THE AMERICAN UNIVERSITY IN CAIRO
الجامعة الأمريكية بالقاهرة

CSCE330101: Computer Architecture

Report: Group Project

Title:

(RISC-V FPGA Implementation and Testing)

MS-4

By

Tarek khalifa: 900172646
Nada Askar: 900171451
Omar Hesham Salah El-din: 900160903

Submitted to:
Dr. Cherif Salama

Submission Date:
July 11, 2019

Brief Description:

In this project, we are required to build a RISC-V pipelined processor with a single memory. As the final milestone, we designed a pipelined RISC-V processor with a single, single port, byte-addressable memory for both the instructions and data. The processor supports the RV32I base integer instruction set with thirty-eight instructions implemented in Verilog. To build the pipelined processor, we added the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB). In addition, we also designed and implemented a forwarding unit that forwards the operands available at the EX/MEM register or at the output of the MUX after the MEM/WB register to the ALU inputs at the EX stage. A hazard detection unit and instruction flushing to handle the incorrect outputs due to hazards was also designed and implemented. After the pipelined processor was done using Verilog description, a datapath block diagram was designed to represent it. The pipelined processor capable of hazard handling we designed is composed of the old single cycle processor datapath elements and control, in addition to the pipeline registers. The datapath includes the PC, Instruction Memory, Register File, Immediate Generator, ALU, Data Memory, Branching Unit, a shifter, forwarding unit, hazard detection unit, instruction flushing and a number of multiplexers to select between multiple inputs to certain units and adders. To implement the processor, we first updated the design of the single cycle processor to include the pipeline registers. Then, we designed, implemented and simulated the forwarding unit, hazard detection unit and the instruction flushing. Then, we modified the two separate memories for the data and the instructions and created one single memory containing both. After that, we started adapting the 38 instructions to the new design. Finally, we created a test program to check the correctness of the processor, then tested the processor's implementation using the FPGA. The test program included sample instructions for all the supported instruction types (such as lw,sw,add). It was used to check the processor's behavior when handling each supported instruction and whether it worked correctly with each. After we finished the basic implementation of the processor, we updated the processor to support the compressed instructions. We also moved the branch outcome and target address computation to the ID stage.

Bonus Features:

- Supporting the compressed instructions to support the full RV32IC instruction set, except for the excluded instructions.
- Moving the branch outcome and target address computation to the ID stage.

Implementation Details:

1. **Stage I: Pipeline Design and Implementation**

   In this stage, we updated the design of the single-cycle processor to support pipelining.

   - To update the design:
     - Pipeline registers were created and incorporated in the old design.
     - To implement that, we used the code provided in the lab with the registers and their connections in it.

2. **Stage II: Hazard Handling**

   In this stage, we designed and implemented the forwarding unit, and instruction flushing.

   - Forwarding Unit:
   - The Forwarding unit was created using the given conditions of how it operates depending on the pipeline register values. We checked on the conditions and used assign statements to give values to the two outputs of the forwarding unit (forwardA, forwardB).
   - Then, two 4x1 multiplexers were created to determine the inputs going into the EX stage using the outputs of the Forwarding unit as the select lines.
   - Instructions Flushing:
     - First of all, the instructions flushing is not implemented as a separate unit. It is implemented directly in the datapath by following certain steps.
     - Instructions Flushing Steps:
       a. Flushing the instruction that is being fetched at the moment: changing that instruction to a NOP instruction. This is implemented by first adding a 2x1 MUX and using the PCSrc signal as its selection line. If the PCSrc is equal to 1, instead of providing the IF/ID register with the read instruction from the instruction memory, it is provided by ‘add x0,x0,x0’ which is a NOP instruction that does nothing.
       b. Flushing the instruction that is being decoded: the control values stored in the ID/EX are all stored as zeros instead of the original control values. This is achieved by changing the selection line of the 2x1 MUX that provides the control values to be the result of

(PCSrc). Therefore, when PCSrc is equal to 1, the control values entering the ID/EX register will become zero and the decoded instruction is hence flushed.

    c. Flushing the instruction that is being executed: similar to flushing the instruction that is being decoded, the instruction that is being executed is flushed by first adding a 2x1 MUX that provides control values to the EX/MEM register. This MUX's select line is controlled to provide control values as zeroes when the PCSrc is equal to 1.

3. **Stage III: Single Memory Creation**

In this stage, we changed the two separate memories with one for data and the other for instructions into a single, byte-addressable memory for both.

- The test instructions were first copied to the new memory. Then, the initiated values in the memory were copied as well to the new single memory.
- The inputs of the new memory are the same as those of the data memory being clk, rst,MemRead signal, MemWrite signal, [9:0] addr, [31:0] data_in,fun3, and the output being data_out. However, a new input is added which is the half clock (hclk).
- The new half clock is created using an always statement; the half clock changes at the positive edge of the original clock. It controls whether the data or the instruction are outputted from the memory.

