



**Faculty Of Engineering And Technology
Electrical and Computer Engineering Department**

**Computer Architecture
ENCS4370**

Project.No.2-Report.No.1

**Title:
Design and Verification of a Simple Pipelined RISC
Processor in Verilog**

Instructor: Dr. Ayman Hroub .

Date of submission: 9/1/2025

Prepared by:

Member : Donia Alshiakh

ID : 1210517

Member : Shahd Yahya

ID : 1210249

2024-2025

1.Design and Implementation:

In this project, we aim to design and implement a multi-cycle processor capable of executing a variety of instruction types efficiently. The processor is based on a modular design that separates functionality into distinct stages, including Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). Each stage is carefully designed to ensure proper handling of instructions while maximizing the reuse of components such as the ALU and memory.

Our approach ensures that the processor can handle a range of R-Type, I-Type, and J-Type instructions, as well as control instructions like branches and jumps. By employing a multi-cycle architecture, we minimize hardware redundancy while maintaining the correctness and flexibility needed to execute instructions accurately.

This section details the design and implementation of the Datapath and control unit, including the components, control signals, state diagrams, and truth tables. We outline the decisions made during the design process, justify our choices, and provide a comprehensive analysis of the overall functionality and correctness of the processor. Finally, we highlight the importance of proper integration and testing to ensure the successful operation of the entire system.

1.1 Datapath Component:

1. Instruction Memory:

The **Instruction Memory** is a 16-bit wide, word-addressable memory module responsible for storing and providing program instructions to the processor. Being word-addressable means that each memory cell stores **2 bytes** or **16 bits** of data. It operates during the **Instruction Fetch (IF)** stage to retrieve the current instruction based on the value of the **Program Counter (PC)**.

- **Size:** Each memory cell stores 16 bits (2 bytes) per instruction.
- **Input:** The 16-bit address from the **Program Counter (PC)**.
- **Output:** A 16-bit instruction fetched from the memory.
- **Purpose:** To supply the current instruction for decoding and execution in subsequent stages.
- **Connection:** The output of the **Instruction Memory** is forwarded to the **Instruction Decode (ID)** stage, where the instruction is split into opcode, register addresses, and immediate values.

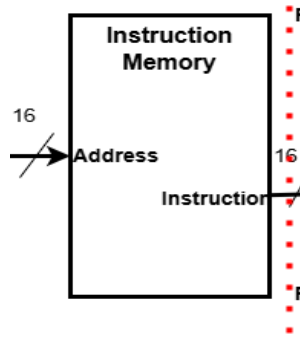


Figure 1: Instruction Memory Block Diagram

2. Register File:

The **Register File** is a crucial part of our processor's Datapath, designed to store and provide operand values during instruction execution. It consists of ten 16-bit registers: eight general-purpose registers and two special-purpose registers. We structured the Register File to support simultaneous read and write operations, ensuring efficient execution of instructions and minimizing delays.

We designed the Register File to include several inputs and outputs. For inputs, it accepts two 3-bit register addresses ((Rs or RR) and (Rt or SR)) for reading, a 3-bit destination register address (Rd, Rt, SR, RR) for writing, a 16-bit write data input (RegDest), and a control signal (RegWrite) to enable writing. The outputs consist of two 16-bit values (ReadData1 and ReadData2), which correspond to the data stored in the registers addressed by Rs and Rt.

To enhance functionality, we included two special-purpose registers. **Register[8]** is dedicated to the **FOR instruction**, storing the loop counter (Rt - 1) after each iteration. This design simplifies loop control without the need for additional computations. **Register[9]**, known as the **Return Register (RR)**, is used during function calls and returns. It stores the return address when executing a **CALL instruction** and restores the program counter (PC) during a **RET instruction**, ensuring seamless function execution.

During the **Instruction Decode (ID)** stage, the Register File reads data from the registers addressed by Rs and Rt. In the **Write Back (WB)** stage, it updates the destination register (Rd, Rt, or a special-purpose register) based on the RegWrite control signal. The data written back may come from the ALU, memory, or the program counter, depending on the instruction type. The ReadData1 and ReadData2 outputs are forwarded to the ALU or other units for further processing, while the RegDest a input is sourced from the relevant component based on the

instruction requirements. By integrating these features, we ensured that the Register File efficiently supports general-purpose execution while enabling advanced functionality like loops and function calls. These additions significantly enhance the processor's ability to handle more complex operations seamlessly.

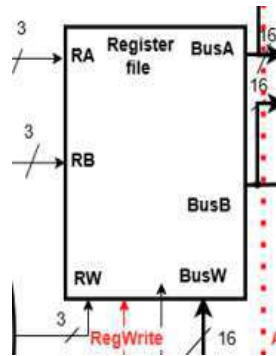


Figure 2: Register File Block Diagram

3. Arithmetic Logic Unit (ALU):

The **Arithmetic Logic Unit (ALU)** is one of the most critical components in our processor's Datapath, as it handles all arithmetic and logical operations. It processes two 16-bit input operands and produces a 16-bit output result, ensuring compatibility with the overall architecture. Additionally, the ALU generates a **Zero flag** to indicate if the result of the operation is zero, and the ALU generates a **for flag** to indicate if the result of the second input is zero, which is essential for branch instructions like BEQ (branch if equal). We designed the ALU to perform various operations, including addition, subtraction, bitwise AND, bitwise OR, and logical shifting, enabling the processor to handle all supported instructions efficiently. In our implementation, the ALU operates during the Execution (EX) stage. It receives inputs either from the Register File or extended immediate values, depending on the control signals. The specific operation is determined by the ALUOp control signal and, for R-type instructions, refined further by the Funct field. The output is then forwarded to subsequent stages for memory access or write-back, while the Zero flag helps decide if a branch should occur. By carefully designing the ALU, we ensured that it provides the necessary functionality to support the processor's instruction set while maintaining consistent performance across all operations.

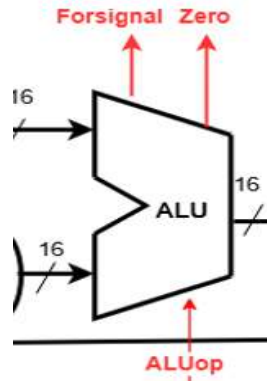


Figure 3: ALU Block Diagram.

4. Data Memory:

We designed the **Data Memory** to handle the storage and retrieval of data during program execution efficiently. It operates as a 16-bit word-addressable memory, enabling seamless read and write operations required by various instructions. The memory takes a 16-bit address as input, which specifies the location for either reading or writing data.

For write operations, the 16-bit data input (Data_in) is stored at the address provided, controlled by the MemWrite signal. For read operations, the memory outputs a 16-bit value (Data_out) from the specified address, enabled by the MemRead signal. By incorporating these control signals, we ensured that the memory interacts smoothly with other datapath components, such as the ALU and Register File, during instruction execution. This design supports a wide range of instructions, including load and store operations, contributing to the processor's overall functionality and efficiency.

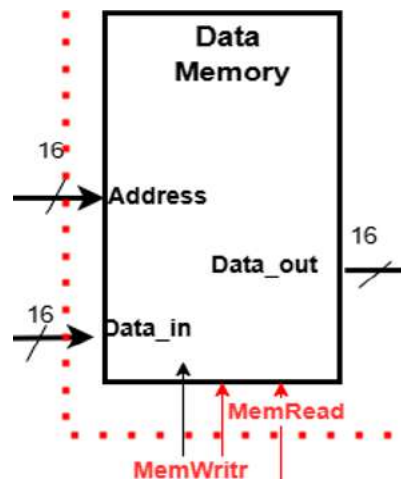


Figure 4: Data Memory Block Diagram

5. Sign Extender:

We designed the **Sign Extender** to play a critical role in handling immediate values in our processor. It takes a 6-bit immediate input (Imm 6) and extends it to a 16-bit value by adding 10 additional bits. This extension ensures compatibility with the processor's 16-bit architecture and allows the immediate values to be used seamlessly in operations like arithmetic, logical, and memory instructions.

To handle both signed and unsigned values, the **Sign Extender** uses the **ExtOp** control signal to determine how the extension is performed:

1. **Signed Extension: (when ExtOp = 1)** If the immediate value is signed, the Sign Extender replicates the most significant bit (MSB) of the 6-bit input across the 10 additional bits. This preserves the sign of the number, ensuring that negative values remain negative and positive values remain positive when extended. For example, an input of 111010 (a negative value in 6-bit representation) would be extended to 1111111111010 in 16-bit format.
2. **Unsigned Extension: (when ExtOp = 0)** If the immediate value is unsigned, the Sign Extender pads the 10 most significant bits with zeros. This ensures that the value remains non-negative after extension. For instance, an input of 001010 (a positive value in 6-bit representation) would be extended to 00000000001010 in 16-bit format.

By allowing the processor to handle both signed and unsigned values effectively, the Sign Extender ensures accurate and flexible execution of immediate instructions. It enables our design to accommodate a wide range of use cases, from basic arithmetic operations to more complex control-flow instructions.

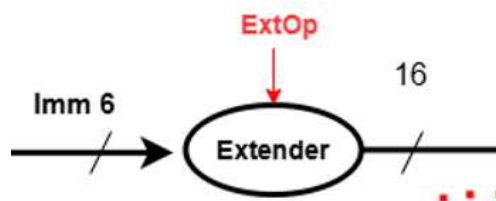


Figure 5: Extender Block Diagram.

6. Program Counter (PC):

We designed the **Program Counter (PC)** as an essential component of our processor,

responsible for keeping track of the address of the next instruction to be executed. The PC is a 16-bit register that plays a critical role in ensuring the proper flow of instruction execution. When updating the PC, we have two possible approaches to determine the **Next PC**. In most cases, we calculate it as $\text{Next PC} = \text{PC} + 2$, which ensures that the processor fetches the next instruction in sequence, as each instruction in our word-addressable memory is 2 bytes wide. Alternatively, we implemented an optimized design where we fix the least significant bit (LSB) of the PC to **0**. In this case, the PC entering the adder is treated as a 15-bit value, excluding the LSB. We then calculate $\text{Next PC} = \text{PC}(\text{15-bit}) + 1$, and the output from the adder is a 15-bit value. Afterward, we append a **0** as the LSB to form a valid 16-bit address.

By supporting these two approaches, we ensure that the PC efficiently handles both sequential instruction execution and non-sequential control flows, such as branches and jumps. This design allows us to maintain accuracy and reliability while optimizing the overall performance of the processor.

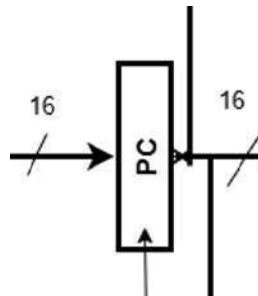


Figure 6: PC Block Diagram.

7. Adder:

We implemented the Adder as a crucial component for computing the next address in the instruction flow. The Adder takes two 16-bit inputs and outputs their sum as a 16-bit result. It plays a pivotal role in updating the Program Counter (PC), ensuring smooth sequential execution or calculating target addresses for branch and jump instructions. By adding the offset or increment value to the current PC, the Adder facilitates both linear and non-linear instruction execution, enhancing the processor's control flow capabilities.

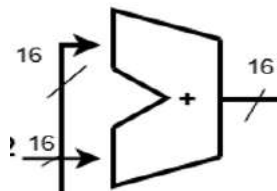


Figure 7: Adder Block Diagram.

8. Mux 8x1 :

We designed the 8x1 Multiplexer (Mux) near the Program Counter (PC) to manage the selection of the next PC address efficiently. This Mux takes eight 16-bit inputs and outputs a single 16-bit value, determined by the 3-bit PCSrc control signal. The control signal specifies the source of the next PC value, allowing flexibility in branching, jumping, or continuing sequential execution. For instance, the Mux selects between the sequential PC increment ($PC + 2$), a jump target ($PC[15:9] \parallel 9\text{-bit offset}$), a branch target address, or BusA (value of Rs). This design ensures proper navigation through the instruction flow.

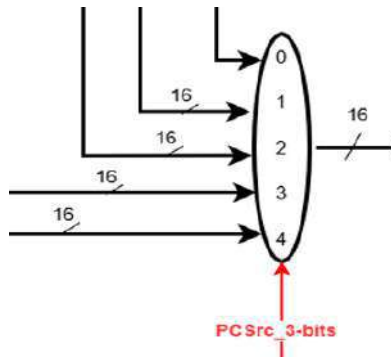


Figure 8: Mux 8x1 Block Diagram.

9. Mux 4x1:

- **Mux 4x1 near RW in the Register File:**

We use this **4x1 Mux** to determine the destination register (RW) for writing data into the register file. The Mux takes four possible 3-bit inputs (e.g., instruction fields Rd, Rt, SR, or RR) and outputs a single 3-bit register identifier. The selection is controlled by the RecSrc2 signal, ensuring that the correct register is updated during the write-back stage based on the operation being performed (e.g., R-type, I-type,).

- **Mux 4x1 near Memory in Write Back :**

This **4x1 Mux** is used to select the source of the data to be written back to the register file. The inputs include data from memory (Data_out), the ALU result, or Next PC address .

The WriteBack control signal determines which source is selected for the 16-bit output. This allows flexibility in updating the register file depending on the instruction type (e.g., load, arithmetic, or control instructions).

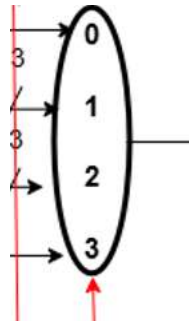


Figure 9: Mux 4x1 Block Diagram.

10) Mux 2x1 :

- **Mux 2x1 near RA in the Register File:**

The **2x1 Mux** near RA controls which register address is used for reading data. It selects between two possible inputs based on the RecSrc1 signal. The inputs may include fields like Rs or RR, depending on the instruction type. The 3-bit output from the Mux is fed into the register file, determining which register's value is read as the first operand.

- **Mux 2x1 near RB in the Register File:**

Similarly, the **2x1 Mux** near RB determines the source of the second register address for reading. The Mux's inputs include Rt or SR. The selection signal **RegSrc2** guides this Mux, and the 3-bit output specifies the second register whose data will be read for use in the ALU or elsewhere.

- **Mux 2x1 near the First Operand in the ALU**

The **2x1 Mux** near the first ALU operand allows us to select the source of the first input to the ALU. The inputs include a value directly from the register file (BusA) or value of Rt. The selection signal AluSrc0 decides which input is passed as the first 16-bit operand to the ALU.

- **Mux 2x1 near the Second Operand in the ALU:**

The **2x1 Mux** near the second ALU operand determines the source of the second input. The inputs include the second register value (BusB) or an extended immediate value from the extender. The control signal AluSrc1 determines which of these 16-bit inputs is passed to the ALU, enabling arithmetic operations with immediate values or register data.

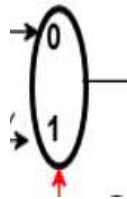


Figure 10: Mux 2x1 Block Diagram.

1.2 :Datapath Assembly:

1. Instruction Fetch (IF)

We begin the instruction fetch stage by using the Program Counter (PC) to provide the address as input to the instruction memory. The instruction memory retrieves the corresponding instruction from memory and outputs it to the instruction decode stage. As shown in the figure, we use a **4x1 Mux** near the PC to select the next PC value. For sequential instructions without branching or jumping, we increment the PC by one. However, for jump and branch instructions, the control unit selects alternative values for the PC using the PCSrc signal, which determines the next address to fetch based on the instruction type.

2. Instruction Decode (ID)

We decode the instruction in this stage using the instruction register (IR) and the control unit. The control unit generates control signals based on the opcode in the instruction. The instruction fields are divided depending on the opcode type. For example, in R-type instructions (opcode “000000”), fields include destination (Rd) and two source registers (Rs and Rt).

The **register file** includes two special registers:

- **RR (Return Register):** Used for Call and Return instructions.
- **SR (Special Register):** Used for For loop control.

The RR connects with Rs and feeds into the **2x1 Mux** near RA, allowing the selection of either RR or Rs based on control signals. Similarly, the SR connects with Rt and feeds into the **2x1 Mux** near RB, selecting either SR or Rt as the second operand. Both RR and SR also connect to the **4x1 Mux** near RW, enabling selection of the appropriate register destination for writing back results.

Additionally, a sign extender is used in this stage for specific instructions that require immediate values or offsets. The branch target address is computed here by adding the offset to the PC for branch instructions.

3. Execution (EX)

We perform arithmetic or logical operations in the execution stage using the ALU. The ALU takes two operands, with the first operand selected by a **2x1 Mux** near the ALU (choosing between BusA and another source like the PC). The second operand is selected by another **2x1 Mux** near the ALU (choosing between BusB and an extended immediate value). The ALU performs operations such as ADD, SUB, and AND based on control signals generated in the decode stage. Some instructions, such as Jump, bypass the execution stage entirely and proceed directly to the next stage.

