



THE AMERICAN UNIVERSITY IN CAIRO

School of Sciences and Engineering

Fall 2023

CSCE-3302: Computer Architecture Course

Project_1 Milestone_3 Report

Submitted to:

Dr. Cherif Salama

Submitted by:

Name: Fekry Mohamed

ID: 900192372

Name: Mario Ghaly

ID: 900202178

Name: Freddy Amgad

ID: 900203088

26/11/2023

Abstract:

The femtoRV32 project aims to implement a RISC-V processor on the Nexys A7 board, supporting all the RV32I base integer instructions. This implementation features a pipelined architecture with effective hazard handling and utilizes a single memory for both instructions and data. The project includes rigorous testing, covering all 40 instructions and addressing potential hazard scenarios. Additionally, this project supports the integer multiplication and division instructions, along with alternative solutions to mitigate structural hazards.

Introduction:

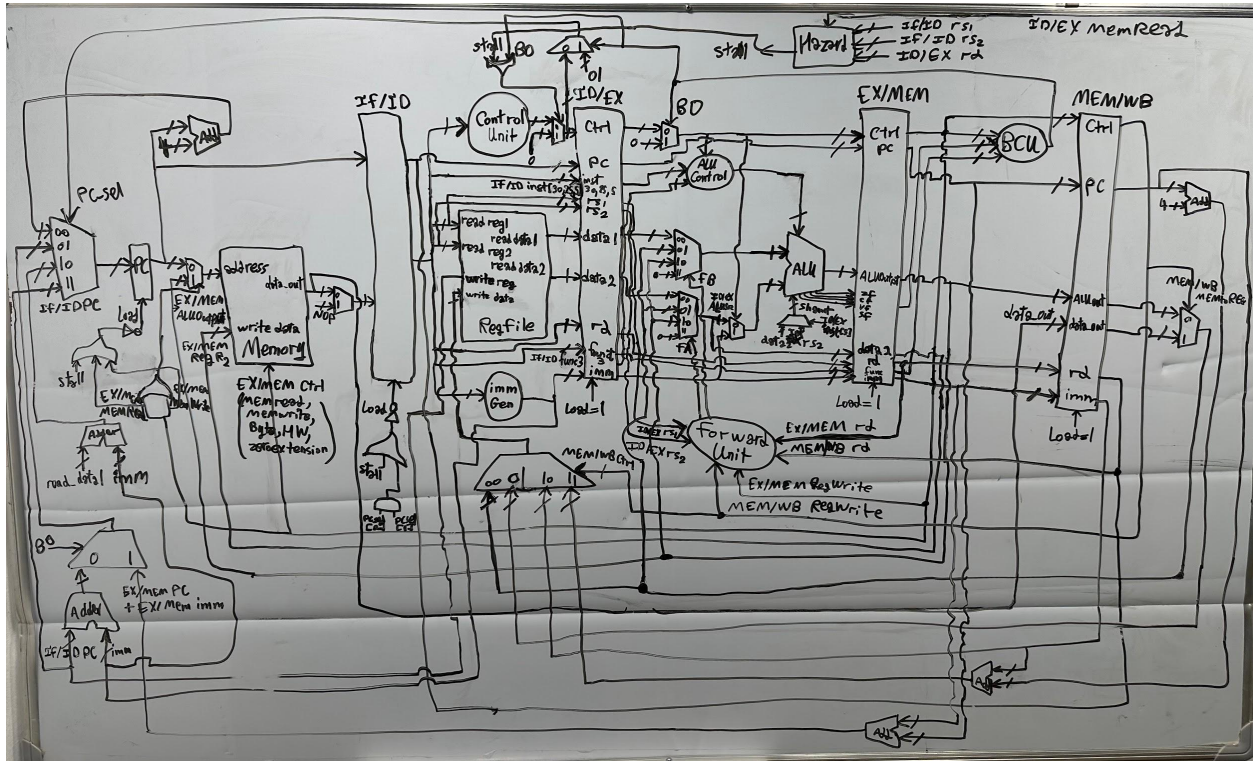
Modern computing systems rely on efficient processor architectures to meet the demands of various applications. The RISC-V instruction set architecture, characterized by its simplicity and modularity, has gained prominence in the field of computer architecture. In this context, the femtoRV32 project seeks to implement a RISC-V processor on the Nexys A7 kit. The project places a strong emphasis on a pipelined architecture, leveraging five stages to execute instructions effectively by introducing an alternative solution to handle the structural hazard. This implementation ensures correct hazard handling, with particular attention to the challenges introduced by using a single-ported memory for both data and instructions. By addressing structural hazards through every-other-cycle instruction issuing, the processor achieves a balance between performance and resource constraints.

Project Objectives:

For milestone 3 of the femtoRV32 project, we implemented the RV32I base integer instruction set for all 40 user-level instructions in the unprivileged ISA. The pipelined is supported in this phase by dividing each instruction into 5 cycles with hazard detection unit to eliminate the potential hazard. All the 40 instructions were implemented as specified in the manual except for ECALL and FENCE, which were implemented as no-op instructions, so we did this by outputting the instruction add x0, x0, x0 from the instruction memory in case any of these two instructions is the target instruction. Also EBREAK was interpreted as a halting instruction ending the execution of the program by preventing loading a new instruction.

The design of the project:

Architecture:



Processor components:

1. Control Unit:

The processor's control unit is the block responsible for generating control signals based on opcode, and function field. Control signals determine various operations within the processor, including ALU operations, memory access, write back operation. Notable control signals include ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, Byte, HalfWord,

ZeroExtention, PC_mux_sel, and WB_mux_sel. These signals collectively orchestrate the execution of instructions within the processor. We added additional control signals, such as load byte and load half word operations, that control the load and store operations of the data memory. Furthermore, the control unit is modified to support the rest of instruction types like I-type and J-type.

