CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Cameron Gilbertson
Mason Kotlarz

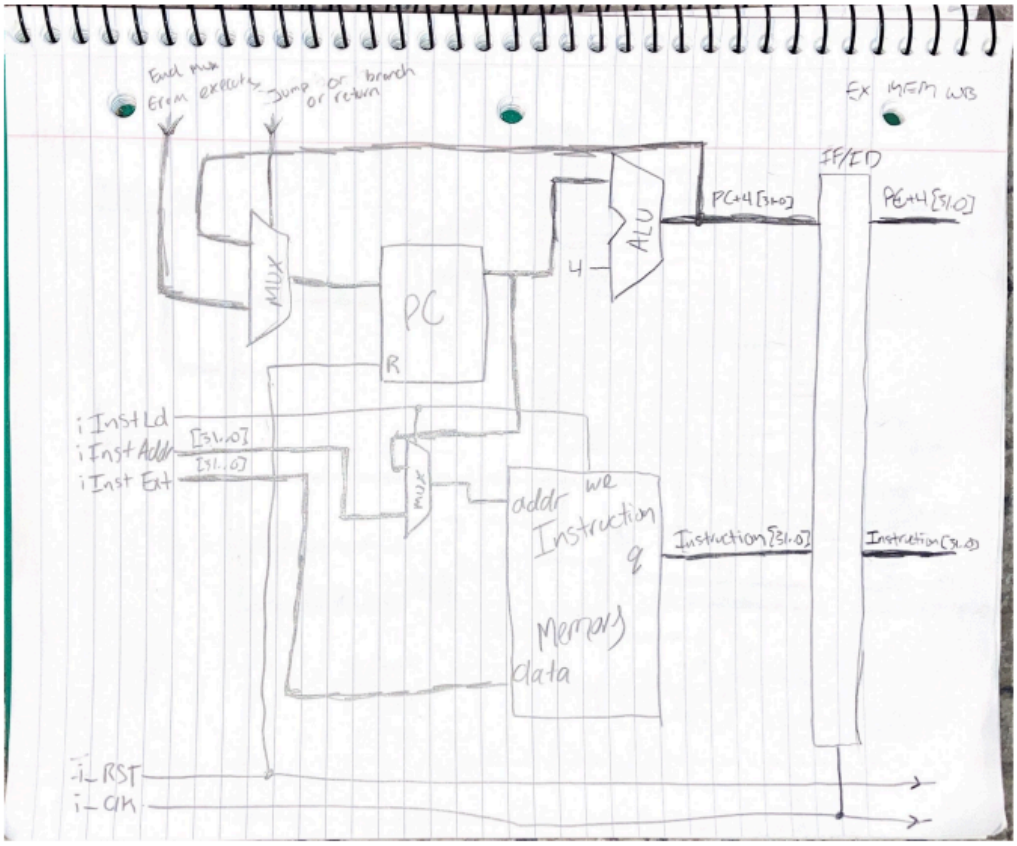
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

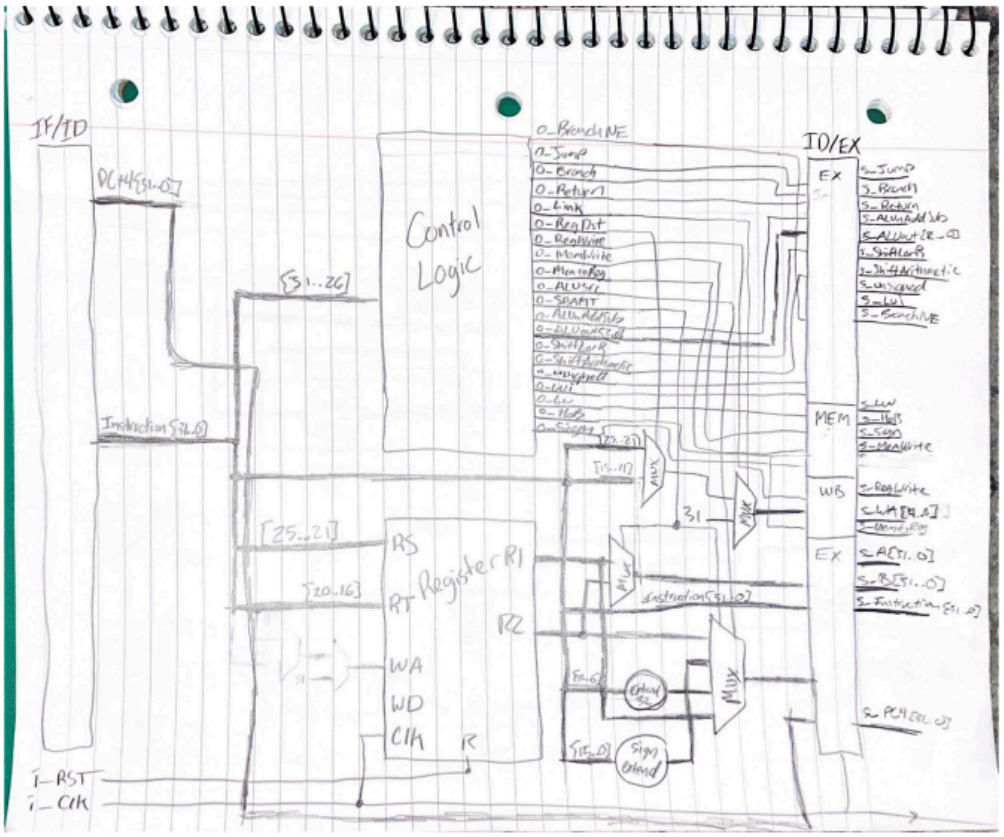
[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

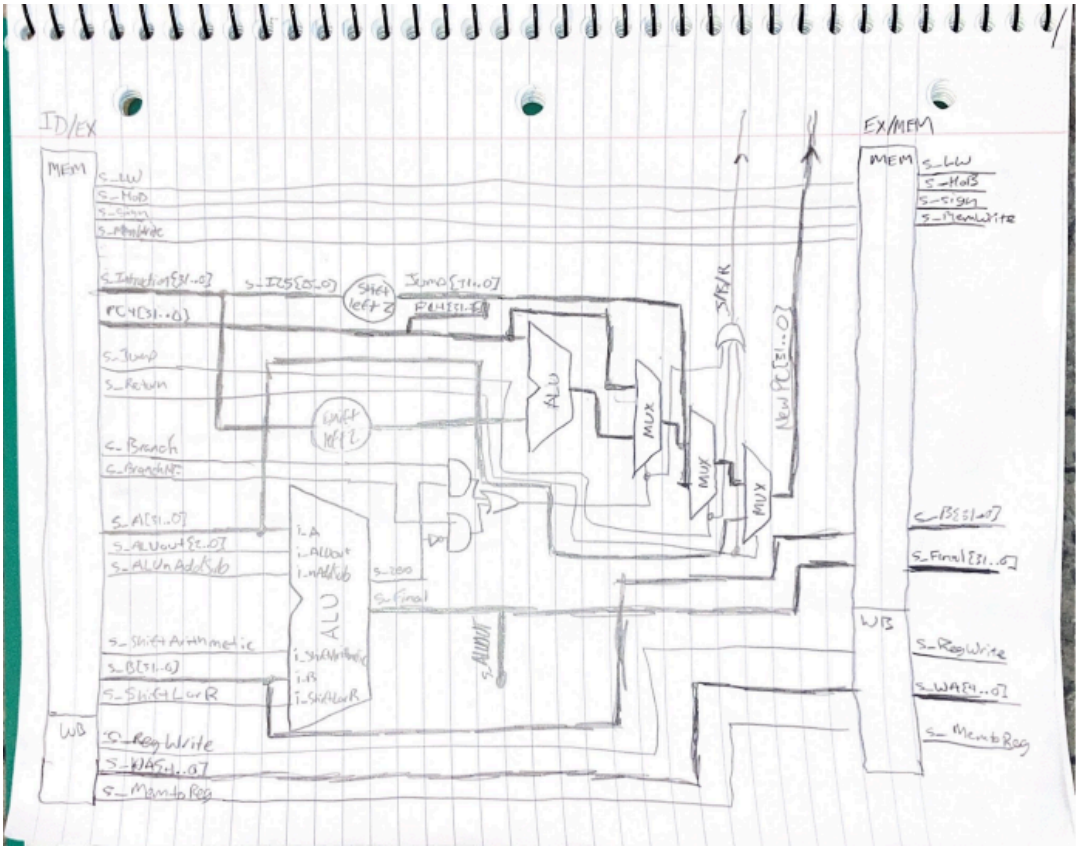
	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	**IF**			**ID**			**EX**			**MEM**			**WB**	
2	**Input**	**Output**	**IF / ID**	**Input**	**Output**	**ID / EX**	**Input**	**Output**	**EX / MEM**	**Input**	**Output**	**MEM / WB**	**Input**	**Output**
3	Jump/Branch/Return	PC+4 [31..0]	Ex	PC+4 [31..0]	PC+4 [31..0]		PC+4 [31..0]	Jump/Branch/Return		PC+4 [31..0]	PC+4 [31..0]		PC+4 [31..0]	
4	New PC [31.00]	Instruction [31..0]		Instruction [31..0]	Instruction [31..0]		Instruction [31..0]	New PC [31..0]						
5					s_Jump		s_Jump							
6					s_branch		s_branch							
7					s_Return		s_Return							
8					s_ALUnAddSub		s_ALUnAddSub							
9					s_ALUout [2..0]	Ex	s_ALUout [2..0]							
10					s_ShiftLorR		s_ShiftLorR							
11					s_ShiftArithemtic		s_ShiftArithemtic							
12					s_unsigned		s_unsigned							
13					s_Lui		s_Lui							
14					s_BranchNE		s_BranchNE							
15					s_A [31..0]		s_A [31..0]	s_Final [31..0]		s_Final [31..0]				
16					s_B [31..0]		s_B [31..0]	s_B [31..0]		s_B [31..0]				
17					s_Lw		s_Lw	s_Lw		s_Lw				
18					s_R2		s_R2	s_R2		s_R2				
19					s_HoB	MEM	s_HoB	s_HoB	MEM	s_HoB				
20					s_Sign		s_Sign	s_Sign		s_Sign				
21					s_MemWrite		s_MemWrite	s_MemWrite		s_MemWrite				
22				s_WD [31..0]	s_MemtoReg		s_MemtoReg	s_MemtoReg		s_MemtoReg	s_WD [31..0]		s_WD [31..0]	s_WD [31..0]
23				s_WA [4..0]	s_WA [4..0]		s_WA [4..0]	s_WA [4..0]		s_WA [4..0]	s_WA [4..0]		s_WA [4..0]	s_WA [4..0]
24				s_RegWrite	s_RegWrite	WB	s_RegWrite	s_RegWrite	WB	s_RegWrite	s_RegWrite	WB	s_RegWrite	s_RegWrite
25					s_Link		s_Link	s_Link		s_Link	s_Link		s_Link	
26					s_Halt		s_Halt	s_Halt		s_Halt	s_Halt		s_Halt	
27														
28														
29														
30														

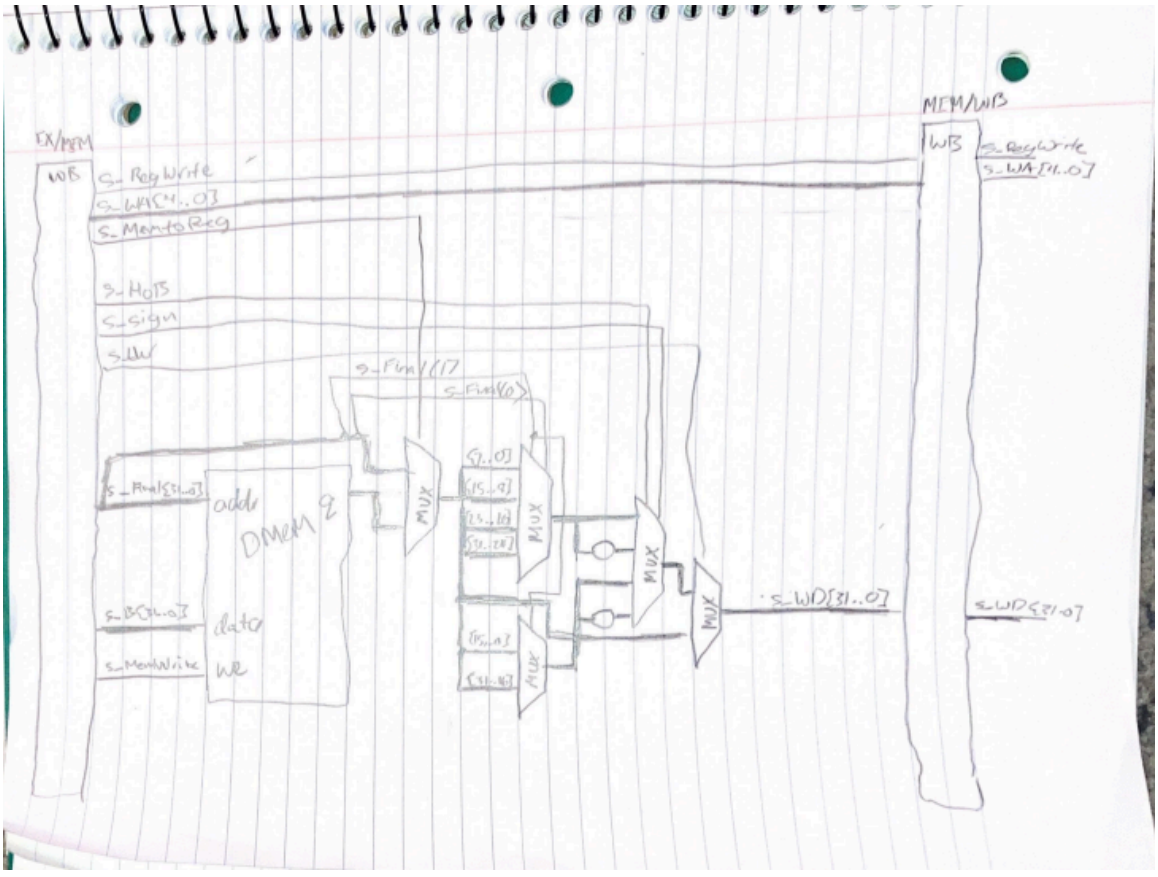
See attached google spreadsheet.

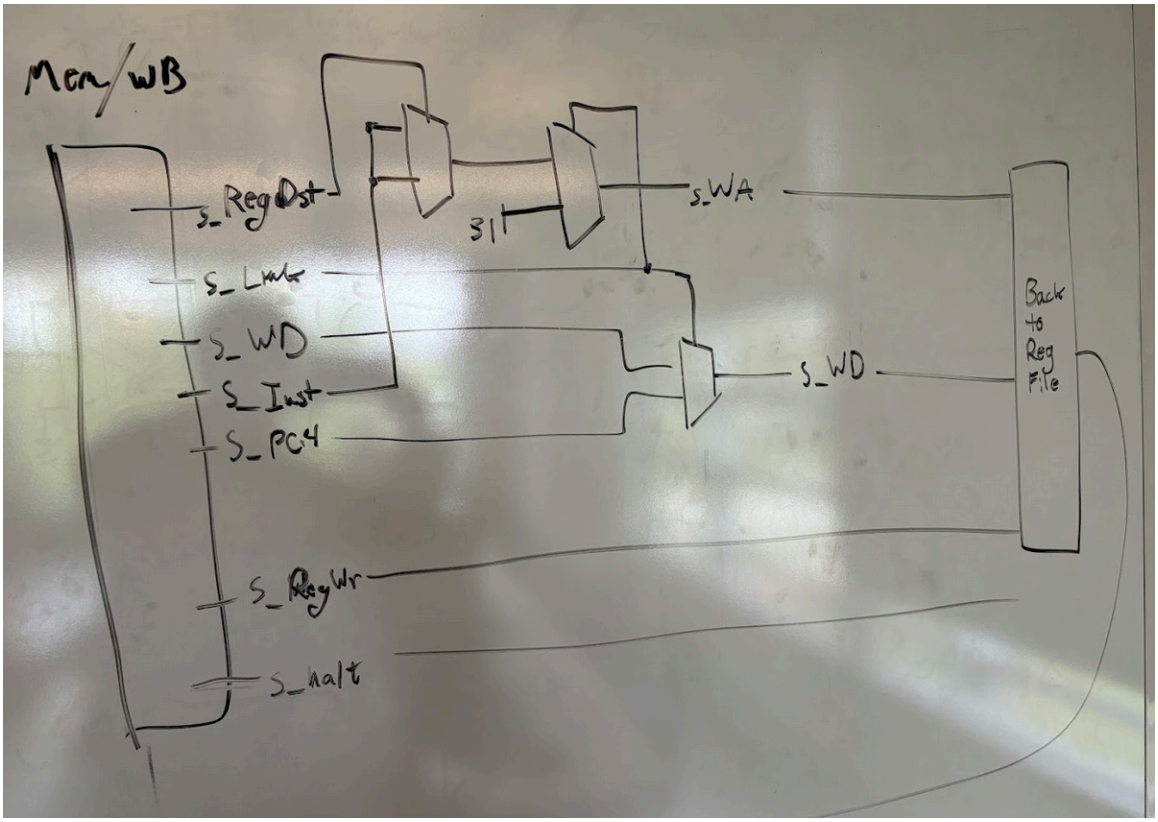
[1.b.ii] high-level schematic drawing of the interconnection between components.


