Faculty of Engineering & Technology Electrical & Computer Engineering Department

COMPUTER ARCHITECTURE-ENCS4370

Project #2

Multi-Cycle RISC Processor

Prepared by:

Name:	Francis Miadi	**ID:**	1210100	**Section:**	1
Name:	Miar Taweel	**ID:**	1210447	**Section:**	1
Name:	Leena Abuhammad	**ID:**	1211460	**Section:**	1

Instructor: Dr. Ayman Hroub

Hand in Date: 14/01/2025

Abstract

This project presents the design and verification of a 16-bit multicycle RISC processor. The processor has a 16-bit program counter (PC), eight 16-bit general-purpose registers, a 16-bit return register (RR) for function calls, and Performance registers that keep track on various program execution metrics. This design uses Word-addressable memories to store data and instructions in two distinct physical memories, supporting three different instruction types: R-, I-, and J-type. Each instruction is carried out by the multicycle architecture in several steps: fetch, decode, execute, access memory, and write back. To ensure the datapath and control path functionality, boolean equations and control signals were created. A testbench for the primary structural module and waveform simulations for every module were used for verification in order to confirm their functionality. Active-HDL was used for RTL design, while Canva was employed for drawing the datapath. This report provides insights into design, implementation, and testing, supported by diagrams and test cases.

Partners Contribution

The Table below shows the contribution of each team member in the project (Note: all of us contributed in implementing separate Functional units that were used in the structural Module):

Name	Contribution
Francis	**Full Micro-Architecture diagram Design (Data and Control paths and Signals) + Report**
Miar	**Full design verification, Debugging, fixing, and testing (testBench) + Report**
Leena	**Structural Module implementation + FSM Diagram + Report**

Table of Contents

Table of Figures

List of Tables

1. Theory

Multi-cycle data path break up instructions into separate steps. It reduces average instruction time. Each step takes a single clock cycle each functional unit can be used more than once in an instruction, as long as it is used in different clock cycles. It reduces the amount of hardware needed. [1]

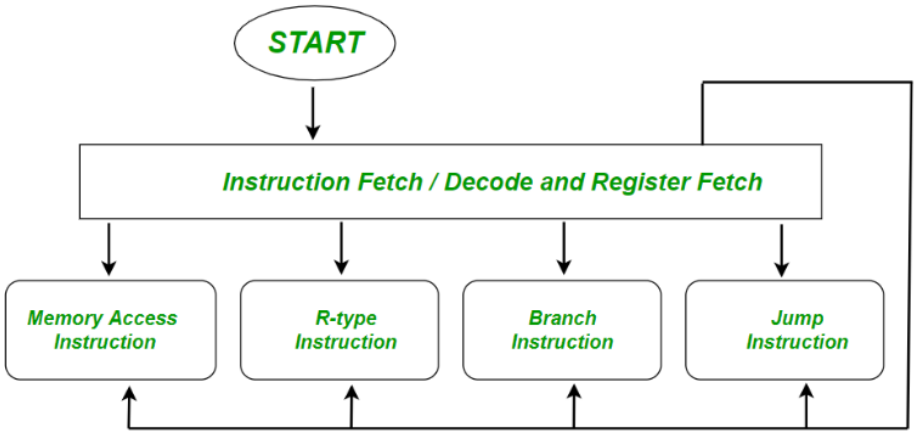


Figure 1.1 – Multi-cycle Processor flow chart [1]

The five stages of the multicycle processor are:

1. **Fetch Instruction:** Retrieves the instruction from the instruction memory using the address stored in the Program Counter (PC).
2. **Decode:** Decodes the instruction to determine the required operations and fetches data and operands from registers.
3. **Execution:** Executes arithmetic, computes memory addresses.
4. **Memory Access:** Accesses data memory for load/store instructions.
5. **Write Back:** Writes results into registers for instructions that have a destination register.

The stages above are separated by clocked registers, to store each stage's result. [1]

2. Pre-implementation

Before starting implementing the data control paths and the RTL, a plan has to be made to ensure the correctness of each implemented phase in the future. This plan is the Pre-implementation phase, it includes studying the Instruction types and their format, Instructions Encoding, determining the required combinational and sequential functional units, analyze each unit and determine its inputs, outputs, and whether there is a clock demand or not in it.

After that, control signals for each instruction were determined, and Boolean expressions for each signal were formulated. This ensured accurate control logic to guide and control a route for the data to flow through the data path components during execution to generate correct results.

2.1 Instruction types and format

2.1.1 R-Type (Register type)

$Opcode^4$	Rd^3	Rs^3	Rt^3	$Function^3$

- **Opcode: 4-bit opcode**
- **Rd: 3-bit destination register**
- **Rs: 3-bit first source register**
- **Rt: 3-bit second register**
- **Function: 3-bit operand to differentiate between instructions with the same opcode**

2.1.2 I-Type (Immediate type)

$Opcode^4$	Rs^3	Rt^3	$Immediate^6$

- **Opcode: 4-bit opcode**
- **Rs: 3-bit source register**
- **Rt: 3-bit destination register**
- ***immediate: 6-bit operand, its value is zero-extended for logical instructions and sign-extended for all other instructions***

2.1.3 J-Type (Jump Type)

$Opcode^4$	$Offset^9$	$Function^3$

- **Opcode: 4-bit opcode**
- **Offset: 9-bit operand for jump address calculation**
- **Function: 3-bit operand to differentiate between instructions with the same opcode**

2.2 Instructions Encoding

The implemented processor supports the following subset of instructions, in order to determine the path that each instruction should go through, and what stages it will enter, each instruction has to be encoded to exactly determine its operands and what route it should take.

2.2.1 R-Type (Register Type)

Instruction	***Meaning***	***Opcode***	***Function***
AND Rd, Rs, Rt	Reg(Rd) = Reg(Rs) & Reg(Rt)	0000	000
ADD Rd, Rs, Rt	Reg(Rd) = Reg(Rs) + Reg(Rt)	0000	001
SUB Rd, Rs, Rt	Reg(Rd) = Reg(Rs) - Reg(Rt)	0000	010
SLL Rd, Rs, Rt	Reg(Rd) = Reg(Rs) << Reg(Rt)	0000	011
SRL Rd, Rs, Rt	Reg(Rd) = Reg(Rs) >> Reg(Rt)	0000	100

- **IF: Instruction ← Instruction_Mem[PC] , pc = pc+1**
- **ID: Reg_A ← Reg(Rs), Reg_B ← Reg(Rt)**
- **EX: ALU_output ← OPCODE (Reg_A, Reg_B)**
- **WB: Reg(Rd) ← ALU_output**

2.2.2 I-Type (Immediate Type)

Instruction	*Meaning*	*Opcode*	*Function*
ANDI Rt, Rs, Imm	Reg(Rt) = Reg(Rs) & zero_ext(Imm)	0010	NA
ADDI Rt, Rs, Imm	Reg(Rt) = Reg(Rs) + zero_ext(Imm)	0011	NA

- **IF: Instruction ← Instruction_Mem[PC], pc = pc+1**
- **ID: Reg_A ← Reg(Rs), Reg_B ← zero_ext(Imm)**
- **EX: ALU_output ← OPCODE (Reg_A, Reg_B)**
- **WB: Reg(Rd) ← ALU_output**

Instruction	*Meaning*	*Opcode*	*Function*
LW Rt, Imm(Rs)	Reg(Rt) = Mem(Reg(Rs) + sign_ext (Imm))	0100	NA

- **IF: Instruction ← Instruction_Mem[PC], pc = pc+1**
- **ID: Reg_A ← Reg(Rs), Reg_B ← sign_ext (Imm)**
- **EX: ALU_output ← ADD (Reg_A, Reg_B)**
- **Mem: Data_out ← Data_Mem[ALU_output]**
- **WB: Reg(Rt) ← Data_out**

Instruction	*Meaning*	*Opcode*	*Function*
SW Rt, Imm(Rs)	Mem(Reg(Rs) + sign_ext (Imm)) = Reg(Rt)	0101	NA

- **IF: Instruction ← Instruction_Mem[PC], pc = pc+1**
- **ID: Reg_A ← Reg(Rs), Reg_B ← sign_ext (Imm)**
- **EX: ALU_output ← ADD (Reg_A, Reg_B)**
- **Mem: Data_Mem [ALU_output] ← Reg(Rt)**

Instruction	*Meaning*	*Opcode*	*Function*
BEQ Rs, Rt, Imm	if (Reg(Rs) == Reg(Rt)) Next PC = branch target else Next PC = PC + 1	0110	NA

- **IF: Instruction ← Instruction_Mem[PC]**
- **ID: Reg_A ← Reg(Rs), Reg_B ← Reg(Rt)**
- **EX: Zero← subtract (Reg_A, Reg_B)**
- **if (Zero == 1)**
  PC = PC + sign_ext (Imm)

  else PC = PC + 1

Instruction	*Meaning*	*Opcode*	*Function*
BNE Rs, Rt, Imm	if (Reg(Rs) != Reg(Rt)) Next PC = branch target else Next PC = PC + 1	0111	NA

- **IF: Instruction ← Instruction_Mem[PC]**
- **ID: Reg_A ← Reg(Rs), Reg_B ← Reg(Rt)**
- **EX: Zero← subtract (Reg_A, Reg_B)**
- **if (Zero == 0)**
  PC = PC + sign_ext (Imm)

  else PC = PC + 1

Instruction	*Meaning*	*Opcode*	*Function*
FOR Rs, Rt	if (Reg(Rt) != 0) Next PC = Reg(Rs) else Next PC = PC + 1	1000	NA

- **IF: Instruction ← Instruction_Mem[PC]**
- **ID: PC ← Reg(Rs), iterations ← Reg(Rt)**

2.2.3 J-Type (Jump Type)

Instruction	*Meaning*	*Opcode*	*Function*
JMP Offset	Next PC = jump target	0001	000

- **IF: Instruction ← Instruction_Mem[PC]**
- **ID: PC ← PC [15:9] || 9-bit offset**

Instruction	*Meaning*	*Opcode*	*Function*
CALL Offset	Next PC = jump target PC + 1 is stored on the RR	0001	001

- **IF: Instruction ← Instruction_Mem[PC]**
- **ID: PC ← PC [15:9] || 9-bit offset, RR ← PC + 1**

Instruction	*Meaning*	*Opcode*	*Function*
RET	Next PC = value of the RR	0001	010

- **IF: Instruction ← Instruction_Mem[PC]**
- **ID: PC ← RR**

The Concept of Multi-Cycle Processor implies that different instruction types require different number of stages as shown in the instructions encoding above, which is also illustrated in the table below:

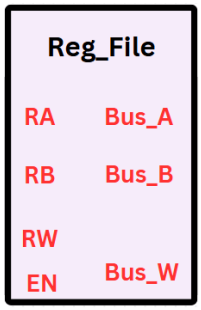
Instruction	Number of stages	Stages
ALU	**4**	**IF,ID,EX,WB**
Load	**5**	**IF,ID,EX,MEM,WB**
Store	**4**	**IF,ID,EX,MEM**
Branch	**3**	**IF,ID,EX**
Jump	**2**	**IF,ID**

Table 2.2.1 – Instructions and Stages

2.3 Functional Units

After analyzing the instruction types, formats, and encoding, it was concluded that functional units are required to perform the operations defined in the instruction set. Additionally, registers and memories are needed to store instructions, data, and results. To achieve this, the following components were identified as essential.

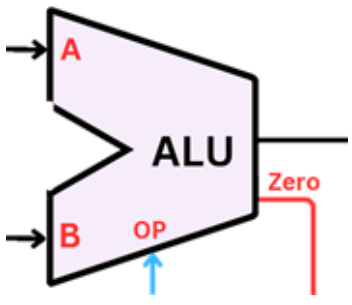
2.3.1 Register File



There are eight 16-bit general-purpose registers in the Register File. It can read from two registers and write to one register at the same time since it has two read busses (A and B) and one write bus. It has five inputs and two outputs, RA and RB are inputs that receive the addresses of the registers that their data is needed and transferred on Bus_A and Bus_B respectively as outputs, RW is the input that takes the address of the register that its data is wanted to be written during the Write Back (WB) stage, in which the control unit generates a write enable signal (EN) that controls whether or not data is written to the register file, if it is enabled, then the data from the input Bus_W will be written on the register with the address RW.

Figure 2.3.1.1 – Register File

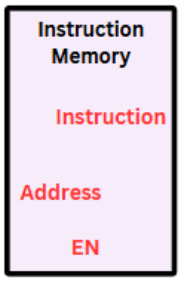
2.3.2 ALU



The ALU has 3 inputs and two outputs, it receives inputs A and B, to perform the operation that is determined by the input (OP) which is generated by the ALU control unit, This ALU provides 5 operations: ADD, AND, SUB, SLL, and SRL. When the operation is done, two outputs will be generated, the result of the operation and the ZERO flag which will be 1 if the result of the operation equals to zero, otherwise it will be 0.

Figure 2.3.2.1 – ALU

2.3.3 Memories

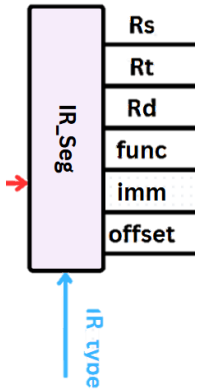


The program instructions that the processor will execute are stored in the instruction memory. This memory has two inputs and one output, it receives an Address as an input which is the value of the PC, if the enable (En) is active then the correct instruction will be fetched from the Instruction Memory during the Instruction Fetch (IF) stage.

The data memory is used to store and retrieve data required during program execution. This memory has four inputs and one output. It receives an Address input to specify the memory location, a Data_in input for storing values, and control signals (W_EN for write enable and R_EN for read enable) to choose the operating mode. If R_EN is active, the data at the input address will be the output of the memory through Data_out. If W_EN is active, the value provided at Data_in will be written to the specified address. This process occurs during the Memory Access (MEM) stage.

Figure 2.3.3.1 – Instruction and Data Memories

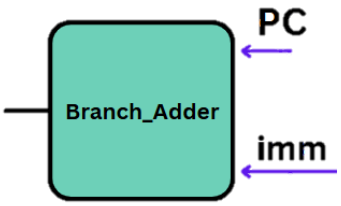
2.3.4 Instruction Segmentation



The Instruction Segmentation (IR_Seg) unit divides its input into segments of variant sizes based on the instruction type input (IR_Type). For IR_Type = 0 (R_Type), it segments the instruction (excluding the opcode) as follows: Rd (first 3 bits), Rs (second 3 bits), Rt (third 3 bits), and Function (last 3 bits), with imm = 0 and offset = 0. Similarly, it segments I_Type and J_Type instructions when IR_Type = 1 or 2, respectively.

Figure 2.3.4.1 – Instruction Segmentation Unit

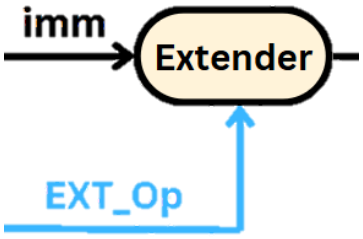
2.3.5 Branch Adder



The Branch Adder takes the current program counter (PC) and a sign extended immediate value as inputs to compute the branch target address during the decode stage. This calculated target is used later in the execution stage if the branch is determined to be taken.

Figure 2.3.5.1 – Instruction Segmentation Unit

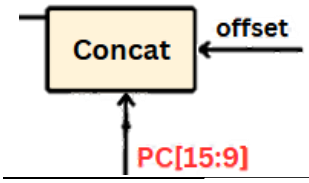
2.3.6 Extender



The Extender takes a 6 bits immediate and extends it to a 16 bits immediate. The type of extension (zero or sign extension) is determined by the EXT_Op input. If EXT_Op = 0 (zero extension), 10 zeros are added to the immediate. If EXT_Op = 1 (sign extension), the 10 bits added are zeros for a positive immediate or ones for a negative immediate.

