CSCE 3301 Project

Milestone 4 Report

Department of Computer Science and Engineering, AUC

Done by:

Kareem Kassab 900182771

Mohammed Zaieda 900181473

Omar Elsayed 900183884

1. Introduction.

This milestone was basically a finalization of all missing instructions and instructions that were not working properly. We also tried to implement the bonus feature of the branch outcome in the decode stage, and also started in implementing the bonus feature of supporting the integer and multiplication instructions but did not instantiate it in the top module.

2. Development & Functionality

Load and Store: The load and store instructions were handled by adding 2 outputs in the ALU control unit. We used them as inputs in the data memory to check if it's a load byte, half-word, or a word. To support them we altered the control unit to allow for more incoming flags along more case statements to handle them in the ALU module. All 8 load and store instructions were supported and produced correct outputs as shown below in the screenshots.

JAL, JALR, AUIPC, LUI: For these instructions, we added a 4x1 mux that takes the following inputs: pc + 4, branch outcome, jalr, and jal. The selection line is a 2 bit register that is set in the control unit to handle the previous instruction with regards the pc output. There is another 4x1 mux before the register file with the following inputs: auipc, jal/r, final_out, and lui. Its selection line was also set in the control unit to handle which result it will write in the register file.

Ecall, Ebreak, Fence: Ecall instruction was supported by adding a one-bit wire as the PCWrite. This bit is initially set to zero and then set to 1 when it gets to an ecall instruction. This wire is declared as a register in an always block in the top module, then the negation of this register is passed to the PC register as its load wire. This prevented the PC to move any forward and halted the simulation. For the fence and ebreak instructions, a module was created that takes the opcode of the instruction and spits out a flag bit that indicates whether this instruction is fence or ebreak or not. If the instruction was any of them, then this flag is set to 1. This flag bit is then used to be a selection line for a 2x1 mux that chooses between the normal instruction getting out from the memory and the nop instruction. If the instruction was ebreak or fence, then a nop instruction is passed to the IF_ID register instead of the normal instruction.

I-Format: The I- format instructions were heavily depending on the immediate generator given, and by changing to the second source register to an immediate. These were supported by adding cases in the control unit and handling them in the given ALU module with no extra changes needed. Note that most of the I-format instructions were initially R-format instruction with slight changes of source registers and handling them using the immediate generator and ALU.

R-Format: For the R-format instruction there were minimal modifications made to support the missing instructions that were not done in the Lab, so what we did is adding more case statements in the control unit to further accept more instructions. These instructions were also tested with provided screenshots as a proof of validity.

Branch: we made a module that takes flags, and [14:12] bits of the instruction. Using if statements we detect the type of the branch instruction. We assign the branch output = 1, when these conditions were satisfied. We also started implementing a branch outcome unit in the decode stage, which required another forwarding unit as explained in the forwarding unit section to handle any data hazards for moving it in the decode stage.

Shifter: The shifters were handled by following the same inputs and output sequence given in the ALU module. The logical operations were simply handled using the shifts operators in Verilog and assigning these to the output register when their conditions are satisfied.

Forwarding Unit: The forwarding unit had zero change from what we have done in the lab course. But in a nutshell, the forwarding unit takes inputs from the Execute/Memory register and the Memory/Writeback register with source registers to check on the specific conditions of whether or not the instruction incoming requires forwarding from a the execute stage or from the memory stage. We added a 4x1 mux before the ALU for both inputs to satisfy the selection lines exiting from the forwarding unit.

Pipeline implementation: The pipeline implementation was very simple in theory; we used the same module of N-Bit register that takes N-bits which were altered whenever we added registers into the pipeline. These 5 registers helped us to execute instructions simultaneously and access register values whenever we wanted in whichever stage. This pipeline implementation was also done in the lab course.

Single Memory: We initialize the memory to be 128 bytes, this memory is divided into two segments, one of them is responsible for storing data while the other one stores the instruction, we made the instructions to be stored at the first 64 bytes of the memory and the data to be stored in the second part, we determine the starting address of the data segment using a base variable that holds the value 64. Each time we instantiate this module, we give it an address, we fetch both the instruction and the data located at that address, then we choose one of them using a 2x1 mux in which its selection line is the slower clock, if the slower clock = 0 then the output to be chosen is the data else if the slower clock = 1 then the output is the instruction. Also, the logic of loads (lw, lb, lh, lbu,lhu) and stores (sb, sh, sw) is managed inside the module. The 2 bit state input determines the instruction is (lw - sw) or (lb - sb) or (lh - sh) and the 1 bit sign input determines whether the instruction is signed or unsigned. If the instruction is signed then the rest of the bytes extended with last bit of the number, but if the instruction was unsigned then the extension happens with zeros. Note that we do not write or read unless we check the control signal of MemRead and MemWrite signals.

