THE AMERICAN UNIVERSITY IN CAIRO
الجامعة الأمريكية بالقاهرة

24/11/2020

Pipelined Implementation of RISC-V Processor

School of Science and Engineering

Submitted to Dr. Cherif Salama

Submitted by:

Salma Aly	900183330
Menna Elzahar	900182968
Mariam Mousa	900183871

CSCE330102 - Computer Architecture (2020 Fall)

Section 02

Technical Summary:

In this final stage of the project, the pipelined RISC-V processor is implemented with the 40 RV32I Base Instruction Set and all hazards handling units supported: forwarding units and hazard detection unit. In addition, our implementation uses a single byte-addressable single-ported memory for both instructions and data. A separate type of register for the PC was created to resolve the potential structural hazards introduced by this single memory. At first, the branch instructions were implemented so that the branching decision is computed in the MEM stage. Thus, the flushing components were supported. Then, the branch instructions were modified so that the branch decision is taken in the ID stage, and thus the flushing components were removed. Here is a brief technical summary of how each part was accomplished.

Single-ported Memory

A module named "memory" was created to represent both the instruction and data memories. It has the following parameters: input clk, input MemRead, input MemWrite, input [2:0] func3, input [11:0] addr, input [31:0] data_in, output reg [31:0] data_out. The memory was placed instead of the data memory's position in the processor design as illustrated in the data path section. The size of the memory is assumed to be 4KB. It is byte-addressable. The instructions used to be stored from byte 2048 (second half of the memory) while the data starts from byte 0. A 32-bit counter was created. If rst is 1, the counter is -1 so that the fetching of the first instruction is when the counter is 0. This counter is incremented with every positive clk cycle. As for the address entered to the memory, it either comes from the PC reg (instruction address) or ALU (data address). Thus, a 2-by-1 mux is created to either choose PC if counter[31] is 0 or ALU_result if counter[31] is 1. Two 2-by-1 multiplexers were created: one before the IF/ID reg (instrMux), and one before the MEM/WB reg (dataMux). Both have 2 inputs: zero and mem_out. Both have a 1-bit selection line that is the counter[31]. If the counter[31] value is even (0), dataMux chooses mem_out while instrMux chooses zero. If the counter[31] value is odd (1), dataMux chooses zero while instrMux chooses mem_out. This is how an instruction is read/fetched from the memory if the counter[31] is odd, and how the data is read if the counter[31] is even.

Structural Hazards Handling

To accomplish this, a mechanism was used to issue an instruction every two clock cycles. This is through letting the fetching stage's clock period equal double the period of the other stages' clock cycle. Therefore, a separate PC register was created inside which there is a clk divider that produces a new clk dedicated for the PC called PC_clk. The frequency of this PC_clk is half that of the normal clk. This is how there will never be an overlap between the IF and the MEM stage. A 32-bit signed counter was created which is initialized to 1 when the rst is 1 and is decremented by 1 with each negative edge of the PC_clk. Therefore, depending on the condition (counter>0)?in+508:in, the instructions are fetched correctly from the second half of the memory when needed.

R- type Instruction (add, sub, and, or)

These instructions were already supported in the old implementation of the pipeline we created in the lab. They were tested again after creating the new implementation, and they are working properly. The rest of the R-instructions (xor, sll, srl, sra, slt, sltu) were added also. They are implemented in the supported ALU and their ALU control signals were only adjusted to fit with our code.

I-type Instructions

A case for the I-type instructions was added in the control unit to set the ALU second input to the immediate instead of rs2. The immediate generator unit that was provided by the professor was integrated with the code. It takes as an input, the instruction, and based on it the immediate to output is decided.

Branch Instructions

Branch Instructions and flushing:

The branch instructions are detected by their opcode and the decision of branching is based on a flag signal we added in the module. This signal is computed based on the following flags (vf, sf, cf, and zf) inside the new comparator module in the ID stage.