Figure 2.3.6.1 – Extender

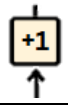
2.3.7 Concatenation Unit



The concatenation unit receives the offset and PC [15:9] and concatenates them to calculate the jump target address as follows: jump target address = PC [15:9] || offset.

Figure 2.3.7.1 – Concatenation Unit

2.3.8 Incrementer



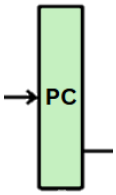
The incrementer is a simple unit that receives the current PC and adds 1 to it.

Figure 2.3.8.1 – Incrementer

2.4 Registers

Each processor requires a program counter register, and the multi cycle processor requires more registers besides the program counter, registers that are between the stages to hold the output of the previous stage in order to use in the next stage at the next clock cycle, and in this project there is an additional required registers, they are called the performance register and the return back register. In total, our processor includes 22 registers, distributed as follows: 8 registers in the register file and 14 additional registers.

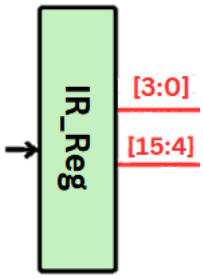
2.4.1 Program Counter (PC)



This Register is called the program counter (PC) register, it is a 16 bit register that stores the address of the instruction to be fetched from the instruction memory during the Instruction Fetch stage (IF). At the beginning of each IF cycle, it will be updated to point to the next instruction to be fetched, i.e. the PC register holds the address of the current instruction until the it ends, After that the pc will store the next instruction address.

Figure 2.4.1.1 – Program Counter (PC)

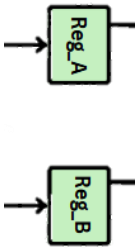
2.4.2 Instruction Register (IR)



This register is called the Instruction Register (IR_Reg). It stores the instruction fetched from the instruction memory during the Instruction Fetch (IF) stage. Once the instruction is loaded into the IR_Reg, it divides the instruction into two outputs: the first 4 bits, which represent the opcode, and the remaining bits, which are sent to the Decode stage as an input to the Instruction segmentation unit. The IR_Reg holds the instruction until it is processed in the next stages.

Figure 2.4.2.1 – Instruction Register (IR)

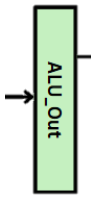
2.4.3 Operand Registers



The operand registers, Reg_A and Reg_B, store the values of Bus_A and Bus_B, respectively, which are outputs from the register file during the Decode stage. These registers then forward their values to the multiplexer that controls the ALU inputs.

Figure 2.4.3.1 – Operand Registers

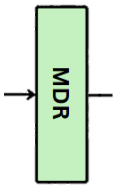
2.4.4 ALU Output Register



The ALU_out register stores the result of the operation performed by the ALU during the Execute stage (EX). This register holds the computed value temporarily and forwards it to the next stage which depends on the instruction type.

Figure 2.4.4.1 – ALU Output Register

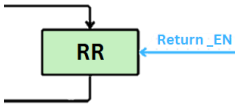
2.4.5 Memory Data Register



The Memory Data Register (MDR) exists in the Write Back stage (WB) but holds the output from the Memory Access stage (Mem). It stores this data to be written to a register in the register file and forwards it to the multiplexer that controls the Bus_W source for the write back operation.

Figure 2.4.5.1 – Memory Data Register

2.4.6 Return Register



This is a 16 bit register that stores the return address during function call when the Return_EN is High.

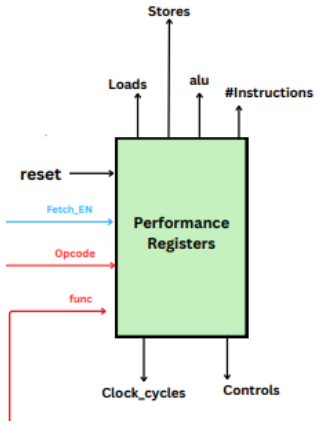
Figure 2.4.6.1 – Return Register

2.4.7 Iterations Register

The Iteration Register receives a number that represent the number of iteration for the “for” instruction, this value can only be written on it when the Enable signal is High, it also has a built in decrementer that decrements the value by one with each executed instruction in the loop.

Figure 2.4.7.1 – Iterations Register

2.4.8 Performance Registers



The Performance Registers Unit increments the appropriate register according to the type of instruction by using the opcode of each instruction. Total executed instructions, total load instructions, total store instructions, total ALU instructions, total control instructions, and total clock cycles are among the metrics it monitors. The performance of the CPU is tracked by updating each of these parameters appropriately.

Figure 2.4.8.1 – Performance Registers

3. Implementation

3.1 Micro-Architecture Diagram

All of the discussed and explained components in the Pre-implementation part will now be assembled to make the Micro-Architecture Diagram as shown in Figure [3.1.1] below, which consists of Data and control paths:

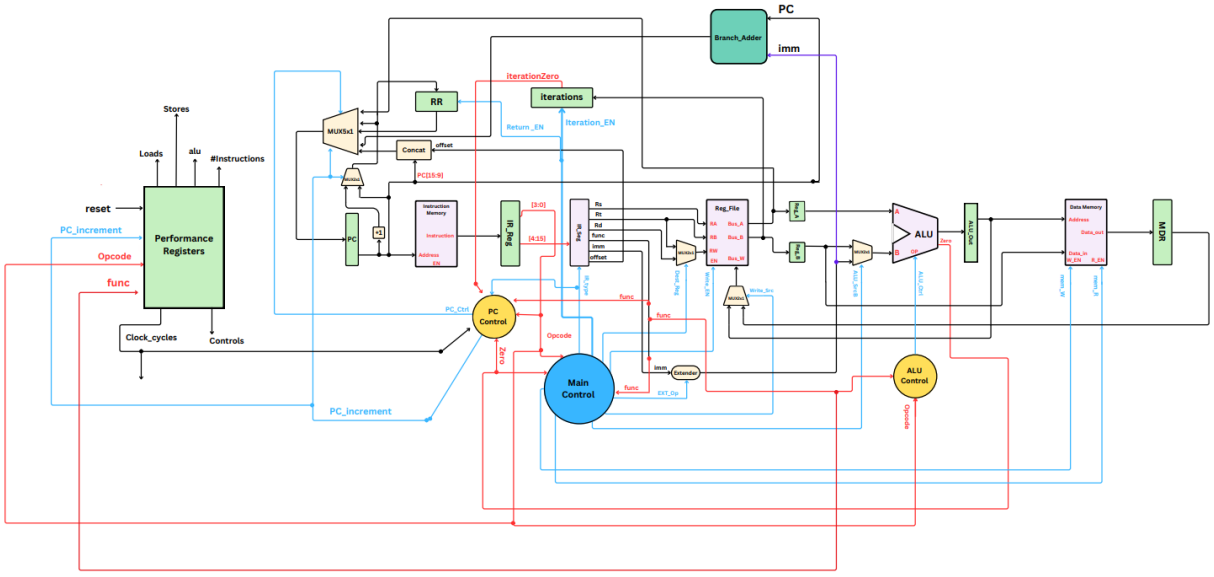


Figure 3.1.1 – Micro-Architecture Diagram

The diagram above represents the data and control path for the designed multi cycle processor, it consists of functional units for preforming operations on the instruction fields, registers to store temporary data throughout the instruction processing, and control units to control the data flow in the data path. The processor operates in five stages: Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute (EX), Memory Access (MEM), and Write Back (WB). Instructions progress through these stages as follows:

- **Instruction Fetch (IF):**

  The PC register contains the address of the instruction to be fetched from the instruction memory, this address has different sources, in which the PC register's input is a 5x1 mux's output, the inputs of this mux are PC +1, RR value, Branch target, jump target, and Rs, allowing the PC to be any of these five values. Once the address is selected, the instruction is fetched and stored in the IR_Reg to ensure the value stability for the next stage.

- **Instruction Decode (ID)**

  The stored Value in the IR_Reg divides the instruction into two outputs: the first 4 bits, which represent the opcode, and the remaining bits, which are sent to the Decode stage as an input to the Instruction segmentation unit, which will divide its input into segments of variant sizes based on the instruction type input (IR_Type). For IR_Type = 0 (R_Type), it segments the instruction (excluding the opcode) as follows: Rd (first 3 bits), Rs (second 3 bits), Rt (third 3 bits), and Function (last 3 bits), with imm = 0 and offset = 0. Similarly, it segments I_Type and J_Type instructions when IR_Type = 1 or 2, respectively, with the appropriate dividing based on each instruction format. After segmentation, the Register File processes the decoded instruction by reading the values from the source registers (Rs and Rt) and forwarding them to the operand registers, Reg_A and Reg_B. These registers hold the values for use in the Execute (EX) stage. Additionally, the branch target address will be calculated in this stage, this calculated target is used later in the execution stage if the branch is determined to be taken.

- **Instruction Execution (EX):**

  The ALU has two inputs, one of them is the value of Reg_A, and the other one is an outputs of a 2x1 muxe, the mux's inputs are either an extended immediate or the stored value in Reg_B, the ALU also has a ALU_ctrl input that determine the operation that will be performed on its inputs. After finishing the operation, two outputs will be generated, ALU output and ZERO output, the ALU output is the result of the performed operation on the two operands, and the ZERO is an output that becomes one when the result of the performed operation equals zero. At the end of the execution stage, the result will be stored in the ALU_Out register to ensure the value stability for the next stage.

- **Memory Access (MEM)**

  This stage is for instructions that has access to the Data memory, i.e. Load and Store instructions, in these instructions case, the ALU_Out register will be holding a calculated address by the ALU in the EX stage, this address will be an input for the Data memory to read from, or write into it. The read or write operation is determined by Write Enable (mem_W) and Read Enable (mem_R). in the case of the Store instruction, the mem_W will be active, thus, the address will be used to write into the Data memory at that address. For the Load instruction, the mem_R will be active, thus, the address will be used to read from. The value in that address will be the output of the memory (Data_out), and this output will be stored in the memory data register (MDR) for further usage in the write back stage (WB).

- **Write Back (WB)**

  This stage is for writing on a destination register in the register file while the input Write_EN is active, using a mux that chooses the data to be from the ALU_out register or the MDR, based on its selection line.

3.2 Control Units and Signals

All of the explained stages above need a manager to control the data flow and the paths that it should take, this manager is the control unit. Actually, there is three control units, Main control, ALU control, and the PC control.

3.2.1 Main Control Unit

This Unit controls the whole data path control signals, it turns components on and off, provides selection lines data to muxes, and changes the mode of operation for some components. All of this signals is generated based on three received inputs, which are the Opcode, function, and ZERO, these three values determine the path that instruction will go through in order to be processed, and the signals are shown in the table below with their corresponding instruction:

	Instruction	Opcode	Func	Zero	IR_type	Iteration_EN	Return_EN	Dest_reg
Rtype	And	"0000"	"000"	0	0 (R-type)	0	0	0 (Rd)
	Add	"0000"	"001"	x	0 (R-type)	0	0	0 (Rd)
	Sub	"0000"	"010"	x	0 (R-type)	0	0	0 (Rd)
	SLL	"0000"	"011"	x	0 (R-type)	0	0	0 (Rd)
	SRL	"0000"	"100"	x	0 (R-type)	0	0	0 (Rd)
I-type	ANDI	"0010"	111 (NA)	x	1(I-type)	0	0	1(Rt)
	ADDI	"0011"	111 (NA)	x	1(I-type)	0	0	1(Rt)
	LW	"0100"	111 (NA)	x	1(I-type)	0	0	1(Rt)
	SW	"0101"	111 (NA)	x	1(I-type)	0	0	X
	BEQ	"0110"	111 (NA)	should be 1 after the sub	1(I-type)	0	0	X
	BNE	"0111"	111 (NA)	should be 0 after the sub	1(I-type)	0	0	X
	FOR	"1000"	111 (NA)	x	1(I-type)	1	0	X
J-Type	JMP Offset	"0001"	"000"	x	2 (J-type)	0	0	X
	CALL Offset	"0001"	"001"	x	2 (J-type)	0	1	X
	RET	"0001"	"010"	x	2 (J-type)	0	0	X

	Instruction	Opcode	Func	Zero	Write_EN	EXT_Op	Write_Src	ALU_SrcB	mem_W	mem_R
Rtype	And	"0000"	"000"	0	1	0 (Zero)	1 (From ALU)	1 (Reg_B)	0	0
	Add	"0000"	"001"	x	1	1 (signed)	1 (From ALU)	1 (Reg_B)	0	0
	Sub	"0000"	"010"	x	1	1 (signed)	1 (From ALU)	1 (Reg_B)	0	0
	SLL	"0000"	"011"	x	1	0 (Zero)	1 (From ALU)	1 (Reg_B)	0	0
	SRL	"0000"	"100"	x	1	0 (Zero)	1 (From ALU)	1 (Reg_B)	0	0
I-type	ANDI	"0010"	111 (NA)	x	1	0 (Zero)	1 (From ALU)	0 (immediate)	0	0
	ADDI	"0011"	111 (NA)	x	1	1 (signed)	1 (From ALU)	0 (immediate)	0	0
	LW	"0100"	111 (NA)	x	1	1 (signed)	0 (from Mem)	0 (immediate)	0	1
	SW	"0101"	111 (NA)	x	0	1 (signed)	X	0 (immediate)	1	0
	BEQ	"0110"	111 (NA)	should be 1 after the sub	0	1 (signed)	X	1 (Reg_B)	0	0
	BNE	"0111"	111 (NA)	should be 0 after the sub	0	1 (signed)	X	1 (Reg_B)	0	0
	FOR	"1000"	111 (NA)	x	0	X	X	X	0	0
J-Type	JMP Offset	"0001"	"000"	x	X	X	X	X	X	X
	CALL Offset	"0001"	"001"	x	X	X	X	X	X	X
	RET	"0001"	"010"	x	X	X	X	X	X	X

Table 3.2.1.1 – Main Control Signals truth table

❖ Boolean Functions

- **Iteration_En =** $= FOR$
- **Return_EN =** $CALL\ Offset$
- **Dest_Reg =** $ANDI + ADDI + LW$
- **Write_EN =** $\overline{SW + BEQ + BNE + FOR}$
- **EXT_Op =** $\overline{And + SLL + SRL + ANDI}$
- **Write_Src =** $\overline{LW}$
- **ALU_SrcB =** $\overline{ANDI + ADDI + LW + SW}$
- **mem_W =** SW
- **mem_R =** LW

