



Department of Electrical and Computer Engineering,  
Gina Cody School of Engineering and Computer Science,  
Concordia University, Montreal, Canada

**COEN 6741 - Computer Architecture and Design**

## **Mini-MIPS**

**Group (12):**

**ABISHEK ARUMUGAM THIRUSELVI- 40218896**

**JAVIER SANDOVAL GONZALES- 40114974**

**MANISH PEJATHAYA- 40194909**

**SRINIDHI HONAKERE SRINIVAS- 40221128**

**Instructor: Prof. Sofiène Tahar**

**APRIL 13, 2022**

# Faculty of Engineering and Computer Science

## Expectations of Originality

This form has been created to ensure that all students in the Faculty of Engineering and Computer Science comply with principles of academic integrity prior to submitting coursework to their instructors for evaluation: namely reports, assignments, lab reports and/or software. All students should become familiar with the University's Code of Conduct (Academic) located at [http://web2.concordia.ca/Legal\\_Counsel/policies/english/AC/Code.html](http://web2.concordia.ca/Legal_Counsel/policies/english/AC/Code.html)




**Please read the back of this document carefully before completing the section below. This form must be attached to the front of all coursework submitted to instructors in the Faculty of Engineering and Computer Science.**

**Course Number:** COEN 6741 **Instructor:** Prof. Sofiène Tahar

**Type of Submission (Please check off responses to both a & b)**

- a. ☒ Report    ☐ Assignment    ☐ Lab Report    ☐ Software    ☐ Media
- b. ☐ Individual submission    ☒ Group Submission (All members of the team must sign below)

Having read both sides of this form, I certify that I/we have conformed to the Faculty's expectations of originality and standards of academic integrity.

Name: <u>ABISHEK ARUMUGAM THIRUSELVI</u> (please print clearly)	ID No: <u>40218896</u>	Signature: <u></u>	Date: <u>13 APRIL 2022</u>
Name: <u>JAVIER SANDOVAL GONZALES</u> (please print clearly)	ID No: <u>40114974</u>	Signature: <u></u>	Date: <u>13 APRIL 2022</u>
Name: <u>MANISH PEJATHAYA</u> (please print clearly)	ID No: <u>40194909</u>	Signature: <u></u>	Date: <u>13 APRIL 2022</u>
Name: <u>SRINIDHI HONAKERE SRINIVAS</u> (please print clearly)	ID No: <u>40221128</u>	Signature: <u></u>	Date: <u>13 APRIL 2022</u>
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**

## **EXPECTATIONS OF ORIGINALITY & STANDARDS OF ACADEMIC INTEGRITY**

### **ALL SUBMISSIONS must meet the following requirements:**

1. The decision on whether a submission is a group or individual submission is determined by the instructor. Individual submissions are done alone and should not be identical to the submission made by any other student. In the case of group submissions, all individuals in the group must be listed on and must sign this form prior to its submission to the instructor.
2. All individual and group submissions constitute original work by the individual(s) signing this form.
3. Direct quotations make up a very small proportion of the text, i.e., not exceeding 5% of the word count.
4. Material paraphrased from a source (e.g., print sources, multimedia sources, web-based sources, course notes or personal interviews) has been identified by a numerical reference citation.
5. All of the sources consulted and/or included in the report have been listed in the Reference section of the document.
6. All drawings, diagrams, photos, maps or other visual items derived from other sources have been identified by numerical reference citations in the caption.
7. No part of the document has been submitted for any other course.
8. Any exception to these requirements are indicated on an attached page for the instructor's review.

### **REPORTS and ASSIGNMENTS must also meet the following additional requirements:**

1. A report or assignment consists entirely of ideas, observations, information and conclusions composed by the student(s), except for statements contained within quotation marks and attributed to the best of the student's/students' knowledge to their proper source in footnotes or references.
2. An assignment may not use solutions to assignments of other past or present students/instructors of this course or of any other course.
3. The document has not been revised or edited by another student who is not an author.
4. For reports, 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>) have been used for this submission.

### **LAB REPORTS must also meet the following requirements:**

1. The data in a lab report represents the results of the experimental work by the student(s), derived only from the experiment itself. There are no additions or modifications derived from any outside source.
2. In preparing and completing the attached lab report, the labs of other past or present students of this course or any other course have not been consulted, used, copied, paraphrased or relied upon in any manner whatsoever.

### **SOFTWARE must also meet the following requirements:**

1. The software represents independent work of the student(s).
2. No other past or present student work (in this course or any other course) has been used in writing this software, except as explicitly documented.
3. The software consists entirely of code written by the undersigned, except for the use of functions and libraries in the public domain, all of which have been documented on an attached page.
4. No part of the software has been used in previous submissions except as identified in the documentation.
5. The documentation of the software includes a reference to any component that the student(s) did not write.
6. All of the sources consulted while writing this code are listed in the documentation.

<b><i>Important: Should you require clarification on any of the above items please contact your instructor.</i></b>
---

## **Table of Contents**

1. INTRODUCTION	1
2. MINI-MIPS	1
2.1 ISA	1
2.2 R-TYPE INSTRUCTION	1
2.3 I-TYPE INSTRUCTION	2
2.4 J-TYPE INSTRUCTION	2
2.5 ENCODING OF MINI-MIPS INSTRUCTION	2
3. MINI-MIPS ARCHITECTURE	3
3.1 INSTRUCTION FETCH STAGE	3
3.2 INSTRUCTION DECODE STAGE	4
3.3 EXECUTION STAGE	5
3.4 ALU CONTROL UNIT	6
3.5 DATA MEMORY STAGE	7
3.6 WRITE BACK STAGE	7
4. HAZARD CONTROL DESIGN	8
4.1 DATA HAZARD	9
4.2 STRUCTURAL HAZARD	10
4.3 CONTROL HAZARD	11
5. TEST BENCH AND SIMULATION RESULT	12
6. CONCLUSION	13
7. RESPONSIBILITIES	13
8. REFERENCES	14

