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

Project B - Report

Name(s) Yao Jiang Cheah / Alain Njipwo

Section/Lab Time Thursday 4.10 - 6

Refer to the highlighted language in the Project B 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).

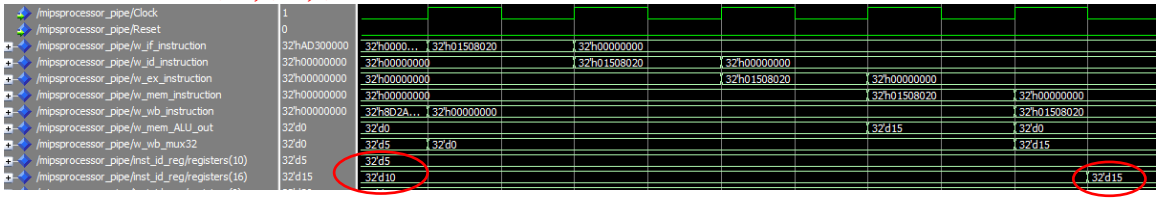
Give an example sequence of instructions that the processor we are designing will not be able to properly handle without additional logic or another change.

The conflict arises when the corresponding few instructions are dependent on the first instruction. This will cause many hazards including data hazards, and control hazards. This happens because the new instruction that is in the ID stage would try to read data from the register but without the value in the register being updated by the previous instruction that the new data is probably still in the EX, MEM, or WB stage. In order to resolve the data hazards, we would first implement a forwarding unit at the EX stage to forward EX/MEM ALU output or MEM/WB final output into the input of the ALU. Data hazard detection unit is needed to stall or flush the pipeline register because with some combination of instruction (lw follow by ALU instruction), a stalling is required because we will have to wait for the data to be computed somewhere in the pipeline before we can forward it using the forwarding unit.

Example:
- Required forwarding
  - addi $4, $0, 1
  - addi $2, $4, 2
  - add $3, $0, $4
- Also required hazard detection unit to stall and flush
  - lw $8, 20($0)
  - add $9, $8, $8

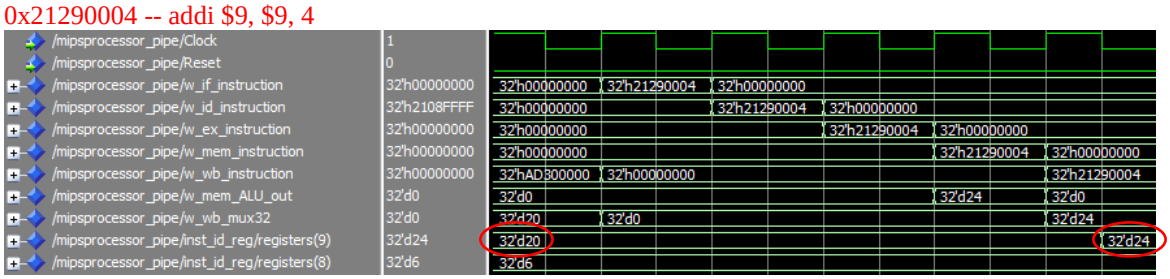
b. [Part. 4] Provide a description of a few test cases (at least one test case for each of the six instructions) and clear screenshots depicting your functioning test cases.
- ADD

  0x01508020 -- add $16, $10, $16



- ADDI

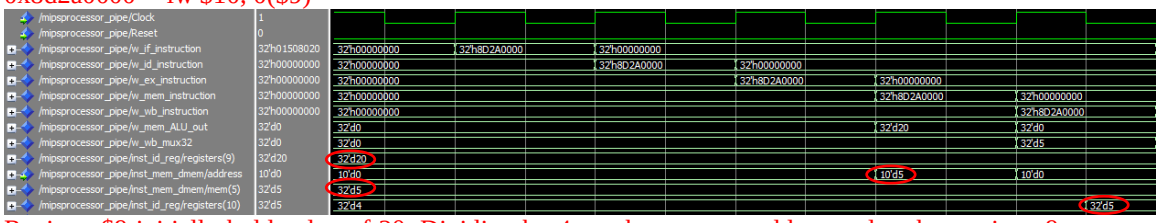
  0x21290004 -- addi $9, $9, 4



  Register $9 hold the value 20 and it's being add with immediate value 4. Therefore, the value updated to 24. We can also see how the value 24 is computed and pass to w_mem_ALU_out and to w_wb_mux_32 and into the register.