❖ State Diagram

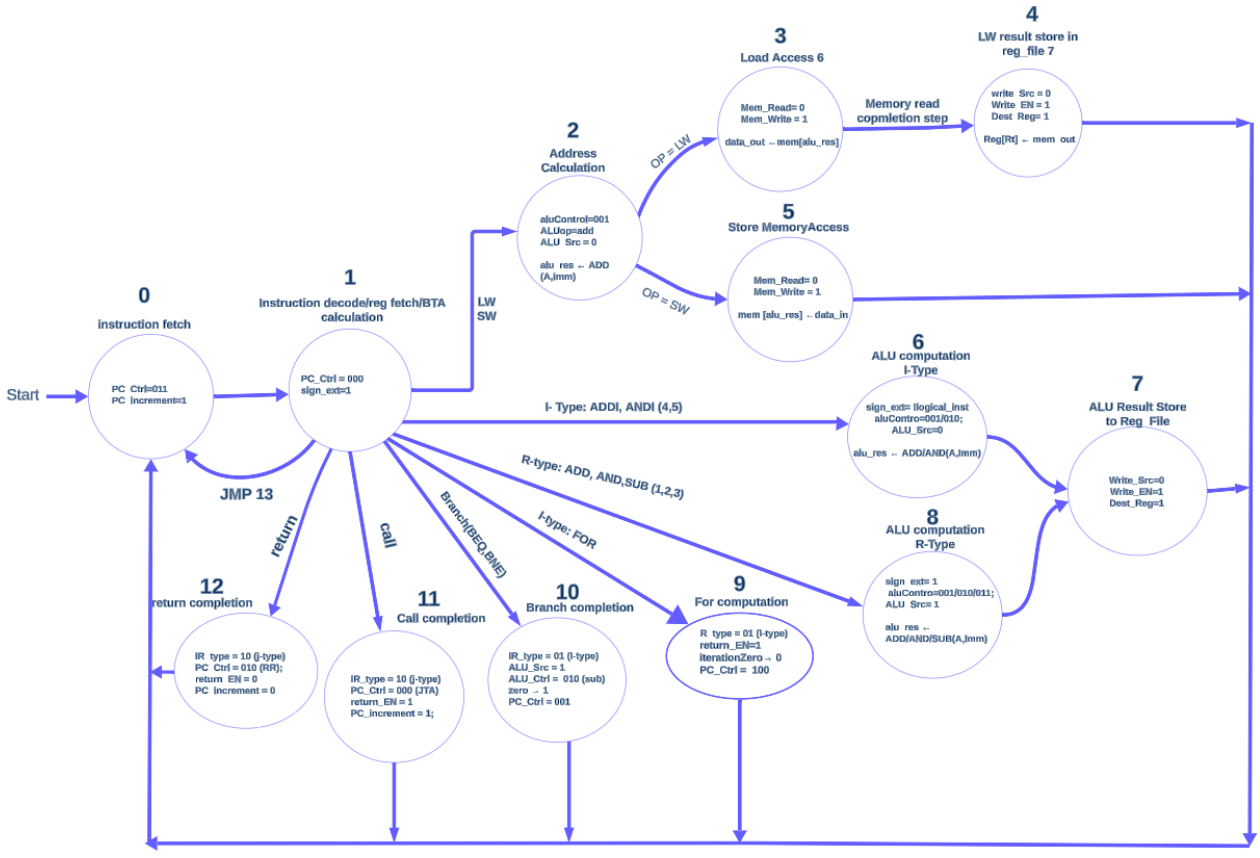


Figure 3.2.1.1 – Processor finite state machine

- **Summary Table of the Instruction Execution Flow:**

State	Description	Key Actions
0	Instruction Fetch	PC increments by 1, fetches the instruction.
1	Instruction Decode/Fetch/BTA Calculation	Decodes instruction, determines operation type.
2	Address Calculation	Calculates address for memory operations (LW/SW).
3	Load Memory Access	Reads data from memory if LW.
4	LW Result Stored in Register File	Stores the loaded data into the specified register.
5	Store Memory Access	Writes data to memory if SW.
6	ALU Computation (I-Type)	Executes ADDI, ANDI instructions using ALU.
7	ALU Result Stored to Register File	Stores ALU result back to register file.
8	ALU Computation (R-Type)	Executes ADD, AND, SUB, etc., using ALU.
9	For Computation	Iterates based on Rt value, exits loop if zero.
10	Branch Completion	Determines if branch (BEQ, BNE) is taken.
11	Call Completion	Stores return address and jumps to subroutine.
12	Return Completion	Returns to the saved address after subroutine.

Table 3.2.1.2 – FSM summarization table

3.2.2 ALU Control Unit

The ALU in this project provides 5 operations: ADD, AND, SUB, SLL, and SRL. The ALU Control Unit controls the operation of the ALU based on the instruction and the kind of logical or arithmetic calculation requirement, it receives the Opcode and the function as inputs, and generates ALU_Ctrl as an output as follows:

	Instruction	Opcode	Func	ALU_Ctrl
Rtype	And	"0000"	"000"	"001"
	Add	"0000"	"001"	"010"
	Sub	"0000"	"010"	"011"
	SLL	"0000"	"011"	"100"
	SRL	"0000"	"100"	"101"
I-type	ANDI	"0010"	111 (NA)	"001"
	ADDI	"0011"	111 (NA)	"010"
	LW	"0100"	111 (NA)	"010"
	SW	"0101"	111 (NA)	"010"
	BEQ	"0110"	111 (NA)	"011"
	BNE	"0111"	111 (NA)	"011"
	FOR	"1000"	111 (NA)	x
J-Type	JMP Offset	"0001"	"000"	x
	CALL Offset	"0001"	"001"	x
	RET	"0001"	"010"	x

Table 3.2.2.1 – ALU Control Signals truth table

3.2.3 PC Control Unit

After finishing processing each instruction, the next instruction has to be fetched from the instruction memory in the IF stage, the PC register's input is connected to a 5x1 mux to provide different PCs updates based on the instruction type, and that's the PC control Unit role, in which it takes the Opcode, Function, and the IR_Type to generate an output that controls the 5x1 mux as follows:

	Instruction	Opcode	Func	IR_type	Zero	PC_Ctrl
Rtype	And	x	x	0 (R-type)	X	"011"
	Add					
	Sub					
	SLL					
	SRL					
I-type	ANDI	"0010"	x	1(I-type)	x	"011"
	ADDI	"0011"		1(I-type)	x	
	LW	"0100"		1(I-type)	x	
	SW	"0101"		1(I-type)	x	
	BEQ	"0110"		1(I-type)	0	"011"
					1	"001"
	BNE	"0111"		1(I-type)	0	"001"
					1	"011"
	FOR	"1000"		1(I-type)	x	"011" if iterations == 0 else "100"
J-Type	JMP Offset	"0001"	"000"	2 (J-type)	x	"000"
	CALL Offset	"0001"	"001"	2 (J-type)	x	"000"
	RET	"0001"	"010"	2 (J-type)	x	"010"

Table 3.2.3.1 – PC Control Signals truth table

4. RTL

4.1 PC module

```
module PC(
    input wire clk,           // Clock signal
    input wire [15:0] new_address, // 16-bit input address
    output reg [15:0] pc_out  // 16-bit output: Current PC value
);

 initial begin
    pc_out = 16'h0000;
end

initial begin
    @(posedge clk);
    pc_out = 16'h0000;
    forever@(posedge clk)
        pc_out <= new_address;
end


endmodule
```

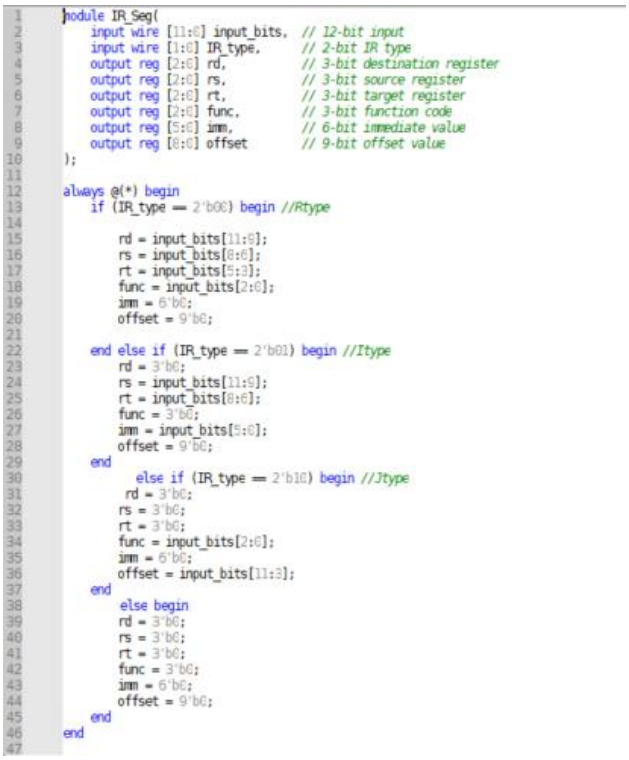
This PC module models a **Program Counter** is a critical part of a processor responsible for managing the flow of instructions. It determines which instruction to fetch next and updates its value during normal execution, jumps, or branches.

4.2 Instruction memory

```
module instruction_memory (
    input [15:0] address,       // 16-bit address input
    input fetch_en,             // Enable signal
    output reg [15:0] instruction // 16-bit instruction output
);

    reg [15:0] instruction_array [0:255]; // 256 words of 16-bit instructions
    `include "Constants.v"
    initial begin

        // R-type:
        instruction_array[0] = {Constants.SLL, 3'b100, 3'b101, 3'b001, 3'b011}; // rd=1, rs=2, rt=3, func=0
        instruction_array[1] = {Constants.SLL, 3'b100, 3'b101, 3'b001, 3'b011}; // rd=1, rs=2, rt=3, func=0
        //instruction_array[1] = {Constants.ADD, 3'b001, 3'b010, 3'b011, 3'b000}; // rd=1, rs=2, rt=3, func=0
        instruction_array[2] = {Constants.SLL, 3'b100, 3'b101, 3'b001, 3'b011}; // rd=4, rs=5, rt=6, func=3

        // I-type:
        instruction_array[3] = {Constants.SLL, 3'b100, 3'b101, 3'b001, 3'b011}; // rd=1, rs=2, rt=3, func=0

        //instruction_array[3] = {Constants.ANDI, 3'b010, 3'b011, 6'b000111}; // rs=2, rt=3, imm=7
        instruction_array[4] = {Constants.SLL, 3'b100, 3'b101, 3'b001, 3'b011}; // rs=5, rt=6, imm=2

        // J-type:
        instruction_array[5] = {Constants.SLL, 3'b100, 3'b101, 3'b001, 3'b011};//{Constants.JMP, 9'b000001010, 3'b000}; // offset=10, func=0
        instruction_array[6] = {Constants.CALL, 9'b000010100, 3'b001}; // offset=20, func=1

            end


    always @(*) begin
        if (fetch_en) begin
            instruction = instruction_array[address];
        end else begin
            instruction = 16'h0000;
        end
    end
endmodule
```

The instruction_memory stores 256 preloaded 16-bit instructions. It outputs the instruction at a given address when fetch_en is high; otherwise, it outputs 16'h0000. This module enables instruction fetching for processors, controlled by the fetch_en signal**.**

4.3 IR_Seg



```
module IR_Seg(
    input wire [11:0] input_bits,  // 12-bit input
    input wire [1:0] IR_type,      // 2-bit IR type
    output reg [2:0] rd,           // 3-bit destination register
    output reg [2:0] rs,           // 3-bit source register
    output reg [2:0] rt,           // 3-bit target register
    output reg [2:0] func,         // 3-bit function code
    output reg [5:0] imm,          // 6-bit immediate value
    output reg [8:0] offset        // 9-bit offset value
);

always @(*) begin
    if (IR_type == 2'b00) begin //Rtype

        rd = input_bits[11:9];
        rs = input_bits[8:6];
        rt = input_bits[5:3];
        func = input_bits[2:0];
        imm = 6'b0;
        offset = 9'b0;

    end else if (IR_type == 2'b01) begin //Itype
        rd = 3'b0;
        rs = input_bits[11:9];
        rt = input_bits[8:6];
        func = 3'b0;
        imm = input_bits[5:0];
        offset = 9'b0;
    end
          else if (IR_type == 2'b10) begin //Jtype
         rd = 3'b0;
        rs = 3'b0;
        rt = 3'b0;
        func = input_bits[2:0];
        imm = 6'b0;
        offset = input_bits[11:3];
    end
        else begin
        rd = 3'b0;
        rs = 3'b0;
        rt = 3'b0;
        func = 3'b0;
        imm = 6'b0;
        offset = 9'b0;
    end
end
```

Function: Decodes the instruction based on its type (R-type, I-type, or J-type). RTL: Splits the 12-bit input_bits into separate fields like rd, rs, rt, func, imm, or offset based on IR_type. Default values are assigned for invalid types.

4.4 IR_Reg

```
module IR_Reg(
    input wire clk,                 // Clock signal
    input wire [15:0] instruction,  // 16-bit instruction input
    output reg [3:0] opcode,        // 4-bit opcode (MSBs of instruction)
    output reg [11:0] rest          // Remaining 12 bits of the instruction
);

always @(posedge clk) begin
    opcode <= instruction[15:12];  // Extract the 4 MSBs as opcode
    rest <= instruction[11:0];     // Extract the remaining 12 bits
end

endmodule
```

Function: Separates the opcode and the remaining bits of the instruction. **RTL**: Captures the 4 MSBs (opcode) and 12 LSBs (rest) from a 16-bit instruction on each positive clock edge.

4.5 Reg_File

```
module reg_file (

    input clk,
    input write_en,
    input [2:0] RA, RB, RW,
    input [15:0] bus_w,
    output reg [15:0] bus_A, bus_B
);

    reg [15:0] regArray [0:7] = '{
    16'h0000,  // 0
    16'h000A,  // 10
    16'hFFF0,  // -16 (two's complement of 16: 0xFFFF - 16 + 1 = 0xFFF0)
    16'h0014,  // 20
    16'hFFEC,  // -20 (two's complement of 20: 0xFFFF - 20 + 1 = 0xFFEC)
    16'h0010,  // 16
    16'hFFF6,  // -10 (two's complement of 10: 0xFFFF - 10 + 1 = 0xFFF6)
    16'h000B  // 11

};


    always @(*) begin //The output is taken asynchronously
            bus_A = regArray[RA];
            bus_B = regArray[RB];
    end

    always @(posedge clk) begin

        if (write_en)
            regArray[RW] <= bus_w;


    end


endmodule
```

Function: Implements a register file with read and write capabilities. **RTL**: Uses asynchronous reads for bus_A and bus_B based on RA and RB and synchronous writes to registers when write_en is active.

4.6 ALU

```
module ALU (

    input [2:0] alu_ctrl,
    input signed [15:0] a, b,
    output zero, overflow,neg,
    output reg signed  [15:0] alu_out
    );

    reg [1:0] carry;
    assign zero = (alu_out == 0); // zero flag
    assign overflow = ( carry[0] ^ carry[1]); // overflow flag
    assign neg = alu_out [15];  // neg flag

    always @(*)begin

        case (alu_ctrl)

            3'd1 : alu_out = a & b;  //AND
            3'd2 : begin
                    {carry[0], alu_out[14:0]} = a[14:0] + b[14:0];    //ADD
                    {carry[1], alu_out[15]} = a[15] + b[15] + carry[0];
                end

            3'd3 : alu_out = a - b;      // SUB
            3'd4 : alu_out = a << b[3:0];  //SLL
            3'd5 : alu_out = a >> b[3:0];  //SLR
             default: alu_out = 16'b0;

        endcase

         end

endmodule
```

Function: Performs arithmetic and logic operations based on alu_ctrl. **RTL**: Implements operations like AND, ADD, SUB, SLL, and SLR with flags (zero, overflow, and neg) for specific conditions.

4.7 Data_Memory

```
module data_memory (
    input [15:0] address,   // 16-bit address input
    input [15:0] data_in,   // 16-bit data input
    input mem_W,            // Memory write enable
    input mem_R,            // Memory read enable
    input clk,              // Clock signal
    output reg [15:0] data_out // 16-bit data output
);


    reg [15:0] memory [0:255]; // 256 words of 16-bit memory


    initial begin
        memory[0] = 16'h0000;
        memory[1] = 16'h0034;
        memory[2] = 16'h0078;
        memory[3] = 16'h00BC;

    end

    // Read operation (combinational logic)
    always @(*) begin
        if (mem_R) begin

            data_out = memory[address];
        end else begin
            // Default output value when not reading
            data_out = 16'h0000;
        end
    end

    // Write operation (synchronized with the clock)
    always @(posedge clk) begin
        if (mem_W) begin
            memory[address] <= data_in;
        end
    end

endmodule

```

Function: Simulates data memory with read and write functionality. **RTL**: Uses combinational logic for reads (mem_R) and synchronous logic for writes (mem_W) to a 256-word memory.

4.8 Branch Adder

```
module BranchAdder(
    input wire [15:0] pc,        // 16-bit Program Counter
    input wire [15:0] offset,  // 16-bit Offset
    output wire [15:0] branch_address // 16-bit Branch Address
);

assign branch_address = pc + offset; // Add PC and Offset

endmodule

```

Function: Calculates the branch target address. **RTL**: Adds the pc value and offset using simple addition logic

4.9 Extender

```
module extender (

    input [5:0] imm,
    input sign_ext,
    output reg [15:0] extImm
);

    always @(*) begin
        if (sign_ext && imm[5])
             extImm = {10'b1111111111, imm};
        else if (sign_ext && ~imm[5])
            extImm = {10'b0000000000, imm};


    end

endmodule
```

Function: Extends a 6-bit immediate value to 16 bits (sign or zero extension). **RTL**: Uses conditional logic to perform sign extension or zero extension based on the sign_ext input.