Exec mux
Error exception
Jump or branch or return
EX MEM WB
IF/ID
ALU
MUX
PC
R
4
PC+4 [31:0]
PC+4 [31:0]
i_InstLd
i_InstAddr [31..0]
i_InstExt [31..0]
MUX
addr we
Instruction
q
Memory
data
Instruction [31:0]
Instruction [31:0]
i_RST
i_CLK

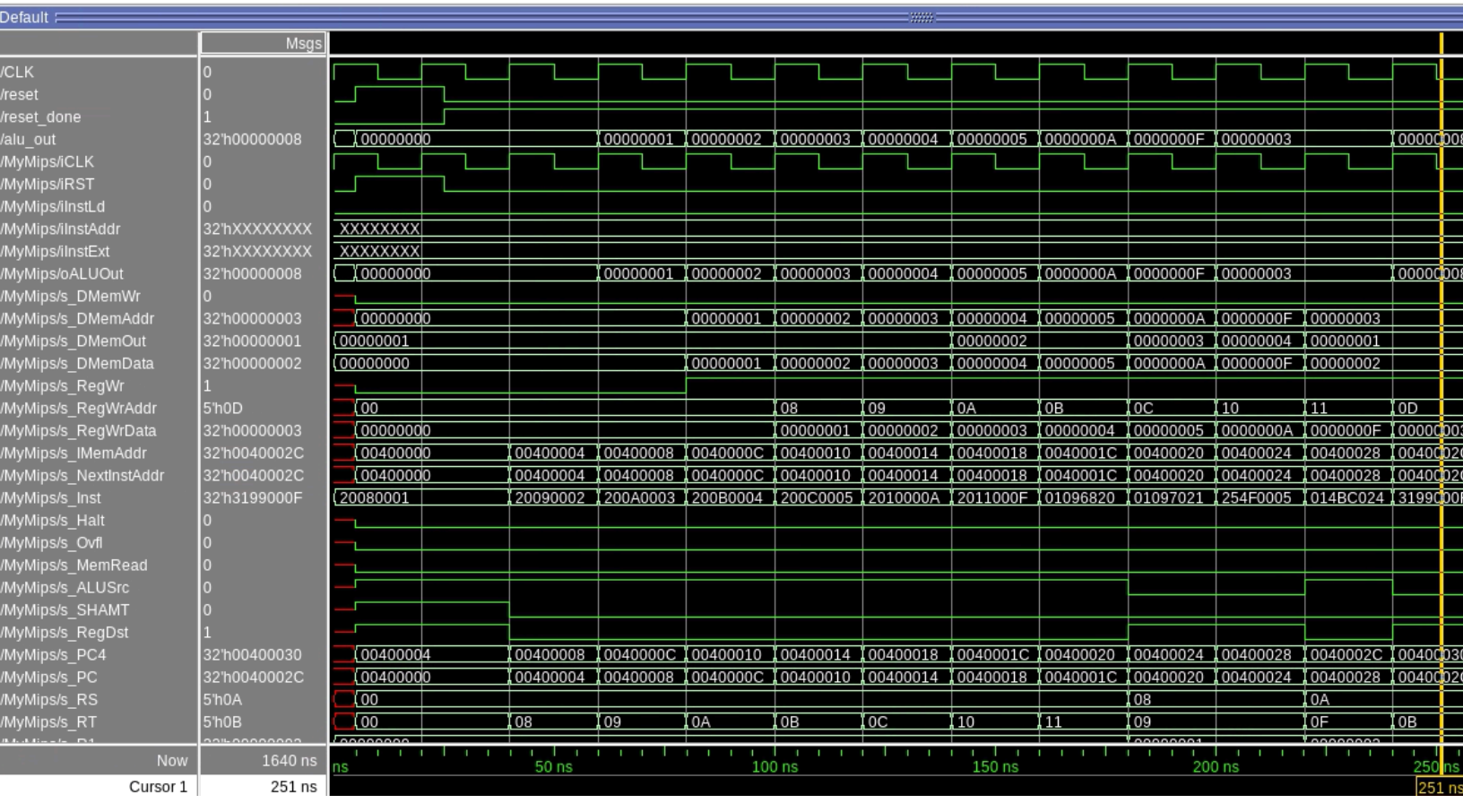

IF/ID
PC+4[31:0]
[31..26]
Instruction [31:0]
Control Logic
O_BranchNE
O_Jump
O_Branch
O_Return
O_Link
O_RegDst
O_RegWrite
O_MemWrite
O_MemtoReg
O_ALUSrc
O_SBAMT
O_ALUAddMode
O_ShiftLorR
O_ShiftArithmetic
O_Unsigned
O_LUI
O_Byte
O_Half
O_SIGAT
ID/EX
EX
MEM
WB
S_Jump
S_Branch
S_Return
S_ALUAddMode
S_ALUOut[2:0]
S_ShiftLorR
S_ShiftArithmetic
S_Unsigned
S_LUI
S_BranchNE
S_LW
S_Half
S_Sign
S_MemWrite
S_RegWrite
S_LW[2:0]
S_A[31:0]
S_B[31:0]
S_Instruction[31:0]
S_PC4[31:0]
[25..21]
[20..16]
RS
RT
Register RF
RD
WA
WD
CLK
R
31
MUX
Instruction[31:0]
Sign extend
i_RST
i_CLK


ID/EX
EX/MEM
MEM
WB
ALU
MUX
s_LW
s_MemWrite
s_RegWrite
s_MemtoReg
s_Branch
s_Jump
s_Return
s_ALUout
s_Final
s_WA
s_WD


EX/MEM
MEM/WB
WB
s_RegWrite
s_MemtoReg
s_HalfS
s_Sign
s_LW
s_Final
s_WD
s_WA
s_MemWrite
addr
data
WE
DMEM
MUX



[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

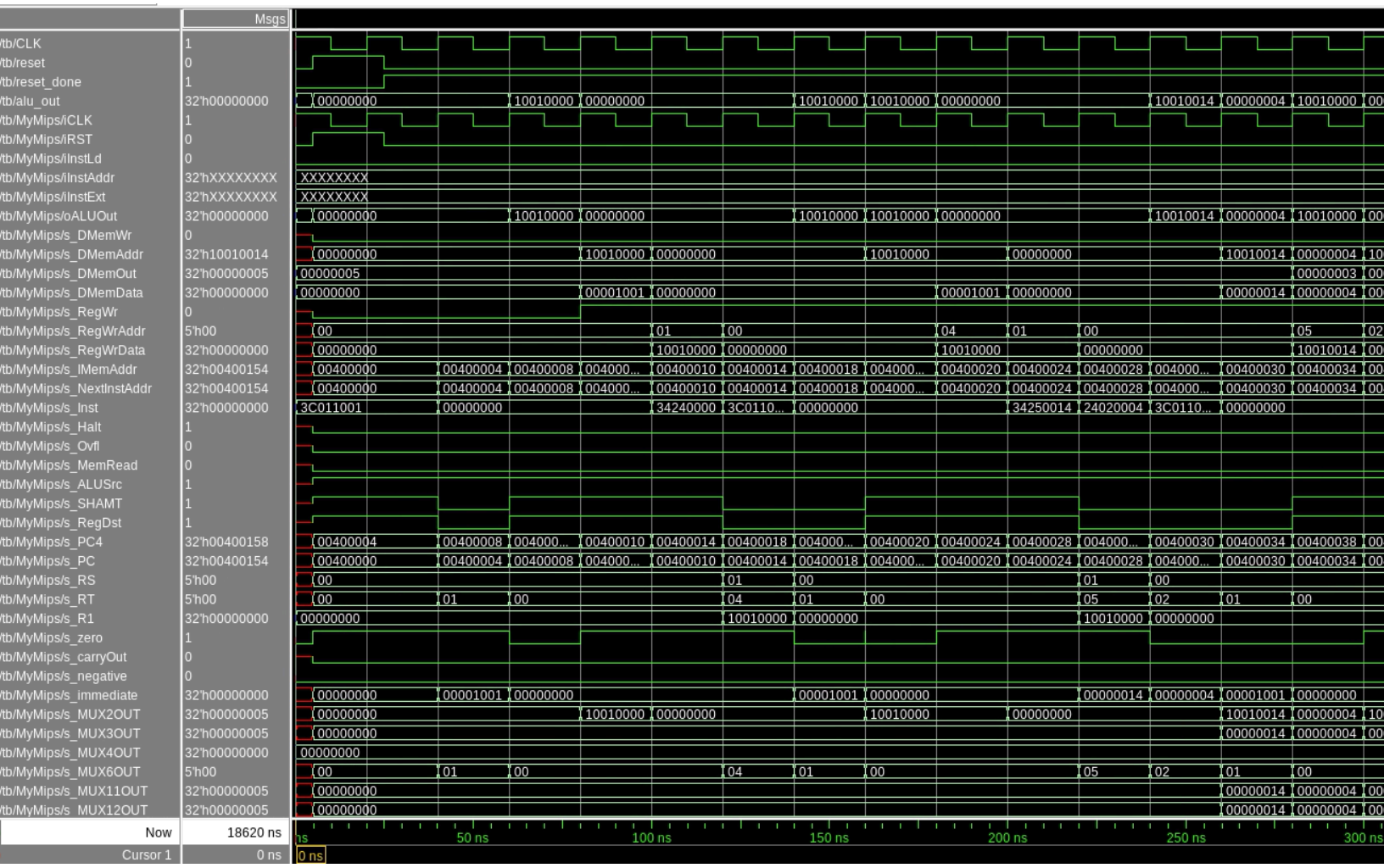


This is the waveform for the start of my base test program. This is first part of the waveform, I can verify correctness but looking at the alu_out and s_RegWrData, and s_RegWrAddr… because the alu_out is correct, and is equal to the s_RegWrData which means the alu is properly writing… The addr is the other important part if this and the data are correct then the instruction was completed properly.

for the first instruction addi $t0, $zero, 1 this is correct addr is the correct instruction address and the data is 1 which is the proper value from 0+1

for the add $t5, $t0, $t1 this is correct because the addr is the correct register, and the data is the the proper value from $t0 and $t1 and the value is set well.

[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your



waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.

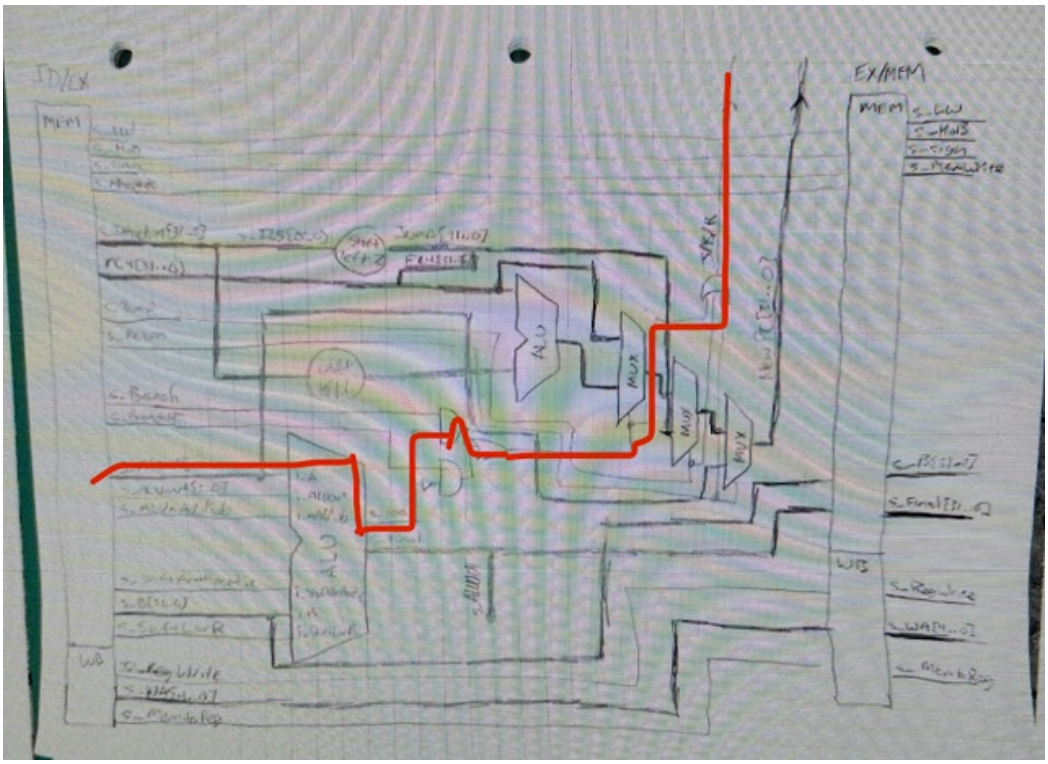
So this is correct because if you look at the first few instructions you can see that the proper value is in the s_RegWrData and the s_RegWrAddr, while the s_RegWrData and the alu_out are matching

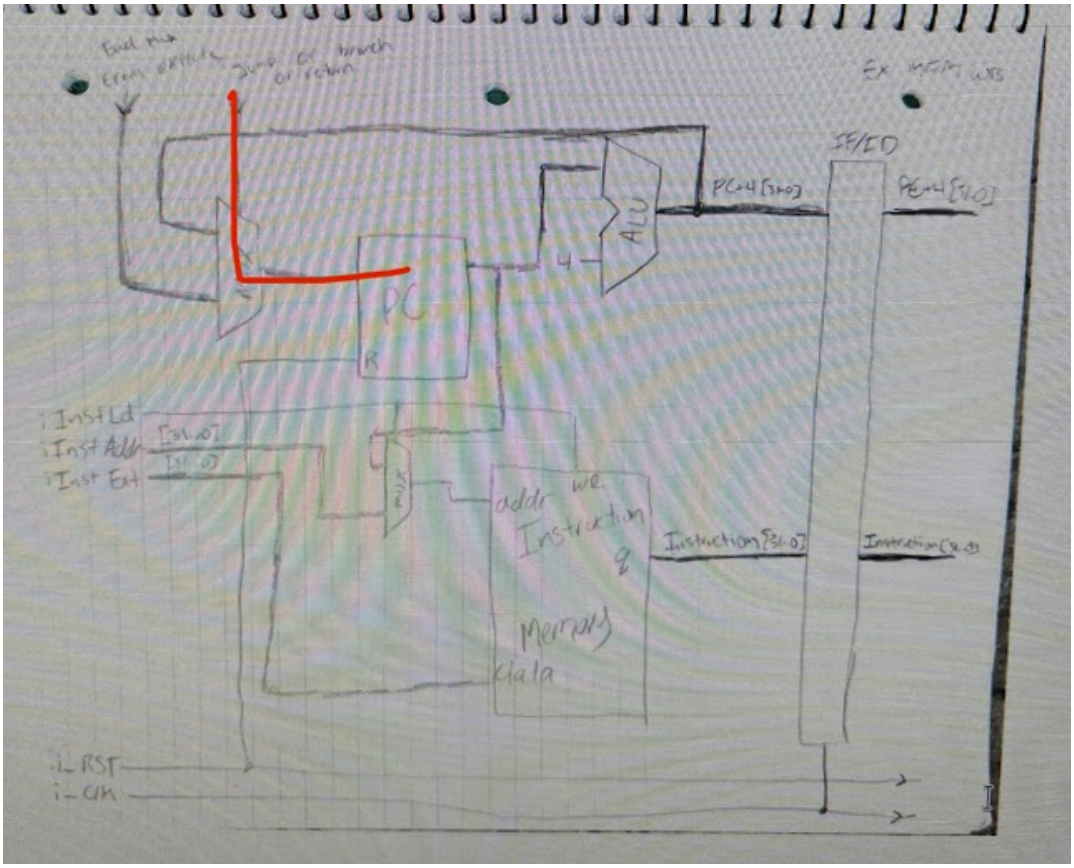
lui $1, 4097 is correct because 4097 = 1001000 and the register is 1 which can be seen in the alu_out and the in s_RegWrData and s_RegWrAddr

Here I had a la instruction which I manually converted into a lui and ori… I had to insert 3 nops because the value of the lui has to be out of the WB stage before the ori can load the value from the $1 register.

