***CSCE 3301 – Computer Architecture Fall 2019***

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

**Introduction**

This project is an implementation of a RISC-V32i central processing unit. This program is implemented using Verilog Hardware description language and displayed through simulations and on an FPGA. The instructions that are implemented on this processor are all the R instructions, I instructions, S instruction, B instructions, U instructions, and J instructions.

**Implementation**

The project is composed of many modules that represent each function that the processor is supposed to do. These modules that make up the processor are mainly the control unit module, the register module, the registers file module, an adder module, the ALU module, the ALU control unit module, the immediate generator module, the instruction memory, and the data memory modules, and a multiplexer module that gets called during many times in the data path. Each of these separate modules was tested independently using a dedicated test bench for it.

Some things were added to these main modules to get all the 47 instructions working correctly. The main things that were added to the data path were

1- A multiplexer before the register file to control the data that will be written into the rd field of the register file so that it writes the expected value in case of LUI, JAL, JALR, AUIPC

2- A branch unit that handles all the branch instructions such as BEQ, BNE, BLT, BGE, and BLTU

3- A multiplexer that chooses between the pc and rs1 before the adder that increments the program counter by the offset provided by the immediate generator to handle writing (rs1+immediate) in the write data field of the register file in case of JALR instruction

Some control signals were also added to the control unit to activate J and U instructions:

1- 2-bit control signal that will get passed to the multiplexer that controls the write data of the register file, so these 2 bit are changed according to the instruction that needs activation

2- 1-bit control signal that controls the multiplexer that chooses between the pc and rs1

A very critical part of the implementation was the control signals that are generated in the control unit as they will basically control every other module in the data path. The table below illustrates the control signals used to control the data path. PC-gen-sel is the signal that acts like the selection line of the multiplexer that chooses between the pc and rs1 to handle the JALR instruction. Rd- sel describes the control signal that controls the multiplexer that chooses which value will be written in the register file.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | BRANCH | memread | memtoreg | Alu-op | memwrite | Alusrc | regwrite | Pc-gen-sel | Rd-sel |
| R-format | 0 | 0 | 0 | 10 | 0 | 0 | 1 | 0 | 00 |
| Load | 0 | 1 | 1 | 00 | 0 | 1 | 1 | 0 | 00 |
| Store | 0 | 0 | 1 | 00 | 1 | 1 | 0 | 0 | 00 |
| Branch | 1 | 0 | 0 | 01 | 0 | 0 | 1 | 0 | 00 |
| JALR | 0 | 0 | 0 | 00 | 0 | 0 | 1 | 1 | 10 |
| JAL | 0 | 0 | 0 | 00 | 0 | 0 | 1 | 0 | 10 |
| I-format | 0 | 0 | 0 | 11 | 0 | 1 | 1 | 0 | 00 |
| AUIPC | 0 | 0 | 0 | 00 | 0 | 0 | 1 | 0 | 01 |
| LUI | 0 | 0 | 0 | 00 | 0 | 0 | 1 | 0 | 11 |

Another critical part that guided the main execution unit (ALU) in the processor is the ALU control unit as it propagates the codes that will instruct the ALU what operation to execute. Hence, the following table lists all the codes that instructs the ALU about the operation it should execute.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Func3 | Func7 | op\_out |
| 00 | xx | xx | 0000 |
| 01 | xx | xx | 0001 |
| 10 | 000 | 1 | 0001 |
|  |  | 0 | 0000 |
|  | 111 | xx | 0101 |
|  | 110 | xx | 0100 |
|  | 100 | xx | 0111 |
|  | 101 | 1 | 1010 |
|  |  | 0 | 1000 |
|  | 001 | xx | 1001 |
|  | 010 | xx | 1101 |
|  | 011 | xx | 1111 |
| 11 | 000 |  | 0000 |
|  | 111 |  | 0101 |
|  | 100 |  | 0111 |
|  | 101 | 1 | 1010 |
|  |  | 0 | 1000 |
|  | 001 |  | 1001 |
|  | 010 |  | 1101 |

**Testing**

To test the project, we started by testing each of the modules separately to make sure that they are working correctly on their own.

We first started by the R instructions, by simulating the processor and working out the calculations ourselves, we made sure that all these instructions are working perfectly by checking the destination register.

To test the load and store instructions, we tried at first to simulate a regular load instruction then through displaying it from the register, we made sure that it is working correctly. Then, to test the store instruction, we loaded the same value that was stored by the instruction to make sure that it loads the same number. We used the same method to test the storing and loading of the half word and bytes.

To test the branch and Jump instructions, the calculations of adding the addresses and the offsets were worked out. Then, by comparing them to the simulation results, we made sure that they are working correctly. The same thing was done to test the jump instructions. It was mainly done through working out and tracing the operations ourselves and making sure that they are working correctly on the simulation.

After making sure of that, all of the modules were integrated together in the data path module to check the functionality of the processor. All the 47 instructions were added to the file from which the instructions get fetched from.

**Results**

* All the test benches of the modules successfully passed all the test cases defined for them in each of the test benches.
* By testing and debugging the integrated module (the data path) that tests the 47 instructions that are handled by the processor, all of them generated the expected output and followed the path that we predicted from the beginning

All the waveforms that show the results of the simulation are included in the folder of the project along with the sources files and the simulation files.