

CSCE 3301 - Computer Architecture

Spring 2025

Project 1: femtoRV32

Final Project Report

Authors:

Ebram Thabet	900214496
--------------	-----------

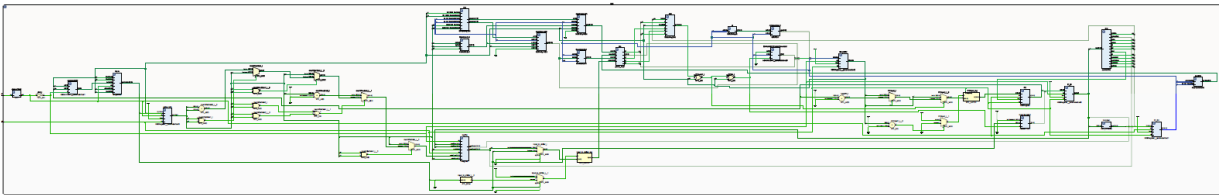
Eman Abd Elhady	900226489
-----------------	-----------

Introduction

This report details the design and testing of femtoRV32 with random instruction generator and support for compressed instructions, The project focuses on supporting the RV32I base instruction set, addressing challenges like pipelining, memory hazards, and custom instruction handling. Key achievements include:

- Implementation of 42 user-level instructions.
- Efficient 3-stage pipelining with hazard mitigation.
- Custom handling of system calls (ECALL, EBREAK).
- Support for compressed instructions.

System Design and Architecture



Datapath Overview

The processor's datapath integrates:

1. Unified Memory: Single-port, byte-addressable memory for instructions/data (structural hazards managed via fetching every 2 clock cycles).
2. Register File: 32 general-purpose registers.
3. ALU: Supports arithmetic (ADD, SUB), logical (AND, OR), and shift operations.
4. Control Unit: Decodes instructions and generates control signals.

Pipelining Strategy

A 3-stage pipeline (6 clock cycles total) optimizes throughput:

- Stage 0: Fetch (C0) + Register Read (C1).
- Stage 1: ALU Execution (C0) + Memory Access (C1).
- Stage 2: Writeback (C0).

Advantages:

Simplified hazard control (forwarding/stalling for RAW/WAW hazards).

Instruction Set Implementation

Core Instructions

1. Arithmetic/Logical:

- ADD, SUB, XOR, SLL (fully implemented in ALU).

2. Branch/Jump:

- BEQ, BNE with static branch prediction.

3. Memory Operations:

- LW/SW with load-use hazard resolution via forwarding.

Custom Handling

- ECALL, EBREAK, FENCE: Halts execution (PC freeze).

Schematic Diagram

Check [*schematic.pdf*](#) for full schematic.

Testing & Validation

Below are some of the tests we ran:

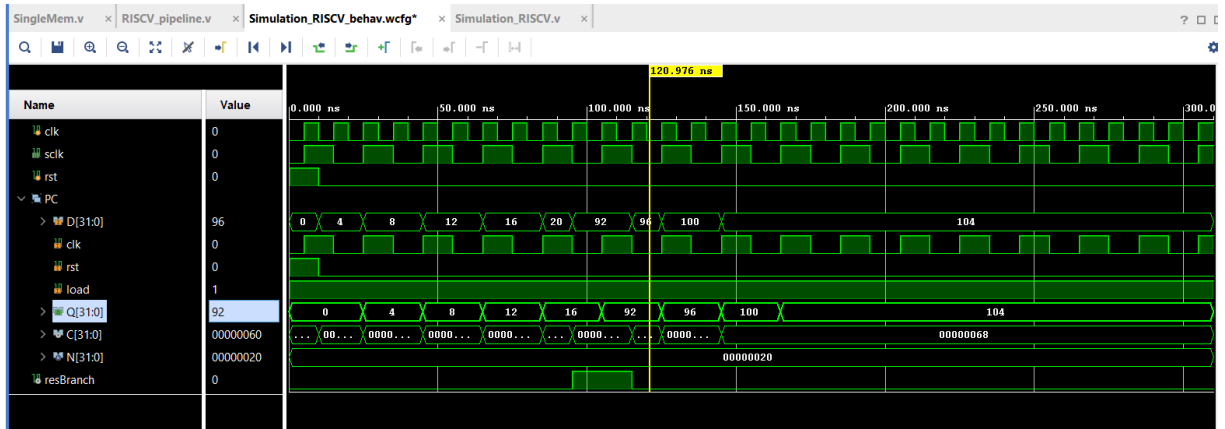
- 1- Branch

```

00700093 #addi x1, x0, 7
00400113 #addi x2, x0, 4
02208663 #beq x1, x2, 44
02209463 #bne x1, x2, 40
fc20c8e3 #blt x1, x2, -48
fc20c8e3 #blt x1, x2, -48
fc20c8e3 #blt x1, x2, -48
fc20c8e3 #blt x1, x2, -48
fc20c8e3 #blt x1, x2, -48
fc20c8e3 #blt x1, x2, -48
fc20c8e3 #blt x1, x2, -48

