



CSCE 3301 – Computer Architecture

Fall 2024

Project 1: femtoRV32

RISC-V processor Implementation and Testing

Milestone 3 Report

Authors:

Saif Abd Elfattah

Adham Ali

14/11/2024

Introduction

This report presents the implementation and testing of the femtoRV32, a RISC-V processor designed for FPGA implementation using the Nexys A7 trainer kit. The primary objective is to support the RV32I base integer instruction set, including handling pipelining and memory hazards, while adhering to the specification outlined by RISC-V. The processor implements 42 user-level instructions, and we focus on achieving an effective performance despite limitations like single-port memory. We also explore the design challenges faced, particularly with respect to pipelining, structural hazards, and custom behavior for certain instructions like ECALL and EBREAK.

System Design and Architecture

DataPath

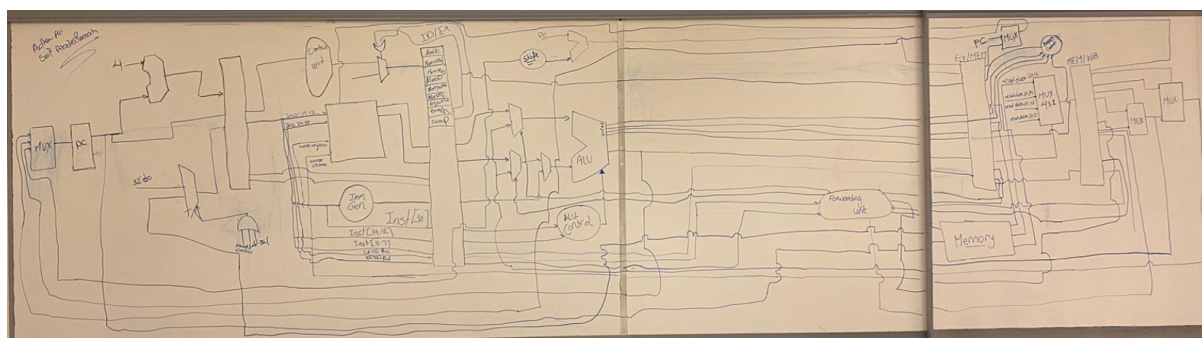


Figure 1: DataPath

The data path represents the flow of data within the processor from instruction fetch to execution and writing back results. A diagram of the datapath should be included here to show the various components involved:

- **Instruction Memory:** A single memory unit is used for both data and instructions, making it a shared resource and creating structural hazards. This memory is byte-addressable and ported for reading and writing.

- **Registers and ALU:** The register file contains 32 general-purpose registers. The ALU performs arithmetic and logical operations, such as ADD, SUB, AND, OR, and others.
- **Control Unit:** The control unit decodes the opcode from the fetched instruction and generates the necessary control signals for various components like the ALU, memory, and registers.
- **Hazard Management:** Special attention is given to hazard detection and resolution. Structural hazards arise due to the use of single-port memory for both data and instructions, while data and control hazards are handled by forwarding and stalling mechanisms.

Pipelining

A notable strength of this RISC-V processor implementation lies in its efficient pipeline design, which divides instruction execution into three distinct stages, with each stage allocated two clock cycles (C0 and C1). This well-structured division not only ensures balanced utilization of the clock cycles but also simplifies hazard detection and control, reducing the risk of pipeline stalls.

Key advantage: By leveraging this 6-cycle split, the CPU maximizes resource usage while maintaining a clear separation between stages:

- **Stage 0:** Combines Instruction Fetch (C0) and Register Read (C1), ensuring that the fetched instruction is immediately decoded and its operands are available for processing.
- **Stage 1:** Pairs ALU execution (C0) with Memory access (C1), efficiently handling arithmetic/logic operations and data memory interactions within the same cycle, minimizing latency.
- **Stage 2:** Handles the Register Write Back (C0) independently, avoiding any conflicts, as no other operations utilize C1 during this stage.

