Faculty of Engineering

Cairo University

Department of Electronics and Electrical Communications

Design and Implementation of an 8-bit Pipelined Processor

ELC 3030 - Advanced Processor Architecture

Student Name	Section	B.N.	I.D.
يوسف محمد عبدالعال بيومي	4	45	9231031
ديفيد سميح فارس كندس	2	09	9220287
يوسف وليد فتحي السيد	4	48	9231041
نبيل إبراهيم عبدالعاطي عبدالرازق	4	26	9230945
محمد مصطفي محمد علي إبراهيم	4	6	9230812

Table of Contents

1. Introduction

The goal of this project is to design and implement a **simple 8-bit RISC-like pipelined processor** in accordance with the given Instruction Set Architecture (ISA) specifications. The processor is described and implemented using **Verilog HDL**, with a **word size of 8 bits**.

2. Processor Specifications

a) Memory Architecture

The processor follows a **Harvard architecture** with separate instruction and data memories. Each memory has a size of 256 bytes and is byte-addressable, allowing independent access to instructions and data during execution.

The instruction memory contains the complete program code executed by the processor, including the main program instructions, interrupt service routine (ISR), and any additional subroutines or functions.

The data memory stores all runtime data used by the processor, such as variables, and stack contents. It is accessed through load and store (L-format) instructions and is also used to support stack operations during subroutine calls, returns, and interrupt handling.

b) Register Set

- Program Counter (PC) ; 8-bit program counter
- (R0–R3) ; Four 8-bit general purpose registers found in the register file block
- Stack Pointer (SP) :=R[3] ; 8-bit stack pointer
- Condition Code Register (CCR: VCNZ) ; 4-bit flags register

c) Input / Output Interface

- Input port ; 8-bit data input port
- Output port ; 8-bit data output port
- Reset pin ; reset signal
- Interrupt pin ; non-maskable interrupt single

d) Interrupt

The processor supports one external interrupt signal. When an interrupt occurs, the current program execution is suspended, and execution is redirected to the interrupt service routine (ISR). The starting address of the ISR, ISR_add, is stored in **data memory location M[1]**, while the ISR instructions themselves reside in the instruction memory.

3. Instruction Set Architecture (ISA)

a) Instruction Formats

i<7:0> = M[PC]<7:0> ; 8-bit instruction word

➢ **A**-format

- opcode<3:0> := i<7:4> ; 4-bit opcode
- ra<1:0> := i<3:2> ; 2-bit operand register and result register field
- rb<1:0> := i<1:0> ; 2-bit operand register field

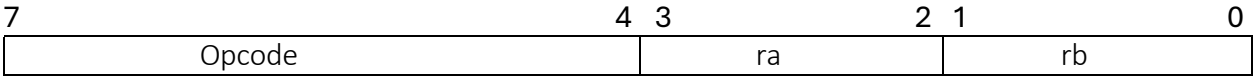


Figure 1: A-format Instructions

➢ **B**-format

- opcode<3:0> := i<7:4> ; 4-bit opcode
- ra<1:0> := i<3:2> ; 2-bit operand register and result register field
- brx<1:0> := i<3:2> ; 2-bit branch index field
- rb<1:0> := i<1:0> ; 2-bit operand register field

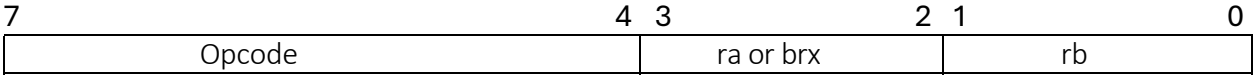


Figure 2: B-format Instructions

➢ **L**-format

- opcode<3:0> := i<7:4> ; 4-bit opcode
- ra<1:0> := i<3:2> ; 2-bit operand register and result register field
- rb<1:0> := i<1:0> ; 2-bit operand register field
- ea<7:0> := M[PC+1]<7:0> ; effective address
- imm<7:0> := M[PC+1]<7:0> ; immediate operand

