



DESIGNING OF PIPELINED PROCESSOR MINI MIPS

COEN 6741

SUBMITTED BY:
GROUP 1

- Mohammed Abu Fares (40137642)
- Syeda Butool Fatima Jafri (40151028)
- Arsal Jawaid (40166161)
- Constantine Kokorogiannis (40032046)

CONTENTS

1.INTRODUCTION	5
1.1.1) CISC VS RISC ARCHITECTURE.....	5
1.2) MINI MIPS	5
2.INSTRUCTION SET DESIGN AND ENCODING	6
2.1) Instruction Set and Encoding for MIPS	6
2.2) Instruction Encoding for our Project	7
3.DATAPATH & CIRCUIT DESIGN	9
4. HAZARDS IN PIPELINE.....	11
4.1) Structural Hazard	11
4.2) Data Hazard	11
4.3) Control Hazard	12
4.CONTROL UNIT DESIGN & HAZARD HANDLING	13
4.1) Control Unit Design.....	13
4.2) Hazard Handling.....	13
5. ASSEMBLY CODE & INSTRUCTIONS FLOW	14
5.1) Instructions Flow.....	14
5.2) Assembly Code.....	15
5.SIMULATION RESULTS	17
6.CONCLUSION	18
References	19



List of Figures

Figure 1: RISC V base instruction formats.....	6
Figure 2: Data path and control unit design	10
Figure 3: Data Hazard example	12
Figure 4: Instructions Flow.....	14
Figure 5: Simulation result part 1	17
Figure 6: Simulation results part 2	18

List of Tables

Table 1: Project Instructions	8
Table 2: Instruction encoding	9
Table 3: Control Signals for Instructions	13
Table 4: Hazard unit control signal	14
Table 5: Assembly Code with explanations	17



ABSTRACT

In this project, we were given the task of building a 32-bit MIPS architecture processor. Our simple “Mini-MIPS” processor is a subset of the 32-bit MIPS architecture which will integrate ISA in the form of 3 instruction formats, R, I and J-Types. We were given a list of instructions (Variant 4) and the design should be hazard free. This project had to be pipelined to improve the efficiency of our processor. This pipelining involves five stages, instruction fetch, instruction decode, execute, data memory and write back.

The instructions we were given to implement were: ORI, SLL, XNOR, NAND, SUBI, ADDUI, BNE, LH, SB, JR and J. They had to be hazard-free with one clock cycle to access memory. We had to analyze and understand the behavior of each instruction and design each part of the pipeline. The final product is a Mini-MIPS processor using 32-bit instruction and was designed completely in VHDL.

1.INTRODUCTION

1.1) BACKGROUND

Instruction set architecture is basically an abstract model of a computer or what known as a central processing unit (CPU). The ISA defines the registers, memory, features, and data types of the computer. Different processors have different ways of interpreting machine language, which is built up from instructions and statements. Machine language can be used to access different registers, particular memory locations and different addressing modes. Examples of this can be setting a register to a constant value, copying data from a memory location, or reading/writing data onto hardware devices. It can also do arithmetic and logic operations such as add, subtract, multiply and divide.

1.1.1) CISC VS RISC ARCHITECTURE

Early processors were based on the 16-bit architecture, whereas the 32-bit ones were developed later. We have two kinds of processors, CISC and RISC. CISC is an instruction set that executes single low-level instructions. CISC tries to complete a task in as few lines as possible. This means that the hardware for CISC is better because it can understand a series of operations. CISC also allows operation for multi-clock instructions. For the RISC approach, it only uses simple instructions, each getting executed at each clock cycle. RISC emphasizes software whereas CISC emphasizes hardware. RISC also allows bigger code sizes. The reason behind having recent processors built using CISC since it allows for multi clock instructions and far more complex instructions to be executed. This will result in the task being done quicker and more efficiently.

1.2) MINI MIPS

Mini MIPS is a subset of the MIPS (Multiple Instructions per second) 32-bit architecture which is based on the RISC (Reduced instruction set architecture). In RISC, there are three types of instructions, which are ALU, load/store, and branch/jump. It also has a special technique that we must implement called pipelining. Pipelining uses five stages to enable running instructions on each clock cycle in parallel. These stages include instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM) and write back (WB). The instruction fetch stage takes the instruction from memory and gives it to the instruction decode stage. The instruction decode stage decodes the instruction and sends commands to the control unit to get ready to execute. In the execute stage, the instruction is executed by the ALU which performs the necessary operations. The memory stage either stores or fetches data in/from memory, and finally the write-back stage writes values to given registers.

For ALU instructions, we typically have two registers or an immediate value to perform the logical operation. These instructions typically have math/logic applied to them, with examples being: ADD, SUB, OR, AND. For the load/store instructions, typically one register is used for the load instruction, with another immediate value to offset where we want to load a value from. Using the store instruction, two registers are usually used, one to hold the value, and another to write to store in memory. For the branch/jump instructions, they are executed when a condition is proved true/false. They are usually used to compare a register to an immediate value or another register.

2.INSTRUCTION SET DESIGN AND ENCODING

2.1) Instruction Set and Encoding for MIPS

An instruction set architecture is basically an abstract model of a computer and generally contains a mixture of registers, data types, memory blocks and a CPU. ISA's can be classified with various complexities, the more common ones being either RISC or CISC. A number of different architectures can be used, in our case, we are using the Mini-MIPS architecture that uses RISC and 32-bit instruction and data. Also, it uses addressing mode of register plus immediate for load/store instructions with fixed instruction format.

