# vsim:/tb/MyMips/HazardUnitNew/i_branchEx
# CprE 381: Computer Organization and Assembly-Level Programming

## Project Part 2 Report

Team Members: Jack Sebahar

Peter Wissman

John Lavigne

Project Teams Group #: 1

***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.

IF/ID stage:
- s_Clk
- s_Flush
- s_Stall
- s_Inst
- s_PC

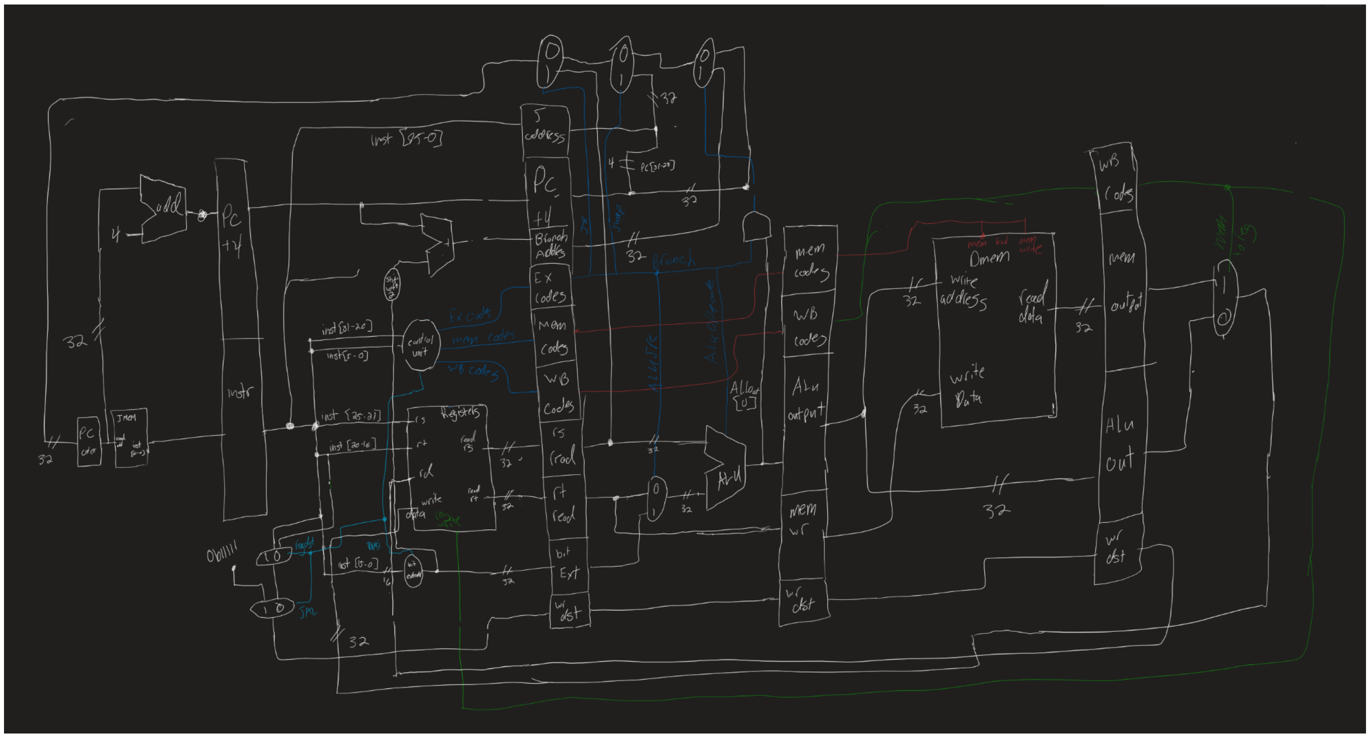
ID/EX stage:
- s_Clk
- s_RST
- s_Stall
- s_ALUSrcSel
- s_ALUOpCode
- s_regDst
- s_MemWrEn
- s_branchSel
- s_memToRegSel
- s_regRsIn
- s_regRtIn
- s_RegWrAddr
- s_JaloDataWrite
- s_regRsOut
- s_regRtOut

- s_immMuxOut
- s_jumpAddr
- s_branchAddr
- s_jumpSel
- s_jumpRegSel
- s_jalSel
- s_memReadSel
- s_regWr
- s_PC
- s_Halt

EX /MEM:
- s_CLK
- s_RST
- s_readData2
- s_aluOut
- s_writeReg
- s_overflow
- s_MemtoReg
- s_weMem
- s_weReg
- s_DMemRead
- s_halt

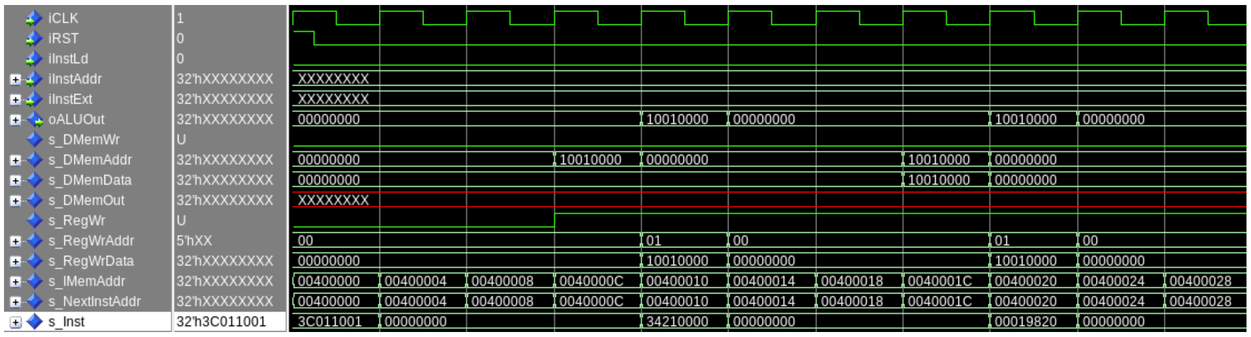
MEM/WB
- s_CLK
- s_RST
- s_memReadData
- s_aluOut
- s_writeReg
- s_overflow
- s_MemtoReg
- s_weReg
- s_halt

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



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

The program tested here is software_test1. It uses every instruction ran by our processor with instructions arranged so that it avoids any hazards. It also uses nops occasionally if the aforementioned strategy does not work. The test was successful.

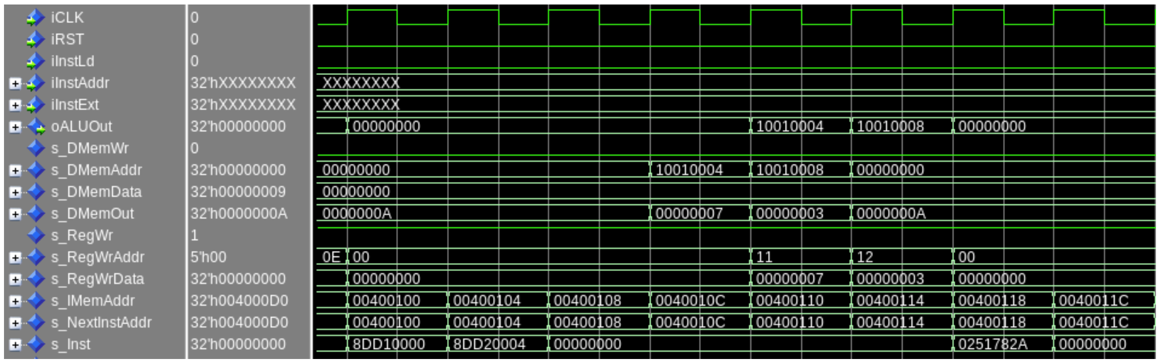


```
iCLK            1
iRST            0
iInstLd         0
iInstAddr       32'hXXXXXXXX   XXXXXXXX
iInstExt        32'hXXXXXXXX   XXXXXXXX
oALUOut         32'hXXXXXXXX   00000000                                     10010000  00000000                                     10010000  00000000
s_DMemWr        U
s_DMemAddr      32'hXXXXXXXX   00000000                           10010000  00000000                           10010000  00000000
s_DMemData      32'hXXXXXXXX   00000000                                                                        10010000  00000000
s_DMemOut       32'hXXXXXXXX   XXXXXXXX
s_RegWr         U
s_RegWrAddr     5'hXX          00                                           01        00                                           01        00
s_RegWrData     32'hXXXXXXXX   00000000                                     10010000  00000000                                     10010000  00000000
s_IMemAddr      32'hXXXXXXXX   00400000  00400004  00400008  0040000C  00400010  00400014  00400018  0040001C  00400020  00400024  00400028
s_NextInstAddr  32'hXXXXXXXX   00400000  00400004  00400008  0040000C  00400010  00400014  00400018  0040001C  00400020  00400024  00400028
s_Inst          32'h3C011001   3C011001  00000000                           34210000  00000000                           00019820  00000000
```

The "s_Inst" signal at the very bottom contains the instruction being loaded. As you can see, the first instruction "3C011001" gets loaded into the IF/ID stage in the first clock cycle. The next instruction was a stall, “00000000”, which gets loaded into the IF/ID stage in the next cycle as the previous instruction moves to the next stage. You can see when an instruction makes it through the pipeline 4 cycles later, when the signal “s_RegWrAddr” changes its value.