2. ALU Control Unit:

The control unit for the Arithmetic Logic Unit (ALU) generates a 4-bit control signal, "ALU_Sel" based on inputs such as "ALUOp" and "funct3". The module employs conditional statements to determine the appropriate ALU operation for different instruction formats. We modified the ALU control unit to support the rest of ALU operation especially the R-type and I-type because we implemented only four operations in the lab.

3. Branch Control Unit:

The Branch Control Unit uses control signals such as the "Branch" signal, "funct3," and the ALU flags like "sf," "zf," "vf," and "cf" to determine if the branch is taken or not. The module effectively determines whether a branch should be taken based on the specified conditions and flag statuses for all the branching instructions.

4. ALU:

The "ALU" is a 32-bit Arithmetic Logic Unit that performs logical and arithmetic operations on two 32-bit inputs. It supports logical operations such as logical AND, OR, XOR. Similarly, it supports arithmetic and shift operations, such as ADD, SLL, and SRL. The ALU is modified to perform the rest of logical and arithmetic operations based on the 4-bit select signal "sel.". Sign, carry, overflow, and zero flags have been added to support controlling the upcoming stages like the branching control.

5. Memory:

In this milestone, we used a single memory for both data and instructions. The memory is single ported and byte addressable. We modified the data memory to be byte addressable to allow loading and storing one byte or half words. It is also modified to allow zero extension for loading byte and half word. To handle the structural hazards, we stall the fetching of instruction during the memory stage of any load/store instructions. If the EX/Mem. ReadData/WriteData is asserted, the memory take the address from the ALU_output. Otherwise, it take the PC output to fetch the following instruction.

6. Forwarding Unit:

The Forwarding Unit resolves data hazards by forwarding data from the output of the ALU or memory stage to the input of the ALU of the following instruction. This ensures that data produced by previous instructions can be used immediately by subsequent instructions, mitigating the need to stall the pipeline. The Forwarding Unit monitors data dependencies between instructions and efficiently routes data to the appropriate stages based on the hazard detection signals.

7. Hazard Detection Unit:

The Hazard Detection Unit is responsible for identifying potential hazards within the pipeline and signaling the need for corrective actions. It monitors load-use dependencies, and the structural hazards that requires the program to stall for one cycle. The Hazard Detection Unit works in tandem with the forwarding unit to identify situations where instruction execution might be affected due to data dependencies or control flow changes. When a hazard is detected,

the Hazard Detection Unit generates control signals to stall the pipeline by setting the control signals of the instruction to zero and stop incrementing the PC value.

8. Register File:

The Register File contains 32-bit registers for reading and storing the data during the instructions execution to increase the program speed. It takes inputs such as read addresses, write address, and write data. Its outputs include data read from specified registers by the input addresses.

9. Immediate Generator:

The Immediate Generator takes a 32-bit instruction and produces a 32-bit immediate value based on the instruction opcode (OPCODE).

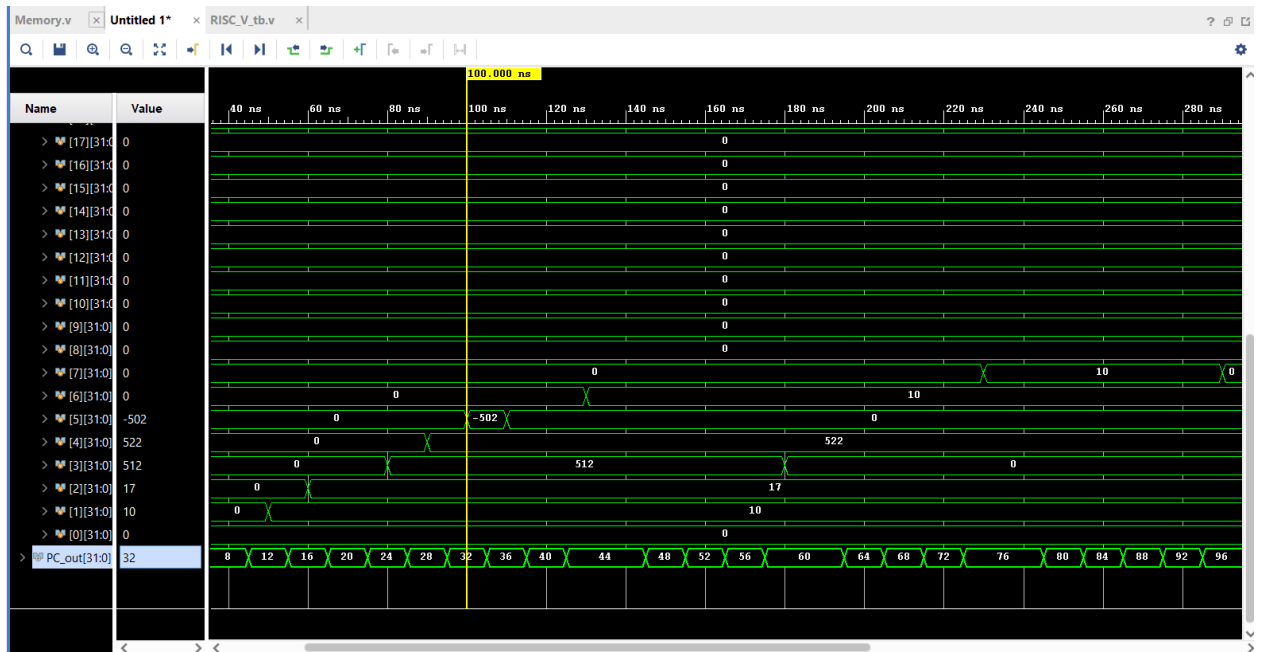
Test cases:

we have used three test cases to validate the functionality of the different instruction. These test cases are commented in the DataMem.v file attached with this report if you need to verify the provided output.