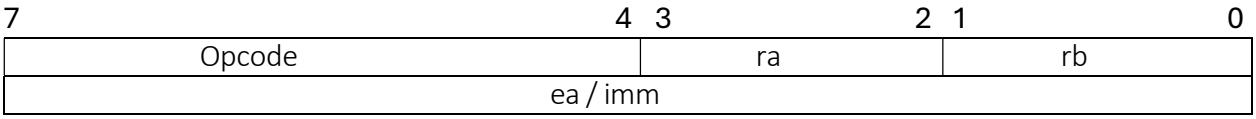


Figure 3: L-format Instructions

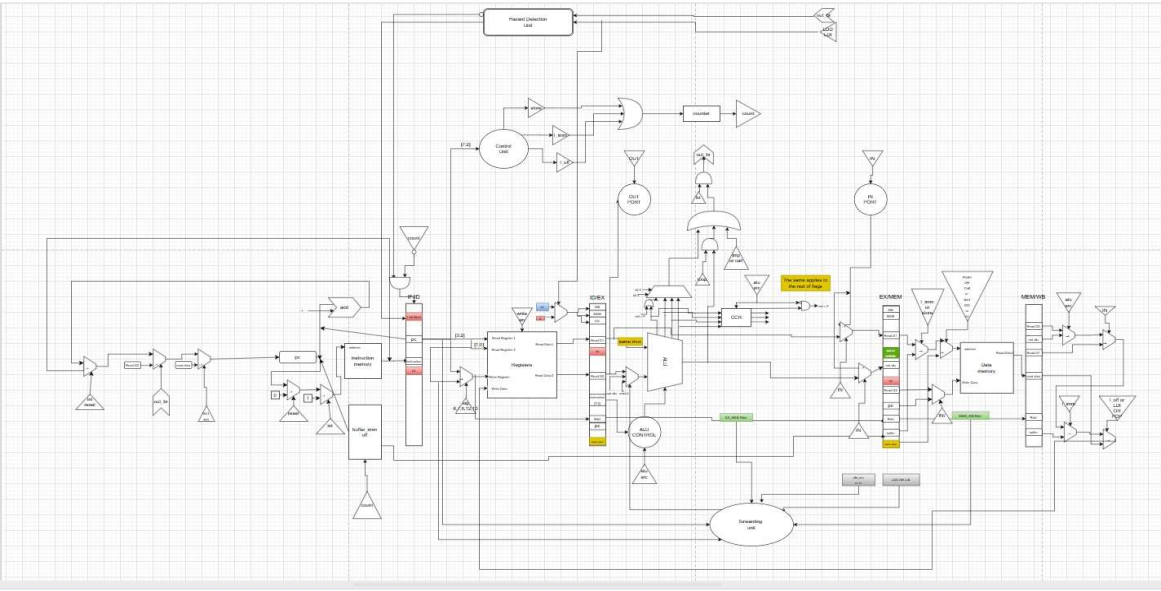
b) Implemented Instructions

	Instruction	Opcode	Length (Byte)	Function
A – Format	NOP	**0b**0000 $(0)_d$	1	PC ← PC + 1
	MOV	**0b**0001 $(1)_d$	1	**R[ra] ← R[rb];**
	ADD	**0b**0010 $(2)_d$	1	**R[ra] ←R[ra] + R[rb];** Change C, V flags , (result = 0): Z ←1; else: Z ←0; (result < 0): N ←1; else: N ←0
	SUB	**0b**0011 $(3)_d$	1	**R[ra] ←R[ra] – R[rb];** Change C, V flags , (result = 0): Z ←1; else: Z ←0 ; (result < 0): N ←1; else: N ←0
	AND	**0b**0100 $(4)_d$	1	**R[ra] ←R[ra] AND R[rb];** (result = 0): Z ←1; else: Z ←0; (result < 0): N ←1; else: N ←0
	OR	**0b**0101 $(5)_d$	1	**R[ra] ←R[ra] OR R[rb];** (result = 0): Z ←1; else: Z ←0; (result < 0): N ←1; else: N ←0
	RLC	**0b**0110 $(6)_d$	1	**(ra = 0):** C ← R[rb]<7>; R[rb] ← R[rb]<6:0> & C;
	RRC	**0b**0110 $(6)_d$	1	**(ra = 1):** C ←R[rb]<0>; R[rb] ← R[rb]<7:1> & C;
	SETC	**0b**0110 $(6)_d$	1	**(ra = 2):** C ←1;
	CLRC	**0b**0110 $(6)_d$	1	**(ra = 3):** C ←0;
	PUSH	**0b**0111 $(7)_d$	1	**(ra = 0):** X[SP--] ← R[rb];
	POP	**0b**0111 $(7)_d$	1	**(ra = 1):** R[rb] ← X[++SP];
	OUT	**0b**0111 $(7)_d$	1	**(ra = 2):** OUT.PORT ← R[rb];
	IN	**0b**0111 $(7)_d$	1	**(ra = 3):** R[rb] ←IN.PORT;
	NOT	**0b**1000 $(8)_d$	1	**(ra = 0):** R[rb] ← 1's Complement (R[rb]); (result = 0): Z ←1; else: Z ←0; (result < 0): N ←1; else: N ←0
	NEG	**0b**1000 $(8)_d$	1	**(ra = 1):** R[rb] ← 2's Complement (R[rb]); (result = 0): Z ←1; else: Z ←0; (result < 0): N ←1; else: N ←0
	INC	**0b**1000 $(8)_d$	1	**(ra = 2):** R[rb] ←R[rb] + 1; Change C, V flags , (result = 0): Z ←1; else: Z ←0; (result < 0): N ←1; else: N ←0
	DEC	**0b**1000 $(8)_d$	1	**(ra = 3):** R[rb] ←R[rb] - 1; Change C, V flags , (result = 0): Z ←1; else: Z ←0; (result < 0): N ←1; else: N ←0