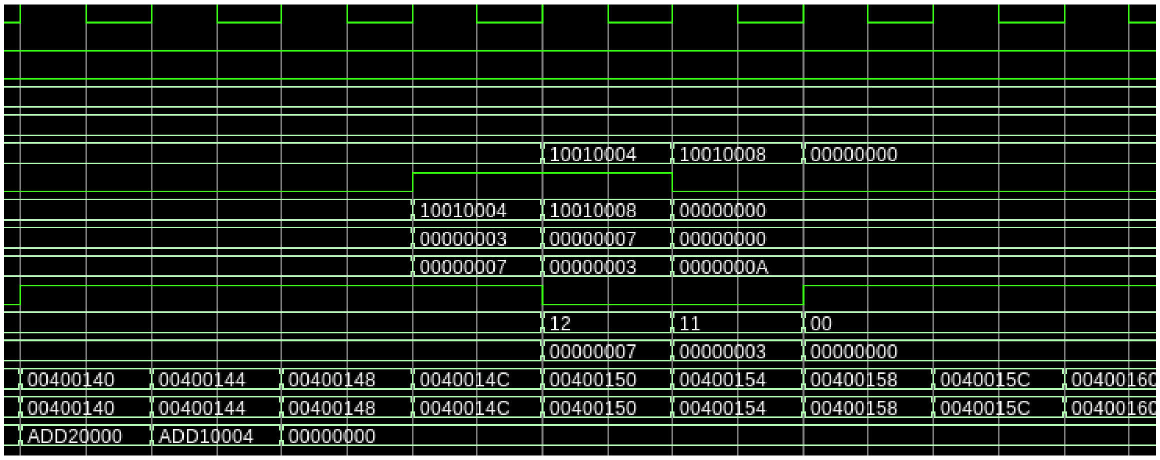
[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.

This test works the same way as the previous one, just with many more instructions.



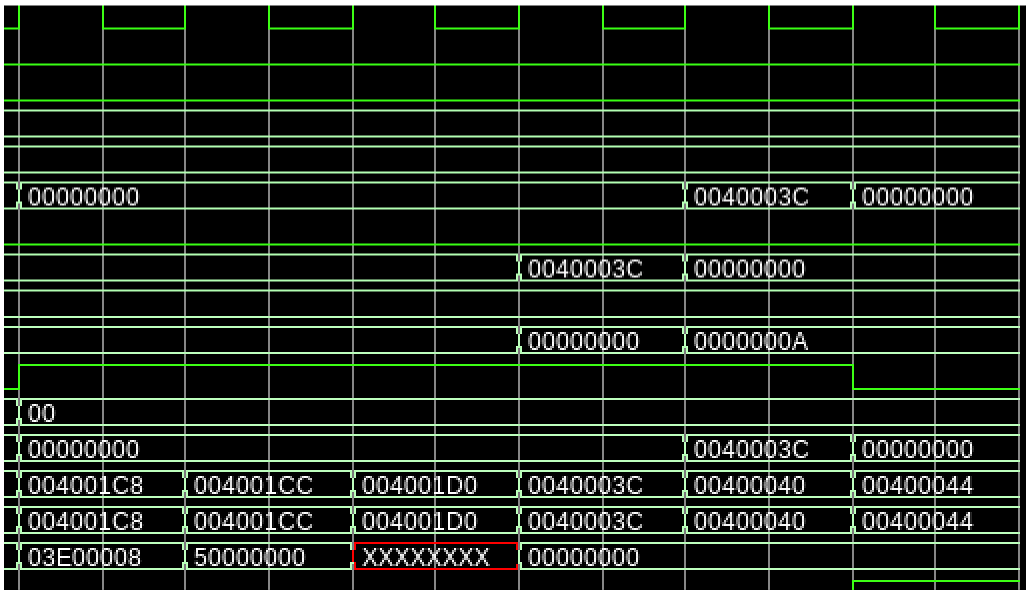
iCLK 0
iRST 0
iInstLd 0
iInstAddr 32'hXXXXXXXX XXXXXXXX
iInstExt 32'hXXXXXXXX XXXXXXXX
oALUOut 32'h00000000 00000000 10010004 10010008 00000000
s_DMemWr 0
s_DMemAddr 32'h00000000 00000000 10010004 10010008 00000000
s_DMemData 32'h00000009 00000000
s_DMemOut 32'h0000000A 0000000A 00000007 00000003 0000000A
s_RegWr 1
s_RegWrAddr 5'h00 0E 00 11 12 00
s_RegWrData 32'h00000000 00000000 00000007 00000003 00000000
s_IMemAddr 32'h004000D0 00400100 00400104 00400108 0040010C 00400110 00400114 00400118 0040011C
s_NextInstAddr 32'h004000D0 00400100 00400104 00400108 0040010C 00400110 00400114 00400118 0040011C
s_Inst 32'h00000000 8DD10000 8DD20004 00000000 0251782A 00000000

In this image, you can see instructions 8DD10000 and 8DD20004 are performed back to back without any nops. This is because these two instructions have different destination registers, the first one using $s1 (11) and the second one using $s2 (12), indicating no hazards.



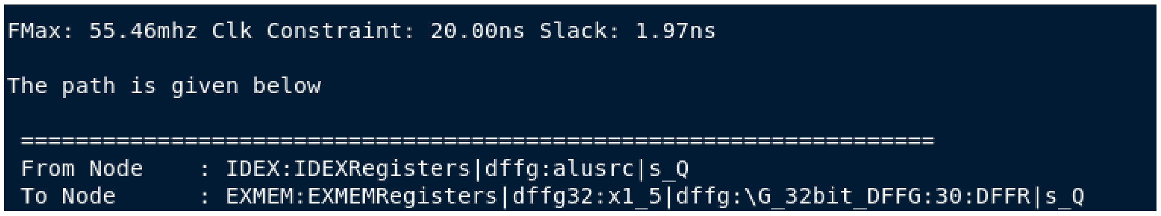
10010004 10010008 00000000
10010004 10010008 00000000
00000003 00000007 00000000
00000007 00000003 0000000A
12 11 00
00000007 00000003 00000000
00400140 00400144 00400148 0040014C 00400150 00400154 00400158 0040015C 00400160
00400140 00400144 00400148 0040014C 00400150 00400154 00400158 0040015C 00400160
ADD20000 ADD10004 00000000

A similar thing is captured in this image but with instructions ADD20000 and ADD1004.



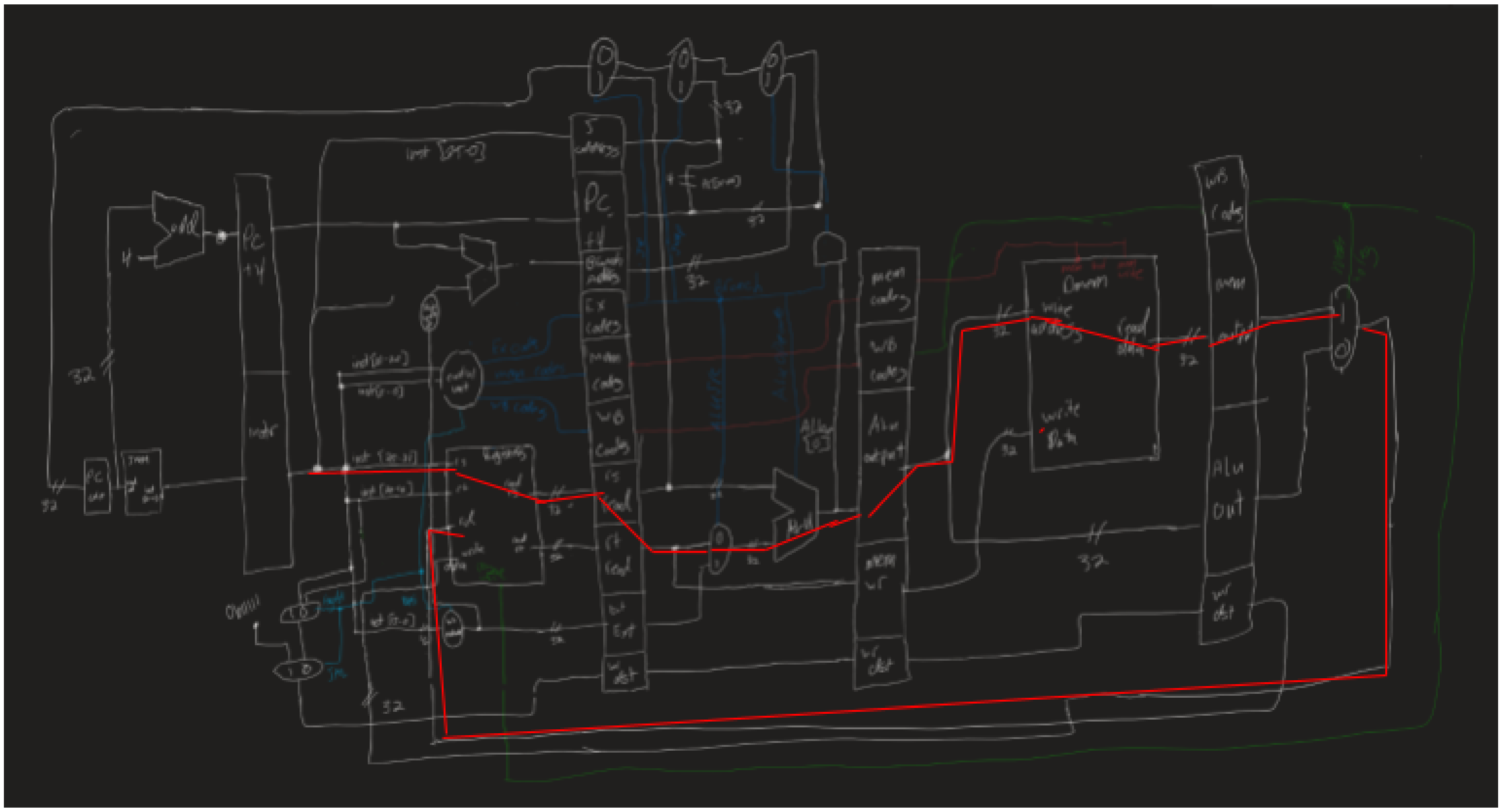
This is an example of a control flow that does not require nops. Instruction 03E00008 is a jr instruction that jumps to 50000000 which halts when it is done. Since we know this is always going to jump and we always know the value of the halt instruction, we did not need any nops.