[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

The maximum frequency is 54.44 mhz which is double the length of the previous processor (25.11 mhz)

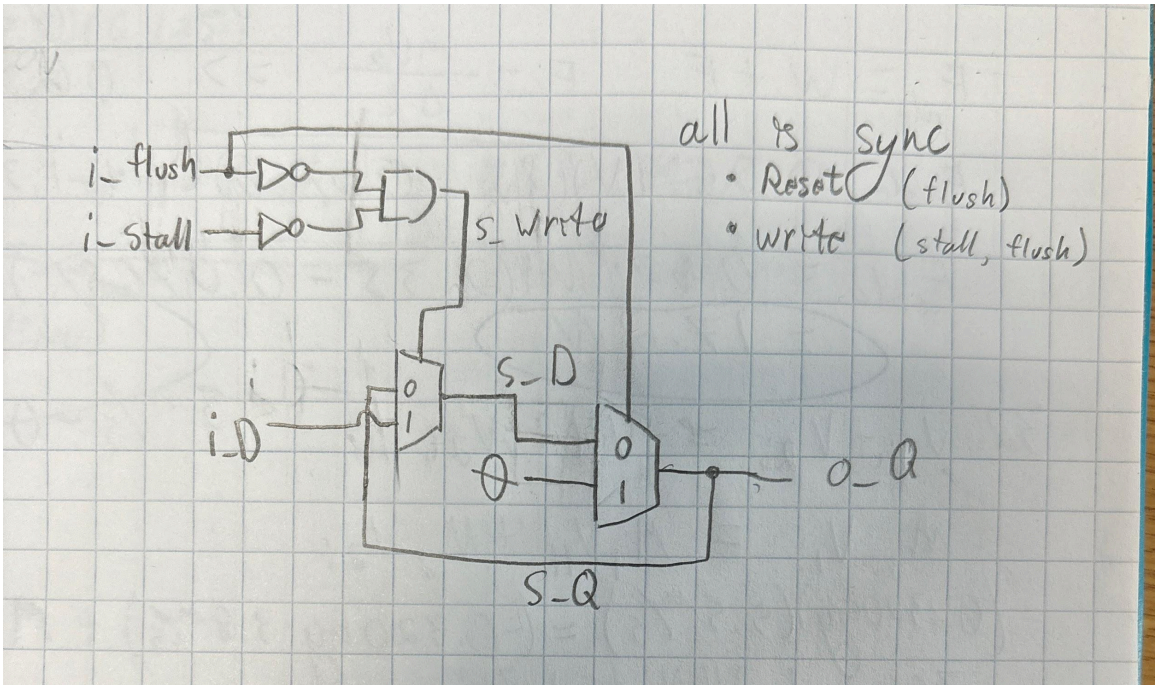




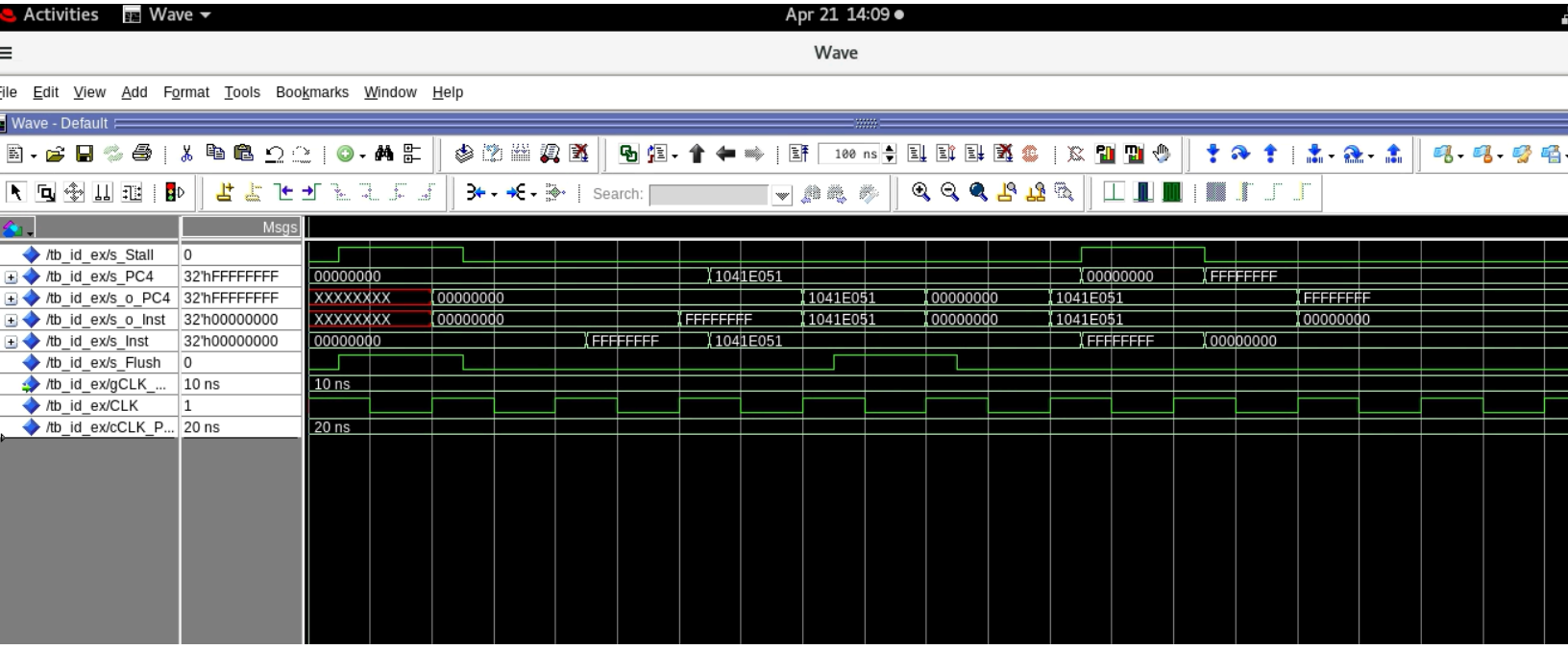
The longest path from dff to dff was in the Execution Stage of the processor where we read the values from the pipeline and the preform a sub instruction then we check if the output of that subtraction was zero or not then we use that signal to check if we should branch or not… in this case a branch must've been taken so then we update the value in the PC in the IF stage with a Mux selecting between a new pc value and pc+4…

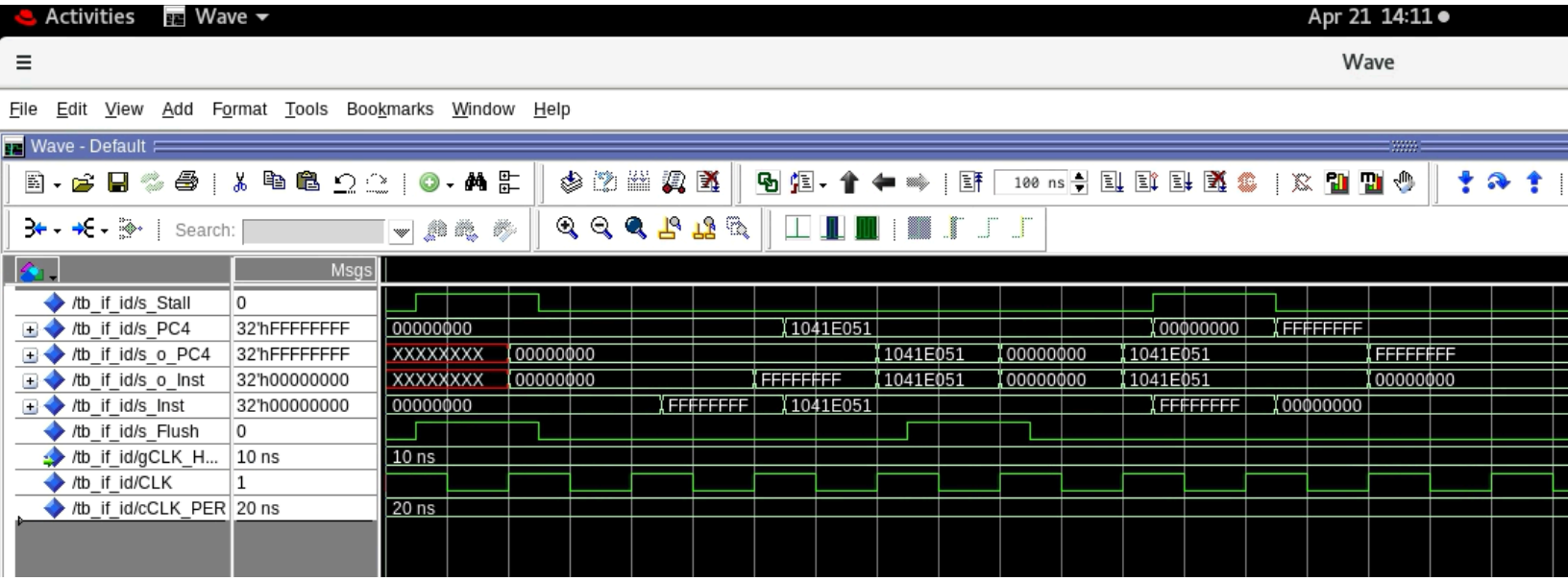
```
#
# CprE 381 toolflow Timing dump
#

FMax: 54.44mhz Clk Constraint: 20.00ns Slack: 1.63ns

The path is given below

 ====================================================================
 From Node    : ID_EX:IDEX|RegFile:REG2|dffg:\G_NBit_REG:0:DFF|s_Q
 To Node      : PC:PC4|dffg:\G_NBit_REG:16:DFF|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
      0.000     0.000           launch edge time
      3.070     3.070  R        clock network delay
      3.302     0.232    uTco  ID_EX:IDEX|RegFile:REG2|dffg:\G_NBit_REG:0:DFF|s_Q
      3.302     0.000 FF  CELL  IDEX|REG2|\G_NBit_REG:0:DFF|s_Q|q
      4.126     0.824 FF    IC  A|adder_subtractor|Nbit_Adder|full_adder_0|g_And1_And2_to_Or|o_C~0|dataa
      4.480     0.354 FF  CELL  A|adder_subtractor|Nbit_Adder|full_adder_0|g_And1_And2_to_Or|o_C~0|combout
      4.867     0.387 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:1:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      4.992     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:1:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      5.241     0.249 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:2:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      5.366     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:2:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      5.623     0.257 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:3:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
      5.904     0.281 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:3:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      6.206     0.302 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:4:rippleAdder|g_And1_And2_to_Or|o_C~0|dataa
      6.630     0.424 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:4:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      6.879     0.249 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:5:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      7.004     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:5:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      7.296     0.292 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:6:rippleAdder|g_And1_And2_to_Or|o_C~0|datab
      7.721     0.425 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:6:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      7.970     0.249 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:7:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      8.095     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:7:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      8.345     0.250 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:8:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      8.470     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:8:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      8.720     0.250 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:9:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      8.845     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:9:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      9.096     0.251 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:10:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      9.221     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:10:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      9.472     0.251 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:11:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      9.597     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:11:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
      9.849     0.252 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:12:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
      9.974     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:12:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     10.223     0.249 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:13:rippleAdder|g_And1_And2_to_Or|o_C~0|datad


      9.974     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:12:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     10.223     0.249 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:13:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     10.348     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:13:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     10.648     0.300 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:14:rippleAdder|g_And1_And2_to_Or|o_C~0|dataa
     11.072     0.424 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:14:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     11.323     0.251 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:15:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     11.448     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:15:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     11.859     0.411 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:16:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     11.984     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:16:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     12.233     0.249 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:17:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     12.358     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:17:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     12.608     0.250 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:18:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     12.733     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:18:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     12.973     0.240 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:19:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     13.098     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:19:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     13.337     0.239 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:20:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     13.462     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:20:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     13.700     0.238 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:21:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     13.825     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:21:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     14.063     0.238 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:22:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     14.188     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:22:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     14.438     0.250 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:23:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     14.563     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:23:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     14.809     0.246 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:24:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
     15.090     0.281 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:24:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     15.813     0.723 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:25:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     15.938     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:25:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     16.175     0.237 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:26:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     16.300     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:26:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     16.544     0.244 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:27:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
     16.825     0.281 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:27:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     17.064     0.239 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:28:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     17.189     0.125 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:28:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     17.435     0.246 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:29:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
     17.716     0.281 FF  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:29:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     17.952     0.236 FF    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:30:rippleAdder|g_XOR_Cin_to_XOR|o_C~0|datad
     18.102     0.150 FR  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:30:rippleAdder|g_XOR_Cin_to_XOR|o_C~0|combout
     18.327     0.225 RR    IC  A|branchchecker|Equal0~9|datac
     18.594     0.267 RF  CELL  A|branchchecker|Equal0~9|combout
     18.863     0.269 FF    IC  FETCH|or2a|o_C~0|datab
     19.252     0.389 FR  CELL  FETCH|or2a|o_C~0|combout
     19.692     0.440 RR    IC  MUX13|\G_NBit_MUX:22:MUXI|o_O~1|datad
     19.847     0.155 RR  CELL  MUX13|\G_NBit_MUX:22:MUXI|o_O~1|combout
     20.629     0.782 RR    IC  MUX13|\G_NBit_MUX:16:MUXI|o_O~6|datac
     20.916     0.287 RR  CELL  MUX13|\G_NBit_MUX:16:MUXI|o_O~6|combout
     21.122     0.206 RR    IC  MUX13|\G_NBit_MUX:16:MUXI|o_O~8|datad
     21.277     0.155 RR  CELL  MUX13|\G_NBit_MUX:16:MUXI|o_O~8|combout
     21.277     0.000 RR    IC  PC4|\G_NBit_REG:16:DFF|s_Q|d
     21.364     0.087 RR  CELL  PC:PC4|dffg:\G_NBit_REG:16:DFF|s_Q
 Data Required Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
     20.000    20.000           latch edge time
     22.966     2.966  R        clock network delay
     22.998     0.032           clock pessimism removed
     22.978    -0.020           clock uncertainty
     22.996     0.018     uTsu  PC:PC4|dffg:\G_NBit_REG:16:DFF|s_Q
 Data Arrival Time  :    21.364
 Data Required Time :    22.996
 Slack              :     1.632
 ====================================================================
```

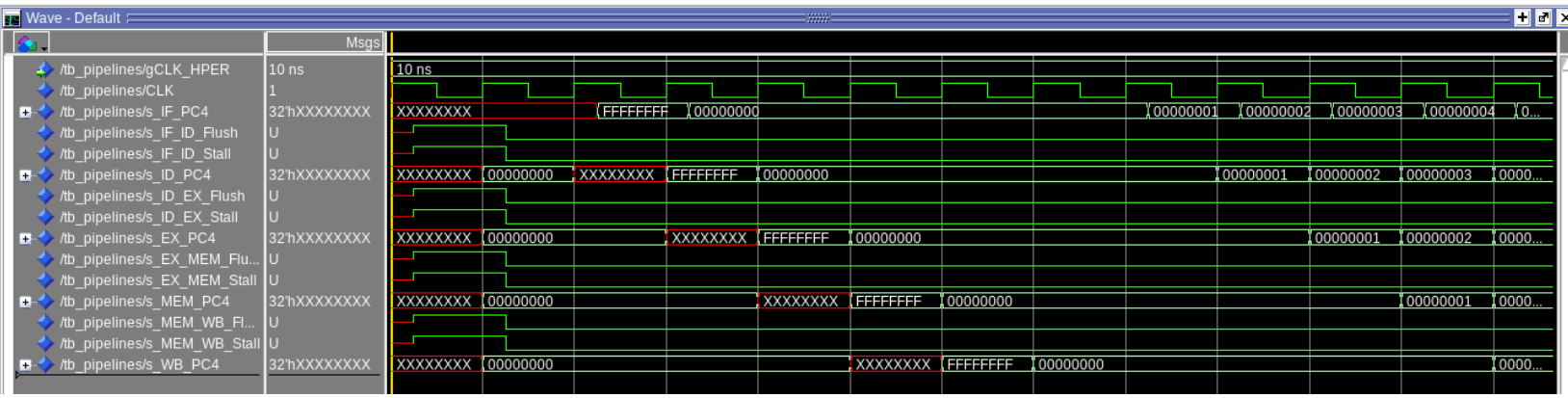
[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.