4.10 Reg

```
module Reg(
    input wire clk,         // Clock signal
    input wire [15:0] in, // 16-bit input
    output reg [15:0] out  // 16-bit output
);


always @(posedge clk) begin
        out <= in;
    end

endmodule

module Reg1(
    input wire en,
    input wire clk,         // Clock signal
    input wire [15:0] in, // 16-bit input
    output reg [15:0] out  // 16-bit output
);


always @(posedge clk && en) begin
        out <= in;
    end

endmodule
```

Function: Both are 16-bit storage registers that update their output (out) on the positive clock edge (clk).

- **Reg Module**: Updates unconditionally on every clock cycle, always capturing the input (in).
- **Reg1 Module**: Updates conditionally, capturing the input (in) only when the enable signal (en) is high. If en is low, the register retains its current value

4.11 regMux2to1

```
module RegMux2to1(
    input wire [2:0] in0,    // 3-bit input 0
    input wire [2:0] in1,    // 3-bit input 1
    input wire  sel,          // 1-bit selector
    output reg [2:0] out     // 3-bit output
);


    always @(*) begin
        case (sel)
            1'b0: out = in0;  //Rd
            1'b1: out = in1;  //Rt
        endcase
    end

endmodule
```

Function: A 2-to-1 multiplexer that selects between two 3-bit inputs (in0 and in1) based on a 1-bit selector signal (sel). **RTL**:

- If sel = 0, the output (out) is assigned the value of in0 (corresponding to Rd).
- If sel = 1, the output (out) is assigned the value of in1 (corresponding to Rt).

4.12 Mux5to1 and Mux2to1

```
module Mux5to1(
    input wire [15:0] in0,    // 16-bit input 0
    input wire [15:0] in1,    // 16-bit input 1
    input wire [15:0] in2,    // 16-bit input 2
    input wire [15:0] in3,    // 16-bit input 3
    input wire [15:0] in4,    // 16-bit input 4
    input wire [2:0] sel,     // 3-bit selector
    output reg [15:0] out     // 16-bit output
);


    always @(*) begin
        case (sel)
            3'b000: out = in0;  //   input 0 =Jump  (pc[15:9] || 9 bit offset )
            3'b001: out = in1;  //   input 1 = branch taken (pc = pc + extended (6 bit immediate) )
            3'b010: out = in2;  //   input 2 = RET (pc= Return address)
            3'b011: out = in3;  //   input 3 = Pc = pc+1
            3'b100: out = in4;  //   input 4 = For
            default: out = in3;
        endcase
    end

endmodule
```

```
module Mux2to1(
    input wire [15:0] in0,    // 16-bit input 0
    input wire [15:0] in1,    // 16-bit input 1
    input wire  sel,          // 1-bit selector
    output reg [15:0] out     // 16-bit output
);


    always @(*) begin
        case (sel)
            1'b0: out = in0;
            1'b1: out = in1;
        endcase
    end

endmodule
```

Function: A 2-to-1 multiplexer that selects between two 16-bit inputs (in0 and in1) based on a 1-bit selector signal (sel). **RTL**:

1. If sel = 0, the output (out) is assigned the value of in0.
2. If sel = 1, the output (out) is assigned the value of in1.

Function: A 5-to-1 multiplexer that selects one of five 16-bit inputs (in0 to in4) based on a 3-bit selector signal (sel). **RTL**: Depending on the value of sel:

i. 3'b000: Output is in0 (Jump address, concatenation of pc[15:9] and 9-bit offset).

ii. 3'b001: Output is in1 (Branch target address, pc + extended immediate).

iii. 3'b010: Output is in2 (Return address stored in the PC).

iv. 3'b011: Output is in3 (Next PC, pc + 1).

v. 3'b100: Output is in4 (For loop address or other custom behavior).

Default: Outputs in3 (Next PC).

4.13 Pc Control

```
module pcControl (
    input [3:0] opcode,     // 4-bit opcode (not the primary control)
    input [2:0] func,       // 3-bit function
    input zero,             // Zero flag
    input [1:0] ir_type,    // 2-bit Instruction Register Type
    input iterationsZero,   //a flag if the iterations are zero
    output reg [2:0] pc_ctrl // 3-bit PC control signal
);

    always @(*) begin
        // Default pc_ctrl value
        pc_ctrl = 3'b011;

        case (ir_type)

          //---------------------- R-type instructions-----------------------------
            2'b00: begin
                pc_ctrl = 3'b011;
            end

          //---------------------- I-type instructions-----------------------------

          2'b01: begin
              case (opcode)
                  // ANDI, ADDI ,LW , SW ----------------------------------------------
                  4'b0010,4'b0011,4'b0100,4'b0101: begin
                      pc_ctrl = 3'b011;
                  end

                  // BEQ----------------------------------------------------------
                  4'b0110: begin
                      if (zero) begin
                          pc_ctrl = 3'b001; // Branch taken
                      end else begin
                          pc_ctrl = 3'b011; // Branch not taken
                      end
                  end
```

```
            end

            // BNE
            4'b0111: begin
              if (zero) begin
                pc_ctrl = 3'b011; // Branch not taken
              end else begin
                pc_ctrl = 3'b001; // Branch taken
              end
            end
            //JUMP
            4'b1000: begin
              if (iterationsZero) begin
                pc_ctrl = 3'b100; //Loop
              end else begin
                pc_ctrl = 3'b011; //next_instruction
              end
            end


        endcase
    end

    //---------------------- J-type instructions-----------------------------
    2'b10: begin
        case (func)
            4'b000: begin //RET
                pc_ctrl = 3'b010;
            end
            default: begin
                pc_ctrl = 3'b000; // Default case
            end
        endcase
    end


    default: begin
        pc_ctrl = 3'b011; // Default case
    end
  endcase
end
```

Function: Generates the program counter (PC) control signal (pc_ctrl) based on the instruction type, opcode, function code, zero flag, and loop iterations flag. **RTL**:

➔ For **R-type instructions** (ir_type = 2'b00): Default PC increment (pc_ctrl = 3'b011).

➔ For **I-type instructions** (ir_type = 2'b01):

✓ ALU operations (ANDI, ADDI, LW, SW): Default PC increment.

- ✓ **BEQ**: Branch taken (pc_ctrl = 3'b001) if zero = 1; otherwise, increment.
- ✓ **BNE**: Branch taken if zero = 0; otherwise, increment.
- ✓ **FOR**: Loop execution based on iterationsZero.

➔ For **J-type instructions** (ir_type = 2'b10):

- ✓ **RET**: Set PC to return address (pc_ctrl = 3'b010).
- ✓ Default: PC increment.

➔ **Default case**: Increment PC by default (pc_ctrl = 3'b011).

4.14 ALU Control

```
module ALUcontrol (

    input [3:0] opcode,
    input [2:0] func,
    output reg [2:0] ALU_ctrl
);

parameter
    AND = 3'b001,
    ADD = 3'b010,
    SUB = 3'b011,
    SLL = 3'b100,
    SRL = 3'b101;

always @(*) begin

    if (opcode == 4'b0000) begin

        case (func)
     //---------------------- R-type instructions----------------------------
            3'b000: ALU_ctrl = AND; // AND
            3'b001: ALU_ctrl = ADD; //ADD
            3'b010: ALU_ctrl = SUB; //SUB
            3'b011: ALU_ctrl = SLL; //SLL
            3'b100: ALU_ctrl = SRL; //SRL
            default: ALU_ctrl = 3'bx;//NONE
        endcase
    end
```

```
    else begin

        case (opcode)
       //---------------------- I-type instructions-
            4'b0010: ALU_ctrl = AND;// ANDI
            4'b0011: ALU_ctrl = ADD;//ADDI
            4'b0100: ALU_ctrl = ADD;//LW
            4'b0101: ALU_ctrl = ADD;//SW
            4'b0110: ALU_ctrl = SUB;//BEQ
            4'b0111: ALU_ctrl = SUB;//BNE
            default: ALU_ctrl = 3'bx;//NONE
        endcase
    end
end

endmodule
```

Function: Decodes the opcode and func fields to generate the appropriate ALU control signal (ALU_ctrl) for both R-type and I-type instructions. **RTL**:

➔ **R-type instructions** (opcode = 4'b0000):

Decodes func to set operations like AND, ADD, SUB, SLL, and SRL.

➔ **I-type instructions**:

Decodes opcode for operations like ANDI, ADDI, LW, SW, BEQ, and BNE.

Sets ALU_ctrl to ADD or SUB as appropriate.

➔ **Default**:

Outputs 3'bx for undefined or unsupported instructions.

4.15 Main Control

```
module main_control_unit (
    input [3:0] opcode,     // 4-bit opcode
    input [2:0] func,       // 3-bit function
    input zero,             // Zero flag
    output reg fetch_en,    // Fetch enable
    output reg [1:0] ir_type,// Instruction Register Type
    output reg return_en,   // Return Enable
    output reg dest_reg,    // Destination Register
    output reg write_en,    // Write Enable
    output reg ext_op,      // Extension Operation
    output reg write_src,   // Write Source
    output reg alu_srcB,    // ALU Source B
    output reg mem_w,       // Memory Write
    output reg mem_r        // Memory Read
);

   reg alu_srcA;
    always @(*) begin


        case (opcode)

            //-------------------- R-type instructions--------------
            4'b0000: begin

                fetch_en = 1;        //instruction memory on
                ir_type = 0;         // r-type
                return_en = 0;       // no RR update
                dest_reg = 0;        // Rd is the destination
                write_en = 1;        // Write to reg

                case (func)
                    3'b000, 3'b011, 3'b100: begin
                        //no changes
                    end
                    3'b001, 3'b010: begin //Add sub
                        ext_op = 1;   //signed extention
                    end

                endcase
            end

            //-------------------- J-type instructions----------------

           4'b0001: begin

                fetch_en = 1;        //instruction memory on
                ir_type = 2;         //J-type
                return_en = 0;       // no RR update

                //Not identified
                dest_reg = 1'bx;
                write_en = 1'bx;
                ext_op = 1'bx;
                write_src = 1'bx;
                alu_srcA = 1'bx;
                alu_srcB = 1'bx;
                mem_w = 1'bx;
                mem_r = 1'bx;

                case (func)
                    3'b000: begin  //Jump
                    end
                    3'b001: begin  //Call
                        return_en = 1;
                    end
                    3'b010: begin  //RET
                        fetch_en = 0;
                    end

                endcase


        //-------------------- I-type instruction------------------

        //ANDI :
        4'b0010: begin

            fetch_en = 1;        //instruction memory on
            ir_type = 1;         // I-type
            return_en = 0;       // no RR update
            dest_reg = 1;        // Rt is the destination
            write_en = 1;        // Write to reg
            ext_op = 0;          // zero extension
            write_src = 1;       // the result is from the ALU
            alu_srcA = 1;        // from Reg_A
            alu_srcB = 0;        // immediate
            mem_w = 0;           // No memory write
            mem_r = 0;           // No memory read

        end

        //ADDI :
        4'b0011: begin

            fetch_en = 1;        //instruction memory on
            ir_type = 1;         // I-type
            return_en = 0;       // no RR update
            dest_reg = 1;        // Rt is the destination
            write_en = 1;        // Write to reg
            ext_op = 1;          // signed extension
            write_src = 1;       // the result is from the ALU
            alu_srcA = 1;        // from Reg_A
            alu_srcB = 0;        // immediate
            mem_w = 0;           // No memory write
            mem_r = 0;           // No memory read

        end

        //LW :
        4'b0100: begin

            fetch_en = 1;        //instruction memory on
            ir_type = 1;         // I-type
            return_en = 0;       // RR update
            dest_reg = 1;        // Rt is the destination
            write_en = 1;        // Write to reg
            ext_op = 1;          // signed extension
            write_src = 0;       // the result is from the memory
            alu_srcA = 1;        // from Reg_A
            alu_srcB = 0;        // immediate
            mem_w = 0;           // No memory write
            mem_r = 1;           // memory read

        end

        //SW :
        4'b0101: begin

            fetch_en = 1;        //instruction memory on
            ir_type = 1;         // I-type
            return_en = 0;       // no RR update
            dest_reg = 1'bx;     // no destination register
            write_en = 0;        // No write to reg
            ext_op = 1;          // signed extension
            write_src = 1'bx;    // No result
            alu_srcA = 1;        // from Reg_A
            alu_srcB = 0;        // immediate
            mem_w = 1;           // memory write
            mem_r = 0;           // no memory read

        end

        //BEQ, BNE :
        4'b0110 ,4'b0111: begin //only the subtraction

            fetch_en = 1;        //instruction memory on
            ir_type = 1;         // I-type
            return_en = 0;       // no RR update
            dest_reg = 1'bx;     // no destination register
            write_en = 0;        // No write to reg
            ext_op = 1;          // signed extension
            write_src = 1'bx;    // No result
            alu_srcA = 1;        // from Reg_A
            alu_srcB = 1;        // from Reg_B
            mem_w = 0;           // no memory write
            mem_r = 0;           // no memory read

        end

        //FOR :
        4'b1000: begin

            fetch_en = 1;        //instruction memory on
            ir_type = 1;         // I-type
            return_en = 1;       // RR update
            dest_reg = 1'bx;     // no destination register
            write_en = 0;        // No write to reg
            ext_op = 1;          // signed extension
            write_src = 1'bx;    // No result to write
            alu_srcA = 1;        // from Reg_A (loop target address)
            alu_srcB = 1'bx;     // (iterations) doesnt enter the ALU
            mem_w = 0;           // no memory write


         // Default values are undefined
         default: begin
            fetch_en = 1'bx;
            ir_type = 1'bx;
            return_en = 1'bx;
            dest_reg = 1'bx;
            write_en = 1'bx;
            ext_op = 1'bx;
            write_src = 1'bx;
            alu_srcA = 1'bx;
            alu_srcB = 1'bx;
            mem_w = 1'bx;
            mem_r = 1'bx;
         end
        endcase
    end
endmodule
```

Function: Decodes the opcode and func fields to generate control signals for fetching, writing, memory operations, ALU source selection and more. **RTL Behavior:**

1. **R-Type Instructions (opcode = 4'b0000)**:
   - ✓ Fetch enabled, ir_type = 0 (R-type).
   - ✓ Writes to Rd register with ALU results (write_en = 1, write_src = 1).
   - ✓ No memory operations.
2. **J-Type Instructions (opcode = 4'b0001)**:
   - ✓ Fetch enabled, ir_type = 2 (J-type).
   - ✓ Control based on func: **CALL**: Saves return address (return_en = 1). **JMP**: Fetch continues without updates. **RET**: Disables fetch (fetch_en = 0).
3. **I-Type Instructions**:
   - ✓ Fetch enabled, ir_type = 1 (I-type).
   - ✓ **ANDI, ADDI**: Writes ALU results to Rt with appropriate extension (write_src = 1, ext_op depends on the operation).
   - ✓ **LW**: Reads from memory to Rt (mem_r = 1, write_src = 0).
   - ✓ **SW**: Writes Rt data to memory (mem_w = 1).
   - ✓ **BEQ, BNE**: Branches conditionally based on zero flag (pc_ctrl set in other modules).
   - ✓ **FOR**: Iterates based on iterationsZero flag.
4. **Default Case**:
   - ✓ Outputs undefined control signals (1'bx) for invalid or unsupported instructions.

4.16 Iterations

```
module Iterations(
    input wire clk,            // Clock signal
    input wire [15:0] in,      // 16-bit input
    input wire Return_enable,  // Enable signal
    output reg [15:0] out,     // 16-bit output
    output reg Zero_iterations // Flag output: 1 if iterations reach zero
);

always @(posedge clk) begin
    if (Return_enable) begin
        out <= in;  // Load new value when enabled
    end else if (out != 0) begin
        out <= out - 1;  // Decrement value if not zero
    end

    // Set Zero_iterations flag
    if (out == 0) begin
        Zero_iterations <= 1;
    end else begin
        Zero_iterations <= 0;
    end
end

endmodule
```

Function: Tracks the number of iterations for loops and sets a flag (Zero_iterations) when the counter reaches zero.