In this processor, after modification the branch decision is computed in the ID stage. There are two new modules that we added which are the comparator and the branch adder input. The comparator takes the instruction operands and the opcode to output the flag signal used to define the branch decision. The branch adder input is used to compute the branch address by adding the pc value and the generated immediate. Given the flag signal and the instruction opcode a branching mux outputs the new corresponding pc value. As the branch is decided in the ID stage and our processor fetches one instruction every two clock cycles, there are no control hazards and no instruction is needed to be flushed. However, there are data hazards that are handled by adding another forwarding unit in the ID stage that forwards the required operands from preceding instructions in the EX or MEM stages.

Load Instructions

The load instructions were implemented by passing the func3 value to the memory to be used to choose which type of load to is needed through a case statement.

```
For LB, data_out = {{24{mem[addr][7]}},mem[addr]}.
For LH, data_out = {{16{mem[addr+1][7]}},mem[addr+1],mem[addr]}
For LW, data_out = {mem[addr+3], mem[addr+2], mem[addr+1], mem[addr]};
Foe LBU, data_out ={{24{1'b0}},mem[addr]};
For LHU, data_out ={{16{1'b0}},mem[addr+1],mem[addr]};
```

Store Instructions

These were implemented by adding a case statement inside the memory to choose which store instruction. In SW, the memory stores all the content of the data input. In SH, the memory stores the least significant 2 bytes of the input data. In SB, the memory stores only the least significant byte of the input data.

JAL&JALR

To implement jal and jalr, we used a signal called flag (also used to implement branch instructions). Flag equals one when the instruction is a jal or a jalr. To calculate the target branch address, jal and jalr use the same adder as branch instructions. An if statement is used inside the adder to decide which address will be output. Then, a mux is used to decide whether the branch address or the address of the next instruction will be the input to the PC based on the opcode and the flag signal.

Both jal and jalr write the value of the return address to a destination register using a mux (mux_RD) which decides which data to be written to the register file based on the Rdsrc signal which is one output of the control unit.

LUI

It is implemented using the immediate generator. The output of the immediate generator travels to the write back stage and the write back mux (mux_RD) decides which data to be written based on the Rdsrc signal.

AUIPC

To implement auipc, we used the branch adder, so the output of the immediate generator is added to the current PC.

Ebreak, Ecall, and Fence Instructions

The Ebreak and Fence instructions were implemented as nops by equating all their control signals to zero in the control unit, and were tested to be found as effective and working properly. The Ecall was implemented in the main processor module by an if condition that checks the opcode and the generated immediate and if satisfied the PC load signal is deserted.

Forwarding Unit

To implement the forwarding unit, we used the two signals forwardA and forwardB, such that forwardA decides which operand will be used for the first ALU input and forwardB will be used to decide which operand will be used for the second ALU input. We used the following conditions:

a. EX_MEM_RegWrite && (EX_MEM_RegisterRd != 0)&& (EX_MEM_RegisterRd == ID_EX_RegisterRs1)
   In this case, we need to forward the ALU result from the previous instruction.
b. (MEM_WB_RegWrite && (MEM_WB_RegisterRd !=0) && (MEM_WB_RegisterRd == ID_EX_RegisterRs1))&& !(EX_MEM_RegWrite && (EX_MEM_RegisterRd != 0)&&(EX_MEM_RegisterRd == ID_EX_RegisterRs1))
   In this case, we need to forward either the ALU result or the data read from the memory from a prior (older?) instruction.
c. Other than that forwardA and forwardB = 00, so no forwarding happens.

Stall Unit (Hazard Detection Unit)

The hazard detection unit is still supported although it is no longer needed since a new instruction is fetched every two clock cycles now instead of one. Therefore, the load use hazard is no longer encountered since the output of the load instruction is available in the MEM stage and can then be forwarded to the execution stage of the using R-instruction. This is illustrated in figure 1.