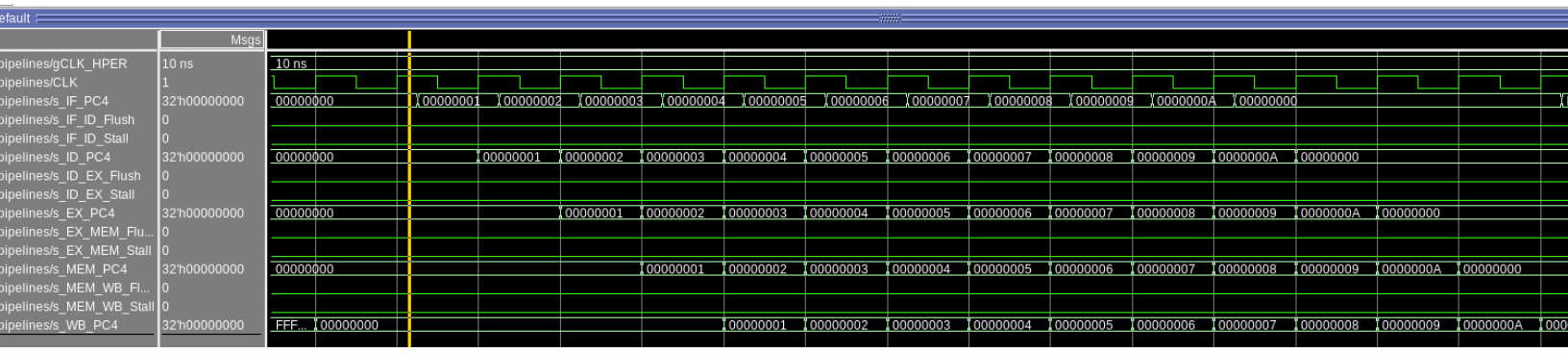
[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench should also test that each pipeline register can be individually stalled or flushed.
These first two screenshots are proof that the pipelines work individually.



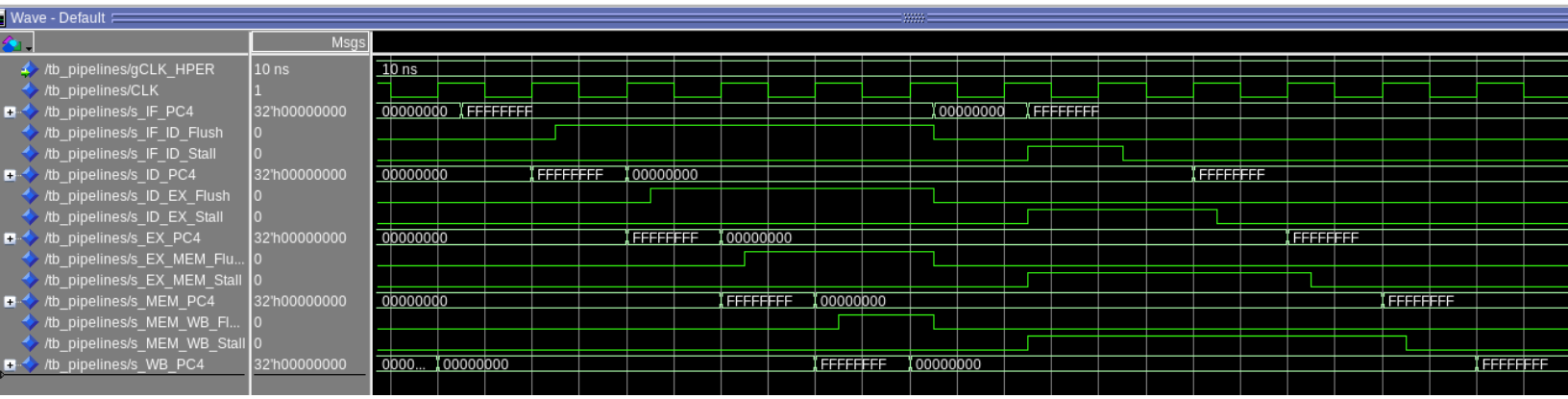




This last screenshot proves that a value written into the IF_PC4 value is available four cycles later. This next screenshot (the screenshot below) shows that values can be written into the pipeline each cycle and as you can see that each value is passed down the pipelines until they are no longer available.



This last screenshot (the screenshot below) shows that the values can be flushed in each pipeline individually, and that each pipeline can be individually stalled.



[2.b.i] list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to.

Instructions	Output Signals
add, add, addi, addiu, and, andi, lui, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sub, subu, sllv, srlv, and srav	s_EX_final, s_MEM_WD, s_WB_WD
lw, lh, lhu, lb, lbu	s_MEM_WB, s_WB_WD
jal	s_WB_WD

[2.b.ii] List which of these same instructions consume values, and what signals in the pipeline these correspond to.

Instructions	Input Signals
add, add, addi, addiu, and, andi, lui, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sub, subu, sllv, srlv, and srav	s_EX_RSA, s_EX_RSB, s_EX_ALUout, s_EX_ALUnAddSub, s_EX_ShiftArithmetic, s_EX_ShiftLorR, s_EX_Unsigned, s_EX_Lui , s_WB_RegWrite
lw,	s_MUX2OUT, s_MEM_lw, s_WB_RegWrite
jal	s_WB_RegWrite, s_WB_Link, s_EX_Jump,
sw	s_EX_RSA, s_MUXnumOUT, s_MEM_MemWrite

lh	s_signedHalfword, s_MEM_HoB, s_MEM_Sign, s_MEM_lw, s_WB_RegWrite
lhu	s_unsignedHalfword, s_MEM_HoB, s_MEM_Sign, s_MEM_lw, s_WB_RegWrite
lb	s_signedByte, s_MEM_HoB, s_MEM_Sign, s_MEM_lw, s_WB_RegWrite
lbu	s_unsignedByte, s_MEM_HoB, s_MEM_Sign, s_MEM_lw, s_WB_RegWrite
j	s_EX_Jump,
beq, bne	s_EX_Branch, s_EX_BranchNE, s_EX_BranchAddr
jr	s_EX_Return, s_EX_RSA

[2.b.iii] generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those that will require hazard stalls.

Data	A data hazard with a distance of three aka. An instruction is writing back to a register currently being read in the decode stage.	None	So, we will modify the reg file to output the new write data to either R1 or R2 if the write address is the same as RS or RT.
Data	A data hazard with a distance of two aka. An instruction is in memory while an instruction is being decoded.	Forward	So we are going to forward from writing back to the ALU input. We are waiting for a cycle so that if it is a load word instruction it has time to get the data from memory. We need to ensure we forward to the correct ALU input, either iA or iB, whether or not it was RS or RT.
Data	A data hazard with a distance of one that is not a load word instruction.	Forward	So, we will forward from the memory address input back to the input of the ALU.

			This has the same thing to consider as the previous hazard.
Data	A data hazard with a distance of one is a load word instruction.	Forward	We will forward from the output of mem.
Data	A data hazard with a distance of one or two	Forward	We will forward to the R2 value of the store word in execute rather than the RT value.

[2.b.iv] global list of the datapath values and control signals that are required during each pipeline stage

	IF			ID			EX			MEM			WB	
	Input	Output	IF / ID	Input	Output	ID / EX	Input	Output	EX / MEM	Input	Output	MEM / WB	Input	Output
	Jump/Branch/Return	PC+4 [31..0]	Ex	PC+4 [31..0]	PC+4 [31..0]		PC+4 [31..0]	Jump/Branch/Return		PC+4 [31..0]	PC+4 [31..0]		PC+4 [31..0]	
	New PC [31.00]	Instruction [31..0]		Instruction [31..0]	Instruction [31..0]		Instruction [31..0]	New PC [31..0]						
					s_Jump		s_Jump							
					s_branch		s_branch							
					s_Return		s_Return							
					s_ALUnAddSub		s_ALUnAddSub							
					s_ALUout [2..0]	Ex	s_ALUout [2..0]							
					s_ShiftLorR		s_ShiftLorR							
					s_ShiftArithemtic		s_ShiftArithemtic							
					s_unsigned		s_unsigned							
					s_Lui		s_Lui							
					s_BranchNE		s_BranchNE							
					s_A [31..0]		s_A [31..0]	s_Final [31..0]		s_Final [31..0]				
					s_B [31..0]		s_B [31..0]	s_B [31..0]		s_B [31..0]				
					s_Lw		s_Lw	s_Lw		s_Lw				
					s_R2		s_R2	s_R2		s_R2				
					s_HoB	MEM	s_HoB	s_HoB	MEM	s_HoB				
					s_Sign		s_Sign	s_Sign		s_Sign				
					s_MemWrite		s_MemWrite	s_MemWrite		s_MemWrite				
				s_WD [31..0]	s_MemtoReg		s_MemtoReg	s_MemtoReg		s_MemtoReg	s_WD [31..0]		s_WD [31..0]	s_WD [31..0]
				s_WA [4..0]	s_WA [4..0]		s_WA [4..0]	s_WA [4..0]		s_WA [4..0]	s_WA [4..0]		s_WA [4..0]	s_WA [4..0]
				s_RegWrite	s_RegWrite	WB	s_RegWrite	s_RegWrite	WB	s_RegWrite	s_RegWrite	WB	s_RegWrite	s_RegWrite
					s_Link		s_Link	s_Link		s_Link	s_Link		s_Link	
					s_Halt		s_Halt	s_Halt		s_Halt	s_Halt		s_Halt	

Most of the signals remain the same for the new pipeline with new ones for the Stalling Unit, Forwarding Unit, and Branch Preditor

Unit	Inputs	Outputs
Stalling Unit	s_EX_Jump s_branch s_branchChoice s_branch_pred	s_flush_IF_ID s_flush_ID_EX

	s_notBranch_pred s_MEM_MemToReg s_MEM_WA s_EX_INSTRUCTION(25 downto 21) (RS) s_EX_INSTRUCTION(20 downto 16) (RT)	
Forwarding Unit	iCLK iRST s_WB_WA s_MEM_WA s_EX_INSTRUCTION(25 downto 21) s_EX_INSTRUCTION(20 downto 16) s_EX_INSTRUCTION(31 downto 26) s_MEM_RegWrite s_WB_RegWrite	s_WB_EX2_RS s_WB_EX2_RT s_WB_EX2_R2 s_MEM_EX1_RS s_MEM_EX1_RT s_MEM_EX1_R2 s_MEM_EX1_RSLW s_MEM_EX1_RTLW s_MEM_EX1_R2LW
Branch Predictor *(For more information on the branch predictor go to the end of this document)	iCLK iRST s_branchChoice s_branch	s_branch_pred s_notBranch_pred

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

Instruction:	Stage
beq	ID
bne	ID
j	EX
jal	EX
jr	EX

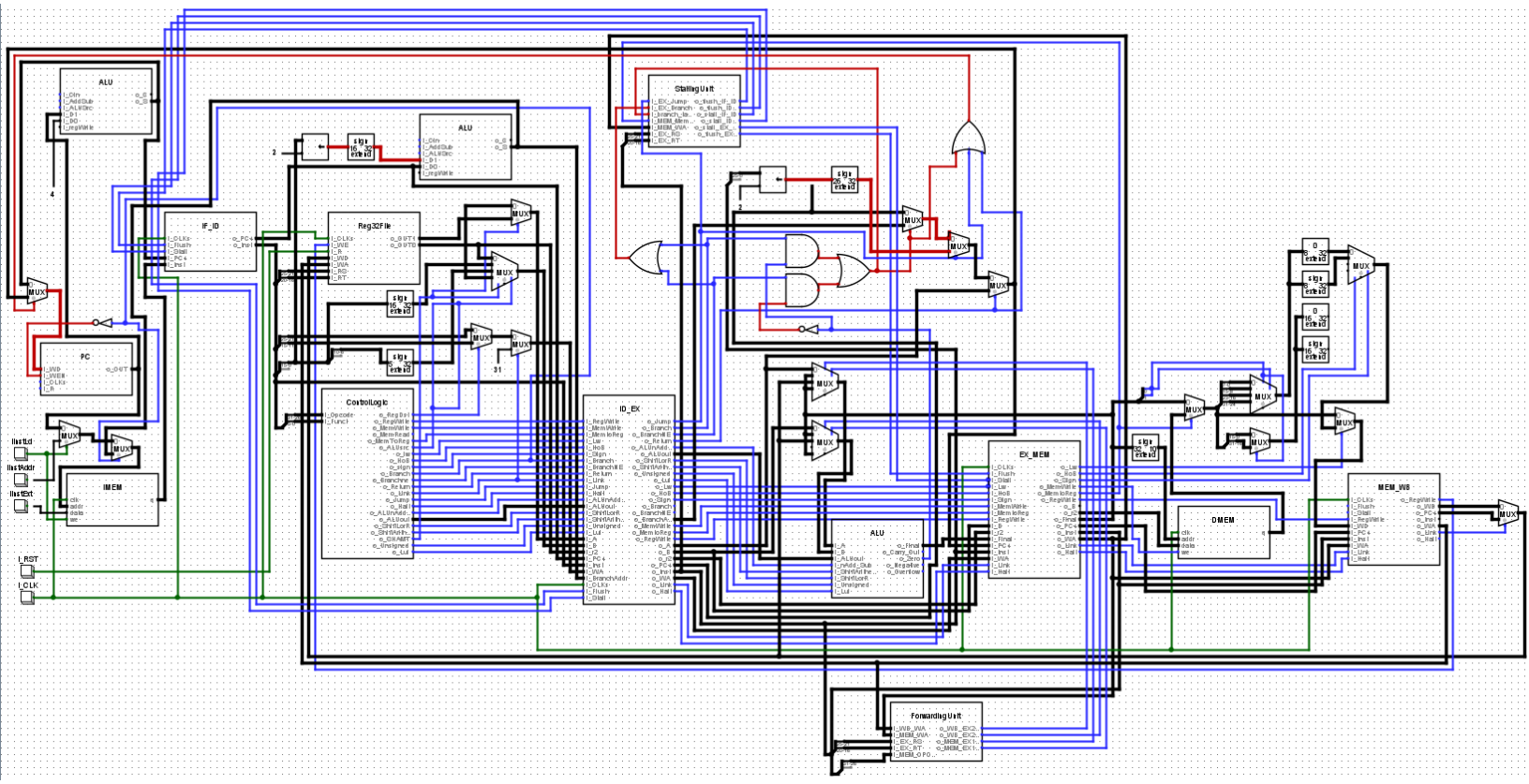
[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.