[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).

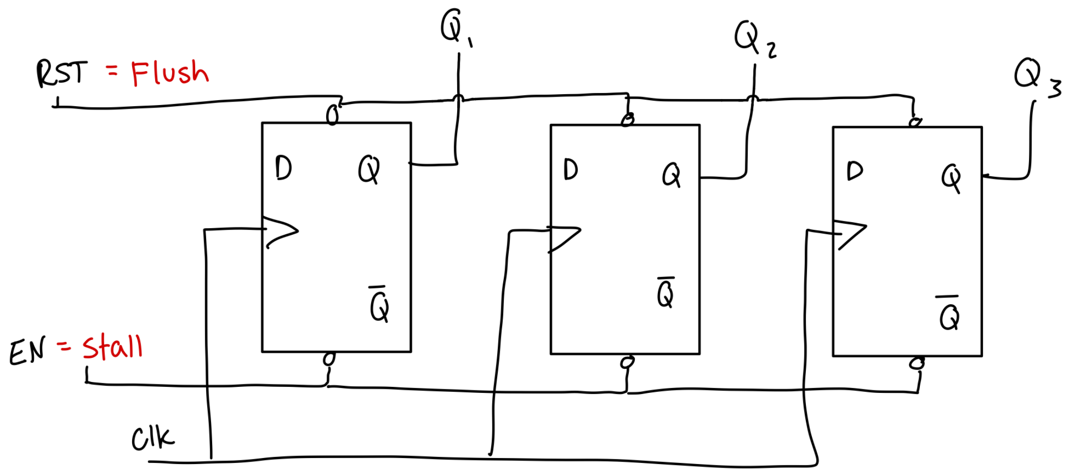


```
FMax: 55.46mhz Clk Constraint: 20.00ns Slack: 1.97ns

The path is given below

 ===================================================================
 From Node    : IDEX:IDEXRegisters|dffg:alusrc|s_Q
 To Node      : EXMEM:EXMEMRegisters|dffg32:x1_5|dffg:\G_32bit_DFFG:30:DFFR|s_Q
```

The maximum frequency of our software-scheduled pipeline is 55.46 mHz. The critical path is listed in the image above and highlighted below.

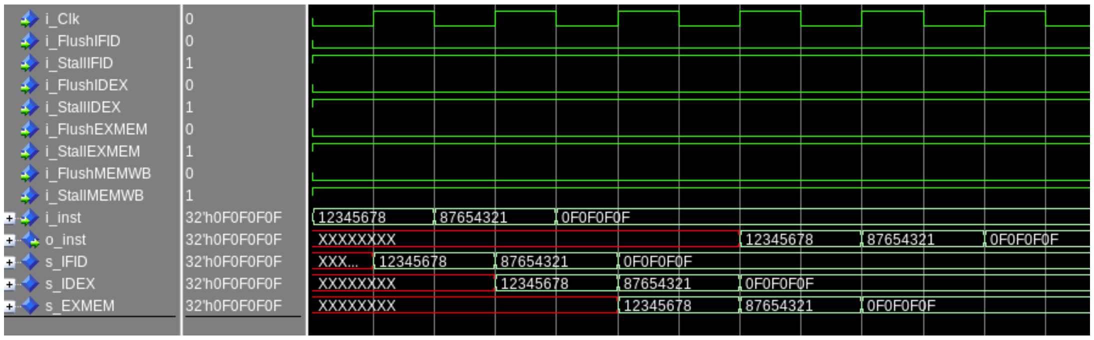


[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

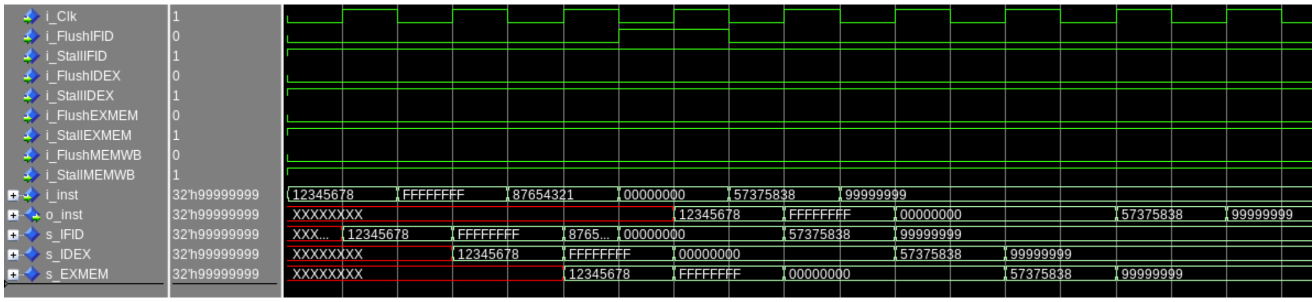


Flush is the register RST and stall is the register EN

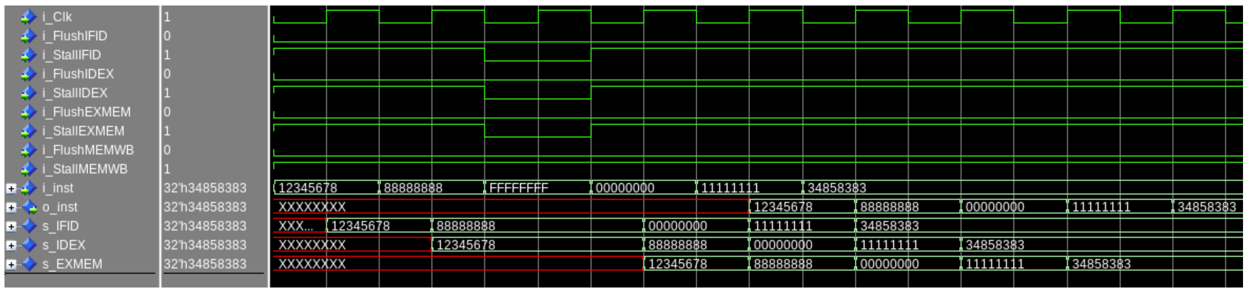
[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench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


i_Clk 0
i_FlushIFID 0
i_StallIFID 1
i_FlushIDEX 0
i_StallIDEX 1
i_FlushEXMEM 0
i_StallEXMEM 1
i_FlushMEMWB 0
i_StallMEMWB 1
i_inst 32'h0F0F0F0F 12345678 87654321 0F0F0F0F
o_inst 32'h0F0F0F0F XXXXXXXX 12345678 87654321 0F0F0F0F
s_IFID 32'h0F0F0F0F XXX... 12345678 87654321 0F0F0F0F
s_IDEX 32'h0F0F0F0F XXXXXXXX 12345678 87654321 0F0F0F0F
s_EXMEM 32'h0F0F0F0F XXXXXXXX 12345678 87654321 0F0F0F0F

Although each register has many more signals, testing was done only with the "inst" signal to show how it goes through the pipeline. This first test is just to show how feeding instructions into the pipeline works. The instruction progresses through each state and then emerges to "o_out" after 4 cycles. New instructions can be fed into the IF/ID stage every clock cycle.


i_Clk 1
i_FlushIFID 0
i_StallIFID 1
i_FlushIDEX 0
i_StallIDEX 1
i_FlushEXMEM 0
i_StallEXMEM 1
i_FlushMEMWB 0
i_StallMEMWB 1
i_inst 32'h99999999 12345678 FFFFFFFF 87654321 00000000 57375838 99999999
o_inst 32'h99999999 XXXXXXXX 12345678 FFFFFFFF 00000000 57375838 99999999
s_IFID 32'h99999999 XXX... 12345678 FFFFFFFF 8765... 00000000 57375838 99999999
s_IDEX 32'h99999999 XXXXXXXX 12345678 FFFFFFFF 00000000 57375838 99999999
s_EXMEM 32'h99999999 XXXXXXXX 12345678 FFFFFFFF 00000000 57375838 99999999

This test shows how an individual stage can be flushed. As you can see, in the third cycle, we introduced a new instruction "87654321". In the next cycle, we flushed the IF/ID stage, which flushed that instruction out of the pipeline. In the final output "o_inst," you can see that each instruction gets outputted after 4 cycles in the same order except for the instruction which was flushed.


i_Clk 1
i_FlushIFID 0
i_StallIFID 1
i_FlushIDEX 0
i_StallIDEX 1
i_FlushEXMEM 0
i_StallEXMEM 1
i_FlushMEMWB 0
i_StallMEMWB 1
i_inst 32'h34858383
o_inst 32'h34858383
s_IFID 32'h34858383
s_IDEX 32'h34858383
s_EXMEM 32'h34858383
12345678 88888888 FFFFFFFF 00000000 11111111 34858383
XXXXXXXX 12345678 88888888 00000000 11111111 34858383
XXX... 12345678 88888888 00000000 11111111 34858383
XXXXXXXX 12345678 88888888 00000000 11111111 34858383
XXXXXXXX 12345678 88888888 00000000 11111111 34858383

And this is an example of stalling the pipeline. Here, the pipeline is stalled in the ID/EX stage, which adds an extra clock cycle when"12345678" is in that stage and the pipeline is paused. You can see everything still gets executed in order, but everything in the pipeline at that point takes 5 cycles instead of 4.