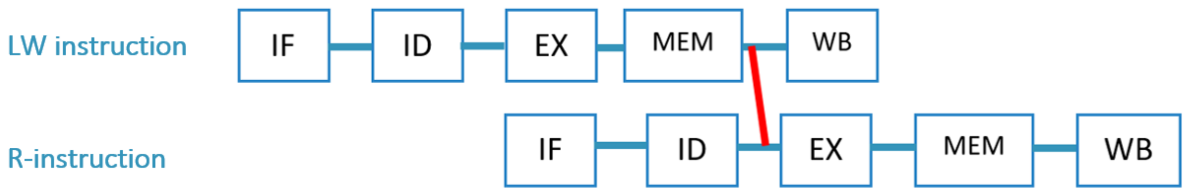
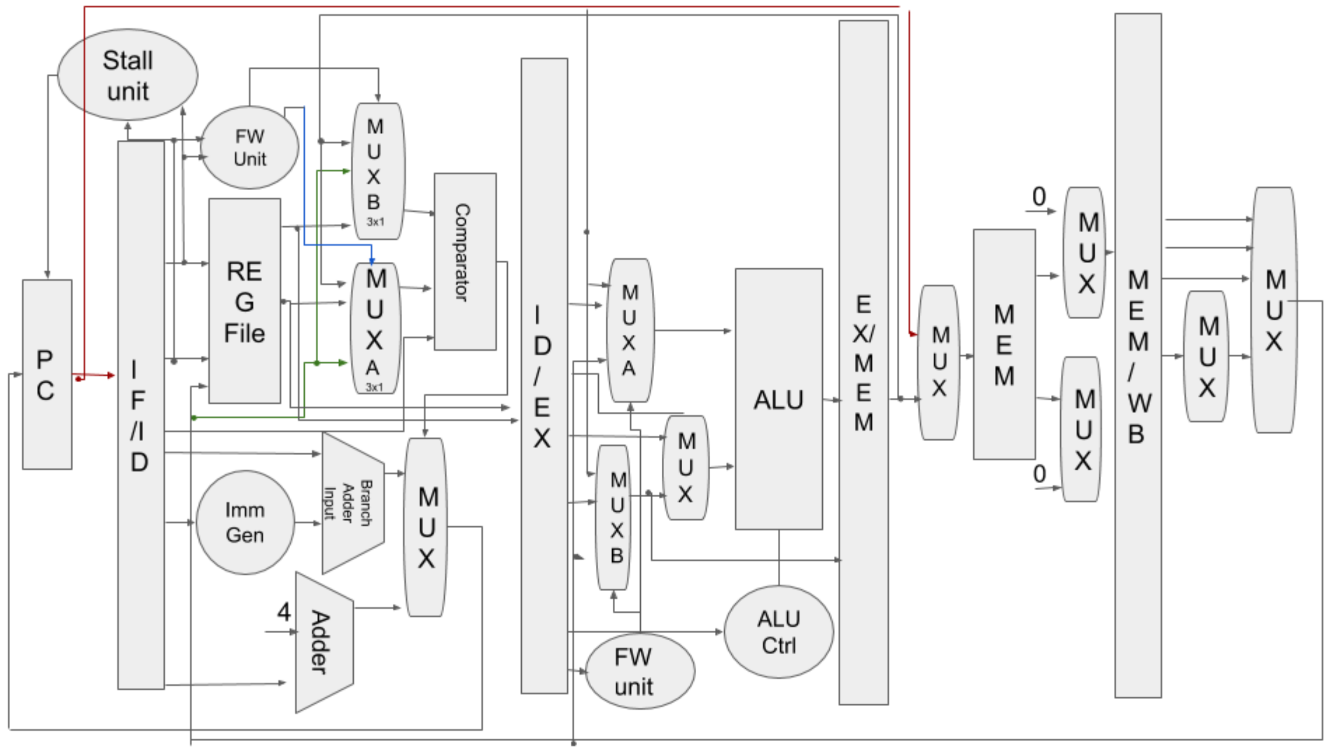


Figure 1: Load-use hazard resolved.

There was not just time to remove it and we believed this is OK since it has no negative impact on the performance of the processor.

New Data Path

Some wires are colored (blue, red and green) for the purpose of clarity.