- LW

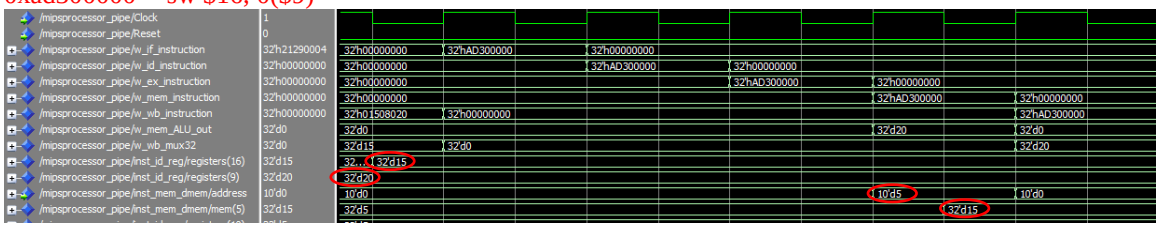
  0x8d2a0000 -- lw $10, 0($9)



  Register $9 initially hold value of 20. Dividing by 4, so the memory address value that register 9 point to is 5. Then the value in memory 5 hold value of 5. It is then being read and store into register 10.

- SW

  0xad300000 -- sw $16, 0($9)



  Register $16 initially holds a value of 15. Register $9 initially holds a value of 20. Dividing by 4, so the register $9 direct to memory address 5. So the value of 15 in register $16 is being saved into memory address 5.

- BEQ

  Branch true:

  0x10000004 -- beq $zero, $zero, test9

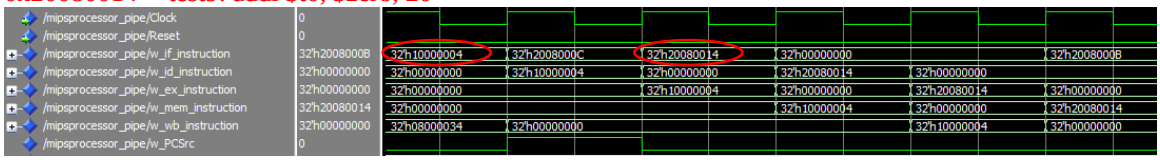
  0x2008000c -- addi $t0, $zero, 12

  0x00000000

  0x00000000

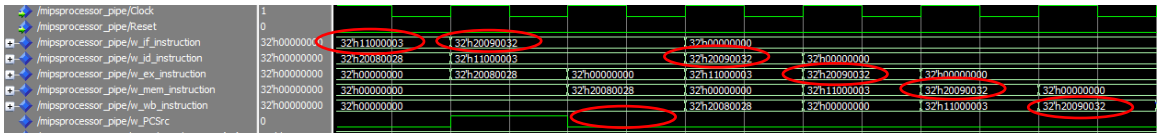
  0x00000000

  0x20080014 -- test9: addi $t0, $zero, 20



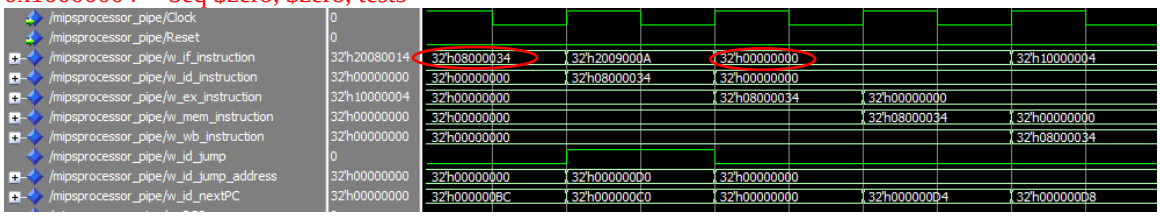
  Waveform above shown the branch instruction is implemented at the ID stage, which is when the instruction 0x2008000C was taken. Then the condition came to true so it branches to 0x20080014. The wire w_PCSrc is the branch combined condition that is the output from output of branch comparator AND branch_control.