Instructions in MIPS are classified into different types:

- **Arithmetic/Logic Operations:**
These are all the math operation instructions which include: ADD, SUB, MUL, AND, ETC.
- **Memory Access Operations:**
These are for load/store instructions.
- **Control Flow Operations:**
These contain all of conditional/unconditional branch instructions.

Encoding in MIPS processor is required to make a specific sequence of bits that will allow different implementation of different instructions. The following are three types of instructions used in MIPS:

-R-type instructions: These types of instructions are used when there are data values in both registers being written to a third. The opcode is the machine code that represents the instruction required. Rs, Rt, and Rd are the registers being used by the instruction. *SHAMT* is the shift amount (usually zero) and *FUNCT* is for instructions that share an opcode. R-type instructions are typically arithmetic/logic ones.

-I-type instructions: These types of instructions usually involve one register with an immediate value which are maximum 16-bit long. These instructions can include ADDI, ORI, and conditional branch compares.

- J-type instructions: This type of instruction is used when a jump is being performed. It updates the PC and points you to the target address.

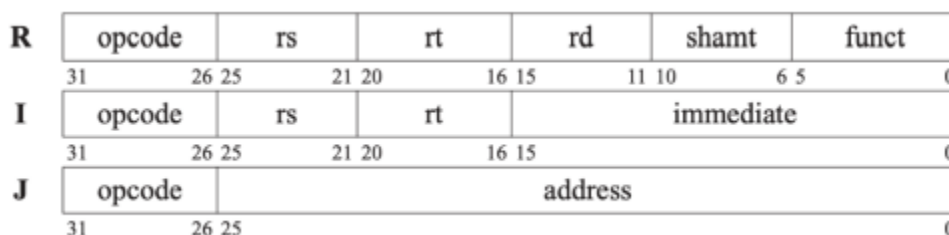


Figure 1: RISC V base instruction formats

2.2) Instruction Encoding for our Project

For the scope of this project, we were given a list of 11 instructions, which are categorized as follows:

- Arithmetic/Logic Operations:
 - ORI - Logical OR with immediate value
 - SLL - Shift Left Logical
 - XNOR - Logical XNOR
 - NAND - Logical NAND
 - SUBI - Subtract with immediate value
 - ADDUI - Add immediate value (ignores overflow)
- Memory Access Operations:
 - LH - Load halfword
 - SB - Store byte
- Control Flow Operations:
 - BNE - Branch Not Equal
 - JR - Jump Register
 - J - Jump

The format of these instruction along with their descriptions can be described from the table below:

INSTRUCTION		OPERATION	TYPE
ORI	OR register-Immediate	logical operations that perform bitwise OR on register Rs and the sign-extended 12-bit immediate and place the result in Rt	I-type
SLL	Shift logical left	Perform logical left shifts on the value in register Rt by the shift amount held in the SHAMT/SA. SLL Moves bits from a lower position to a higher position. logical shift: fill in 0s when value moved to the left by number of positions	R-type
XNOR	Exclusive NOR	Boolean: performs the XNOR of Rs and Rt, results are written to the destination Rd.	R-type
NAND	NAND	Boolean: performs the NAND of Rs and Rt, results are written to the destination Rd.	R-type
SUBI	Subtract immediate	Subtracts the sign-extended 16-bit immediate from register Rs and place the result in Rt	I-type
ADDUI	Add immediate unsigned	Adds the sign-extended 16-bit immediate to register Rs and place the result in Rt	I-type
BNE	Branch not equal	take the branch if registers Rs and Rt are unequal	I-type

LH	Load half word	LH loads a 16-bit value from memory before storing in Rt	I-type
SB	Store byte	store 8-bit values from the low bits of register Rt to memory.	I-type
JR	Jump register	Jump to PC with reference to value in Rs	I-type
J	Jump	Unconditional direct jump. [25:0] of instruction J is shifted to left by 2 bits. Then the resulted [28:0] bits are replaced with the [28:0] bits of PCPlus4D, which will result the Jump address.	J-type

Table 1: Project Instructions

In addition to that, the table below illustrates the encoding code used in the simulation:

CODE	LH	Rs	RT	IMM	32BIT INST	HEX
LH R1, 8(R0)	100001	00000	00001	00000000000001000	10000100000000001000000000000000 0001000	84010008

CODE	OpCode	RS	RT	IMM	32BIT INST	HEX
ORI R3, R1,1	001101	00001	00011	00000000000000001	00110100001000110000000000000000 0000001	34230001

CODE	OpCode	Rs	RT	RD	SA	Func	32BIT INST	HEX
SLL R4,R3,2	000001	00000	00011	00100	00010	000000	000001000000000011001000001 0000000	04032080

CODE	OpCode	RS	RT	RD	SA	Func	32BIT INST	HEX
XNOR R7,R6,R5	010010	00101	00110	00111	00000	000000	0100100010100110001110000 0000000	48A63800

CODE	OpCode	RS	RT	IMM	32BIT INST	HEX
ADDUI R8, R3, 7	001101	00011	01000	00000000000000111	0011010001101000000000000000 0000111	34680007

CODE	OpCode	RS	RT	IMM	32BIT INST	HEX
BNE R8, R9, 1	000101	01000	01001	00000000000000001	0001010100001001000000000000 0000001	15090001

CODE	OpCode	RS	RT	RD	SA	Func	32BIT INST	HEX
NAND R14, R12,R13	000111	01101	01100	01110	00000	000000	0001110110101100011100000 0000000	1DAC7000