4. Memory Access (MEM)

We access data memory in this stage for load (LW) and store (SW) instructions, as well as stack-related operations like Push and Pop. For store instructions, we use a **2x1 Mux** to select the value written to the memory. The Mux selects between BusB for regular store instructions and PC + 1 for specific instructions. Note that Call and Return instructions do not interact with the Data Memory directly; instead, they update the PC through other paths. Many instructions, such as R-type and certain I-type instructions, bypass this stage entirely as they do not involve memory access.

5. Write Back (WB)

We write the results back to the register file in the final stage. For load instructions, we use a **4x1 Mux** to select the value to write back to the destination register. This Mux chooses between the memory output (data_out) or the ALU result based on the control signal (WriteBack). For instructions like LW.POI, the value of Rs1 is also updated simultaneously with the destination register. We utilize multiple buses to ensure that the required data is written back efficiently to the appropriate register.

1.3 Control Signals :

1. Main Control Signals:

Signal	Effect when '0'	Effect when '1'	Effect when '2'	Effect when '3'
RegWrite	Disables writing to the register file	Enables writing to the register file	-	-
MemRead	Data memory is not accessed	Data is read from the memory	-	-
MemWrite	Data is not written to memory	Data is written to memory	-	-
ALUSrc0	ALU operand comes from BusA	ALU operand comes from value of Rt (BusB)	-	-
ALUSrc1	ALU operand comes from BusB	ALU operand comes from extended immediate	-	-
ExtOp	Immediate values are not sign-extended	Immediate values are sign-extended	-	-
WriteBack (2-bits)	Selects ALU result for write-back	Selects memory output for write-back	Selects PC + 1 for write-back (Call)	-
RecSrc1	Selects Rs as an input to Mux near RA	Selects RR as an input to Mux near RA	-	-

RecSrc2	Selects Rt as an input to Mux near RB	Selects SR as an input to Mux near RB	-	-
RecDest (2-bits)	Selects destination register Rd for R-type instructions	Selects destination register Rt for I-type instructions	Selects SR (Special Register) for loop control	Selects RR (Return Register) for Call/Return instructions

Table 1: Main Control Signals.

2. ALU Control Signals (3-bit) :

ALU Control Signal	Effect when '000'	Effect when '001'	Effect when '010'	Effect when '011'	Effect when '100'	Effect when '101'	Effect when '110'	Effect when '111'
ALU Operation	Add	Subtract	AND	OR	Set on less than	XOR	Shift left	Shift right
Example Operation	A + B	A - B	A AND B	A OR B	A < B (set to 1 or 0)	A XOR B	Shift A left by 1 bit	Shift A right by 1 bit

Table 2: ALU Control Signals.

3. PC Control Signals :

Signal	Effect when '0'	Effect when '1'	Effect when '2'	Effect when '3'	Effect when '4'
PCSrc	PC = PC + 2	PC = {PC[15:9], offset_9bit} (Jump)	PC = Branch Target Address (BTA)	PC = Return Address (RA)	PC = Next Address for For-Loop Instruction

Table 3: PC Control Signals.

1.4: Main Control Signals - Truth Table:

1.Main Control Signals - Truth Table:

Opera nd..	Reg Dst	Reg Wr	Reg__ Src1	Reg__ Src2	Alu_S rc0	Alu_S rc1	Mem Rd	Mem Wr	Ext____ Op	ForSig nal	WB_dat a
R-type	00= Rd	1	0 = Rs	0 = Rt	0=Bus A	0=Bus B	0	0	x	x	0=Alu
ANDI	01= Rt	1	0 = Rs	0 = Rt	0=Bus A	1=Im m	0	0	0=zer o	x	00=Alu
ADDI	01= Rt	1	0 = Rs	0 = Rt	0=Bus A	1=Im m	0	0	1=sign	x	00=Alu
LW	01= Rt	1	0 = Rs	0 = Rt	0=Bus A	1=Im m	1	0	1=sign	x	01=Me m
SW	x	0	0 = Rs	0 = Rt	0=Bus A	1=Im m	0	1	1=sign	x	x
BEQ	x	0	0 = Rs	0 = Rt	0=Bus A	0=Bus B	0	0	1=sign	x	x
BNE	x	0	0 = Rs	0 = Rt	0=Bus A	0=Bus B	0	0	1=sign	x	x
for	10= SR	1	0 = Rs	0 = Rt	1=Bus B	1=Im m	0	0	1=sign	0	00=Alu
for	10= SR	1	0 = Rs	1 = SR	1=Bus B	1=Im m	0	0	1=sign	1	00=Alu
Jump	x	0	x	x	x	x	0	0	x	0	x
Call	11= RR	1	x	x	x	x	0	0	x	0	10=Nex tPC
RET	x	0	1=RR	x	x	x	0	0	x	0	x

Table 4 :Main Control Signals - Truth Table.

2.ALU Control Signals - Truth Table:

Op	Funct	ALUOp	3-bit code
R-type	AND	AND	000
R-type	ADD	ADD	001
R-type	SUB	SUB	010
R-type	SLL	SLL	011
R-type	SLR	SLR	100
ANDI	x	AND	000
ADDI	x	ADD	001
LW	x	ADD	001
SW	x	ADD	001
BEQ	x	SUB	010
BNQ	x	SUB	010
For	x	SUB	010
J-type	JUMP	x	x
J-type	CALL	x	x
J-type	RET	x	x

Table 5 :ALU Control Signals - Truth Table.

3.PC Control Signals - Truth Table:

Op	Funct	Zero Flag	BusB	PCSrc
R-type	AND	x	x	000=Next PC
R-type	ADD	x	x	000=Next PC
R-type	SUB	x	x	000=Next PC
R-type	SLL	x	x	000=Next PC
R-type	SLR	x	x	000=Next PC
ANDI	x	x	x	000=Next PC
ADDI	x	x	x	000=Next PC
LW	x	x	x	000=Next PC
SW	x	x	x	000=Next PC
BEQ	x	0	x	000=Next PC
BEQ	x	1	x	010=BTA
BNE	x	0	x	010=BTA
BNE	x	1	x	000=Next PC
For	x	x	16b'0	000=Next PC
For	x	x	16b'1	011=RS
J-type	JUMP	x	x	001=JTA
J-type	CALL	x	x	001=JTA
J-type	RET	x	x	100=Rs

Table 6 :PC Control Signals - Truth Table.

1.5: Boolean Equations :

1. RegDst

- Based on the table:
 - 00 for R-Type.
 - 01 for ANDI, ADDI, LW.
 - 10 for for.
 - 11 for Call.

- Equation:

$$RegDst[0] = \sim(R\text{-Type} + ANDI + ADDI + LW)$$

$$RegDst[] = \sim(R\text{-Type} + for)$$

3. RegSrc1

- Always 1 when op = call
- Equation:

$$RegSrc1 = Call$$

4. RegSrc2

- Equal 1 when execute for instruction and for Signal = 1 .
- Equation:

$$RegSrc2 = (for \& forSignal)$$

5. AluSrc0

- Equal 1 when execute for instruction
- Equation:

$$AluSrc0 = for$$

6. AluSrc1

- Based on the table:
 - 1 for ANDI, ADDI, LW, SW, BEQ, BNE, and for.
 - 0 for R-Type , BEQ, BNE, .
- Equation:

$$AluSrc1 = \sim(R\text{-Type} + BEQ + BNE)$$

7. MemRd

- Based on the table:
 - 1 for LW.
 - 0 for all other instructions.

- **Equation:**

$$MemRd=(LW)$$

8. MemWr

- Based on the table:
 - 1 for SW.
 - 0 for all other instructions.

- **Equation:**

$$MemWr=(SW)$$

9. ExtOp

- Based on the table:
 - zero for ANDI.
 - sign for ADDI, LW, SW, BEQ, BNE, and for.

- **Equation:**

$$ExtOp=\sim(ANDI)$$

10. WBData

- Based on the table:
 - 00 for R-Type, ANDI, ADDI, and for.
 - 01 for LW.
 - 10 for Call.

- **Equation:**

$$WBData[1]= (LW)$$

$$WBData[0]= (Call)$$

1. PCSrc

- **Observations:**

- 000 for R-type, ANDI, ADDI, LW, SW, BEQ (Zero=0), BNE (Zero=1), and For (16b'0).
- 010 for BEQ (Zero=1) and BNE (Zero=0).
- 001 for JUMP and CALL.
- 011 for For (16b'1).
- 100 for RET.

• **Equations:**

$$PCSrc[2] = (JUMP + CALL + (for . BusB))$$

$$PCSrc[1] = (BEQ . Zero) + (BNE . !Zero) + (for . BusB)$$

$$PCSrc[0] = RET$$

1.6: Multi-Cycle DataPath and Control:

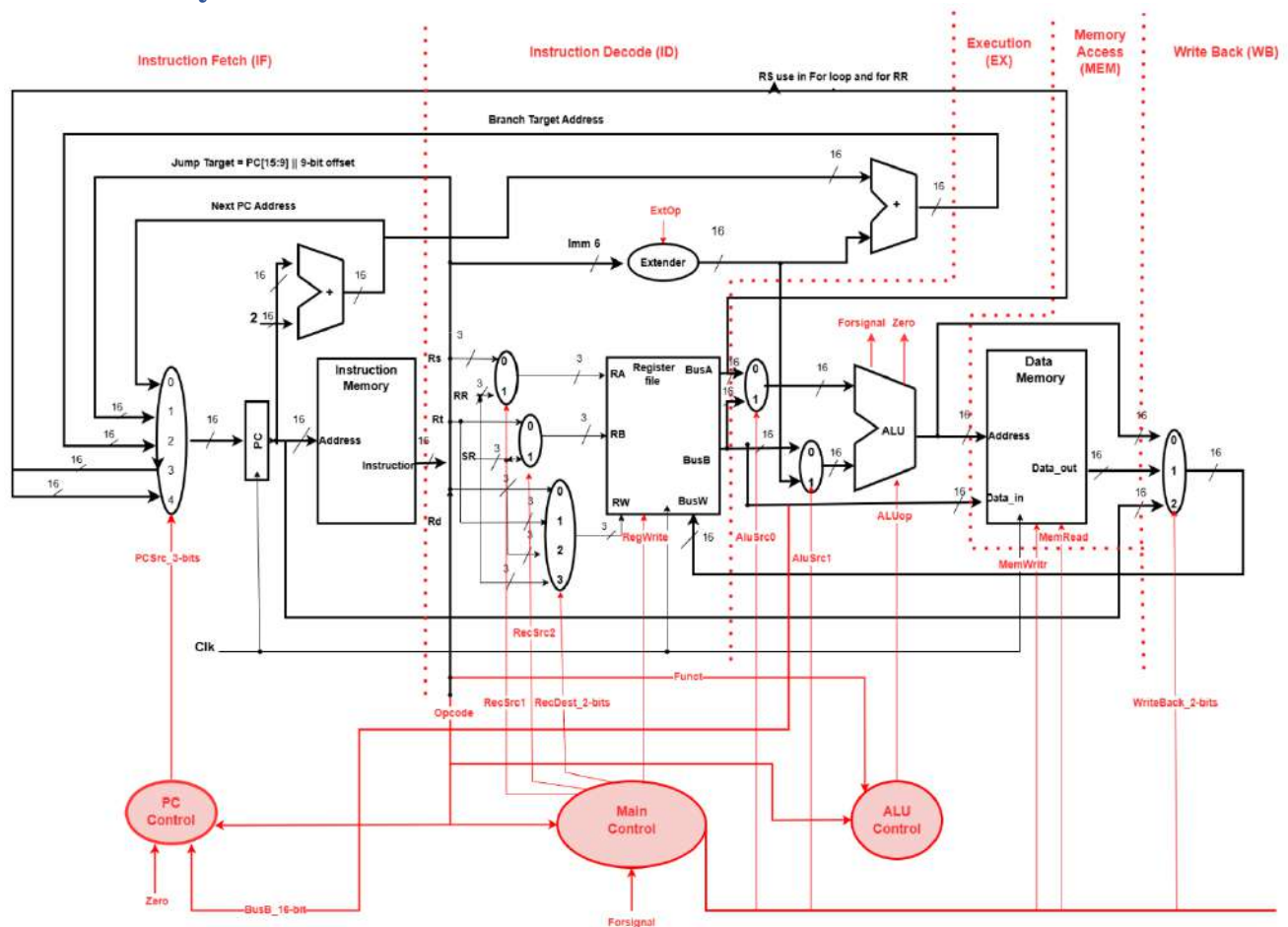


Figure 11: DataPath and control Block Diagram.

2. Instruction Sequence and WaveForms:

```

initial begin
    DataMemory[7] = 16'h0009;
    DataMemory[8] = 16'h0004;
    DataMemory[12] = 16'h0007;
    DataMemory[14] = 16'h0006;
    DataMemory[15] = 16'h0006;
    DataMemory[16] = 16'h000A;
    DataMemory[17] = 16'h0006;
    DataMemory[27] = 16'h000f;
    DataMemory[33] = 16'h0006;
    DataMemory[35] = 16'h0007;
    DataMemory[36] = 16'h0004;
end
initial begin

//Program 1
Imemory[0] = 16'b0000001010111000; // AND -> reg(1)=reg(2)& reg(7) -> reg(1)= 3 & 9 -> reg(1) = 1
Imemory[1] = 16'b0000010001101001; // ADD -> reg(2) = reg(1) + reg(5) -> reg(2) = 1+7 -> reg(2) = 8
Imemory[2] = 16'b0000111010011010; //SUB -> reg(7) = reg(2) - reg(3) -> reg(7) = 8-4 -> reg(7) = 4
Imemory[3] = 16'b0000011110001011; //SLL -> (reg(3) = reg(6) << reg(1)) -> (reg(3) = 8 << 1) -> reg(3) = 16 (10 in
Imemory[4] = 16'b0000101111001100; // SRL -> reg(5) = reg(7) >> reg(1) -> reg(5) = 4 >> 1 -> reg(5) = 2

Imemory[5] = 16'b1000000101000000; // for reg(0),reg(5) Rs is 0 and Rt is 2 so loop twice

//the values after the next iteration
//reg(1)=reg(2)& reg(7) -> reg(1) = 8 & 4 = 0
//reg(2) = reg(1) + reg(5) -> reg(2) = 0 + 2 = 2
//reg(7) = reg(2) - reg(3) -> reg(7) = 2 - 16 = -14 (ffff in hexa)
//reg(3) = reg(6) << reg(1) -> reg(3) = 8 << 0 = 8
//reg(5) = reg(7) >> reg(1) -> reg(5) = -14 >> 0 = -14 (ffff in hexa)

Imemory[6] = 16'b0110110011000010; // BEQ if(reg(6) == reg(3)) nextPC = BTA else : nextPC=PC+1 (note:imm=2)
Imemory[7] = 16'b0011010111001001; // ADDI -> reg(7) = reg(2) + 9 -> reg(7) = 2 + 9 = 11 (b in hexa) (if the brai
Imemory[8] = 16'b0100011100001000; // LW -> reg(4) = mem(reg(3)+8) -> reg(4) = mem(8+8) = mem(16) = 10 (A in hexa)
Imemory[9] = 16'b0101001101000111; //SW -> mem(reg(1)+7) = reg(5) -> mem(0+7)= mem(7) -> mem(7) = reg(5) -> mem(7)

Imemory[10]= 16'b0010010101000110; // ANDI -> reg(5)=reg(2)& 6 -> reg(5) = 2 & 6 = 2 -> reg(5) = 2
Imemory[11]= 16'b0111000001000011; // BNE if(reg(0) != reg(1)) nextPC = BTA else : nextPC=PC+1 (note:imm=3)

Imemory[12] = 16'b0000001010111000; // AND -> reg(1)=reg(2)& reg(7) -> reg(1)= 2 & fff2 -> reg(1) = 2 (the bran
Imemory[13] = 16'b0000010001101001; // ADD -> reg(2) = reg(1) + reg(5) -> reg(2) = 2+2 -> reg(2) = 4 (the bran

Imemory[14]= 16'b0001000010001000; // JMP offset nextPC = jumptarget (offset = 17), nextPC = {PC[15:9],offset} =
Imemory[15] = 16'b0011010111001001; // ADDI (skipped because of jump instruction)
Imemory[16] = 16'b0000011110001011; //SLL (skipped because of jump instruction)

Imemory[17]= 16'b0001000010101001; // CALL offset nextPC = jumptarget ,PC + 1 stored on the RR (offset = 20), n
Imemory[18] = 16'b0000001010111000; // AND -> reg(1)=reg(2)& reg(7) -> reg(1)= 4 & d -> reg(1) = 4
Imemory[19] = 16'b0000011001101001; // ADD -> reg(3) = reg(1) + reg(5) -> reg(3) = 4+2 -> reg(3) = 6 (consider
Imemory[20] = 16'b1111111111111111; // exit the program
Imemory[21] = 16'b0011010111001001; // ADDI -> reg(7) = reg(2) + 9 -> reg(7) = 4 + 9 = 13 (d in hexa)

Imemory[22]= 16'b0001000000000010; // RET nextPC = value of the RR ,the 9-bit field is ignored in this instructi

```

Figure 12: Instructions

As shown in the figure above, we have a sequence of various instructions stored in memory. These instructions include R-type, I-type, and J-type. To begin, we initialize the Program

Counter (PC) to point to the first instruction. During the fetch stage, we retrieve the first instruction from memory, which is an "AND" operation. This instruction is identified by its opcode "0000" and function code "000."