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

- add, addu, and, not, nor, xor, or, slt, sll, srl, sra, sub, addi, addiu, andi, lui, lw, xori, ori, slti, sw, jal, repl.qb, movn
    - Signal - **ALUOut**. Since all of these instructions produce values, we care about where the value gets produced, which is in the ALU. The signal ALUOut contains the value that gets produced by these instructions

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

- add, addi, addiu, addu, and, andi, lui, lw, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sw, sub, subu, repl.qb, movn
    - Signal - **RegWrAddr.** Since all of these instructions consume values, we care about where the value gets consumed, which is when the destination register gets written back to. The signal RegWrAddr writes the data back to the destination register, which "consumes" the value of the operation.

[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 dependencies that will require hazard stalls.

DM/WB forward to ID/EX(ALU) OR DM/WB forward to ID/EX with stall
ID/EX forward to ALU
PC, IF/ID will be stalled and ID/EX will be flushed
IF/ID will be flushed for jump and PC, IF/ID will be stalled and ID/EX will be flushed for JR
Branch: PC and ID/ID will be stalled and ID/EX will be flushed

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

IF/ID: **INPUTS:** Clk, Flush, Stall, inst, PC

**OUTPUTS:** inst, PC

ID/EX: **INPUTS:** Clk, RST, Stall, ALUSrcSel, ALUOpCode, regDst, MemWrEn**,** i_branchSel, memToRegSel, regRsIn, regRtIn, RegWrAddr, JaloDataWrite, regRsOut, regRtOut, immMuxOut, jumpAddr, branchAddr, jumpSel, jumpRegSel, jalSel, memReadSel, regWr, PC, Halt, inst, opcode
**OUTPUTS:** inst, opcode, ALUSrcSel, ALUOpCode, regDst, MemWrEn, branchSel, memToRegSel, regRsIn, regRtIn, RegWrAddr, JaloDataWrite, regRsOut, regRtOut, jumpAddr, branchAddr, immMuxOut, jumpSel, jumpRegSel, jalSel, memReadSel, regWr, PC, Halt.

EX/MEM: **INPUTS:** Clk, RST, EN, readData2, aluOut, writeReg, overflow, MemtoReg, weMem, weReg, DMemRead, Halt, imm, inst, branch, jump, opcode,
**OUTPUTS:** inst, branch, jump, opcode, readData2, aluOut, writeReg, overflow, MemtoReg, weMem, weReg, DMemRead, Halt, imm

MEMWB: **INPUTS:** CLK, RST, EN, memReadData, aluOut, writeReg, overflow, MemtoReg, weReg, halt, imm, branch, jump, inst,
**OUTPUTS:** inst, memReadData, aluOut, writeReg, overflow, MemtoReg, weReg, halt, imm

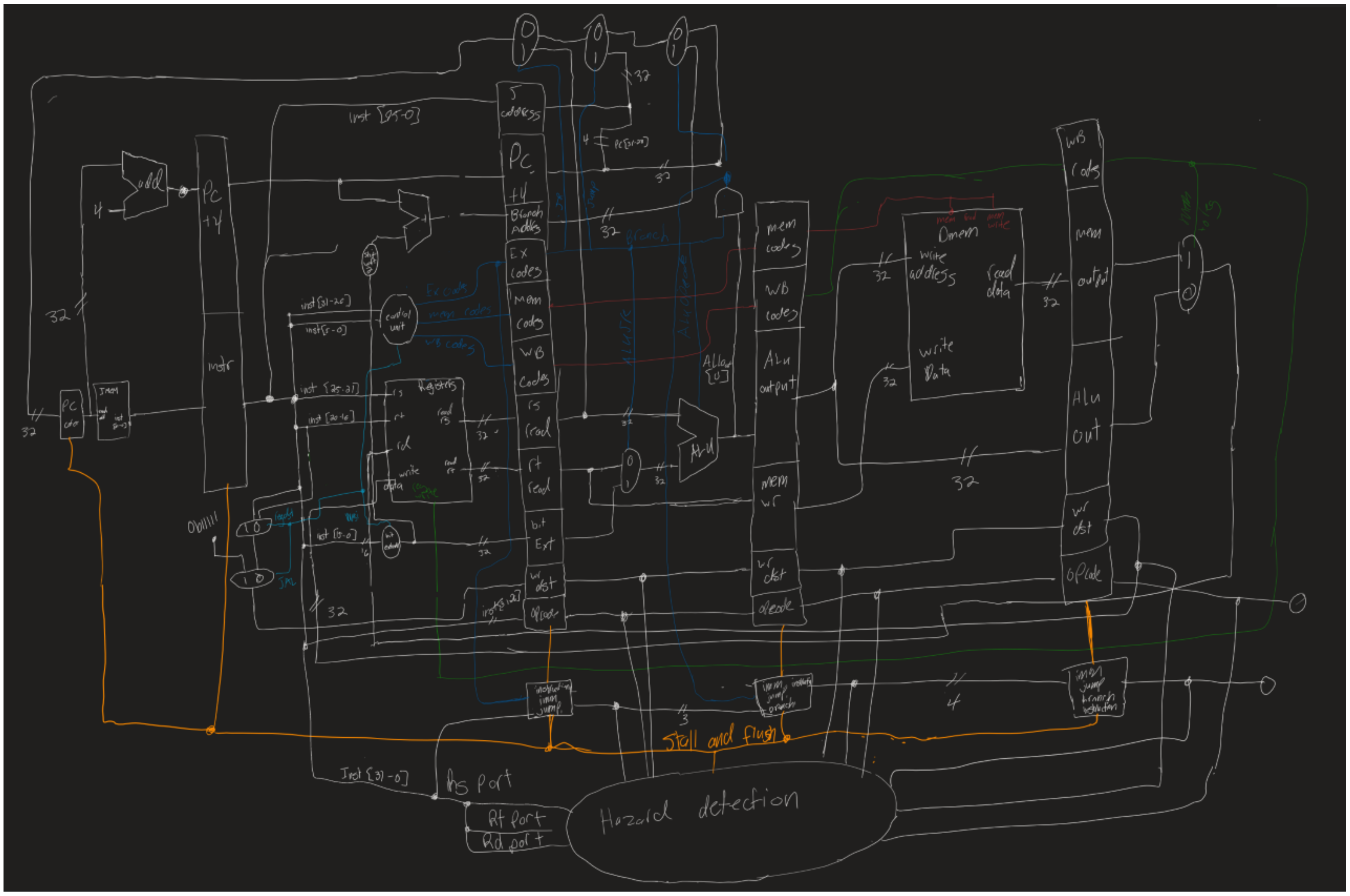
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

J, JAL, Jr, BNE and BEQ all result in a non sequential PC update.
BNE and BEQ require a signal from the ALU to tell our processor to branch rather than use the next sequential PC update. This happens in the Execution stage (EX).
J, JAL and Jr require a signal from the control unit to tell the processor to jump to a new PC address rather than use the next sequential PC update. This also happens in the Execution stage (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.