Instruction:	IF_ID	ID_EX	EX_MEM	MEM_WB
beq (in EX)	flush*	flush*	n/a	n/a
bne (in EX)	flush*	flush*	n/a	n/a
j (in EX)	flush	flush	n/a	n/a
jal (in EX)	flush	flush	n/a	n/a
jr (in EX)	flush	flush	n/a	n/a

* this is because we will put the pc4 following then once the branch reaches ID we will change the PC value to be changed to the new pc value. so this will force us to flush a single instruction

every time rather than needing to flush two instructions sometimes… so the asterisk shows that the pipeline could be flushed (it just depends on if the branch is taken or not)

[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.



[2.e – i, ii, and iii] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

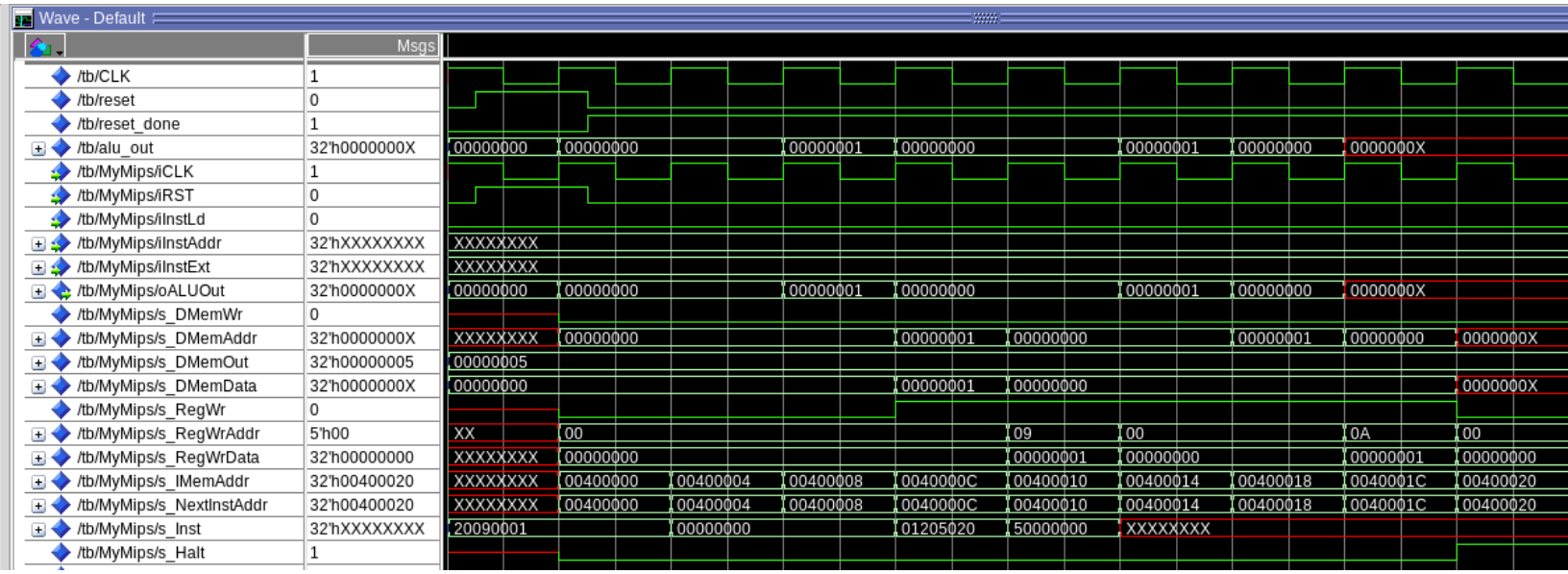
Type of Hazard	Description of Hazard	Stall or Forward or Flush	Description of Solution
Control	This hazard is caused by a branch instruction being used in the hardware pipeline. The processor has yet to determine if the branch will be taken. So it has two cycles in the pipeline to fill while waiting for the branch to hit the execution stage. (Branch)	Flush	So we will fetch the next instruction in the program order so PC+4. Then the next instruction we will load in will be the instruction following the branch address. This way, we only have to flush one instruction. While it means we will always flush an instruction.
Control	Any Jump instruction will cause this hazard. The processor only knows the jump address once it gets calculated and returned in the execute stage. (Jump)	Stall, Stall	So, we will stall for 2 cycles so that jump, jump link, or jump return can be calculated properly in the execute stage. We need to disable updating the pc during the two NOP cycles.
Data	A data hazard with a distance of two aka. An instruction is in memory while an instruction is being decoded.	Forward	So we are going to forward from writing back to the ALU input. We are waiting for a cycle so that if it is a load word instruction it has time to get the data from memory. We need to ensure we forward to the correct ALU input, either iA or iB, whether or not it was RS or RT.
Data	A data hazard with a distance of one that is not a load word instruction.	Forward	So, we will forward from the memory address input back to the input of the ALU. This has the same thing to consider as the previous hazard.
Data	A data hazard with a distance of one is a load word instruction.	Stall, Forward	So we will stop the entire pipeline in its tracks and add a nop between the load word and the other instruction so that the data can be collected from memory. Then it can be forwarded once the load word instruction is in write back.

Test Cases:			
Control Hazards	Halt		
	J		
	Jal Jr		
	Branch (not taken)		
	Branch (taken)		
Data Hazards	Distance 3		RS read EX write
	Distance 2		RT read EX write
	Distance 1		RS read MEM write
	Distance 1 (lw)		RT read MEM write
	Distance 3 (writing to $zero)		RS read EX write (writing to $zero)
	Distance 2 (writing to $zero)		RT read EX write (writing to $zero)
	Distance 1 (writing to $zero)		RS read MEM write (writing to $zero)
	Distance 1 (lw) (writing to $zero)		RT read MEM write (writing to $zero)

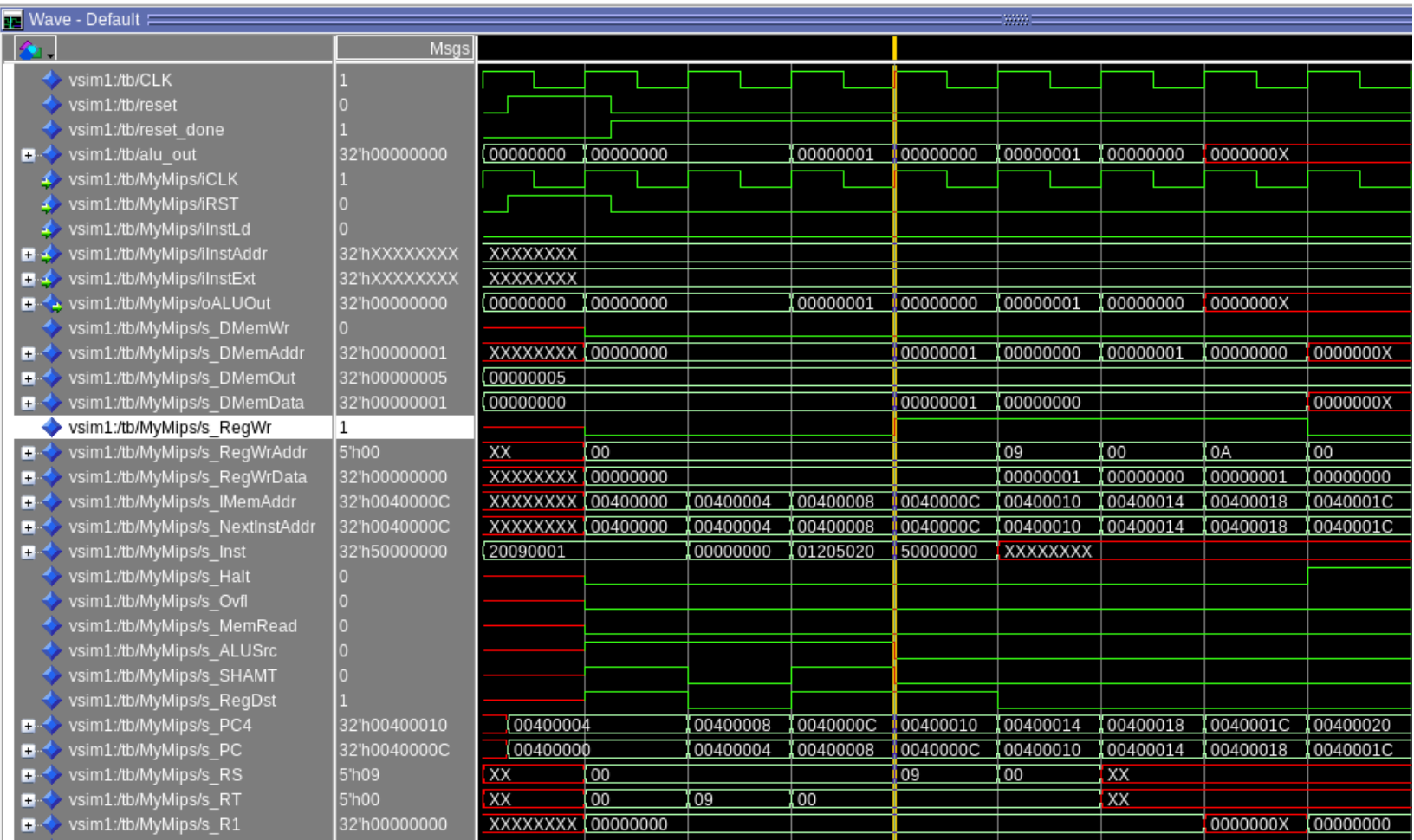
Here are all the test cases we wrote for the forwarding and stalling unit we implemented. We believe this accounts for every possible hazard situation. (There is one not listed in the table a WB hazard where the value has not been writing into the reg file yet, however this has been accounted for because we have the register on a negative edge trigger now.)

[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

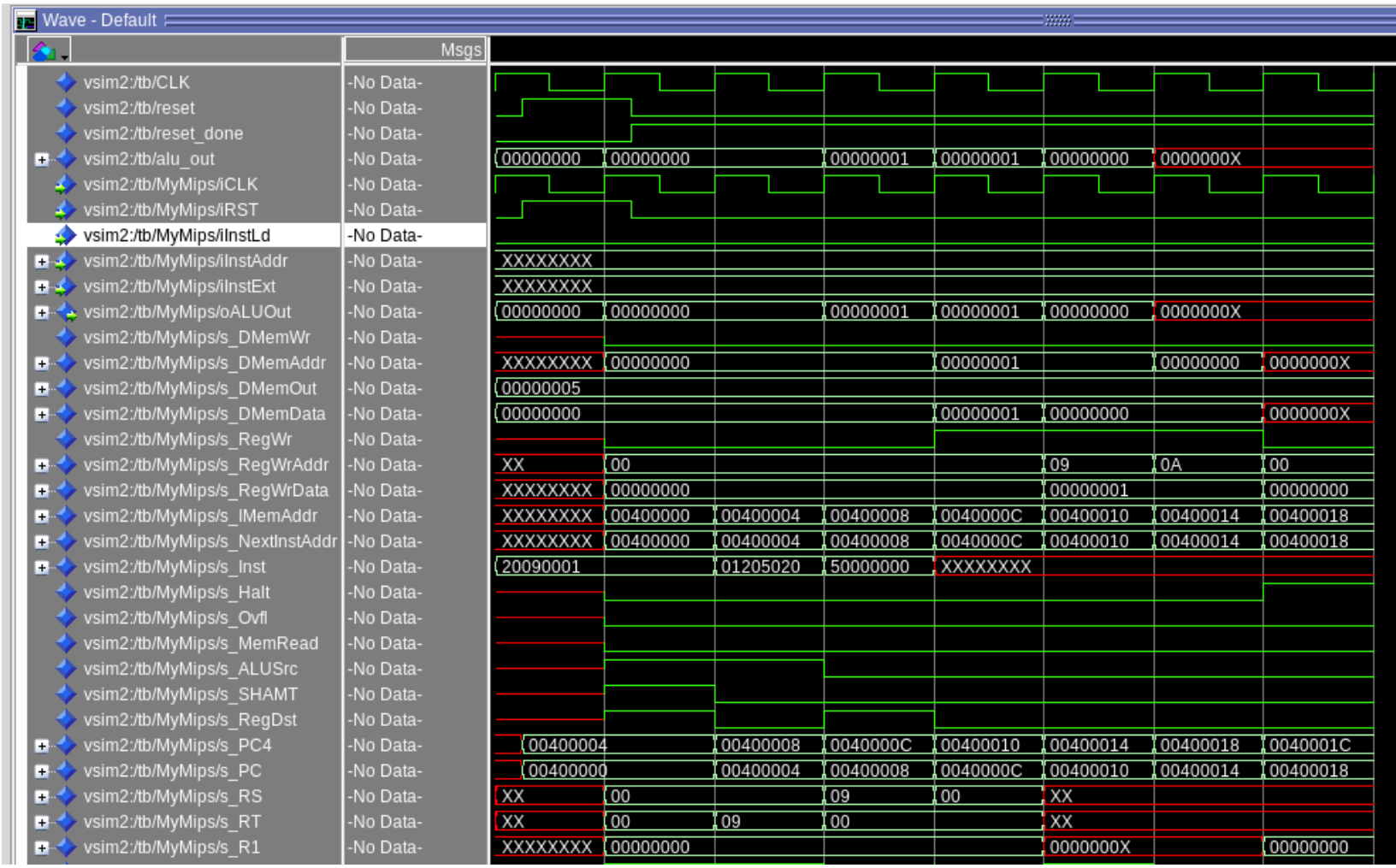
See attached google spreadsheet. Here is the data hazard test cases:



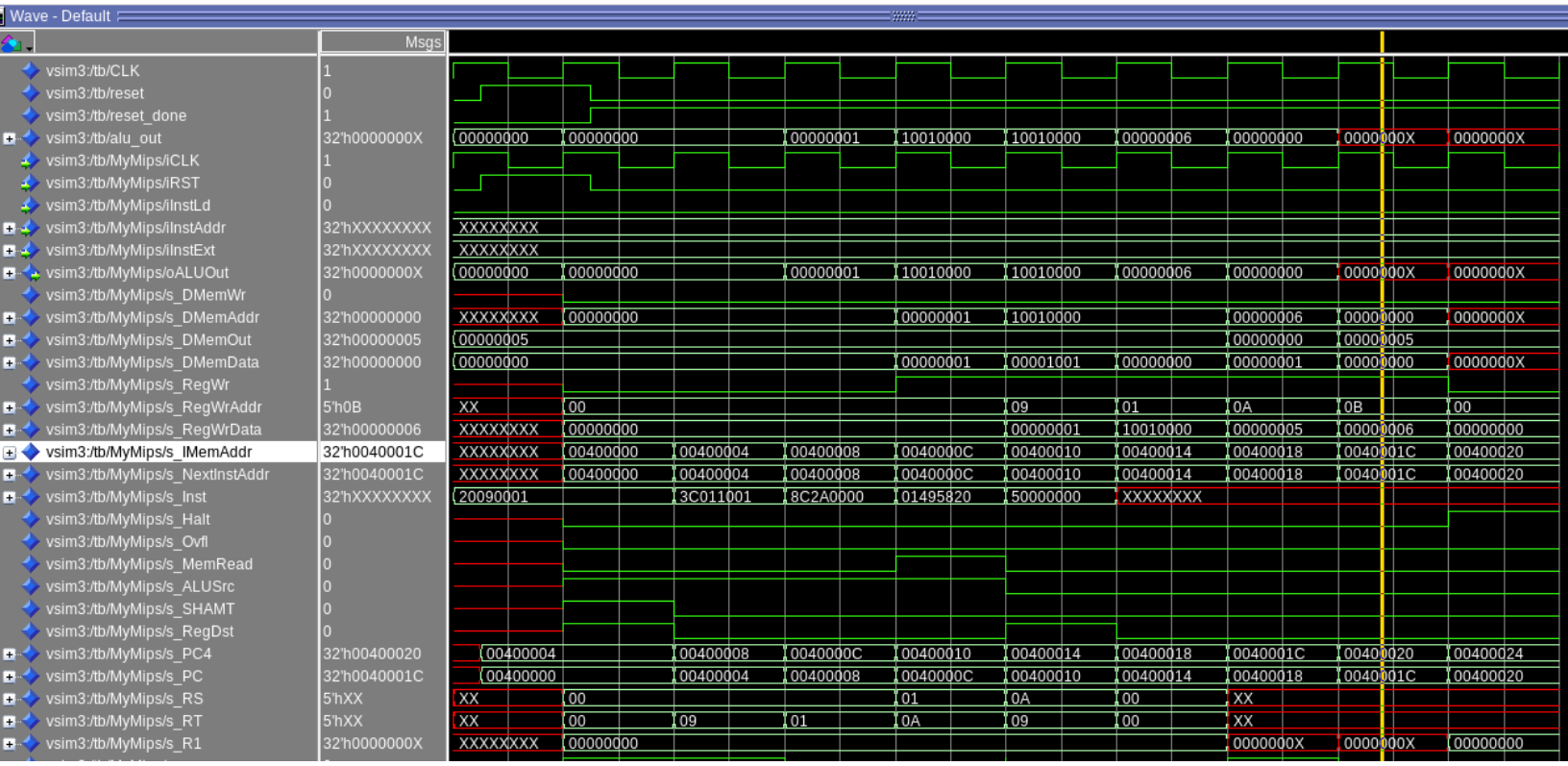
This screenshot (the one above) is the distance3.s test program working… This is correct because we can verify that the value sent into the alu and the register are the correct values. I wrote and included nops to verify that the forwarding from each state was working correctly.