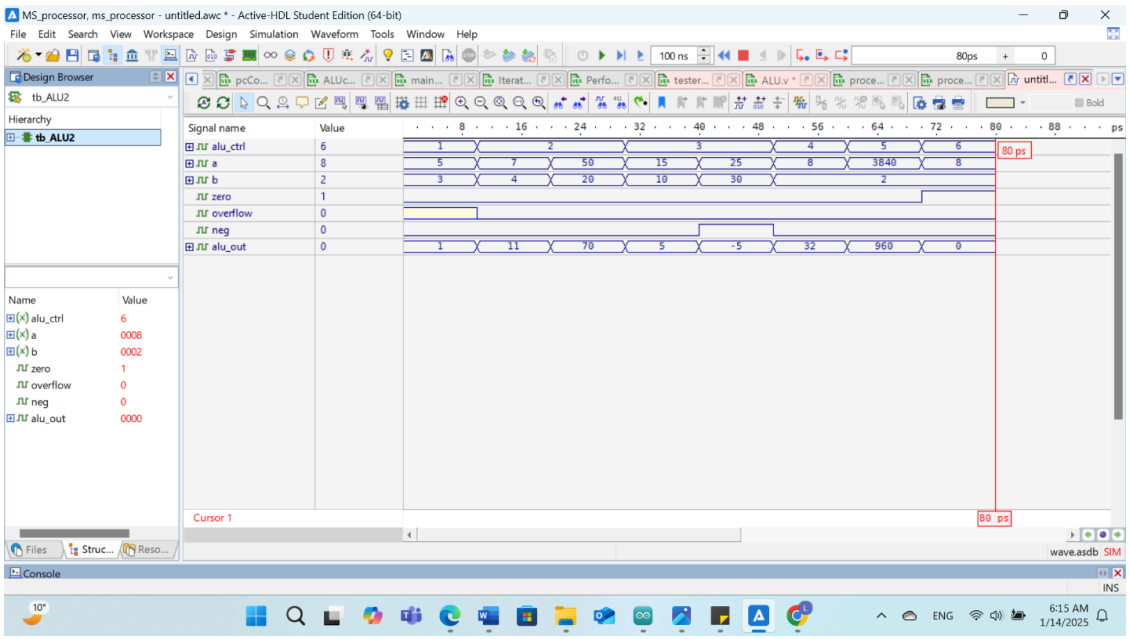
4.17 Performance registers

```
module PerformanceRegisters (

     input clk,
    input reset,
    input fetch_en,
    input [3:0] opcode,
    input [2:0] func,
    output reg [15:0] total_instructions,
    output reg [15:0] total_loads,
    output reg [15:0] total_stores,
    output reg [15:0] total_alu,
    output reg [15:0] total_controls,
    output reg [15:0] clock_cycles
);

initial begin

        total_instructions = 0;
        total_loads = 0;
        total_stores = 0;
        total_alu = 0;
        total_controls = 0;
        clock_cycles = 0;
    end


    always @(posedge clk) begin

        if (reset) begin

            total_instructions <= 0;
            total_loads <= 0;
            total_stores <= 0;
            total_alu <= 0;
            total_controls <= 0;
            clock_cycles <= 0;

        end else begin

            clock_cycles <= clock_cycles + 1;


            if (fetch_en) begin

                total_instructions <= total_instructions + 1;

                if (opcode == 4'b0000) begin //R-type instructions

                    case (func)
                        3'b000, 3'b001, 3'b010, 3'b011, 3'b100: total_alu <= total_alu + 1;

                        default: ;
                    endcase

                end else begin //I-type and J-type instructions

                case (opcode)

                    4'b0010, 4'b0011: total_alu <= total_alu + 1;  // I-type ALU => ANDI,ADDI

                    4'b0100: total_loads <= total_loads + 1;       // Load => LW

                    4'b0101: total_stores <= total_stores + 1;     // Store => SW

                    4'b0110, 4'b0111, 4'b0001: total_controls <= total_controls + 1; //control => BEQ,BNE,J-type

                        default: ;

                    endcase
                end
            end
        end
    end
endmodule
```

Function: Tracks various performance metrics of a processor, such as the total number of instructions executed, loads, stores, ALU operations, control instructions, and clock cycles.

5. Main modules simulation

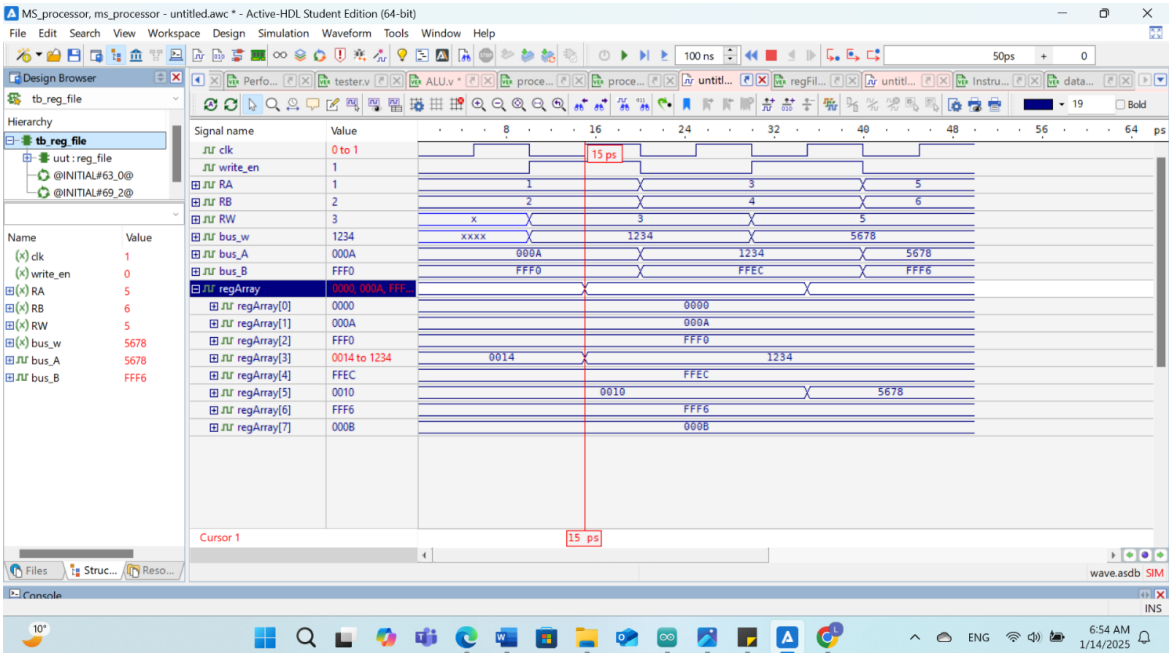
5.1 ALU Waves



➔ **Operations tested:**

```
initial begin
    $monitor("alu_ctrl=%d, a=%d, b=%d, alu_out=%d, zero=%b, overflow=%b, neg=%b",
             alu_ctrl, a, b, alu_out, zero, overflow, neg);

    //AND operation
    alu_ctrl = 3'd1; a = 16'd5; b = 16'd3; #10;

    //ADD operation
    alu_ctrl = 3'd2; a = 16'd7; b = 16'd4; #10;
    alu_ctrl = 3'd2; a = 16'sd50; b = 16'sd20; #10;

    //SUBTRACT operation
    alu_ctrl = 3'd3; a = 16'd15; b = 16'd10; #10;
    alu_ctrl = 3'd3; a = 16'sd25; b = 16'sd30; #10;

    //Logical Shift Left
    alu_ctrl = 3'd4; a = 16'b0000_0000_0000_1000; b = 2; #10;

    //Logical Shift Right
    alu_ctrl = 3'd5; a = 16'b0000_1111_0000_0000; b = 2; #10;

    //Default case
    alu_ctrl = 3'd6; a = 16'd8; b = 16'd2; #10;


    $finish;
end
```

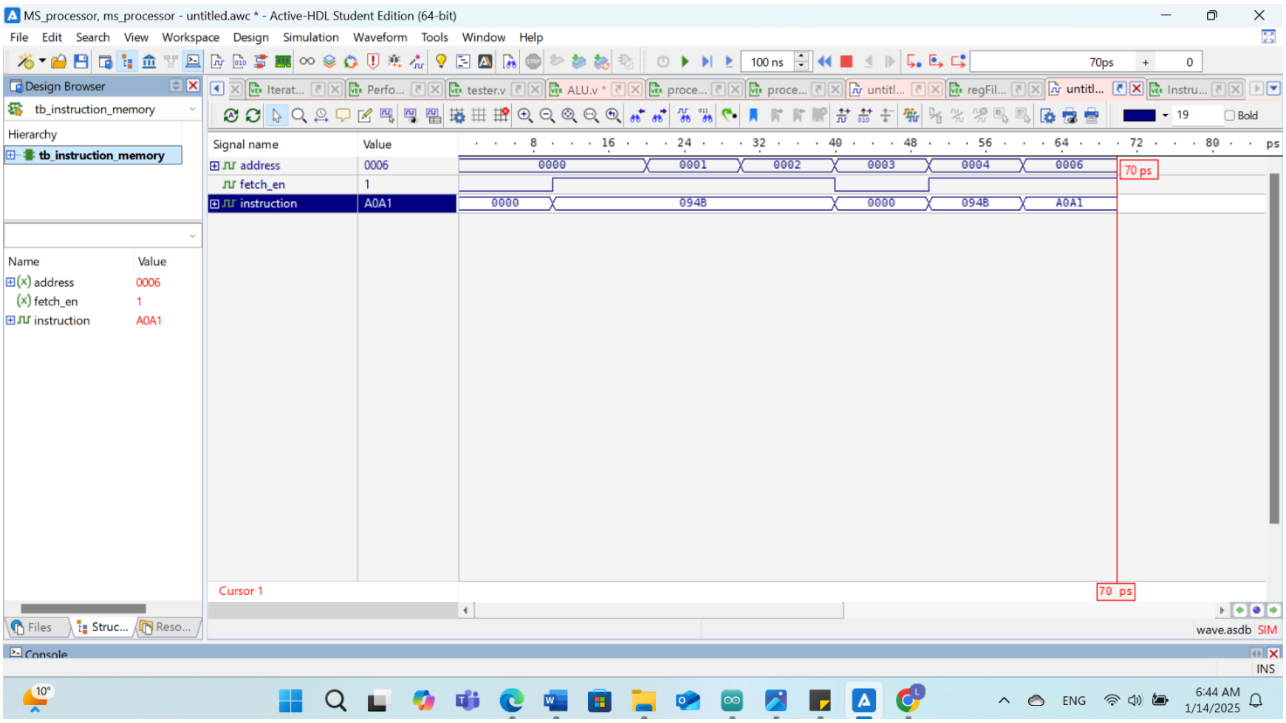
5.2 Reg_file Waves



➔ Tested Cases:

```
69
70      initial begin
71
72          $monitor("Time=%0t | RA=%d | RB=%d | RW=%d | bus_w=%h | bus_A=%h | bus_B=%h",
73                   $time, RA, RB, RW, bus_w, bus_A, bus_B);
74
75          //Initial read (no write)
76          write_en = 0;
77          RA = 3'd1; // Read register 1
78          RB = 3'd2; // Read register 2
79          #10;
80
81          // Write to register 3
82          write_en = 1;
83          RW = 3'd3;
84          bus_w = 16'h1234;
85          #10;
86
87          //Disable write and read back register 3
88          write_en = 0;
89          RA = 3'd3;
90          RB = 3'd4; // Read register 4
91          #10;
92
93          // Write to register 5
94          write_en = 1;
95          RW = 3'd5;
96          bus_w = 16'h5678;
97          #10;
98
99          //Disable write and read back register 5
00          write_en = 0;
01          RA = 3'd5;
02          RB = 3'd6; // Read register 6
03          #10;
04
05
06          $finish;
07      end
```

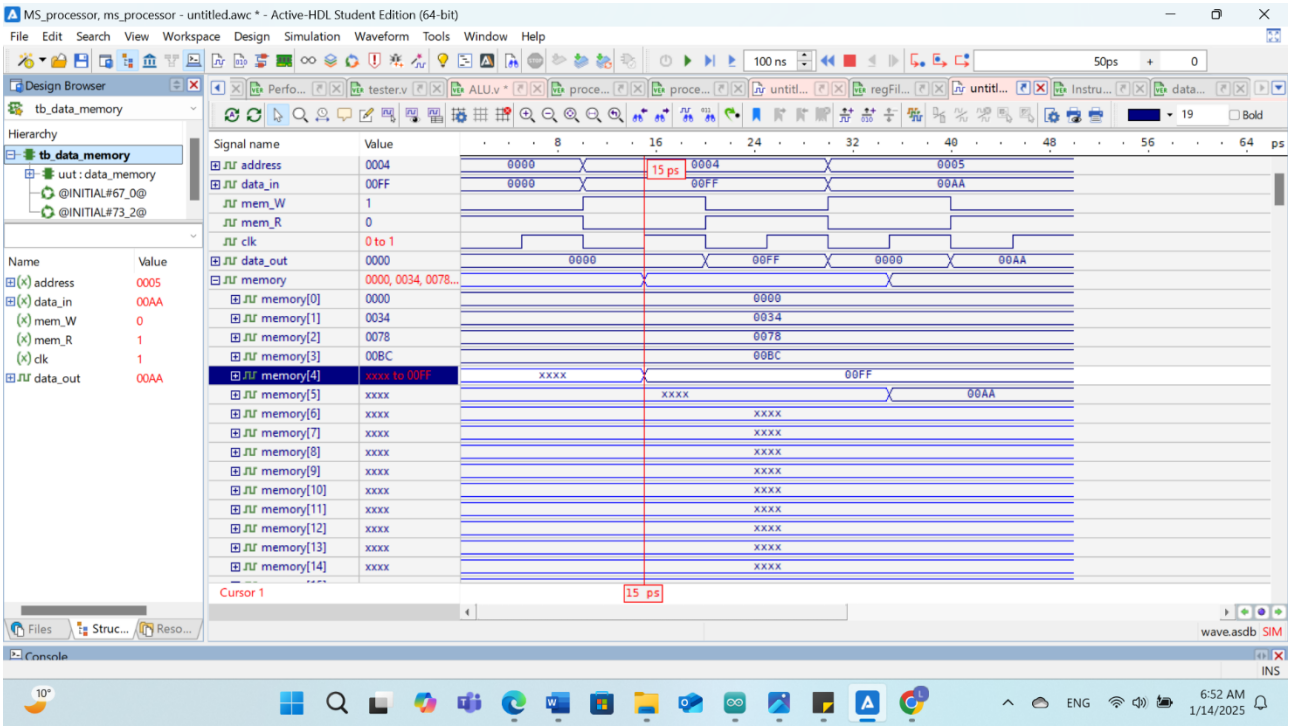
5.3 Instruction memory Waves



➔ **Tested Cases:**

```
60      initial begin
61          $monitor("Time=%0t | Address=%d | Fetch Enable=%b | Instruction=%h",
62                   $time, address, fetch_en, instruction);
63
64          // Initialize inputs
65          fetch_en = 0;
66          address = 16'd0;
67
68          // Test 1: Fetch instruction at address 0
69          #10 fetch_en = 1;
70          address = 16'd0;
71          #10;
72
73          // Test 2: Fetch instruction at address 1
74          address = 16'd1;
75          #10;
76
77          // Test 3: Fetch instruction at address 2
78          address = 16'd2;
79          #10;
80
81          // Test 4: Disable fetch and check output
82          fetch_en = 0;
83          address = 16'd3;
84          #10;
85
86          // Test 5: Fetch instruction at address 4
87          fetch_en = 1;
88          address = 16'd4;
89          #10;
90
91          // Test 6: Fetch instruction at address 6
92          address = 16'd6;
93          #10;
94
95          // Finish simulation
96          $finish;
97      end
```