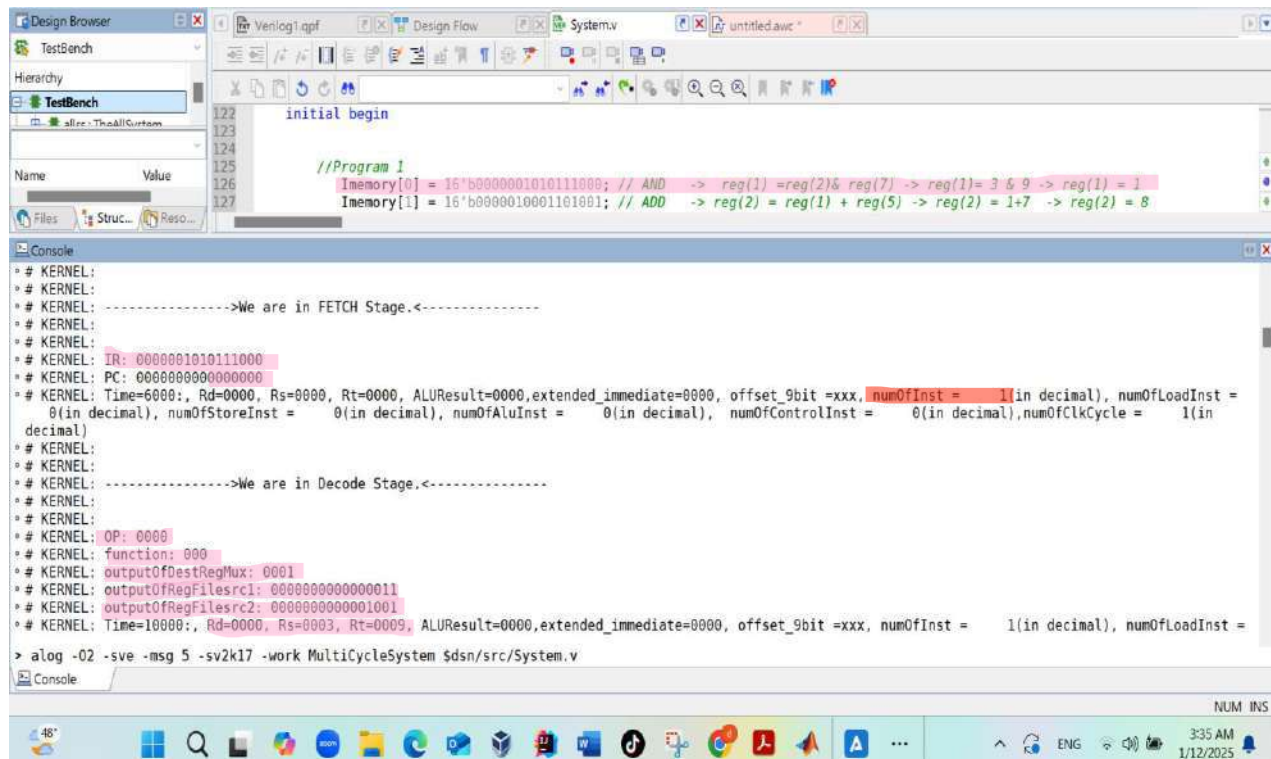


Figure 13: Fetch and Decode of first Instruction(R-Type Instruction).

As seen in the figure above, the opcode is "0000" and funct "000" which is the "and" instruction. In the decode stage, the fields are assigned based on the opcode. The destination address is "0001", thus the result of the add instruction should be stored in R1. The two operands are in the registers R2 "010" and R7 "111" respectively, having the following content: 0x3 and 0x9 respectively.

```

initial begin
// 16 32-bit general-purpose registers: from R0 to R10.
Registers[0] = 16'h0000;
Registers[1] = 16'h0002;
Registers[2] = 16'h0003;
Registers[3] = 16'h0004;
Registers[4] = 16'h0005;
Registers[5] = 16'h0007;
Registers[6] = 16'h0008;
Registers[7] = 16'h0009;
Registers[8] = 16'h0000; // to store the counter of the for loop
Registers[9] = 16'h0000; // to store the return address

end
////////////////

```

Register[1]
before write back

SR

RR

Figure 14 : The values of Register before start program.

```

* # KERNEL: ----->We are in Execution Stage.<-----
* # KERNEL:
* # KERNEL: PC: 0000000000000000
* # KERNEL: outputOfBTA: 0000000000000000
* # KERNEL: zero flag: x
* # KERNEL: Time=14000:, Rd=0000, Rs=0003, Rt=0009, ALUResult=0001, extended immediate=0000, offset_9bit=xxx, numOfInst = 1(in decimal), numOfLoadInst = 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 1(in decimal), numOfControlInst = 0(in decimal), numOfClkCycle = 3(in decimal)
* # KERNEL: ----->We are in Write Back Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL: :MuxOut : 0001
* # KERNEL:
* # KERNEL: Register File Contents:
* # KERNEL: Registers[0] : 0000
* # KERNEL: Registers[1] : 0001
* # KERNEL: Registers[2] : 0003
* # KERNEL: Registers[3] : 0004
* # KERNEL: Registers[4] : 0005
* # KERNEL: Registers[5] : 0007
* # KERNEL: Registers[6] : 0008
* # KERNEL: Registers[7] : 0009
* # KERNEL: Registers[8] : 0000
* # KERNEL: Registers[9] : 0000
* # KERNEL:
> alog -O2 -sve -msg 5 -sv2k17 -work MultiCycleSystem $dsn/src/System.v

```

Figure 15: Execution and Writeback of first Instruction.

In the execution stage, as shown in the figure above, we perform the AND operation between the two operands. Specifically, the values 0x3 and 0x9 are ANDed, and the result, 0x1, is displayed in the "ALUResult" (as seen in Figure 15). Referring to Figure 14, we notice that the value in R1 changes from 0x2 to 0x1. The "AND" instruction skips the memory stage and proceeds directly to the write-back stage, where the result, 0x1, is written back into the register file. The "Muxout" selects the value from the ALU result, which is then displayed on the screen.

Performance Registers Explanation:

The AND instruction is an ALU operation. During the Execution stage (refer to Figure 15, highlighted in red), the **numOfAluInst** register is incremented by 1, indicating that an ALU instruction has been executed. Meanwhile, during the Fetch stage (refer to Figure 14, highlighted in red), the **numOfInst** register is incremented by 1 for every instruction that is fetched.

This mechanism ensures that **numOfInst** tracks the total number of instructions fetched, while **numOfAluInst** counts the number of ALU-specific instructions executed, and **numOfClkCycle** become 3 because we pass through 3 stages (Fetch , Decode and Execution) and increment by 1 when triggered by positive edge in Clock .These registers are particularly useful for performance analysis, such as calculating the total number of stall cycles in the multicycle.

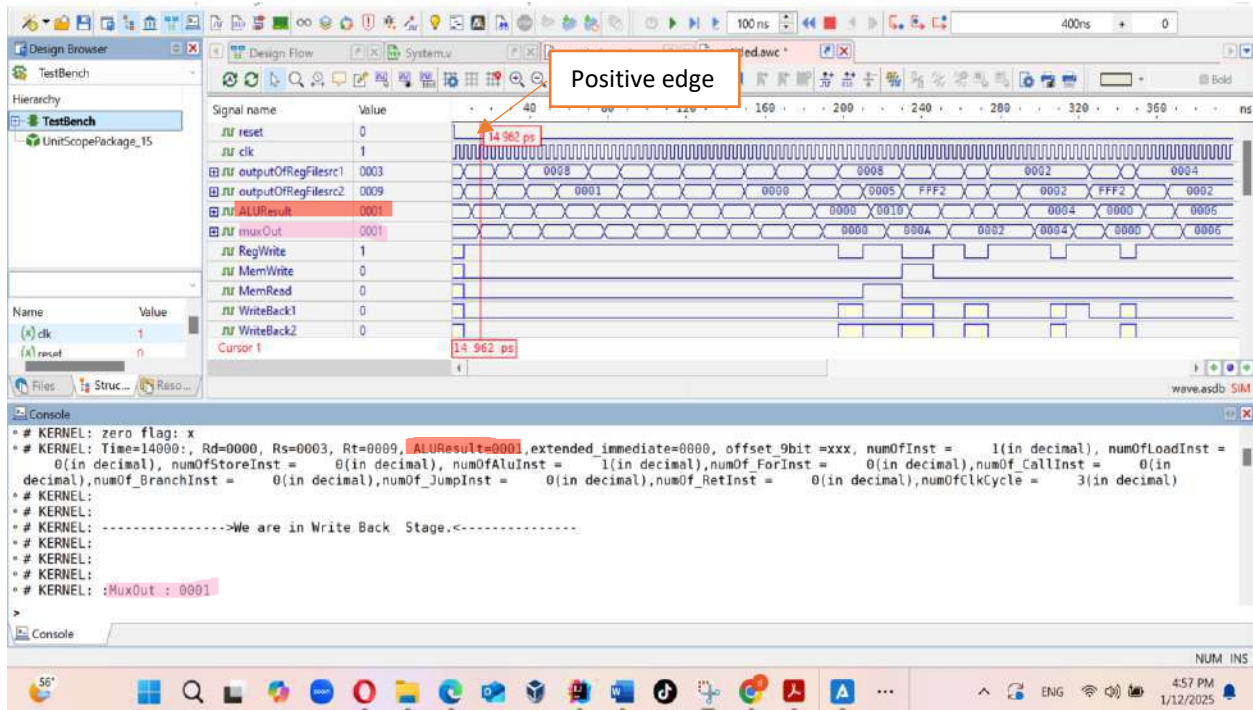


Figure 16: Waveform of first Instruction.

This waveform illustrates how the "AND" instruction operates, along with key signals that demonstrate its functionality. Each stage is executed on the positive edge of the clock (clk). We observe that the **ALUResult** is accurate, and the **RegWrite** signal is activated to allow the result to be written back to the register file. Since memory is not involved in this operation, both **MemRead** and **MemWrite** signals remain at 0. Additionally, we assign **WriteBack1** and **WriteBack2** the value 0 because the output is generated by the ALU rather than being fetched from memory.

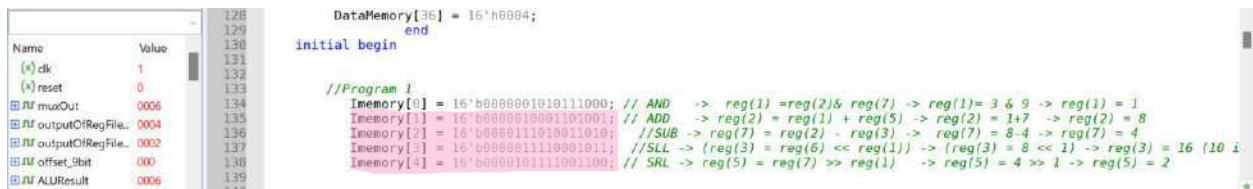


Figure 17: Other instruction in R-Type.

In Figure 17, we demonstrate how we execute other R-type instructions stored in **Imemory[1]** to **Imemory[4]** (ADD, SUB, SLL, SLR) using the same method applied for the AND instruction. We use the same values for the control signals to ensure consistent operation. For each instruction, we increment the **PC** by one unit, advancing sequentially until it reaches **Imemory[5]**. This systematic approach allows us to process each R-type instruction efficiently.

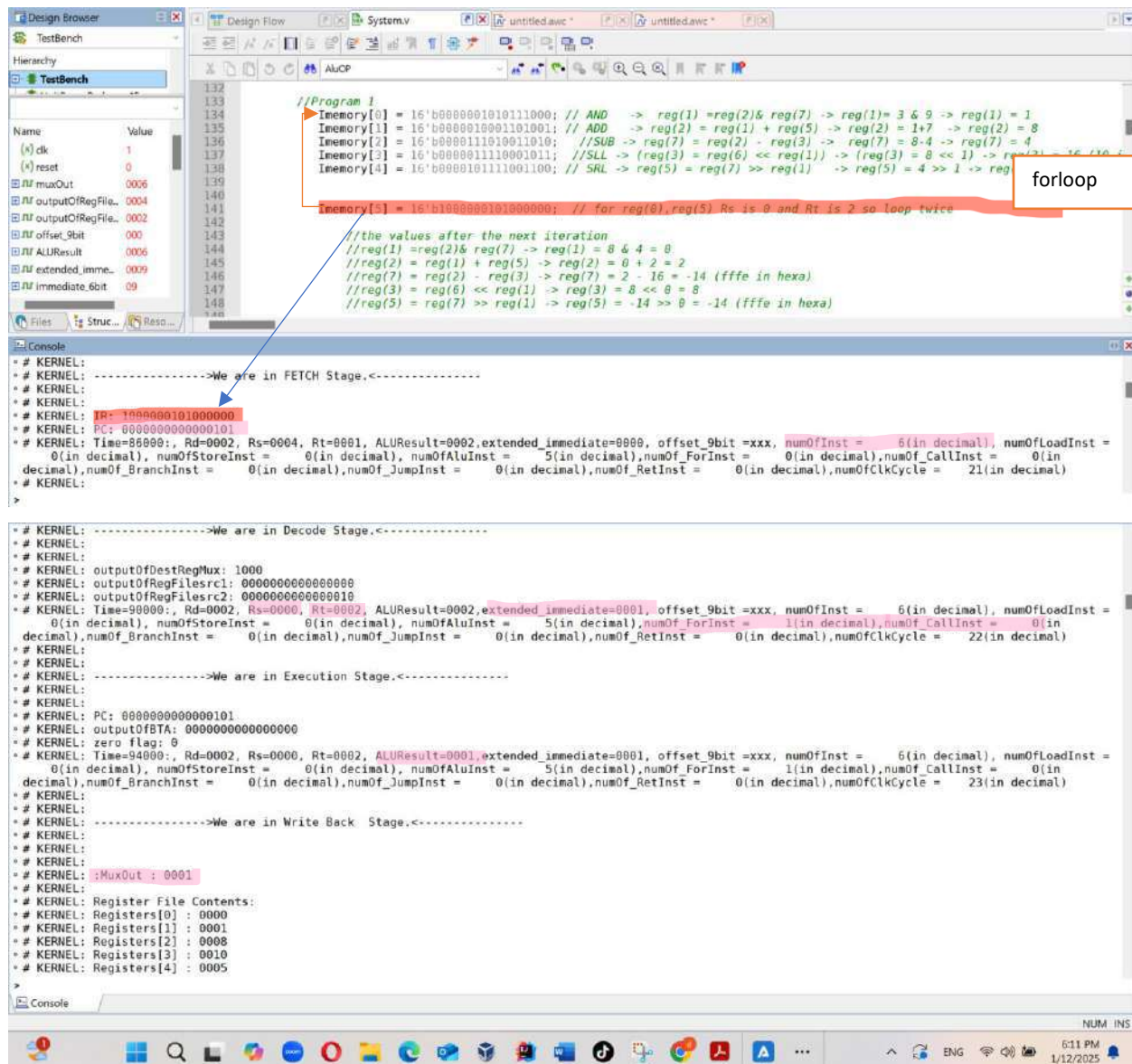


Figure 18: for instruction in I-Type

The **FOR** instruction is an I-type instruction with a specific format: FOR Rs, Rt. For this instruction, we set **ExtOp** to 1, indicating that the second operand to the ALU is the extended immediate value. The **Rs** register stores the loop target address, which is the address of the first instruction in the loop block. For example, as shown in the figure, **Rs = 0x0000**, meaning the program counter (**PC**) includes the address **Memory[0]**, where the first instruction in our program resides (in this case, an AND logic operation).

The **Rt** register holds the initial number of loop iterations, serving as the initial value of the loop counter. At the end of each iteration, we decrement the value in **Rt**. The loop exits when the content of **Rt** becomes zero. In the figure, we see that **Rt = 2**, meaning there are two iterations in this loop. During each iteration, we decrement **Rt** by 1 and store the **ALUResult** in **SR** (Register[8]). The **MuxOut** directs the result to **BusW**, and it is stored in **SR** using **RegDst = 01**, which selects **SR** as the destination.

We then check the **ALUResult**. If **ALUResult = 0**, we set **ForSignal = 0**, and the loop terminates, moving to the next instruction (**Next PC = the instruction after the loop**). However, if **ALUResult \neq 0**, we set **ForSignal = 1**, causing the program to jump back to the loop's starting address (**Next PC = Rs**, the first instruction in the program).

The **ForSignal** is critical as it controls **RegSrc2**, determining the source for the second operand in the register file. When **ForSignal = 1**, we set **RegSrc2 = 1**, selecting **SR** as the second operand. When **ForSignal = 0**, we set **RegSrc2 = 0**, selecting **Rt** as the second operand. In the first iteration of the **FOR** loop, we use **Rt** as the second operand in the register file. In subsequent iterations, we always use **SR** as the second operand. For each iteration, we continue this process until **Rt - 1 = 0**, at which point the loop terminates, and we proceed to the next instruction outside the loop. The **ALUResult** is always stored in **SR** during the loop, ensuring consistency in the loop's operation. The **numOffForInst** increment by 1 in each iteration .