Branch false:
0x11000003 -- beq $t0, $zero test11
0x20090032 -- addi $t1, $zero, 50
0x00000000
0x00000000
0x00000000 – test 11:



In this case, the branch condition is not taken. Therefore, after stalling the isntruction 0x20080032 for one clock cycle, the program decided to compute it since the branching condition comes to false. In this case, due to data hazard, the branch condition w_PCSrc that we are looking at is the second clock cycle of instruction 0x20080032.

- J
  0x08000034 -- j test8
  0x2009000a -- add $t1, $zero, 10
  0x00000000
  0x00000000
  0x00000000
  0x00000000 -- test8:
  0x00000000
  0x10000004 -- beq $zero, $zero, test9

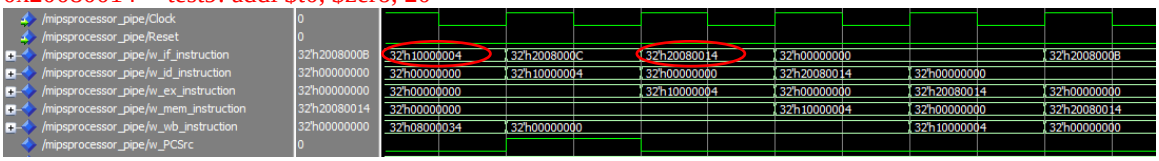


In the above case, the instruction jump at the ID stage, which is the same time when instruction 0x2009000a is being read into IF stage. Then the instruction 0x2009000a is being flushed and the program jump to the test8 line which is the 2nd last 0x00000000 instruction before beq instruction. After two clock cycles, it officially enters the 0x10000004 instruction.

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

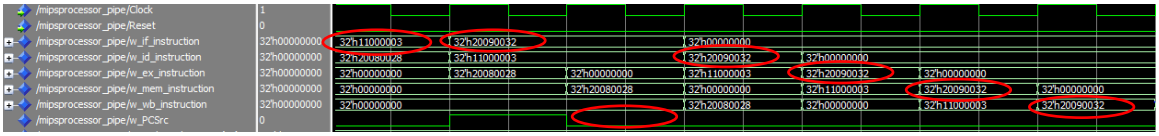
Since we have already implemented ID stage branch resolution in the previous part, it will be the same:

Branch true:
0x10000004 -- beq $zero, $zero, test9
0x2008000c -- addi $t0, $zero, 12
0x00000000
0x00000000
0x00000000
0x20080014 -- test9: addi $t0, $zero, 20



Waveform above shown the branch instruction is implemented at the ID stage, which is when the instruction 0x2008000C was taken. Then the condition came to true so it branches to 0x20080014.
The wire w_PCSrc is the branch combined condition that is the output from output of branch comparator AND branch_control.

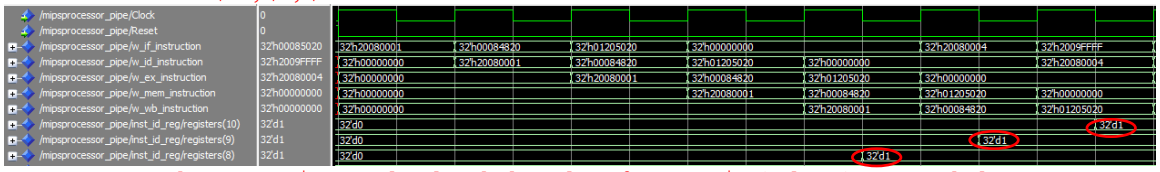
Branch false:
0x11000003 -- beq $t0, $zero test11
0x20090032 -- addi $t1, $zero, 50
0x00000000
0x00000000
0x00000000 – test 11:



In this case, the branch condition is not taken. Therefore, after stalling the isntruction 0x20080032 for one clock cycle, the program decided to compute it since the branching condition comes to false. In this case, due to data hazard, the branch condition w_PCSrc that we are looking at is the second clock cycle of instruction 0x20080032.