B – Format	JZ	**0b**1001 $(9)_d$	1	**(brx = 0):** Z=1: PC ←R[rb]; Z=0: PC ← PC + 1
	JN	**0b**1001 $(9)_d$	1	**(brx = 1):** N=1: PC ←R[rb]; N=0: PC ← PC + 1
	JC	**0b**1001 $(9)_d$	1	**(brx = 2):** C=1: PC ←R[rb]; C=0: PC ← PC + 1
	JV	**0b**1001 $(9)_d$	1	**(brx = 3):** V=1: PC ←R[rb]; V=0: PC ← PC + 1
	LOOP	**0b**1010 $(10)_d$	1	R[ra] ←R[ra] – 1; (result != 0): PC ←R[rb]; (result = 0): PC ← PC + 1
	JMP	**0b**1011 $(11)_d$	1	**(brx=0):** PC ←R[rb]
	CALL	**0b**1011 $(11)_d$	1	**(brx=1):** X[SP--] ← PC + 1; PC ← R[rb]
	RET	**0b**1011 $(11)_d$	1	**(brx=2):** PC ← X[++SP]
	RTI	**0b**1011 $(11)_d$	1	**(brx=3):** PC ← X[++SP]; Flags restored
L – Format	LDM	**0b**1100 $(12)_d$	2	**(ra = 0):** R[rb] ← imm; PC ← PC + 2
	LDD	**0b**1100 $(12)_d$	2	**(ra = 1):** R[rb] ← M[ea]; PC ← PC + 2
	STD	**0b**1100 $(12)_d$	2	**(ra = 2):** M[ea] ←R[rb]; PC ← PC + 2
	LDI	**0b**1101 $(13)_d$	1	R[rb] ← M[R[ra]]; PC ← PC + 1
	STI	**0b**1110 $(14)_d$	1	M[R[ra]] ←R[rb]; PC ← PC + 1

4. Processor Architecture

a) Block Diagram



b) Module Hierarchy

The processor consists of several key modules that work together to implement the pipeline architecture. Each module is designed to handle a specific stage or function of the processor.

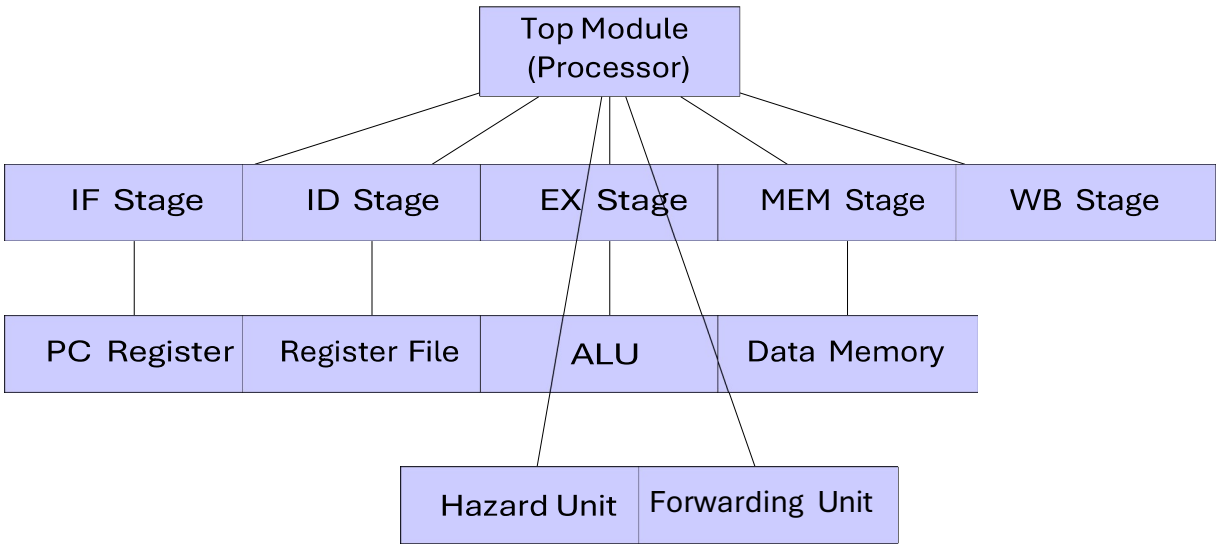


Figure 5.1: Module Hierarchy

Program Counter Module

The program counter keeps track of the current instruction address and updates based on control signals from branches, jumps, and sequential execution.