◦ # KERNEL:	◦ # KERNEL:
◦ # KERNEL: Register File Contents:	◦ # KERNEL: Register File Contents:
◦ # KERNEL: Registers[0] : 0000	◦ # KERNEL: Registers[0] : 0000
◦ # KERNEL: Registers[1] : 0001	◦ # KERNEL: Registers[1] : 0000
◦ # KERNEL: Registers[2] : 0008	◦ # KERNEL: Registers[2] : 0008
◦ # KERNEL: Registers[3] : 0010	◦ # KERNEL: Registers[3] : 0010
◦ # KERNEL: Registers[4] : 0005	◦ # KERNEL: Registers[4] : 0005
◦ # KERNEL: Registers[5] : 0002	◦ # KERNEL: Registers[5] : 0002
◦ # KERNEL: Registers[6] : 0008	◦ # KERNEL: Registers[6] : 0008
◦ # KERNEL: Registers[7] : 0004	◦ # KERNEL: Registers[7] : 0004
◦ # KERNEL: Registers[8] : 0000	◦ # KERNEL: Registers[8] : 0001
◦ # KERNEL: Registers[9] : 0000	◦ # KERNEL: Registers[9] : 0000
◦ # KERNEL:	◦ # KERNEL:

a)Before one iteration b)After one iteration

Figure 19: Register [8] after and before one iteration loop

After one iteration loop the SR Register [8] will be change from 0x0000 to 0x0001 (The Aluresult) .

- ❖ Second iteration : (Register[8] = RT = 1) && (PC = Rs) :

```

//Program 1
Memory[0] = 16'b000000101011000; // AND -> reg(1)=reg(2)& reg(7) -> reg(1)= 3 & 9 -> reg(1) = 1
Memory[1] = 16'b0000010001101001; // ADD -> reg(2) = reg(1) + reg(5) -> reg(2) = 1+7 -> reg(2) = 8
Memory[2] = 16'b0000111010011010; // SUB -> reg(7) = reg(2) - reg(3) -> reg(7) = 8-4 -> reg(7) = 4
Memory[3] = 16'b0000011110001011; // SLL -> (reg(3) = reg(6) << reg(1)) -> (reg(3) = 8 << 1) -> reg(3) = 16 (10 i

```

```

# KERNEL: ----->We are in FETCH Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: IR: 000000101011000
# KERNEL: PC: 0000000000000000
# KERNEL: Time=102000: Rd=0001, Rs=0000, Rt=0002, ALUResult=0001, extended_immediate=0001, offset_9bit=xxx, numOfInst = 7(in decimal), numOfLoadInst = 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 5(in decimal), numOf_ForInst = 1(in decimal), numOf_CallInst = 0(in decimal), numOf_BranchInst = 0(in decimal), numOf_JumpInst = 0(in decimal), numOf_RetInst = 0(in decimal), numOfClkCycle = 25(in decimal)
# KERNEL:
# KERNEL: ----->We are in Decode Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: OP: 0000
# KERNEL: function: 000
# KERNEL: outputOfDestRegMux: 0001
# KERNEL: outputOfRegFilesrc1: 0000000000001000
# KERNEL: outputOfRegFilesrc2: 0000000000000100
# KERNEL: Time=106000: Rd=0001, Rs=0000, Rt=0004, ALUResult=0001, extended_immediate=0001, offset_9bit=xxx, numOfInst = 7(in decimal), numOfLoadInst = 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 6(in decimal), numOf_ForInst = 1(in decimal), numOf_CallInst = 0(in decimal), numOf_BranchInst = 0(in decimal), numOf_JumpInst = 0(in decimal), numOf_RetInst = 0(in decimal), numOfClkCycle = 26(in decimal)
# KERNEL:
>
# KERNEL:
# KERNEL: ----->We are in Execution Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: PC: 0000000000000000
# KERNEL: outputOfBTA: 0000000000000000
# KERNEL: zero flag: 0
# KERNEL: Time=110000: Rd=0001, Rs=0000, Rt=0004, ALUResult=0000, extended_immediate=0001, offset_9bit=xxx, numOfInst = 7(in decimal), numOfLoadInst = 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 6(in decimal), numOf_ForInst = 1(in decimal), numOf_CallInst = 0(in decimal), numOf_BranchInst = 0(in decimal), numOf_JumpInst = 0(in decimal), numOf_RetInst = 0(in decimal), numOfClkCycle = 27(in decimal)
# KERNEL:
# KERNEL:
# KERNEL: ----->We are in Write Back Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: :MuxOut: 0000
# KERNEL:
# KERNEL:
# KERNEL: Register File Contents:
# KERNEL: Registers[0] : 0000
# KERNEL: Registers[1] : 0000
# KERNEL: Registers[2] : 0008
# KERNEL: Registers[3] : 0010
# KERNEL: Registers[4] : 0005
# KERNEL: Registers[5] : 0002
# KERNEL: Registers[6] : 0008
# KERNEL: Registers[7] : 0004
# KERNEL: Registers[8] : 0001
# KERNEL: Registers[9] : 0000
# KERNEL:
# KERNEL:
# KERNEL:

```

Figure 20 : First Instruction in Seconed iteration from for Instruction.

After execute all instruction in second iteration and reach the for instruction in second time.

```

Design Browser
TestBench
Hierarchy
TestBench
Name Value
139
140
141
142
Imemory[5] = 16'b1000000101000000; // for reg(0).reg(5) Rs is 0 and Rt is 2 so loop twice

Console
# KERNEL: ----->We are in FETCH Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: IR: 1000000101000000
# KERNEL: PC: 0000000000000101
# KERNEL: Time=182000: Rd=fff2, Rs=fff2, Rt=0000, ALUResult=fff2,extended_immediate=0001, offset_9bit =xxx, numOfInst = 12(in decimal), numOfLoadInst = 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 10(in decimal), numOf_ForInst = 1(in decimal), numOf_CallInst = 0(in decimal), numOf_BranchInst = 0(in decimal), numOf_JumpInst = 0(in decimal), numOf_RetInst = 0(in decimal), numOfClkCycle = 45(in decimal)
# KERNEL:
# KERNEL: ----->We are in Decode Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: outputOfDestRegMux: 1000
# KERNEL: outputOfRegFilesrc1: 0000000000000000
# KERNEL: outputOfRegFilesrc2: 0000000000000001
# KERNEL: Time=186000: Rd=fff2, Rs=0000, Rt=0001, ALUResult=fff2,extended_immediate=0001, offset_9bit =xxx, numOfInst = 12(in decimal), numOfLoadInst = 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 10(in decimal), numOf_ForInst = 2(in decimal), numOf_CallInst = 0(in decimal), numOf_BranchInst = 0(in decimal), numOf_JumpInst = 0(in decimal), numOf_RetInst = 0(in decimal), numOfClkCycle = 46(in decimal)
# KERNEL:
# KERNEL: ----->We are in Execution Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL:
# KERNEL: ----->We are in Execution Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: PC: 0000000000000101
# KERNEL: outputOfBTA: 0000000000000000
# KERNEL: zero flag: 0
# KERNEL: Time=190000: Rd=fff2, Rs=0000, Rt=0001, ALUResult=0000,extended_immediate=0001, offset_9bit =xxx, numOfInst = 12(in decimal), numOfLoadInst = 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 10(in decimal), numOf_ForInst = 2(in decimal), numOf_CallInst = 0(in decimal), numOf_BranchInst = 0(in decimal), numOf_JumpInst = 0(in decimal), numOf_RetInst = 0(in decimal), numOfClkCycle = 47(in decimal)
# KERNEL:
# KERNEL: ----->We are in Write Back Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL:
# KERNEL: MuxOut : 0000
# KERNEL:
# KERNEL: Register File Contents:
# KERNEL: Registers[0] : 0000
# KERNEL: Registers[1] : 0000
# KERNEL: Registers[2] : 0002
# KERNEL: Registers[3] : 0008
# KERNEL: Registers[4] : 0005
# KERNEL: Registers[5] : fff2
# KERNEL: Registers[6] : 0008
# KERNEL: Registers[7] : fff2
# KERNEL: Registers[8] : 0000
# KERNEL: Registers[9] : 0000
# KERNEL:
# KERNEL:
# KERNEL: *****DONE The INST*****
# KERNEL:
# KERNEL:

```

Figure 21 : Reach to the for instruction in second time.

When executing the **FOR** instruction, we observe that **Rt = SR = 1**. After the ALU operation, **Rt - 1 = Rt**, indicating that the loop has reached its termination condition. As a result, the loop ends, and the program counter (**PC**) moves to the next instruction at **Imemory[6]**, exiting the loop and continuing with the program execution . And for signal become 0.

❖ Waferome of for instruction in one iteration :

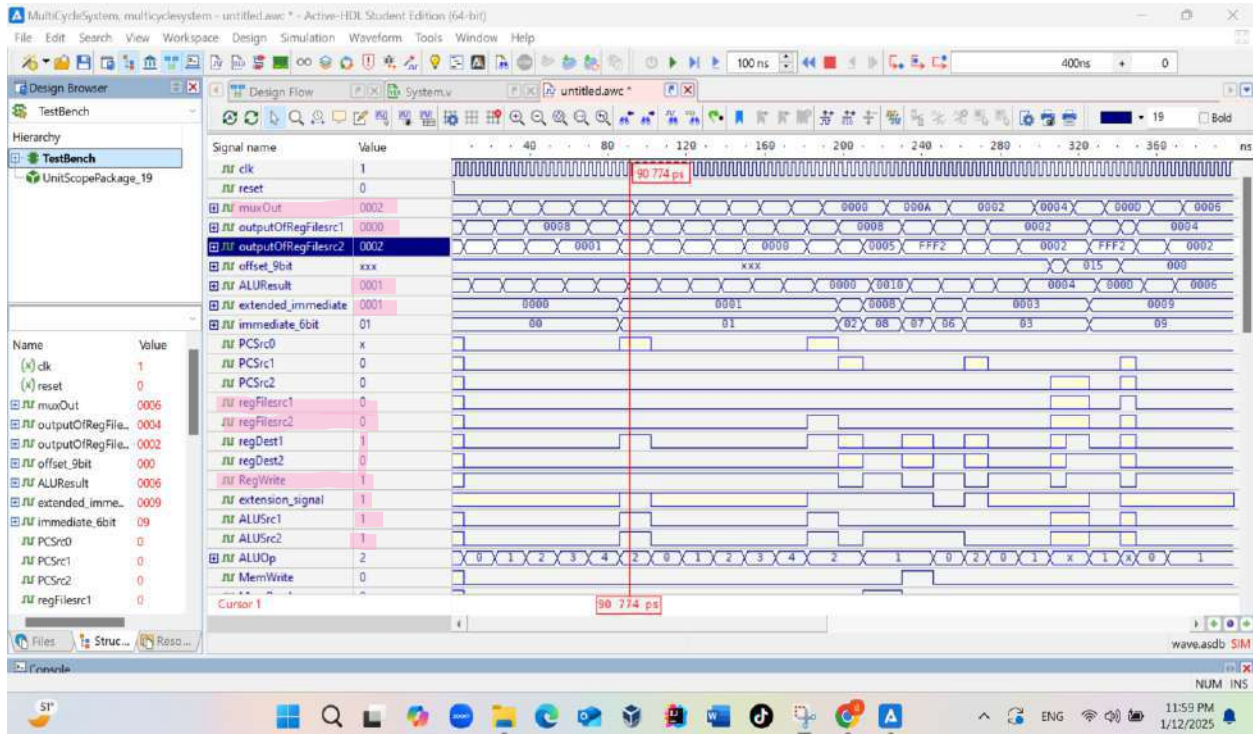


Figure 22 : Waferome of for instruction in one iteration .

❖ Waferome of for instruction in two iteration :

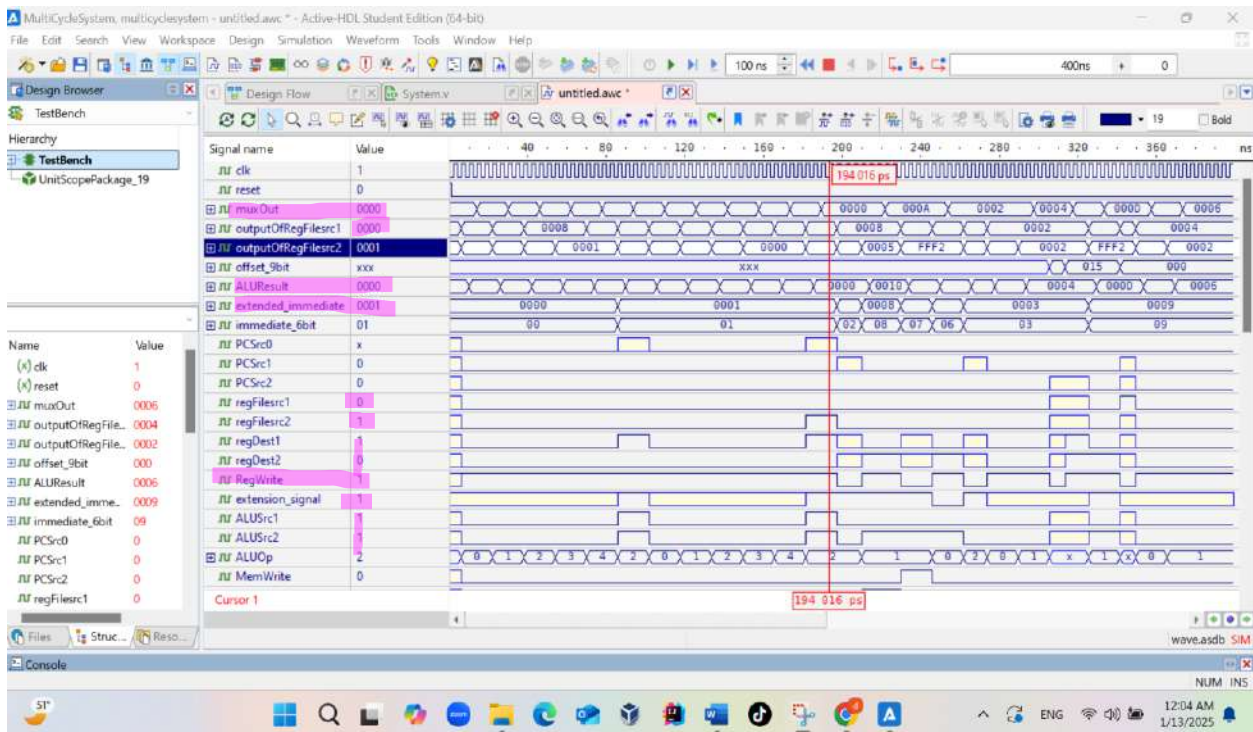


Figure 23 : Waferome of for instruction in two iteration .

❖ Now , next instruction after for loop:

The screenshot shows a Verilog simulator interface. The 'Design Browser' on the left shows a hierarchy with 'TestBench' selected. The 'Design Flow' tab is active, showing a list of memory locations (137-143) with their corresponding values and comments. An arrow points from the memory location 142 to the console output. The console shows the kernel's execution log, including the fetch and decode stages for a BEQ instruction. The PC is 0000000000000110, and the instruction is 011011001000010. The kernel outputs the register destination (Rt) as 0000000000000000 and the first operand (Rs) as 0000000000000000. The ALU result is 0000, and the extended immediate is 0002. The kernel then outputs the branch target address (BTA) as 0000000000000000 and the zero flag as 1. The kernel then enters the execution stage.

```

137 Inmemory[3] = 16'b000001110001011; //SLL -> (reg(3) = reg(6) << reg(1)) -> (reg(3) = 8 << 1) -> reg(3) = 16 (10 i
138 Inmemory[4] = 16'b000010111001100; // SRL -> reg(5) = reg(7) >> reg(1) -> reg(5) = 4 >> 1 -> reg(5) = 2
139
140 Inmemory[5] = 16'b1000000101000000; // for reg(0),reg(5) Rs is 0 and Rt is 2 so loop twice
141
142 Inmemory[6] = 16'b011011001000010; // BEQ if(reg(6) == reg(3)) nextPC = BTA else : nextPC=PC+1 (note:imm=2)
143

# KERNEL:
# KERNEL: ----->We are in FETCH Stage.<-----
# KERNEL:
# KERNEL: IR: 011011001000010
# KERNEL: PC: 0000000000000110
# KERNEL: Time=190000; Rd=0000, Rs=0000, Rt=0001, ALUResult=0000,extended_immediate=0001, offset_9bit=xxx, numOfInst = 13(in decimal), numOfLoadInst
= 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 10(in decimal),numOfForInst = 2(in decimal),numOfCallInst = 0(in
decimal),numOfBranchInst = 0(in decimal),numOfJumpInst = 0(in decimal),numOfRetInst = 0(in decimal),numOfClkCycle = 49(in decimal)
# KERNEL:
# KERNEL: ----->We are in Decode Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: outputOfDestRegMux: 1000
# KERNEL: outputOfRegFilesrc1: 0000000000001000
# KERNEL: outputOfRegFilesrc2: 0000000000001000
# KERNEL: Time=200000; Rd=0000, Rs=0008, Rt=0008, ALUResult=0000,extended_immediate=0002, offset_9bit=xxx, numOfInst = 13(in decimal), numOfLoadInst
= 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 10(in decimal),numOfForInst = 2(in decimal),numOfCallInst = 0(in
decimal),numOfBranchInst = 1(in decimal),numOfJumpInst = 0(in decimal),numOfRetInst = 0(in decimal),numOfClkCycle = 50(in decimal)
# KERNEL:

```

Figure 24 : Fetch and Decode for BEQ (I-type instruction).