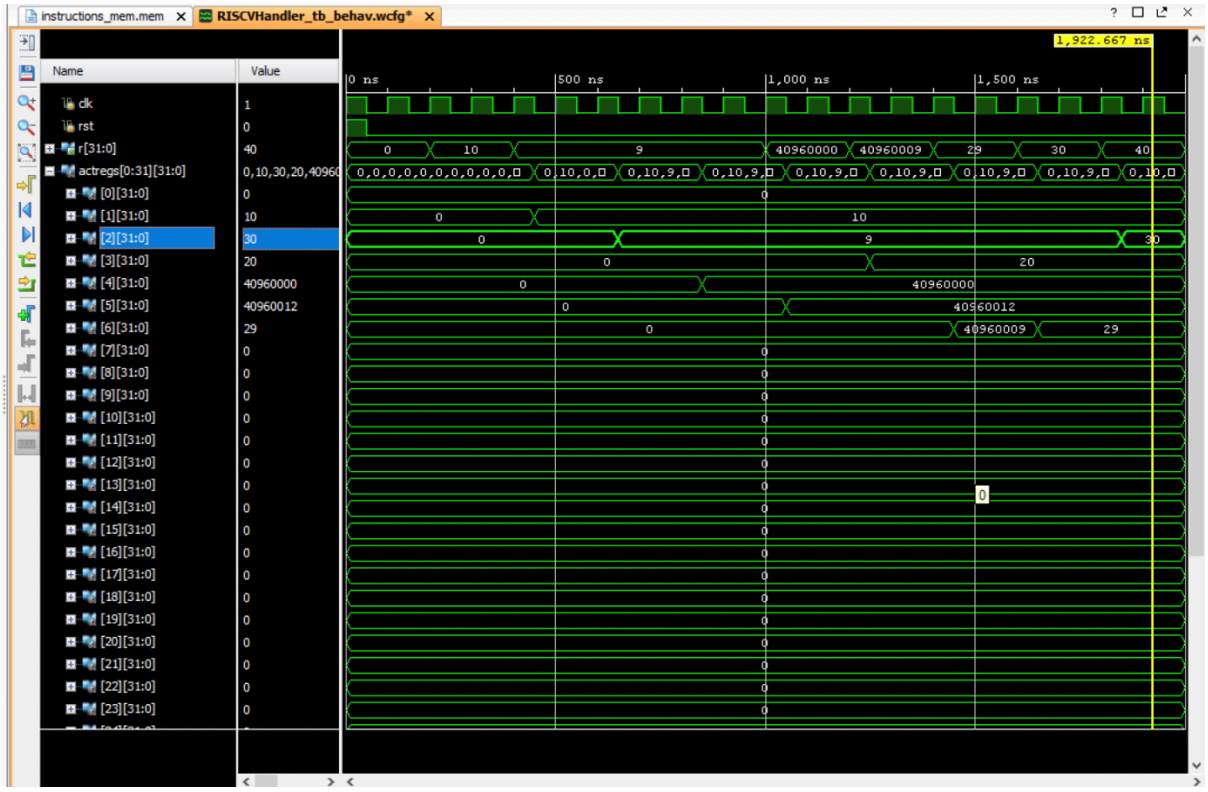
RISCV32M: In this instruction set we were required us to handle 8 more instructions. These were basically variations of integer multiplication and division. We had to add cases in the control unit such that it accepts the opcodes of such instructions (these data were brought from the instruction reference), and handles certain output registers that are read in the ALU control unit. The ALU control unit reads these selection lines and sends to the ALU further outputs. The logical operations in the ALU handled the extended logic of such instructions in this instruction set.

Branch in Decode: In this module ,the logic of the branch outcome was done all over again, which is basically taking two the two input sources and assigning flags with certain values whenever the conditions are satisfied. This included handling the unsigned part of the branches which is basically taking the opposite of a register whenever the last bit is 1,

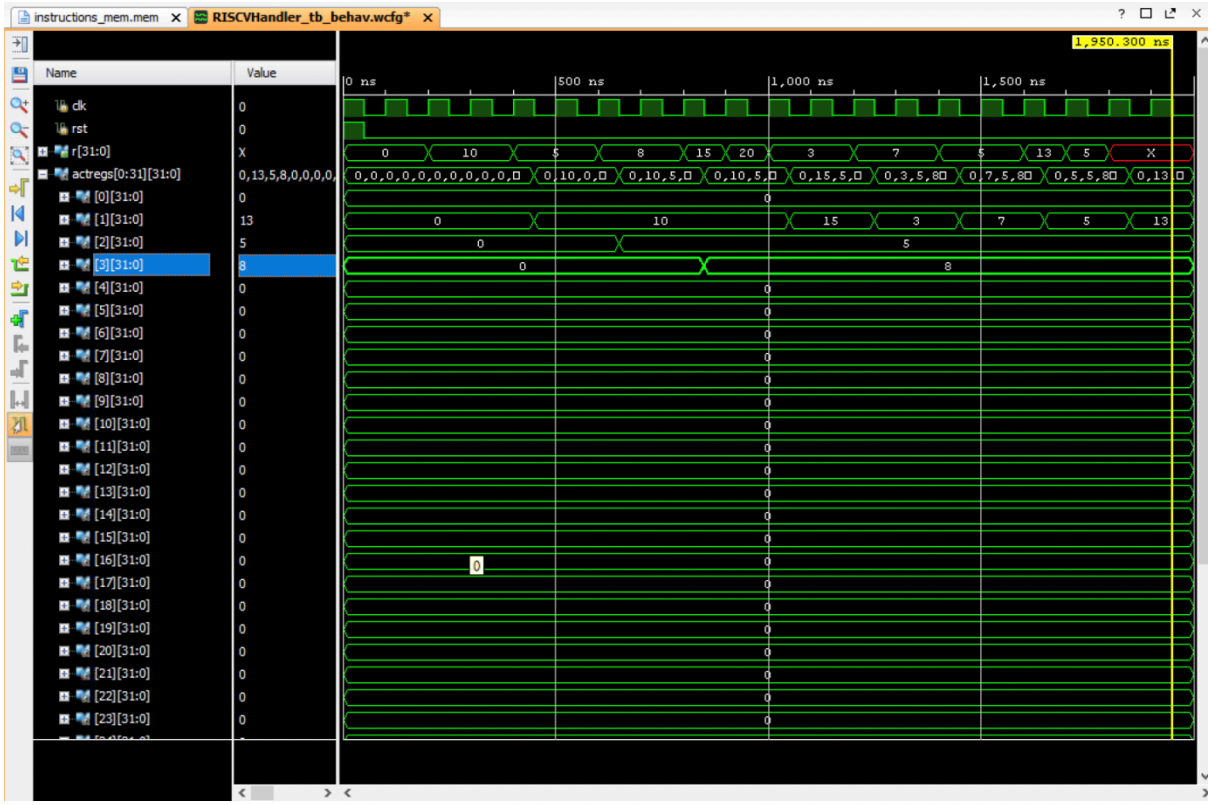
indicating its negative. We've also handled data hazards by adding another instantiation of the forwarding unit in the decode stage to prevent any form of data hazards.