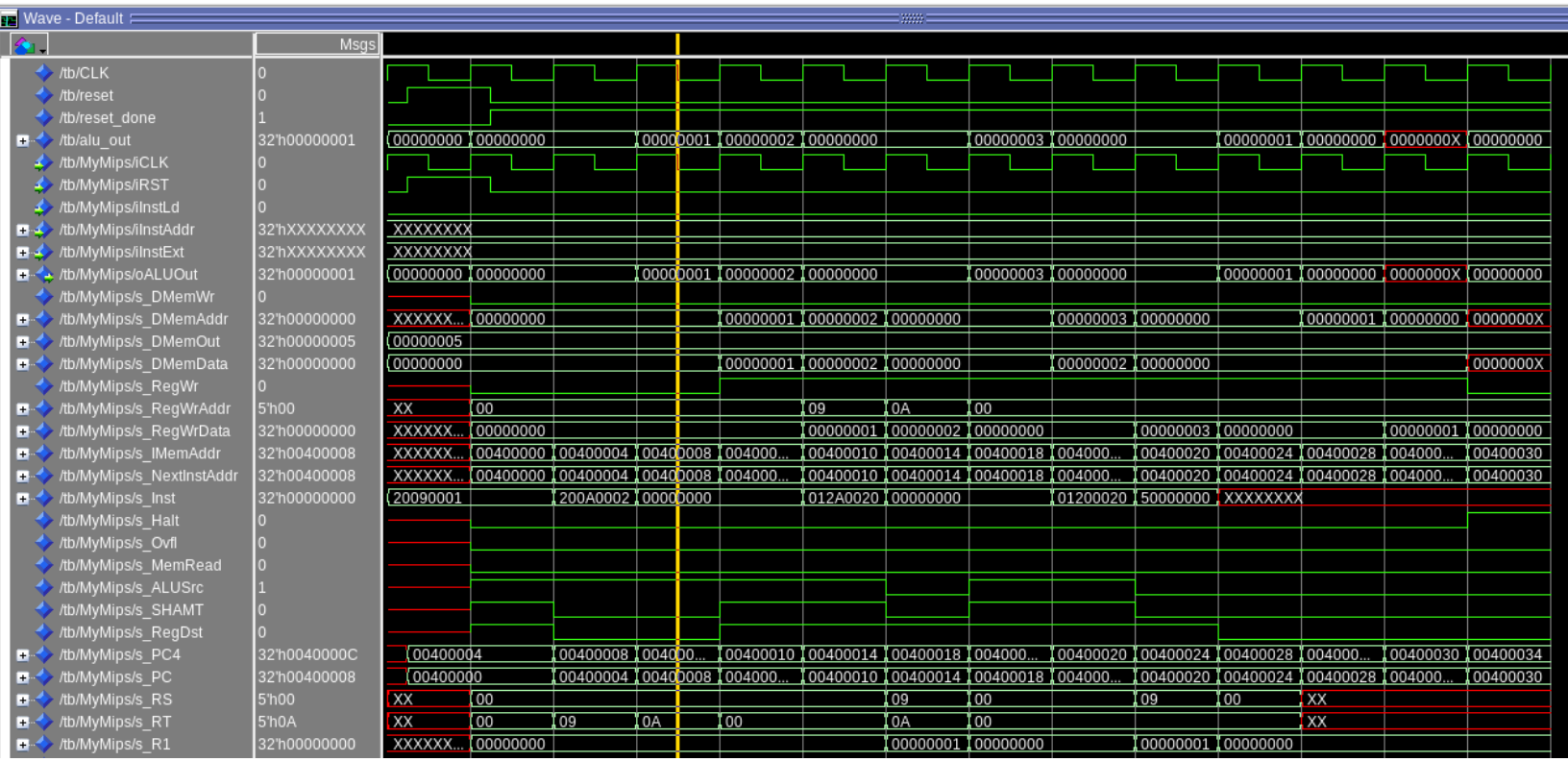
This screenshot (the one above) is the distance2.s test program working because the value from the previous instruction with a distance of 2 with a nop included just to focus on testing a forward with this distance between the instructions. We can see that the forward worked because the ALU has the correct value



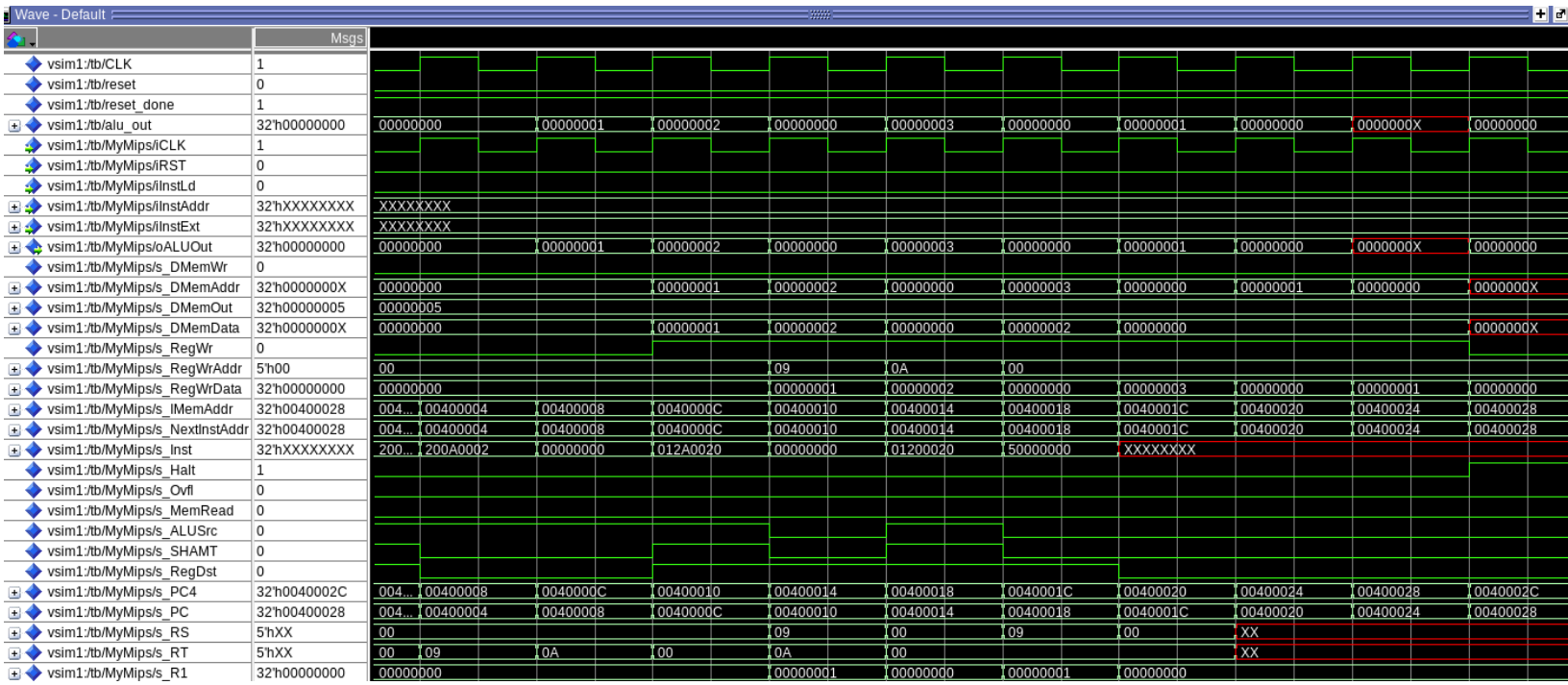
This screenshot (the one above) is the distance1.s test program working. This is correct because we can see no nops and a proper forward from the instruction down to the next data dependency. The RegWr, RegWrAddr, and RegWrData are all correct for each of the instructions



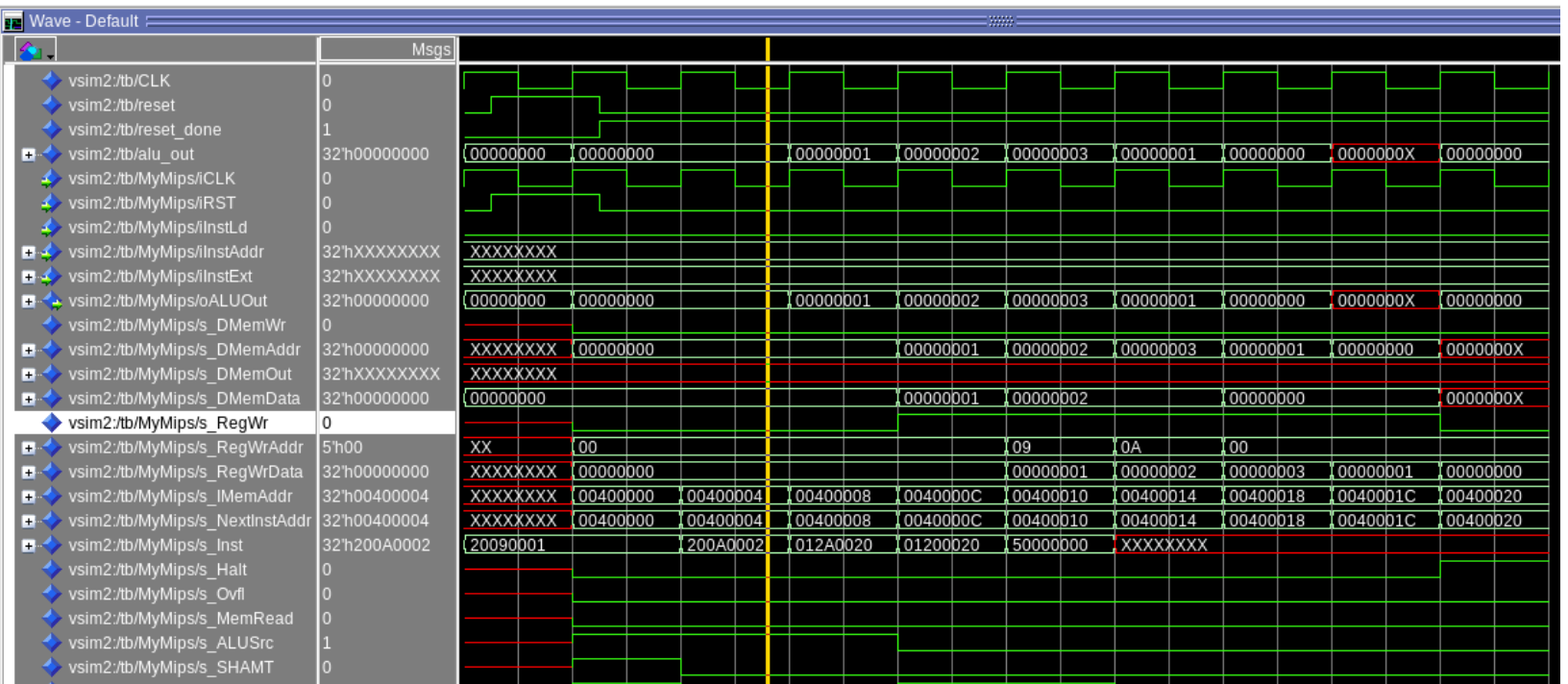
This screenshot (the one above) is the distance1_lw.s test program working. This is correct because we can see a stall was properly inserted and the ALU values are matching correctly with the RegWrData, and Addr as intended



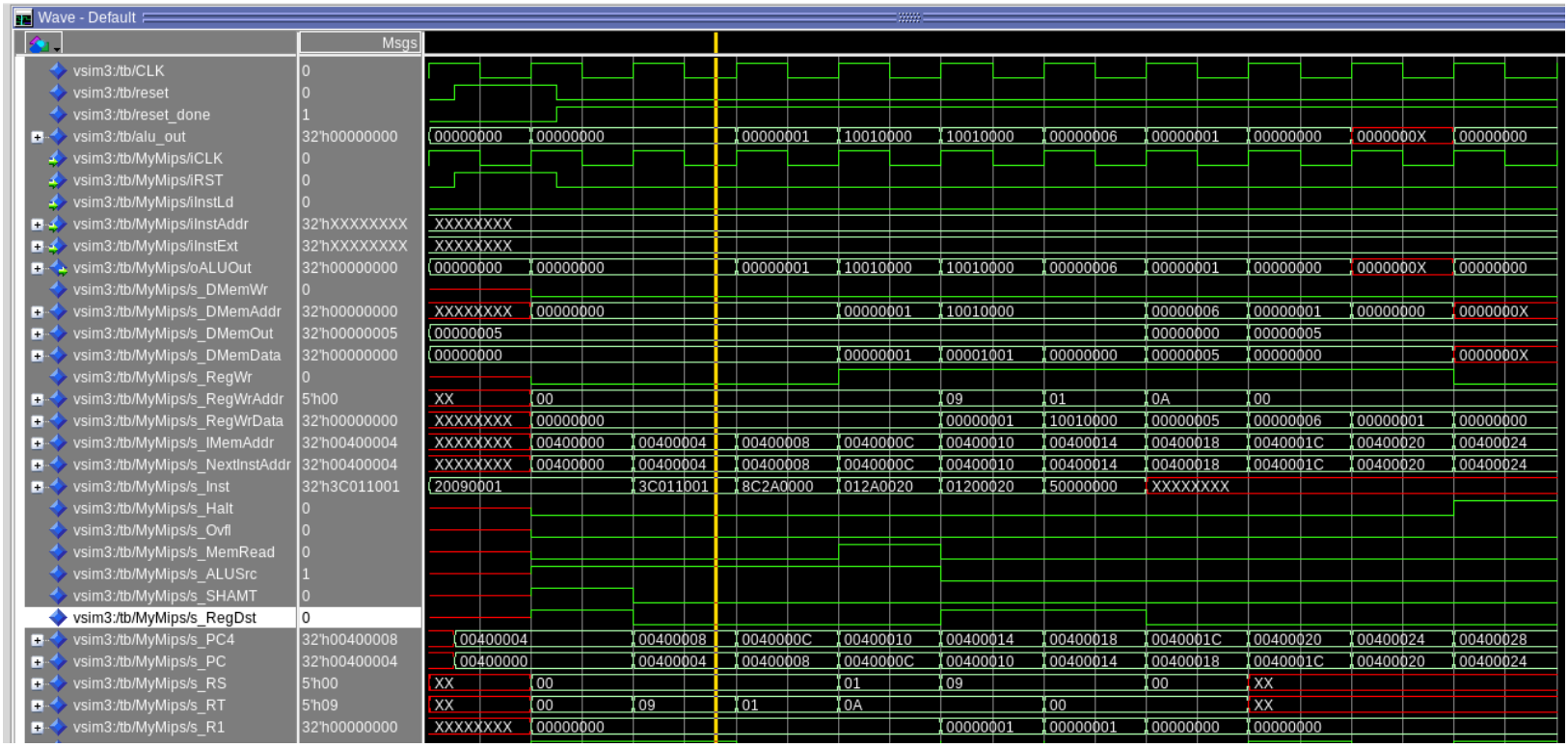
This screenshot (the one above) is the distance3_zero.s test program working. Here we take a step back to check forwarding with the 0th register and make sure no value written into the zero register can get forwarded. Here we see that the value thats to be written into the 0 register isnt forwarded and used later