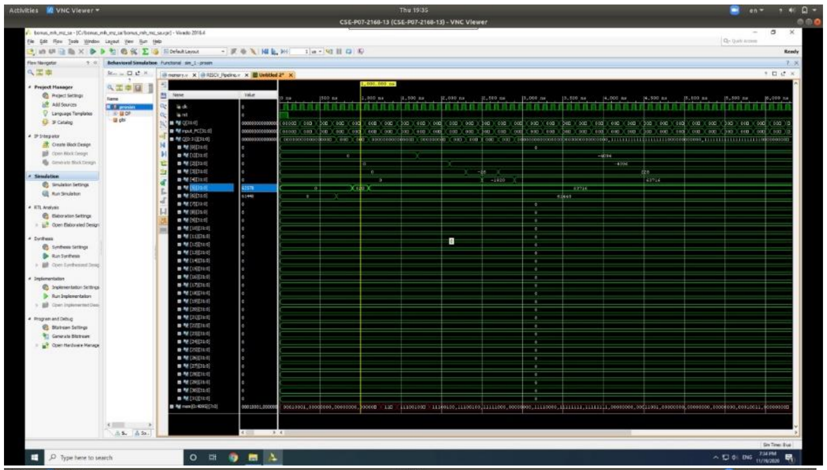


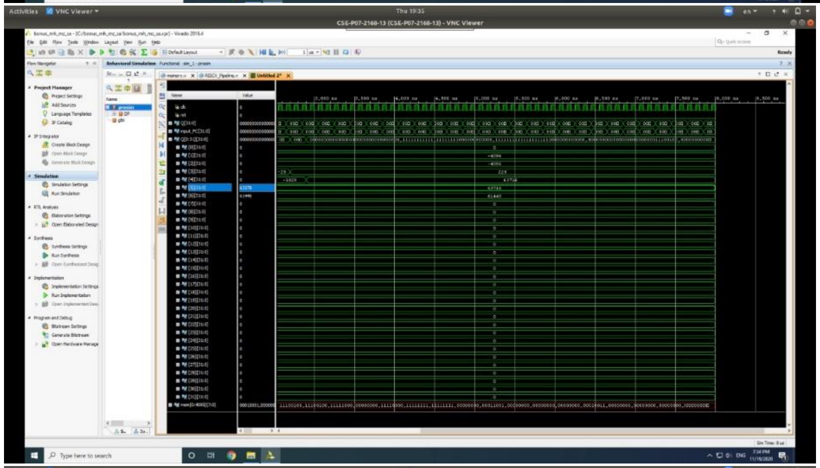
Results

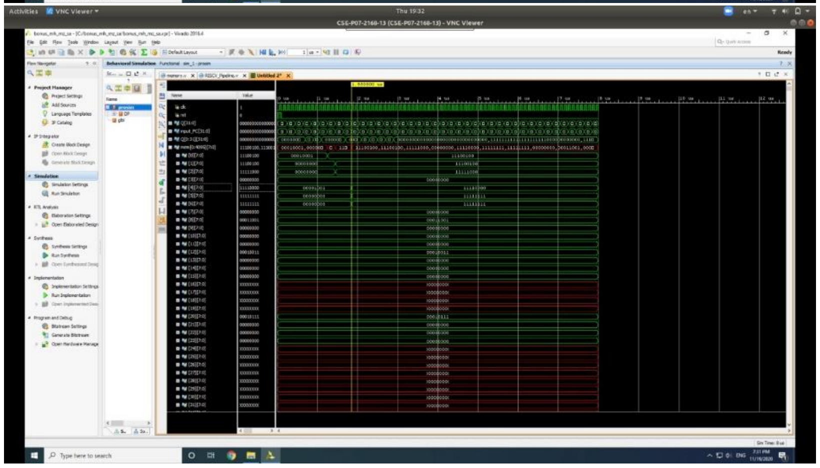
Six test programs were created to test all of the 40 instructions supported in the processor including a test file dedicated for testing all possible cases of any potential hazards. The results of each test program are provided in a separate excel sheet. The FPGA is not working well. We tried testing the code on the FPGA several times. Synthesis and implementation steps are done successfully, but the bitstream is not generated. There is an error. The memory size is changed to 1kB instead of 4 KB while testing on the FPGA only.