3. Testing:

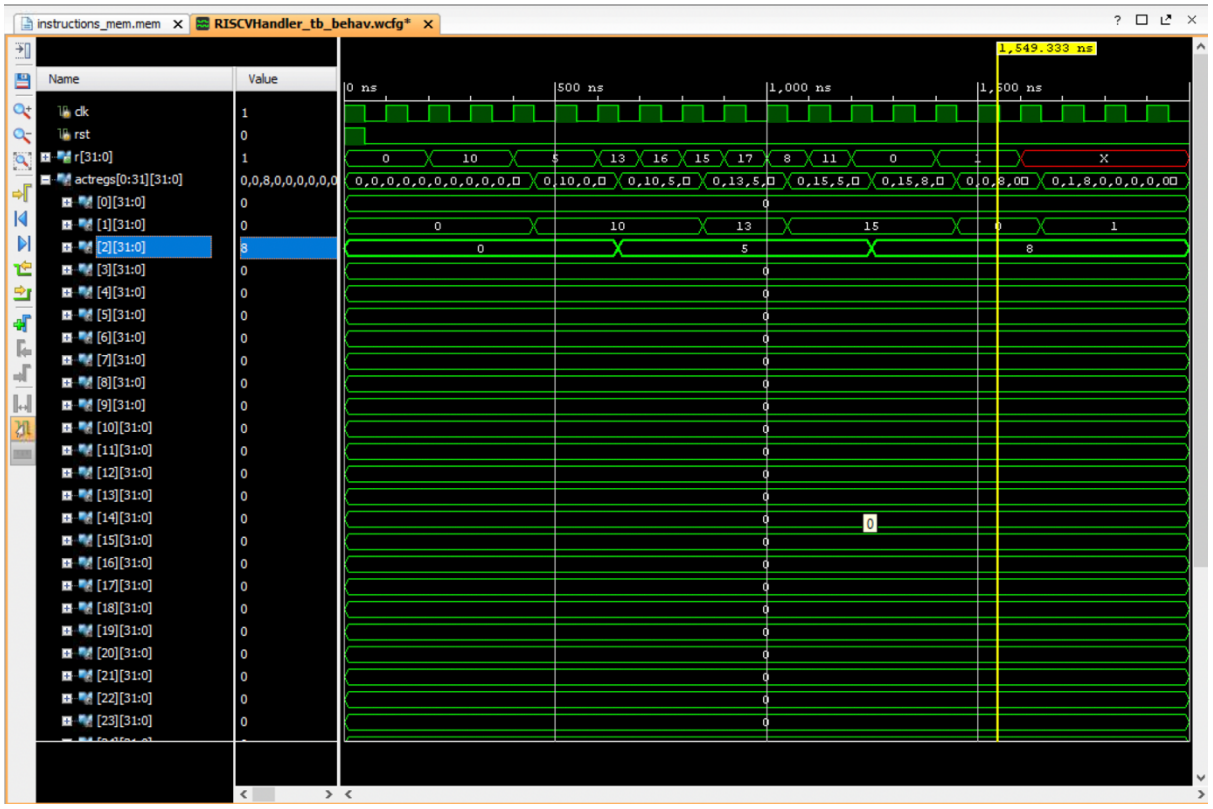
This is lui-jal/r-auipc.txt file test case.