Register File Module

The register file contains four 8-bit general-purpose registers (R0-R3), with R3 serving dual purpose as the stack pointer initialized to 255.

ALU Module

The Arithmetic Logic Unit performs all arithmetic, logical, and shift operations as speci- fied by the instruction set.

Hazard Detection Unit

This unit monitors pipeline stages to detect data hazards and control hazards, generating appropriate stall and flush signals.

Forwarding Unit

The forwarding unit implements data bypassing to resolve hazards by forwarding results from later pipeline stages back to earlier stages when needed.

c) Pipeline Architecture

The processor implements a classic five-stage pipeline architecture:

1. **Instruction Fetch (IF):** Fetches instruction from memory using PC
2. **Instruction Decode (ID):** Decodes instruction and reads register operands
3. **Execute (EX):** Performs ALU operations or address calculations
4. **Memory (MEM):** Accesses data memory for load/store operations
5. **Write Back (WB):** Writes results back to register file

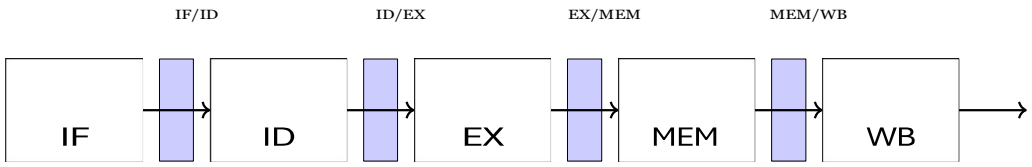


Figure 3.1: Five-Stage Pipeline Architecture

d) Datapath Design

1. **Program Counter (PC):** Maintains current instruction address
2. **Instruction Memory:** Stores program instructions
3. **Register File:** Contains four 8-bit general-purpose registers
4. **ALU (Arithmetic Logic Unit):** Performs arithmetic and logical operations
5. **Data Memory:** Stores program data
6. **Control Unit:** Generates control signals for datapath
7. **Hazard Detection Unit:** Detects pipeline hazards
8. **Forwarding Unit:** Implements data forwarding to resolve hazards
9. **Pipeline Registers:** Store intermediate values between stages

5. Control Unit Design (FSM)

The control unit is responsible for decoding the instruction opcode and generating the appropriate control signals required for correct processor operation. It coordinates data flow between the ALU, registers, memory, and stack by enabling and configuring each component based on the current instruction.

Control Signals

- reg_write**: Enables writing data into a register.**
- mem_read**: Controls read operations from memory.**
- mem_write**: Controls write operations to memory.**
- mem_to_reg**: Selects memory data to be written back into a register.**
- alu_src**: Selects the ALU source operand (0 = register, 1 = immediate).**
- is_branch**: Indicates a conditional branch instruction.**
- is_jump**: Indicates an unconditional jump, call, or return instruction.**
- is_two_byte**: Indicates that the instruction occupies two bytes.**
- update_flags**: Enables updating of the condition flags.**
- is_stack_op**: Indicates a stack-related operation (PUSH, POP, CALL, RET).**
- sp_increment**: Increments the stack pointer (used in POP and RET).**
- sp_decrement**: Decrements the stack pointer (used in PUSH and CALL).**
- alu_op**: Selects the operation performed by the ALU.**

6. Hazard detection unit

The hazard detection unit monitors instructions in the **IF/ID** and **ID/EX** pipeline stages to identify data and control hazards that could lead to incorrect execution. It specifically detects **EX–ID read-after-write (RAW) hazards**, ensuring that if an instruction in the execute stage writes to a register that the following instruction in the decode stage needs to read, the pipeline is stalled to prevent using stale data.

In addition, the unit handles **load-use hazards** and **consecutive stack operation hazards** by asserting a stall whenever an instruction depends on data that has not yet been written back or when back-to-back stack operations could corrupt the stack pointer or memory. By generating the **stall** signal whenever any of these hazards are detected, the unit guarantees correct data flow and reliable pipeline operation.

7. Interrupt Handling

Interrupt Mechanism

The processor supports a single non-maskable interrupt with the following characteristics:

1. **Interrupt Detection:** Rising edge on INTR.IN signal
2. **Context Saving:**
   - PC is pushed onto stack: Stack[SP–] ← PC
   - Flags are preserved in shadow registers
3. **Vector Jump:** PC ← M[1] (load interrupt vector)
4. **Interrupt Service:** Execute interrupt service routine
5. **Return:** RTI instruction restores PC and flags

Interrupt Handling FSM

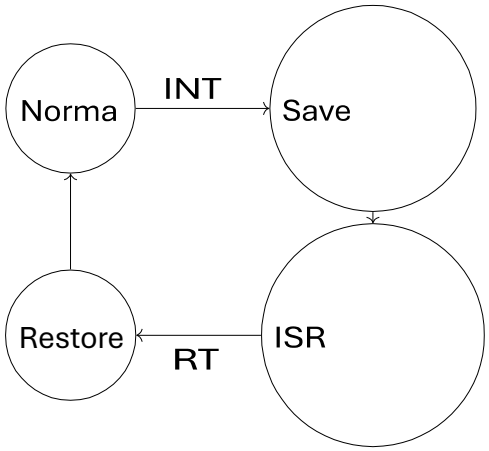


Figure 6.1: Interrupt Handling State Machine

Implementation Code

The interrupt controller is implemented as a finite state machine that handles the interrupt sequence: detecting the interrupt signal, saving context to the stack, jumping to the interrupt service routine, and restoring context on return.