The following are the screenshots of the simulation for each of the tested program.

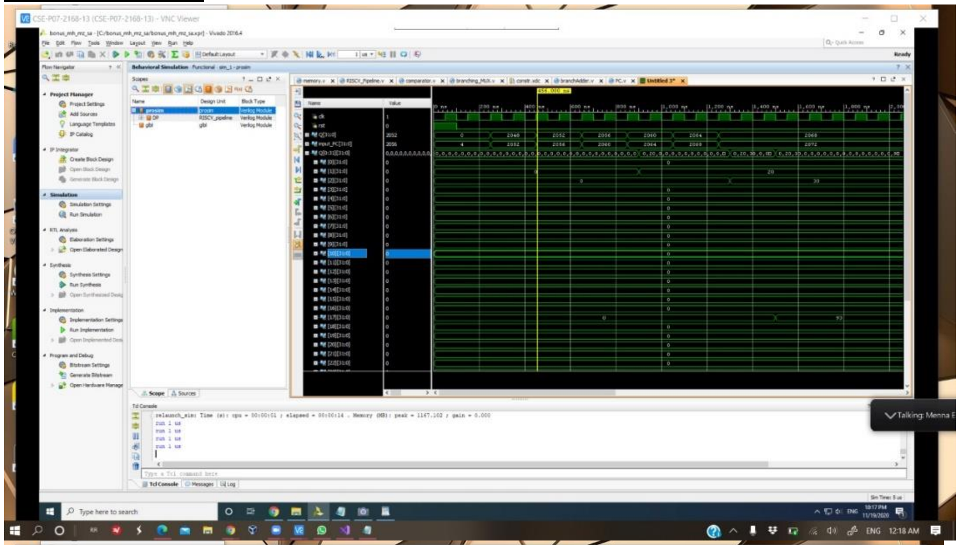
loads_stores



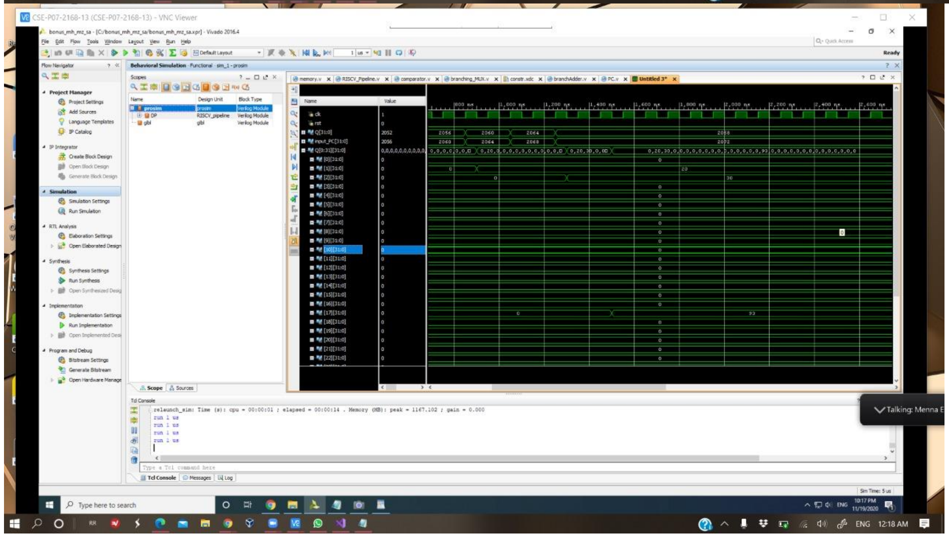




EcallEbeakFence

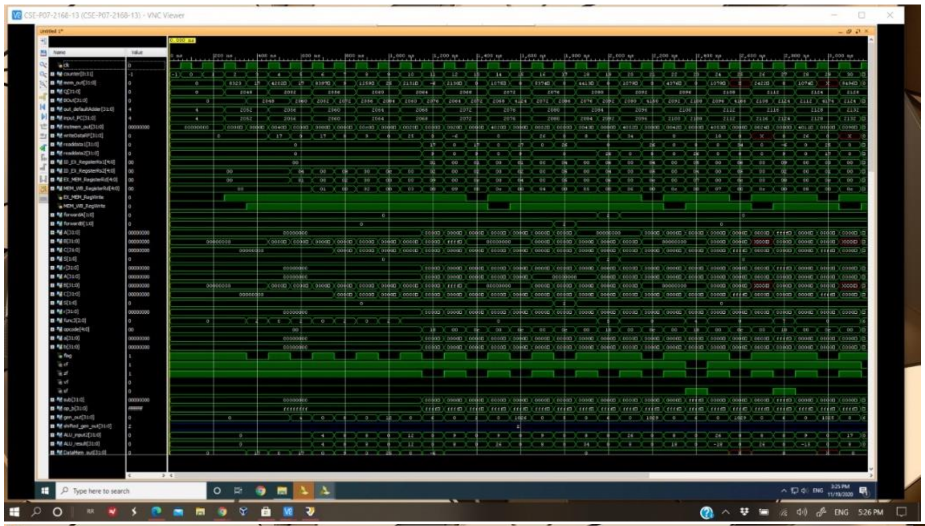


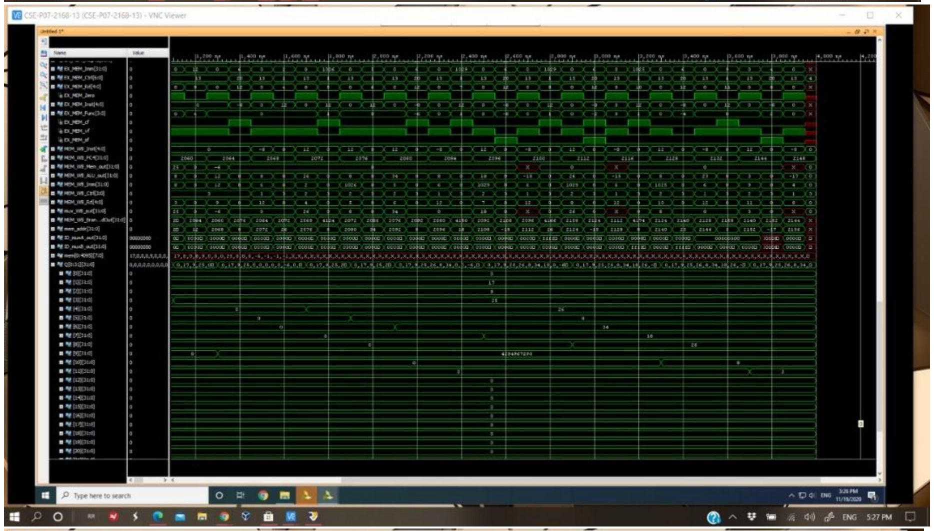
```