## List of Figures

Figure Number	Figure Name	Page Number
Figure 2.1	MIPS 32-bit instruction format	1
Figure 2.2	Figure 2.2 R-type instruction format	1
Figure 2.3	Data path for R-type instruction	1
Figure 2.4	I-type instruction format	2
Figure 2.5	Data path for I-type instruction	2
Figure 2.6	J-type instruction format	2
Figure 3.1	MINI-MIPS architecture	3
Figure 3.2	Block diagram of instruction fetch stage	4
Figure 3.3	RTL diagram of instruction fetch stage	4
Figure 3.4	Block diagram of instruction decode stage	5
Figure 3.5	RTL diagram of instruction decode stage	5
Figure 3.6	Block diagram of instruction execution stage	5
Figure 3.7	RTL diagram of instruction execution stage	6
Figure 3.8	Block diagram for EX, DM, WB stage	7
Figure 3.9	RTL diagram for write-back operation	7
Figure 3.10	RTL diagram for DM operation.	7
Figure 4.1	Read After Write hazard	9
Figure 4.2	Write After Read hazard	9
Figure 4.3	Figure 4.3 Write After Write Hazard	9
Figure 4.4	Solving RAW by reordering instructions.	10
Figure 4.5	Data forwarding to solve RAW hazard	10
Figure 4.6	Pipelined system with structural hazard	10
Figure 4.7	Stalls to avoid structural hazards	10
Figure 4.8	Stalls to prevent Control hazard	11
Figure 5.1	IF stage simulation	12
Figure 5.2	Program counter (part of IF stage) simulation	12
Figure 5.3	Simulation for IF_ID buffer	12
Figure 5.4	Simulation for ID stage (Register file)	12
Figure 5.5	Simulation for ALU block (EX stage)	13
Figure 5.6	Simulation for MEM Stage	13

## List of Tables

<b>Table number</b>	<b>Table Name</b>	<b>Page number</b>
Table 1	Instruction set types and syntax of implemented instructions	2
Table 2	Encoding of the Instruction Set for ALU control ID	6
Table 3	Encoding of the Instruction Set	8
Table 4	Responsibilities	13

## 1. INTRODUCTION

Reduced instruction set computing processors consume way less power when compared to the Complex instruction set computing processors as the instructions in RISC are way simple, and the hardware is optimized correctly. The advantage of using a MIPS processor (RISC-based) is that millions of instructions are executed simultaneously without any interlocking.

We have tried to implement a mini-MIPS design, a 32-bit architecture (similar to the RISC-V architecture). The mini-MIPS design contains R, I, and J type instruction, which is the same as that for MIPS, in order to implement the following 11 instructions: ORI, SLL, XNOR, NAND, SUBI, ADDIU, BNE, LH, SB, JR and J. The following project has been designed using Verilog being simulated on Model-Sim software, and we are using the Precision RTL tool to obtain the RTL design blocks.

## 2. MINI-MIPS

### 2.1 ISA

The Mini MIPS RISC processor is designed using MIPS 32-bit instruction Register type instruction, immediate type instruction, jump type instructions MIPS 32-bit Register type instruction format is as shown below in Figure 2.1.

Field size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R-Format	Opcode	Rs	Rt	Rd	Shift	Function
I-Format	Opcode	Rs	Rt	Address/immediate value		
J-Format	Opcode	Branch target address				

Figure 2.1 MIPS 32-bit instruction format

### 2.2 R-TYPE INSTRUCTION

In Register type instruction format, the first six bits represent the function performed on the operands.

The following five represent the value to be shifted left or right. The next fifteen bits represent the first source register, second source register, and destination register where the output will be placed. The last 6 bits represent the opcode.

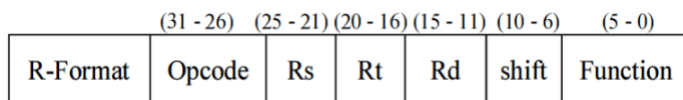


Figure 2.2 R-type instruction format

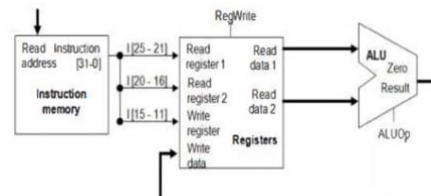


Figure 2.3 Data path for R-type instruction

## 2.3 I-TYPE INSTRUCTION

In the immediate type instruction format, the first sixteen bits represent the Immediate value that is not from memory and is assigned locally. The next ten bits represent the source and a destination register. The last six bits represent the opcode.

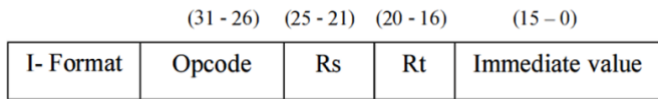


Figure 2.4 I-type instruction format

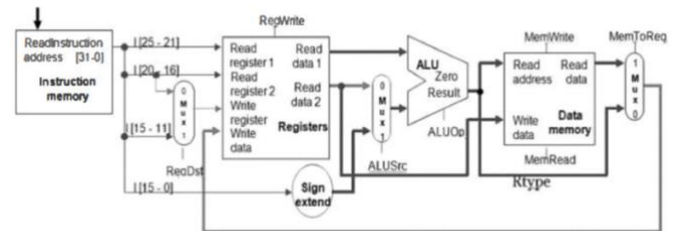


Figure 2.5 Data path for I-type instruction

The data path for I-Type instruction can be depicted in Figure 2.5. It shows that the Rt register can be used both as source and destination accordingly, and last 6 bits is the immediate value sent to sign extend and then to ALU for performing the required operation. It is used for ADDI, ANDI, and ORI operations. E.g., addi Rt, R. Here (Rs) + 5 is stored in the destination register Rt. 5 is an immediate value.

## 2.4 J-TYPE INSTRUCTION

Figure 2.6 shows the J-Type instruction format. The First 5 bits of this instruction format represent the type of branch operation to be performed. The remaining 26 bits represent the branch offset in 2's complement format. This number is added to the value of the PC to obtain the branch target address.