CODE	OpCode	26 bits		32BIT INST		HEX
J 26 bit value is 11	000010	0000000000000000000000001011		000010000000000000000000000000000001011		0800000B

CODE	OpCode	RS	RT	IMM	32BIT INST	HEX
SB R10, 11(R11)	101000	01011	01010	00000000000001011	101000010110101000000000000000000001011	A16A000B

CODE	OpCode	RS	RT	IMM	32BIT INST	HEX
SUBI R15, R4,6	110000	00100	01111	00000000000000110	1100000010001111000000000000000000000110	C08F0006

CODE	OpCode	RS	RT	IMM	32BIT INST	HEX
JR R24,R0,0	001000	11000	00000	00000000000000000	0010001100000000000000000000000000000000	23000000

CODE	LH	Rs	RT	IMM	32BIT INST	HEX
LH R2,12(R25)	100001	11001	00010	00000000000001100	10000111001000100000000000000000000001100	8722000C

Table 2: Instruction encoding

3.DATAPATH & CIRCUIT DESIGN

The data path involves various collection of various functional units including the arithmetic logic units (ALU's) and multiplexer, which are required to perform data operations on registers and buses. Mini MIPS is composed of five pipelines namely Instruction Fetch, Instruction Decode, Execution Stage, Memory Stage, and the writeback stage. The memory used here is byte-addressable. Any potential of hazards are detected by Hazard Unit, which prevents different hazards to occur such as structural, data, and control hazards. Figure below shows the circuit and following list explain different main blocks used in the MIPS circuit datapath:

- **Instruction Memory:** Required to store instructions
- **Register File:** Hold registers of operands
- **Control Unit:** Generates the required control signals for each specific instruction
- **ALU:** Perform arithmetic operations
- **Data Memory:** Store and load data
- **Buffers:** Hold and release data when required on falling edge
- **Unconditional Jump block:** Used to set PCsrc to 1 when we have J or JR instructions.
- **J calculator block:** Used to calculate the J address by shifting first 26 bits of the J instruction to left by 2, and replace it with PCPlus4D.
- **Sign Extend:** used to complete the sign of last bit for 15:0 bits.
- **2's compliment:** Used to take 2's compliment for negative numbers coming from execution stage to avoid having negative integers interpreted by data memory
- **Shamt:** Generates high signal when we have shift instruction to allow SHAMT to enter ALU