5.4 Data memory Waves



➔ Tested Cases:

```
    initial begin
        $monitor("Time=%0t | Address=%d | Data In=%h | Mem Write=%b | Mem Read=%b | Data Out=%h",
                 $time, address, data_in, mem_W, mem_R, data_out);

        // Initialize inputs
        address = 16'd0;
        data_in = 16'h0000;
        mem_W = 0;
        mem_R = 0;

        // Test 1: Read from address 0
        mem_R = 1;
        address = 16'd0;
        #10;

        // Test 2: Write to address 4
        mem_R = 0;
        mem_W = 1;
        address = 16'd4;
        data_in = 16'h00FF;
        #10;

        // Test 3: Read from address 4
        mem_W = 0;
        mem_R = 1;
        address = 16'd4;
        #10;

        // Test 4: Write to address 5
        mem_R = 0;
        mem_W = 1;
        address = 16'd5;
        data_in = 16'h00AA;
        #10;

        // Test 5: Read from address 5
        mem_W = 0;
        mem_R = 1;
        address = 16'd5;
        #10;
```

6. Simulation and testing

6.1 Instruction Set

A Set of ten dependent and independent instructions of different types have been implemented in the instruction memory to test the execution of the processor , the instructions implemented are viewed in the figure down below :

```
    initial begin
    // R-type instructions:
    instruction_array[0] = {Constants.ADD, 3'b000, 3'b000, 3'b000, 3'b001};

    instruction_array[1] = {Constants.SW , 3'b010, 3'b001, 6'b000010};
    instruction_array[2] = {Constants.LW, 3'b011, 3'b101, 6'b000000};
    instruction_array[3] = {Constants.ADD, 3'b101, 3'b011, 3'b101, 3'b001};

    // I-type instructions:
    instruction_array[4] = {Constants.ADDI, 3'b010, 3'b101, 6'b000011};
    instruction_array[5] = {Constants.BEQ, 3'b111, 3'b110, 6'b000010};


    // J-type instructions:
    instruction_array[6] = {Constants.ADDI, 3'b010, 3'b011, 6'b000010};
    instruction_array[7] = {4'b0001, 9'b000001001, 3'b001};
    instruction_array[8] = {Constants.SUB, 3'b110, 3'b010, 3'b111, 3'b010};
    instruction_array[9] = {Constants.SLL, 3'b101, 3'b011, 3'b101, 3'b011};
    instruction_array[10] = {4'b0001, 9'b000001001, 3'b010};
end
```

➔ The Values of the Registers initially are :

```
reg [15:0] regArray [0:7] = '{

    16'h0000, //R0
    16'h000A, //R1
    16'h0002, //R2
    16'h0004, //R3
    16'h0003, //R4
    16'h0000, //R5
    16'h000B, //R6
    16'h000B  //R7
```

➔ Thus, The Instructions sequentially are clarified to be :

0. **Dummy Instruction**
1. **SW R1 , 2(R2)**
2. **LW R5, 0 (R3)**
3. **ADD R5,R3,R5**
4. **ADDI R5,R2, 3**
5. **BEQ R7, R6 , 2**
6. **ADDI R3,R2, 2**
7. **Call 9**
8. **SUB R6, R2, R7**
9. **SLL R5,R3,R5**
10. **RET**

6.2 Test Bench Structure

After implementing the instructions, all these instructions were tested by implementing a test bench, as viewed down below:

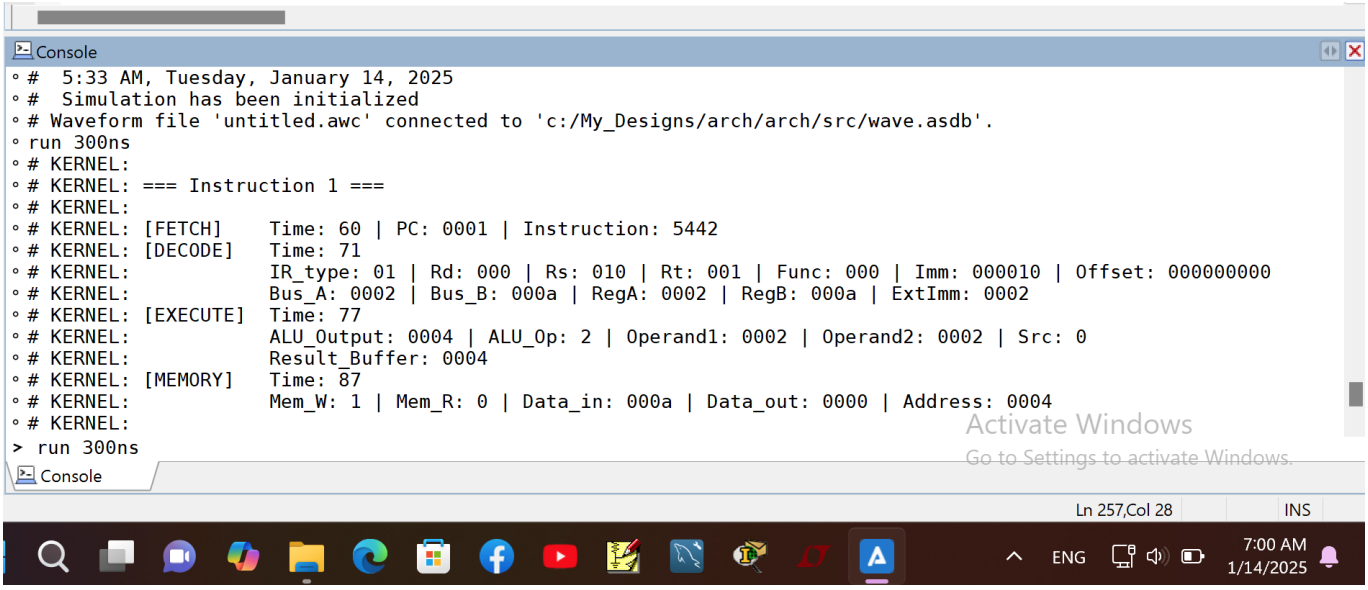
```
module datapath_tb2;

// Declare testbench variables
reg clk;
wire [15:0] current_pc, instruction_out, next_pc, pc_out1, JTA, Bus_w, B, A, regA, regB, extimm, alu_opernad2;
wire signed [15:0] alu_output, result_buffer;
wire [3:0] opcode;
wire [2:0] alu_ctrl, pc_ctrl;
wire [11:0] reg1_rest;
wire [1:0] IR_type;
wire [2:0] func, Rd, Rt, Rs, Rw;
wire [5:0] imm;
wire [8:0] offset;
wire zero, write_src, fetch_enable, dst_reg, write_enable, ext_op, alu_src, mem_W, mem_R, return_en, zero_iterations;

// Instantiate the module under test (MUT)
datapath uut (
    .clk(clk)
);

// Generate clock signal with a period of 10 time units (5ns high, 5ns low)
initial begin
    clk = 0;
    forever #5 clk = ~clk; // 10 time unit clock period
end

// Stimulus and sequential monitoring
initial begin
    #57; // Skip the dummy instruction

    // Loop through multiple instructions
    repeat (9) begin
        // Fetch stage
```

```
#11; // Move to Decode

// Decode stage
$display("[DECODE]   Time: %0t", $time);
$display("           IR_type: %b | Rd: %b | Rs: %b | Rt: %b | Func: %b | Imm: %b | Offset: %b",
         uut.IR_type, uut.Rd, uut.Rs, uut.Rt, uut.func, uut.imm, uut.offset);
$display("           Bus_A: %h | Bus_B: %h | RegA: %h | RegB: %h | ExtImm: %h",
         uut.A, uut.B, uut.regA, uut.regB, uut.extimm);

// Execute stage ((for specific instruction types)
if (uut.opcode != 4'b0001) begin
    #6;
    $display("[EXECUTE]  Time: %0t", $time);
    $display("           ALU_Output: %h | ALU_Op: %h | Operand1: %h | Operand2: %h | Src: %h",
             uut.alu_output, uut.alu_ctrl, uut.regA, uut.alu_opernad2, uut.alu_src);
    #5;
    $display("           Result_Buffer: %h", uut.result_buffer);
end


if (uut.opcode == 4'b0001 && uut.func == 3'b010) begin
    #11;
end

// Memory stage (for specific instruction types)
if (uut.opcode != 4'b0000 && uut.opcode != 4'b0010 && uut.opcode != 4'b1000 &&
    uut.opcode != 4'b0011 && uut.opcode != 4'b0111 && uut.opcode != 4'b0110 && uut.opcode != 4'b0001) begin
    #5;
    $display("[MEMORY]   Time: %0t", $time);
    $display("           Mem_W: %b | Mem_R: %b | Data_in: %h | Data_out: %h | Address: %h",
             uut.mem_W, uut.mem_R, uut.regB, uut.memo_out, uut.result_buffer);
    #5;
end

// Write-back stage (for specific instruction types)
if (uut.opcode != 4'b0101 && uut.opcode != 4'b0111 && uut.opcode != 4'b0110 &&
    uut.opcode != 4'b1000 && uut.opcode != 4'b0001) begin
    #5;
    $display("[WRITEBACK]Time: %0t", $time);
    $display("           Write_en: %b | Bus_W: %h | Destination_Reg=R%0d", uut.write_enable, uut.Bus_w, uut.Rw);
    #5;
end

   // Display Performance Registers
   $display("\n=== Performance Registers ===");
   $display("Total Instructions: %d", uut.total_instructions);
   $display("Total Loads:        %d", uut.total_loads);
   $display("Total Stores:       %d", uut.total_stores);
   $display("Total ALU Ops:      %d", uut.total_alu);
   $display("Total Controls:     %d", uut.total_controls);
   $display("Clock Cycles:       %d", uut.clock_cycles);

   // End the simulation
   $finish;
end

endmodule
```

The simulation begins with a clock signal that toggles every 5 time units, driving the synchronous operations in the module. After an initial delay to skip a dummy instruction, a loop iterates through 9 instructions, emulating the pipeline stages. For each instruction, the testbench displays detailed information about the current state, including the program counter, instruction content, control signals, register values, ALU outputs, and memory interactions. Conditional checks determine whether certain stages (such as Execute or Memory) are skipped based on the opcode. After completing the loop, the testbench prints performance statistics, such as the total number of instructions executed, loads, stores, ALU operations, control instructions, and clock cycles

6.3 Test Bench Execution Output

➔ Instruction One : SW R1 , 2(R2)

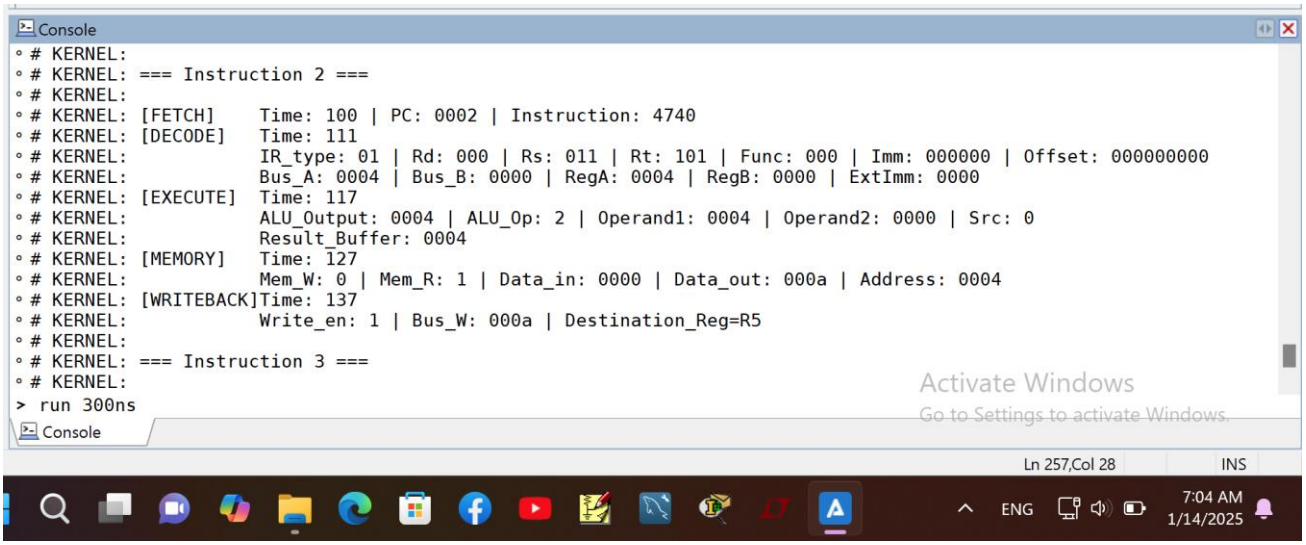


The console output demonstrates the correct functioning of the datapath module across the stages. During the Fetch stage, the instruction 5442 “SW R1 , 2(R2)” is fetched from PC = 0001. In the Decode stage, the instruction is correctly interpreted as an I-Type with the fields (IR_type = 01, Rd = 000, Rs = 010, Rt = 001, Imm = 000010), and the appropriate register values (Bus_A = 0002, Bus_B = 000A, ExtImm = 0002) are routed.

The Execute stage performs the ALU operation (ALU_Op = 2), producing the expected result of 0004 by adding operands 0002 and 0002. In the Memory stage, a write operation occurs with data (Data_in = 000A) written to address 0004, as indicated by control signals (Mem_W = 1, Mem_R = 0), therefore a successful four stage store instruction has been executed.

➔ Instruction Two: LW R5, 0 (R3):

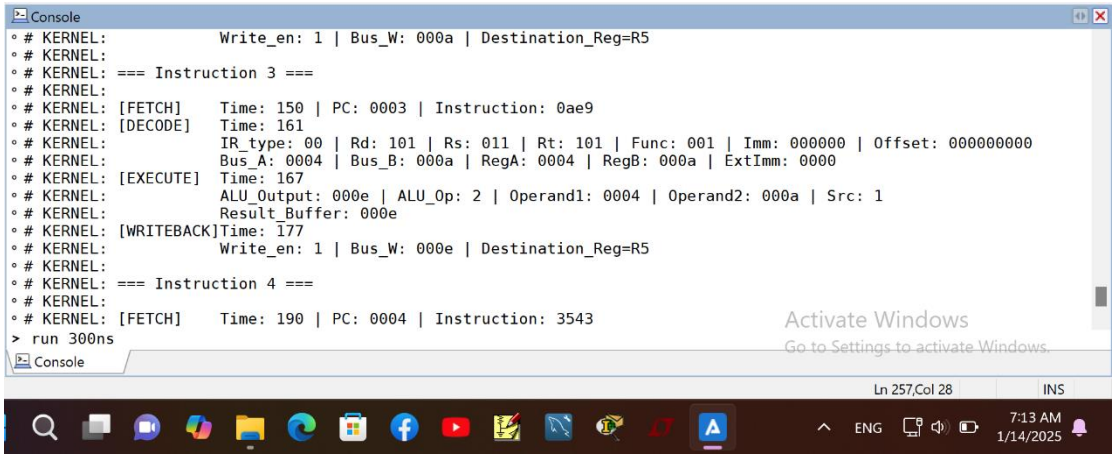
To Ensure that the previous store operation was executed successfully, the same address written to , is loaded from in this instruction:



The load instruction (h'4740) LW R5, 0 (R3) is successfully executed as a multicycle operation, demonstrating proper retrieval of the value stored during the previous store instruction, since in

- the Fetch stage, the instruction is fetched at PC = 0002,
- and in the Decode stage, the base address (0004) from Rs and an offset of 0 are extracted.
- The Execute stage calculates the effective address (0004) using the base address and offset.
- In the Memory stage, a read operation retrieves the stored value (000A) from address 0004. Finally, in the Write-Back stage, the value (000A) is written to register R5 with write-back enabled. This successful execution confirms that the datapath correctly handles memory read and register update operations.

