6-stage Pipelined RISC Processor

Syomantak Chaudhuri 170070004
Shubham Anand Jain 17D070010
Titas Chakraborty 170070019
Sanjoli Narang 17D100013

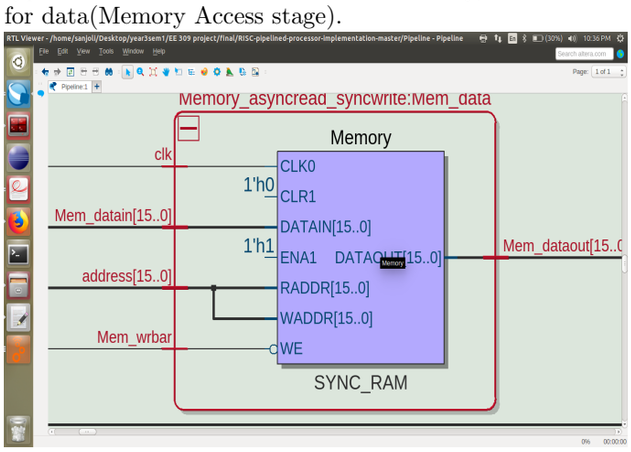
October 2019

1 Abstract

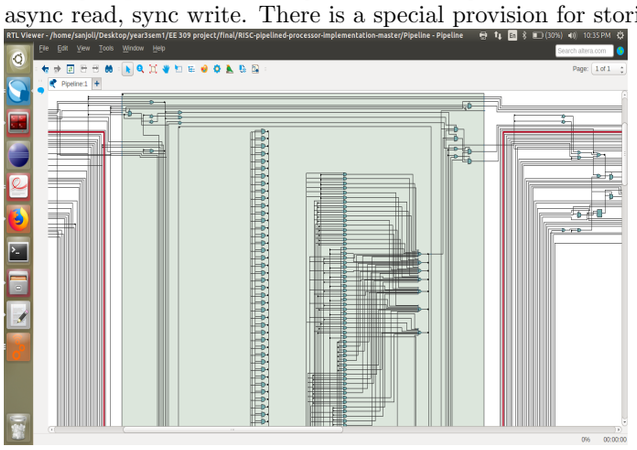
In this course project, we implement a 6-stage pipelined RISC based 8-register, 16 bit microprocessor, the six stages being Instruction Fetch, Instruction Decode, Register Read, Execution Tasks, Memory access and Write Back to registers.The pipeline is supported with Interface registers at the interface between consecutive stages to implement three lines of forwarding to take care of inter-instruction dependencies. There is another Hazard detection and mitigation logic that detects such dependencies and conditional branches and does necessary tasks. The ISA given has 16 instructions including arithmetic, logical and branch operations.

2 Resources and Components

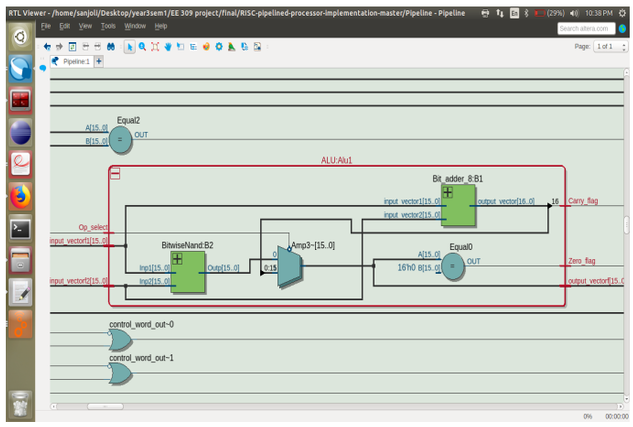
- **Memory :** The memory instantiated is async read, sync write 16-bit address and data bus type. There are two instances of the memory, one for instruction(Fetch stage) and the other one for data(Memory Access stage).



- **Register File :** There is only one instance of register file consisting of 8 16-bit registers that is being implemented in Register Read stage. The RF is two read and two-write port based, async read, sync write. There is a special provision for storing current PC into R7.



- **ALU :** There is one instance of ALU in the execution stage that computes both data and addresses, flags and jump conditons. There is an instance of a 16-bit adder also in Decode stage to compute jump address beforehand to save cycles.

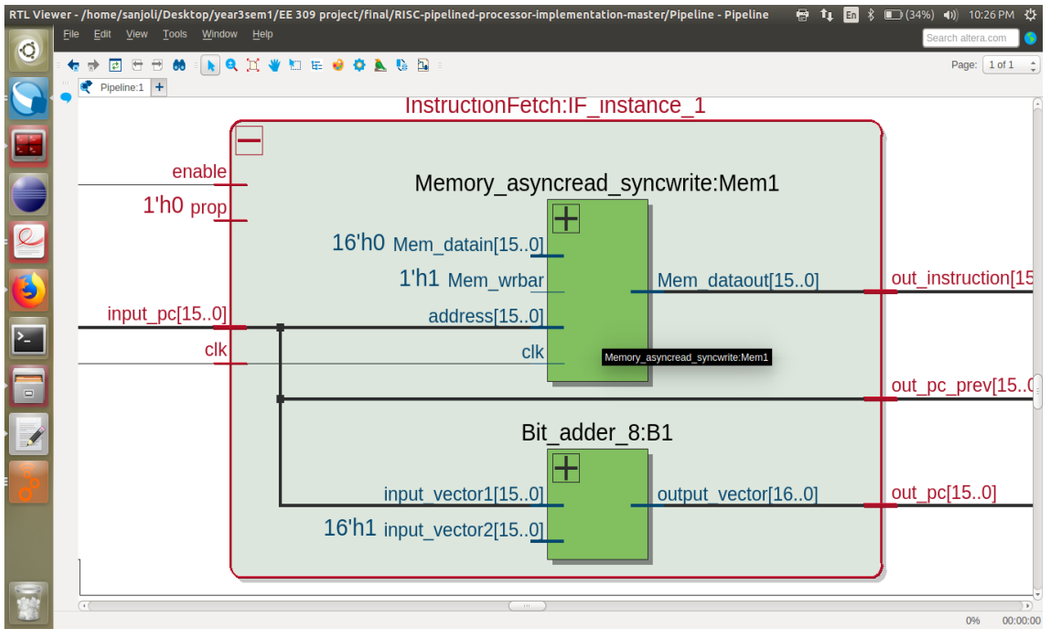


3 Detailed design of each component

3.1 Instruction Fetch

This stage does the following tasks :

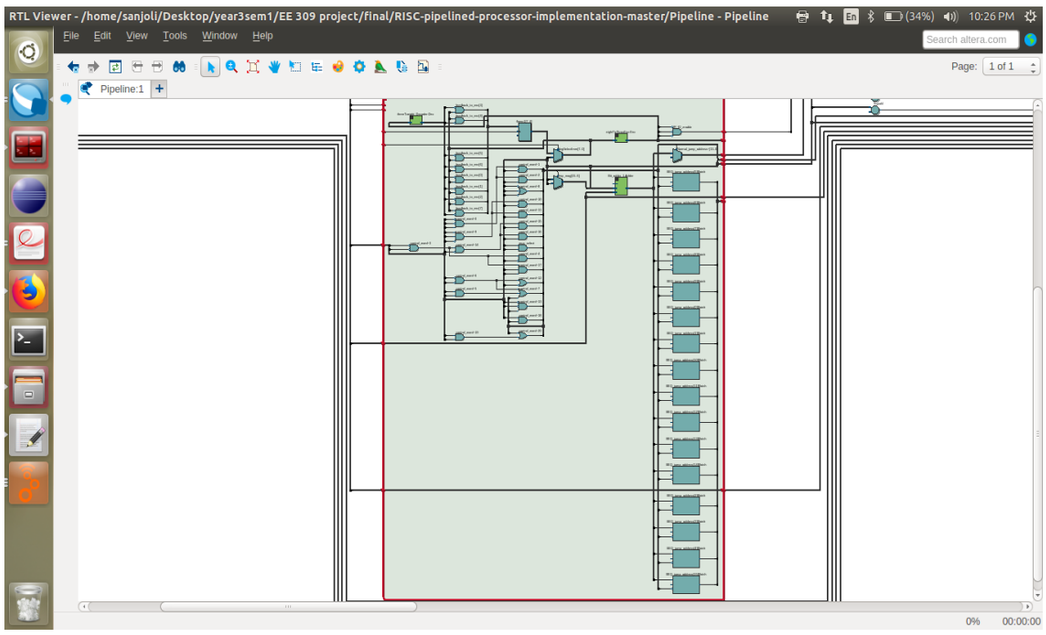
- Sending instruction address stored in PC to Instruction Memory, to acquire instruction in the Instruction register.
- Updating the PC depending upon what is the status of the top level of the design, i.e. whether there is an unconditional jump or BEQ, the control being done using a multiplexer controlled by top level.
- Passing on the required data, i.e. PC, IR to the IF/ID interface register



3.2 Instruction Decode

This stage does the following tasks :

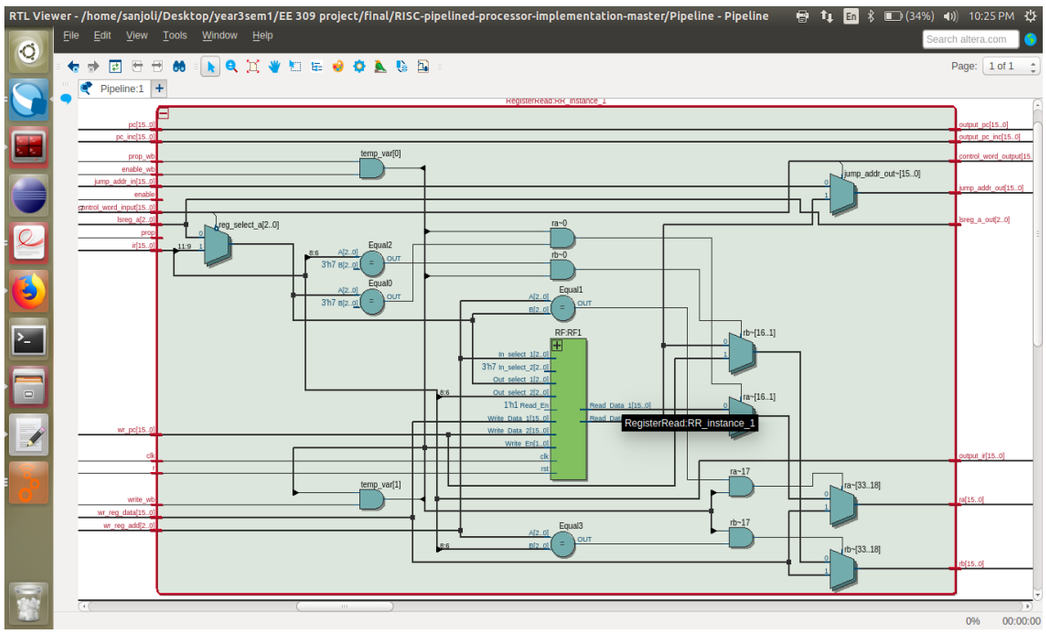
- The most important task is to generate the **control word** for each instruction minimising the logic as well. The control word is then used extensively in Execution stage.
- The stage extracts the Instruction register from the interface and forwards the relevant data like PC, incremented PC, etc to the next ID/RR interface
- There is a 16-bit adder that computes the jump address for both conditional and unconditional branches and passes the values to the control mux in the Fetch stage.
- The Decode stage also breaks the LM/SM instruction into smaller ones extracting the register addresses and passing them one by one to the subsequent stages using a priority encoder and a decoder. The control for this is made at Top Level



3.3 Register Read

This stage does the following tasks :

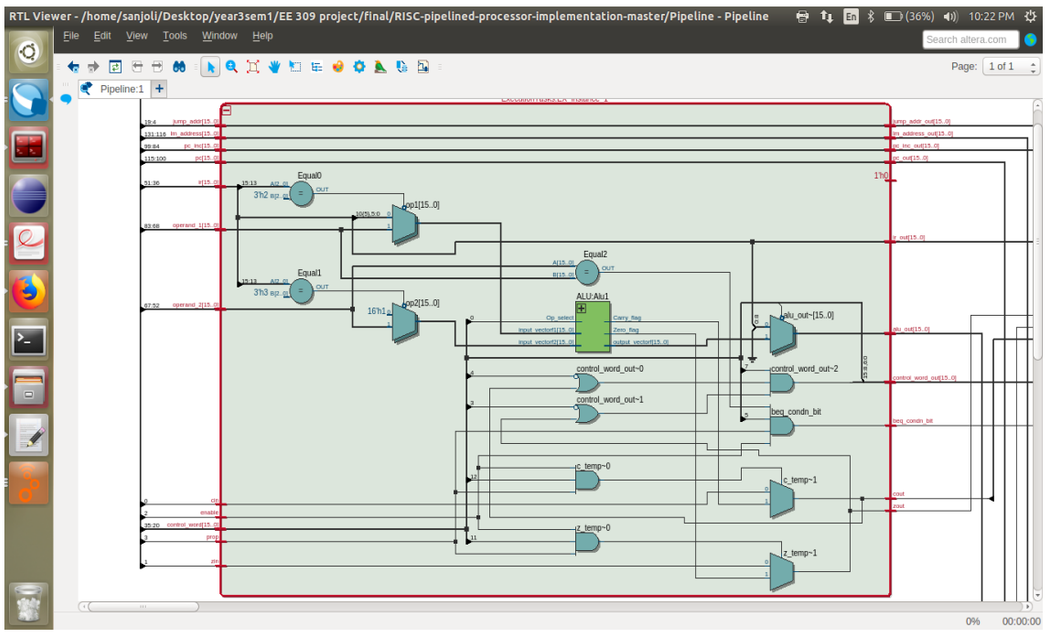
- The stage gets input from Decode stage and accordingly input the addresses of registers to the RF, the address can be either the one from IR or can be that computed for LM/SM in decode stage depending on mux control from top level. RF is instantiated here.
- This stage also receives the data to be written to RF from the Write Back stage and PC as well, and on clock transition writes on both register and R7 respectively.
- It also propagates the required data like control words or PC to the Execution stage.



3.4 Execution tasks

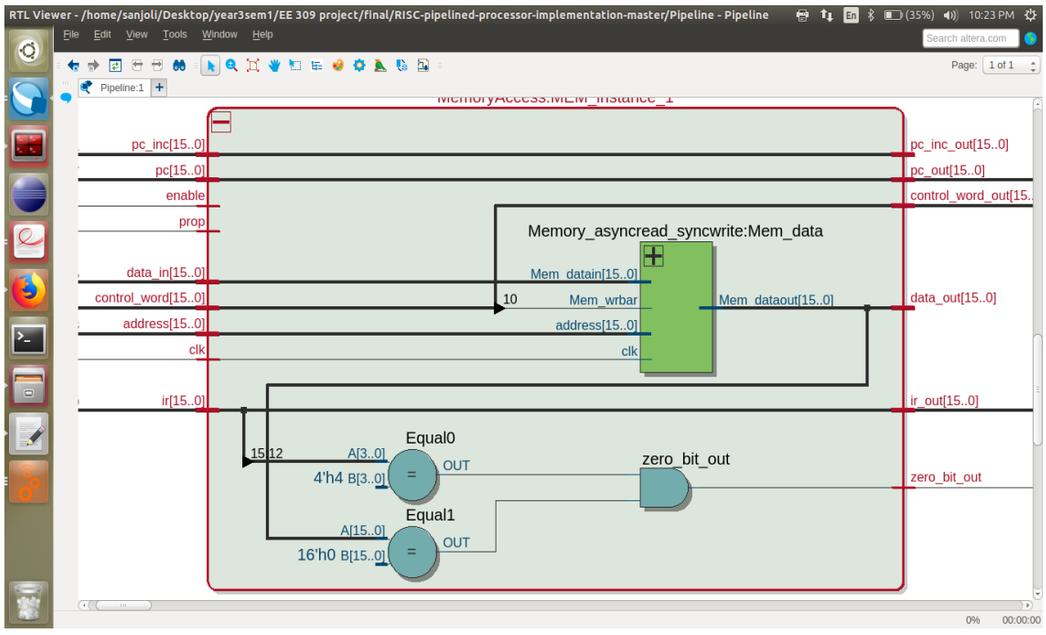
This stage does the following tasks :

- The stage computes arithmetic and logical operations, modifies the flags and the control word accordingly. It also computes the address of the memory to perform load and store instructions, It does constant and immediate additions as well
- The interface before Execution has feedback data from subsequent stages, the hazard logic controls which operands to input to the execution stage.
- This stage checks for branches, forwards the modified data to the next stages and generates control signals for top level.



3.5 Memory Access

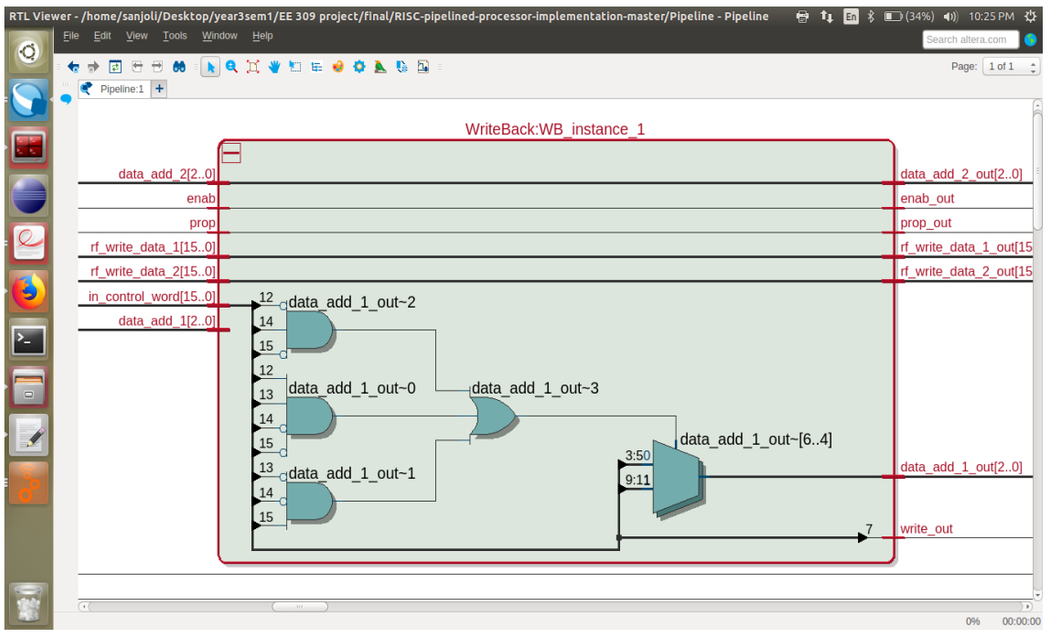
- This stage is used for all data memory accesses for all the load and store instructions. This also forwards the control word and other useful data to the write back stage.
- The data memory is synchronous write type, Data memory is instantiated in this stage.



3.6 Write Back

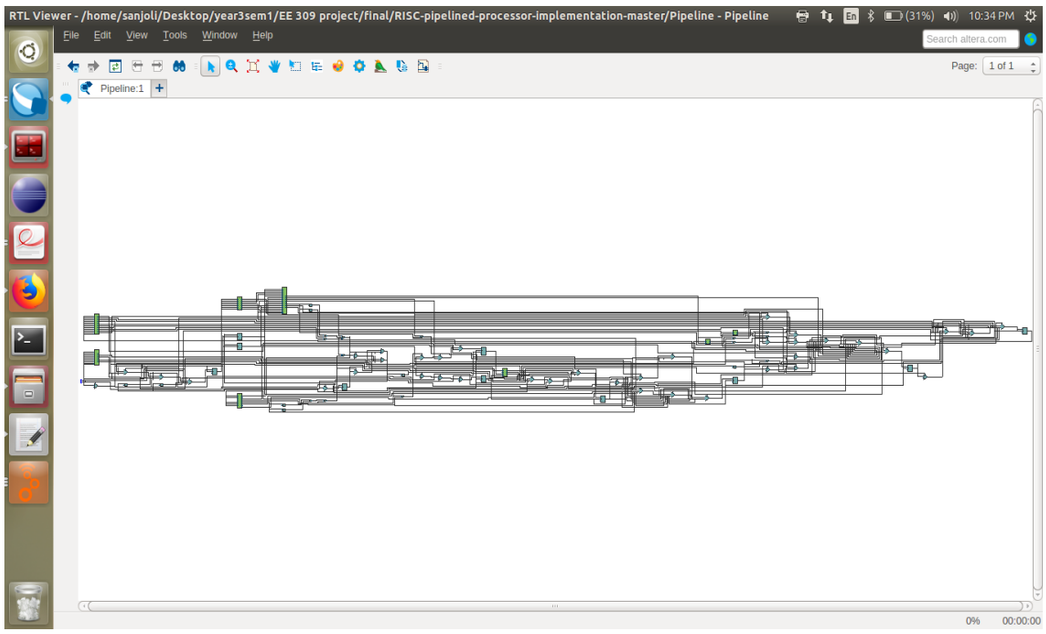
This stage does the following tasks :

- This stage generates control signal for writing to the RF in the register read stage. It also generates Address in RF to be written.
- This also decides whether to write output to the RF depending on the flags set in the execution stage.



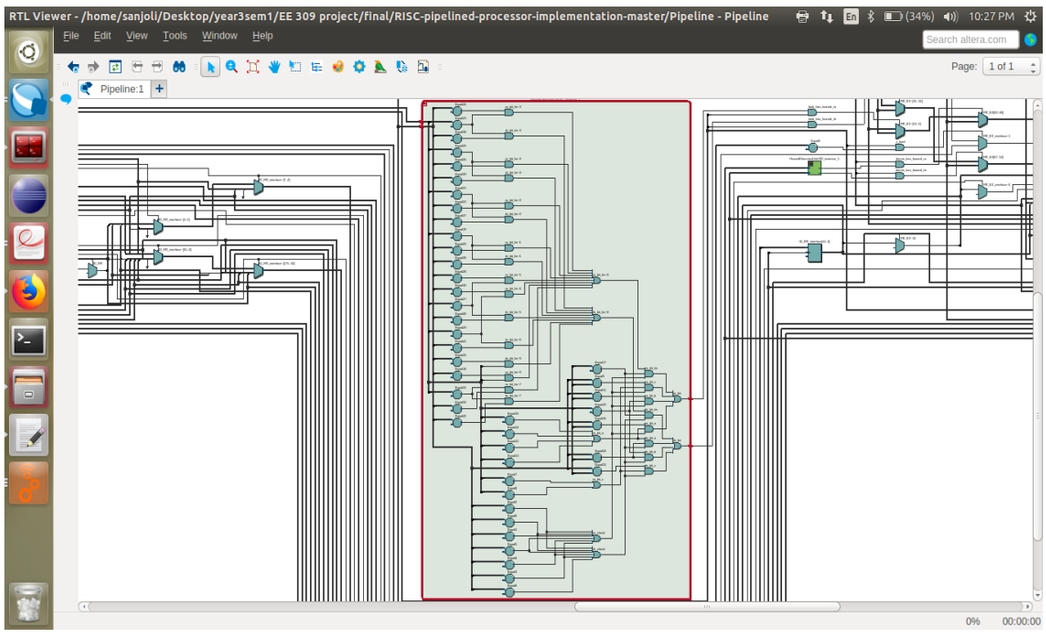
3.7 Top Level

This is the main frame of the entire design where all the stages of the pipeline are instantiated along with all the interface registers, forwarding paths, hazard and mitigation logic, and the FSM for deciding all the control signals, using the control word Store.



4 Hazard detection and mitigation

Hazard detection Logic is a combinational block that takes as input two instruction registers data values and finds out the dependency among the registers. The top level module takes the dependency and decidess which forwarding path to enable, how many cycles to stall.



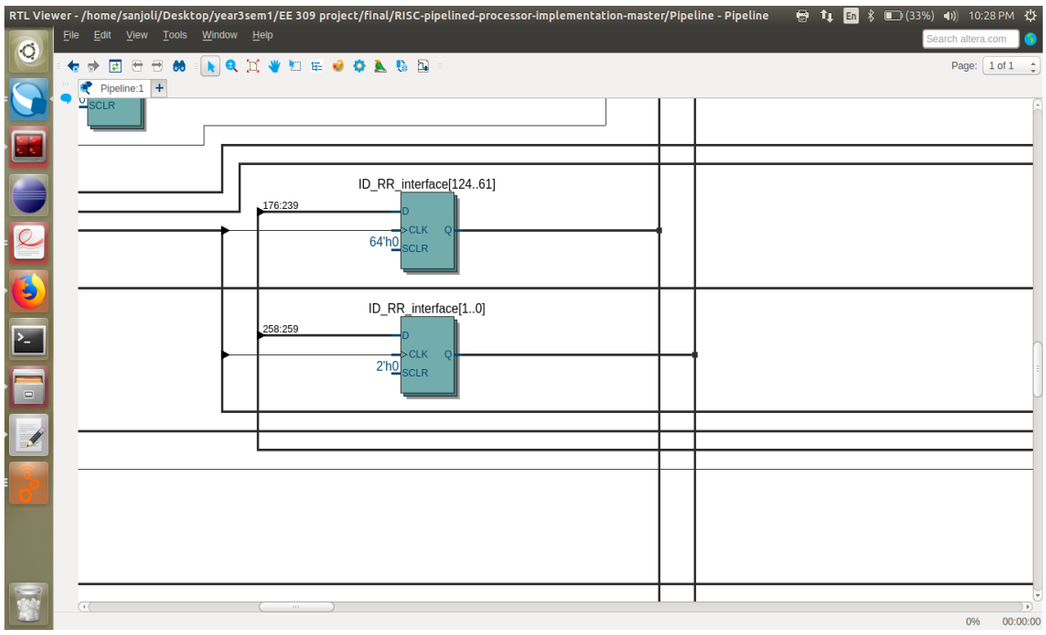
5 Forwarding Logic

1. The purpose of this unit is to eliminate the reduction in CPI due to dependencies. It decides what data to use in the ALU of the execution stage based on the instruction, the register values and the memory values.

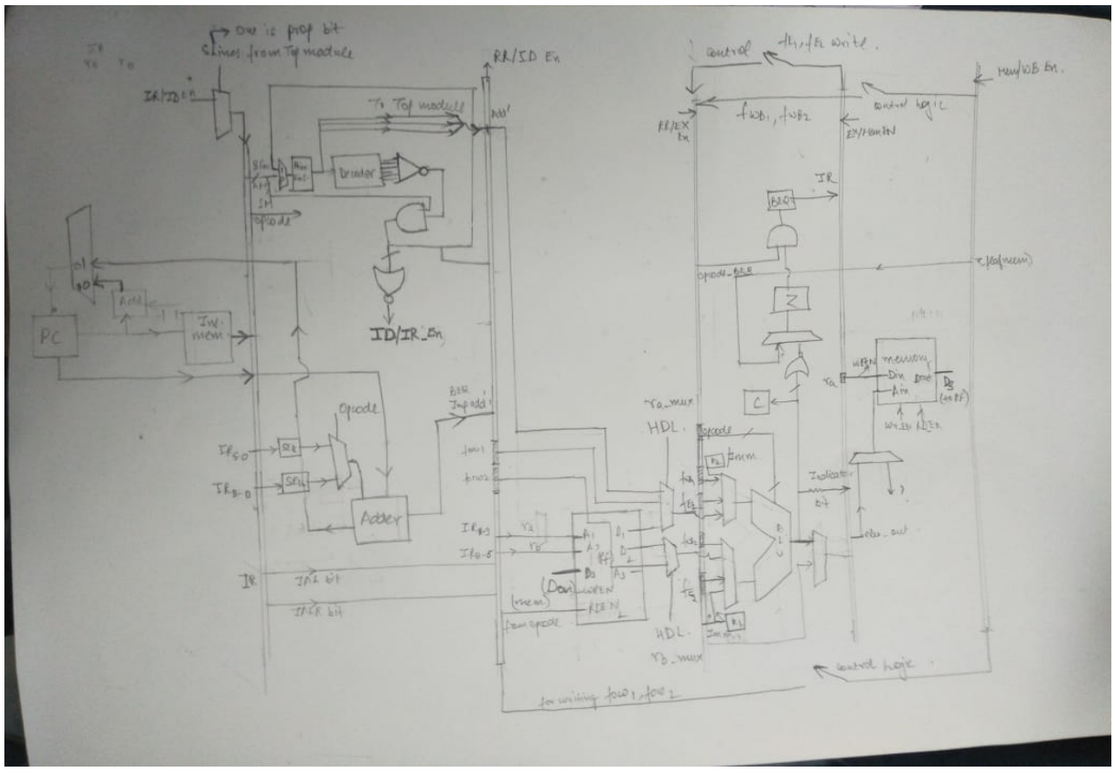
2. The logic for this stage is as follows: if data dependency is found in instruction (for which a bit is set in hazard detection logic), we set the mux to enable this forwarded data. Else, we set the mux to consider normal data.

3. To note is that, for load instruction, we need to stall for one clock cycle and then set the enable bit.

4. There are two forwarding paths from Execution stage end to the execution beginning state, and from memory stage to the execution stage.

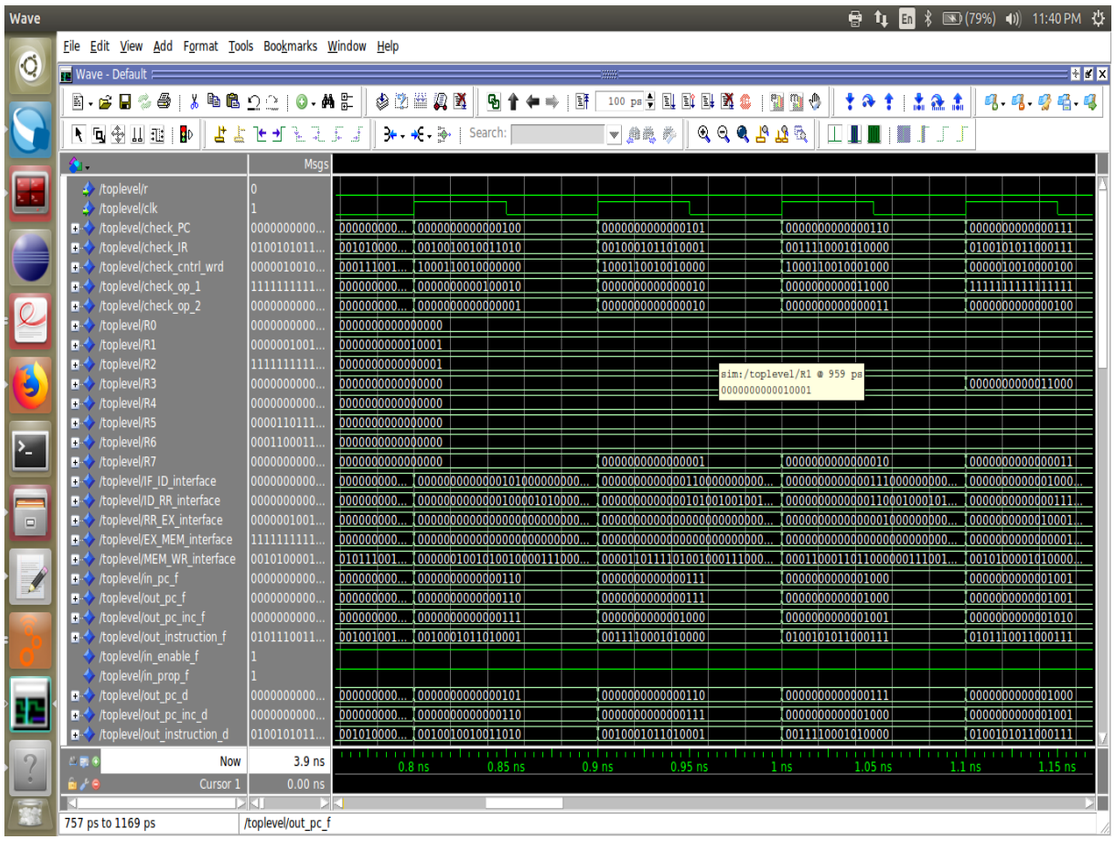


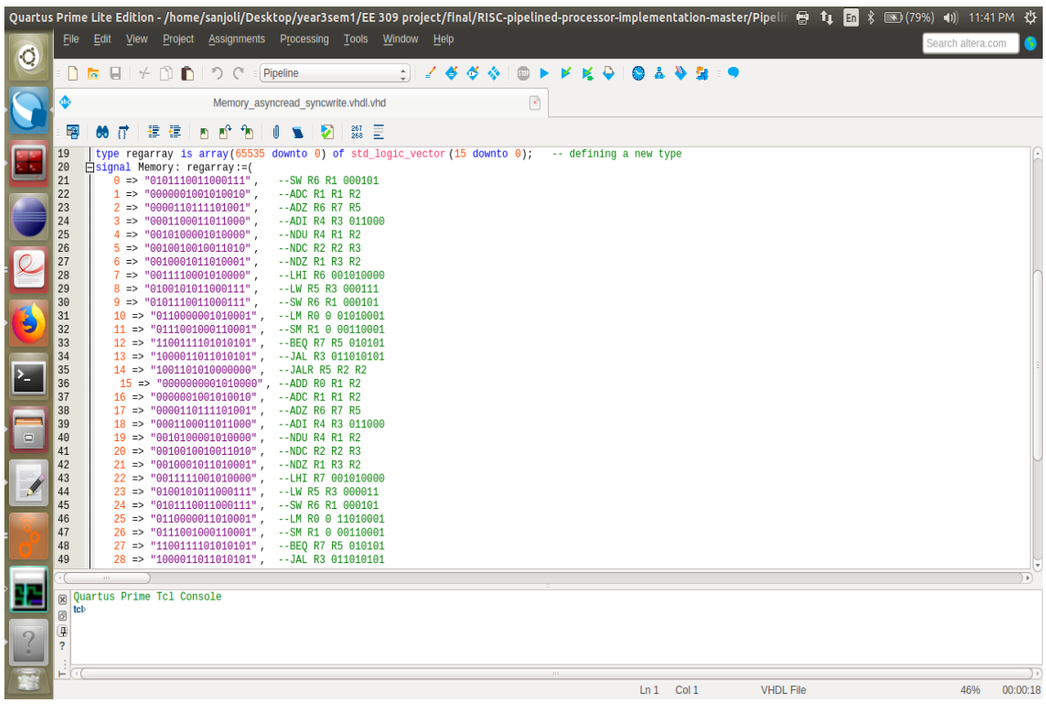
6 Final Datapath



7 Simulation results

Following is the simulation run of the instructions hard coded in memory




Quartus Prime Lite Edition - /home/sanjoli/Desktop/year3sem1/EE 309 project/final/RISC-pipelined-processor-implementation-master/Pipeli
File Edit View Project Assignments Processing Tools Window Help
Search altera.com
Pipeline
Memory_asyncread_syncwrite.vhdl.vhd
type regarray is array(65535 downto 0) of std_logic_vector(15 downto 0); -- defining a new type
signal Memory: regarray:=(
0 => "0101110011000111", --SW R6 R1 000101
1 => "0000001001010010", --ADC R1 R1 R2
2 => "0000110111101001", --ADZ R6 R7 R5
3 => "0001100011011000", --ADI R4 R3 011000
4 => "0010100001010000", --NDU R4 R1 R2
5 => "0010010010011010", --NDC R2 R2 R3
6 => "0010001011010001", --NDZ R1 R3 R2
7 => "0011110001010000", --LHI R6 001010000
8 => "0100101011000111", --LW R5 R3 000111
9 => "0101110011000111", --SW R6 R1 000101
10 => "0110000001010001", --LM R0 0 01010001
11 => "0111001000110001", --SM R1 0 00110001
12 => "1100111101010101", --BEQ R7 R5 010101
13 => "1000011011010101", --JAL R3 011010101
14 => "1001101010000000", --JALR R5 R2 R2
15 => "0000000001010000", --ADD R0 R1 R2
16 => "0000001001010010", --ADC R1 R1 R2
17 => "0000110111101001", --ADZ R6 R7 R5
18 => "0001100011011000", --ADI R4 R3 011000
19 => "0010100001010000", --NDU R4 R1 R2
20 => "0010010010011010", --NDC R2 R2 R3
21 => "0010001011010001", --NDZ R1 R3 R2
22 => "0011111001010000", --LHI R7 001010000
23 => "0100101011000111", --LW R5 R3 000011
24 => "0101110011000111", --SW R6 R1 000101
25 => "0110000011010001", --LM R0 0 11010001
26 => "0111001000110001", --SM R1 0 00110001
27 => "1100111101010101", --BEQ R7 R5 010101
28 => "1000011011010101", --JAL R3 011010101
Quartus Prime Tcl Console
tcl>
Ln 1 Col 1 VHDL File 46% 00:00:18