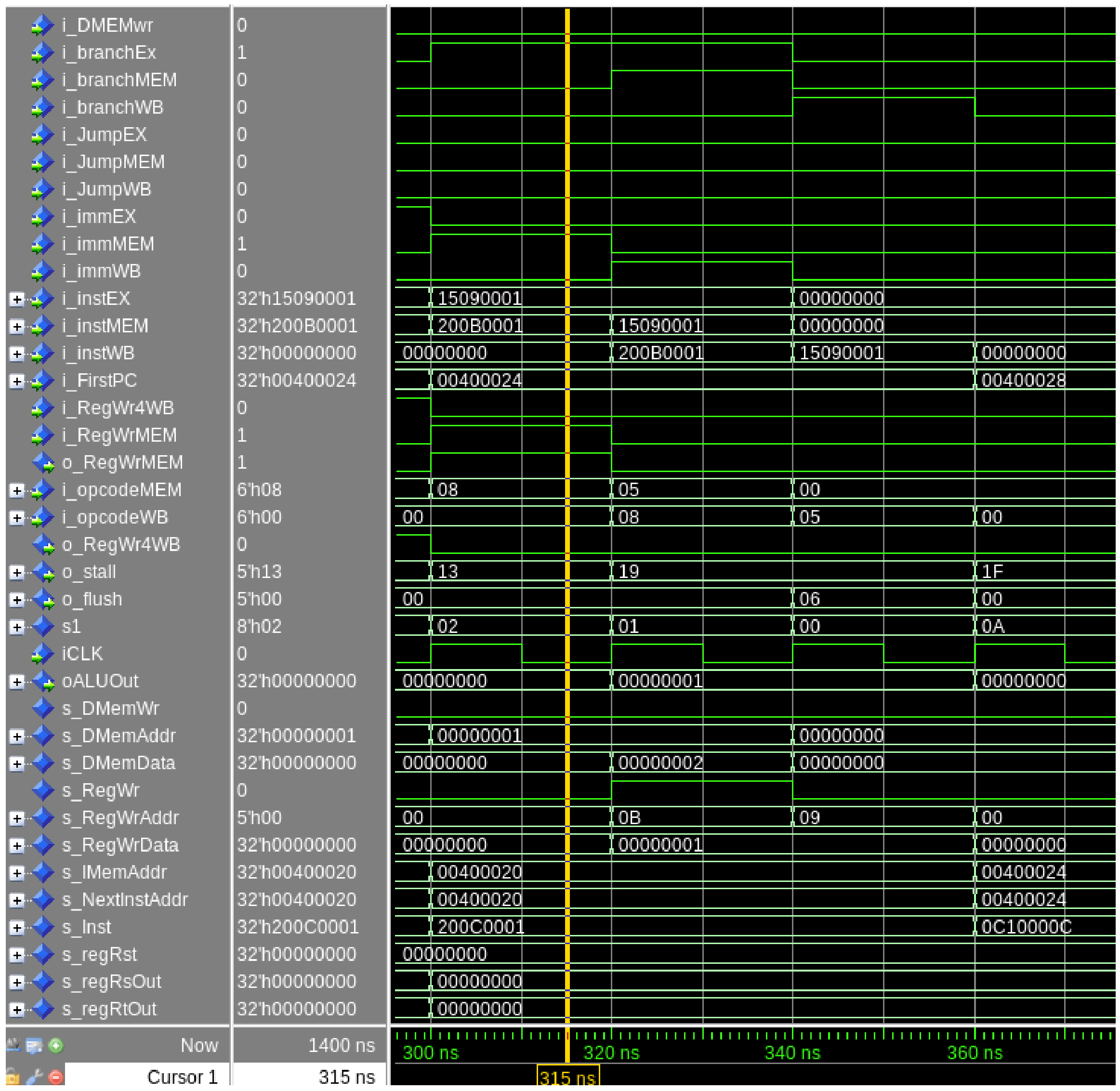
IF/ID: Load - Stall, J - flush, Jal - flush, Jr - stall, Branch - stall
ID/EX: Load - Flush, Jr - flush, Branch - flush

[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 Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

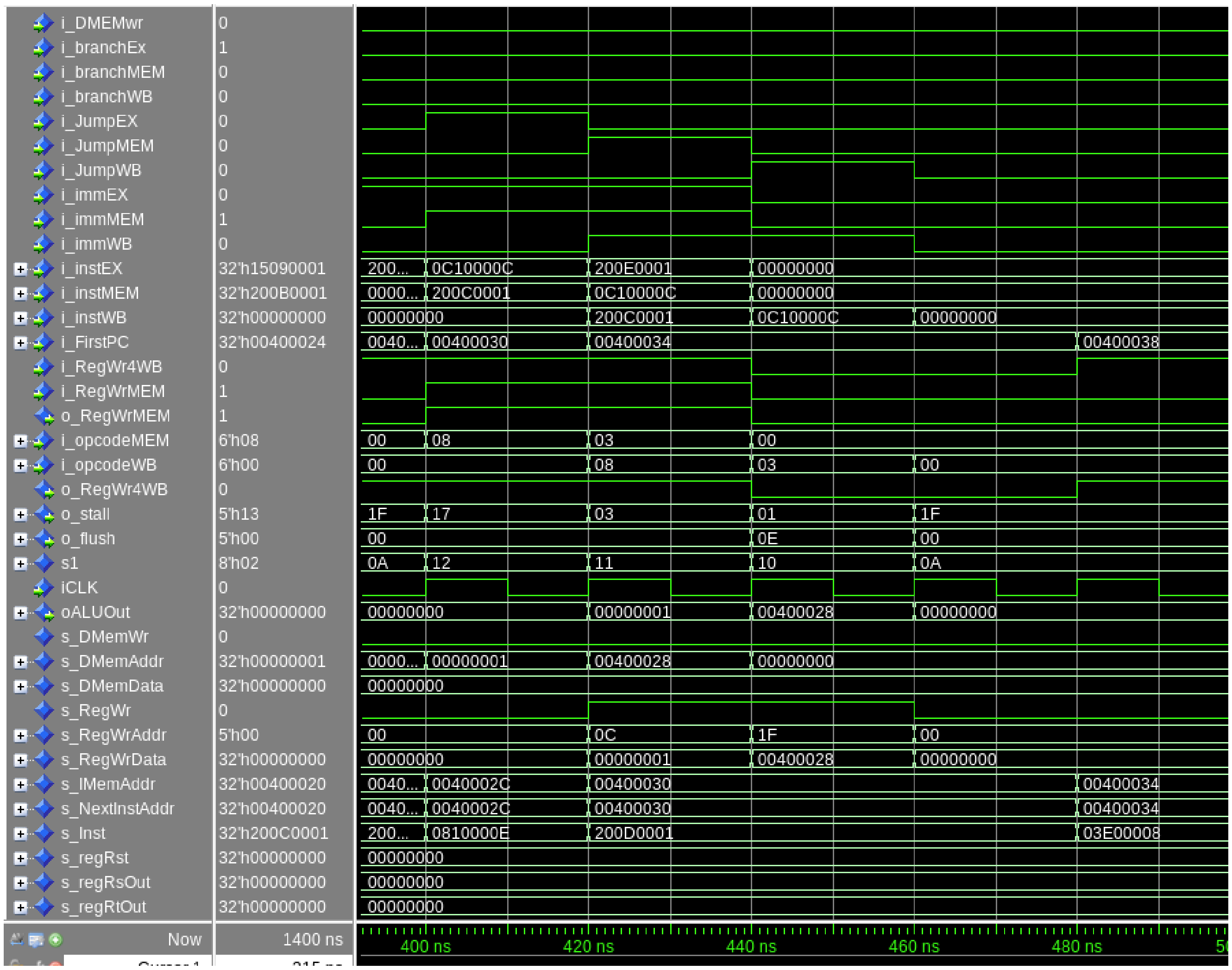
Control Hazards Branching:



| | |
|---|---|
| i_DMEMwr | 0 |
| i_branchEx | 1 |
| i_branchMEM | 0 |
| i_branchWB | 0 |
| i_JumpEX | 0 |
| i_JumpMEM | 0 |
| i_JumpWB | 0 |
| i_immEX | 0 |
| i_immMEM | 1 |
| i_immWB | 0 |
| i_instEX | 32'h15090001 |
| i_instMEM | 32'h200B0001 |
| i_instWB | 32'h00000000 |
| i_FirstPC | 32'h00400024 |
| i_RegWr4WB | 0 |
| i_RegWrMEM | 1 |
| o_RegWrMEM | 1 |
| i_opcodeMEM | 6'h08 |
| i_opcodeWB | 6'h00 |
| o_RegWr4WB | 0 |
| o_stall | 5'h13 |
| o_flush | 5'h00 |
| s1 | 8'h02 |
| iCLK | 0 |
| oALUOut | 32'h00000000 |
| s_DMemWr | 0 |
| s_DMemAddr | 32'h00000001 |
| s_DMemData | 32'h00000000 |
| s_RegWr | 0 |
| s_RegWrAddr | 5'h00 |
| s_RegWrData | 32'h00000000 |
| s_IMemAddr | 32'h00400020 |
| s_NextInstAddr | 32'h00400020 |
| s_Inst | 32'h200C0001 |
| s_regRst | 32'h00000000 |
| s_regRsOut | 32'h00000000 |
| s_regRtOut | 32'h00000000 |
| Now | 1400 ns |
| Cursor 1 | 315 ns |

Here is an example of the Branching control hazard detection working properly. It sees that we are branching in the EX stage, and tell the hazard detection unit to stall the pipeline and wipe the next instruction. Then a couple of cycles later, after making sure the previous instructions make it through the pipeline, the program branches. Here, it only branches to the PC + 4 but this works in other cases as well, such as bubble sort.