8. Verification and Testing

Test Strategy

The processor was verified through comprehensive testing using five main test scenarios. Each scenario was designed to test specific functionality and ensure correct operation of the processor.

Test Scenarios

Test Scenario 1: ALU Operations

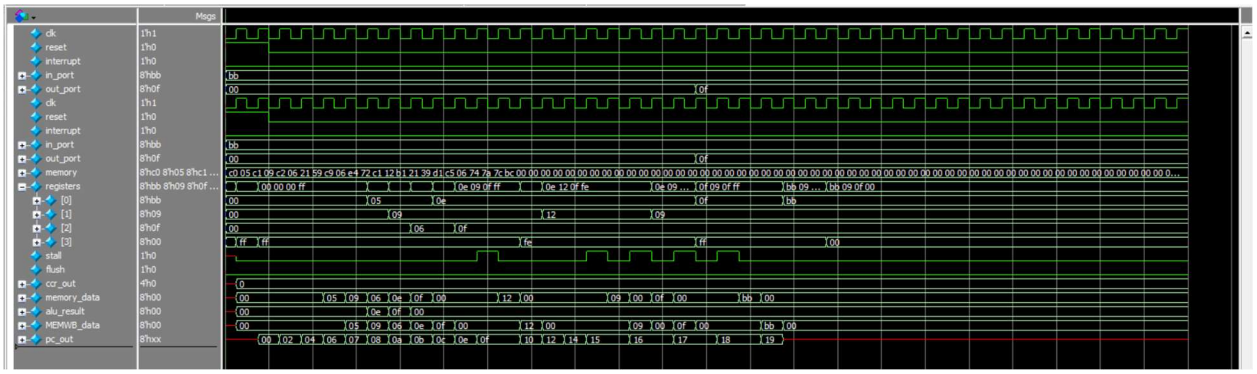
This test verifies all arithmetic and logical operations including ADD, SUB, AND, OR, NOT, NEG, INC, and DEC instructions. The test checks correct result computation and proper flag updates (Z, N, C, V flags).

Test Coverage:

- Addition with and without carry
- Subtraction with borrow detection
- Logical AND and OR operations
- Bitwise NOT and two's complement NEG
- Increment and decrement with flag updates
- Zero flag detection
- Negative flag detection
- Carry and overflow flag computation

Result: All ALU operations executed correctly with proper flag updates.

```
1         uut.imem_inst.memory[00] = 8'hC0;  // LDM R0, 5
2         uut.imem_inst.memory[01] = 8'h05;
3         uut.imem_inst.memory[02] = 8'hC1;  // LDM R1, 9
4         uut.imem_inst.memory[03] = 8'h09;
5         uut.imem_inst.memory[04] = 8'hC2;  // LDM R2, 6
6         uut.imem_inst.memory[05] = 8'h06;
7         uut.imem_inst.memory[06] = 8'h21;  //ADD R0,R1
8         uut.imem_inst.memory[07] = 8'h59;  // OR R2,R1
9         uut.imem_inst.memory[08] = 8'hC9;  //STD , R1 AT 6
10        uut.imem_inst.memory[09] = 8'h06;
11        uut.imem_inst.memory[10] = 8'hE4;  //STI R0 AT R1
12        uut.imem_inst.memory[11] = 8'h72;  //PUSH R2
13        uut.imem_inst.memory[12] = 8'hC1;  // LDM R1, 12
14        uut.imem_inst.memory[13] = 8'h12;
15        uut.imem_inst.memory[14] = 8'hB1;  //JMP R1
16        uut.imem_inst.memory[15] = 8'h21;  //ADD R0,R1
17        uut.imem_inst.memory[16] = 8'h39;  //SUB R2,R1
18        uut.imem_inst.memory[17] = 8'hD1;  //LDI
19        uut.imem_inst.memory[18] = 8'hc5;  //LDD R1 FROM 6
20        uut.imem_inst.memory[19] = 8'h06;
21        uut.imem_inst.memory[20] = 8'h74;  //POP R0
22        uut.imem_inst.memory[21] = 8'h7A;  //OUT R2
23        uut.imem_inst.memory[22] = 8'h7C;  //IN R0
24        uut.imem_inst.memory[23] = 8'hBC;  //RET
```



The instruction memory is initialized as follows. First, **R0** is loaded with the value **5**, **R1** with **9**, and **R2** with **6**. An **ADD** instruction then adds the contents of **R1** to **R0**, followed by an **OR** operation between **R2** and **R1** stored in **R2**. Next, the value of **R1** is stored directly in memory location **6**, and the value of **R0** is stored indirectly at the memory address contained in **R1**.