4. **Stage IV: Modules Update**

In this stage, we included the pipelined processor module. Then, updated that top module to include the new single memory, the forwarding unit, and the instructions flushing. In addition, we modified certain modules to match the new design.

- The Modules:
- RISCV:
    - First, we added the pipeline registers to the module.
    - Then, we removed the two instantiated memories and replaced them with an instantiation of the single memory we created.
    - We instantiated the forwarding unit and the newly created 4x1 multiplexers that complete the forwarding phase[explained in detail in a previous stage].
    - We implemented instructions flushing using 2x1 multiplexers [explained in detail in a previous stage].

- ForwardingUnit:
  - In this module, we created the forwarding unit by using the assign statement to give values to the outputs forwardA and forwardB after checking conditions based on the given inputs.
- Mux4_1 and Mux4_2:
  - In those modules, output forwardA is used as a select line to Mux4_1 while the output forwardB is used as a select line to Mux4_2 and both are used to provide the inputs entering the EX stage.

5. Stage V: Bonus Features

In this stage, we designed and implemented two bonus features. We updated the RISCV main module after each bonus to include them correctly.

- Supporting the compressed instructions:
  - First, we adjusted the 'defines' file adding the Opcode, func3, func2,func2 of the R form,fun4, and fun5 of the compressed instructions.
  - Secondly, we created a module to handle the compressed instructions: In this module, the compressed instruction is first taken as input.
  - Then, the instructions are checked by passing the opcodes of the instructions and their func3 values as cases to determine what type of instruction it is. After determining the instruction, it is converted into its corresponding 32- bit instruction by concatenating the instruction fields and the needed extra numbers such as extra zeroes to fill in the bits in the 32-bit instruction.
  - Finally, we added a 4x1 multiplexer to determine the correct instruction. Also, we added a flag to indicate if we have a 32-bit instruction or a compressed one.
- Moving the branch outcome and target address computation to the ID stage:
  - First, we moved the branch control unit and the PC Control Unit to the ID stage.
  - Secondly, we implemented the forwarding mechanism to handle the case where either rs1 or rs2 are not ready with the correct values yet.
  - Thirdly, to perform branching, we flushed the instruction by zeroing the control signals entering the EX stage.
  - Fourthly, we used the stalling mechanism to stall when the load instruction is directly preceding a branch instruction.

6. Stage VI: Testing

- In this stage, we created a test program to check the behavior of the pipelined RISC-V processor. The program has instructions that correspond to the instruction types. The program is available in the attached files.
- We then created a test bench to simulate the processor.
- The test program and the separate test cases are attached with the project. The test program also exists in the memory, but commented.