```

Output:



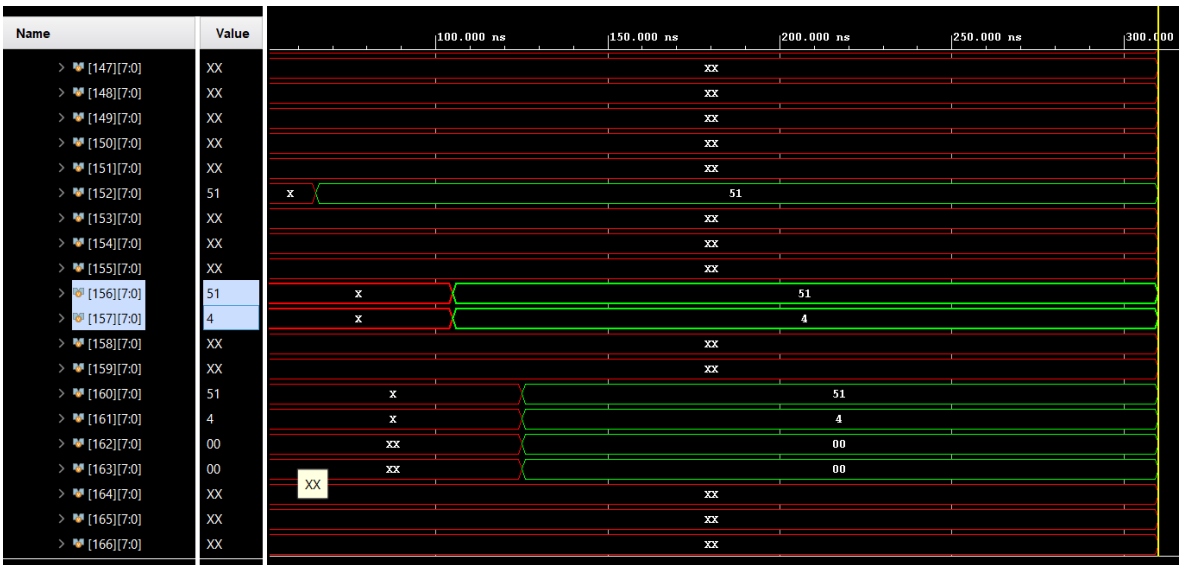
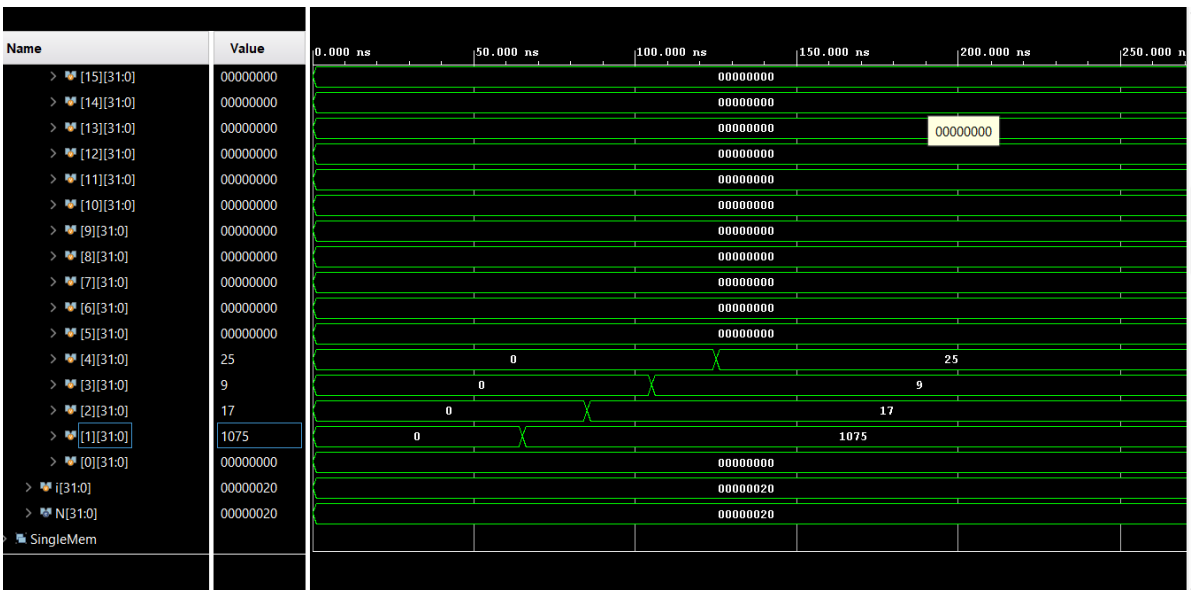
2- Load and store

```

43300093 #addi x1, x0, 1075
00100c23 #sb x1, 24(x0)
01800103 #lb x2, 24(x0)
00101e23 #sh x1, 28(x0)
01c01183 #lh x3, 28(x0)
02102023 #sw x1, 32(x0)
02002203 #lw x4, 32(x0)