After that, **R2** is pushed onto the stack. **R1** is then loaded with the value **12**, and a **JMP** instruction transfers program execution to the address contained in **R1 (address 18).**

The program then loads **R1** from memory location **6**, pops the top of the stack into **R0**, outputs the value of **R2**, reads input data into **R0**, and finally executes a **RET** instruction to return from the subroutine.

Test Scenario 2: Memory Store and Load

This test verifies all memory access instructions including LDM, LDD, STD, LDI, and STI. Both direct and indirect addressing modes were tested.

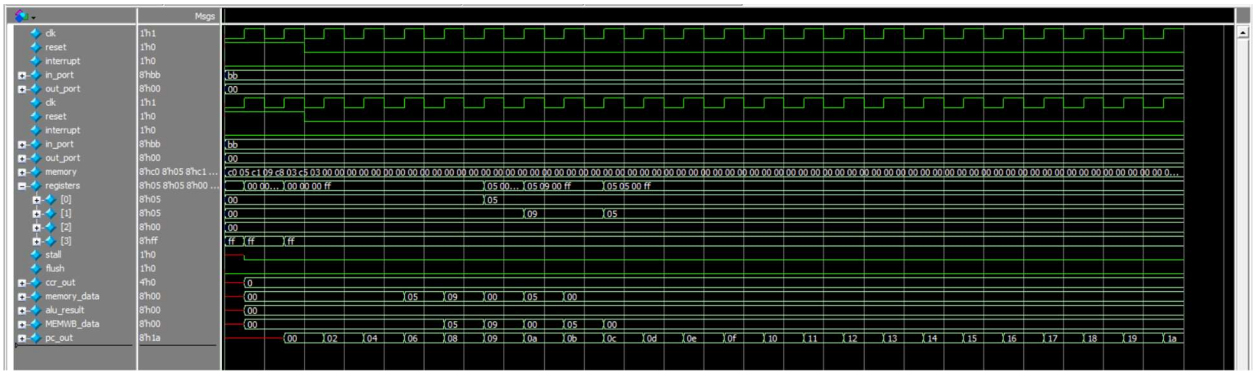
Test Coverage:

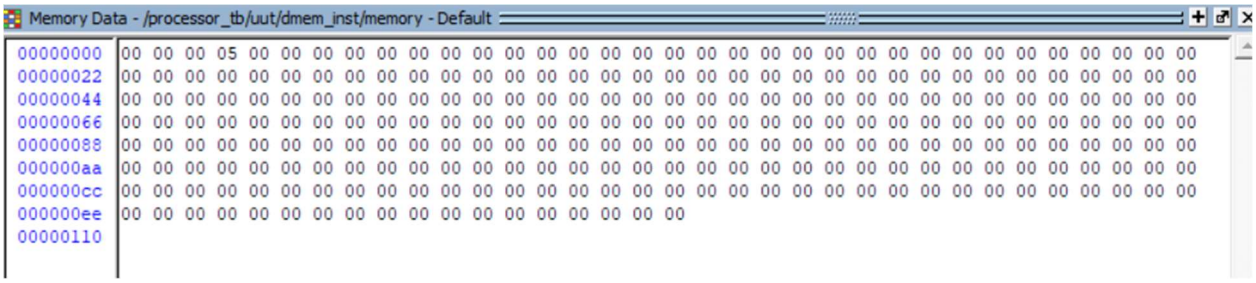
- Load immediate value (LDM)
- Direct load from memory (LDD)
- Direct store to memory (STD)
- Indirect load using register pointer (LDI)
- Indirect store using register pointer (STI)
- Memory address calculation
- Data integrity verification

Result:

memory.txt

```
1        uut.imem_inst.memory[0] = 8'hC0;  // LDM R0, 5
2        uut.imem_inst.memory[1] = 8'h05;
3        uut.imem_inst.memory[2] = 8'hC1;  // LDM R1, 9
4        uut.imem_inst.memory[3] = 8'h09;
5        uut.imem_inst.memory[4] = 8'hC8;  //STD R0, 3
6        uut.imem_inst.memory[5] = 8'h03;
7        uut.imem_inst.memory[6] = 8'hC5;   //LDD R1,3
8        uut.imem_inst.memory[7] = 8'h03;
9        uut.imem_inst.memory[8] = 8'h00;  // NOP
```





As expected, **R0** takes the value **5** and **R1** takes the value **9**. The third memory location then stores the value **in R0** , after which **R1** is updated to hold the value stored in the third memory location

Test Scenario 3: Input/Output Ports

This test verifies the IN and OUT instructions for communication with external devices through input and output ports.