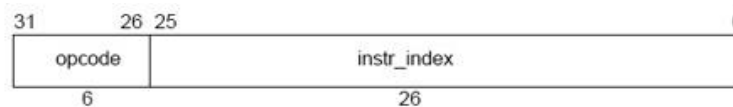


Figure 2.6 J-type instruction format

## 2.5 ENCODING OF MINI-MIPS INSTRUCTION

Table 1: Instruction set types and syntax of implemented instructions

Instruction	Type	Syntax
ORI	I	ORI R1<- R2+4;
SLL	R	SLL R2<-R3<<3
XNOR	R	XNOR R4 <- ~ (R5 ^ R6)
NAND	R	NAND R7 <- ~ (R8 & R9)
SUBI	I	SUBI R10 <- R11 - 4
ADDUI	I	ADDUI R12 <- R13 +7



		.....
BNE	I	BNE (R14! =R15) ADDR = PC+4+20; //5<<2 R14 R15 5
LH	I	LH R16, 2(R17)
SB	I	SB R18, R19(2)
JR	R	JR R2
J	J	J 2

### 3. MINI-MIPS ARCHITECTURE

MIPS architecture Is nothing but an architecture on which a pipelined system Is Implemented. The architecture In MIPS Is designed into a 5-stage pipeline system. The five stages are Instruction Fetch, Instruction Decode, Execution stage, Memory Access, and Write Back stage. Pipelining Is a system where more than one operation Is performed in the same pipeline stage. The advantage of a pipelined system Is that the overall efficiency and performance of the processor Are drastically Improved.

When comes to a multicycle CPU, It contains multiple processes. Let us consider the example of load and a branch Instruction; when we have a load Instruction, It Is expected to take five cycles, but another branch.

Instruction beq will only consume three cycles. So, rather than waiting for one process to complete, we can concurrently start another process. We need to make it work so that the previous process is not disturbed. For this to happen, we introduce buffers between each stage. These buffers store the values from the previous stage for the previous stage to start a new process. This process will lead to an increase in the throughput of the design.

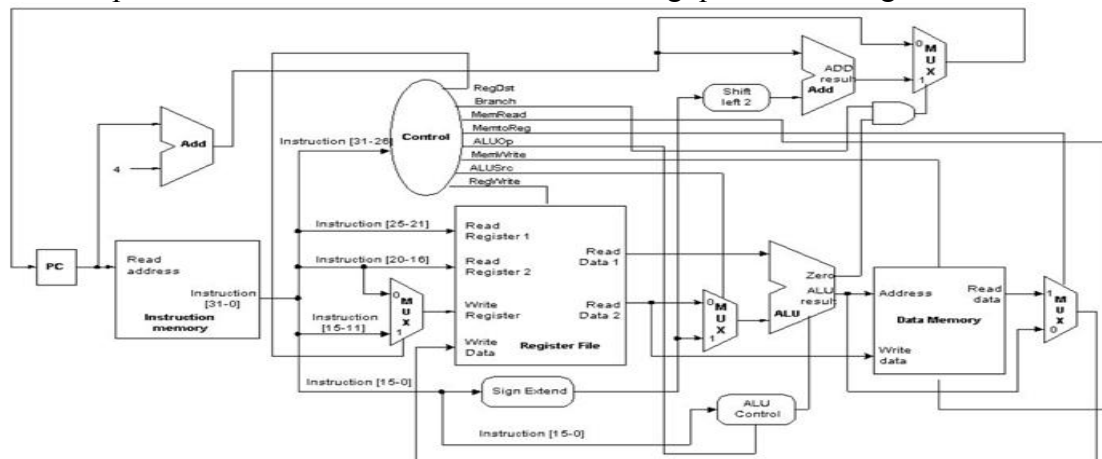


Figure 3.1: MINI-MIPS Architecture

#### 3.1 INSTRUCTION FETCH STAGE

In the instruction fetch stage initially, the Program counter will be pointing to Instruction Memory [0]. Once the first instruction is fetched, the Program counter will be incremented to 4 through the adder as indicated in the diagram. One of the inputs to the mux is the adder value and other inputs.

We consider the program counter to be 32bit wide and contain the following instruction address to be executed.

The instruction memory has 8bit each row, i.e., each instruction is 32bit wide, and we have a total of 512 rows. We can store 64 instructions in total.

In our project, the program counter forwards pc\_in to pc\_out on the positive edge of the clock, and if the program counter control is set to 1. Pc adder will increment pc\_out to pc\_out+4. The mux unit will forward Branch's address if mux\_control is set. Else it will forward PC+4 to the program counter.

Immediate register <= Memory [Program counter];  
Next\_Program\_counter = Program\_counter+4;

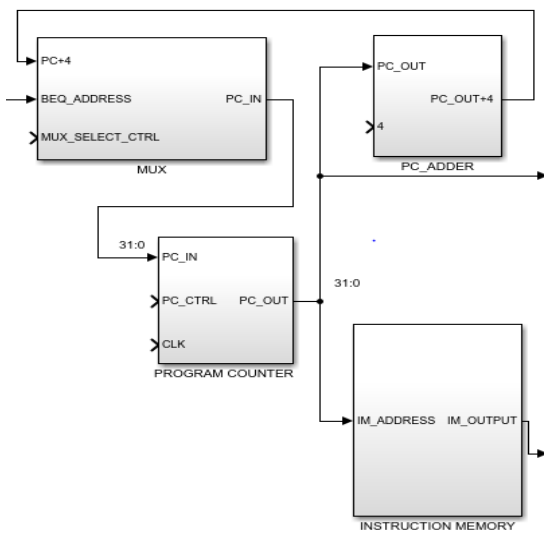


Figure 3.2 Block diagram of IF stage

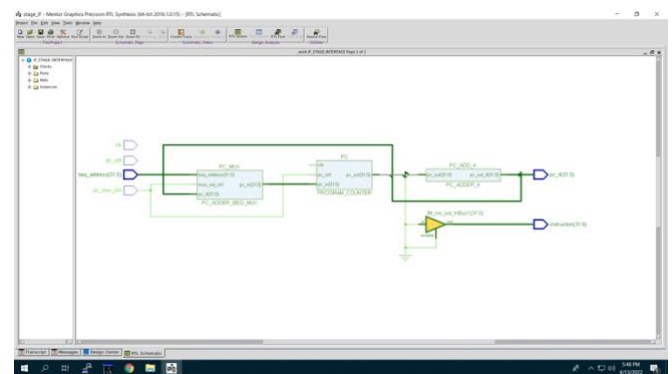


Figure 3.3 RTL diagram for IF stage

### 3.2 INSTRUCTION DECODE STAGE

The 32bit MIPS instruction will be fetched from the IF stage, the first 6bits, i.e., function code, will be forwarded to the ALU control unit, and the last six bits opcode will be sent to the control unit.

At the decode stage in our project, we assume 13 registers, and it is initialized decimal values from 0 to 13 for the registers 0 to 13. when the read enable is high, the instructions are decoded, and the register file is read into output registers. When the instruction is of immediate type, the sixteen bits are sign-extended for positive value zeros are appended and for negative value to the right. The ID/EX buffer gets data during the rising clock edge.