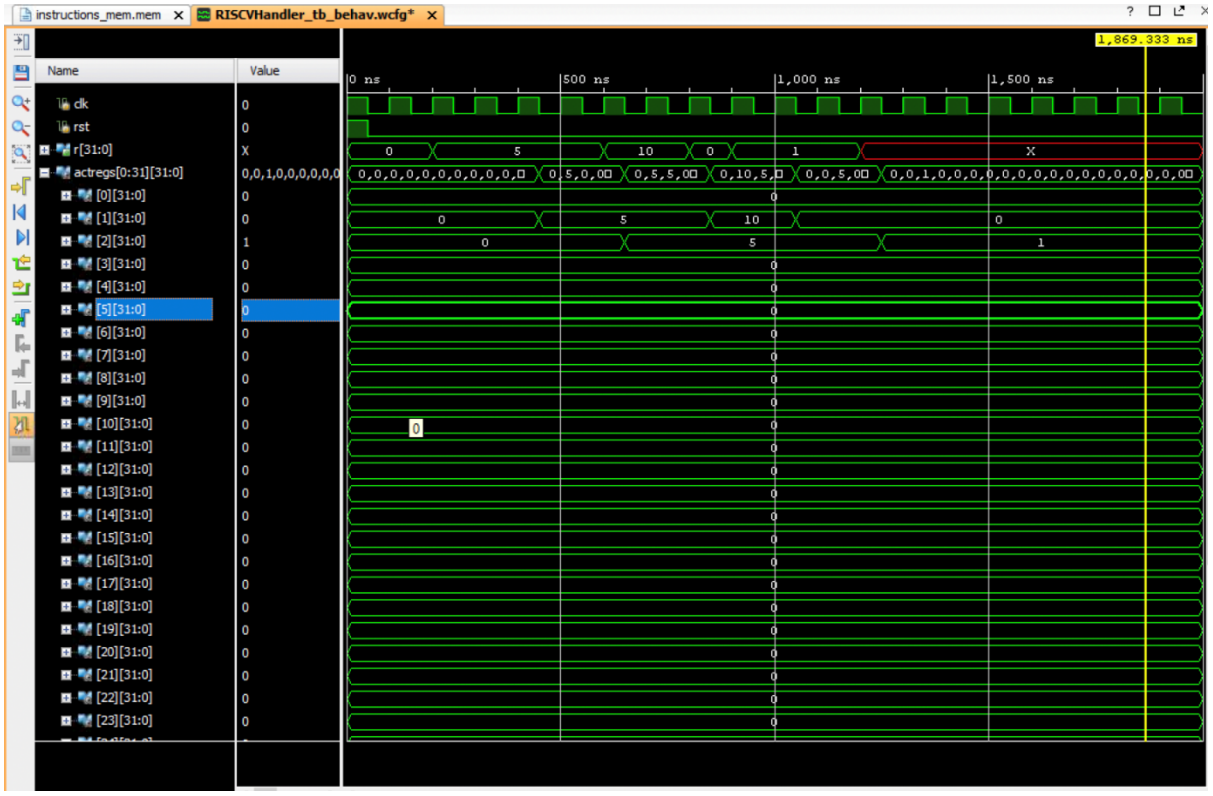
This is R.txt file test case.



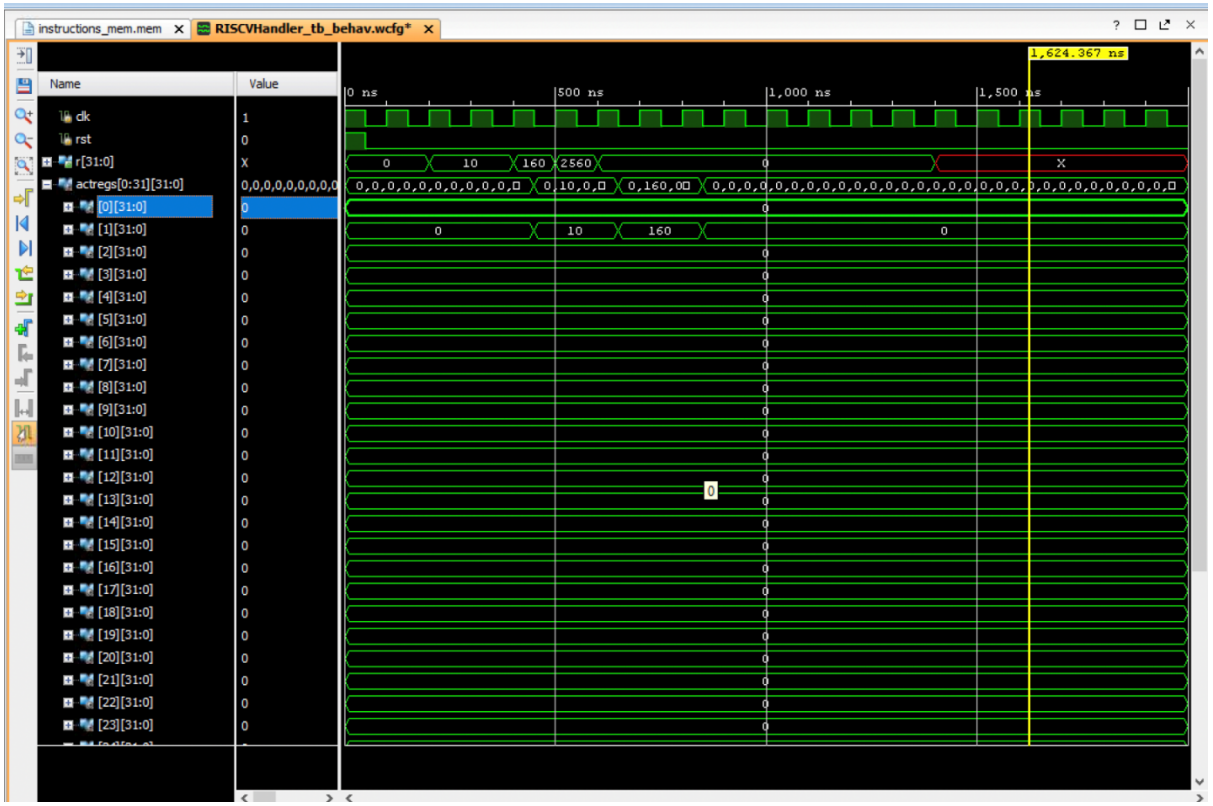
This is R-I.txt file test case.