Instruction Set Implementation

The RV32I base integer instruction set comprises 42 user-level instructions that our processor must implement. These instructions include arithmetic, logic, branch, and load/store operations. The following points highlight the implementation approach:

Arithmetic and Logical Operations

The arithmetic and logical instructions include basic operations like `ADD`, `SUB`, `XOR`, `AND`, `OR`, and shifts. Each instruction is executed by the ALU, and the result is written back to a destination register. Here's a detailed breakdown:

- `ADD`: Adds two register values.
- `SUB`: Subtracts two register values.
- `AND`, `OR`, `XOR`: Performs bitwise operations between registers.
- `SLL`, `SRL`, `SRA`: Performs logical and arithmetic shifts.

These instructions are straightforward in their implementation, relying on the ALU for computation.

Branch Instructions

Branch instructions include `BEQ` (branch if equal), `BNE` (branch if not equal), `BLT` (branch if less than), and `BGE` (branch if greater than or equal). These instructions modify the program counter depending on the result of a comparison between two registers. We implemented branch prediction mechanisms to handle potential control hazards.

Load and Store Instructions

These instructions interact with the memory system. Examples include:

- **LW (Load Word)**: Loads 32-bit data from memory into a register.
- **SW (Store Word)**: Stores the contents of a register into memory.

Memory operations like these involve potential load-use hazards. To handle them efficiently, the processor uses data forwarding and stalling.

Custom Instructions

The following instructions deviate from the specification:

- **ECALL**: Interpreted as a halting instruction to stop the execution of the program. This is done by preventing the program counter from updating further.
- **EBREAK, PAUSE, FENCE, FENCE.TSO**: These are treated as no-op instructions that effectively do nothing in the context of this project.

Schematic Diagram

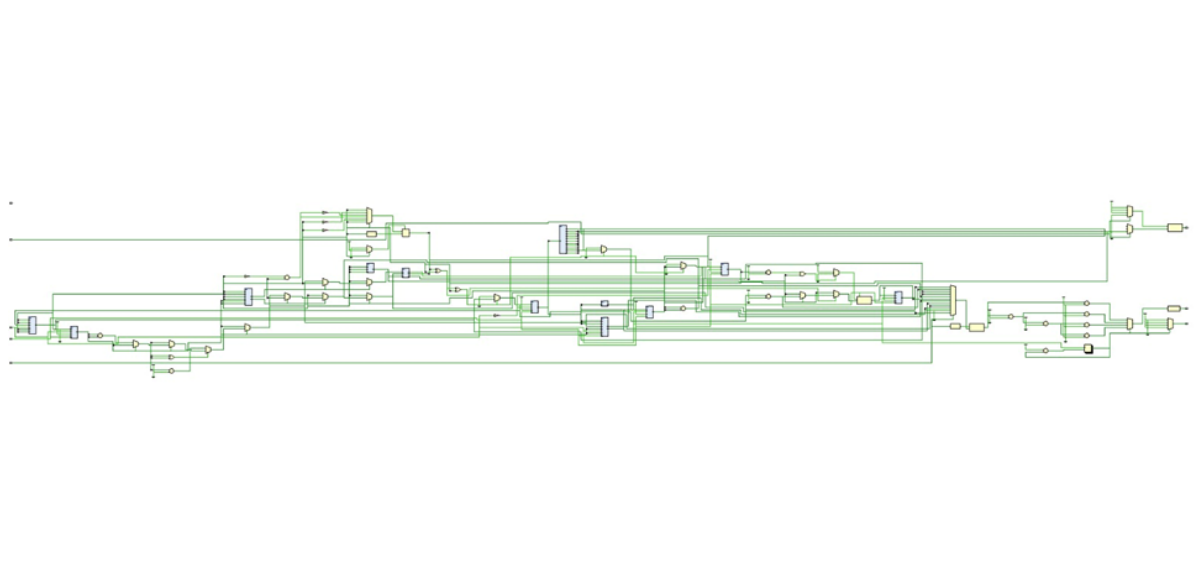


Figure 2: Schematic Diagram

Testing

Test 1

Instructions in assembly:

```
1 addi x1, x0, 5
2 addi x2, x0, 5
3 addi x4, x0, 40
4 addi x3, x0, 0
5 addi x5, x0, 10
6 addi x6, x0, 20
7 beq x1, x2, 8
8 addi x3, x0, -1
9 jalr x0, x4, 0
10 addi x3, x0, 1
11 jalr x0, x4, 0
```