Read\_data\_1 <= Register\_File[Read\_Data\_1]  
Read\_data\_2 <= Register\_File[Read\_Data\_2]  
Temporary\_Register\_Immediate <= Sign\_extended\_field\_of\_IR

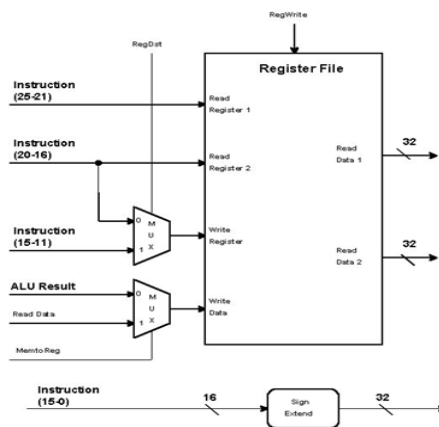


Figure 3.4 Block diagram for ID stage

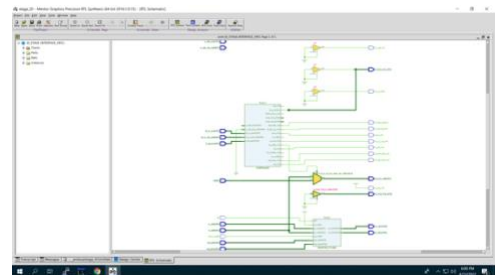


Figure 3.5 RTL diagram for ID stage

### 3.3 EXECUTION STAGE

Depending on the function code, the function code is sent from the decode stage to the execution stage.

The calculations are performed based on functional code.

- For Register -Register ALU instruction  
 $\text{Alu\_Result} \leq \text{Read\_input\_1 (function code) Read\_input\_2.}$
- For Register -Immediate ALU instruction  
 $\text{Alu\_Result} \leq \text{Read\_input\_1 (function code) Sign\_extended\_immediate.}$
- Memory reference  
 $\text{Alu\_Result} \leq \text{Read\_input\_1} + \text{Sign\_extended\_immediate.}$
- Branch operation  
 $\text{Alu\_Result} \leq \text{Next\_program\_counter} + \text{Sign\_extended\_immediate.}$   
 $\text{Condition} \leq (\text{Read\_input\_1 (function code) zero})$

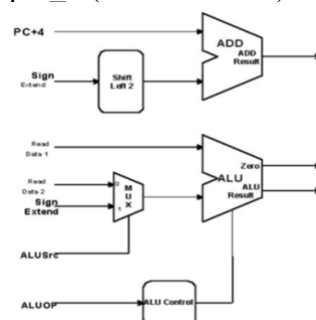


Figure 3.6 A block diagram of the execution stage.

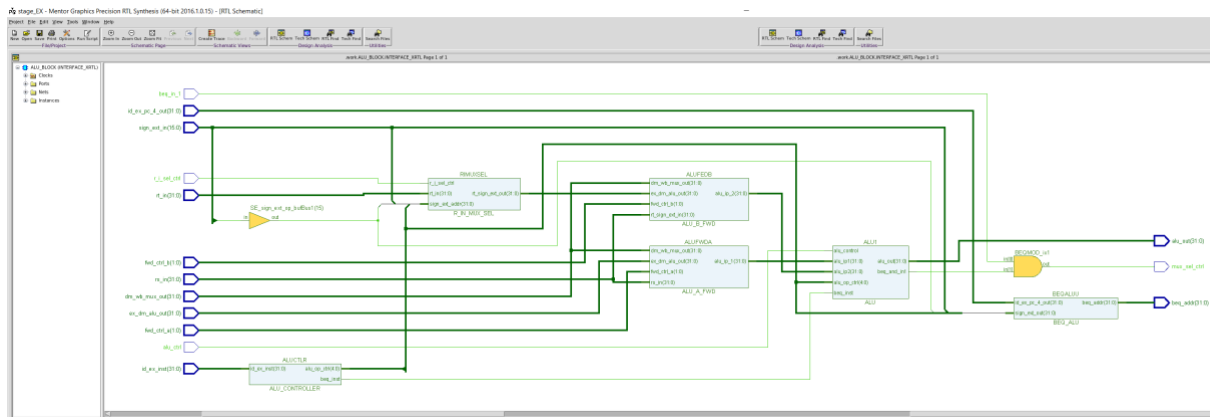


Figure 3.7 RTL diagram of the execution stage

### 3.4 ALU CONTROL UNIT

The ALU functional code is 6bits. It's decoded from the instruction address [31:26] to check the instruction type and depend on 6bits functional code [5:0]; it's assigned with a unique ALU control id based on arithmetic instructions.

For Branch instructions, The ALU functional code of 6bits is decoded from the instruction address [31:26] to check its branch type. In the next step, it is assigned with standard code for all branch instructions. Furthermore, if Alu output is equal to zero, the branch AND gate are set to one.

Arithmetic and logical Operation:

$Alu\_out(0000) = Alu\_ip1 \mid Alu\_ip2$  ; for OR operation  
 $Alu\_out(0001) = Alu\_ip1 \ll Alu\_ip2$  ; for SHIFT operation  
 $Alu\_out(0010) = \sim (Alu\_ip1 \wedge Alu\_ip2)$  ; for XNOR operation  
 $Alu\_out(0100) = Alu\_ip1 - Alu\_ip2$  ; for SUB operation  
 $Alu\_out(0101) = Alu\_ip1 + Alu\_ip2$  ; for ADD operation  
 $Alu\_out(0011) = \sim (Alu\_ip1 \& Alu\_ip2)$  ; for NAND operation

Table 2 Encoding of the Instruction Set for ALU control ID

Instruction	Type	Encoding	Opcode [31:26]	FUNC [5:0]	Alu control ID
ORI	I	b001101_00100_00010_00001_00000_000001	001101	000001	0000
SLL	R	b000000_00011_00011_00010_00000_000010	000000	000010	0001
XNOR	R	b000000_00101_00110_00100_00000_000100	000000	000100	0010
NAND	R	b000000_01001_01000_00111_00000_000101	000000	000101	0011
SUBI	I	b100010_00100_01011_01010_00000_000110	100010	000110	0100
ADDUI	I	b001001_00111_01101_01100_00000_000111	001001	000111	0101
BNE	I	b000101_00101_01111_01110_00000_001000	000101	001000	NA
LH	I	b100001_00001_10001_10000_00000_001010	100001	001010	NA
SB	I	b101000_00001_10011_10010_00000_001001	101000	001001	NA
JR	R	b001000_00000_00010_00000_00000_001010	001000	001010	NA
J	J	b000010_00000_00000_00000_00000_000010	000010	000010	NA