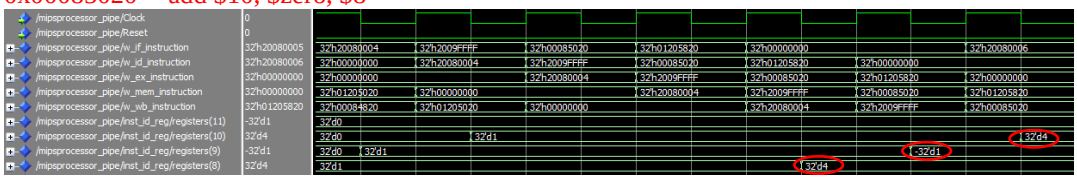
d. [Part 6a] Implement a forwarding unit (using VHDL) to support the following data dependent cases. Give simulation screenshots of correct forwarding for each case, make sure to provide an explanation of the instructions you ran to show the below hazards:

i) ALU producer to ALU consumer at distance 1 (e.g. ADD $1, $2, $3; ADD $4, $1, $2)
0x20080001 -- addi $8, $zero, 1
0x00084820 -- add $9, $zero, $8
0x01205020 -- add $10, $9, $zero



As we can see, the register $9 is updated with the value of register $8 (value 1) even with distance 1 ALU data hazard.

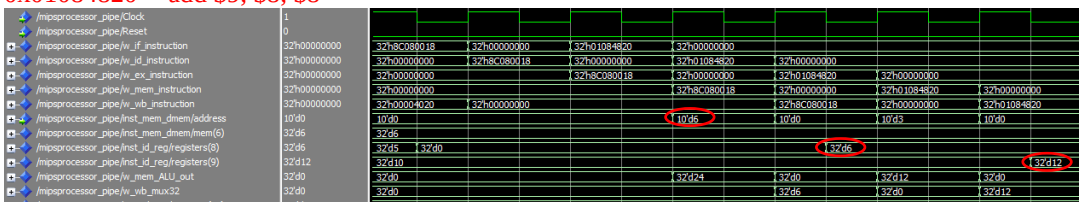
ii) ALU producer to ALU consumer at distance 2 (e.g. ADD $1, $2, $3; <INST>; ADD $4, $1, $2)
0x20080004 -- addi $8, $zero, 4
0x2009ffff -- addi $t9, $zero, -1
0x00085020 -- add $10, $zero, $8



As we can see, the register $10 is updated with the value of register $8 (value 4) even with distance 2 ALU data hazard.

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

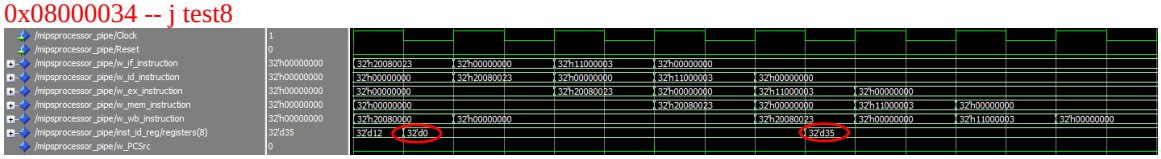
0x8c080018 -- lw $8, 24($zero)
0x00000000
0x01084820 -- add $9, $8, $8

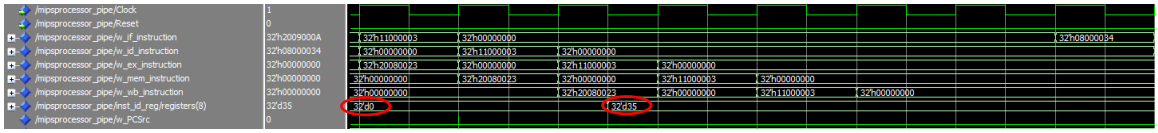


The immediate value 24 corresponds to address 6 in the memory. From the waveform above, the value of memory 6 is 6 and it is being updated to register $8. Then the value in register $8 is being added to itself and save to register $9.