Instructions in hex:

```
1 00500093
2 00500113
3 02800213
4 00000193
5 00a00293
6 01400313
7 00208463
8 fff00193
9 00020067
10 00100193
11 00020067
```

Tracing:

- `addi x1, x0, 5`: Sets `x1 = 5`.

- `addi x2, x0, 5`: Sets `x2 = 5`.
- `addi x4, x0, 40`: Sets `x4 = 40`.
- `addi x3, x0, 0`: Sets `x3 = 0`.
- `addi x5, x0, 10`: Sets `x5 = 10`.
- `addi x6, x0, 20`: Sets `x6 = 20`.
- `beq x1, x2, 8`: Since `x1 (5)` equals `x2 (5)`, the program branches by 8 bytes (skipping the next instruction).
- `addi x3, x0, -1`: This is skipped due to the branch.
- `jalr x0, x4, 0`: Jumps to the address stored in `x4 (40)`, and execution resumes there.
- `addi x3, x0, 1`: Sets `x3 = 1`.
- `jalr x0, x4, 0`: Jumps again to the address in `x4 (40)`, looping back.

Simulation Output:

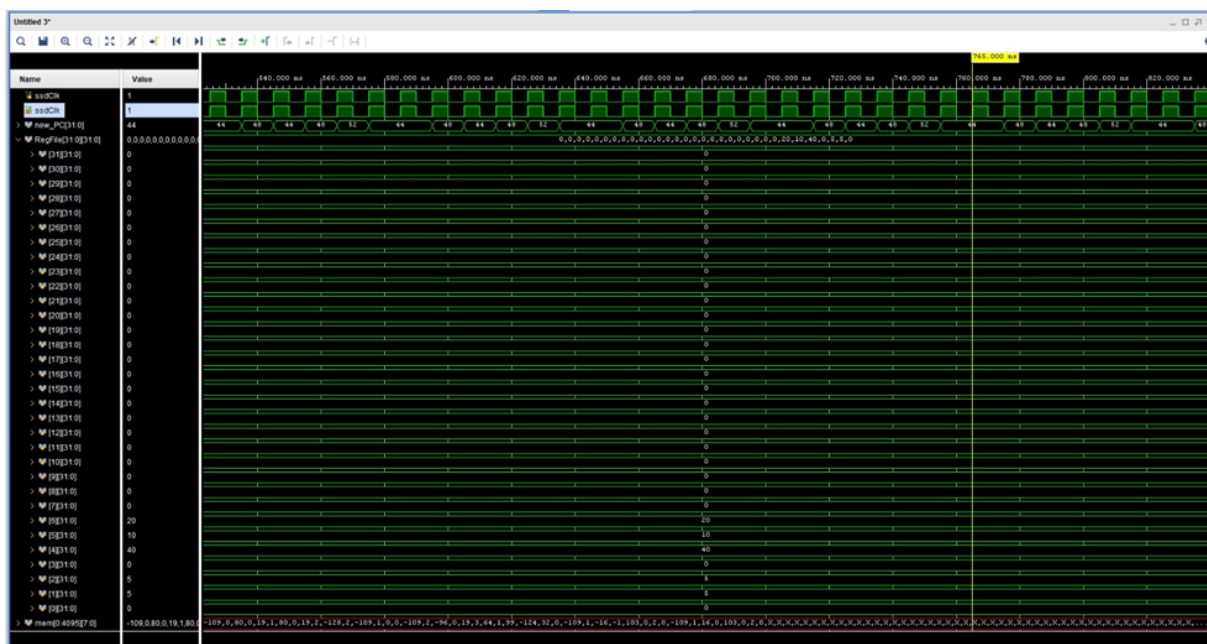


Figure 3: Test1 Simulation Output

Verified Output:

Input your RISC-V code here:

```

1  addi x1, x0, 5
2  addi x2, x0, 5
3  addi x4, x0, 40
4  addi x3, x0, 0
5  addi x5, x0, 10
6  addi x6, x0, 20
7  beq x1, x2, 8
8  addi x3, x0, -1
9  jalr x0, x4, 0
10 addi x3, x0, 1
11 jalr x0, x4, 0
12
13
14
15

```

Reset Stop CPU: 32 Hz

```

[line 11]: jalr x0, x4, 0
[line 11]: jalr x0, x4, 0
[line 11]: jalr x0, x4, 0
[line 11]: jalr x0, x4, 0
[line 11]: jalr x0, x4, 0

```

Features

- Reset to load the code, Step one instruction, or Run all instructions
- Set a breakpoint by clicking on the line number (only for Run)
- View registers on the right, memory on the bottom of this page

Supported Instructions

- Arithmetics: ADD, ADDI, SUB
- Logical: AND, ANDI, OR, ORI, XOR, XORI
- Sets: SLT, SLTI, SLTU, SLTIU
- Shifts: SRA, SRAI, SRL, SRLI, SLL, SLLI
- Memory: LW, SW, LB, SB
- PC: LUI, AUIPC
- Jumps: JAL, JALR
- Branches: BEQ, BNE, BLT, BGE, BLTU, BGEU

RISC-V Reference: [riscv-spec-v2.2.pdf](#)

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	00000000000000000000000000000000
0	x1 (ra)	5	0x00000005	00000000000000000000000000000101
0	x2 (sp)	5	0x00000005	00000000000000000000000000000101
0	x3 (gp)	0	0x00000000	00000000000000000000000000000000
0	x4 (tp)	40	0x00000028	000000000000000000000000000101000
0	x5 (t0)	10	0x0000000a	00000000000000000000000000001010
0	x6 (t1)	20	0x00000014	000000000000000000000000000010100
0	x7 (t2)	0	0x00000000	00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	00000000000000000000000000000000
0	x18 (s2)	0	0x00000000	00000000000000000000000000000000
0	x19 (s3)	0	0x00000000	00000000000000000000000000000000
0	x20 (s4)	0	0x00000000	00000000000000000000000000000000
0	x21 (s5)	0	0x00000000	00000000000000000000000000000000
0	x22 (s6)	0	0x00000000	00000000000000000000000000000000
0	x23 (s7)	0	0x00000000	00000000000000000000000000000000
0	x24 (s8)	0	0x00000000	00000000000000000000000000000000
0	x25 (s9)	0	0x00000000	00000000000000000000000000000000
0	x26 (s10)	0	0x00000000	00000000000000000000000000000000
0	x27 (s11)	0	0x00000000	00000000000000000000000000000000
0	x28 (t3)	0	0x00000000	00000000000000000000000000000000
0	x29 (t4)	0	0x00000000	00000000000000000000000000000000
0	x30 (t5)	0	0x00000000	00000000000000000000000000000000
0	x31 (t6)	0	0x00000000	00000000000000000000000000000000

Figure 4: Test 1 Verification

Test 2

Instructions in assembly:

```

1  addi x1, x0, 1
2  sltiu x2, x1, 2
3  ori x3, x0, -1
4  slti x4, x3, 0
5  xori x5, x1, 3
6  andi x6, x1, 3
7  slli x7, x1, 1
8  srli x8, x3, 1
9  srai x9, x3, 1
10 add x10, x1, x2

```


11	sub x11, x5, x1
12	sll x12, x1, x1
13	srl x13, x3, x1
14	sra x14, x3, x1
15	sltu x15, x1, x7
16	slt x16, x3, x0
17	or x17, x1, x0
18	xor x17, x17, x5
19	and x18, x17, x1
20	sw x13, 0(x0)
21	sh x13, 5(x0)
22	sb x1, 4(x0)
23	lhu x19, 4(x0)
24	lh x20, 4(x0)
25	lbu x21, 5(x0)
26	lb x22, 5(x0)
27	lw x23, 0(x0)
28	lui x24, 1
29	auipc x25, 1