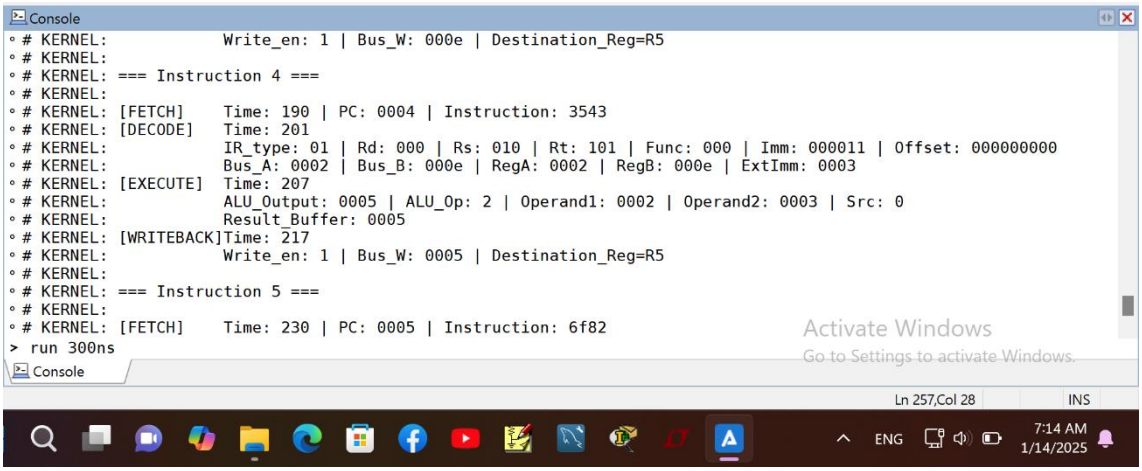
➔ Instruction three : ADD R5,R3,R5



The ADD R5, R3, R5 instruction (0A09) is successfully executed:

- In the Fetch stage, the instruction is fetched from PC = 0003.
- In the Decode stage, the fields are correctly interpreted as an R-Type ADD operation, with R3 (0004) and R5 (000A) being the same register loaded to in the previous section
- The Execute stage performs the addition in the ALU, producing the correct result (000E) for 4 + 10 = 14
- Finally, in the Write-Back stage, the result (000E) is written back to the destination register R1 with write-back enabled. The control signals and data flow through all stages operate seamlessly, confirming the success of the instruction execution

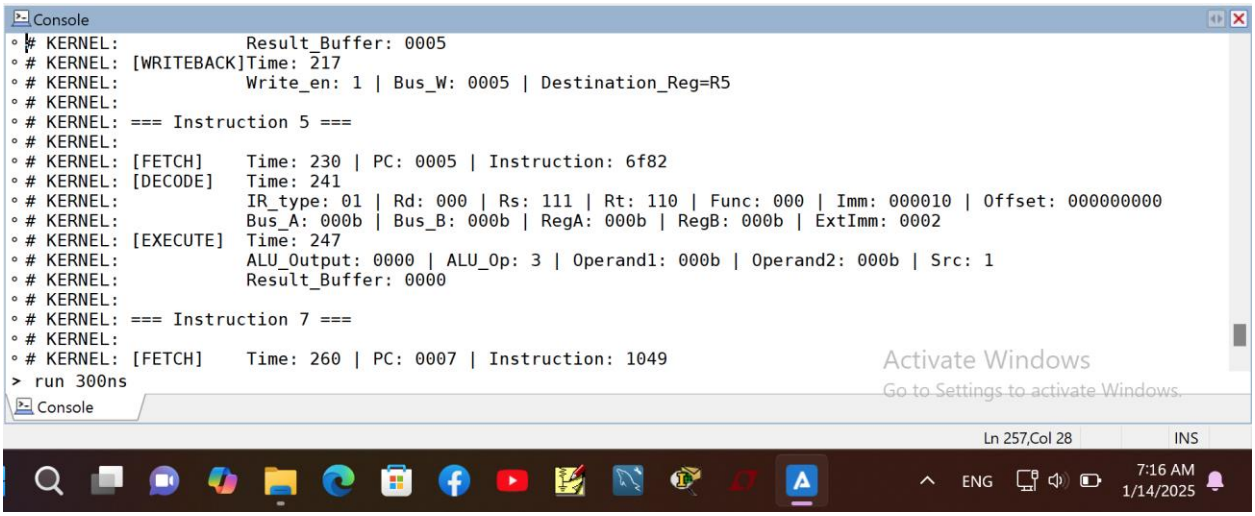
➔ Instruction four : ADDI R5,R2, 3



- In the Fetch stage, the instruction is fetched from PC = 0004.
- In the Decode stage, it is identified as an I-Type instruction where R2 (0002) is the source register, and the immediate value (0003) is extended.
- The Execute stage performs the addition operation in the ALU, adding R2 (0002) and the immediate value (0003), resulting in 0005.

In the Write-Back stage, the result (0005) is written back to the destination register R5 with write-back enabled. The correct decoding, computation, and register update confirm the successful execution of this I type instruction.

➔ Instruction Five: BEQ R7, R6 , 2



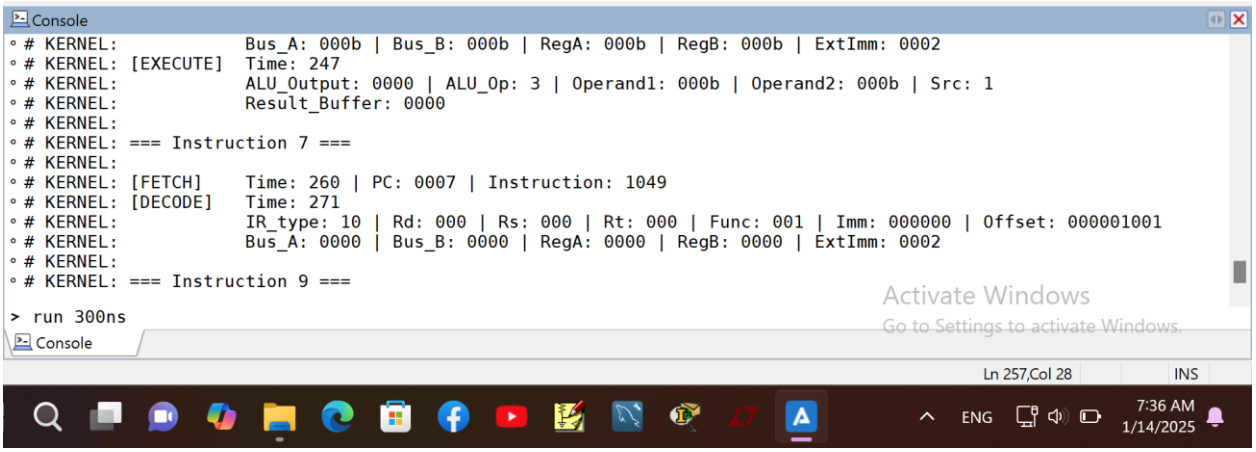
The BEQ R7, R6, 2 instruction (6782) executes as a branch instruction where:

- In the Fetch stage, the instruction is fetched from PC = 0005.
- In the Decode stage, it is identified as an I-Type branch instruction, where R7 (000B) and R6 (000B) are the source registers to compare, and the immediate value (2) is extended to calculate the branch offset.
- In the Execute stage, the ALU determines that R7 is equal to R6 (000B == 000B), by subtracting the two numbers so the branch is taken.

- As a result, the PC changes to 0007 (current PC(5) + 2) based on the branch offset. Thus the next instruction will be instruction number 7 not 6

➔ Instruction Six: ADDI R3,R2, 2 (Branched Over Skipped)

➔ Instruction Seven: Call 9

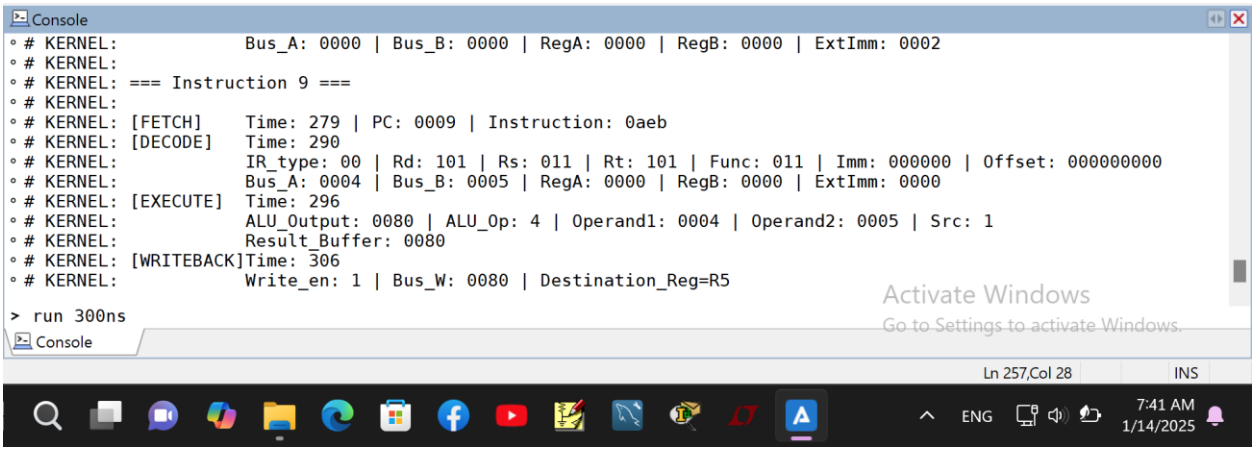


The CALL 9 instruction (1049) executes as a jump-and-link instruction:

- In the Fetch stage, the instruction is fetched from PC = 0007.
- In the Decode stage, it is identified as a J-Type instruction, with the immediate value (9) being the jump target address. Additionally, the return address (PC + 1 = 0008) is prepared to be saved in the return register.
- In the Execute stage, the effective jump address (9) is calculated, and the program counter (PC) is updated to 0009. The return address (0008) is stored for use in the return register when returning from the subroutine using RET.

➔ Instruction Eight : SUB R6, R2, R7 (Jumped Over)

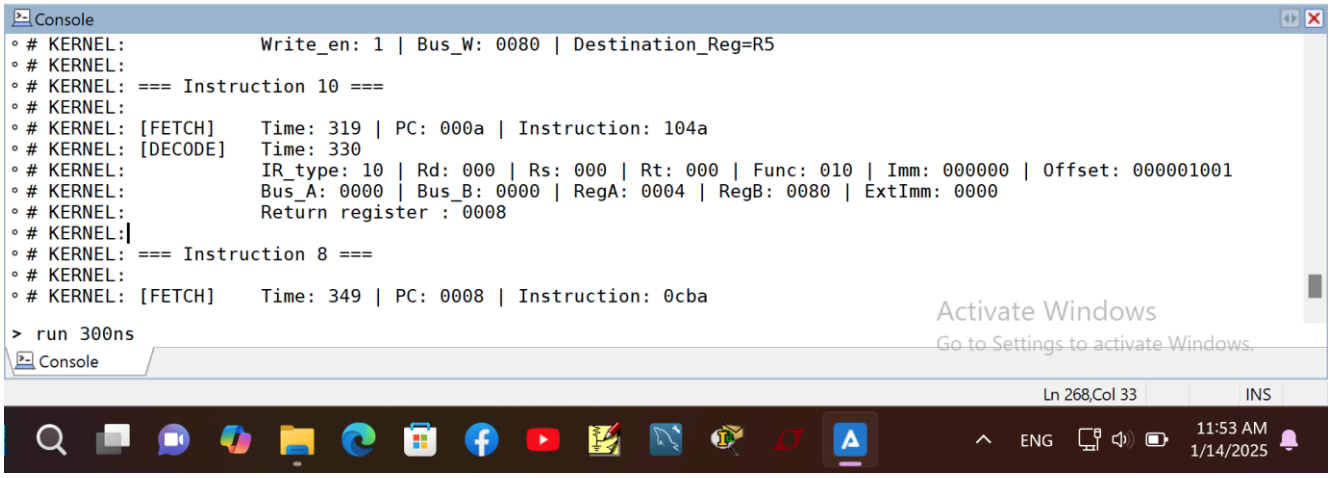
➔ Instruction Nine: SLL R5,R3,R5



The SLL R5, R3, R5 instruction (0AEB) executes successfully:

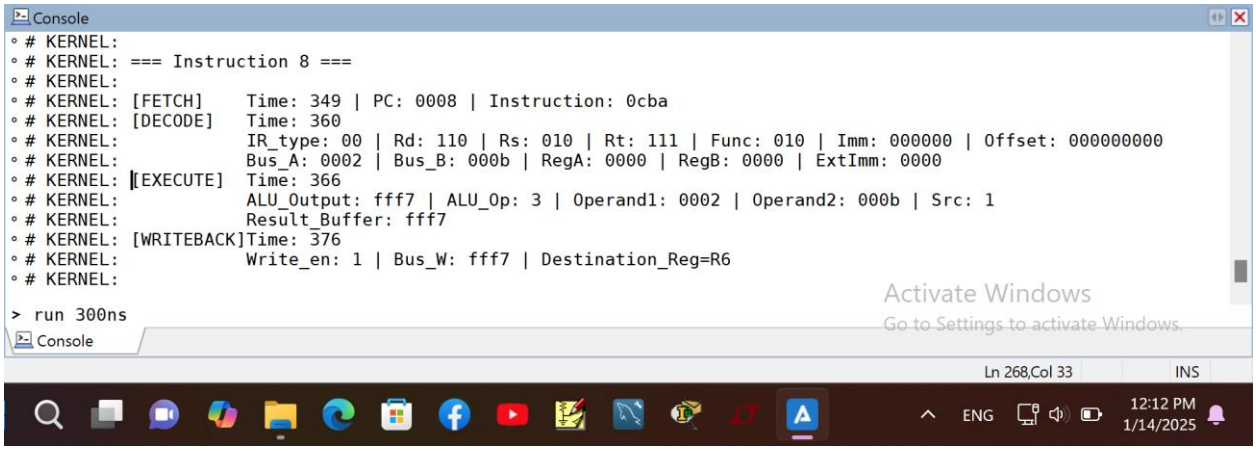
- In the Fetch stage, the instruction is fetched from PC = 0009. In the Decode stage, it is identified as an R-Type instruction, with R3 (0004) as the first source operand, R5 (0005) as the second source operand, and R5 as the destination register.
- The shift operation is specified by Func = 011. In the Execute stage, the ALU performs the shift-left logical operation, using R3 (0004) and the shift amount (0005), producing the result (0x0080).
- In the Write-Back stage, the result (0x0080) is written back to the destination register R5 with write-back enabled. This confirms the successful execution of the shift-left logical operation.

➔ Instruction Ten: RET



- The RET instruction (104A) is successfully executed as a return-from-subroutine operation.
- In the Fetch stage, the instruction is fetched from PC = 000A.
- In the Decode stage, it is identified as a J-Type instruction with the return address (0008) retrieved from the return register.
- The Execute stage prepares to update the program counter (PC) to the return address (0008). As a result, control is transferred back to the instruction following the original CALL instruction.
- This confirms the successful execution of the return operation, ensuring proper subroutine functionality by jumping back to the stored return address

➔ Instruction Eight : SUB R6, R2, R7 (Returned to)



The SUB R6, R2, R7 instruction (0CBA) executes successfully after returning to it from the subroutine.

