**CSCE 3301 – Computer Architecture Spring 2025**

**Project 1: femtoRV32  
 RISC-V FPGA Implementation and Testing**

**Team Members:**

Rodayna Elkhouly 900221860

Karim Fady 900222806

**Instructor:** Dr. Cherif Salama

**Date:** 5 May, 2025

***Project Objective:***

The goal of this project was to design and implement a **pipelined RISC-V processor** that supports the **RV32I instruction set**. The RV32I is the base integer instruction set for RISC-V, and the implementation must support all the instructions in this set, excluding certain halt instructions (ECALL, EBREAK, PAUSE, FENCE, and FENCE.TSO). The processor must be pipelined to enhance throughput and performance, while also managing potential hazards (data, control, and structural) that arise due to the use of a single-ported memory system. Additionally, the project involves testing the processor thoroughly to ensure it meets all functional and performance requirements as defined by the RISC-V specification.

***Implementation:***

The first step in the project was to implement all 42 instructions required by the RV32I instruction set. Each of these instructions was then tested thoroughly to ensure that the processor correctly executes them.These include arithmetic (ADD, SUB etc…), logic (AND, OR, load etc…), /store (LW, SW etc…), branch (BEQ, JAL etc…), and immediate (ADDI, LUI etc…), instructions.

Testing Process: For every instruction, we developed test cases, and we confirmed proper operation by comparing the outcomes in simulation and FPGA hardware. Verifying register values after execution, guaranteeing proper memory access, and making sure control and data dangers were appropriately managed were all examples of test cases.

The second step is that, the processor was implemented with a 5-stage pipeline, which is more common in modern processors to improve throughput. The stages are:

1. Instruction Fetch (IF): Fetches the instruction from memory based on the program counter (PC).
2. Instruction Decode (ID): Decodes the instruction, reads registers, and prepares necessary data.
3. Execute (EX): Executes the ALU operation or calculates branch target addresses.
4. Memory Access (MEM): Accesses memory if required (load or store operations).
5. Write-back (WB): Writes the result back to the register file.

Instead of utilising hazard detecting units, we handled hazards through operations and, where needed, included stopping. This made it possible to effectively manage hazards without adding more complexity.  
  
Data Hazards: Upon detecting a data hazard, such as a register that is being written to but not yet accessible for reading, the pipeline is halted until the necessary data.  
  
Control Hazards: To prevent executing the wrong instructions, the processor stalls or appropriately resolves branch predictions.

Instead of using **single-ported memory**, your team implemented **two memory units**; one for **instruction memory** and another for **data memory.** This avoids structural hazards caused by simultaneous access to both instruction fetch and data read/write operations.

***Issues***

**Memory Issue:** The use of single-ported memory caused structural hazards when both instruction fetch and memory operations occurred simultaneously, degrading performance.

 **Solution:** We switched to two memory units (one for instruction fetch and one for data access), resolving the issue and improving performance. Only for the submission and we are still trying to implement this before our demo.

**Branching Issue:**  
We encountered challenges with handling branching in the processor, particularly in ensuring that branch instructions were executed correctly across the pipeline. To address this, we developed a dedicated **branching module** that centralizes and manages all branch-related operations, ensuring accurate branch prediction and seamless execution of branch instructions.

**Pipelining and Wiring Issues:**  
During the implementation of the pipelined architecture, we faced several challenges related to the **wiring and connections** between the different pipeline stages. Ensuring proper synchronization and data flow between stages was complex, especially with the addition of hazard handling and memory access. These issues initially led to incorrect data propagation and timing mismatches across the pipeline.

Screenshots:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated A screenshot of a computer program

Description automatically generated A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

Schemantic:

A diagram of a machine

Description automatically generated

**Bonus implementation:**

**Test Program Generator:**

We implemented this feature using C++ ; which is an additional feature that generates random but valid test programs for the processor, useful for automated testing.

A screen shot of a computer

Description automatically generated