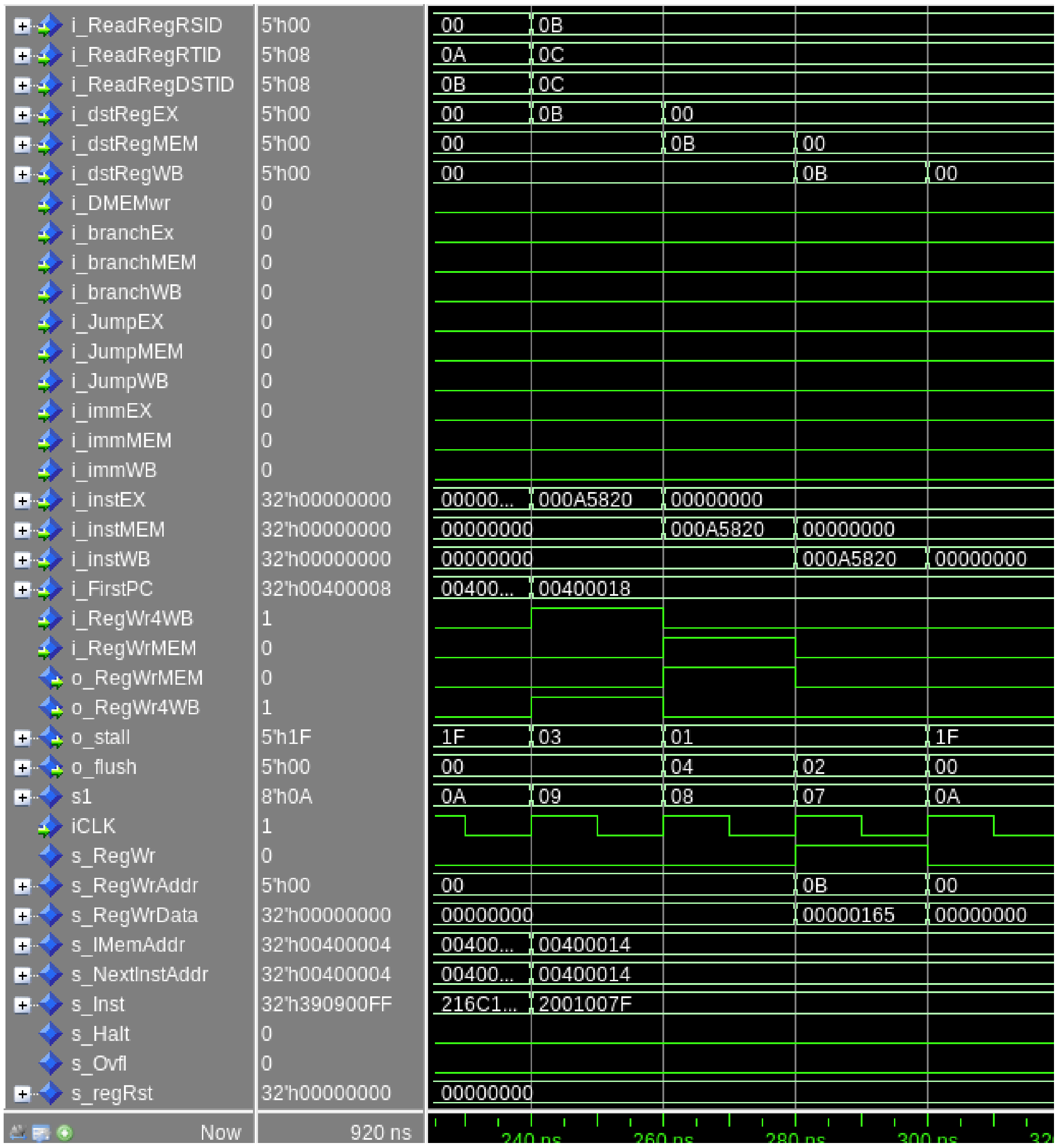
Control hazards Jumping:



| | |
|---|---|
| i_DMEMwr | 0 |
| i_branchEx | 1 |
| i_branchMEM | 0 |
| i_branchWB | 0 |
| i_JumpEX | 0 |
| i_JumpMEM | 0 |
| i_JumpWB | 0 |
| i_immEX | 0 |
| i_immMEM | 1 |
| i_immWB | 0 |
| i_instEX | 32'h15090001 |
| i_instMEM | 32'h200B0001 |
| i_instWB | 32'h00000000 |
| i_FirstPC | 32'h00400024 |
| i_RegWr4WB | 0 |
| i_RegWrMEM | 1 |
| o_RegWrMEM | 1 |
| i_opcodeMEM | 6'h08 |
| i_opcodeWB | 6'h00 |
| o_RegWr4WB | 0 |
| o_stall | 5'h13 |
| o_flush | 5'h00 |
| s1 | 8'h02 |
| iCLK | 0 |
| oALUOut | 32'h00000000 |
| s_DMemWr | 0 |
| s_DMemAddr | 32'h00000001 |
| s_DMemData | 32'h00000000 |
| s_RegWr | 0 |
| s_RegWrAddr | 5'h00 |
| s_RegWrData | 32'h00000000 |
| s_IMemAddr | 32'h00400020 |
| s_NextInstAddr | 32'h00400020 |
| s_Inst | 32'h200C0001 |
| s_regRst | 32'h00000000 |
| s_regRsOut | 32'h00000000 |
| s_regRtOut | 32'h00000000 |
| Now | 1400 ns |

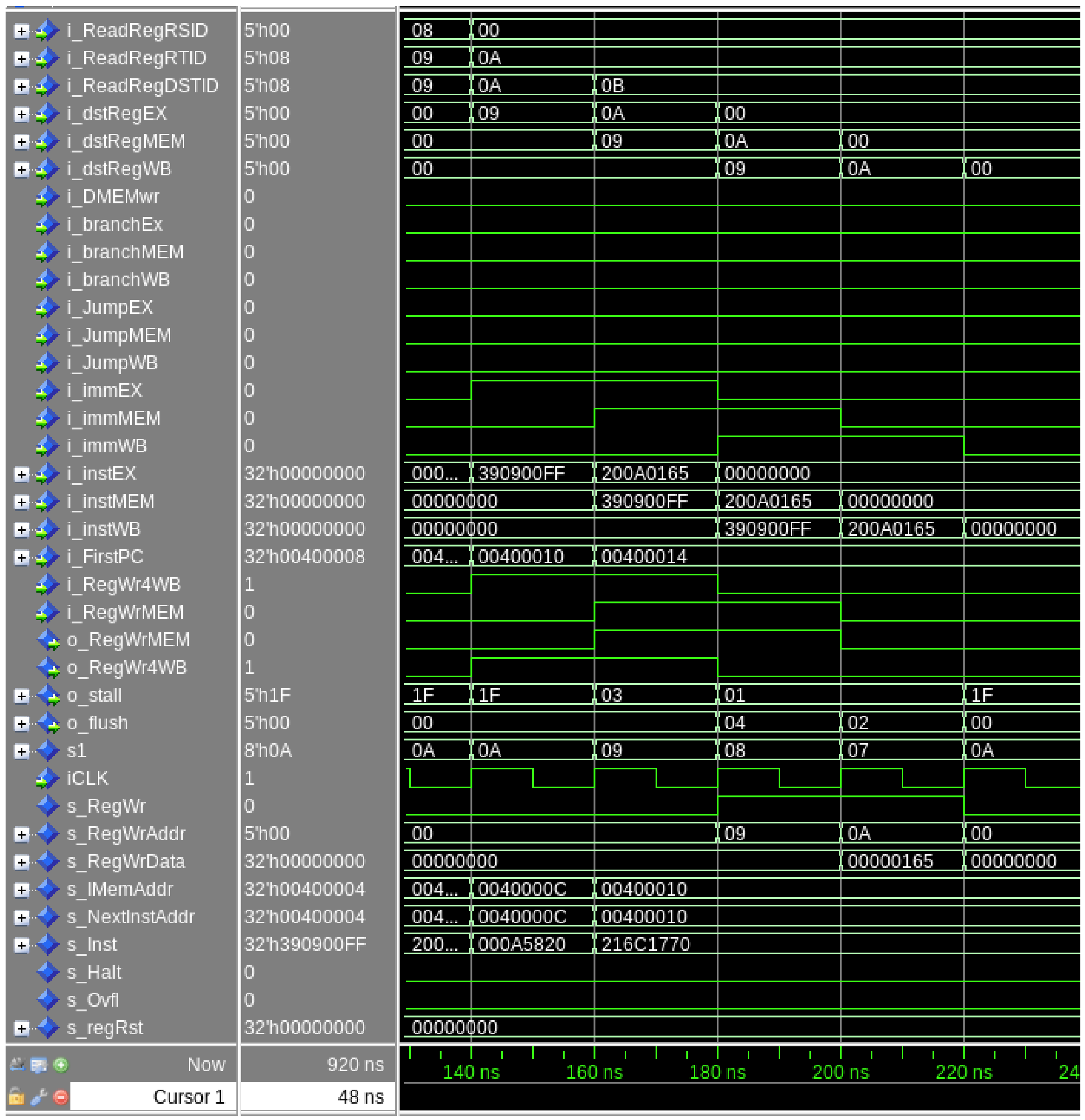
Here, we can see the program wants to jump, so we do the same thing as branching seen above. Let the previous instructions before the jump finish through, jump to the correct address and flush the instruction after the jump. This one specifically needs to wait until the jump instruction gets to the WB stage as JAL needs to write to the register file before jumping. This example only jumps to the next line, but this hazard detection also works with other programs like bubble sort.