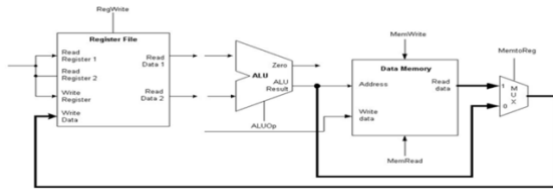


Figure 3.8 Block diagram for EX, DM, WB stage

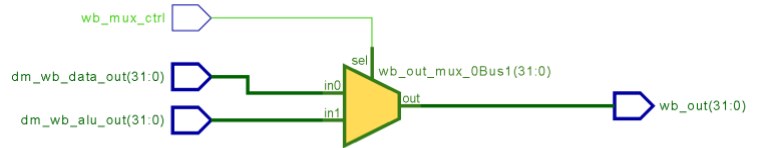


Figure 3.9 RTL diagram for write-back operation.

### 3.5 DATA MEMORY STAGE

#### Memory Read:

During load instruction the control input for memory read will be enabled and if load instruction is with offset the ALU\_Output will be, [Register\_Data\_1+Sign\_extended\_Immediate\_value] then data memory output will be memory[Register\_Data\_1+Sign\_extended\_Immediate\_value].

#### Memory Write:

The control input for memory\_write will be enabled during store instruction, and if store instruction is with offset, the ALU\_Output will calculate address [Register\_data\_1+Sign\_extended\_Immediate\_value]. The Register data with store content will be forwarded to Write data of data; then the registration data will be written in memory location Memory[Register\_data\_1+Sign\_extended\_Immediate\_value] <= Register\_data\_1.

#### Branch:

During branch operation, if the condition is satisfied, the program counter value will be updated with an output of ALU\_Output.

### 3.6 WRITE BACK STAGE

Write backstage is used when we have to write the memory value or the result of the ALU operation.

Into register file. The multiplexer at writes backstage has a control signal coming from the control unit. The control signal selects the value written into the register file. If the control signal is zero, it means the control unit determined that the instruction is of Register-Register type. Only the output of the Alu will be written back to memory. If the control signal is one, the control unit determines that instruction is of Load type, and read data from memory will be written back to the register.

Register\_file[Register\_destination\_address] <= Alu\_Output  
Register\_file[Register\_destination\_address] <= Load\_Memory\_data

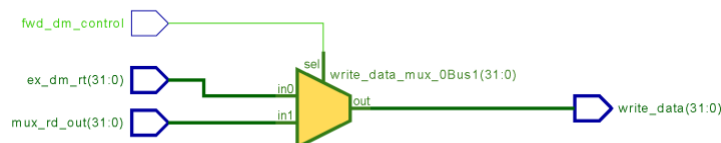


Figure 3.10 RTL diagram for DM operation.

Table 3: Encoding of the Instruction Set

Instruction	Type	Encoding	Syntax	Opcode [31:26]	Func Code[5:0]
ORI	I	b001101 _00100_00010_00001_00000 _000001	ORI R1<- R2+4;	001101	000001
SLL	R	b000000 _00011_00011_00010_00000 _000010	SLL R2<-R3<<3	000000	000010
XNOR	R	b000000 _00101_00110_00100_00000 _000100	XNOR R4 <- ~ (R5 ^ R6)	000000	000100
NAND	R	b000000 _01001_01000_00111_00000 _000101	NAND R7 <- ~ (R8 & R9)	000000	000101
SUBI	I	b100010 _00100_01011_01010_00000 _000110	SUBI R10 <- R11 - 4	100010	000110
ADDUI	I	b001001 _00111_01101_01100_00000 _000111	ADDUI R12 <- R13 +7	001001	000111
BNE	I	b000101 _00101_01111_01110_00000 _001000	BNE (R14!=R15) ADDR = PC+4+5; R14 R15 5	000101	001000
LH	I	b100001 _00001_10001_10000_00000 _001010	LH R16, 2(R17)	100001	001010
SB	I	32'b101000 _00001_10011_10010_00000 _001001	SB R18, R19(2)	101000	001001
JR	R	32'b001000 _00000_00010_00000_00000 _001010	JR R2	001000	001010
J	J	b000010_00000_00000_00000_ 00000_000010	J 2	000010	000010

#### 4. HAZARD CONTROL DESIGN

When we are running instructions on a pipelined system, we may come across a few errors due to dependencies of the operands on other instructions, i.e., one instruction may need an operand value computed by another instruction. These types of errors in a pipelined system are known as hazards.

Following are the three hazards that we come across during execution

- Data hazard
- Structural hazard
- Control Hazard

## 4.1 DATA HAZARD

These types of errors occur when one instruction, say instruction I is dependent on the result of another instruction, says instruction J. There are three types of data hazards the Read After Write (RAW) hazard, the Write After Read (WAR) hazard, and the Write After Write (WAW) hazard.

Read After Write hazard: The following hazard occurs when instruction J attempts to read an operand value before instruction I writes it.

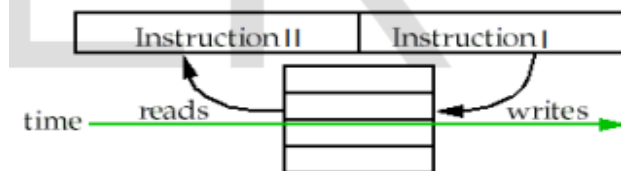


Figure 4.1 Read After Write hazard



Figure 4.2 Write After Read hazard

Write After Read hazard: The following hazard occurs when instruction J attempts to write an operand value before instruction I reads it.

Write After Write hazard: The following hazard occurs when instruction J attempts to write an operand value before instruction I finishes its write operation.