- Test 1(R_type,beq,bne,bge,setlessthan)

addi x1,x0,10	x1=10
addi x2,x0,17	x2=17
addi x3,x0,512	x3=512
add x4,x1,x3	x4=522
sub x5,x1,x3	x5=-502
slt x5,x3,x1	x5=0
sltu x5,x3,x1	x5=0
addi x6,x0,10	x6=10
<u>bne</u> x1,x2,L1	branch
<u>bne</u> x1,x6,L1	No branch
add x0,x0,x0	x0=0
L1: add x3,x0,x0	x3=0
<u>beq</u> x1,x6,L2	branch
<u>beq</u> x1,x2,L1	No branch
add x0,x0,x0	
L2:xor x7,x1,x3	x7=10
bge x1,x0,L3	branch
bge x0,x1,L3	No branch
add x0,x0,x0	
L3:or x7,x1,x3	x7=10
and x7,x1,x3	0

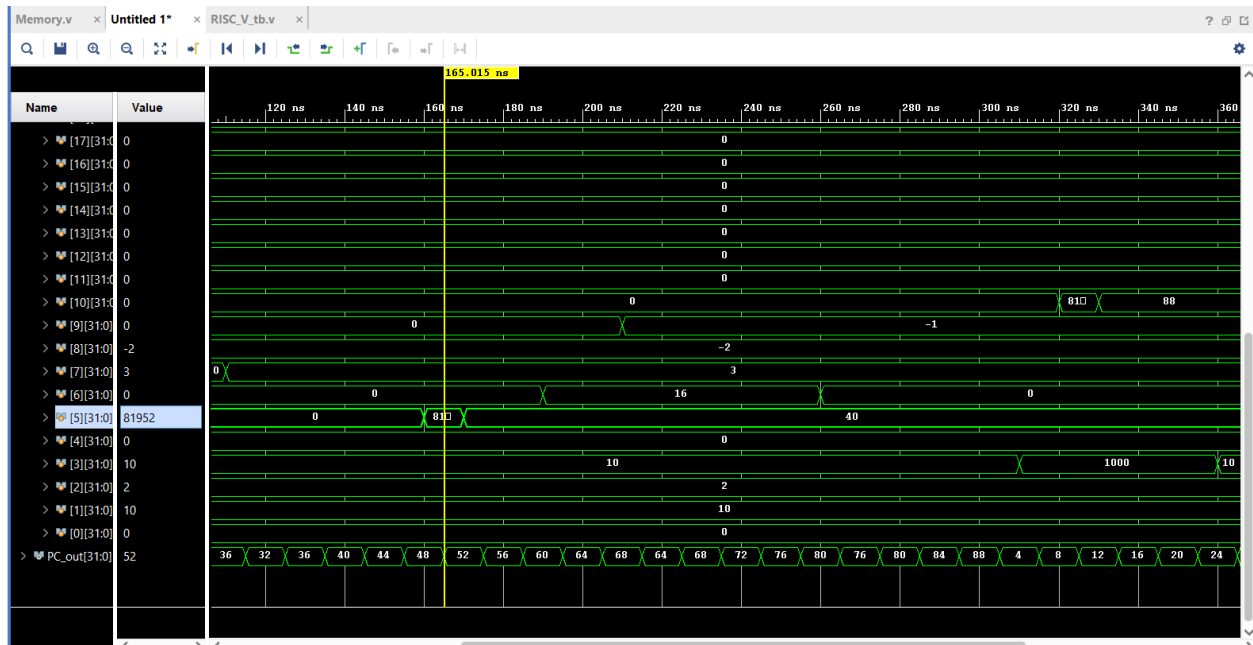
- Output




- Test 2(jump,auipc,lui,shifting,blt,bltu,and bgeu)

	addi x1 x0 10	x1=10
	addi x2 x0 2	x2=2
⋮ ⚙ +	addi x3 x0 10	x3=10
	addi x8 x0 -2	x8=-2
	blt x1 x2 16	No branch
	addi x7 x0 3	x7=3
	blt x2 x1 8	No branch
	addi x4 x0 1000	
	auipc x5 20	x5=81952
	jal x5 8	x5=40
	addi x4 x0 1000	
	sll x6 x2 x7	x6=16
	bltu x1 x2 16	No branch
	sra x9 x8 x2	x9=-1
	bltu x2 x1 8	branch
	addi x4 x0 1000	
	srl x6 x2 x7	x6=0
	bgeu x1 x2 8	branch
	addi x4 x0 1000	x4=0
	addi x3 x0 1000	x3=1000
	lui x10 200	819200
	jalr x10 x0 4	Return to 4