<u>Data Hazards:</u>



| | |
|---|---|
| i_ReadRegRSID | 5'h00 |
| i_ReadRegRTID | 5'h08 |
| i_ReadRegDSTID | 5'h08 |
| i_dstRegEX | 5'h00 |
| i_dstRegMEM | 5'h00 |
| i_dstRegWB | 5'h00 |
| i_DMEMwr | 0 |
| i_branchEx | 0 |
| i_branchMEM | 0 |
| i_branchWB | 0 |
| i_JumpEX | 0 |
| i_JumpMEM | 0 |
| i_JumpWB | 0 |
| i_immEX | 0 |
| i_immMEM | 0 |
| i_immWB | 0 |
| i_instEX | 32'h00000000 |
| i_instMEM | 32'h00000000 |
| i_instWB | 32'h00000000 |
| i_FirstPC | 32'h00400008 |
| i_RegWr4WB | 1 |
| i_RegWrMEM | 0 |
| o_RegWrMEM | 0 |
| o_RegWr4WB | 1 |
| o_stall | 5'h1F |
| o_flush | 5'h00 |
| s1 | 8'h0A |
| iCLK | 1 |
| s_RegWr | 0 |
| s_RegWrAddr | 5'h00 |
| s_RegWrData | 32'h00000000 |
| s_IMemAddr | 32'h00400004 |
| s_NextInstAddr | 32'h00400004 |
| s_Inst | 32'h390900FF |
| s_Halt | 0 |
| s_Ovfl | 0 |
| s_regRst | 32'h00000000 |
| Now | 920 ns |

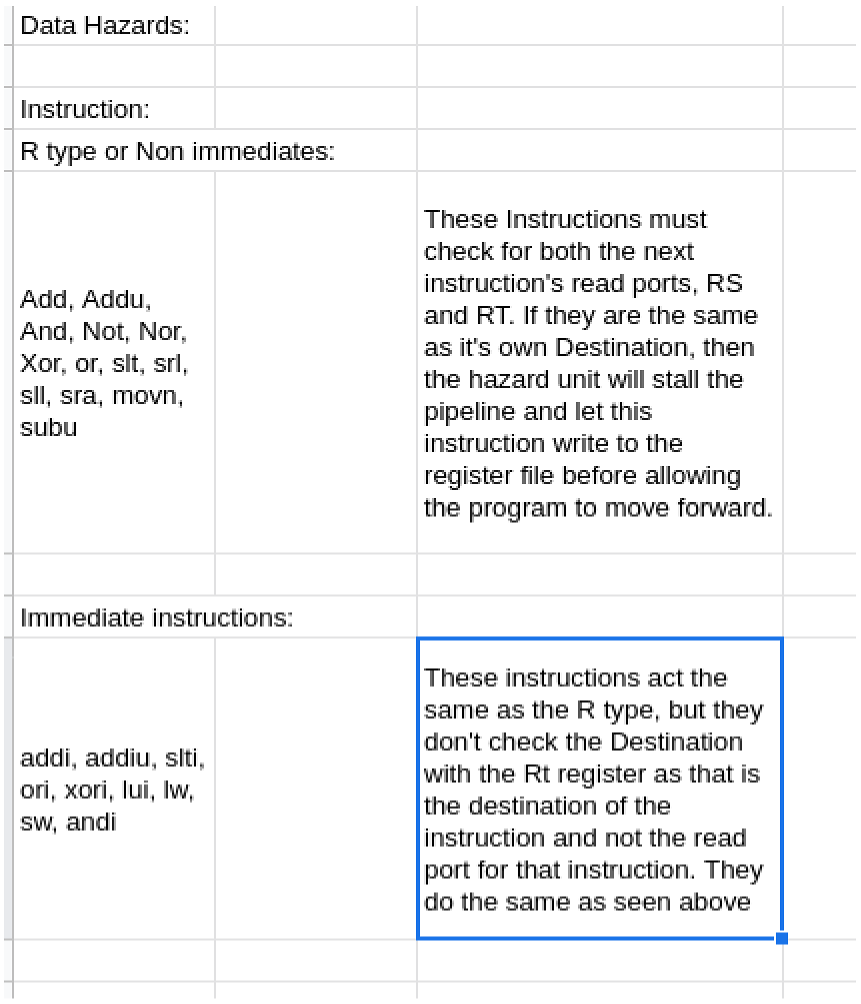
Above we have a typical Data hazard between 2 normal instructions (no immediate values). Seeing that the destination is the same as the readport for the next instruction, the hazard unit stalls the program and lets the instruction in the pipeline finish its stages and writes back. Then, execution begins as normal. In this example, the dstRegEX = 0x0B and the ReadRegRSID = 0x0B. This causes the hazard unit to kick into action and begins stalling and flushing the program. This example also works with the RT read port. The signal S1 is a debug signal used to tell the programmer which hazard was found. This made creating and debugging the hazard unit way easier.



| | |
|---|---|
| i_ReadRegRSID | 5'h00 |
| i_ReadRegRTID | 5'h08 |
| i_ReadRegDSTID | 5'h08 |
| i_dstRegEX | 5'h00 |
| i_dstRegMEM | 5'h00 |
| i_dstRegWB | 5'h00 |
| i_DMEMwr | 0 |
| i_branchEx | 0 |
| i_branchMEM | 0 |
| i_branchWB | 0 |
| i_JumpEX | 0 |
| i_JumpMEM | 0 |
| i_JumpWB | 0 |
| i_immEX | 0 |
| i_immMEM | 0 |
| i_immWB | 0 |
| i_instEX | 32'h00000000 |
| i_instMEM | 32'h00000000 |
| i_instWB | 32'h00000000 |
| i_FirstPC | 32'h00400008 |
| i_RegWr4WB | 1 |
| i_RegWrMEM | 0 |
| o_RegWrMEM | 0 |
| o_RegWr4WB | 1 |
| o_stall | 5'h1F |
| o_flush | 5'h00 |
| s1 | 8'h0A |
| iCLK | 1 |
| s_RegWr | 0 |
| s_RegWrAddr | 5'h00 |
| s_RegWrData | 32'h00000000 |
| s_IMemAddr | 32'h00400004 |
| s_NextInstAddr | 32'h00400004 |
| s_Inst | 32'h390900FF |
| s_Halt | 0 |
| s_Ovfl | 0 |
| s_regRst | 32'h00000000 |
| Now | 920 ns |
| Cursor 1 | 48 ns |

Above you can see an example of when there's a data hazard with an immediate value. Unlike with normal data hazards, instructions that have immediate values only care about if the instruction in front of itself changes the RS register. Meaning, if the instruction in front has the same destination as the immediate instruction’s read port, then it needs to stall the program for the instruction to write the correct value before reading. This can be seen above with the i_imm control signals being stalled and pushed through the pipeline. You can also see the RS and DST registers are the same so the hazard unit has to stall the program.

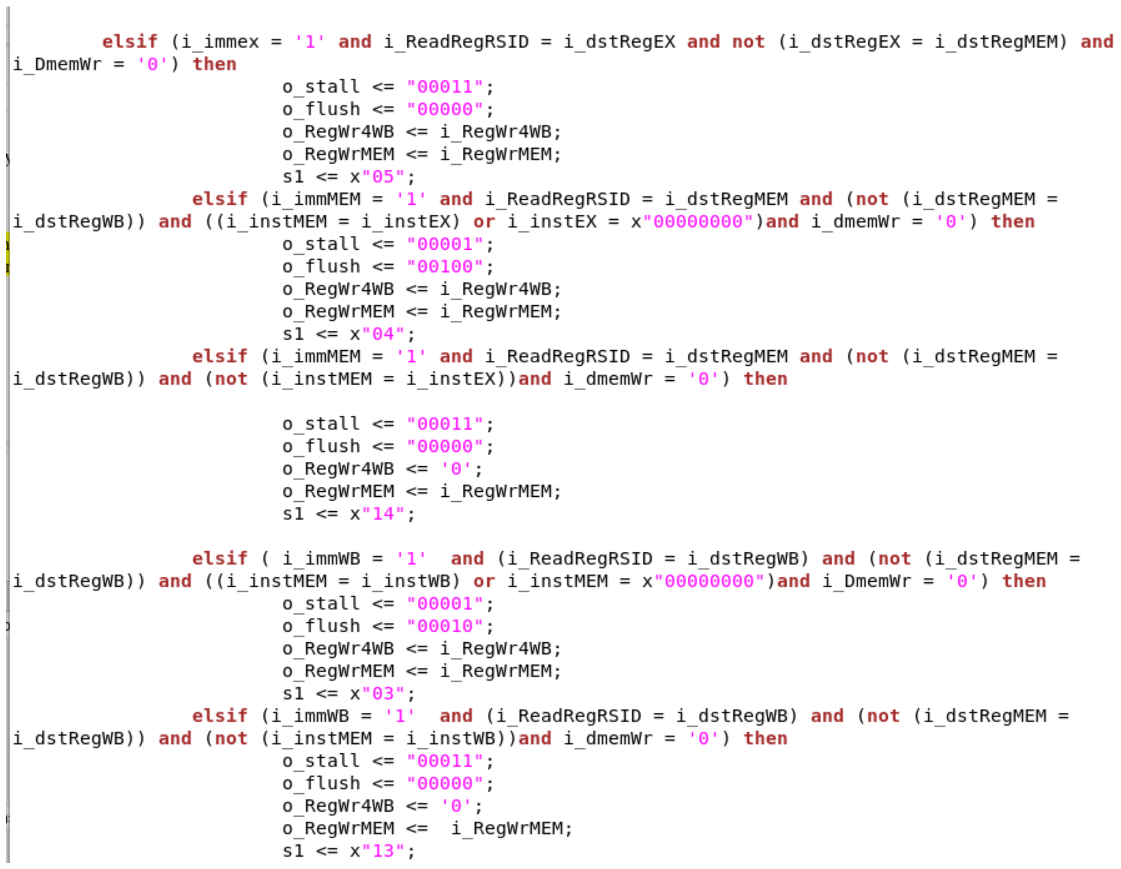
[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.