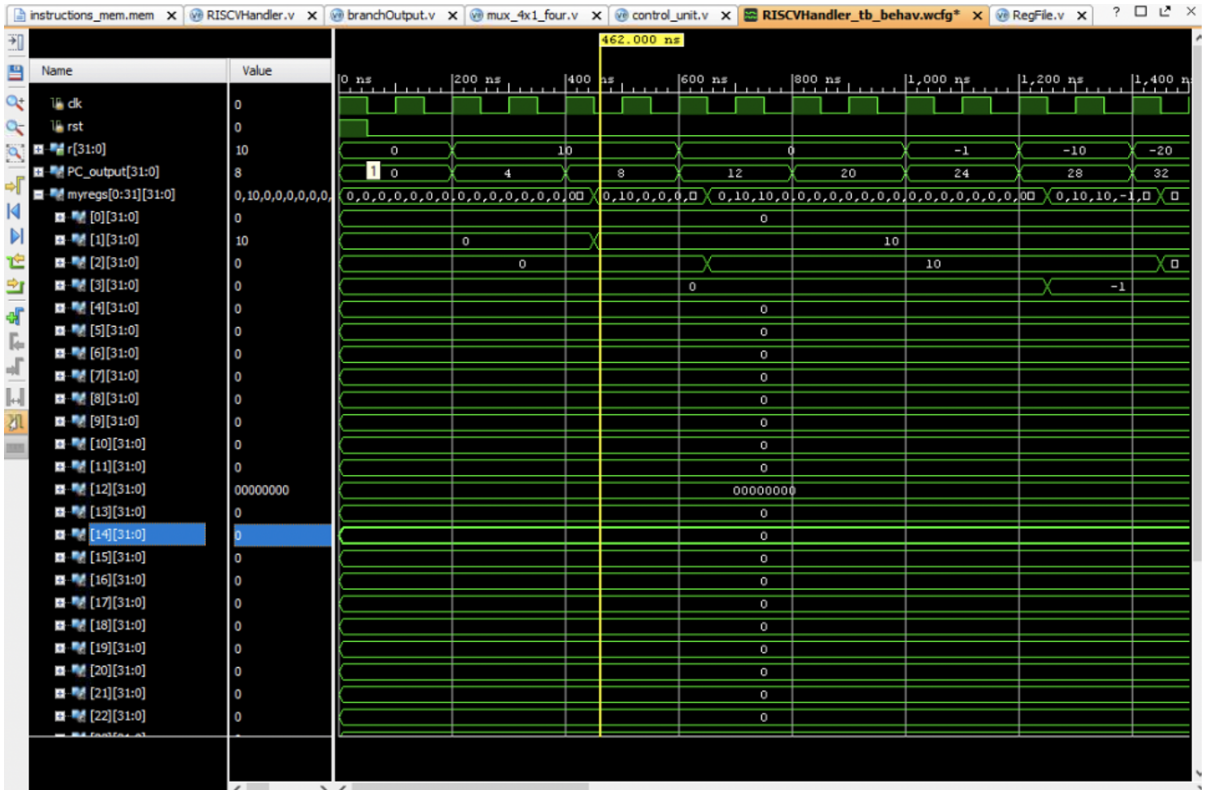
This is sets.txt file test case.



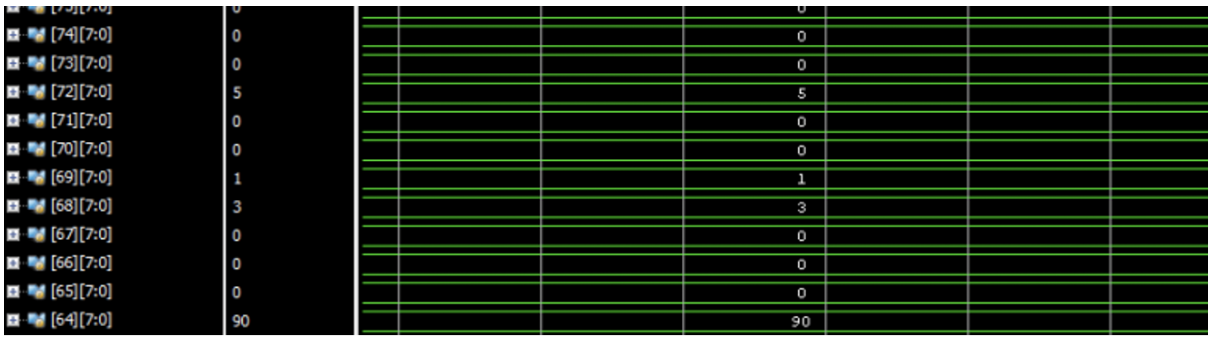
This is shift-I.txt file test case.