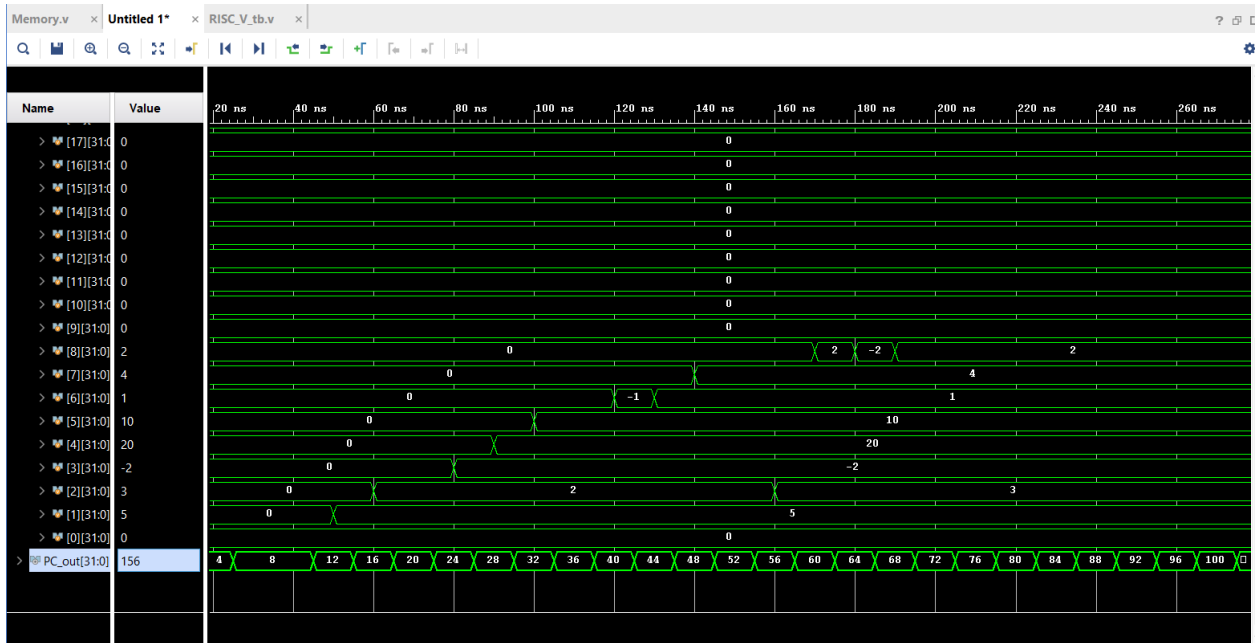
- Output



- Test 3(Multiplication , division,Remainder)

addi x1 x0 5	x1=5
addi x2 x0 2	x2=2
addi x3 x0 -2	x3=-2
addi x4 x0 20	x4=20
mul x5 x2 x1	x5=10
mulh x6 x2 x1	x6=0
mulh x6 x1 x3	x6=-1
mulhu x6 x2 x3	x6=1
div x7 x4 x1	x7=4
divu x7 x4 x1	x7=4
addi x2 x0 3	x2=3
rem x8 x4 x2	x8=2
rem x8 x3 x2	x8=-2
remu x8 x3 x2	x8=2
	

- **Output**



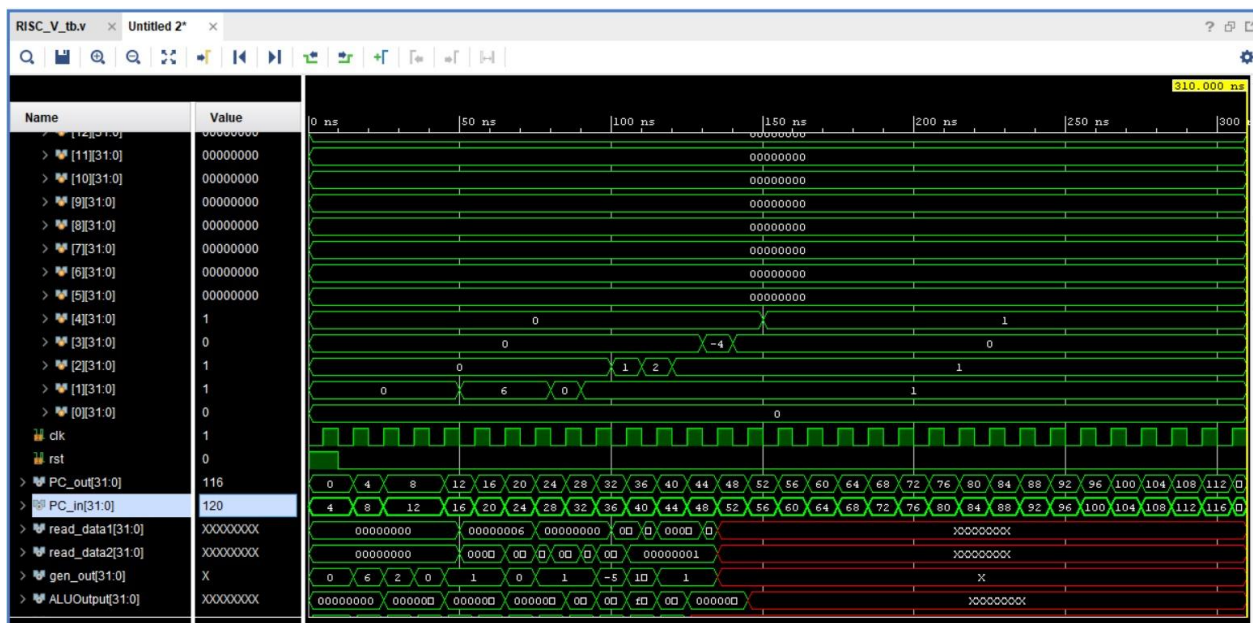
- Test 4(Itype):

```

addi x1, x0, 6
ori x1, x1, 2
andi x1, x1, 1
xori x1, x1, 1
addi x2, x1, 0
slli x2, x2, 1
srli x2, x2, 1
addi x3, x2, -5
srai x3, x2, 1
slti x4, x3, 1
sltiu x4, x3, 1