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

0x20080000 -- addi $t0, $zero, 0
0x00000000
0x00000000
0x00000000
0x20080023 -- addi $t0, $zero, 35
0x00000000
0x11000003 -- beq $t0, $zero, test7
0x00000000
0x00000000
0x00000000
0x00000000 -- test7:
0x00000000
0x00000000
0x08000034 -- j test8





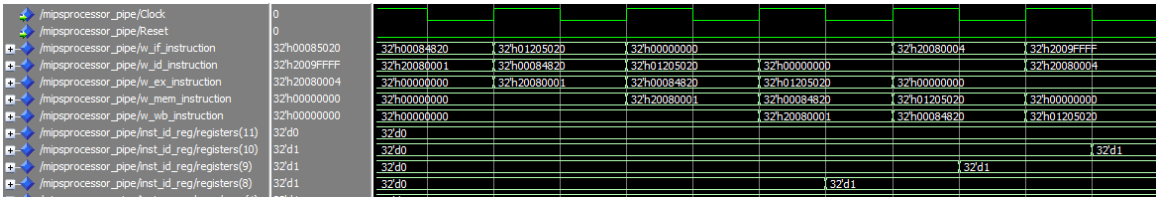
The two simulations above are the same instructions but it will not fit all outputs so we included two screenshots. As we can see, the branching condition still works whereas it comes to false and clock at 0x00000000 for 6 cycles before entering the jump instruction because it took the value 35 (new value of register $8) instead of the value 0 (old value of register $8) initially that was saved using the instruction 0x20080000.

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.

f.

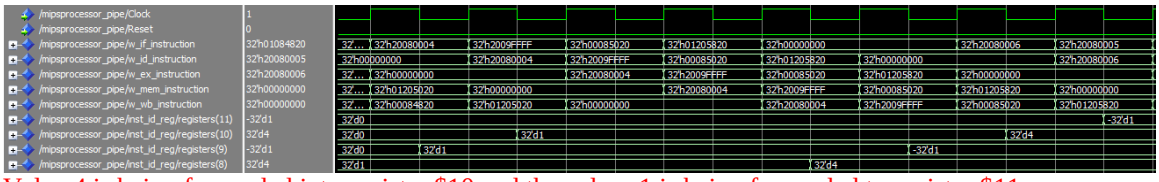
Hazard detection and forwarding unit test (HDandFUtest):

Test 1: ALU Producer to ALU Consumer EX/MEM to EX



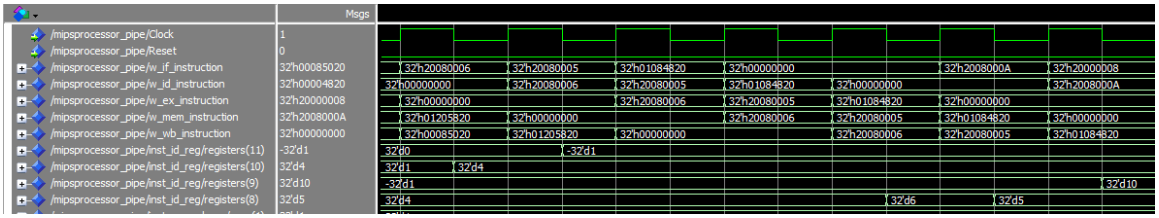
Value 1 is being forwarded from register $8 at distance 1.

Test2: ALU Producer to ALU Consumer MEM/WB to EX



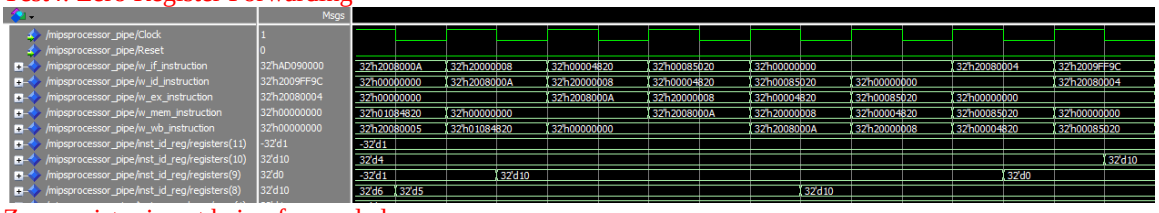
Value 4 is being forwarded into register $10 and the value -1 is being forwarded to register $11.