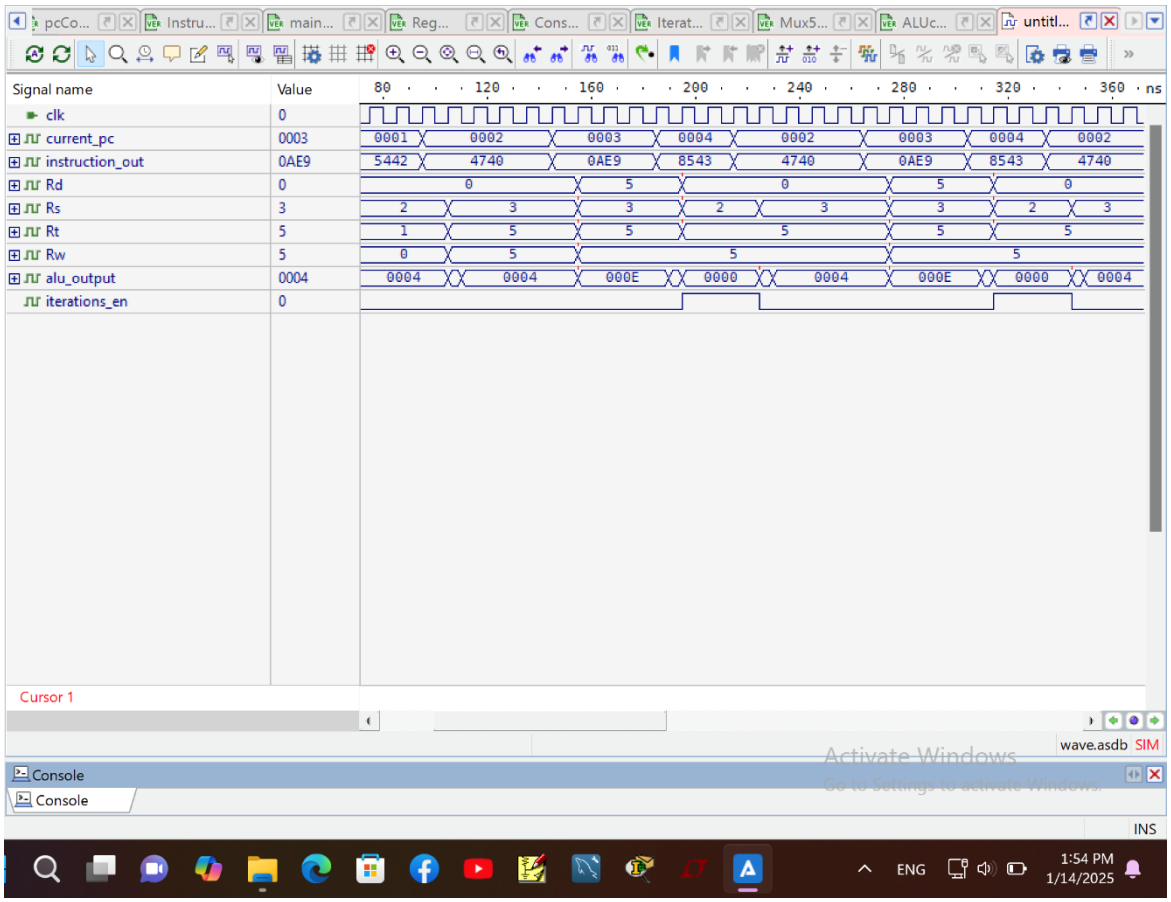
- In the Fetch stage, the instruction is fetched from PC = 0008.
- In the Decode stage, it is identified as an R-Type instruction with R2 (0002) and R7 (000B) as the source registers, and R6 as the destination register. The function code (010) specifies the subtraction operation. I
- n the Execute stage, the ALU performs the subtraction operation (R2 - R7), resulting in FFF7 (signed result of 2 - 11) which is -9.
- In the Write-Back stage, the result (FFF7) is written back to the destination register R6 with write-back enabled. This confirms the successful execution of the subtraction operation following the return from the subroutine.

6.4 Test Bench Execution Output after adding FOR

The Fourth instruction in the instruction file , was replaced by the FOR to ensure its successful execution, the new instruction Set :

0. **Dummy Instruction**
1. **SW R1 , 2(R2)**
2. **LW R5, 0 (R3)**
3. **ADD R5,R3,R5**
4. **FOR R5,R2**
5. **BEQ R7, R6 , 2**
6. **ADDI R3,R2, 2**
7. **Call 9**
8. **SUB R6, R2, R7**
9. **SLL R5,R3,R5**
10. **RET**

➔ **Execution Output:**

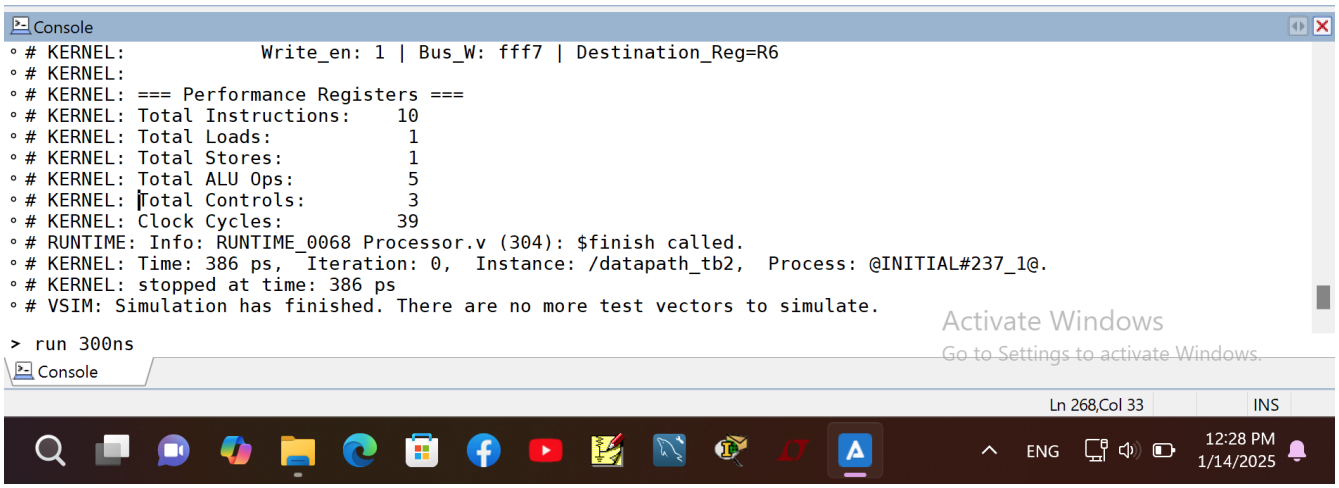


- As observed from the previous waveform when the PC reached the FOR instruction (PC=4), the system branched to PC=2 , since Rs stores the loop target address, the address of the first instruction in the loop block, and in this case Rs = R5 = 2

- After branching the Program continues executing until it reaches the FOR again (PC=4), to repeat the process multiple times until the number of iterations held in Rt= R3=3 , are decremented to zero , which are kept in the iterations Register and only accesses it when FOR is reached by an enable called iterations_en as viewed in the previous waveform.

6.5 Performance Registers Output

After executing the instructions , the contents of the performance register files are displayed :



From the previous figure;

- **Total executed Instructions : 10**

Dummy Instruction
SW R1 , 2(R2)
LW R5, 0 (R3)
ADD R5,R3,R5
ADDI R5,R2, 3
BEQ R7, R6 , 2
ADDI R3,R2, 2
Call 9
SUB R6, R2, R7
SLL R5,R3,R5
RET

In this Testbench , Only the instructions in red were executed

-Which are 10 instructions

- **Total Loads : 1**

Dummy Instruction
SW R1 , 2(R2)
LW R5, 0 (R3)
ADD R5,R3,R5
ADDI R5,R2, 3
BEQ R7, R6 , 2
ADDI R3,R2, 2
Call 9
SUB R6, R2, R7
SLL R5,R3,R5
RET

Only one Load instruction

- **Total Stores: 1**

Dummy Instruction
SW R1 , 2(R2)
LW R5, 0 (R3)
ADD R5,R3,R5
ADDI R5,R2, 3
BEQ R7, R6 , 2
ADDI R3,R2, 2
Call 9
SUB R6, R2, R7
SLL R5,R3,R5
RET

Only one Store instruction

- **Total ALU ops : 5**

Dummy Instruction
SW R1 , 2(R2)
LW R5, 0 (R3)
ADD R5,R3,R5
ADDI R5,R2, 3
BEQ R7, R6 , 2
ADDI R3,R2, 2 "wasn't executed"
Call 9
SUB R6, R2, R7
SLL R5,R3,R5
RET

Since the dummy instruction is

ADD R0,R0,R0

The Alu operations are 5

- **Total control ops : 3**

Dummy Instruction
SW R1 , 2(R2)
LW R5, 0 (R3)
ADD R5,R3,R5
ADDI R5,R2, 3
BEQ R7, R6 , 2
ADDI R3,R2, 2
Call 9
SUB R6, R2, R7
SLL R5,R3,R5
RET

Branch , call and return

The control operations are 3

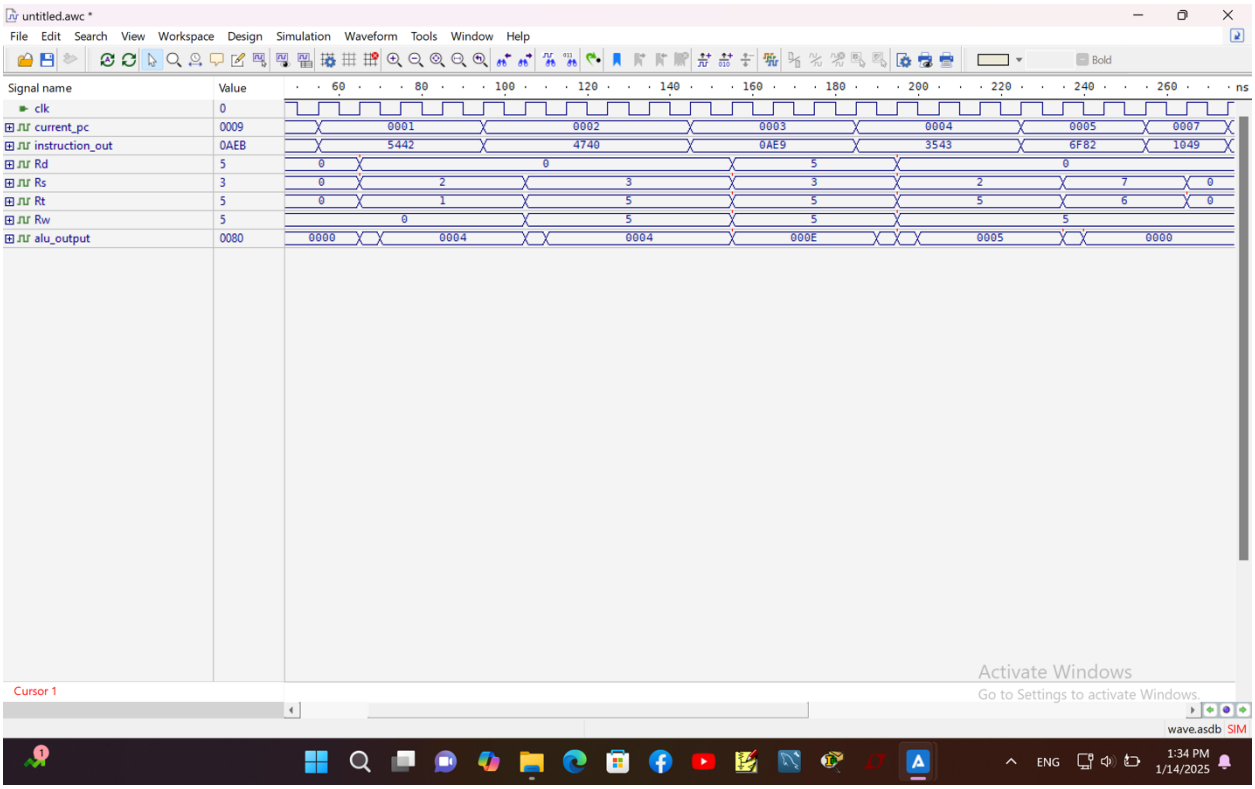
- **Total clock cycles : 39**

Dummy Instruction :6 cycles
SW R1 , 2(R2) :4 cycles
LW R5, 0 (R3) :5 cycles
ADD R5,R3,R5 :4 cycles
ADDI R5,R2, 3 :4 cycles
BEQ R7, R6 , 2 :3 cycles
ADDI R3,R2, 2 :"not executed"
Call 9 :2 cycles
SUB R6, R2, R7. :4 cycles
SLL R5,R3,R5 :4 cycles
RET :3 cycles

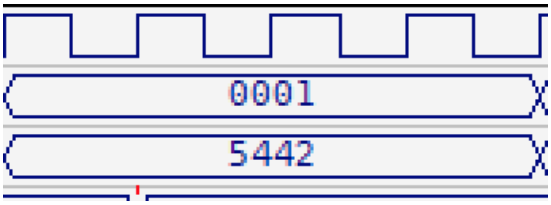
Cycles = 5*4+ 5+ 3+3+2+6 = 39 cycles

6.6 Multi-Cycle Scheduling

To Ensure that the processor in hand performs as a multi-cycle machine , the cycles of each instruction types is observed :



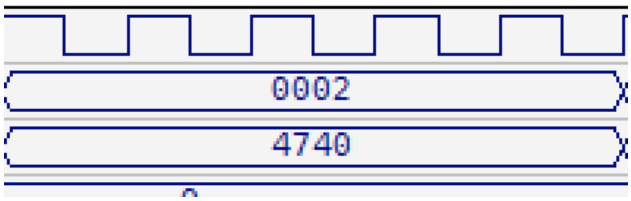
- **Store Instruction** at PC =1 , from the previous figure this instruction requires **four clock cycles**



Which is correct since this instruction requires 4 stages to execute:

- Fetch , Decode, Execute, Memory

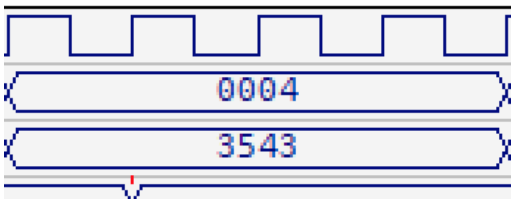
- **Load Instruction** at PC =2, from the previous figure this instruction requires **five clock cycles**



Which is correct since this instruction requires 5 stages to execute:

- Fetch , Decode, Execute, Memory ,Write Back

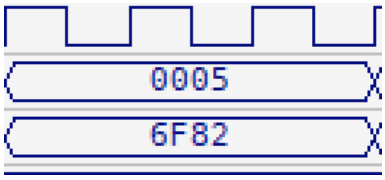
- **Alu operations Instructions** at PC =3 (ADD), from the previous figure this instruction requires **4 clock cycles**



Which is correct since this instruction requires 4stages to execute:

- Fetch, Decode, Execute, Memory

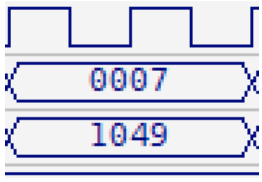
- **Branch Instruction** at PC =5 , from the previous figure this instruction requires **3 clock cycles**



Which is correct since this instruction requires 3 stages to execute:

- Fetch , Decode, Execute

- **J- type Instructions** at PC =7, from the previous figure this instruction requires **2 clock cycles**



Which is correct since this instruction requires 2 stages to execute:

- Fetch , Decode

7. Conclusion

To conclude, this project successfully demonstrates the design and verification of a 16-bit multicycle RISC processor. The processor includes a 16-bit program counter (PC), eight 16-bit general-purpose registers, a 16-bit return register (RR) for function calls, and performance registers to monitor program execution metrics. The architecture uses word-addressable memories, with separate physical memories for data and instructions, and supports R-, I-, and J-type instructions.

The multicycle design executes each instruction through distinct steps: fetch, decode, execute, memory access, and write-back. Boolean equations and control signals were developed to ensure the proper functioning of both the datapath and control path. Verification was achieved through a comprehensive testbench for the primary structural module and waveform simulations for each module.

Active-HDL was employed for RTL design and simulation, while Canva assisted the creation of detailed datapath diagrams. This report highlights the processor's design, implementation, and testing, providing valuable insights through supporting diagrams and test cases, ultimately validating the functionality of the proposed architecture.

8. References

1] https://www.geeksforgeeks.org/multi-cycle-data-path-and-control/