```

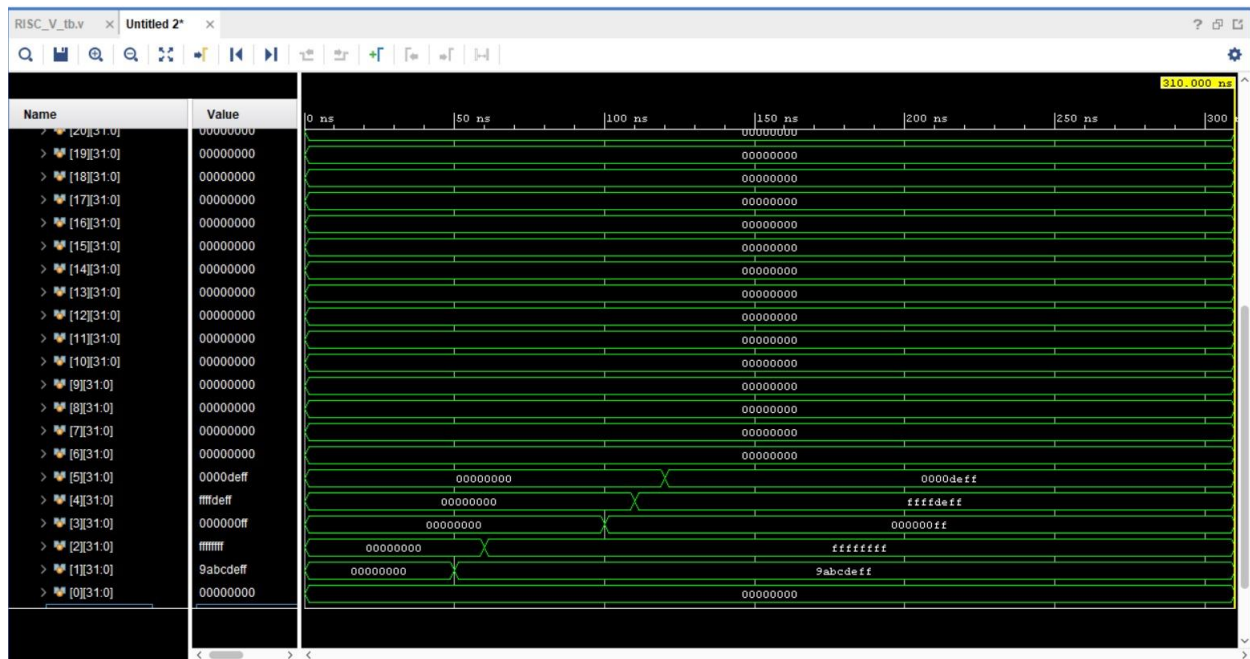
- Output



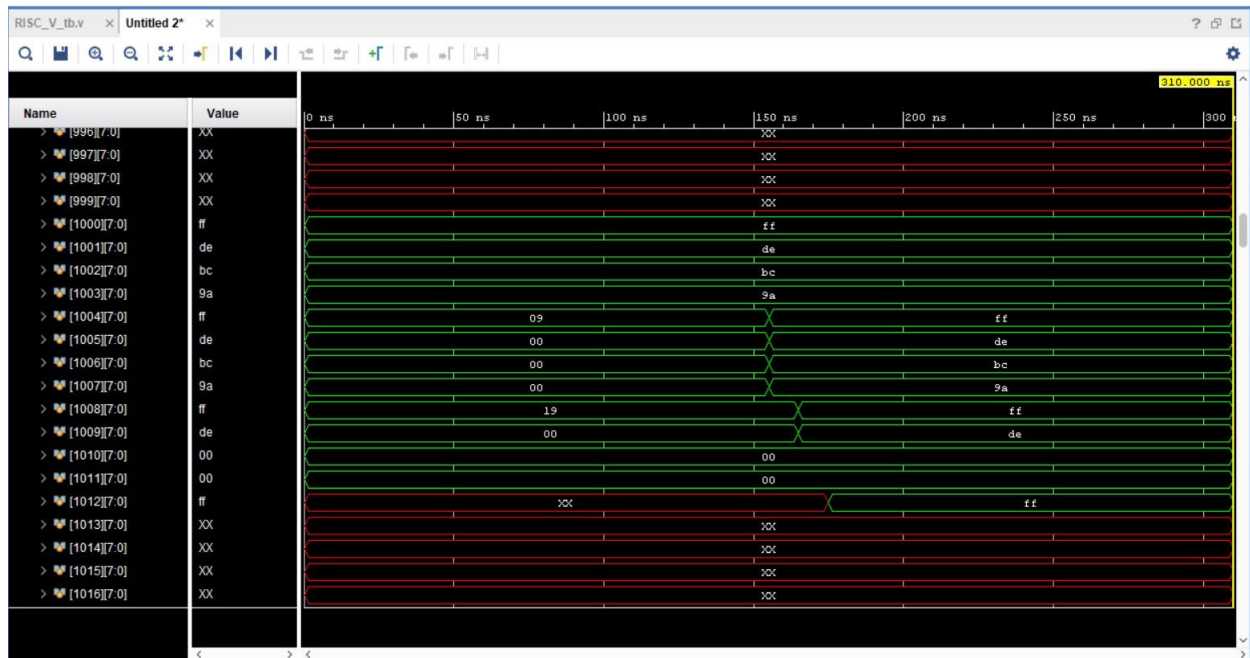
- Test 5(Sw and LW)

```
lw x1, 1000(x0) #x1= 9abcdeff
lb x2, 1000(x0) #x2= ffffffff
lbu x3, 1000(x0) #x3= 000000ff
lh x4, 1000(x0) #x4= ffffdeff
lhu x5, 1000(x0) #x5 = 0000deff
sw x1, 1004(x0) #x1=9abcdeff
sh x1, 1008(x0) #x2=0000deff
sb x1, 1012(x0) #x3=000000ff
```

- Output (LW)



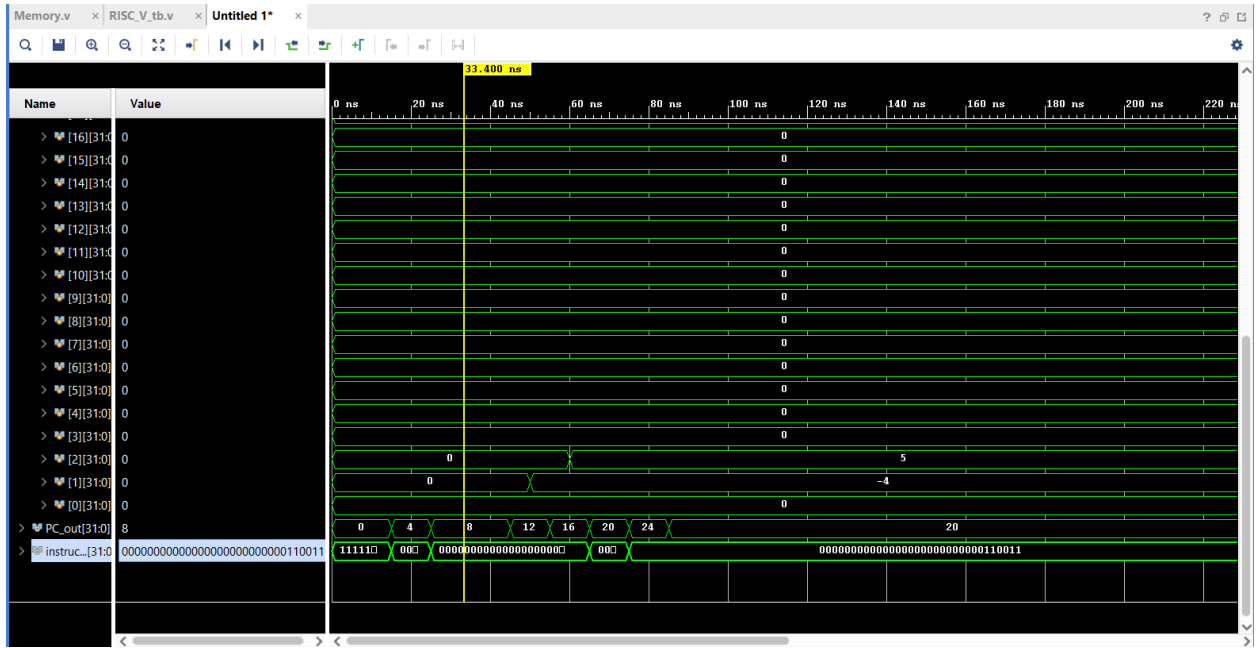
- Output (SW)