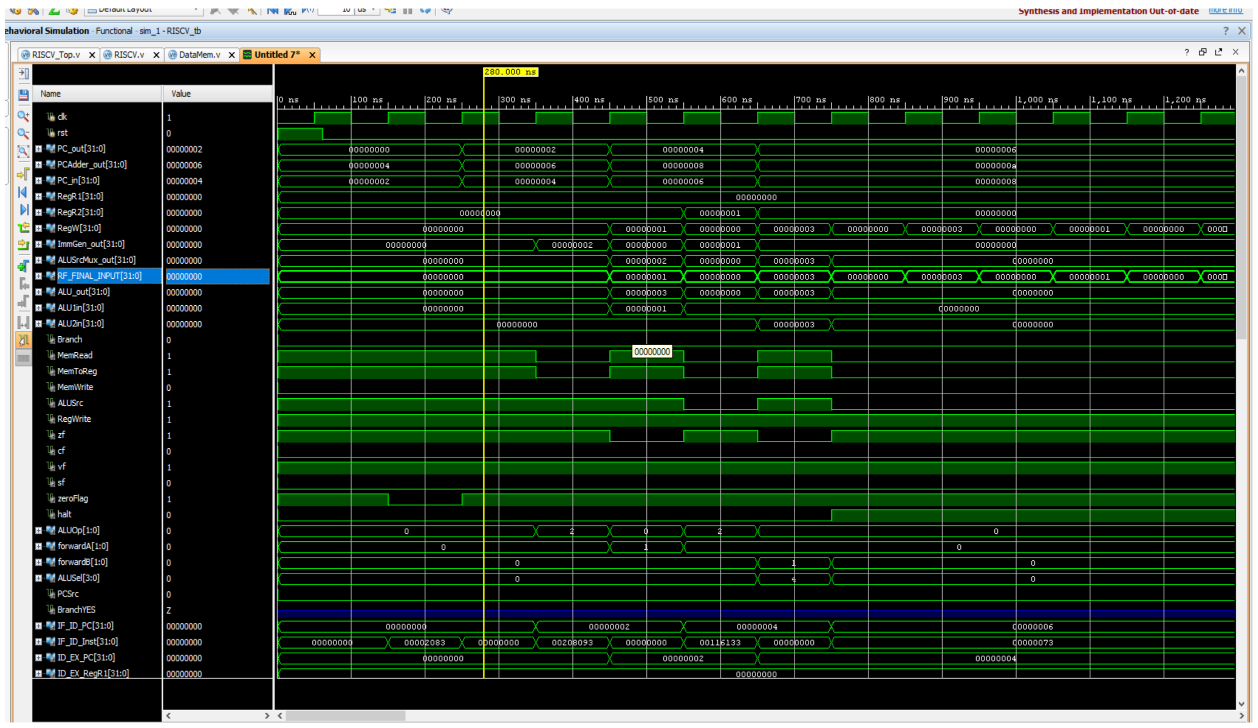
<u>FPGA Implementation:</u>

- The FPGA Implementation did not succeed. Although the bitstream was correctly generated, it did not work correctly on the FPGA. However, the simulation we ran previously on the project behaved correctly and as expected in both the basic pipleined implementation with hazard handling of the project and with the two added bonuses. The simulation behaved accurately with all the tested instructions.

<u>Simulation:</u>

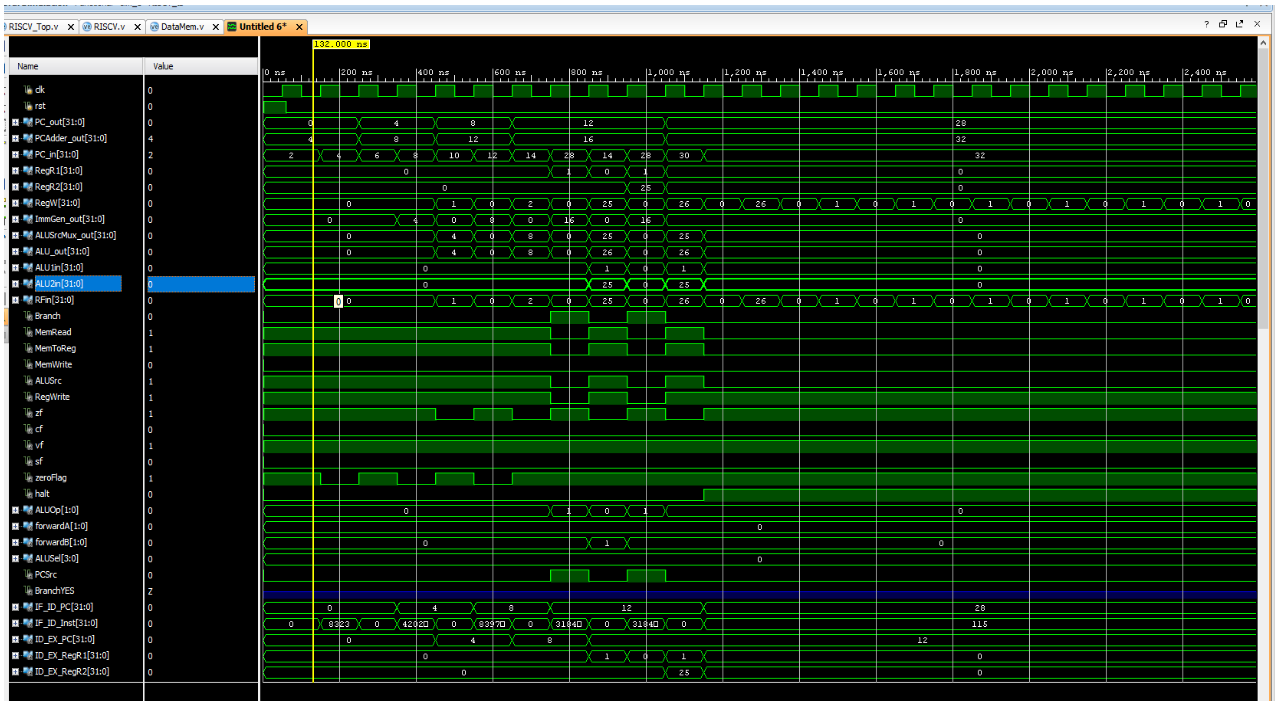
- The following screenshots show the behavior of the pipelined processor with respect to the test program's instructions.
- The instructions were processed correctly and the processor behaved as expected according to the test program available at the memory (it is currently commented since we used some other test cases after.)
- The following screenshots show the processor's behavior in the two cases of the added bonus features.

First feature: The comptessed instructions



The instruction loaded succesfully into the register after the compressed instruction was converted into a normal instruction. 1 was loaded to the register and then 2 was added so the output was 3 as shown in the signal final rf input. Oring x2 with 3 resulted in 3 because x2 is 0. Then the ebreak worked correctly.

Second feautre: Moving the branch outcome and target address computation to the ID stage



In this screenshot, the test case performs a load instruction followed by a branch instruction. This leads to stalling. After the stalling, the instruction jumped successfully which means we really stall and the branching was moved to the id stage. The ebreak ended the program and PC freezed correctly.