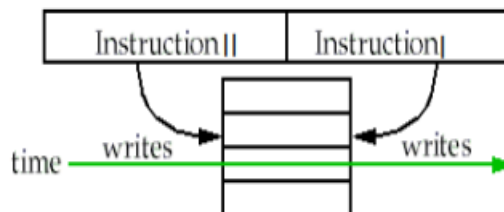


Figure 4.3 Write After Write Hazard

The solution to RAW Hazard

Consider the following Instruction,

```
ADD R11, R10, R9  
MUL R15, R11, R12
```

As we can see from the above instructions, Instruction MUL tries to the operand value R11 before the Instruction ADD has completed its writing process. The instruction MUL reads the wrong value of operand R11 to execute the instruction, causing the hazard.

These types of hazards can be solved using some special techniques.

Solution 1: Reordering the instruction

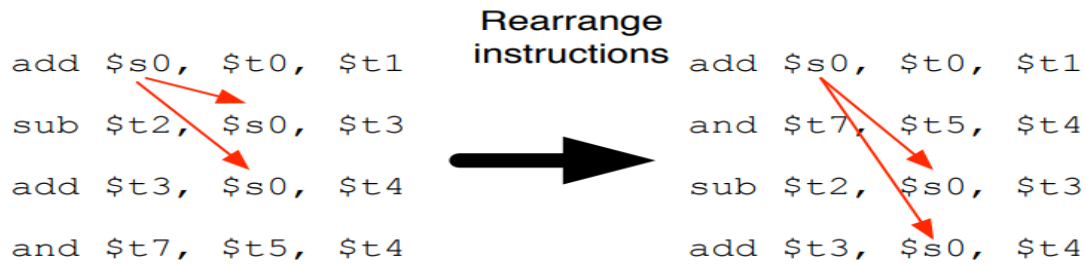


Figure 4.4 Solving RAW by reordering instructions.

As shown in Figure 4.4, the second and third instructions (SUB and ADD) are dependent on the value `s0` written by the first instruction ADD. This will end up causing a RAW hazard and hence needs to be resolved. As seen in Figure 4.4, the instructions are rearranged by moving instructions 2 and 3 to positions 4 and 5. This will result in a system without the RAW error.

#### Solution 2: Data forwarding

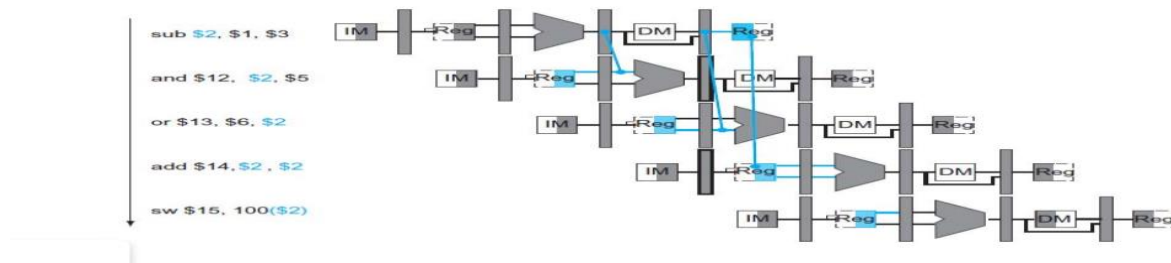


Figure 4.5 Data forwarding to solve RAW hazard

As seen in Figure 4.5, instructions 2, 3, and 4 use operand `$2` written by the first SUB instruction online. We can see that the value of `$2` is being computed at the execute stage of the first instruction and is being forwarded to all the other instructions requiring it.

## 4.2 STRUCTURAL HAZARD

Structural hazard occurs in a pipelined system when two or more instructions simultaneously try to access the same resource.

Instruction / Cycle	1	2	3	4	5
I <sub>1</sub>	IF(Mem)	ID	EX	Mem	
I <sub>2</sub>		IF(Mem)	ID	EX	
I <sub>3</sub>			IF(Mem)	ID	EX
I <sub>4</sub>				IF(Mem)	ID

Figure 4.6 Pipelined system with structural hazard

Cycle	1	2	3	4	5	6	7	8
I <sub>1</sub>	IF(Mem)	ID	EX	Mem	WB			
I <sub>2</sub>		IF(Mem)	ID	EX	Mem	WB		
I <sub>3</sub>			IF(Mem)	ID	EX	Mem	WB	
I <sub>4</sub>				-	-	-	IF(Mem)	

Figure 4.7 Stalls to avoid structural hazards



As we can see from Figure 4.6, Instruction 1 and Instruction 4 are attempting to access the same resource and can hence lead to structural hazards.

The solution to stall structural hazard:

Figure 4.7 shows that stalls are added for the fourth instruction in cycles 4, 5, and 6 to escape the memory access conflict.

### 4.3 CONTROL HAZARD

When we have a branch instruction, the outcome of the branch will not be calculated for some time, and the pipeline may end up fetching the wrong instruction.

Consider the following instruction,

```

1:ADD R1, R2, R3
2:SUB R4, R5, R6
3:BEQ R1, R4, 40
4:MUL R4, R7, R2
.
.
.
40: ADD R4, R7, R2

```

Here the branch will not be computed until the execution stage, and the pipeline system will start fetching and decoding the operands for MUL instruction which may not be needed if the branch is taken.

Solution: Introduce stalls

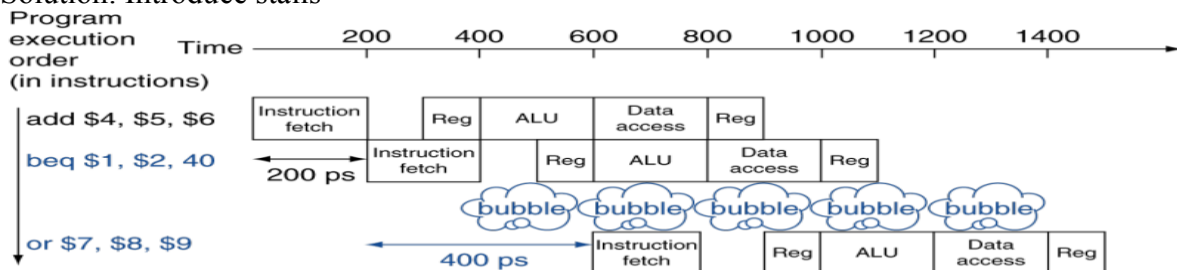


Figure 4.8 Stalls to prevent Control hazard

As seen in Figure 4.8, stalls are added until the branch decision is computed to get a proper branch decision and not to compute unwanted instructions.