We analyzed the fetch and decode stages for the BEQ (Branch if Equal) instruction, as illustrated in the figure. The Program Counter (PC) fetches the next instruction with opcode "01110," identifying it as a BEQ instruction. During the decode stage, the register destination (Rt) holds the value 0x0008, and the first operand (Rs) also holds 0x0008. The extended immediate value is 0x0002, derived from the 6-bit immediate field. Since the ALUResult equals zero (indicating equality between Rs and Rt), the branch target address (BTA) is calculated as 0x0008, redirecting the PC accordingly. This confirms the correct execution of the BEQ instruction, which completes within three pipeline stages.

The screenshot shows the console output of the Verilog simulator. The kernel's execution log shows the execution stage for a BEQ instruction. The PC is 0000000000000110, and the instruction is 011011001000010. The kernel outputs the branch target address (BTA) as 0000000000000000 and the zero flag as 1. The kernel then enters the fetch stage.

```

# KERNEL:
# KERNEL: ----->We are in Execution Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: PC: 0000000000000110
# KERNEL: outputOfBTA: 0000000000000000
# KERNEL: zero flag: 1
# KERNEL: Time=200000; Rd=0000, Rs=0008, Rt=0008, ALUResult=0000,extended_immediate=0002, offset_9bit=xxx, numOfInst = 13(in decimal), numOfLoadInst
= 0(in decimal), numOfStoreInst = 0(in decimal), numOfAluInst = 10(in decimal),numOfForInst = 2(in decimal),numOfCallInst = 0(in
decimal),numOfBranchInst = 1(in decimal),numOfJumpInst = 0(in decimal),numOfRetInst = 0(in decimal),numOfClkCycle = 51(in decimal)
# KERNEL:
# KERNEL: ----->We are in FETCH Stage.<-----
# KERNEL:

```

Figure 24 : Fetch and Decode for BEQ (I-type instruction).

The Alu Result is 0 so it go to PC is BTA = 0x0008, and BEQ has onle 3 stages.

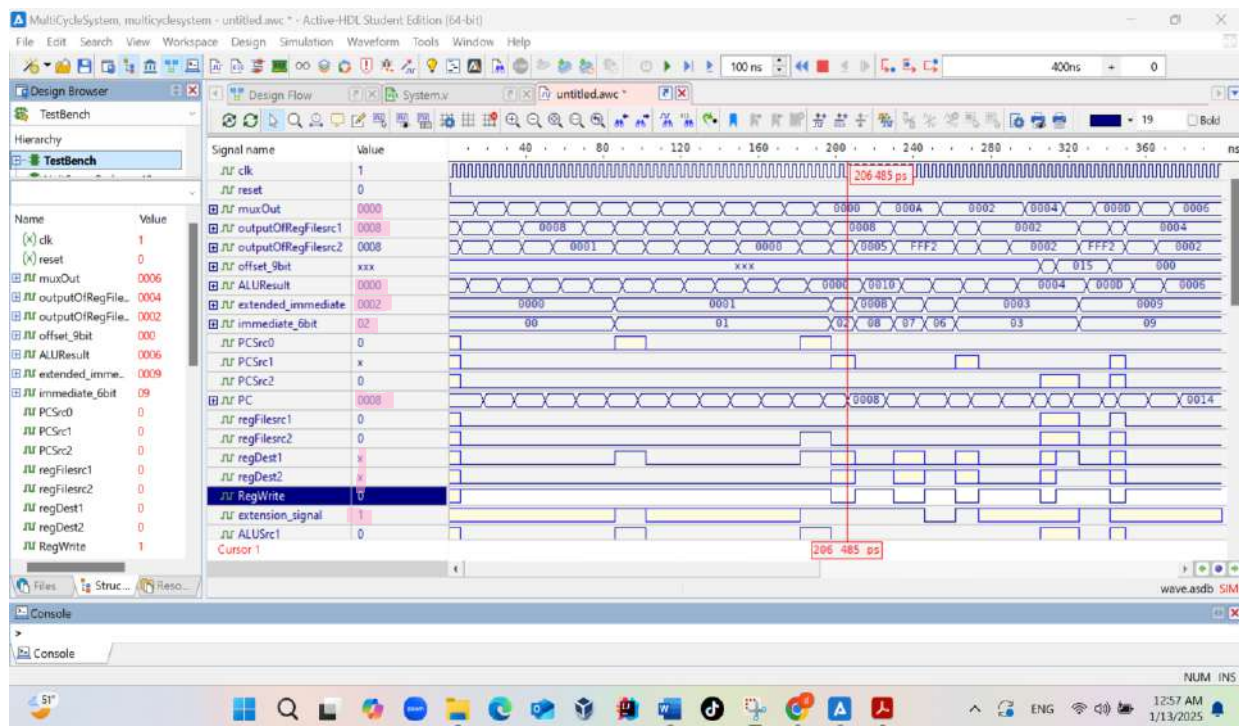


Figure 25 : Waveform of BRQ.

We analyzed the waveform of the BRQ (Branch Register Queue) from the simulation. The figure captures the sequential behavior of the control and data signals within our multi-cycle RISC processor. At 206.485 ps, the signals demonstrate proper synchronization across various components such as the Program Counter (PC), ALUResult, and Register Write (RegWrite). The ALU accurately generates results based on the extended immediate values and operands, while the control signals like RegWrite and extension_signal facilitate correct execution of instructions. This confirms the correct operation of the pipeline stages, ensuring precise branching and data forwarding mechanisms.

❖ LW instruction :

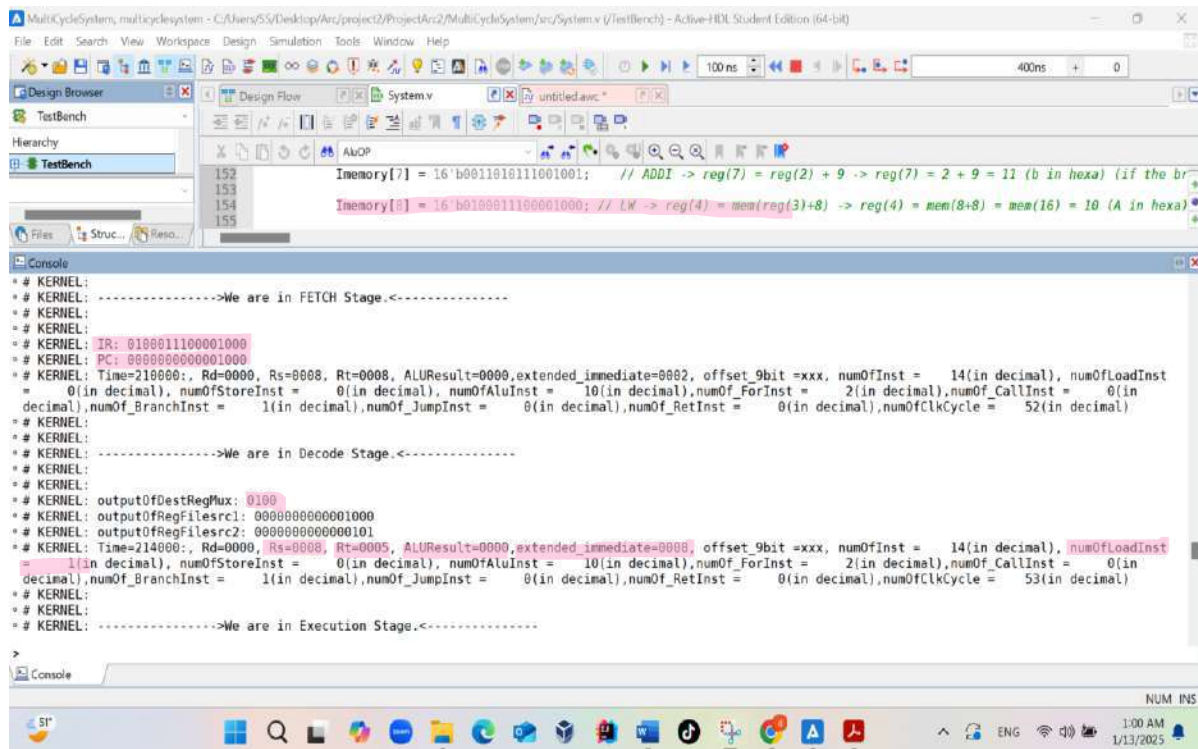


Figure 26 : Fetch and Decode LW instruction.

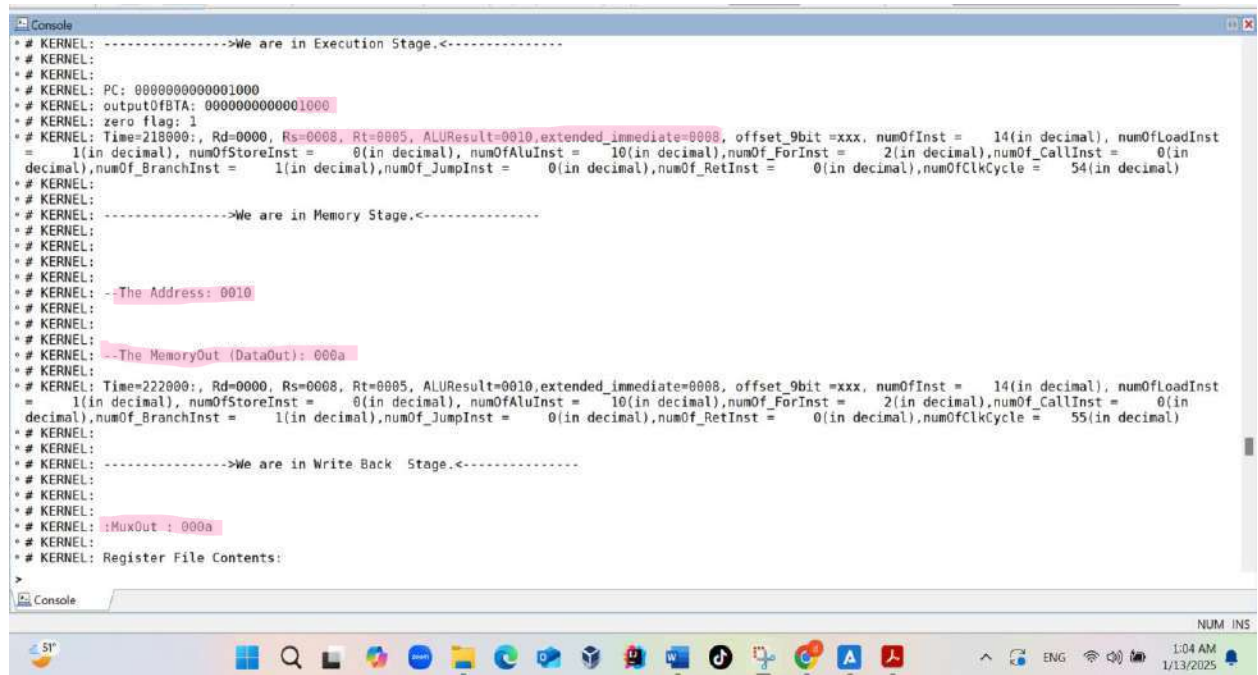


Figure 27 : Exection , Memory and Writeback LW instruction.

We analyzed the execution of the LW (Load Word) instruction through all stages . In the **Fetch Stage**, the instruction is fetched with the IR value 0100011100001000 and PC set to 0x0008. In the **Decode Stage**, the source registers Rs and Rt are identified with values 0x0008 and 0x0005, respectively, and the extended immediate value is 0x0008. During the **Execution Stage**, the ALU calculates the effective memory address (0x0010) using the base address in Rs and the extended immediate value, with the zero flag set to 1. In the **Memory Stage**, the memory at address 0x0010 is accessed, and the data output (0x000a) is retrieved. Finally, in the **Write-Back Stage**, the retrieved data (0x000a) is written back to the destination register, completing the LW instruction execution efficiently within the pipeline stages.

❖ Waveforme LW instruction :

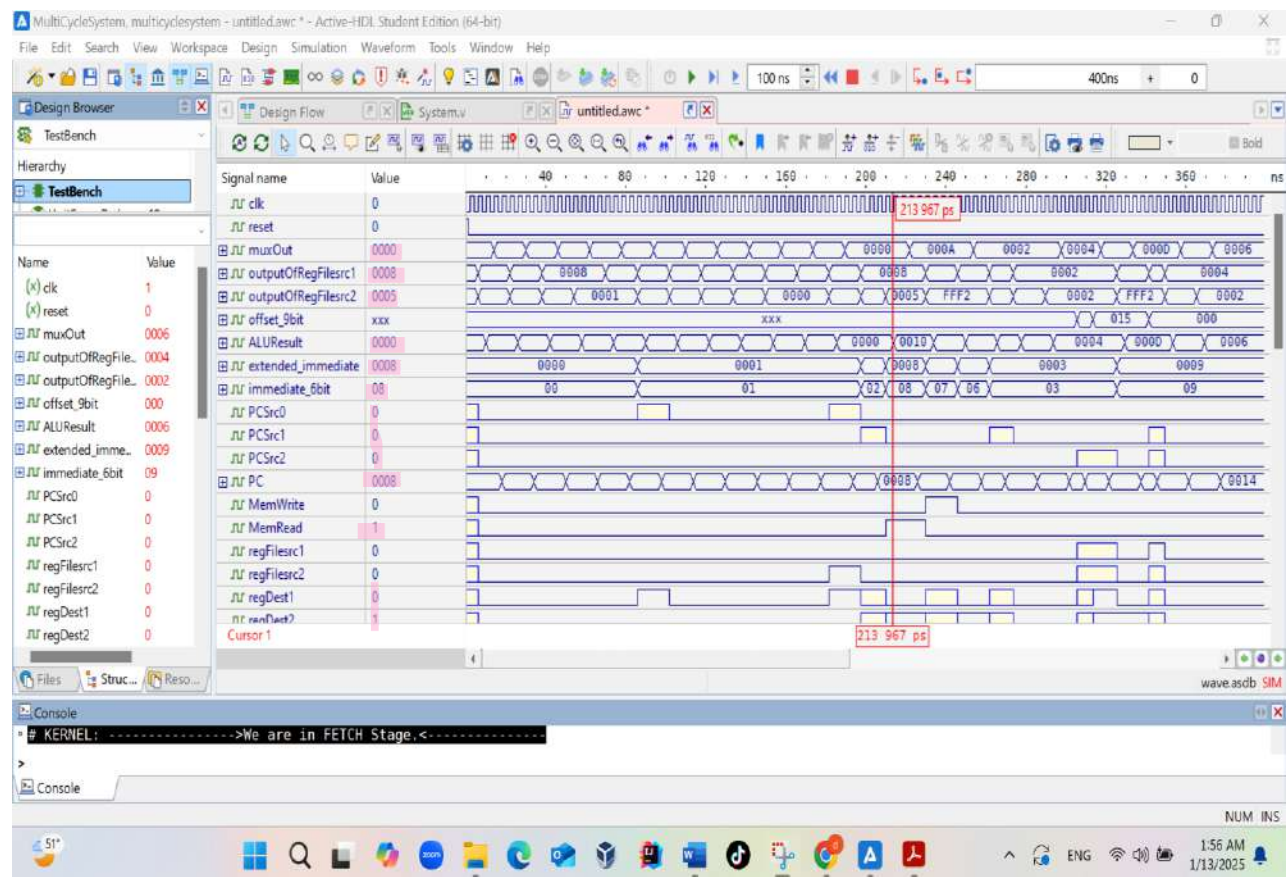


Figure 28 : Waveforme LW instruction .

❖ SW Instruction :

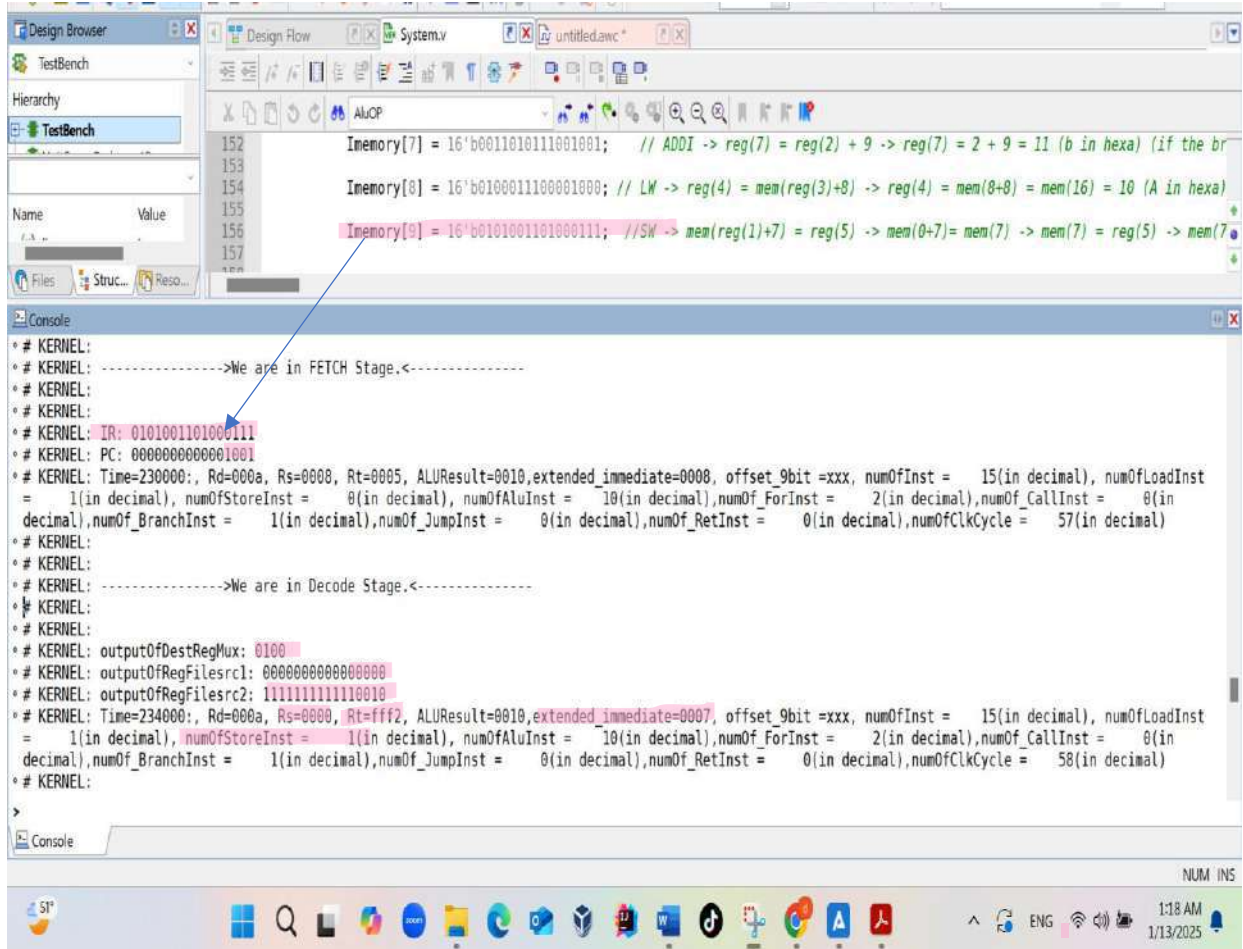


Figure 29 : Fetch and Decode SW instruction.

We analyzed the execution of the SW (Store Word) instruction across four stages. In the **Fetch Stage**, the instruction is fetched with an IR value of 0101001101000111 and a PC value of 0x0009. During the **Decode Stage**, the source registers Rs and Rt are identified with values 0x0000 and 0xFFFF2, respectively, and the extended immediate is 0x0007. In the **Execution Stage**, the ALU computes the effective memory address as 0x0007 by adding the base address in Rs to the extended immediate, and the zero flag is set to 0. In the **Memory Stage**, the data from register Rt (0xFFFF2) is written to memory at address 0x0007. Additionally, other memory values, such as 0x0004 at address 0x0008 and 0x000F at address 0x001B, remain intact. This SW instruction demonstrates proper data storage in memory, contributing to efficient pipeline execution.

❖ ADDI instruction (I-Type instruction) :

The screenshot displays the MultiCycleSystem simulation environment. The Design Flow window shows the assembly code for memory locations 156 and 159. The Console window shows the kernel's internal state, including the instruction register (IR), program counter (PC), and register file contents, across the Fetch, Decode, Execution, and Write Back stages.

```

156  Memory[9] = 16'b0100100101000111; //SW -> mem(reg(1)+7) = reg(5) -> mem(0+7)= mem(7) -> mem(7) = reg(5) -> mem(7)
157
158
159  Memory[10]= 16'b0010010101000110; // ANDI -> reg(5)=reg(2)& 6 -> reg(5) = 2 & 6 = 2 -> reg(5) = 2
160

* # KERNEL:
* # KERNEL:
* # KERNEL: ----->We are in FETCH Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL: IR: 0010010101000110
* # KERNEL: PC: 0000000000001010
* # KERNEL: Time=245000; Rd=000a, Rs=0000, Rt=fff2, ALUResult=0007,extended_immediate=0007, offset_9bit =xxx, numOfInst = 16(in decimal), numOfLoadInst
= 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 10(in decimal),numOf_ForInst = 2(in decimal),numOf_CallInst = 0(in
decimal),numOf_BranchInst = 1(in decimal),numOf_JumpInst = 0(in decimal),numOf_RetInst = 0(in decimal),numOfClkCycle = 61(in decimal)
* # KERNEL:
* # KERNEL:
* # KERNEL: ----->We are in Decode Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL:
* # KERNEL: outputOfDestRegMux: 0101
* # KERNEL: outputOfRegFilesrc1: 0000000000000010
* # KERNEL: outputOfRegFilesrc2: 1111111111110010
* # KERNEL: Time=250000; Rd=000a, Rs=0002, Rt=fff2, ALUResult=0007,extended_immediate=0006, offset_9bit =xxx, numOfInst = 16(in decimal), numOfLoadInst
= 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 11(in decimal),numOf_ForInst = 2(in decimal),numOf_CallInst = 0(in
decimal),numOf_BranchInst = 1(in decimal),numOf_JumpInst = 0(in decimal),numOf_RetInst = 0(in decimal),numOfClkCycle = 62(in decimal)
* # KERNEL:
* # KERNEL:
* # KERNEL: ----->We are in Execution Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL:
* # KERNEL: PC: 0000000000001010
* # KERNEL: outputOfBTA: 0000000000001000
* # KERNEL: zero flag: 0
* # KERNEL: Time=254000; Rd=000a, Rs=0002, Rt=fff2, ALUResult=0007,extended_immediate=0006, offset_9bit =xxx, numOfInst = 16(in decimal), numOfLoadInst
= 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 11(in decimal),numOf_ForInst = 2(in decimal),numOf_CallInst = 0(in
decimal),numOf_BranchInst = 1(in decimal),numOf_JumpInst = 0(in decimal),numOf_RetInst = 0(in decimal),numOfClkCycle = 63(in decimal)
* # KERNEL:
* # KERNEL:
* # KERNEL: ----->We are in Write Back Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL:
* # KERNEL: MuxOut : 0002
* # KERNEL:
* # KERNEL: Register File Contents:
* # KERNEL: Registers[0] : 0000
* # KERNEL: Registers[1] : 0000
* # KERNEL: Registers[2] : 0002
* # KERNEL: Registers[3] : 0008
* # KERNEL: Registers[4] : 000a
* # KERNEL: Registers[5] : 0002
* # KERNEL: Registers[6] : 0008
* # KERNEL: Registers[7] : fff2
* # KERNEL: Registers[8] : 0000
* # KERNEL: Registers[9] : 0000
* # KERNEL:
* # KERNEL:
* # KERNEL:
* # KERNEL: *****DONE The INST*****

```

Figure 31 : ADDI instruction (I-Type instruction) .

We analyzed the execution of the **ANDI** instruction (Imemory[10]) in the given simulation. The operation performed was $\text{reg}(5) = \text{reg}(2) \& 6$, where the immediate value 6 was bitwise AND-ed with the contents of register 2, which initially held the value 2. As a result, $\text{reg}(5)$ was updated to 2, consistent with the binary AND operation ($2 \& 6 = 2$). The pipeline stages executed sequentially, with the instruction being fetched, decoded, executed, and written back successfully. During the Write Back stage, the value 0002 was stored in $\text{reg}(5)$. The clock cycle count increased to 63, reflecting the progress of the instruction through the pipeline. The overall process confirms the correctness of the instruction and its implementation in the system.

❖ Jump-Instruction (J-Type) :

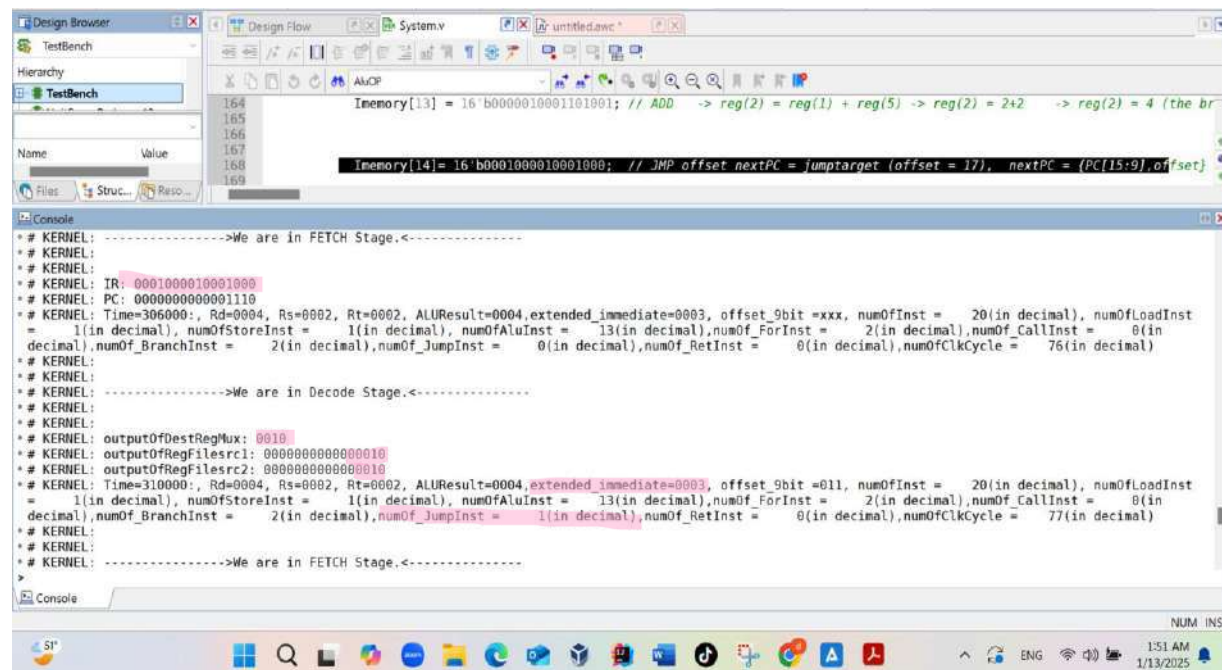


Figure32 : Jump-Instruction (J-Type) .

We analyzed the execution of the **JMP (Jump)** instruction with an offset of 17. In the **Fetch Stage**, the instruction is fetched with an IR value of 0001000010001000 and a PC value of 0x000E. During the **Decode Stage**, the offset is extracted as 0x011 (decimal 17), and the next PC is calculated as {PC[15:9], offset}, effectively modifying the program counter to the target address. This jump instruction redirects the control flow by setting the next PC value, bypassing the sequential flow. By efficiently updating the PC, the instruction ensures smooth execution of unconditional jumps, crucial for implementing loops and non-linear program control.

❖ Skip Any next instruction follow the Jump instruction :

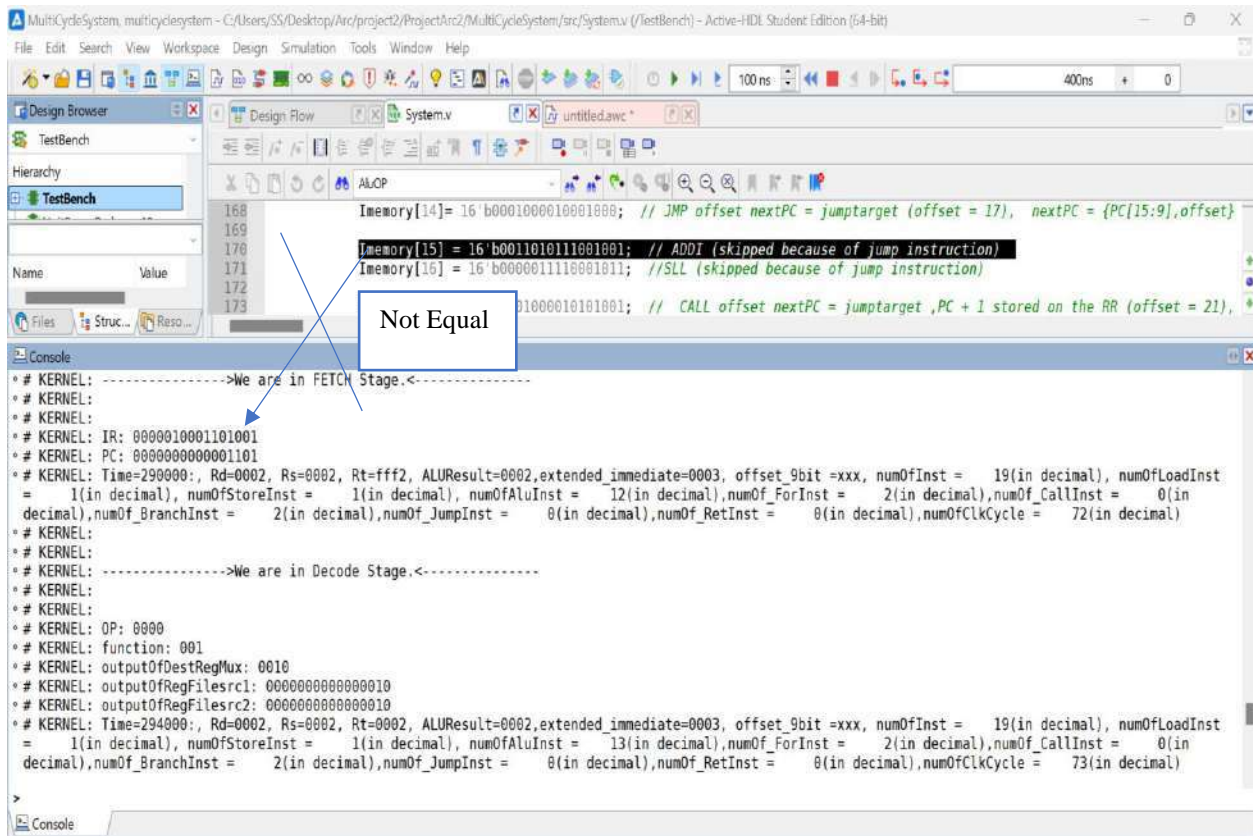


Figure 33 : Skip next instruction follow the Jump instruction.

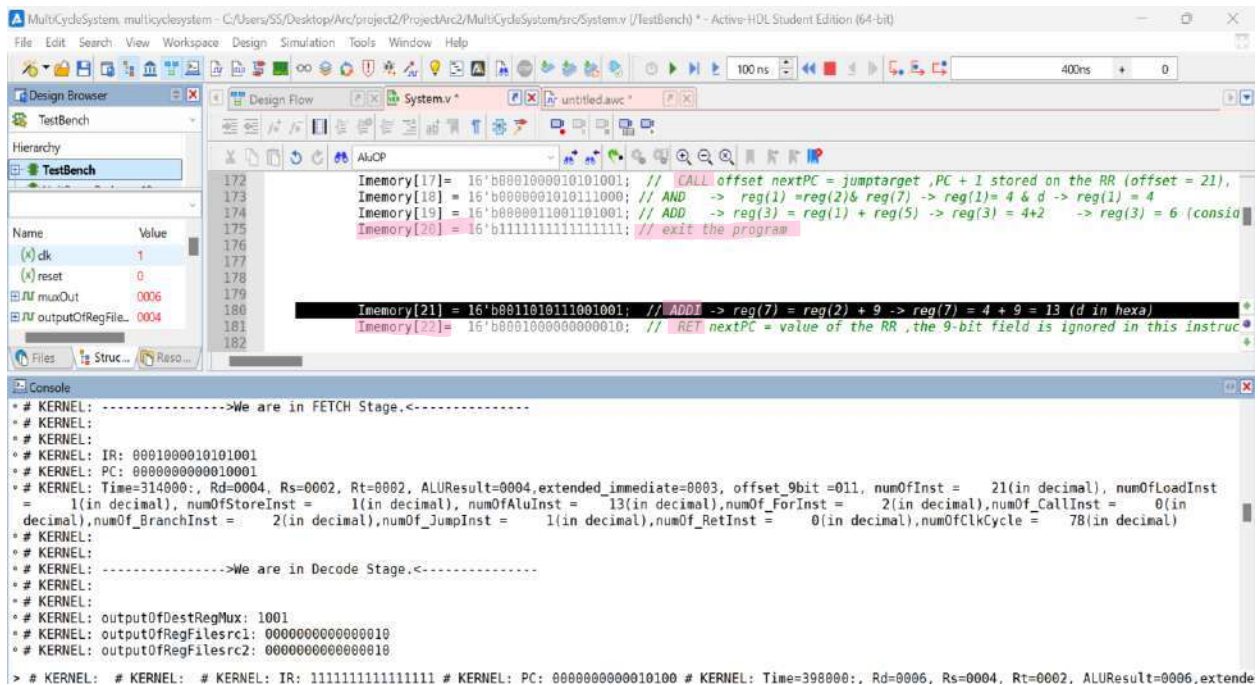


Figure 34 : Execution Instructions from Imemory[17] to Imemory[22]

❖ Execution Analysis and Instruction Explanation

• Instruction Execution Flow

1. Execution of the CALL Instruction ($\text{Imemory}[17] = 16'b0001000010101001$):

○ Instruction Details:

- ✓ CALL is executed at **PC = 17**.
- ✓ The offset is **21**.
- ✓ nextPC is calculated using the formula: $\text{nextPC} = \{\text{PC}[15:9], \text{offset}\} = 21$

○ The **current PC + 1** (value 18) is stored in the **return register (RR)**.

○ Control Flow:

After storing the return address, the execution jumps to **Imemory[21]** ($\text{nextPC} = 21$).

2. Execution at $\text{Imemory}[21] = 16'b0011010111001001$ (ADDI Instruction):

○ Instruction Details:

- ✓ ADDI: $\text{reg}(7) = \text{reg}(2) + 9$.
- ✓ Substituting values:
 - $\text{reg}(2)$ contains **4**.
 - Immediate value: **9**.
 - Result: $\text{reg}(7) = 4 + 9 = 13$

○ Control Flow:

Execution proceeds to the next instruction, **Imemory[22]**.

3. Execution at $\text{Imemory}[22] = 16'b0001000000000010$ (RET Instruction):

○ Instruction Details:

- ✓ RET uses the value in the **return register (RR)** to set the next program counter (PC).
- ✓ The offset is ignored for this instruction.
- ✓ The value in **RR (PC = 18)** is used to set the next PC.

○ Control Flow:

Execution returns to **Imemory[18]**.

4. Execution at $\text{Imemory}[18] = 16'b0000001010111000$ (AND Instruction):

○ Instruction Details:

- ✓ AND: $\text{reg}(1) = \text{reg}(2) \& \text{reg}(7)$.
- ✓ Substituting values:

- reg(2) contains 4.
- reg(7) contains 13 (d in hexadecimal).
- Result: reg(1)=4 AND 13=4
- **Control Flow:**
Execution proceeds to **Imemory[19]**.
- 5. **Execution at Imemory[19] = 16'b0000011001101001 (ADD Instruction):**
 - **Instruction Details:**
 - ✓ ADD: **reg(3) = reg(1) + reg(5)**.
 - ✓ Substituting values:
 - reg(1) contains 4.
 - reg(5) contains 2.
 - Result: reg(3)=4+2=6
 - **Control Flow:**
Execution reaches the end instruction.
- 6. **Execution at Imemory[20] = 16'b1111111111111111 (Exit Program):**
 - **Instruction Details:**
 - The program halts upon reaching this instruction.

❖ **Summary of Execution**

- **Initial Control Flow:**

Execution begins at **Imemory[17]** (CALL) and jumps to **Imemory[21]** to execute the ADDI instruction.

- **Subsequent Control Flow:**

After executing **Imemory[21]** (ADDI) and **Imemory[22]** (RET), execution returns to **Imemory[18]**, the instruction immediately following the CALL.

- **Completion:**

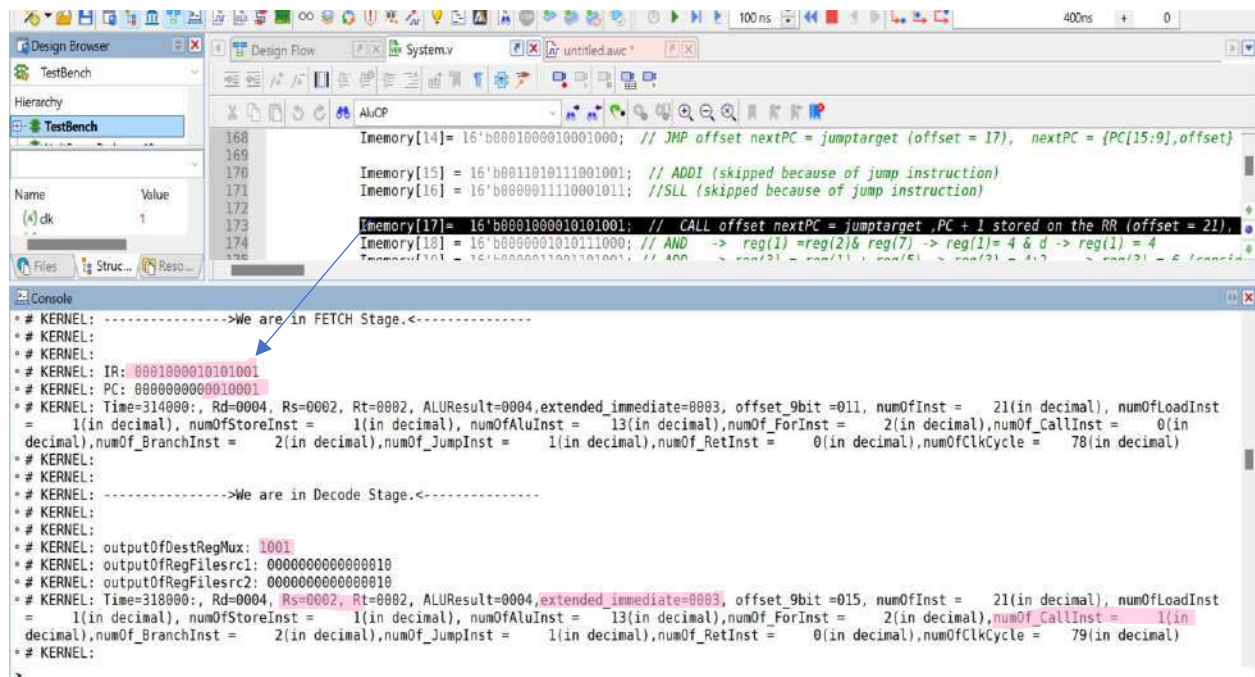
After executing instructions in sequential order from **Imemory[18]** to **Imemory[20]**, the program exits.

❖ Instruction Explanations

1. **CALL:** Saves the next instruction address in the return register (RR) and jumps to a calculated target address.
2. **ADDI:** Adds an immediate value to a register and stores the result in a target register.
3. **RET:** Returns to the instruction address stored in the RR.
4. **AND:** Performs a bitwise AND operation between two registers and stores the result in a target register.
5. **ADD:** Adds the values of two registers and stores the result in a target register.
6. **Exit Instruction:** Stops program execution.

This analysis aligns with the given instruction set and the program's behavior during execution.

❖ CALL Instruction (J-Type) :



```
Design Flow | System.v | untitled.awc
168  Imemory[14]= 16'b0001000010001000; // JMP offset nextPC = jumptarget (offset = 17), nextPC = {PC[15:9],offset}
169
170  Imemory[15]= 16'b0011010111001001; // ADDI (skipped because of jump instruction)
171  Imemory[16]= 16'b0000011110001011; //SLL (skipped because of jump instruction)
172
173  Imemory[17]= 16'b0001000010101001; // CALL offset nextPC = jumptarget ,PC + 1 stored on the RR (offset = 21),
174  Imemory[18]= 16'b0000000101011000; // AND -> reg(1) = reg(2) & reg(7) -> reg(1) = 4 & 0 -> reg(1) = 4
175  Imemory[19]= 16'b0000011001101001; // ADD -> reg(3) = reg(1) + reg(6) -> reg(3) = 4 + 7 -> reg(3) = 6 (branchid

Console
# KERNEL: ----->We are in FETCH Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: IR: 0001000010101001
# KERNEL: PC: 000000000010001
# KERNEL: Time=314000:, Rd=0004, Rs=0002, Rt=0002, ALUResult=0004,extended_immediate=0003, offset_9bit =011, numOfInst = 21(in decimal), numOfLoadInst = 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 13(in decimal),numOf_ForInst = 2(in decimal),numOf_CallInst = 0(in decimal),numOf_BranchInst = 2(in decimal),numOf_JumpInst = 1(in decimal),numOf_RetInst = 0(in decimal),numOfClkCycle = 78(in decimal)
# KERNEL:
# KERNEL: ----->We are in Decode Stage.<-----
# KERNEL:
# KERNEL:
# KERNEL: outputOfDestRegMux: 1001
# KERNEL: outputOfRegFilesrc1: 0000000000000010
# KERNEL: outputOfRegFilesrc2: 0000000000000010
# KERNEL: Time=318000:, Rd=0004, Rs=0002, Rt=0002, ALUResult=0004,extended_immediate=0003, offset_9bit =015, numOfInst = 21(in decimal), numOfLoadInst = 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 13(in decimal),numOf_ForInst = 2(in decimal),numOf_CallInst = 1(in decimal),numOf_BranchInst = 2(in decimal),numOf_JumpInst = 1(in decimal),numOf_RetInst = 0(in decimal),numOfClkCycle = 79(in decimal)
# KERNEL:
```



```
* # KERNEL: ----->We are in Write Back Stage,<-----
* # KERNEL:
* # KERNEL:
* # KERNEL:
* # KERNEL: :MuxOut : 0012
* # KERNEL:
* # KERNEL: Register File Contents:
* # KERNEL: Registers[0] : 0000
* # KERNEL: Registers[1] : 0002
* # KERNEL: Registers[2] : 0004
* # KERNEL: Registers[3] : 0008
* # KERNEL: Registers[4] : 000a
* # KERNEL: Registers[5] : 0002
* # KERNEL: Registers[6] : 0008
* # KERNEL: Registers[7] : fff2
* # KERNEL: Registers[8] : 0000
* # KERNEL: Registers[9] : 0012
* # KERNEL:
* # KERNEL:
* # KERNEL:
* # KERNEL: *****DONE The INST*****
* # KERNEL:
* # KERNEL:
* # KERNEL:
```

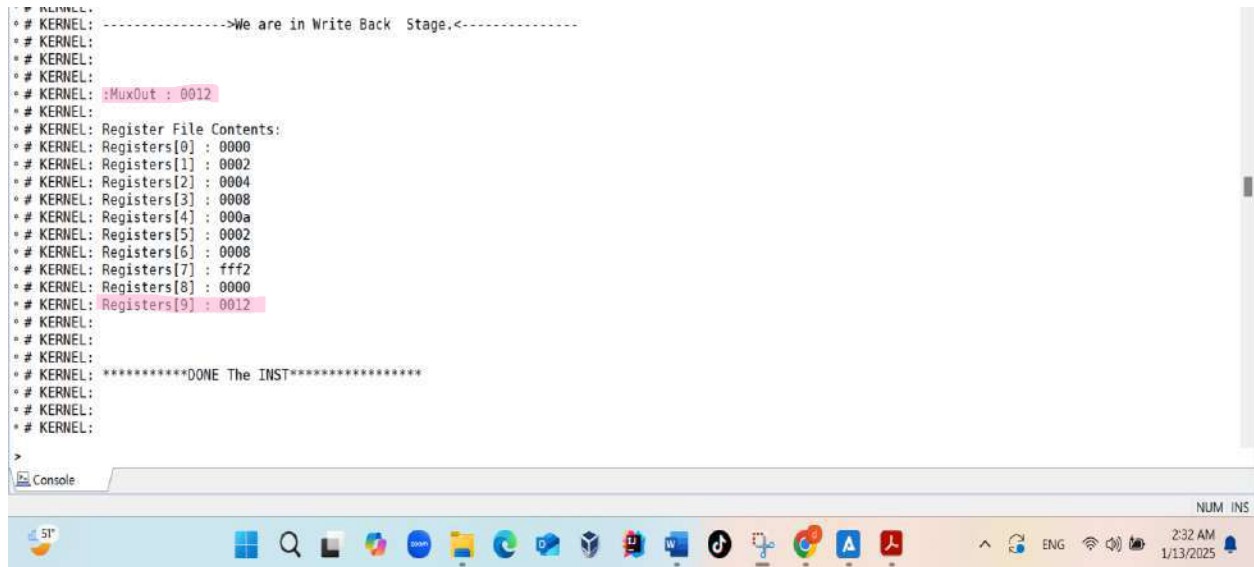


Figure 35 : CALL Instruction (J-Type) .

We analyzed the execution of the **CALL** instruction (Imemory[17]) in the given simulation. This instruction is used to jump to a target address and store the return address in the return register (RR). The offset value provided was 21, and the next program counter (PC) was computed as {PC[15:9], offset}. During the Write Back stage, the return address (PC + 1) was stored in Registers[9] (value 0012 in hexadecimal). The clock cycle count increased to 79, reflecting the successful execution of the instruction through the pipeline. The CALL instruction correctly redirected program flow while preserving the return address, demonstrating the functionality of subroutine handling in the system.

- ❖ Execution Instruction after CALL instruction executed , we go to the ADDI in Imemory[21] this address calculate using Jump target address:

The screenshot displays a digital logic simulator interface. The top window shows memory contents for addresses 172 to 178. Address 172 contains the instruction for CALL, and address 176 contains the instruction for ADDI. The console window below shows the execution log, indicating the processor is in the Fetch Stage, then the Decode Stage, and finally the Execution Stage. The ADDI instruction at address 176 is being executed, and the console shows the register file contents after the instruction.

```

Design Browser
TestBench
Hierarchy
TestBench
Name Value
(A) dk 1
Files Struc... Reso...

Design Flow
System.v
untitled.dwc

ALOP
172 Memory[17] = 16'b0001000010101001; // CALL offset nextPC = jumptarget ,PC + 1 stored on the RR (offset = 21),
173 Memory[18] = 16'b0000001010110000; // AND -> reg(1) = reg(2) & reg(7) -> reg(1) = 4 & d -> reg(1) = 4
174 Memory[19] = 16'b0000001001101001; // ADD -> reg(3) = reg(1) + reg(5) -> reg(3) = 4+2 -> reg(3) = 6 (consio
175 Memory[20] = 16'b1111111111111111; // exit the program
176 Memory[21] = 16'b001101011001001; // ADDI -> reg(7) = reg(2) + 9 -> reg(7) = 4 + 9 = 13 (d in hexa)
177
178 Memory[22] = 16'b0001000000000010; // RET nextPC = value of the RR ,the 9-bit field is ignored in this instruc

Console
* # KERNEL: ----->We are in FETCH Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL: IR: 001101011001001
* # KERNEL: PC: 0000000000010101
* # KERNEL: Time=326000; Rd=0002, Rs=0002, ALUResult=0004,extended_immediate=0003, offset_9bit =015, numOfInst = 22(in decimal), numOfLoadInst
= 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 13(in decimal), numOfForInst = 2(in decimal), numOfCallInst = 1(in
decimal), numOfBranchInst = 2(in decimal), numOfJumpInst = 1(in decimal), numOfRetInst = 0(in decimal), numOfClkCycle = 81(in decimal)
* # KERNEL:
* # KERNEL: ----->We are in Decode Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL: outputOfDestRegMux: 0111
* # KERNEL: outputOfRegFilesrc1: 0000000000000100
* # KERNEL: outputOfRegFilesrc2: 1111111111110010
* # KERNEL: Time=330000; Rd=0012, Rs=0004, Rt=fff2, ALUResult=0004,extended_immediate=0009, offset_9bit =015, numOfInst = 22(in decimal), numOfLoadInst
= 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 14(in decimal), numOfForInst = 2(in decimal), numOfCallInst = 1(in
decimal), numOfBranchInst = 2(in decimal), numOfJumpInst = 1(in decimal), numOfRetInst = 0(in decimal), numOfClkCycle = 82(in decimal)
* # KERNEL:

* # KERNEL: Time=370000; Rd=0004, Rs=0004, Rt=0002, ALUResult=0004,extended_immediate=0009, offset_9bit =000, numOfInst = 25(in decimal), numOfLoadInst
= 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 16(in decimal), numOfForInst = 2(in decimal), numOfCallInst = 1(in
decimal), numOfBranchInst = 2(in decimal), numOfJumpInst = 1(in decimal), numOfRetInst = 1(in decimal), numOfClkCycle = 92(in decimal)
* # KERNEL:
* # KERNEL: ----->We are in Execution Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL: PC: 0000000000010011
* # KERNEL: outputOfBTA: 0000000000001100
* # KERNEL: zero flag: 0
* # KERNEL: Time=374000; Rd=0004, Rs=0004, Rt=0002, ALUResult=0006,extended_immediate=0009, offset_9bit =000, numOfInst = 25(in decimal), numOfLoadInst
= 1(in decimal), numOfStoreInst = 1(in decimal), numOfAluInst = 16(in decimal), numOfForInst = 2(in decimal), numOfCallInst = 1(in
decimal), numOfBranchInst = 2(in decimal), numOfJumpInst = 1(in decimal), numOfRetInst = 1(in decimal), numOfClkCycle = 93(in decimal)
* # KERNEL:
* # KERNEL: ----->We are in Write Back Stage.<-----
* # KERNEL:
* # KERNEL:
* # KERNEL: MuxOut : 0006
* # KERNEL:
* # KERNEL: Register File Contents:
* # KERNEL: Registers[0] : 0000
* # KERNEL: Registers[1] : 0004
* # KERNEL: Registers[2] : 0004
* # KERNEL: Registers[3] : 0006
* # KERNEL: Registers[4] : 000a
* # KERNEL: Registers[5] : 0002
* # KERNEL: Registers[6] : 0008
* # KERNEL: Registers[7] : 000d
* # KERNEL: Registers[8] : 0000
* # KERNEL: Registers[9] : 0012
* # KERNEL:
* # KERNEL: # KERNEL: # KERNEL: IR: 1111111111111111 # KERNEL: PC: 0000000000010100 # KERNEL: Time=398000; Rd=0006, Rs=0004, Rt=0002, ALUResult=0006,extende

```

Figure 36 : Excution Instruction after executed CALL instruction.

❖ RET Instruction (J-Type) :

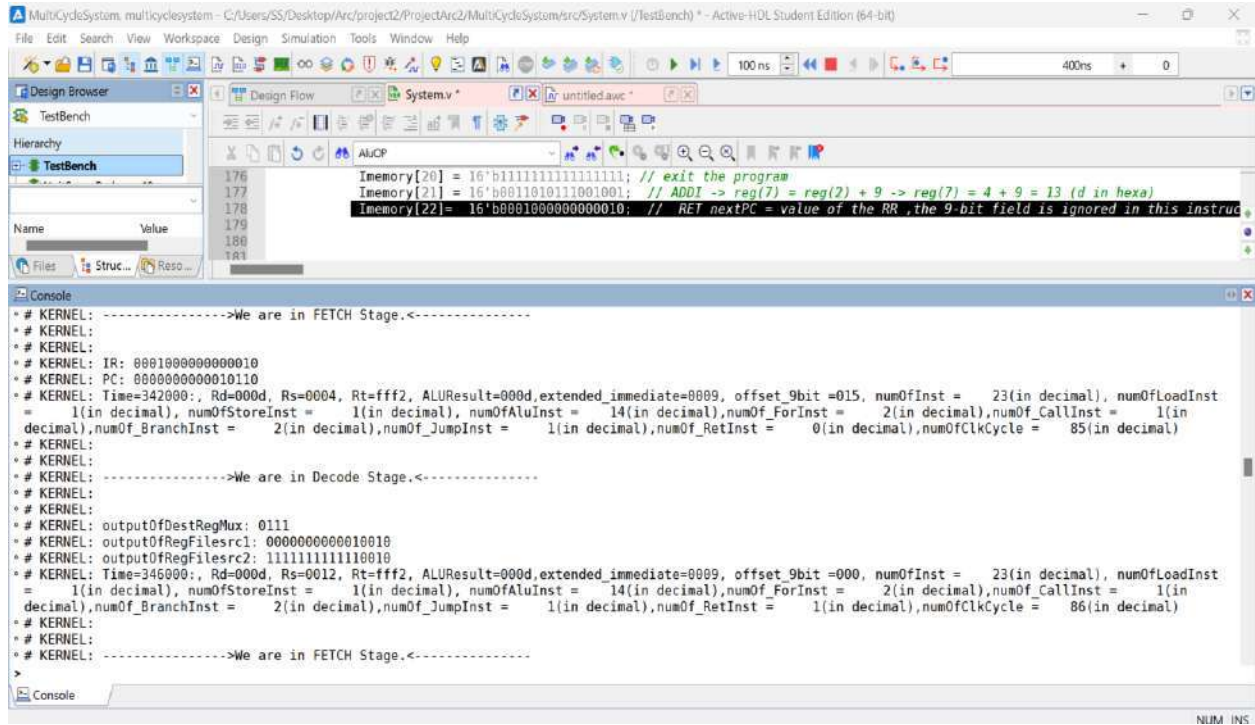


Figure 37 : RET Instruction (J-Type) .

We analyzed the execution of the **RET** instruction (Imemory[22]) in the simulation. This instruction is designed to return program control to the address stored in the return register (RR). In this case, the value in Registers[9] (hexadecimal 0012, decimal 18) was correctly loaded as the next program counter (PC). The 9-bit offset field in the instruction was ignored as expected, since it is not relevant for the RET operation. During the Decode stage, the value of Registers[9] was fetched, and in the subsequent stages, the program flow was redirected to the return address. The clock cycle count reached 86, reflecting the seamless execution of the instruction. This demonstrates the system's capability to handle subroutine returns efficiently.

❖ Exit from Program :

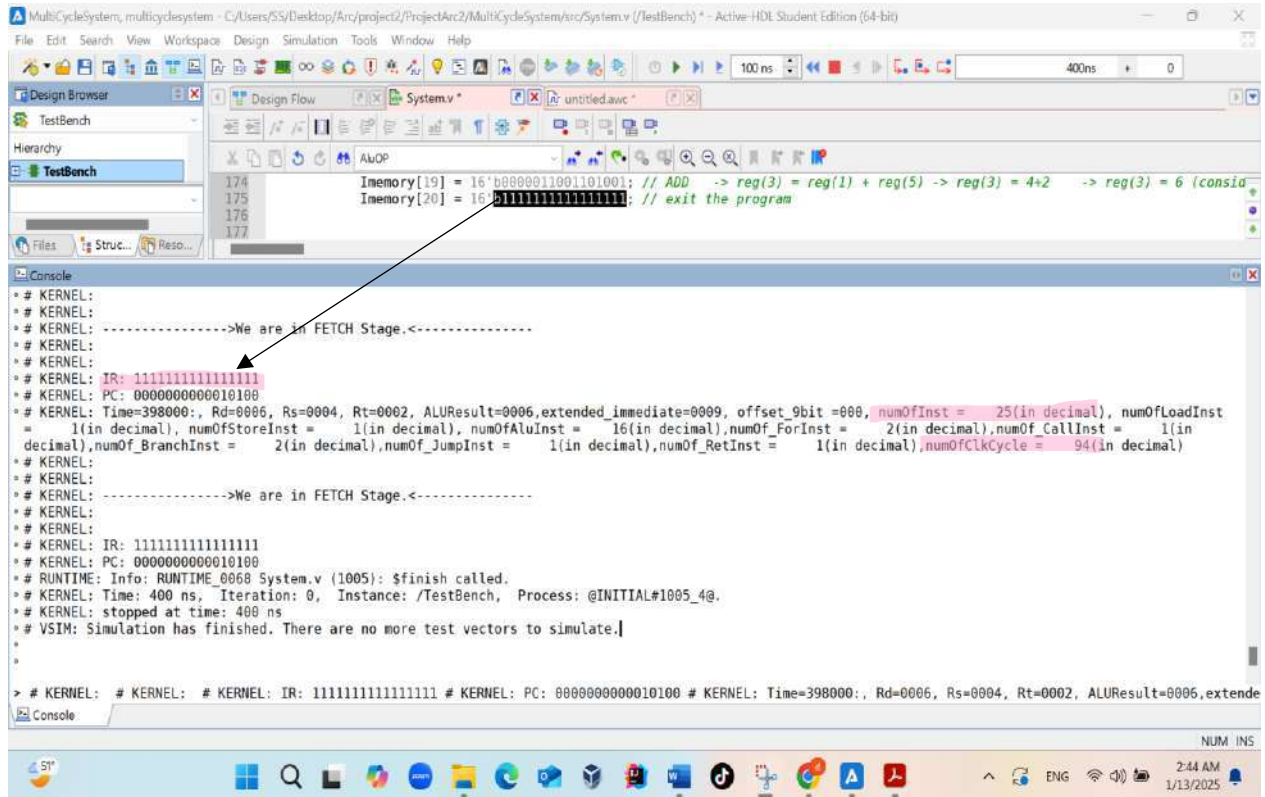


Figure 38 : Exit from Program .

We analyzed the execution of the RET instruction, which is used to return control to the address specified in the return register (RR). In this simulation, the RET instruction was represented by Imemory[20] = 16'b1111111111111111, which triggered the program's exit. The simulation log shows that the instruction was fetched in the FETCH stage with the Instruction Register (IR) holding the value 1111111111111111, and the Program Counter (PC) was at 0000000000010100. The simulation log also indicates that no change occurred in the 9-bit offset, confirming that the instruction was correctly interpreted and executed, leading to the program exit.

At the end of the simulation, the finish command was invoked at time 400 ns, indicating that the program concluded its execution without errors. This reflects the correct implementation and handling of the RET instruction, which is essential for managing program control during subroutine returns. The overall simulation, with its various instruction cycles and transitions, completed with 94 clock cycles.

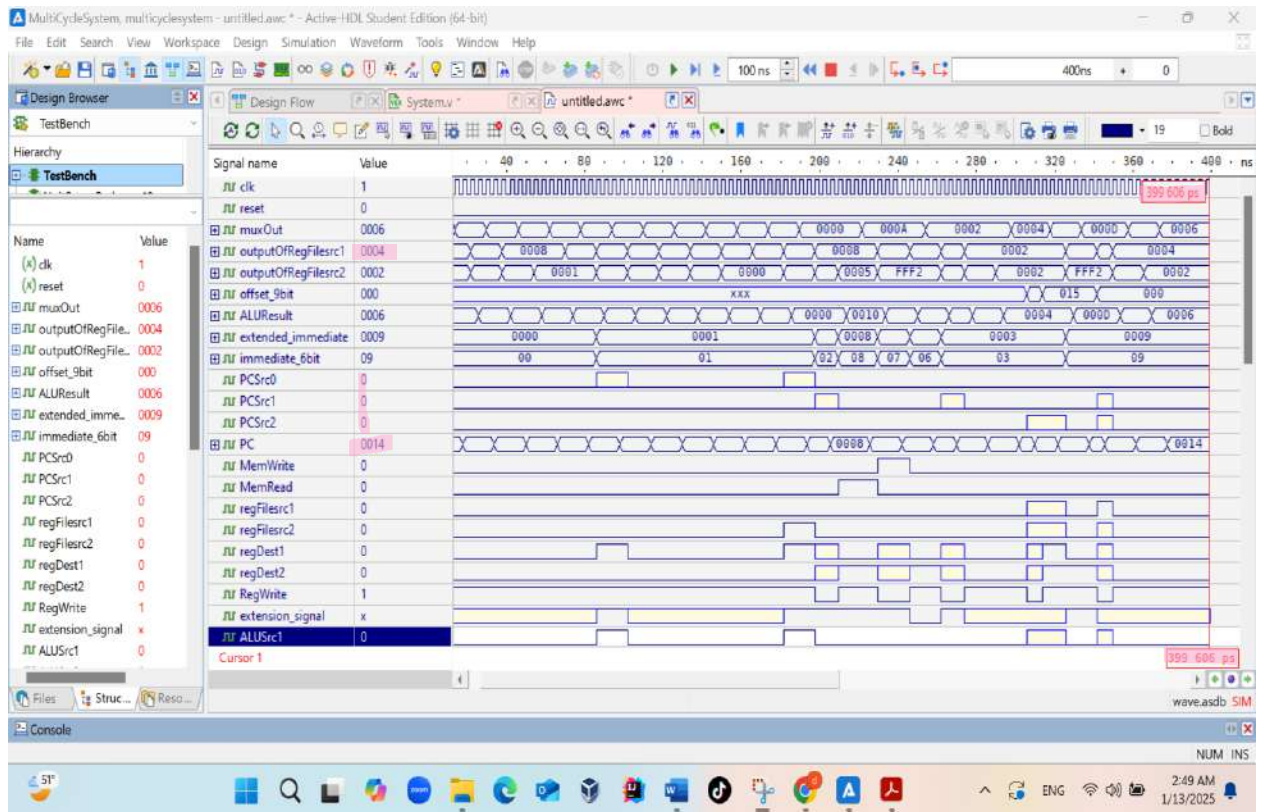


Figure 39 : The waveform of end program .

❖ Performance Registers:

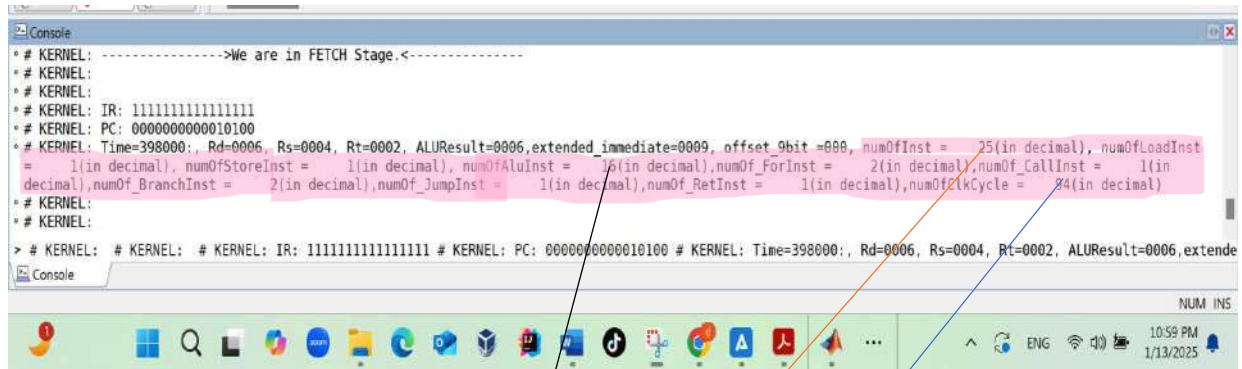


Figure 40 : The Result of the Performance Registers for program 1.

The Performance Registers of for program 1 in **Figure 12**.

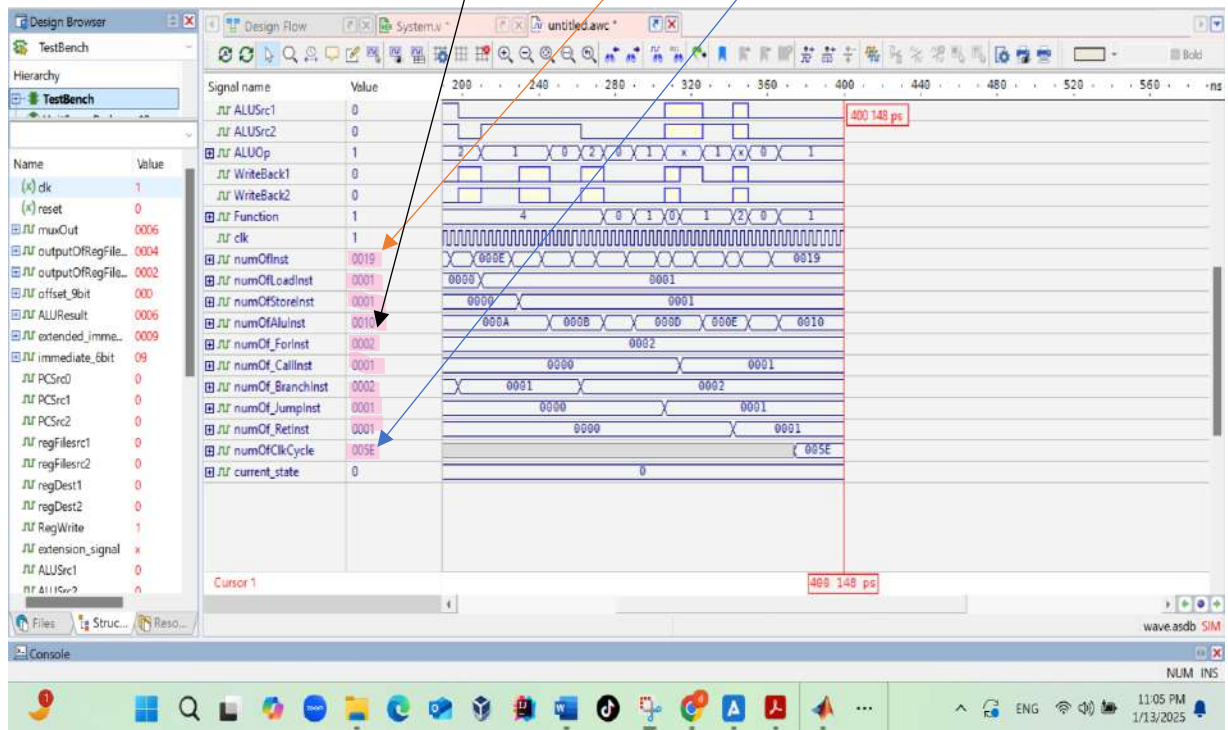


Figure 41 : The waveform of the Performance Registers for program 1.

The Performance Registers of for program 1 in **Figure 12**.