Test3: ALU Producer to ALU Consumer Precedence Test



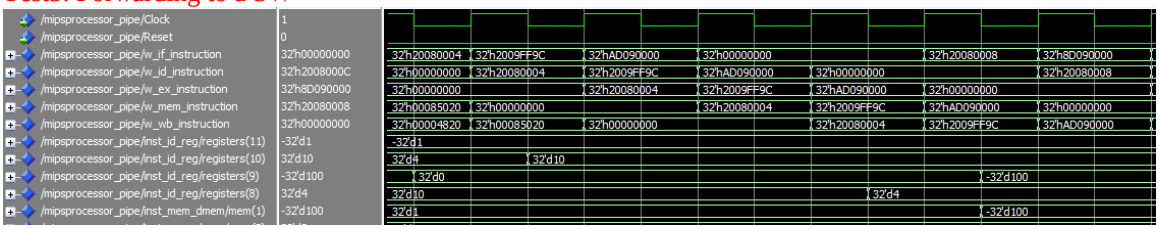
Value 5 is being forwarded for adding with itself.

Test4: Zero Register Forwarding



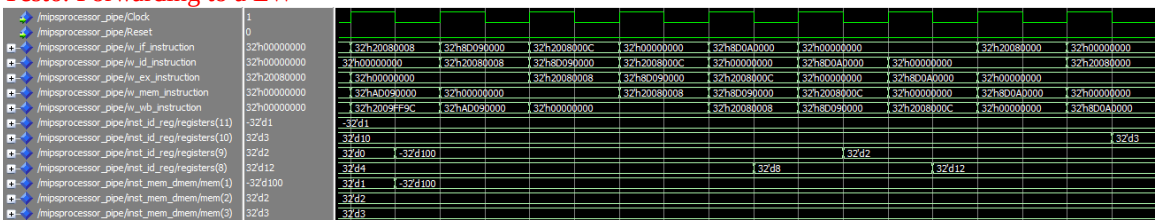
Zero register is not being forwarded.

Test5: Forwarding to a SW



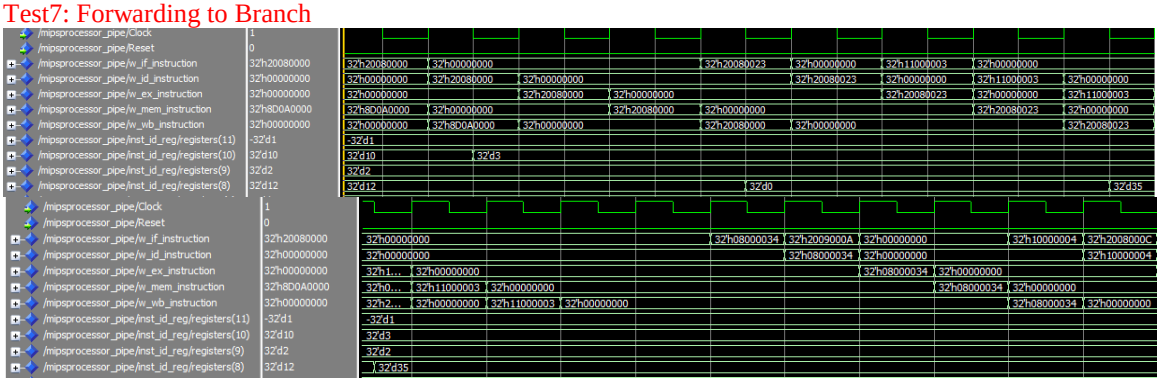
The value -100 is being forwarded for sw, mem(1) is updated to value -100.

Test6: Forwarding to a LW



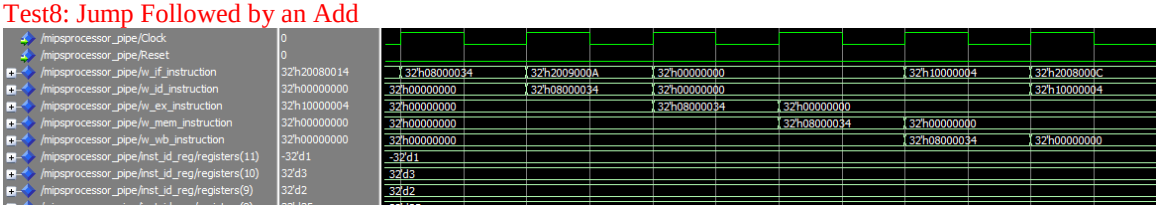
Value 8 is being forwarded to lw as a loading address rs.