Instructions in hex:

1	00100093
2	0020b113
3	fff06193
4	0001a213
5	0030c293
6	0030f313
7	00109393
8	0011d413
9	4011d493
10	00208533
11	401285b3
12	00109633
13	0011d6b3

14	4011d733
15	0070b7b3
16	0001a833
17	0000e8b3
18	0058c8b3
19	0018f933
20	00d02023
21	00d012a3
22	00100223
23	00405983
24	00401a03
25	00504a83
26	00500b03
27	00002b83
28	00001c37
29	00001c97

Tracing:

- `addi x1, x0, 1`: Adds 1 to x0 (0 by definition), setting x1 = 1.
- `sltiu x2, x1, 2`: Compares unsigned x1 (1) with 2; sets x2 = 1.
- `ori x3, x0, -1`: Bitwise OR between x0 (0) and -1; sets x3 = -1.
- `slti x4, x3, 0`: Compares signed x3 (-1) with 0; sets x4 = 1.
- `xori x5, x1, 3`: XOR between x1 (1) and 3; sets x5 = 2.
- `andi x6, x1, 3`: AND between x1 (1) and 3; sets x6 = 1.
- `slli x7, x1, 1`: Left-shifts x1 (1) by 1; sets x7 = 2.
- `srli x8, x3, 1`: Logical right shift on x3 (-1); sets x8 = 2147483647.
- `srai x9, x3, 1`: Arithmetic right shift on x3 (-1); sets x9 = -1.

Arithmetic and Logical Operations:

- `add x10, x1, x2`: Adds `x1` (1) and `x2` (1); sets `x10` = 2.
- `sub x11, x5, x1`: Subtracts `x1` (1) from `x5` (2); sets `x11` = 1.
- `sll x12, x1, x1`: Left shift on `x1` (1) by `x1`; sets `x12` = 2.
- `srl x13, x3, x1`: Logical right shift on `x3` (-1); sets `x13` = 2147483647.
- `sra x14, x3, x1`: Arithmetic right shift on `x3` (-1); sets `x14` = -1.
- `sltu x15, x1, x7`: Sets `x15` to 1 if `x1` (1) \lessdot `x7` (2) (unsigned); sets `x15` = 1.
- `slt x16, x3, x0`: Sets `x16` to 1 if `x3` (-1) \lessdot `x0` (0); sets `x16` = 1.

Bitwise Operations:

- `or x17, x1, x0`: Bitwise OR between `x1` (1) and `x0` (0); sets `x17` = 1.
- `xor x17, x17, x5`: XOR between `x17` (1) and `x5` (2); sets `x17` = 3.
- `and x18, x17, x1`: AND between `x17` (3) and `x1` (1); sets `x18` = 1.

Memory Operations:

- `sw x13, 0(x0)`: Stores `x13` (2147483647) to memory address 0.
- `sh x13, 5(x0)`: Stores lower half-word of `x13` at memory address 5.
- `sb x1, 4(x0)`: Stores byte value of `x1` (1) at memory address 4.

Load Operations:

- `lhu x19, 4(x0)`: Loads unsigned half-word from address 4; sets `x19` = 65281.
- `lh x20, 4(x0)`: Loads signed half-word from address 4; sets `x20` = -255.
- `lbu x21, 5(x0)`: Loads unsigned byte from address 5; sets `x21` = 255.
- `lb x22, 5(x0)`: Loads signed byte from address 5; sets `x22` = -1.

- `lw x23, 0(x0)`: Loads word from address 0; sets `x23 = 2147483647`.

Upper Immediate Operations:

- `lui x24, 1`: Loads upper 20 bits with 1; sets `x24 = 4096`.
- `auipc x25, 1`: Adds immediate 1 to PC; sets `x25 = 4208`.