## 5. TEST BENCH AND SIMULATION RESULT

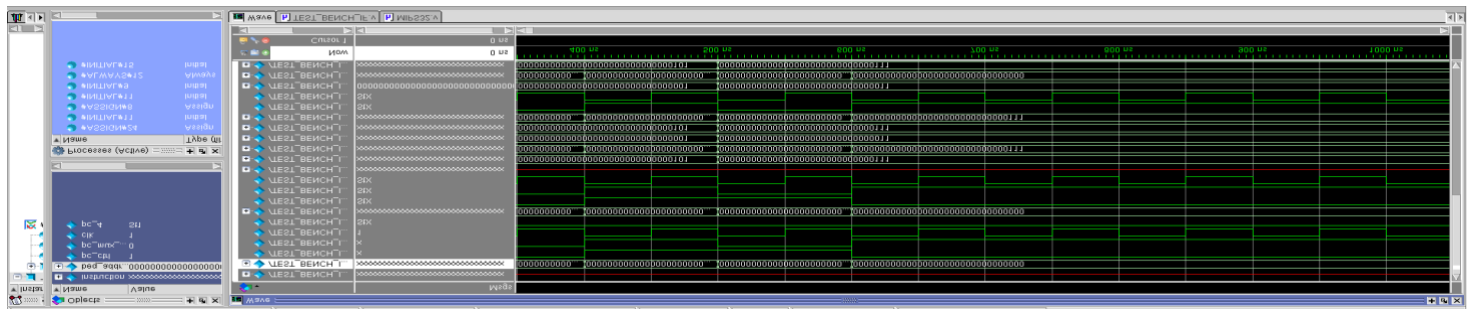


Figure 5.1 IF stage simulation

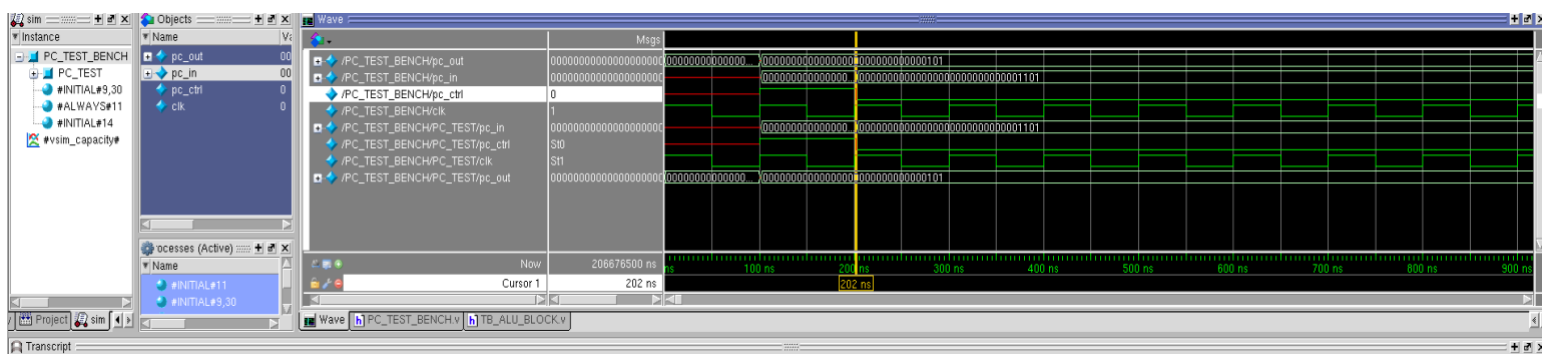


Figure 5.2: Program counter (part of IF stage) simulation

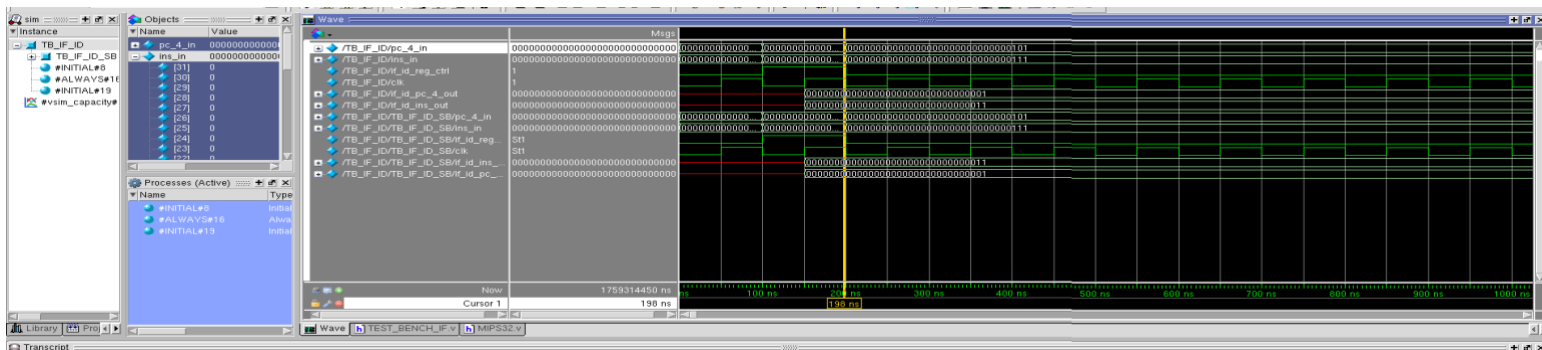


Figure 5.3 Simulation for IF\_ID buffer

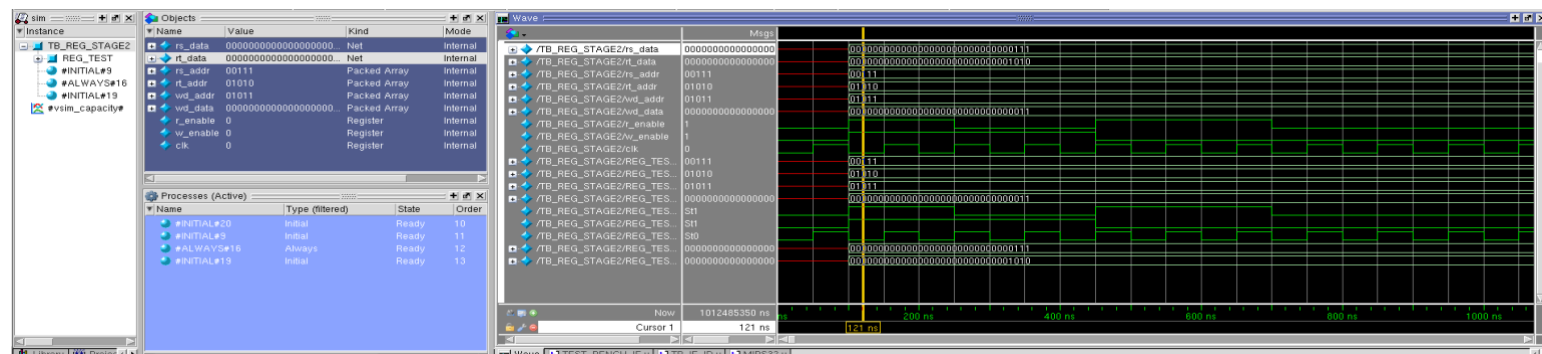


Figure 5.4 Simulation for ID stage (Register file)

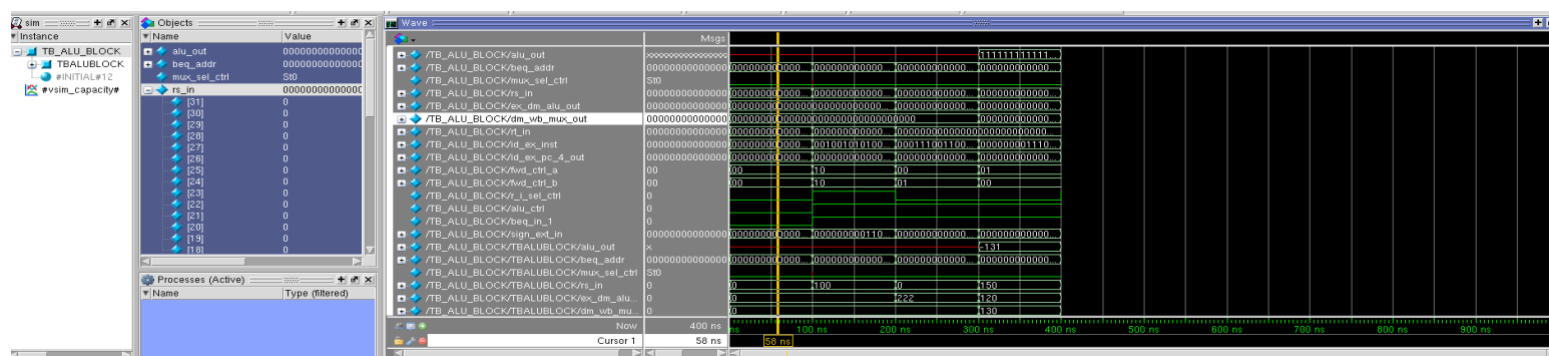


Figure 5.5 Simulation for ALU block (EX stage)

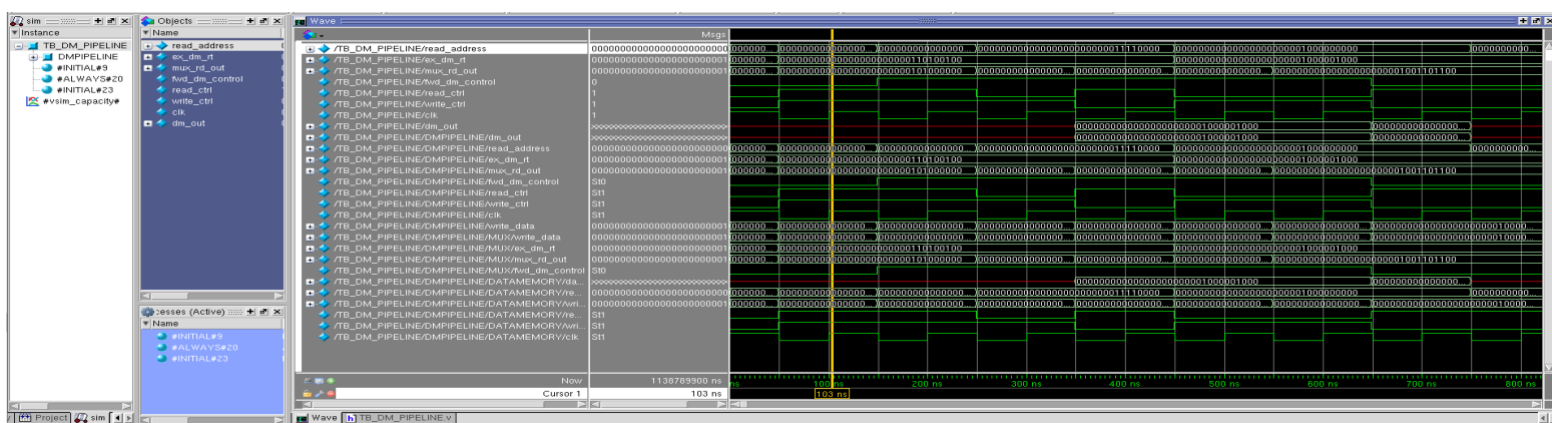


Figure 5.6 Simulation for MEM Stage

## 6. CONCLUSION

The main goal of this project was to successfully implement a 32-bit mini-MIPS microprocessor that can run 11 instructions in a pipelined system. The code for the processor was written using Verilog and has been run on ModelSim. The code was tested using test benches, and the simulation result for each architecture stage was shown. A precision RTL tool was used to synthesize the design, and the corresponding RTL diagrams were shown.

## 7. RESPONSIBILITIES

Team Members	Responsibilities				
	Group Leader	Coding and testing the code	Report writing	Poster Design	RTL Diagram
Abishek Arumugam	✓	✓	✓		✓
Javier Sandoval Gonzales		✓	✓		✓
Manish Pejathaya		✓	✓	✓	
Srinidhi Honakere Srinivas		✓	✓	✓	

Table 4 Responsibilities

## 8. REFERENCES

- [1] "[https://www.econcordia.com/courses/computer\\_architecture/lessons.aspx](https://www.econcordia.com/courses/computer_architecture/lessons.aspx)"
- [2] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [3] Bhimani, Husainali S., Hitesh N. Patel, and Abhishek A. Davda. "Design of 32-bit 3-stage pipelined processor based on MIPS in Verilog HDL and implementation on FPGA Virtex7." *International Journal of Applied Information Systems* 10.9 (2016).