- Test6(EBREAK, ECALL,andEFENCE)

```
#addi x1 x0 -4
#addi x2 x0 5
#ecall
#ecall
#fence
#ebreak
#addi x1 x0 -4
#add x0 x0 x0
#add x0 x0 x0
```


- **Output**

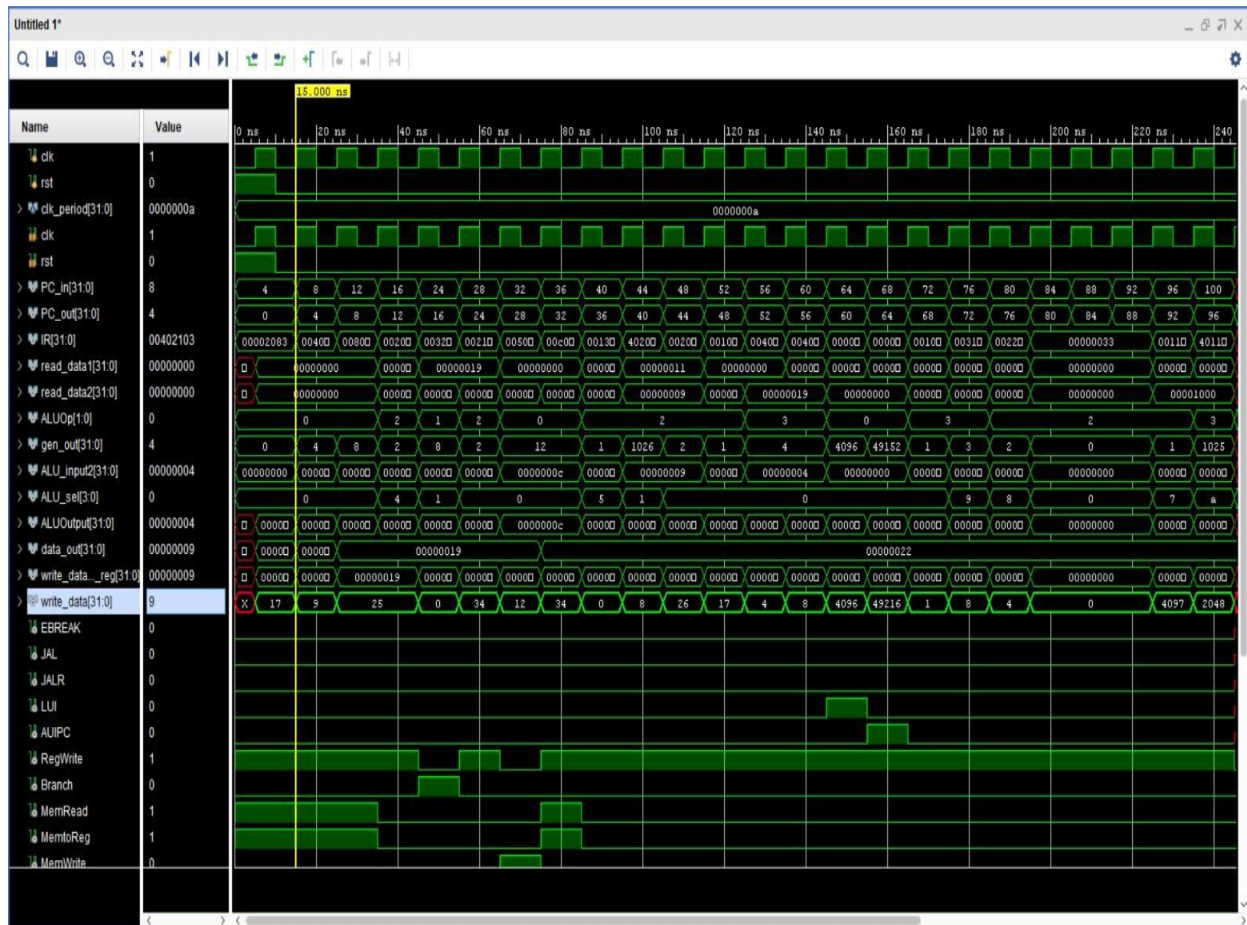


Extra test cases:

- **Test_1:**

```
• initial begin
•     mem[0]=32'b000000000000_00000_010_00001_0000011 ; //lw x1, 0(x0)
•     mem[1]=32'b000000000100_00000_010_00010_0000011 ; //lw x2, 4(x0)
•     mem[2]=32'b000000001000_00000_010_00011_0000011 ; //lw x3, 8(x0)
•     mem[3]=32'b0000000_00010_00001_110_00100_0110011 ; //or x4, x1, x2
•     mem[4]=32'b0_000000_00011_00100_000_0100_0_1100011; //beq x4, x3,4
•     mem[5]=32'b0000000_00010_00001_000_00011_0110011 ; //add x3, x1,x2
•     mem[6]=32'b0000000_00010_00011_000_00101_0110011 ; //add x5, x3,x2
•     mem[7]=32'b0000000_00101_00000_010_01100_0100011; //sw x5, 12(x0)
•     mem[8]=32'b000000001100_00000_010_00110_0000011 ; //lw x6, 12(x0)
•     mem[9]=32'b0000000_00001_00110_111_00111_0110011 ; //and x7, x6,x1
•     mem[10]=32'b0100000_00010_00001_000_01000_0110011 ; //sub x8,x1,x2
•     mem[11]=32'b0000000_00010_00001_000_00000_0110011 ; //add x0,x1,x2
•     mem[12]=32'b0000000_00001_00000_000_01001_0110011 ;//add x9, x0,x1
•     mem[13]=32'b00000000010000000000000010010011; //addi x1, x0, 4
•     mem[14]=32'b000000000100_00001_000_00001_0010011; //addi x1, x1, 4
•     mem[15] = 32'b0000000000000000000000001000010110111; //lui x1, 1
•     mem[16] = 32'b0000000000000000000011000000100010111; //auipc x2, 12
•     mem[17] = 32'b00000000000010000000000000100010011; //addi x2, x0, 1
• //x2 = 1
•     mem[18] = 32'b0000000000011_00010_001_00100_0010011; //slli x4,x2,3
• //x4 = 8
•     mem[19] = 32'b00000000_00010_00100_101_00100_0110011; //srl x4,x4,x2
• //x2 = 4
•     mem[20] = 32'b00000000000000000000000000001110011; //ECALL
•     mem[21] = 32'b0000111111111100000000000000001111; //FENCE
•     mem[22] = 32'b00000000000010000000000000001110011; //EBREAK
•     mem[23] = 32'b0000000000001000101000000110110011; //xor x3, x2, x1
•     mem[24] = 32'b010000000000100011101000110010011; //srai x3, x3, 1
•
• end
```

- **The output:**



- **Test_2:**

```

• //test_case_2 --shifting
•   initial begin
•       mem[0] = 32'b00000000000100000000000010010011; //addi x1, x0, 1
•       //x2 = 1
•       mem[1] = 32'b00000000000100000000000010010011; //addi x2, x0, 1
•       //x2 = 1
•       mem[2] = 32'b00000000000100010001000100010011; //slli x2, x2, 1
•       //x2 = 2
•       mem[3] = 32'b00000000000100010101000100010011; //srli x2, x2, 1
•       //x2 = 1

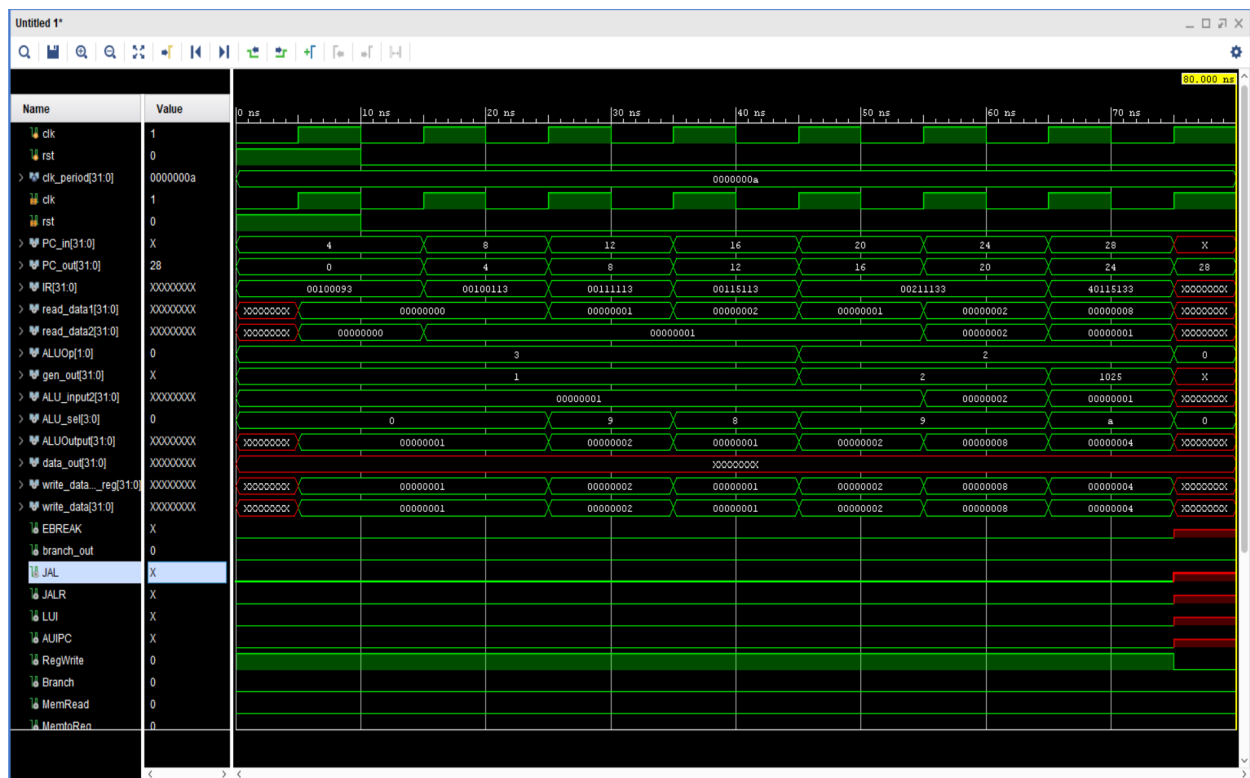
```

```

•      mem[4] = 32'b00000000001000010001000100110011; //sll x2, x2, x2
•      //x2 = 2
•      mem[5] = 32'b00000000001000010001000100110011; //sll x2, x2, x2
•      //x2 = 8
•      mem[6] = 32'b01000000000100010101000100110011; //sra x2, x1, x2
•      //x2 = 4
•      end

