Computer Architecture Project MS3 Report

Due:5th May 2020

Name: Omar Mahdy

ID: 900160493

Name: Mohamed Ashraf Taha

ID: 900172754

- **<u>Project Goals & Design:</u>**

In this submission we try and implement a pipeline RISCV processor supporting All RV32I Base Instruction with CSR, Fence, and FenceI implemented as nops and Ecall implemented as a termination instruction ,Set as well as eight instructions from the RV32M Standard Extension (Multiplication & Division). Our main goal is for our implementation to work correctly on all our implemented instructions and for it to work on all combinations of inputs. This is still not achievable for this millstone as we are only required to test basic cases, but we plan in future milestones to test all possible combinations for all instruction to achieve this goal. Also, we aim to make our code readable as much as possible. We achieve this by simply trying to label all wires and reg appropriately especially in the top module titled Full_Datapath. You can see that most of the wires and registers and modules are named exactly the same like that one in the lectures slides and our block diagram or with similar naming making it easier for you to know which wire corresponds to that in the diagram and all Multiplexers are labeled as MUX's with a name to signify what they are for. Also, you will find that all outputs in the top module of any module will have the word “out” in them to show that they are outputs for a certain module, except if they are used for a selection line for something then they will have the word “sel” in them. Moreover, for human readability we try and used the defined bits given in the define file. We do this to make our code less prone to human errors, as well as make it as much readable as possible. Lastly, we try and use more case statements than if and else statements to make the code easier to read and compile. Another thing we try to avoid doing is using the shorthand if’s instead of MUXS to make the code clearer. In addition, we try and implement we as little hardware cost as possible in our capabilities. For instance, you will find that if a single bit is not needed for a case or if statement, we will not take, this can be seen clearly in the ALU,the Data memory and ALU control modules. In the pipelined implementation we aim to make all instructions work and avoid data, control and structural hazards from the single ported memory.

- **<u>Implementation</u>**

In our pipeline implementation we converted our single cycle implementation to pipelined. Firstly, we created wires for all 4 registers in the pipelined implementation both for input and output from and to the registers, respectively. While creating the registers we count the number of bits for input into each register the create then the appropriate bit size. We then add the correct variables to the correct wires in the pipeline approach for instance we added wire IF_ID, IF_ID_out, IF_IDdataout, IF_ID_PC. We set

IF_ID_PC to our PC_Out and , IF_IDdataout to our instruction then we set the most significant 32 bits of IF_ID to IF_ID_PC and the least significant 32 bits of IF_ID to IF_IDdataout using concatenation in Verilog . We then pass IF_ID which now has 64 bits to the register to have the appropriate delay ans to fill IF_ID_out. We implement the 3 others registers in the same way as well. In our single memory approach, we created a slow clock called clk_slow that changes every 2 clock cycles of the normal clock, this clock enters the pc register to delay it. And we input it also in the single memory module to act as a selection line to as when it's 1 we want the single memory to act as data memory and if it is 0 we want it to act as instruction memory. The way we implement our memory to avoid over writing of instruction or data memory outputs is that all data memory outputs are put at the end of the memory and all instruction outputs are put the beginning. We set out memory to a very huge size to minimize the chance the 2 sides collide. Lastly, we added the forwarding unit, the hazard detection unit and branch flushing to our pipeline. The forwarding unit works by checking RS1 and RS2 to RD in both the memory and writeback stages, if either Rs1 or Rs2 equal to either Rd we then forward appropriately. For branch Flushing we check the branch control signal of the instruction. And if it is 1. flushing happens for freezing the PC using the stall signal generated from the hazard detection unit.

- **Testing Procedure and Results**

We tested by adding the machine code, the 32 bit instruction in the instruction memory and commented the equivalent assembly code next to it. We got the machine code from the RARS emulator we wrote the instruction we needed and rars gives us the instruction in hexa we then use an online converter to convert the hexa instruction to the equivalent 32-bit instruction, as well as the expected output for a certain instruction. We tested and checked all 47 instructions at least once and checked their outputs and they we found to be correct for the cases we choose. We first tested without hazard detection using 3 nop instructions after each instruction then after implementing the hazard detection we then removed the nops. In both cases we found the results we as expected.