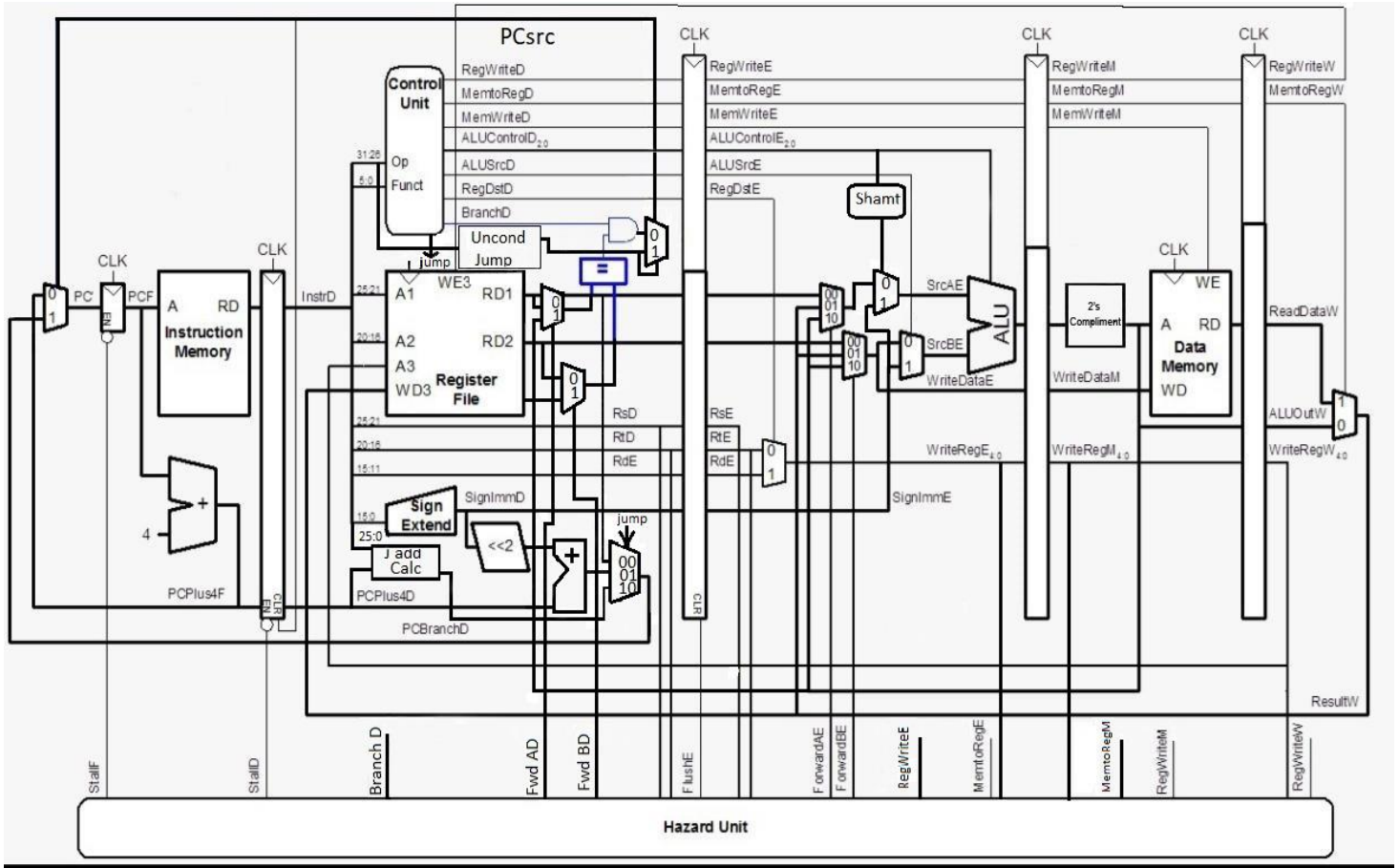



Figure 2: Data path and control unit design

MIPS implements a five-stage pipeline architecture as mentioned earlier. Following are the five stages of the MIPS pipeline each of which requires one cycle:

Instruction Fetch stage (IF): In the instruction stage, the instruction is fetched from memory and placed into the instruction register (IR). The instruction is 32 bits and gets decoded in the next stage (ID). The program counter (PC) is incremented by 4 and written back into the PC to point to the next instruction. The PC is used to access the instructions by pointing at them in memory. This stage occurs in every single instruction.

Instruction Decode stage (ID): The instruction is decoded into control signals. These control signals vary depending on the type of instruction. Also, in this stage, the register file is accessed and the operands are chosen based on the inserted instruction. If we have a control instruction such as branch, jump, or jump register, the new address is calculated in this stage.

Execution stage (EX): In this stage, the instruction is executed. If it was an R type instruction, arithmetic or logical operation is performed with the registers provided. If it was a load-store instruction, the address is computed (I-type). All this is done by the logic circuit called the arithmetic-logical unit (ALU).



Memory Read/Write stage (MEM): This is the stage where we can access memory, typically through a load/store instruction. The memory can either be read or written to. In this stage, ALU result can be forwarded to the previous stages, saving time in the process. This forwarding ensures that instructions always write their results so that one writes port can be used and is always available.

Write Back (WB): This is where the results of the operations are written to their destination registers. This stage accesses the register file as well, meaning that the decode stage and writeback stage are both accessing this file. This can cause a hazard if one register is being read/written at the same time.

Hazard Unit (HU): This section is responsible in detecting any possible hazard and control the data path accordingly. Next sections will describe this part in more details.

4. HAZARDS IN PIPELINE

MIPS processor uses the idea of pipelining to try and execute an instruction per clock cycle. Each instruction passes through every stage mentioned earlier to make this happen. There can be certain setbacks however and these setbacks include structural hazards, data hazards and control hazards.

4.1) Structural Hazard

Structural hazards arise usually from hardware issues. It happens when a resource conflicts with another when they are both trying to use the same resource simultaneously at a different point in the pipeline. This problem can happen for many reasons, especially if we don't have enough resources such as one register port or one memory register. If different stages of the pipeline try and use the same register, it'll cause a structural hazard. To resolve these types of hazards we can use multiple resources. This hazard cannot occur in our design since we have different memories for the instructions and data.

4.2) Data Hazard

Data hazards occur when the result of a previous instruction is not updated to the correct value by the time the next instruction uses it in the pipeline. The important stage to minimize this issue is the Execution stage. We need to be aware if a value is changing in an instruction that was issued previously before the next instruction uses it. An easy fix for this issue is to rearrange the order of the instructions to avoid data hazards. Another way to fix this issue is by implementing more stalls to make sure a value is not read/written before it is ready.

There are three main types of data hazards, which are Read-After-Write, Write-After-Read, and Write-After-Write. The list below describes each type and address whether it can happen in MIPS or not:

a) **Read-After-Write (RAW):**

This hazard happens when an instruction reads a specific value before being written from previous instruction. This type of hazard can happen in MIPS.

b) **Write-After-Read (WAR):**

This hazard happens when an instruction writes an operand before reading it by another instruction. This type can't happen in MIPS because reads are always in stage 2, and writes are always in stage 5.

c) **Write-After-Write (WAW):**

This hazard happens when a write happens on same register after being written. This type also cannot happen in MIPS because writing can only happen in stage 5.