This screenshot (the one above) is the distance2_zero.s test program working. Again here we have correctness because the value to be written into the 0th register is that we are avoiding a forwarding hazard because RegWrData is properly zero when a forward could forward a 0th register hazard



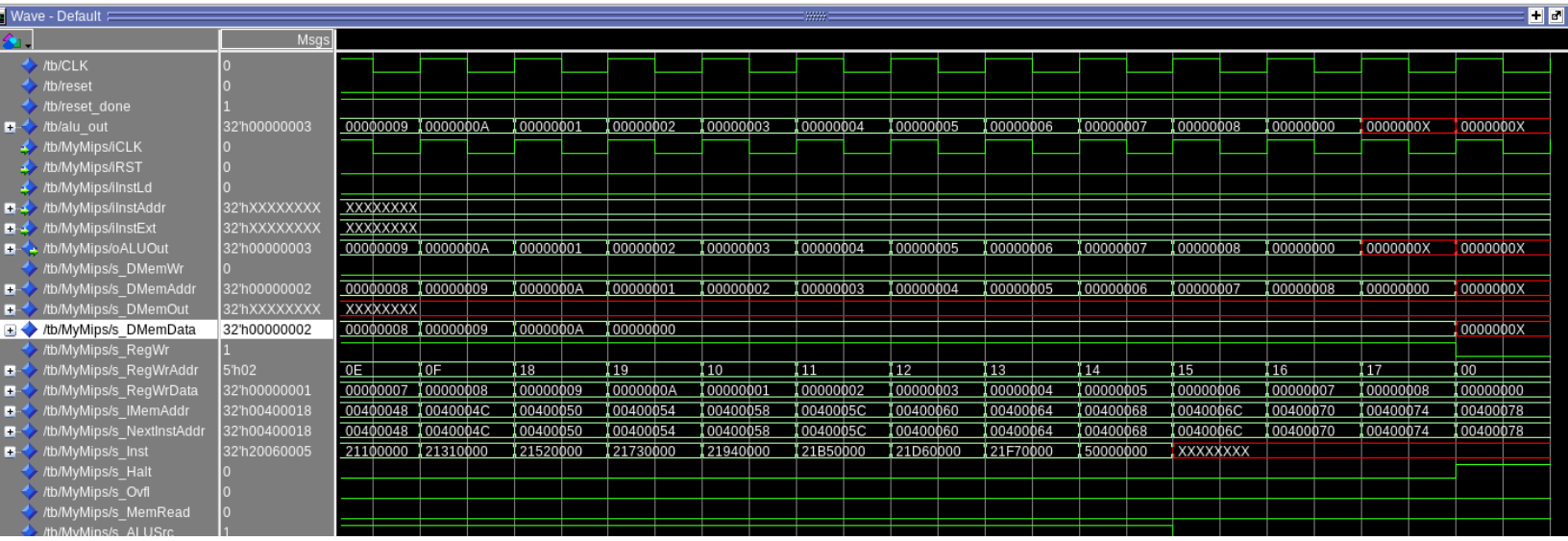
This screenshot (the one above) is the distance1_zero.s test program working we have the same thing as before where the value to be written from the 0th register doesnt get forwarded into another instruction we can check this by looking at the Addr destination for the instruction



This screenshot (the one above) is the distance1_lw_zero.s test program working. We can see that is working and is correct because the RegWr, RegWrAddr, and RegWrData contain the correct values, where no 0th register is written. We can also see a stall is implemented because of teh LW instruction.

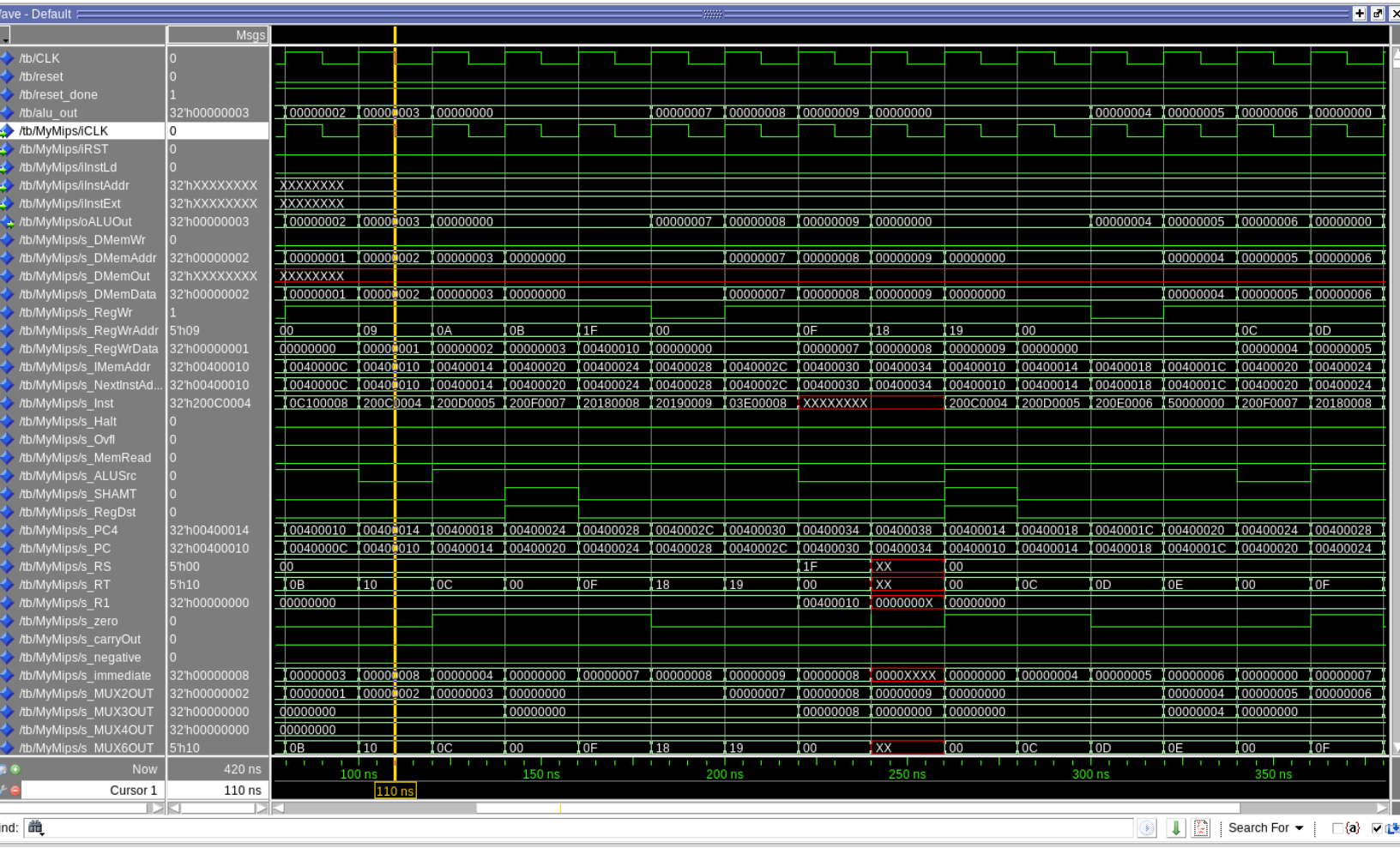
[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

See attached google spreadsheet. Here is the control hazard test cases:

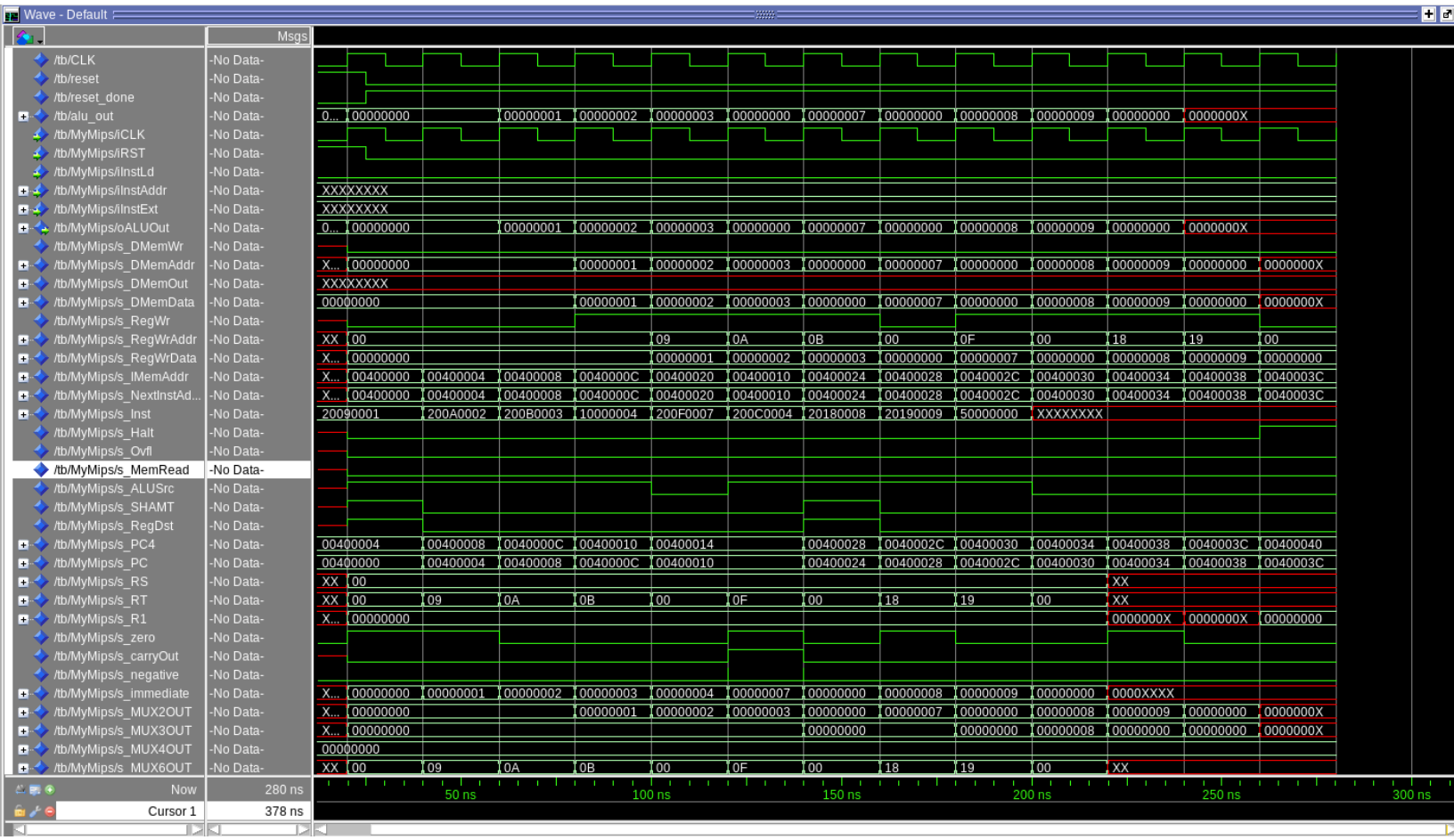


This is the screenshot of the Halt.s code. This is correct because no instructions are skipped. we write to register $s7 the correct value of 8 which was in $t7 and this is properly written before the program halts.

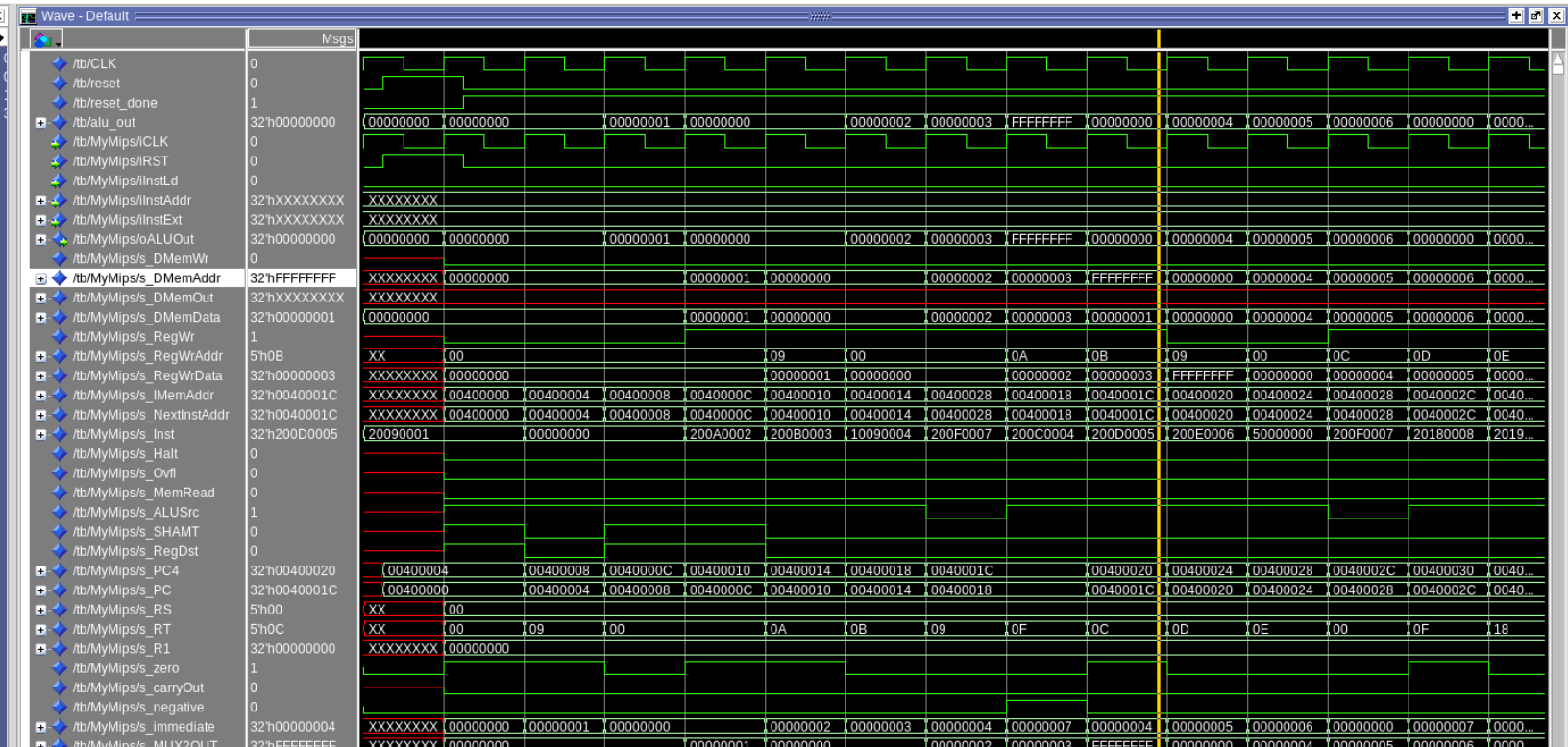
This is the screenshot of the J.s code, here it is correct because the idea of the process is to skip writing into registers $t4, $t5, and $t6… if the control logic is working properly it should write into those registers. As we can see the DMemAddr never gets set to those registers and skips to registers $t7, $t8, and $t9 properly.