```

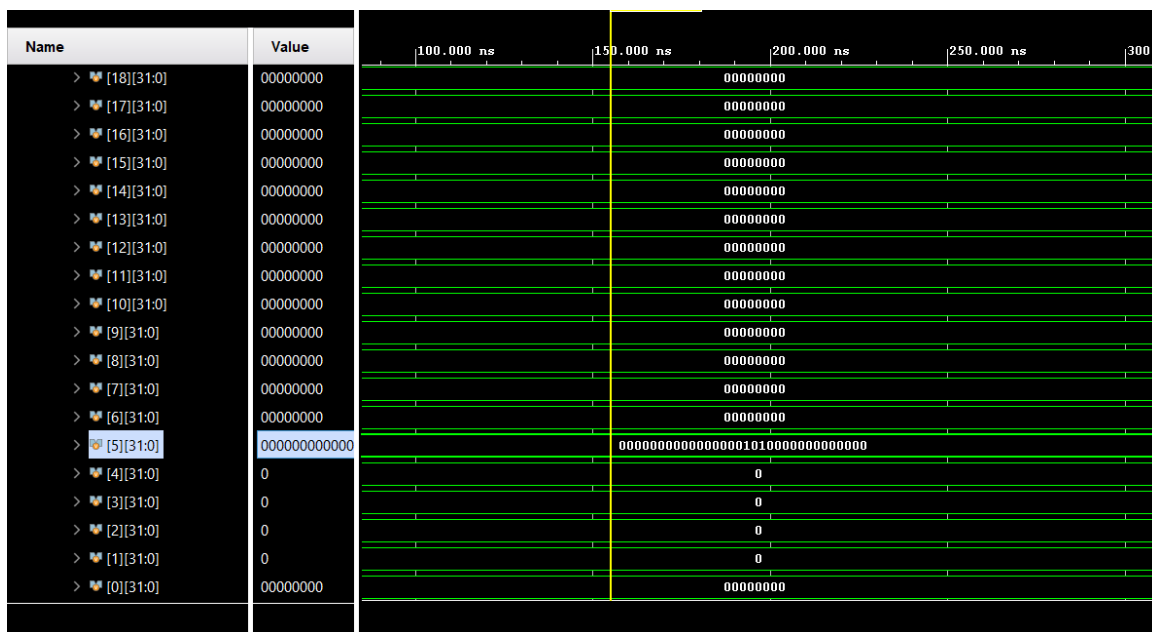
Output:



3- Lui

```
0000A2B7 #lui x5, 10
```

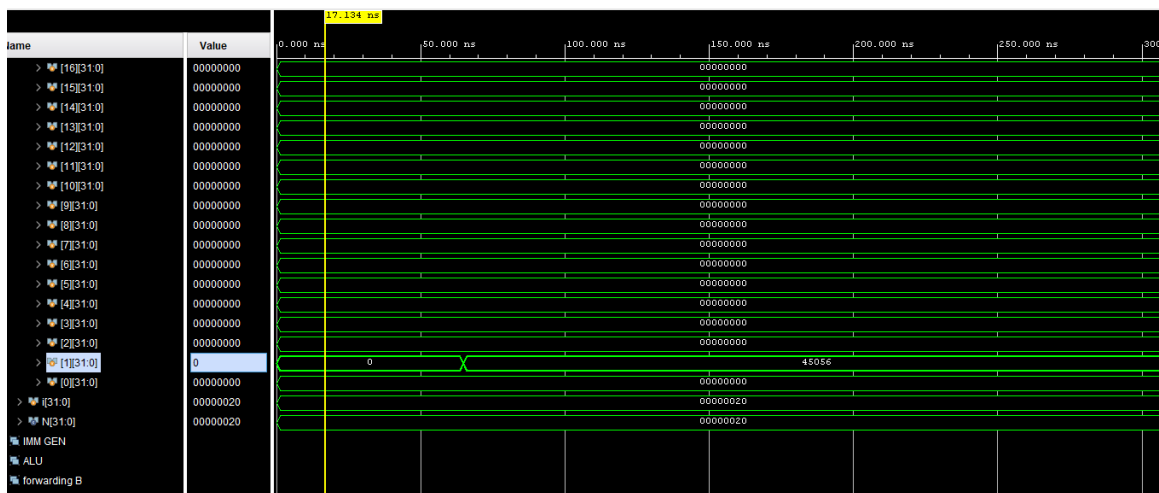
Output:



4- Auipc

```
0000b097 #auipc x1, 11
```

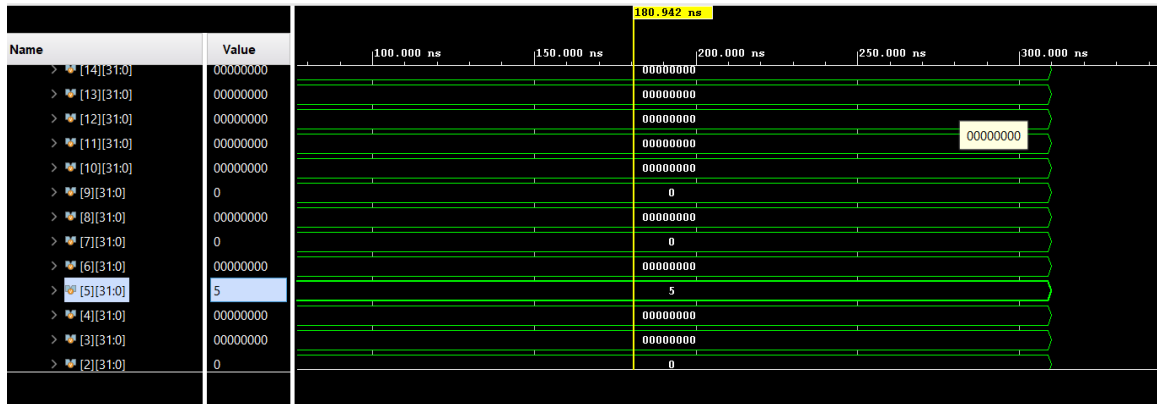
Output:



5- Ecall

```
00500293 #addi x5, x0, 5
00000073 #ecall
00500113 #addi x2, x0, 5
```