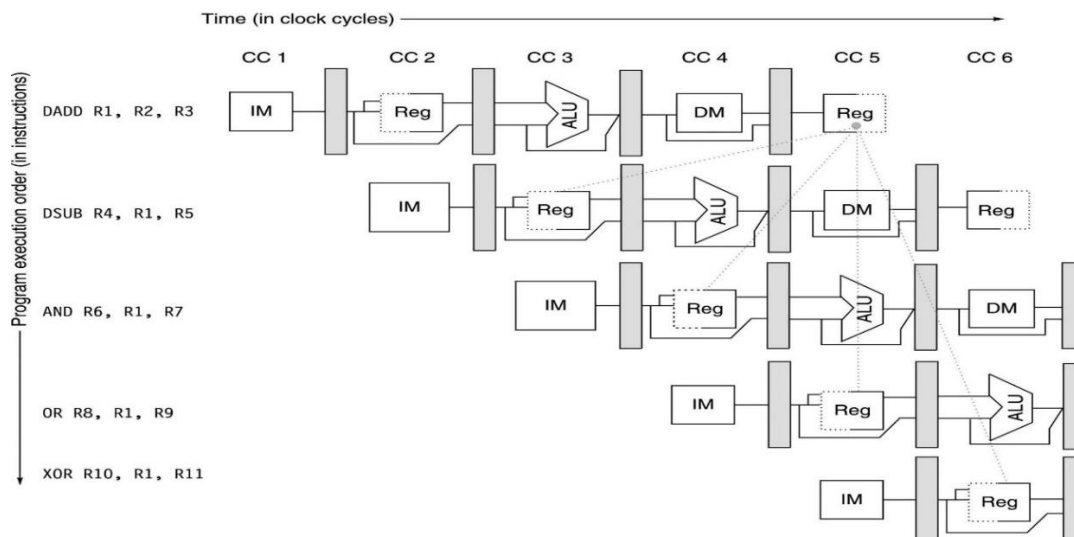


Figure 3: Data Hazard example

4.3) Control Hazard

Control hazards usually occur with branch instructions or others that change the flow of the program, such as the program counter. These hazards can cause huge performance loss, much more than the other two hazards. When a branch is executed, it might or might not change the program counter, usually called taken branches/not taken branches. The problem arises in the MEM stage when a branch is taken. This is where the execution takes place to determine the new program counter. The easiest fix is to stall the pipeline until the instruction reaches this stage, and this delay caused by stall is called branch delay. Another way to solve this hazard is to calculate the branch address and make the comparison in EX stage, which is the case in our project.

4.CONTROL UNIT DESIGN & HAZARD HANDLING

4.1) Control Unit Design

The purpose of the control unit is to maintain and preserve the flow of instructions in the pipeline stages so that each stage is aware of its operations. It is implemented in the decode stage of the pipeline. The opcode and function field are extracted from the instruction set and fed to the control unit which generates various control signals for the proper execution and completion of instruction. The following table summarizes the control signals for our project's instructions:

Type	Instruction	OpCode	ALU Control	RegWrite	MemtoReg	MemWrite	ALUSrc	RegDst	BranchD	Jump
R	NAND	000111	100	1	0	0	0	1	0	00
	XNOR	010010	011	1	0	0	0	1	0	00
	SLL	000001	010	1	0	0	0	1	0	00
I	ORI	001101	001	1	0	0	1	0	0	00
	ADDUI	001101	000	1	0	0	1	0	0	00
	LH	100001	000	1	1	0	1	0	0	00
	SB	101000	000	0	0	1	1	0	0	00
	SUBI	110000	101	1	0	0	1	0	0	00
	BNE	000101	000	0	0	0	0	0	1	01
	JR	001000	000	0	0	0	0	0	0	00
J	J	000010	000	0	0	0	0	0	0	10

Table 3: Control Signals for Instructions

4.2) Hazard Handling

The purpose of Hazard unit is as mentioned earlier to detect any possible hazard in the circuit, especially the data dependency. Hazard unit compares the destination address of a specific instruction with the operands of the next instructions. Then, depending on stage place of detection, the unit will choose control stalling, forwarding, and flushing whenever required. Table below summarizes the signals generated by the Hazard Unit along with its task. In addition to that, some signals from control units are entering the hazard unit, and this is needed to provide more control ability of different cases. For example, when we insert control signal such as **MemtoReg_E** in the hazard unit, we can use it as a condition to detect a load hazard and stall accordingly if required.

Signal	Description
Stall_F	Will stall PC when required
Stall_D	Will Stall the buffer of IF stage when required
Forward_AD	Select the required signal mux to forward the needed value if needed , otherwise, it is set to 0
Forward_BD	
Flush_E	Reset the EX buffer when required
Forward_AE	Select the required signal mux to forward the needed value if needed, otherwise, it is set to 00
Forward_BE	

Table 4: Hazard unit control signal

5. ASSEMBLY CODE & INSTRUCTIONS FLOW

5.1) Instructions Flow

Here we describe the step-by-step execution of the code sequence in section 5.2 and identify potential hazards introducing stalls and data forwarding techniques to overcome them. The figure below shows the instruction flow in relation with table 5. The arrows indicate the places where forward occurred. For instance, at cycle 5, we can see that a forward happened from WB stage of instruction LH to EX stage of ORI instruction in the same cycle. In addition to that, the S indicates the cycle where a stall was required. Next section provides more details on each instruction flow.

CPU Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
LH R1, 8(R0)	IF	ID	EX	MEM	WB																
ORI R3,R1,1		IF	S	ID	EX	MEM	WB														
SLL R4, R3, 2			S	IF	ID	EX	MEM	WB													
XNOR R7,R6,R5					IF	ID	EX	MEM	WB												
ADDUI R8, R3, 7						IF	ID	EX	MEM	WB											
BNE R8, R9, 1							IF	S	ID	EX	MEM	WB									
LH R2,12(R25)								S	IF (FLUSHED)	X	X	X	X								
NAND R14, R12,R13										IF	ID	EX	MEM	WB							
J 26 bit value is 11											IF	ID	EX	MEM	WB						
Random ADDUI instruction												IF (FLUSHED)	X	X	X	X					
Random ORI instruction												Skipped									
SB R10, 11(R11)													IF	ID	EX	MEM	WB				
SUBI R15, R4,6														IF	ID	EX	MEM	WB			
JR															IF	ID	EX	MEM	WB		
Random SUBI instruction																IF (FLUSHED)	X	X	X	X	
Random ORI instruction																Skipped					
Random ADDUI instruction																Skipped					
LH R2,12(R25)																	IF	ID	EX	MEM	WB

Figure 4: Instructions Flow

5.2) Assembly Code

Based on table 2 illustrated earlier, a specific sequence of assembly code was created before simulation. The assembly code was created in a way to have different possibilities of hazards. Also, it shows cases where we need to use previously written and stored values. Moreover, random instructions were included to be jumped over and skipped when we have control instructions. The table below shows the assembly code with detailed explanation on what is happening in each instruction.

#	Assembly Code	Explanations and Details	PC
1	LH R1, 8(R0)	<p>-R0 value in decimal is 0, and imm value is 8.</p> <p>-Loaded value to R1 is MEM [0 +8].</p> <p>-Binary value of MEM [8] is 0000000000110100000000000001100, and loading HW of this value will yield a decimal of 12.</p> <p>-This 12 (Hex' C) is written in register R1 at cycle 5.</p>	0
2	ORI R3, R1,1	<p>-R1 value from previous instruction should be 'Or'ed with imm 1, and store in R3.</p> <p>-Since here we have load data dependency hazard (RAW), a stall and forward are required. We stall because R1 value from instruction 1 will only be ready at WB. Then, we forward from WB to EX stage.</p> <p>-R3 is 12 OR 1 = 13 (Hex'D) and will be written at cycle 7</p>	4
3	SLL R4, R3, 2	<p>-Value of R3 from previous instruction (13) should be shifted by value of 2 (SA or SHAMT value in instruction) and write value in R4.</p> <p>-Here we also have data dependency, but the forwarding happens from MEM to EX stage without the need of stall.</p> <p>-So R4 = 13 SLL 2 = 52 (Hex'34) and will be written at cycle 8</p>	8
4	XNOR R7, R6, R5	<p>-R5 XNOR R6 and store in R7. Where R5 is 000000000000000000000000000011 and R6 is 000000000000000000000000000101000</p> <p>-So R7 should be 11111111111111111111111111101001 (Hex'FFFFFFE9) at cycle 9</p>	12
5	ADDUI R8, R3, 7	<p>-Value of R3 from instruction 2 is added to value of imm 7.</p> <p>-R8 value is 13+7 =20 (Hex'14) and written in cycle 10.</p>	16
6	BNE R8, R9, 1	<p>-R8 and R9 should be compared, and imm value of 1.</p> <p>-By shifting imm value by two bits to left, the value will be 4x1=4.</p> <p>-Branch address is calculated by adding PCPlus4D (24) to shifted value. So, the branch PC address is 24+4 =28.</p>	20

		<p>-A stall and forward from MEM of previous instruction to ID of this instruction is required. We stall because we detect the hazard in cycle 8 between EX and ID, but R8 value is only ready to be forwarded at MEM stage, so we waited for next cycle.</p> <p>- R8 forwarded value (20) is compared with R9 value (0). So, since they are not equal, it will enable the PC to take the new PC value of 28 and flush the instruction of PC 24</p>	
7	LH R2,12(R25)	<p>-This instruction is flushed and skipped because of BNE. We flush it because we don't need it to be executed since the branch condition is true. This flush will cause a 1 cycle delay.</p> <p>NOTE: We will jump to this instruction when we reach JR instruction coming next soon.</p>	24
8	NAND R14, R12, R13	<p>-R12 NAND R13, and write in R14. Where R12 is 00000000000000000000000011110 and R13 is 000000000000000000000000101000</p> <p>-So R14 is 11111111111111111111111110111 (Hex'FFFFFF7) at cycle 14</p>	28
9	J 26-bit value is 11	<p>-First, we shift the imm 26 bit of instruction J by 2 bits to left, and then we replace it with the first 28 bits of the value in PCPlus4D.</p> <p>- Jump address of $11 \times 4 = 44$.</p> <p>-Therefore, this jump instruction will flush and skip next 2 instructions (PC of 36 and 40)</p>	32
10	Random ADDUI instruction	Just random instruction that was included in the code of instruction memory and will be flushed and skipped due to previous J instruction	36
11	Random ORI instruction	Just random instruction that was included in the code of instruction memory and will be skipped due to previous J instruction	40
12	SB R10, 11(R11)	<p>- R10 has a value of 263, and MSB byte from it will be 7. R11 has a value of 1, and imm value is 11.</p> <p>- MEM [R11 + imm] == MEM [12].</p> <p>- MEM [12] will store value of 7 (Hex'7).</p>	44
13	SUBI R15, R4,6	<p>-Imm is 6 and R4 has a value of 52, which was calculated in instruction 3 (SLL).</p> <p>-This instruction will subtract immediate value of 6 from 52 and write the value in R15. So, R15 will have a value of $52 - 6 = 46$ (Hex'2E) at cycle 18</p>	48
14	JR R24, R0, 0	<p>-Value in R24 is 24. So, JR will insert a PC of 24.</p> <p>-PC of 24 is the same LH instruction that was skipped earlier due to BNE</p>	52
15	Random SUBI instruction	Just random instruction that was included in the code of instruction memory and will be flushed and skipped due to previous JR instruction	56
16	Random ORI instruction	Just random instruction that was included in the code of instruction memory and will be skipped due to previous JR instruction	60
17	Random ADDUI instruction	Just random instruction that was included in the code of instruction memory and will be skipped due to previous JR instruction	64

18	LH R2,12(R25)	-This LH instruction will allow us to load the value that was stored by instruction 12 (SB). -Stored value was 7 at MEM [12]. -Since R25 is 0 and imm is 12, then MEM[12], which is 7 (Hex'7'), will be loaded to R2 at cycle 21	24
----	---------------	--	----

Table 5: Assembly Code with explanations

5.SIMULATION RESULTS

We analyzed simulation result based on cycles where writing is occurring at WB stages, stall is occurring, and flush is occurring. Figure 4 and table 5 are the references to follow up with results.

Firstly, for cycles when write is happening, from figure 4 we can see that writing values should happen at cycles 5,7,8,9,10, 14,18, and 21. Therefore, in simulation results, we should have a correct value of writing and have **RegWrite_WB** control signal as high to allow writing in register block in those cycles. On the other hand, for stalling, we can see that it should happen in cycles 3 and 8, where instruction is being held for extra cycle. Lastly, flush should be requested in cycles 9, 12, and 16, but we note that zeros will appear in the cycles that are after them (so 10, 13, and 17).

Red boxes were used to mark when we have writing of values, blue boxes we used to mark stalling, and orange boxes to mark flush cycles. Results are shown in hexadecimal form. Figures below shows the simulation results.

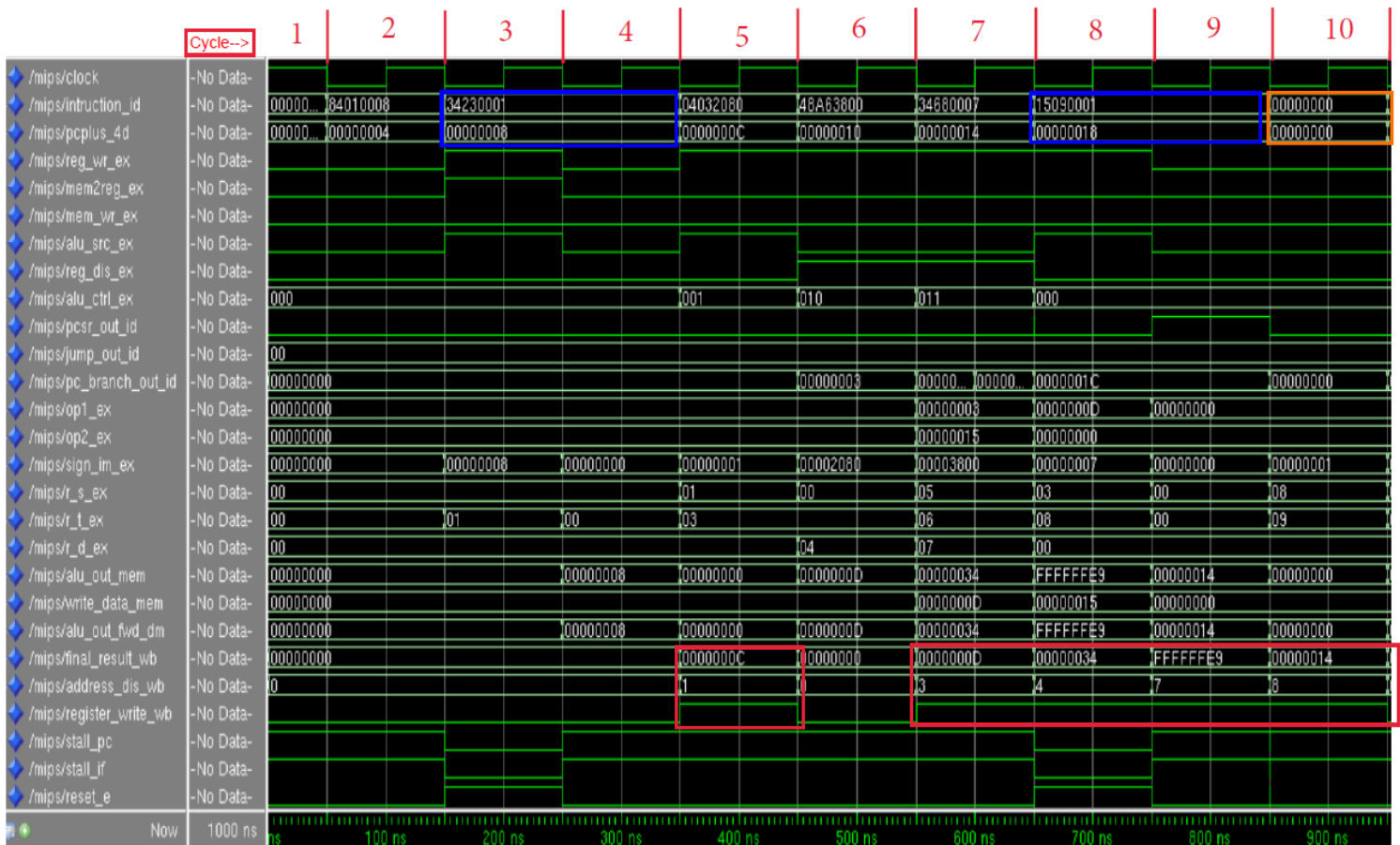


Figure 5: Simulation result part 1

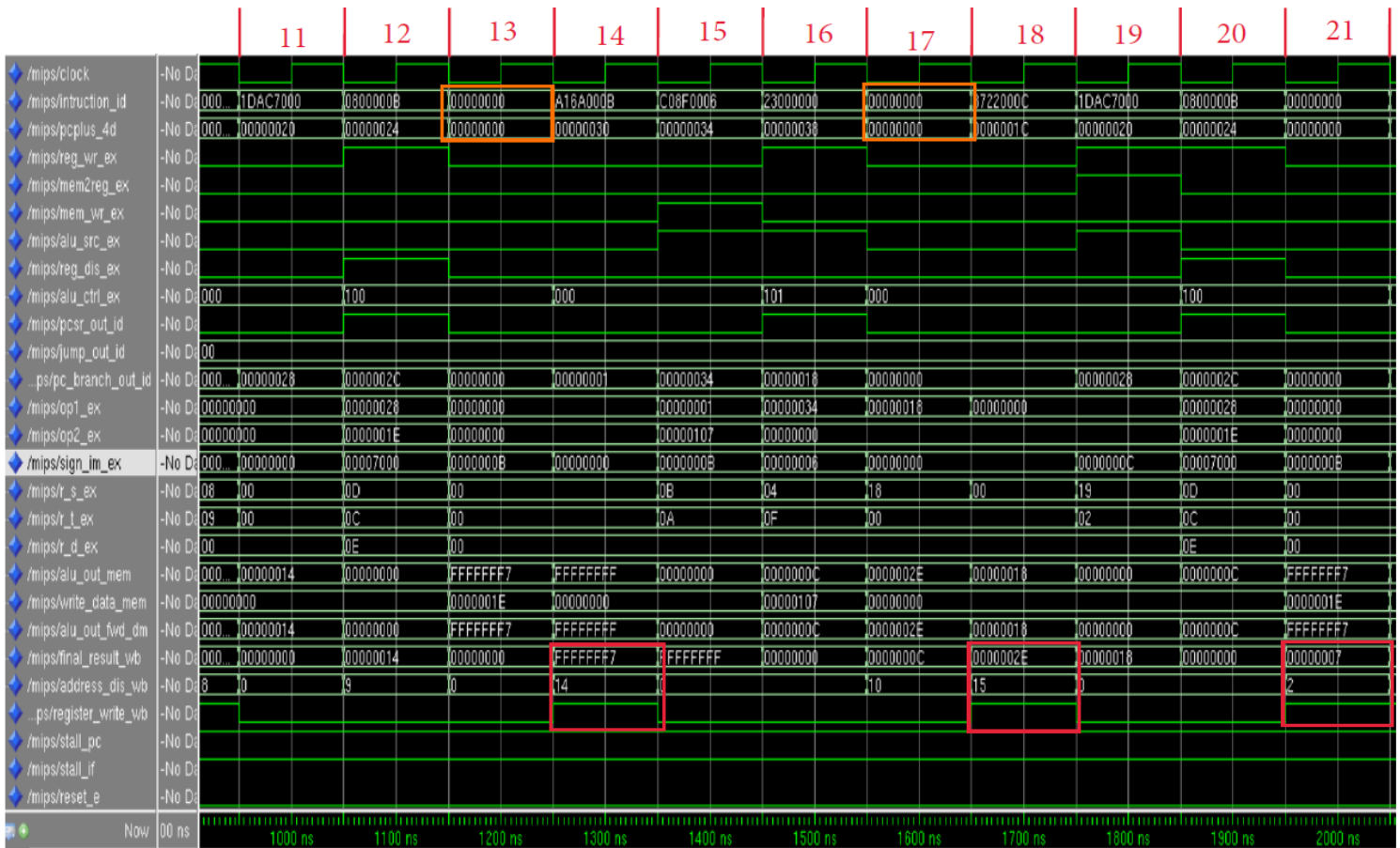


Figure 6: Simulation results part 2

From figures 5 and 6, we can see that whenever we do not need writing to register file, the control signal **RegWrite_WB** is always 0. In addition to that, when we stall, we should also flush the buffer of EX stage. So, in figure 5, we can see that when **reset_e** signal is high (cycles 3 & 8), then all signal values in EX stage in following cycle (4 & 9) are zeros.

6.CONCLUSION

In conclusion, Mini-MIPS processor was simulated using VHLD. There are mainly three types of instructions, which are R, I, and J instructions. Furthermore, the MIPS comprises of 5 stages and a hazard control unit that were pipelined to allow correct flow of parallel execution of instructions. We were given Variant 4 instructions, and they were encoded and made in a sequence to have hazards and different load, store, and arithmetic scenarios of instructions. There are different types of hazards such as structural, data, and control. Structure hazard cannot occur in our simulation since we are using separate blocks of memory for instructions and data. On the other hands, data hazard occurred and was solved by stalling and forwarding. For control hazards, the calculation of branch and jump address were executed in the ID stage to minimize the delay. Simulation result showed results and behavior as expected in the correct clock cycles.

References

- [1] D. A. P. John L Henessy, *Computer Architecture: A Quantitative Approach*, Elsevier, 1989.
- [2] M. N. T. G. D. K. S. P. Ritpurkar, "Design and simulation of 32-Bit RISC architecture based on MIPS using VHDL," in *International Conference on Advanced Computing and Communication Systems (ICACCS)*, Coimbatore, 2015.
- [3] P. a. Hennessy, "MIPS Reference Data," in *Computer Organization and Design*, Elsevier, Inc, 2009.
- [4] MIPS, "MIPS," 2021. [Online]. Available: <https://www.mips.com/products/architectures/mips32-2/>. [Accessed 01 April 2021].
- [5] A. university, "MIPS Reference Sheet," [Online]. Available: <https://uweb.engr.arizona.edu/~ece369/Resources/spim/MIPSReference.pdf>. [Accessed 1 March 2021].
- [6] U. o. Wisconsin-Milwaukee, " MIPS Instruction Code Formats," [Online]. Available: <http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s07.html>. [Accessed 1 March 2021].