relaunch_sim: Time (s): cpu = 00:00:01 ; elapsed = 00:00:14 . Memory (MB): peak = 1167.102 ; gain = 0.000
run 1 us
run 1 us
run 1 us
run 1 us
```

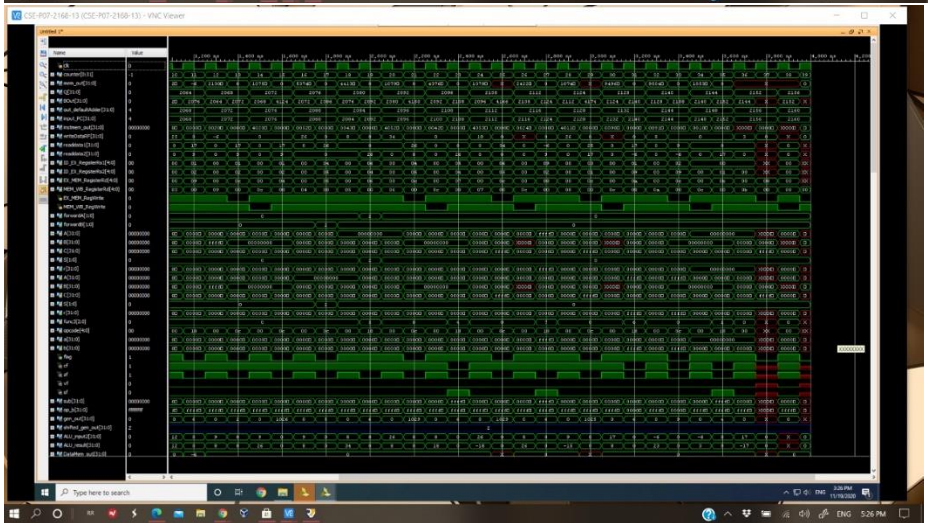


```
relaunch_sim: Time (s): cpu = 00:00:01 ; elapsed = 00:00:14 . Memory (MB): peak = 1167.102 ; gain = 0.000
run 1 us
run 1 us
run 1 us
run 1 us
```