This is the screenshot for the Jal_Jr.s control hazard test program. This works because no instruction is executed more than once and the processor successfully executes the instructions in the correct order. Because looking at the ALU and the DMemAddr, $t7, $t8, and $t9 are written before $t4, $t5, and $t6 are written into.



This is the Branch_taken.s test program this is correct because it doesn't run any extra instructions and successfully flushes one of the instructions in between the 7 and 8 alu out. this is correct because we load in the jumpaddr and then pc+4… in this case the pc+4 is properly flushed.



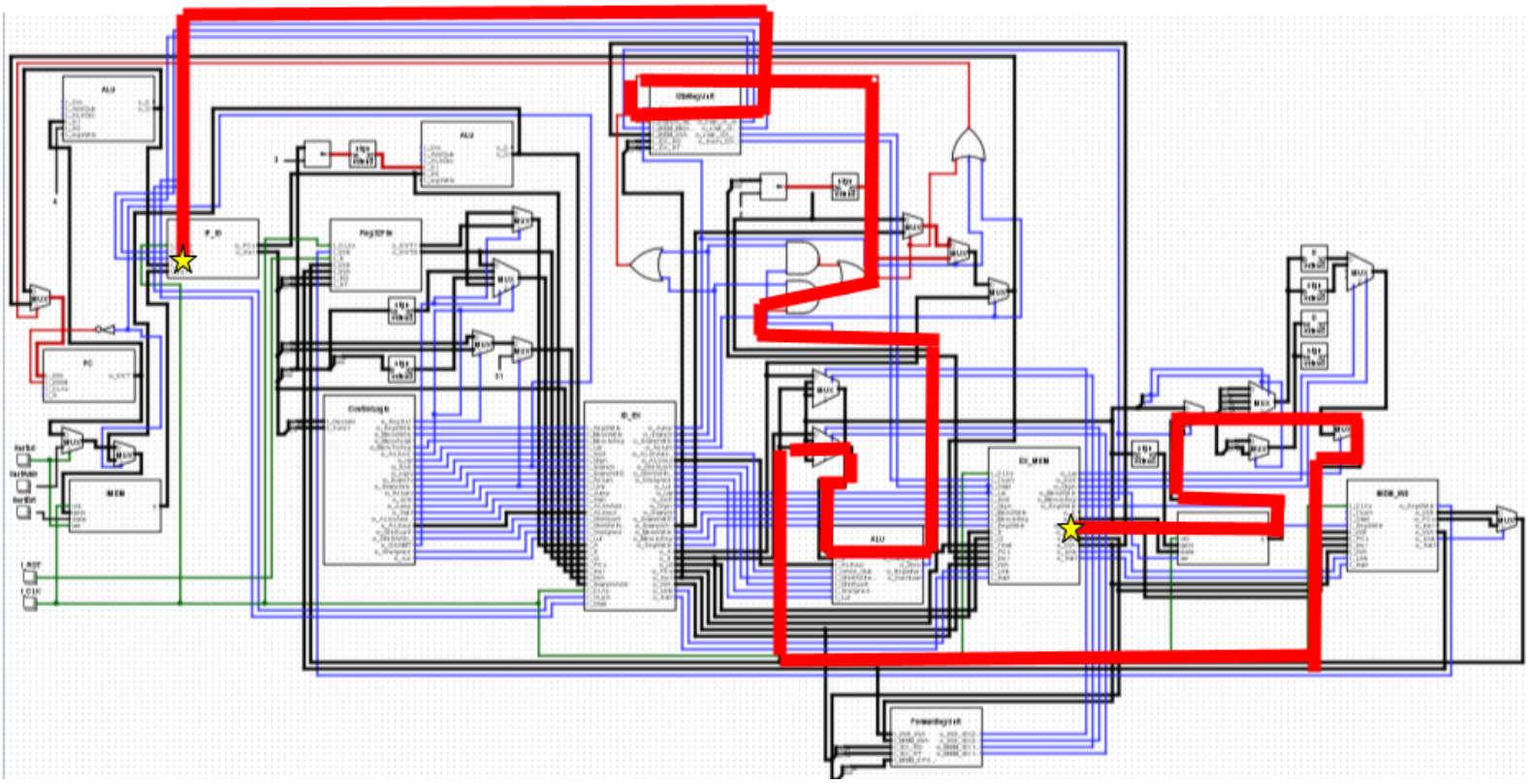
Above is the screenshot of the Branch_not_taken.s, this is correct because it successfully write the correct values into all of the registers and never writes into $t7, $t8, and $t9 registers. It doesnt jump and the RegWrData value is 0 so the value FFFF_FFFF isnt written into the register correctly.

[2.f] report the maximum frequency your hardware-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

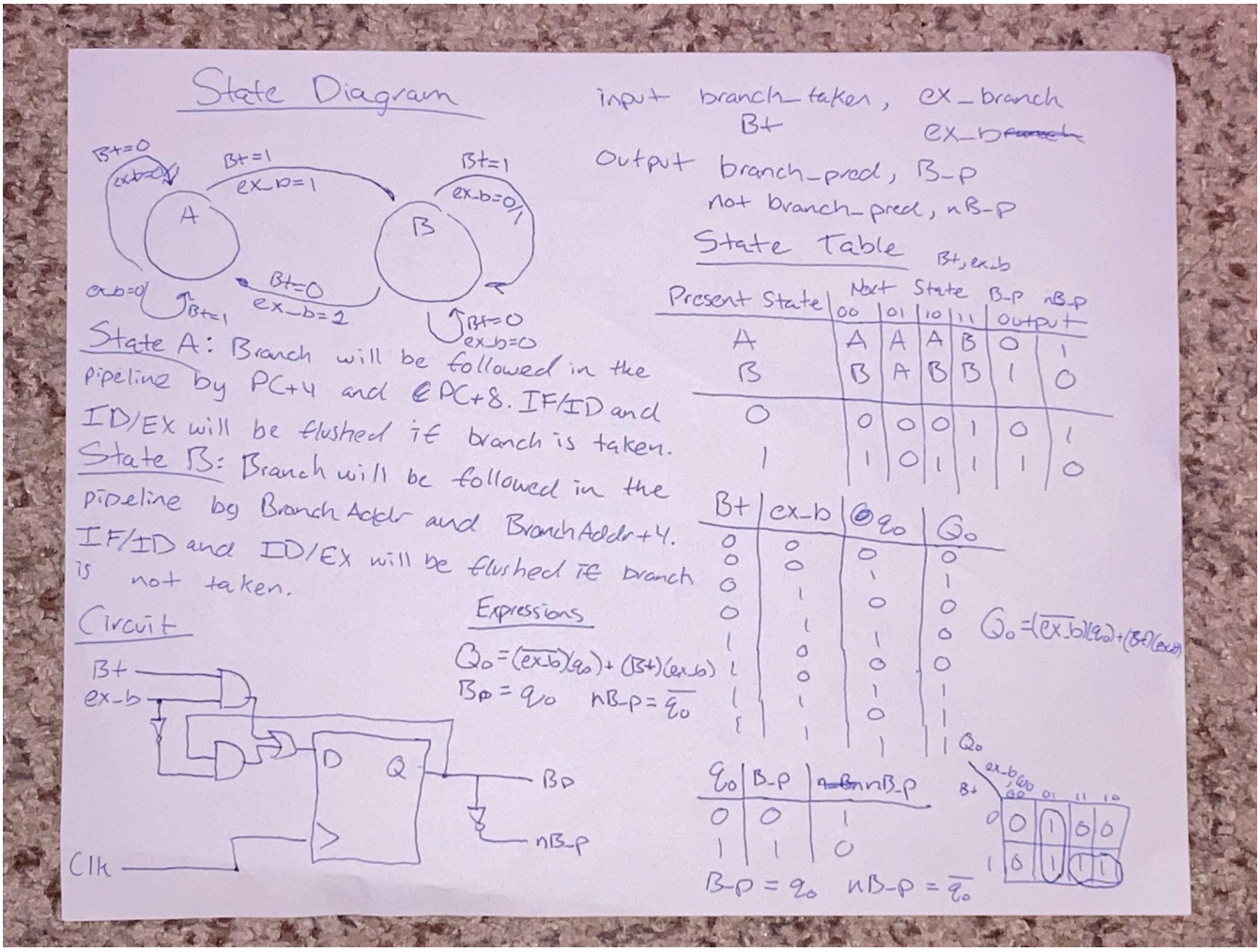
32.09mhz is the maximum frequency for the hw processor with a branch predictor

```
#
# CprE 381 toolflow Timing dump
#

FMax: 32.09mhz Clk Constraint: 20.00ns Slack: -11.17ns

The path is given below

 ================================================================
 From Node    : EX_MEM:EXMEM|RegFileSync:REG2|dffgSync:\G_NBit_REG:4:DFF|s_Q
 To Node      : IF_ID:IFID|RegFileSync:REG1|dffgSync:\G_NBit_REG:20:DFF|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ==================================
      0.000     0.000           launch edge time
      3.063     3.063  R        clock network delay
      3.295     0.232     uTco  EX_MEM:EXMEM|RegFileSync:REG2|dffgSync:\G_NBit_REG:4:DFF|s_Q
      3.295     0.000 FF  CELL  EXMEM|REG2|\G_NBit_REG:4:DFF|s_Q|q
      5.176     1.881 FF    IC  DMem|ram~39514|dataa
      5.600     0.424 FF  CELL  DMem|ram~39514|combout
      7.203     1.603 FF    IC  DMem|ram~39515|datad
      7.353     0.150 FR  CELL  DMem|ram~39515|combout
      7.558     0.205 RR    IC  DMem|ram~39518|datad
      7.713     0.155 RR  CELL  DMem|ram~39518|combout
      9.525     1.812 RR    IC  DMem|ram~39521|datab
      9.903     0.378 RF  CELL  DMem|ram~39521|combout
     10.172     0.269 FF    IC  DMem|ram~39532|datab
     10.597     0.425 FF  CELL  DMem|ram~39532|combout
     10.829     0.232 FF    IC  DMem|ram~39543|datac
     11.110     0.281 FF  CELL  DMem|ram~39543|combout
     11.335     0.225 FF    IC  DMem|ram~39586|datad
     11.460     0.125 FF  CELL  DMem|ram~39586|combout
     11.691     0.231 FF    IC  DMem|ram~39629|datac
     11.972     0.281 FF  CELL  DMem|ram~39629|combout
     13.598     1.626 FF    IC  DMem|ram~39630|datac
     13.879     0.281 FF  CELL  DMem|ram~39630|combout
     14.104     0.225 FF    IC  \MUX12_32:0:MUX12|o_O~0|datad
     14.229     0.125 FF  CELL  \MUX12_32:0:MUX12|o_O~0|combout
     14.499     0.270 FF    IC  \MUX4_32:0:MUX4|o_O~1|datab
     14.924     0.425 FF  CELL  \MUX4_32:0:MUX4|o_O~1|combout
     15.154     0.230 FF    IC  \MUX4_32:0:MUX4|o_O~2|datad
     15.279     0.125 FF  CELL  \MUX4_32:0:MUX4|o_O~2|combout
     15.991     0.712 FF    IC  \RTBLW:0:MUXWBRT|o_O~2|datad
     16.116     0.125 FF  CELL  \RTBLW:0:MUXWBRT|o_O~2|combout
     16.398     0.282 FF    IC  A|adder_subtractor|Nbit_Adder|full_adder_0|g_And1_And2_to_Or|o_C~0|datac
     16.659     0.261 FR  CELL  A|adder_subtractor|Nbit_Adder|full_adder_0|g_And1_And2_to_Or|o_C~0|combout
     16.886     0.227 RR    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:1:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
     17.173     0.287 RR  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:1:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     17.396     0.223 RR    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:2:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
     17.683     0.287 RR  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:2:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     17.908     0.225 RR    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:3:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     18.063     0.155 RR  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:3:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     18.292     0.229 RR    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:4:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     18.447     0.155 RR  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:4:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     18.671     0.224 RR    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:5:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
     18.958     0.287 RR  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:5:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
     19.186     0.228 RR    IC  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:6:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
     19.341     0.155 RR  CELL  A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:6:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
```

18.958	0.287	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:5:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
19.186	0.228	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:6:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
19.341	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:6:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
19.569	0.228	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:7:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
19.724	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:7:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
19.950	0.226	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:8:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
20.105	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:8:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
20.330	0.225	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:9:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
20.617	0.287	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:9:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
20.844	0.227	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:10:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
20.999	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:10:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
21.226	0.227	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:11:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
21.381	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:11:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
21.609	0.228	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:12:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
21.764	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:12:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
21.992	0.228	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:13:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
22.147	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:13:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
22.374	0.227	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:14:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
22.529	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:14:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
22.931	0.402	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:15:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
23.086	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:15:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
23.314	0.228	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:16:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
23.469	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:16:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
23.698	0.229	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:17:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
23.853	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:17:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
24.081	0.228	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:18:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
24.236	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:18:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
24.446	0.210	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:19:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
24.601	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:19:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
24.815	0.214	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:20:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
24.970	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:20:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
25.194	0.224	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:21:rippleAdder|g_And1_And2_to_Or|o_C~0|datac
25.481	0.287	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:21:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
25.694	0.213	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:22:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
25.849	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:22:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
26.076	0.227	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:23:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
26.231	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:23:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
26.442	0.211	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:24:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
26.597	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:24:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
26.824	0.227	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:25:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
26.979	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:25:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
27.404	0.425	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:26:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
27.559	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:26:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
27.771	0.212	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:27:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
27.926	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:27:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
28.687	0.761	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:28:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
28.842	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:28:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
29.054	0.212	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:29:rippleAdder|g_And1_And2_to_Or|o_C~0|datad
29.209	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:29:rippleAdder|g_And1_And2_to_Or|o_C~0|combout
29.418	0.209	RR	IC	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:30:rippleAdder|g_XOR_Cin_to_XOR|o_C|datad
29.573	0.155	RR	CELL	A|adder_subtractor|Nbit_Adder|\G_NBit_full_adder:30:rippleAdder|g_XOR_Cin_to_XOR|o_C|combout
30.498	0.925	RR	IC	A|branchchecker|Equal0~0|datad
30.637	0.139	RF	CELL	A|branchchecker|Equal0~0|combout
30.869	0.232	FF	IC	FETCH|or2a|o_C~3|datac
31.149	0.280	FF	CELL	FETCH|or2a|o_C~3|combout
31.433	0.284	FF	IC	FLUSH_ID|o_C~1|datad
31.558	0.125	FF	CELL	FLUSH_ID|o_C~1|combout
33.998	2.440	FF	IC	IFID|REG1|\G_NBit_REG:20:DFF|s_Q|sclr
34.578	0.580	FR	CELL	IF_ID:IFID|RegFileSync:REG1|dffgSync:\G_NBit_REG:20:DFF|s_Q



BRANCH PREDICTOR



This is the logic I used for the branch predictor. It is a two-state Moore Machine with an enable bit because we do not want to update the state machine unless there is currently a branch in execution.

There are two outputs (branch_pred and notBranch_pred). These are used to control muxes in the main processor file.

When branch_pred is 0, and notBranch_pred is 1. This is state A, which will choose PC +4 and PC+8 to fetch after the branch instruction. This will flush IF/ID and ID/EX if the branch is taken and will move to state B.

When branch_pred is 1, and notBranch_pred is 0. This is state B, which will choose branch +4 and branch +8 to fetch after the branch instruction. This will flush IF/ID and ID/EX if the branch is not taken and will move to state A.