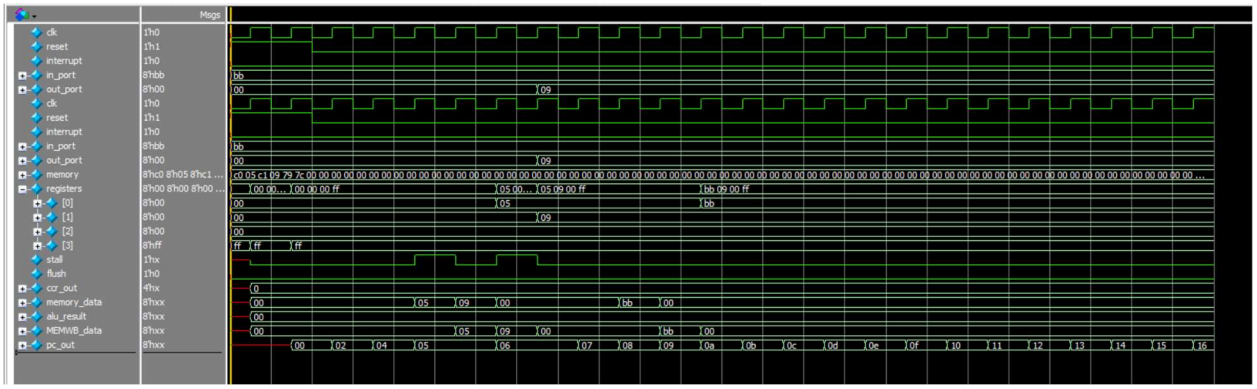
Test Coverage:

- Reading data from input port (IN)
- Writing data to output port (OUT)
- Port timing and synchronization
- Data transfer accuracy

Result: I/O operations functioned correctly with proper data transfer between pro- cessor and external ports.

in_out_port.txt

```
1        uut.imem_inst.memory[0] = 8'hC0;  // LDM R0, 5
2        uut.imem_inst.memory[1] = 8'h05;
3        uut.imem_inst.memory[2] = 8'hC1;  // LDM R1, 9
4        uut.imem_inst.memory[3] = 8'h09;
5        uut.imem_inst.memory[5] = 8'h79;  // outport R1
6        uut.imem_inst.memory[6] = 8'h7c;  // IN_PORT R0
7        uut.imem_inst.memory[8] = 8'h00;  // NOP
```



As expected, **R0** is loaded with **5** and **R1** is loaded with **9**. The **out_port** then takes the value of **R1**, after which **R0** is loaded with the data from the **input port**.

Test Scenario 4: Data Forwarding

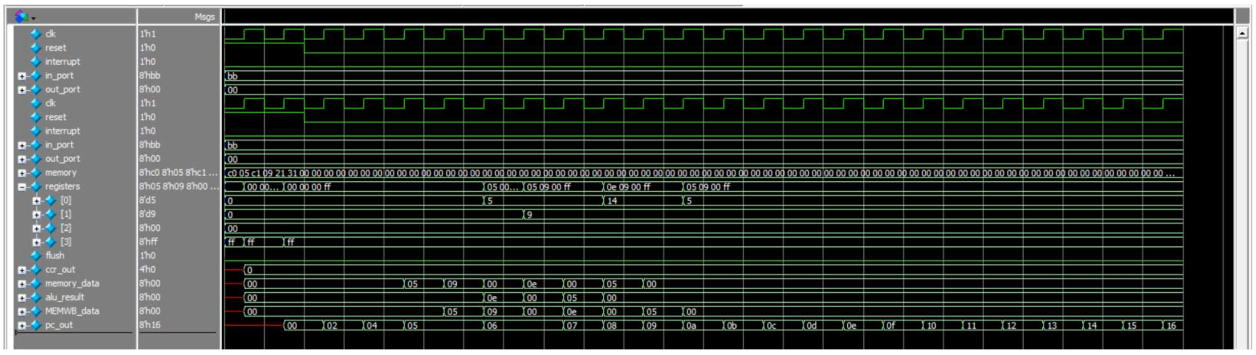
This test verifies the forwarding unit functionality by creating data hazard situations and ensuring correct data forwarding from EX and MEM stages.