Test7: Forwarding to Branch



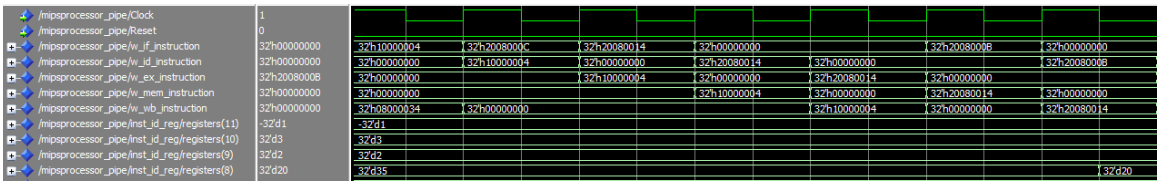
Two waveforms above are continuous of each other. The code branched at ID stage.

Test8: Jump Followed by an Add



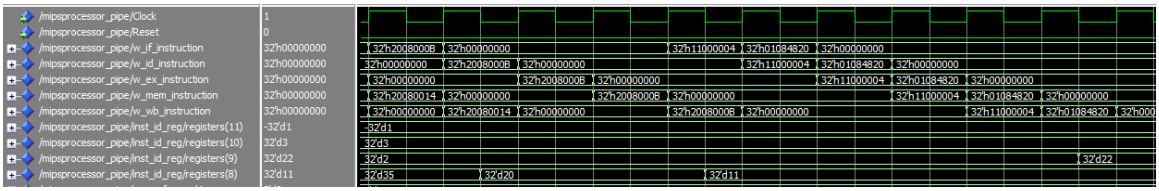
The add statement after jump was being flused therefore it is not computed.

Test9: Branch Taken then Add



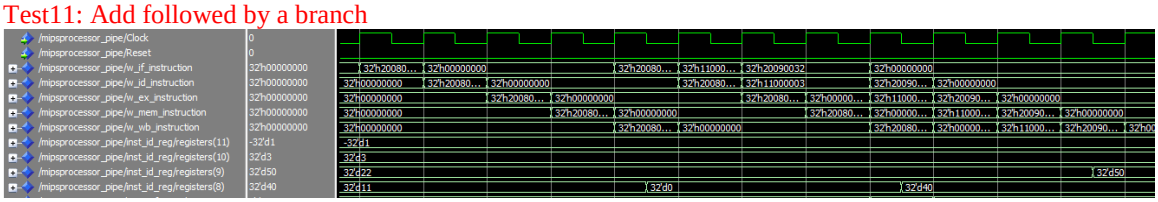
Branch is taken at ID stage so the instruction addi is being flushed.

Test10: Branch Not Taken then add



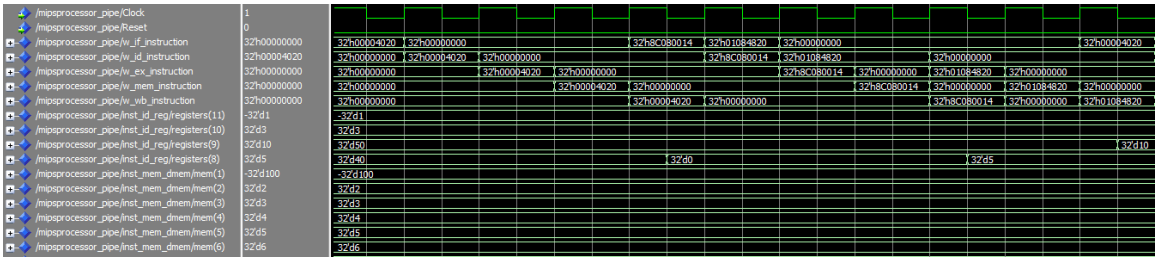
Branch not taken so the instruction add is not being flushed at its ID stage.