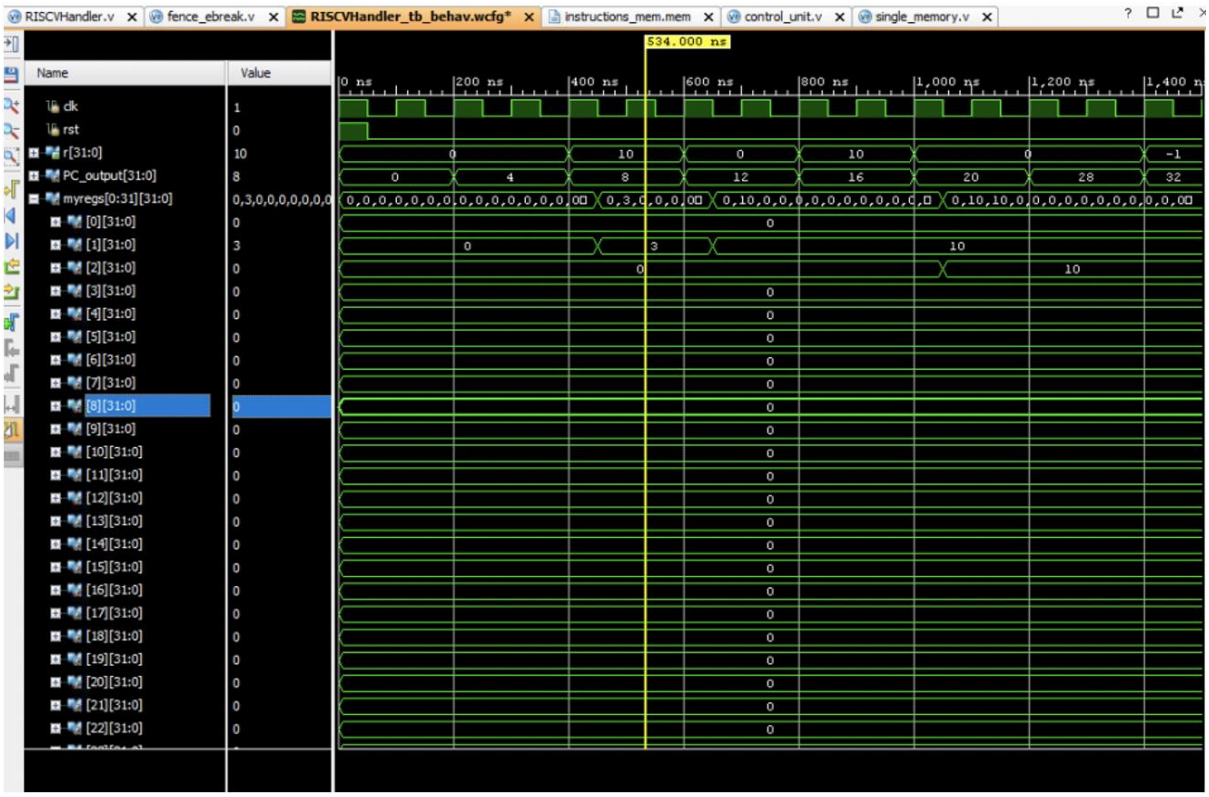
This is sample of branch.doc file test case.



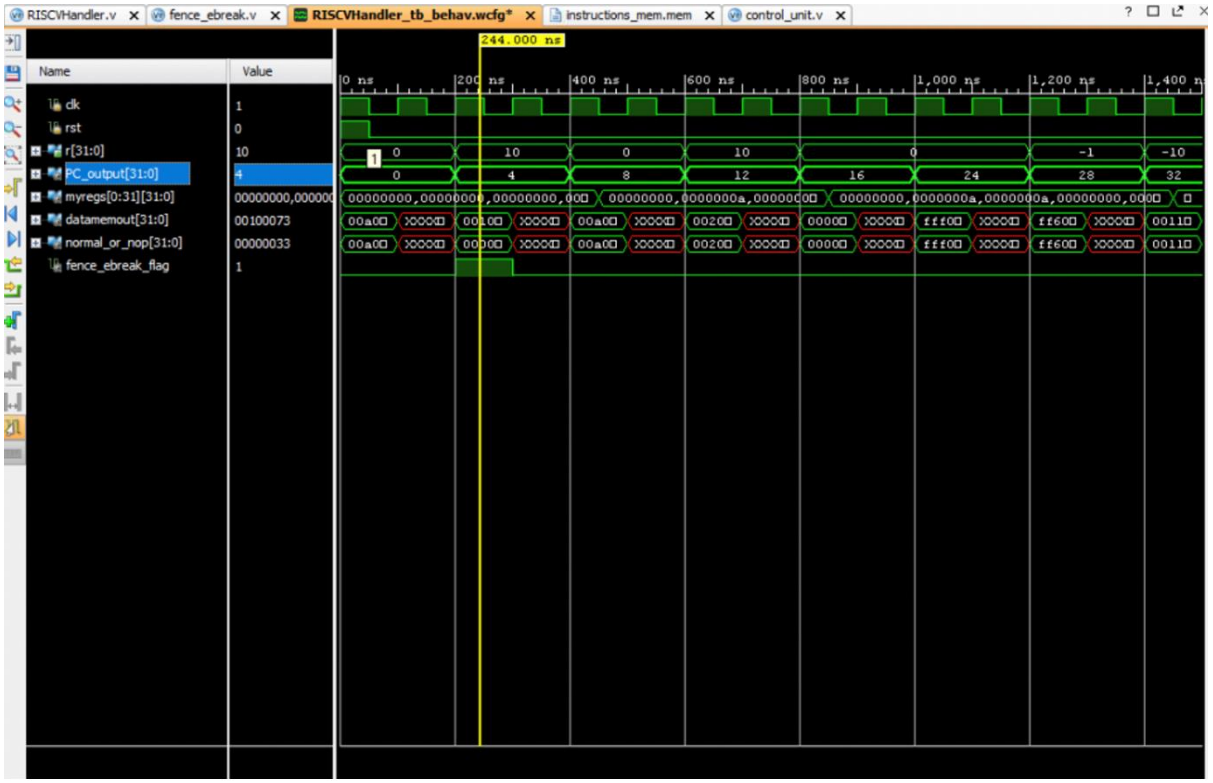
This is a sample of store.doc file test case.