Test Coverage:

- RAW (Read After Write) hazards
- Forwarding from EX/MEM stage
- Forwarding from MEM/WB stage
- Priority handling when multiple forwarding paths exist
- Pipeline stall avoidance through forwarding

Result: Forwarding unit successfully resolved data hazards without requiring pipeline stalls in most cases.

```
1        uut.imem_inst.memory[0] = 8'hC0;  // LDM R0, 5
2        uut.imem_inst.memory[1] = 8'h05;
3        uut.imem_inst.memory[2] = 8'hC1;  // LDM R1, 9
4        uut.imem_inst.memory[3] = 8'h09;
5        uut.imem_inst.memory[4] = 8'h21;  // ADD R0, R1
6        uut.imem_inst.memory[5] = 8'h31;  // sub R0, R1
7        uut.imem_inst.memory[6] = 8'h00;  // NOP
```



as expected **R0** is first loaded with the value **5**, followed by loading **R1** with the value **9**. An **ADD** instruction is then executed to add the contents of **R1** to **R0**, after which a **SUB** instruction subtracts the contents of **R1** from **R0**. Finally, a **NOP** instruction is executed, performing no operation , showing he work of forwarding unit

Test Scenario 5: Branching and Hazard Detection

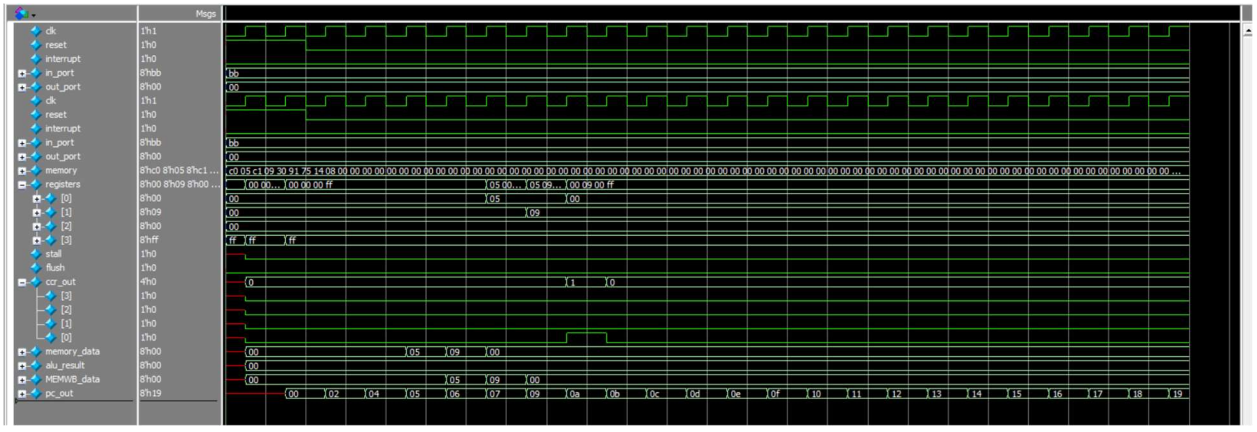
This test verifies branch instructions (JZ, JN, JC, JV, JMP, CALL, RET) and the hazard detection unit's ability to detect and resolve control hazards and load-use hazards.

Test Coverage:

- Conditional branches with flag testing
- Unconditional jump (JMP)
- Subroutine call and return (CALL, RET)
- Loop instruction (LOOP)
- Branch taken and not taken scenarios
- Pipeline flush on branch
- Load-use hazard detection
- Pipeline stall insertion when necessary

Result: All branch instructions executed correctly, and hazard detection unit properly identified and resolved hazards through stalls and flushes.

```
1        uut.imem_inst.memory[0] = 8'hC0;  // LDM R0, 5
2        uut.imem_inst.memory[1] = 8'h05;
3        uut.imem_inst.memory[2] = 8'hC1;  // LDM R1, 9
4        uut.imem_inst.memory[3] = 8'h09;
5        uut.imem_inst.memory[4] = 8'h30;  // sub R0, R0
6        uut.imem_inst.memory[5] = 8'h91;  // JZ R1
7        uut.imem_inst.memory[6] = 8'h75;  // out R0
8        uut.imem_inst.memory[7] = 8'h14;  // Mov R0,R1
9        uut.imem_inst.memory[8] = 8'h08;
10       uut.imem_inst.memory[9] = 8'h00;  // NOP
```



The instruction memory is initialized by first loading **R0** with the value **5**, followed by loading **R1** with the value **9**. A **SUB** instruction is then executed to subtract **R0** from itself, resulting in zero. Since the zero flag is set, a **JZ** instruction is evaluated using **R1**. The program flush all next instruction and pc jump to value of **R1 (9)**

9. Synthesis

Vivado is used to verify that the processor design is fully synthesizable. During synthesis, the HDL code is analyzed to ensure that all modules are correctly described, free of syntax or structural errors, and suitable for implementation on FPGA hardware. Successful synthesis confirms that the processor architecture and control logic can be reliably mapped to physical resources.