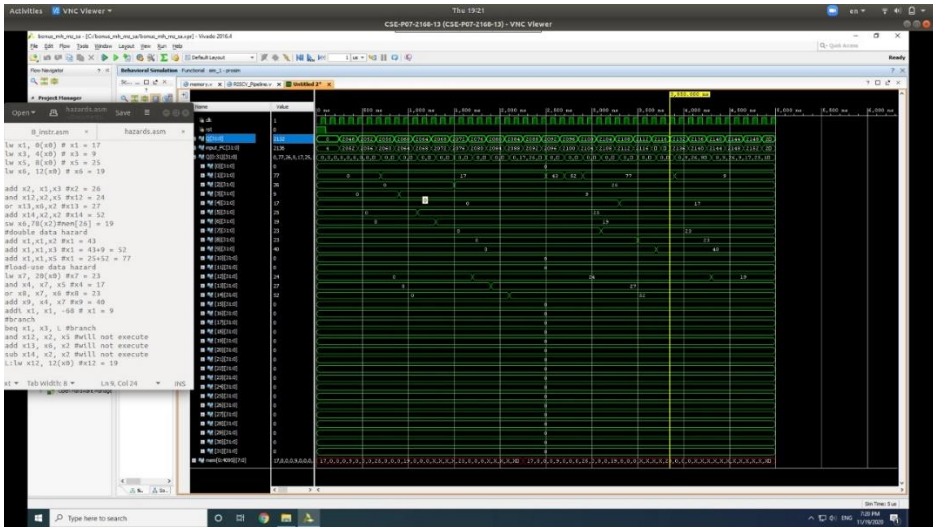
B_instr





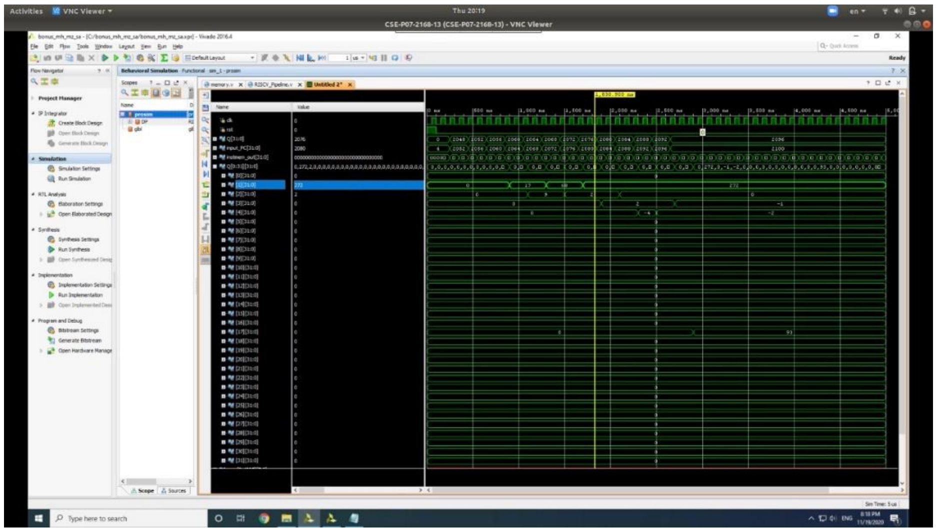


hazards

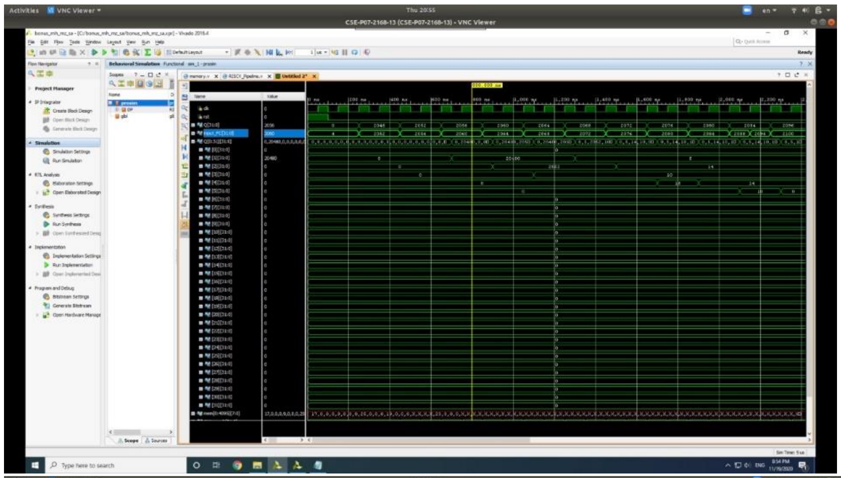


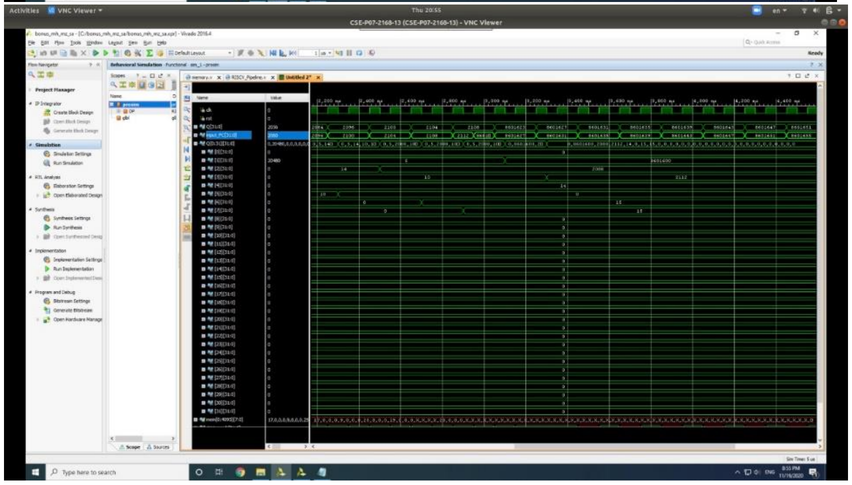
```
lw x1, 0(x0) # x1 = 17
lw x3, 4(x0) # x3 = 9
lw x5, 8(x0) # x5 = 25
lw x6, 12(x0) # x6 = 19

add x2, x1,x3 #x2 = 26
and x12,x2,x5 #x12 = 24
or x13,x6,x2 #x13 = 27
add x14,x2,x2 #x14 = 52
sw x6,78(x2)#mem[26] = 19
#double data hazard
add x1,x1,x2 #x1 = 43
add x1,x1,x3 #x1 = 43+9 = 52
add x1,x1,x5 #x1 = 25+52 = 77
#load-use data hazard
lw x7, 20(x0) #x7 = 23
and x4, x7, x5 #x4 = 17
or x8, x7, x6 #x8 = 23
add x9, x4, x7 #x9 = 40
addi x1, x1, -68 # x1 = 9
#branch
beq x1, x3, L #branch
and x12, x2, x5 #will not execute
add x13, x6, x2 #will not execute
sub x14, x2, x2 #will not execute
L:lw x12, 12(x0) #x12 = 19
```

shifttest



auipc_jal_jalr_lui





X