Test11: Add followed by a branch



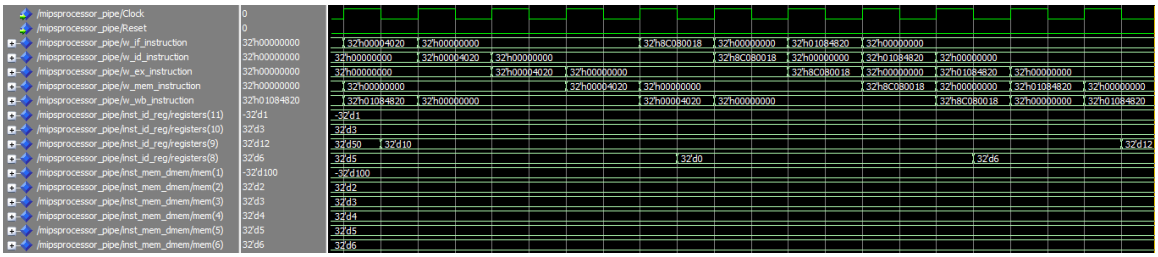
The addi operation stall the beq (at ID/EX) for one cycle to wait for the result to be done before forwarding it to the beq.

Test12: Lw to Add distance 1



The LW stall the add for one cycle (at ID/EX) to wait for the result to be done before forwarding it to the add.

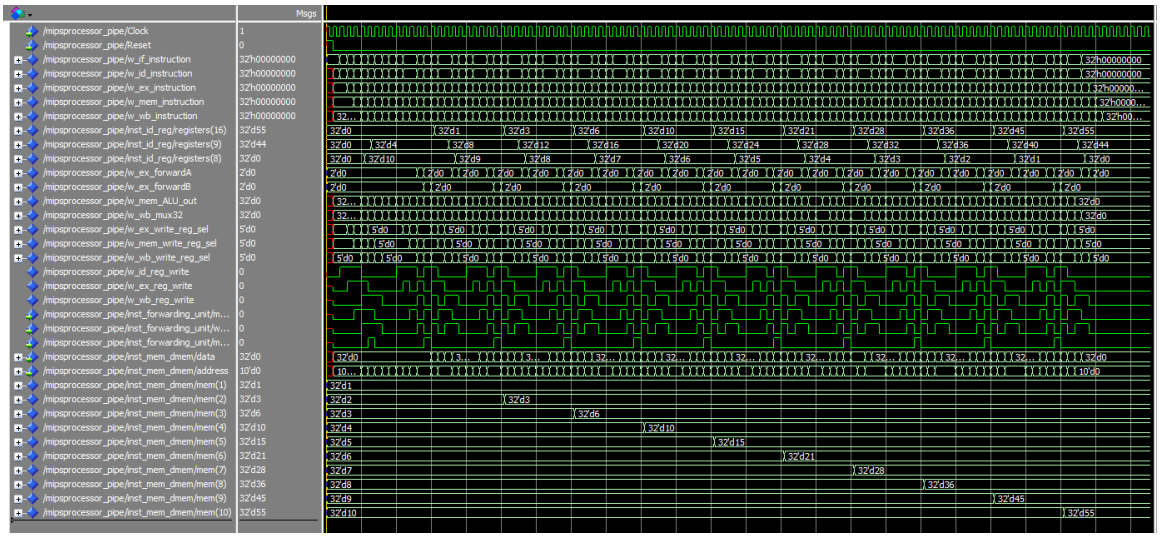
Test13: Lw to Add distance 2



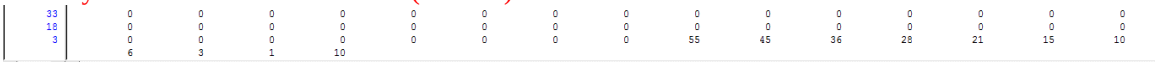
The lw flushed and stalled the add instruction for one clock cycle at the IF stage then only compute it.

g. [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.

From the simulation of "Summation test":

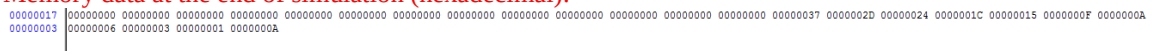


Memory data at the end of simulation (decimal):



33	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	55	45	36	28	21	15	10
	6	3	1	10										

Memory data at the end of simulation (hexadecimal):



00000017	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000037 0000002D 00000024 0000001C 00000015 0000000F 0000000A
00000003	00000006 00000003 00000001 0000000A

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).
  The memory is also falling edge trigger so if the output value from register is forwarded to the memory data input, it may cause a problem.

- 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.