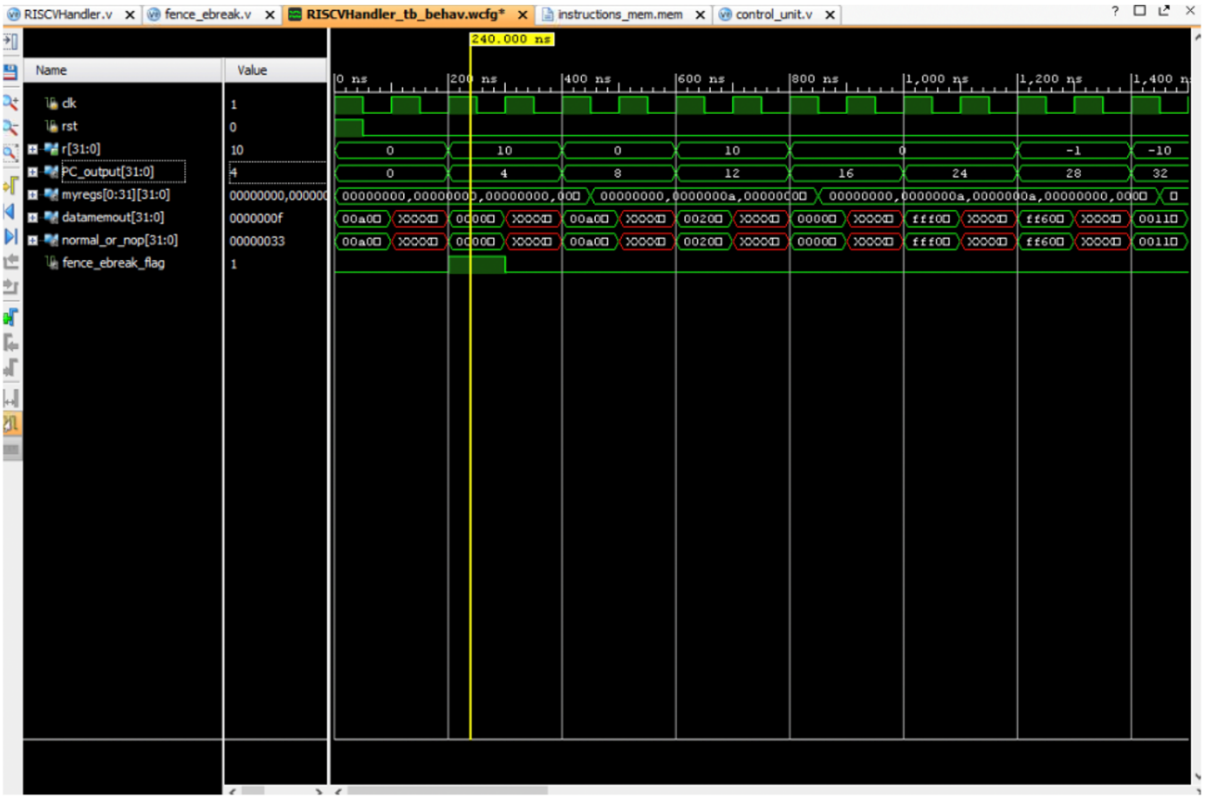
This is a sample of load.doc file test case.



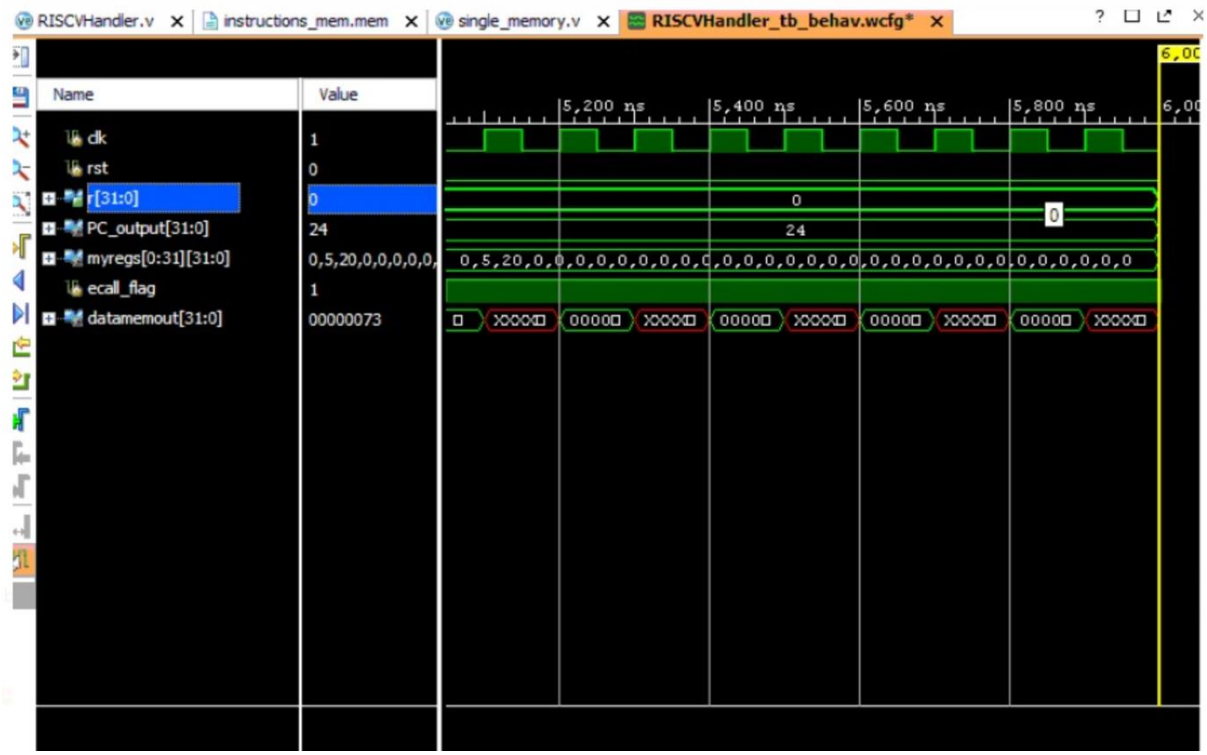
This is a sample of ebreak.doc file test case.