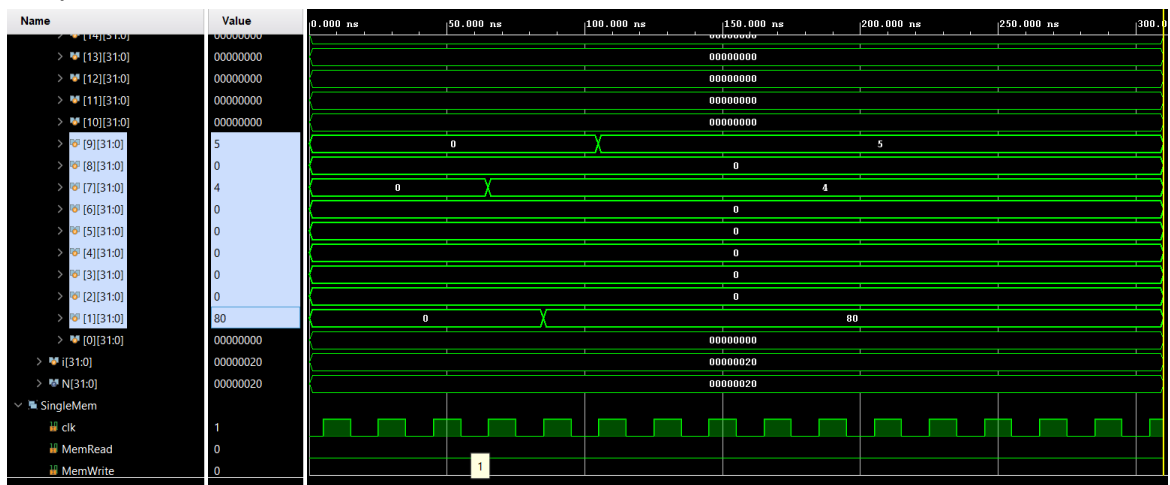
Output:



6- Srl

```
00400393 #addi x7, x0, 4
05000093 #addi x1, x0, 80
0070d4b3 #srl x9, x1, x7
```

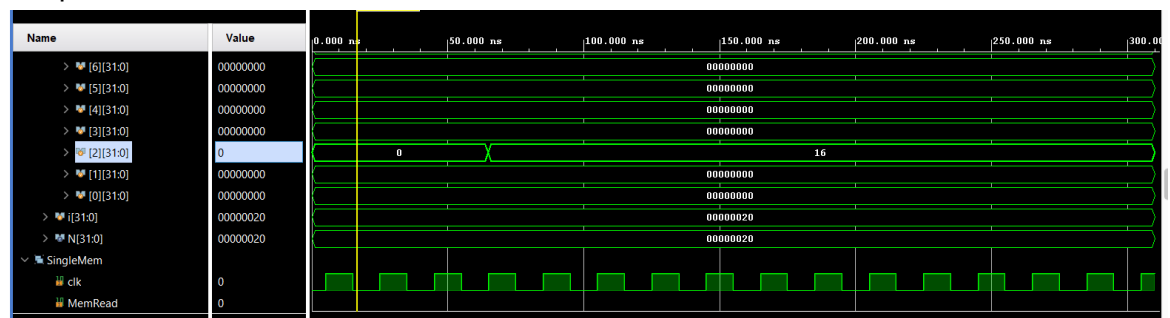
Output:



7- C.addi

```
0141 #c.addi x2, 16
```

Output:



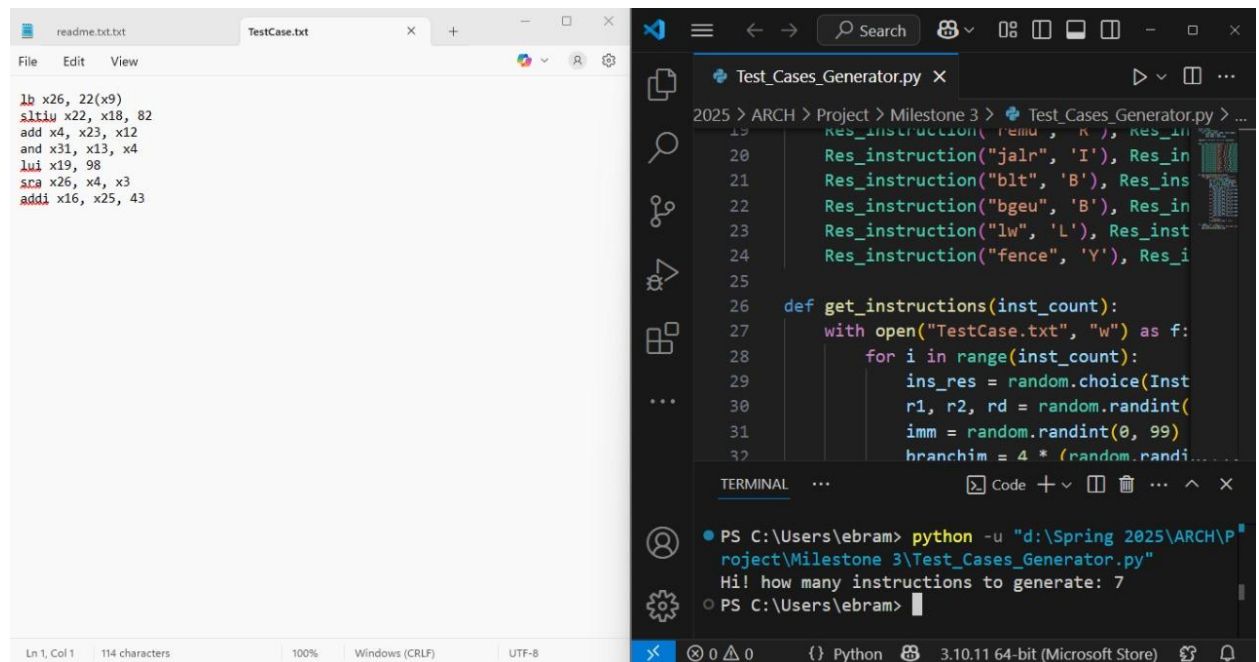
Bonus Features

1) Random Test Generator

This feature is implemented in python to create a list of random valid instructions to test the processor. The only input to the program is the number of instructions we want to generate. And the output is a .txt file containing the generated instructions.

Example input: 7

Output generated: File with 7 RISC-V instruction.



The screenshot shows a code editor with two windows. The left window, titled 'readme.txt.txt', displays the following RISC-V instructions:

```
lb x26, 22(x9)
sltiu x22, x18, 82
add x4, x23, x12
and x31, x13, x4
lui x19, 98
sra x26, x4, x3
addi x16, x25, 43
```

The right window, titled 'Test_Cases_Generator.py', shows the Python script used to generate these instructions. The script defines a function `get_instructions` that takes an `inst_count` and generates random RISC-V instructions. The terminal output shows the command to run the script and the prompt for the number of instructions to generate.

```
2025 > ARCH > Project > Milestone 3 > Test_Cases_Generator.py > ...
19 res_instruction("remu", 'R'), Res_in
20 Res_instruction("jalr", 'I'), Res_in
21 Res_instruction("blt", 'B'), Res_ins
22 Res_instruction("bgeu", 'B'), Res_in
23 Res_instruction("lw", 'L'), Res_inst
24 Res_instruction("fence", 'Y'), Res_i
25
26 def get_instructions(inst_count):
27     with open("TestCase.txt", "w") as f:
28         for i in range(inst_count):
29             ins_res = random.choice(Inst
30             r1, r2, rd = random.randint(
31             imm = random.randint(0, 99)
32             branchim = 4 * (random.randi...
```

Terminal output:

```
PS C:\Users\ebam> python -u "d:\Spring 2025\ARCH\Project\Milestone 3\Test_Cases_Generator.py"
Hi! how many instructions to generate: 7
PS C:\Users\ebam>
```

2) Supporting compressed instructions

Our pipeline implementation supports compressed instructions. The instruction gets fetched and then checked to see if it compressed or not by the `compressionUnit`. If it is compressed, the instruction is then decompressed and outputted from the unit.

Example:

```
|0141 #c.addi x2, 16
```

Output:

