[CprE 381] Computer Organization and Assembly-Level Programming, Fall 2018

Project B - Report

Name(s) Ben and Bryan_____

Section/Lab Time _____F_____

Refer to the highlighted language in the <u>Project B</u> instruction for the context of the following questions.

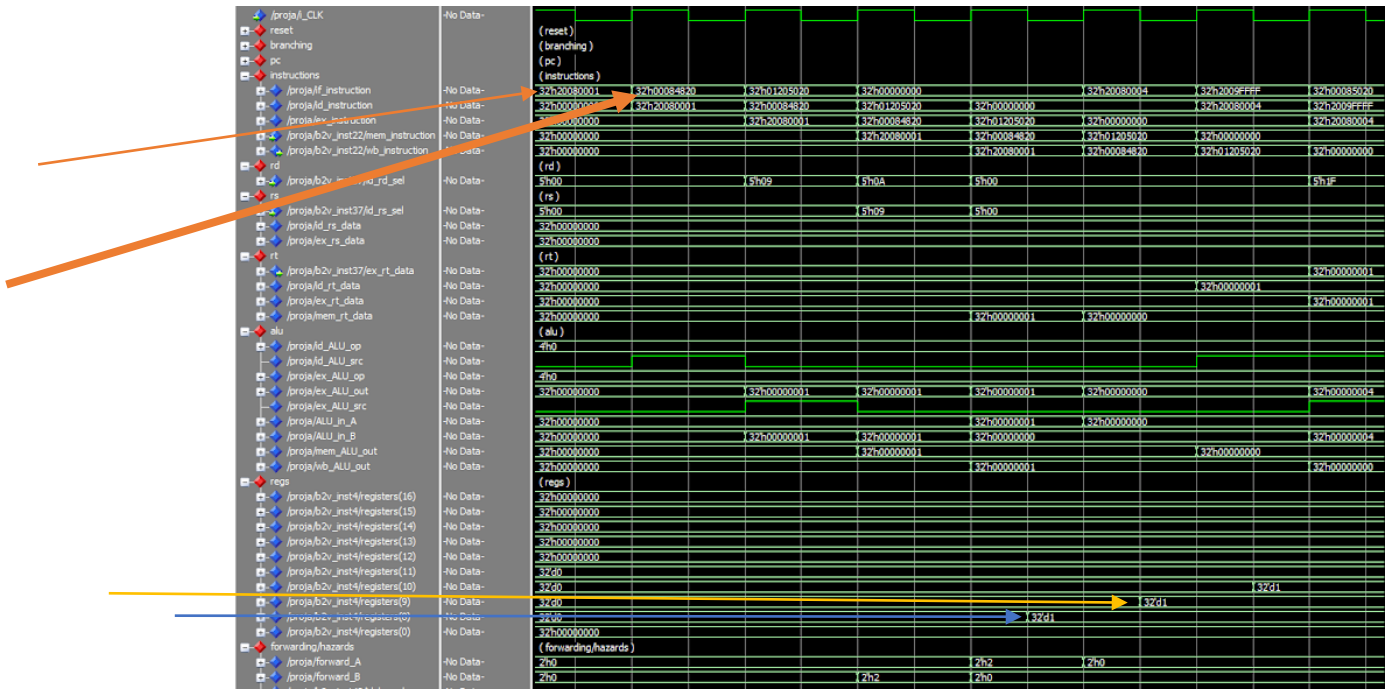
a. [Part 1] Explain the conflict and how it relates to assumption that we will be making for forwarding and hazard detection (Hint: It has to do with the fact that we assume an instruction in the ID stage will get the data being written by the WB stage).

The conflict comes that the data that are being read in the ID stage might not be correct. For an example if the register was modified in the instruction previous, then it hasn't been updated in the memory. An example of this would be if you have an instruction set like
addi R3, R1, 10
This would be a problem because R1 hasn't be written into memory before the addi instruction would try to read it.

b. [Part. 4] Provide a description of a few test cases (at least one test case for each of the six instructions) and clear screenshots depicting your functioning test cases.

- ADD
  Test: have one register that was set beforehand (either in the text document or via an addi) and add the 2 registers and check in register memory that it was set properly.

[Grab your reader's attention with a great quote from the document or use this space to emphasize a key point. To place this text box anywhere on the page, just drag it.]



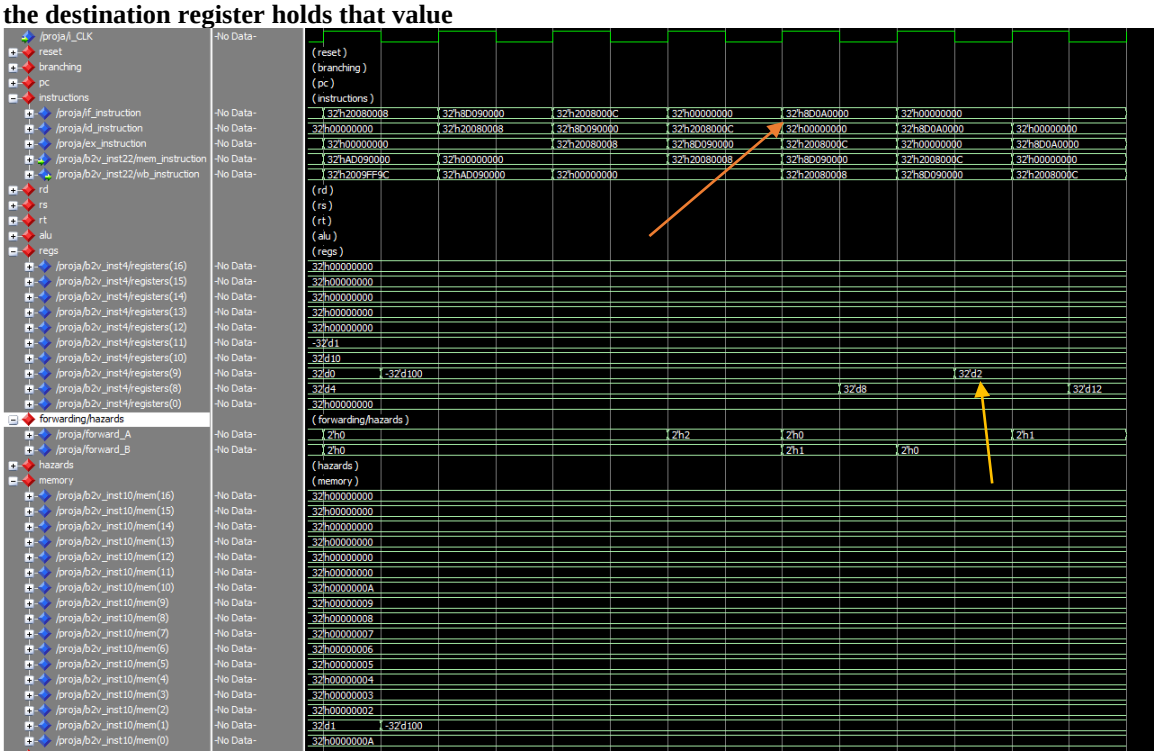
The first 2 arrows (orange) shows addi and add respectively

The next arrow (yellow) shows that register 9 (t1) is also 1 as expected in the add instruction.

The arrow at the bottom (blue)shows that 1 is being stored to $t0 (as expected for the first addi instruction)

The ALU_out should stay 1 for the entire test, which is does for this picture

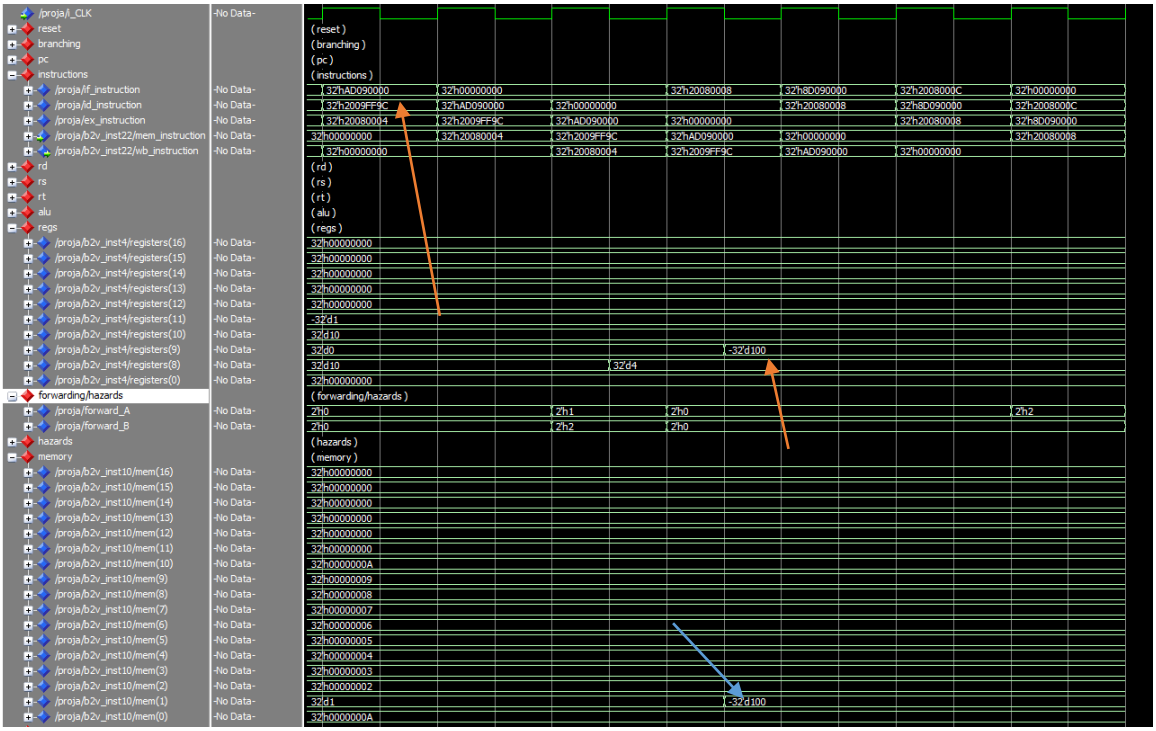
- ADDI
  Have a register hold the result of adding $zero and some immediate and check that the register holds the immediate in register memory
- LW
  Set a register to have the address and call load word, using mem on ModelSim verify that the destination register holds that value



The orange arrow shows the instruction lw $t2,0($t0)
The yellow line shows that the instruction is pulling info from memory 8 (t0) and storing it to t2 (number 9)

- SW

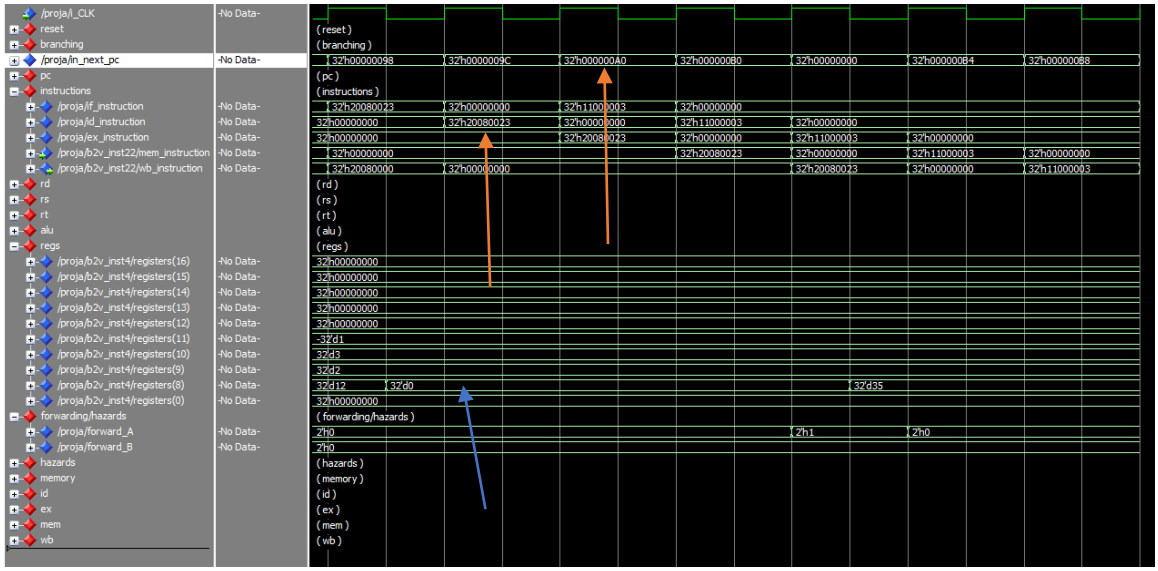
  Using the information provided in the text file, store the address in a register and call store word. Verify that the destination register holds the value expected.



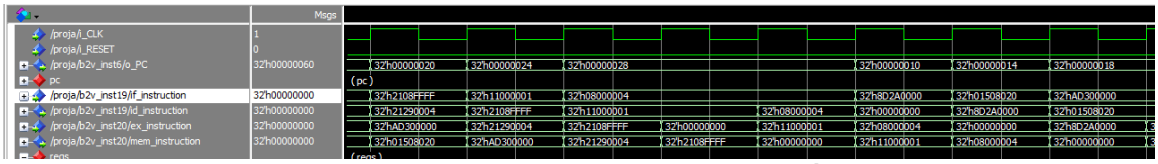
The orange arrows show that the instructions for from whatever is at $t2 and store it to address at $t0 (blue arrow)

- BEQ

  Set 2 registers so that they hold the same value, made another address the PC where we want to go if equal. Call the MIPS and verify that the program went to the address expected.



The first orange arrow shows the BEQ instruction. The (Blue) arrow show that it's comparing $zero with $t0. The second orange arrow shows the new address



The jump works because the instruction doesn't change by 4. 0x00000028 → 0x00000010

- J
  Call jump with a set address and verify that PC_Next is that address we wanted it to.

c. [Part 5] Implement ID stage branch resolution and provide a legible simulation-screenshot of a taken-branch instruction correctly executing.

***In reference to the BEQ picture* the branch is calculated in the Id stage, by evidence of that it's one clock cycle after it was fetched.**

d. [Part 6a] Implement a forwarding unit (using VHDL) to support the following data dependent cases. Give simulation screenshots of correct forwarding for each case, make sure to provide an explanation of the instructions you ran to show the below hazards:

i) ALU producer to ALU consumer at distance 1 (e.g. ADD $1, $2, $3; ADD $4, $1, $2)

***In reference to the addi/add photo* the picture also shows forwarding for this instruction. Information from the last add is being used in the next.**

ii) ALU producer to ALU consumer at distance 2 (e.g. ADD $1, $2, $3; <INST>; ADD $4, $1, $2)

***In reference to the addi/add photo* This is also shown in this picture because some of the information used in the first addi is used in the 3rd add instruction**

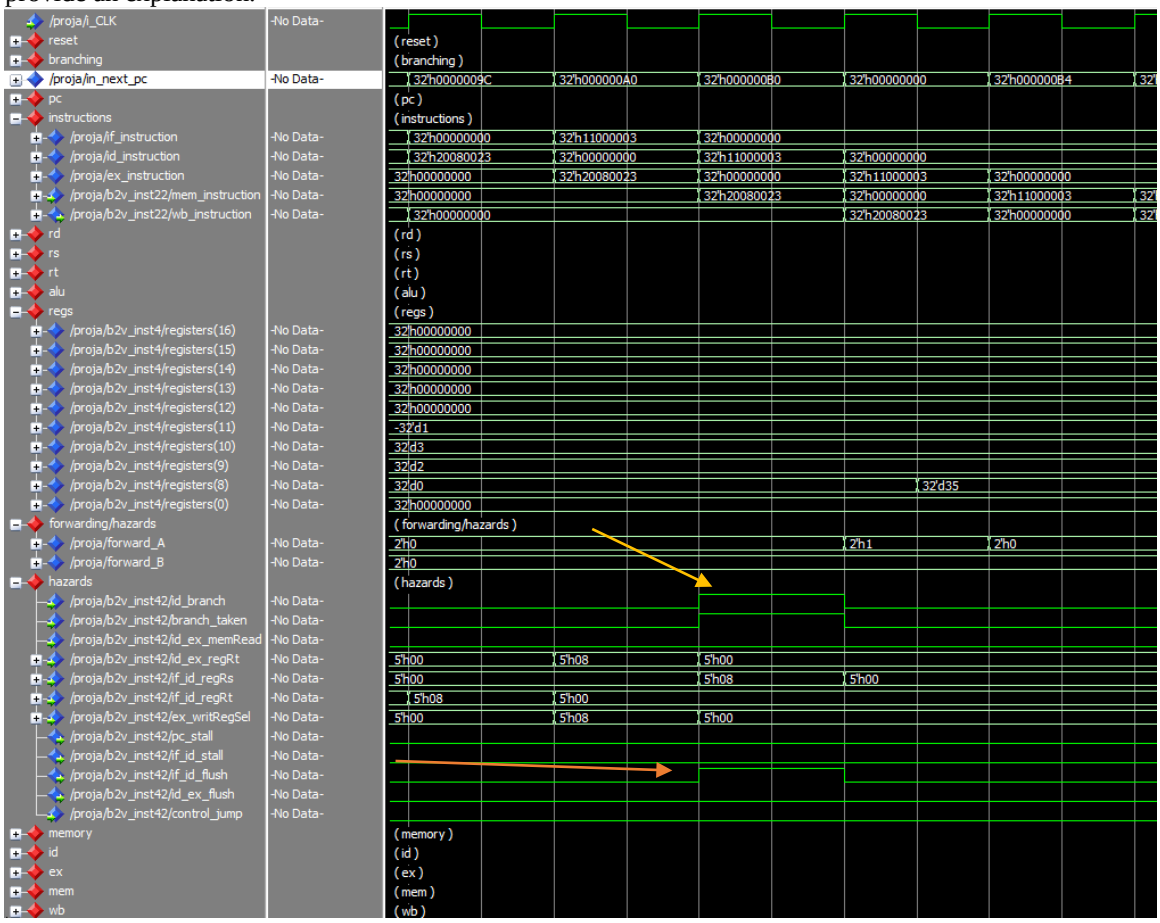
iii) Load producer to ALU consumer distance 2 (e.g. LW $1, 0($10); <INST>; ADD $5, $1, $r2)

***in reference to the LW picture* after the noop the instruction uses the information used in the load word.**

iv) ALU producer to BEQ consumer at distance 2 (e.g. ADD $1, $2, $3; <INST>; BEQ $1, $2, label)

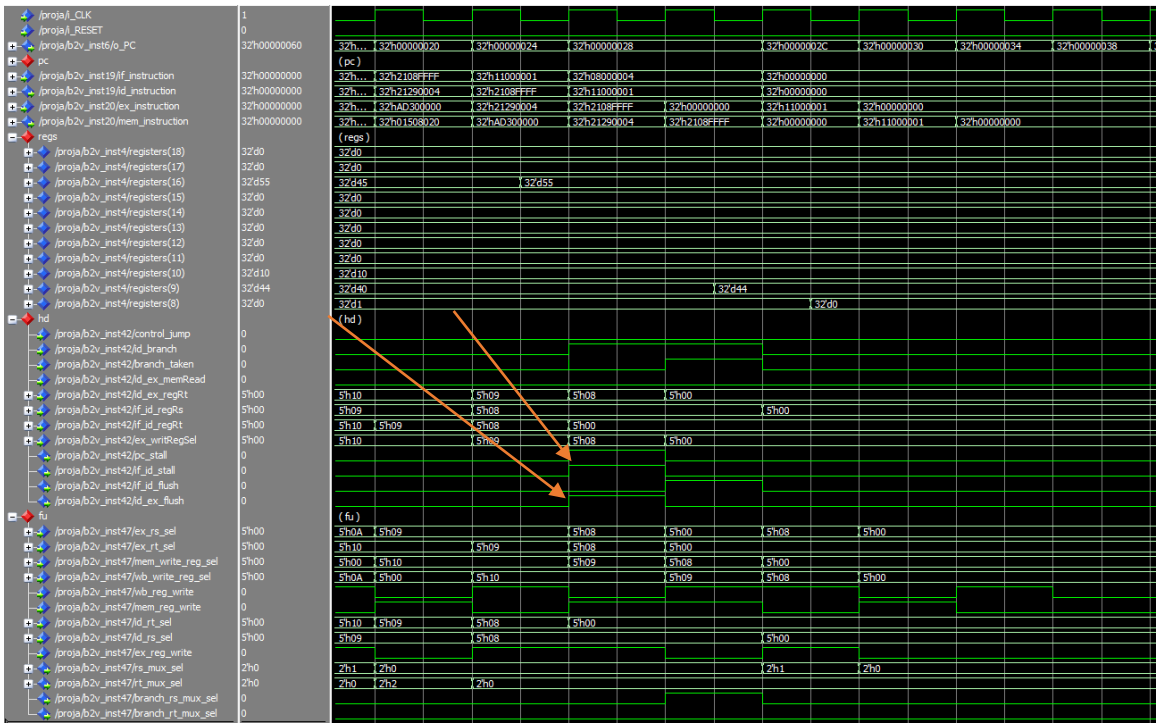
***in reference to the BEQ picture* there is an add instruction followed by a noop. With that, the information for the BEQ is forwarded so that it can be used.**

e. [Part 6b] Implement a hazard detection unit (using VHDL) to support the following data dependent cases. Give simulation screenshots of correct forwarding for the cases described in the lab manual, provide an explanation.



The orange arrow shows that a flush was used in a beq instruction. The branch taken (hence needing the flush) is shown by the yellow arrow.

f. [Part 6c] Connect your forwarding and hazard detection units to your pipelined processor and provide a simulation screenshot showing that your pipeline correctly executes the given test program.



The orange arrows show the stall and flush which correspond to the hazard detection working (evidence with ex_rs = if_id_rs)

Extra Credit (5 points) (This is not required):

- Explain the potential impact that the falling edge trigger register fill has on the critical path for the processor we are designing (2.5 points).

  If you only read on the falling edge you insure that you are reading after the WB has happened whole only writing on falling edge means the register won't be updated enough and the processor might have incorrect information in its system.

- Implement a fix (in VHDL) to the issue we have caused that will allow the register file to continue to write on the rising edge. Draw (or screenshot) the change that you have made and explain how it solves the potential issue.