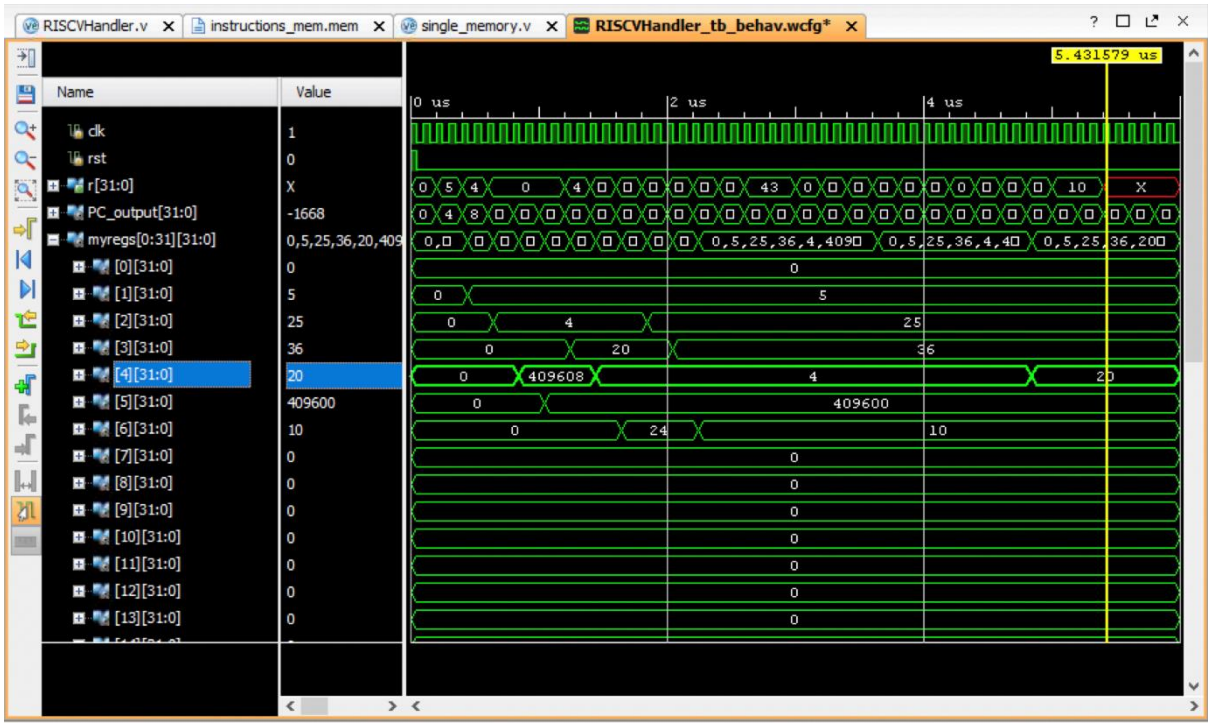
This is a sample of fence.doc file test case.



This is a sample of ecall.doc file test case.



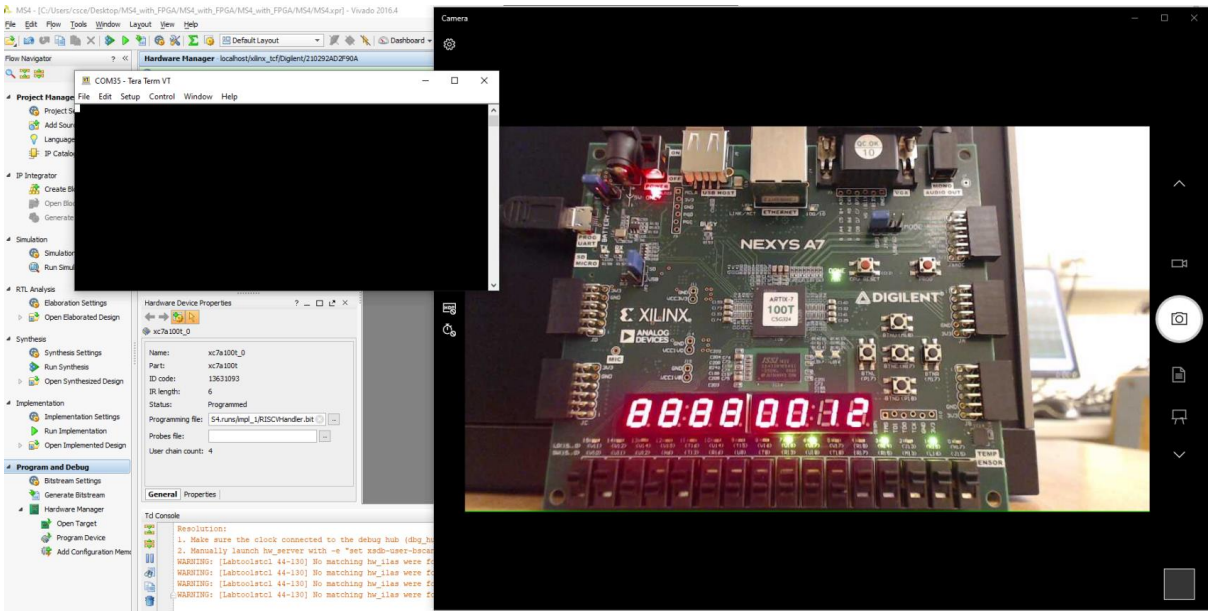
A combination of all instructions in one program:



All hexadecimal representation is in the testcase directory.

More advanced testing and screenshots are found in the testcases folder.

FPGA



This is the project working on the FPGA, however, the FPGA only displayed the PC Output and did not display any other values, we tried to write the instructions in the memory directly instead of reading from a file but it did not display other values as well.

4. DataPath

This is attached in a separate file.