Simulation Output:

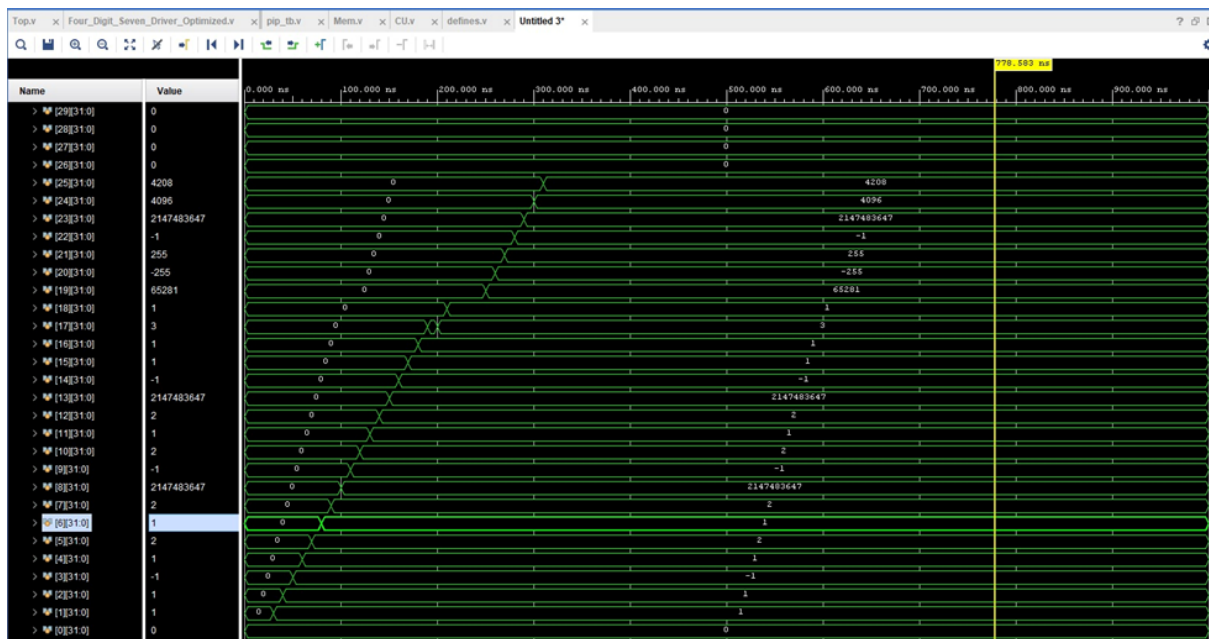


Figure 5: Test2 Simulation Output

The verification of both Test1 and Test2 simulations shows correct behavior. The registers and memory states match the expected values, demonstrating proper implementation of arithmetic, logical, bitwise, and memory operations. This detailed trace validates the processor's functionality and confirms correct handling of all instructions.

Bonus

1) Test Program Generator

A significant feature of this project is the test program generator, implemented in C++. This generator creates a sequence of random but valid instructions to test the processor. The input to the program is the number of instructions to generate, and the output is a `.txt` file containing the generated instructions.

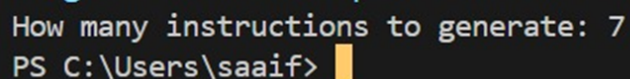
Example Output:

- **Input:** 7 instructions
- **Output:** A file with 7 random but valid RISC-V instructions

TestCase.txt generated (from input 7) example:

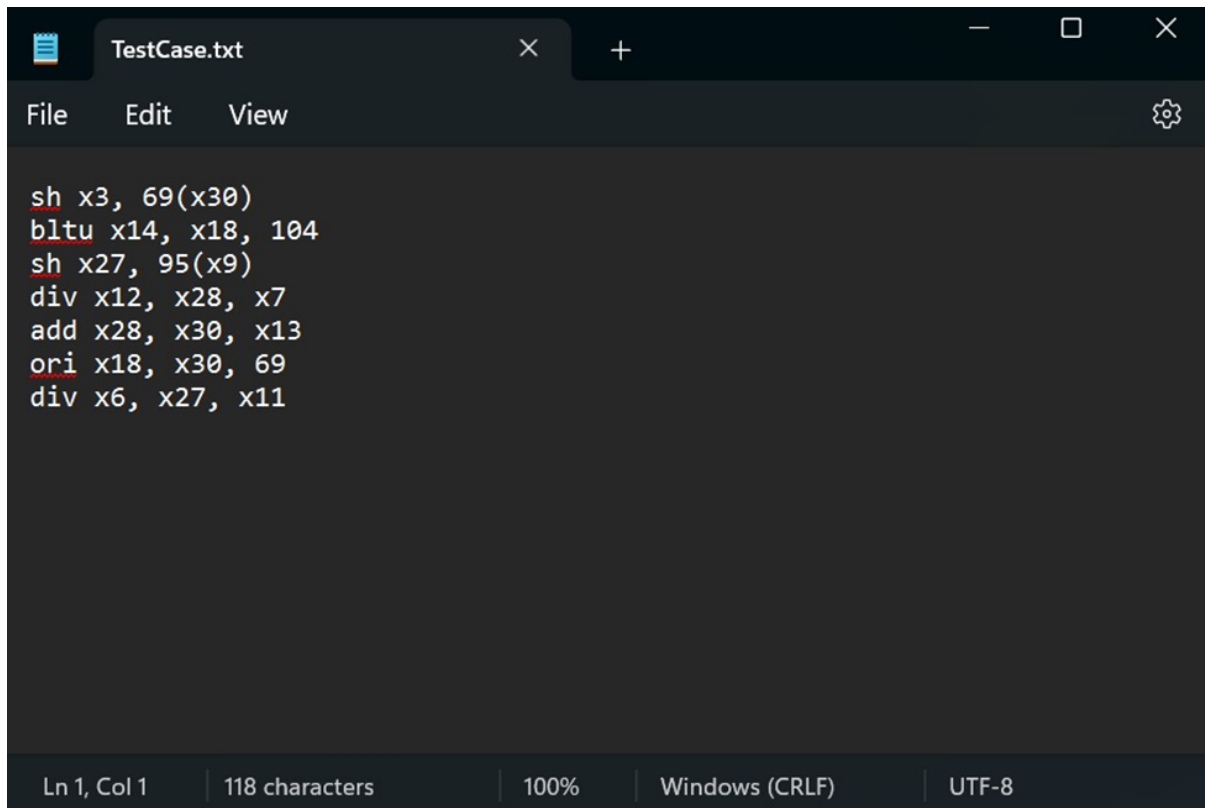
```
1 sh x3, 69(x30)
2 bltu x14, x18, 104
3 sh x27, 95(x9)
4 div x12, x28, x7
5 add x28, x30, x13
6 ori x18, x30, 69
7 div x6, x27, x11
```

This tool is useful for ensuring comprehensive testing by automatically generating diverse instruction sequences that cover different instruction types and potential edge cases.



```
How many instructions to generate: 7
PS C:\Users\saaif>
```

Figure 6: Instructions input



```
sh x3, 69(x30)
bltu x14, x18, 104
sh x27, 95(x9)
div x12, x28, x7
add x28, x30, x13
ori x18, x30, 69
div x6, x27, x11
```

Ln 1, Col 1 | 118 characters | 100% | Windows (CRLF) | UTF-8

Figure 7: Enter Caption

FPGA Testing

FPGA testing on the Nexys A7 platform has proven to be an essential part of the testing. The FPGA allows for real-world testing of the processor design, validating the performance of the processor in hardware. The FPGA platform helped identify any timing issues, incorrect hazard handling, or pipeline drawbacks that might not have been obvious during the simulation.

Conclusion

This report outlines the successful implementation and testing of the femtoRV32 processor. By following the RISC-V specifications and incorporating pipelining and hazard management techniques, we achieved a functional processor capable of executing a wide

range of instructions. The use of FPGA hardware for testing further validated the design, providing confidence in the correctness of the implementation.

References

- RISC-V Specifications
- Nexys A7 FPGA Documentation