```

• The output:



• Test_3:

```

•      //test_case_3 --loading and branching
•      initial begin
•          mem[0]=32'b000000001100_00000_010_00001_0000011 ; //lw x1, 12(x0)
•          //9abcdeff
•          mem[1]=32'b000000001100_00000_000_00001_0000011 ; //lb x1, 13(x0)
•          //ffffffff
•          mem[2]=32'b000000001100_00000_100_00001_0000011 ; //lbu x1, 13(x0)
•          //000000ff

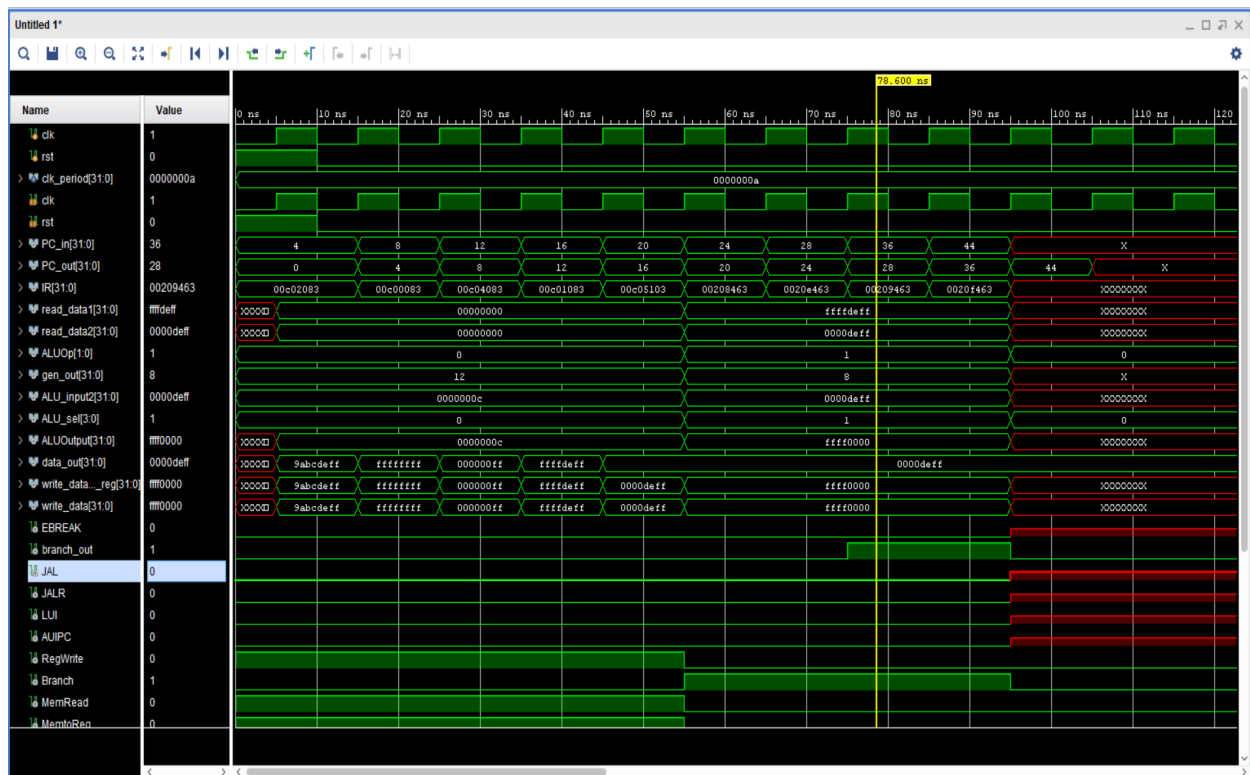
```

```

•      mem[3]=32'b000000001100_00000_001_00001_0000011 ; //lh x1, 12(x0)
//ffffdeff
•      mem[4]=32'b000000001100_00000_101_00010_0000011 ; //lhu x2, 12(x0)
//0000deff
•      mem[5]=32'b0_000000_00010_00001_000_0100_0_1100011; //beq x1,x2,4
//false
•      mem[6]=32'b0_000000_00010_00001_110_0100_0_1100011; //bltu x1,x2,4
//false
•      mem[7]=32'b0_000000_00010_00001_001_0100_0_1100011; //bne x1,x2,4
//true
•      mem[8]=32'b00000000_00010_00001_110_00100_0110011 ;//or x4,x1,x2
//skipped
•      mem[9]=32'b0_000000_00010_00001_111_0100_0_1100011; //bgeu x1,x2,4
•      #true
•
•      end

```

- The output:



Faced difficulties:

We did not face many difficulties in this project. The main one was recognizing the optimum and simplest way to implement the additional branch and load instruction. However, Dr. Cherif Salama provided a supporting lecture for the project that addressed this issue that helped us implement the branch control block.

Bonus Features:

1. Support for integer multiplication and division effectively as a support for the full RV32IM instruction set
2. Providing another solution for handling the structural hazard of using a single memory.
This was implemented by replacing the issuing of one instruction every 2 clock cycles with issuing one instruction every 2 clock cycles only when there is a load or store instruction. Otherwise, an instruction is issued every 1 clock cycle
3. A random instruction generator for R-type instructions

Conclusion:

In conclusion, for Milestone #3 of the femtoRV32 project, we successfully implemented the RV32I base integer instruction set, covering all 40 user-level instructions within the unprivileged ISA. The implemented design features a pipelined architecture, divided into five stages, and addresses potential hazards through a Hazard Detection Unit and a Forwarding Unit. Throughout the implementation, we have adhered to the principles of the RISC-V instruction set architecture, emphasizing simplicity, modularity, and effective hazard handling. Rigorous testing has been conducted, covering all 40 instructions and addressing

potential hazard scenarios. Additionally, the processor supports integer multiplication and division instructions, showcasing its versatility.