceedings of the International Conference on Computer and Communication Engineering pp.839 – 842, May 2008
7. http://www.cim.mcgill.ca/~langer/273/13-notes.pdf