| | | | |
|---|---|---|---|
| Data Hazards: | | | |
| | | | |
| Instruction: | | | |
| R type or Non immediates: | | | |
| Add, Addu, And, Not, Nor, Xor, or, slt, srl, sll, sra, movn, subu | | These Instructions must check for both the next instruction's read ports, RS and RT. If they are the same as it's own Destination, then the hazard unit will stall the pipeline and let this instruction write to the register file before allowing the program to move forward. | |
| | | | |
| Immediate instructions: | | | |
| addi, addiu, slti, ori, xori, lui, lw, sw, andi | | These instructions act the same as the R type, but they don't check the Destination with the Rt register as that is the destination of the instruction and not the read port for that instruction. They do the same as seen above | |
| | | | |

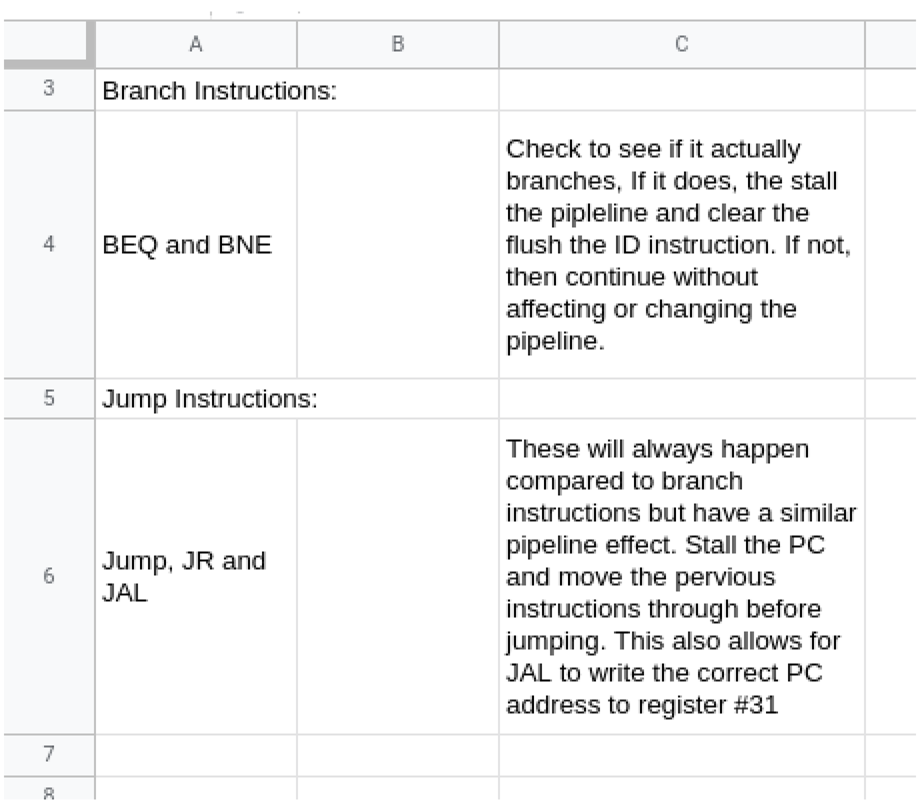
Here's the R-type and I-type hazard detection code

```
                elsif (((i_ReadRegRSID = i_dstRegWB) or (i_ReadRegRTID = i_dstRegWB))and ((not (i_ReadRegRSID = "00000"))
or (not (i_ReadRegRTID = "00000")))  and (not (i_dstRegWB = "00000")) and ((i_instMEM = i_instWB) or i_instMEM =
x"00000000")) then
                        o_stall <= "00001";
                        o_flush <= "00010";
                        o_RegWr4WB <= i_RegWr4WB;
                        o_RegWrMEM <= i_RegWrMEM;
                        s1 <= x"07";
                elsif (((i_ReadRegRSID = i_dstRegWB) or (i_ReadRegRTID = i_dstRegWB))and ((not (i_ReadRegRSID = "00000"))
or (not (i_ReadRegRTID = "00000")))  and (not (i_dstRegWB = "00000")) and (not (i_instMEM = i_instWB))) then
                        o_stall <= "00011";
                        o_flush <= "00000";
                        o_RegWr4WB <= '0';
                        o_RegWrMEM <= i_RegWrMEM;
                        s1 <= x"17";

                elsif (((i_ReadRegRSID = i_dstRegMEM) or (i_ReadRegRTID = i_dstRegMEM))and ((not (i_ReadRegRSID =
"00000")) or (not (i_ReadRegRTID = "00000")))  and (not (i_dstRegMEM = "00000")) and ((i_instMEM = i_instEX) or i_instEX
= x"00000000")) then
                        o_stall <= "00001";
                        o_flush <= "00100";
                        o_RegWr4WB <= i_RegWr4WB;
                        o_RegWrMEM <= i_RegWrMEM;
                        s1 <= x"08";

                elsif (((i_ReadRegRSID = i_dstRegMEM) or (i_ReadRegRTID = i_dstRegMEM))and ((not (i_ReadRegRSID =
"00000")) or (not (i_ReadRegRTID = "00000")))  and (not (i_dstRegMEM = "00000")) and (not (i_instMEM = i_instEX))) then
                        o_stall <= "00011";
                        o_flush <= "00000";
                        o_RegWr4WB <= '0';
                        o_RegWrMEM <= i_regWrMEM;
                        s1 <= x"18";


        elsif (((i_ReadRegRSID = i_dstRegEX) or (i_ReadRegRTID = i_dstRegEX)) and ( (not (i_ReadRegRSID = "00000")) or
(not (i_ReadRegRTID = "00000"))) and (not (i_dstRegEX = "00000"))) then
                        o_stall <= "00011";
                        o_flush <= "00000";
                        o_RegWr4WB <= i_RegWr4WB;
                        o_RegWrMEM <= i_RegWrMEM;
                        s1 <= x"09";

        elsif ((i_dstRegWB = i_dstRegEX) and i_opcodeMEM = "100011") then
                        o_stall <= "00001";
                        o_flush <= "00010";
                        o_RegWr4WB <= i_RegWr4Wb;
                        o_RegWrMEM <= i_RegWrMEM;
```



```
        elsif (i_immex = '1' and i_ReadRegRSID = i_dstRegEX and not (i_dstRegEX = i_dstRegMEM) and
i_DmemWr = '0') then
                        o_stall <= "00011";
                        o_flush <= "00000";
                        o_RegWr4WB <= i_RegWr4WB;
                        o_RegWrMEM <= i_RegWrMEM;
                        s1 <= x"05";
                elsif (i_immMEM = '1' and i_ReadRegRSID = i_dstRegMEM and (not (i_dstRegMEM =
i_dstRegWB)) and ((i_instMEM = i_instEX) or i_instEX = x"00000000")and i_dmemWr = '0') then
                        o_stall <= "00001";
                        o_flush <= "00100";
                        o_RegWr4WB <= i_RegWr4WB;
                        o_RegWrMEM <= i_RegWrMEM;
                        s1 <= x"04";
                elsif (i_immMEM = '1' and i_ReadRegRSID = i_dstRegMEM and (not (i_dstRegMEM =
i_dstRegWB)) and (not (i_instMEM = i_instEX))and i_dmemWr = '0') then

                        o_stall <= "00011";
                        o_flush <= "00000";
                        o_RegWr4WB <= '0';
                        o_RegWrMEM <= i_RegWrMEM;
                        s1 <= x"14";

                elsif ( i_immWB = '1'  and (i_ReadRegRSID = i_dstRegWB) and (not (i_dstRegMEM =
i_dstRegWB)) and ((i_instMEM = i_instWB) or i_instMEM = x"00000000")and i_DmemWr = '0') then
                        o_stall <= "00001";
                        o_flush <= "00010";
                        o_RegWr4WB <= i_RegWr4WB;
                        o_RegWrMEM <= i_RegWrMEM;
                        s1 <= x"03";
                elsif (i_immWB = '1'  and (i_ReadRegRSID = i_dstRegWB) and (not (i_dstRegMEM =
i_dstRegWB)) and (not (i_instMEM = i_instWB))and i_dmemWr = '0') then
                        o_stall <= "00011";
                        o_flush <= "00000";
                        o_RegWr4WB <= '0';
                        o_RegWrMEM <=  i_RegWrMEM;
                        s1 <= x"13";
```

[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.



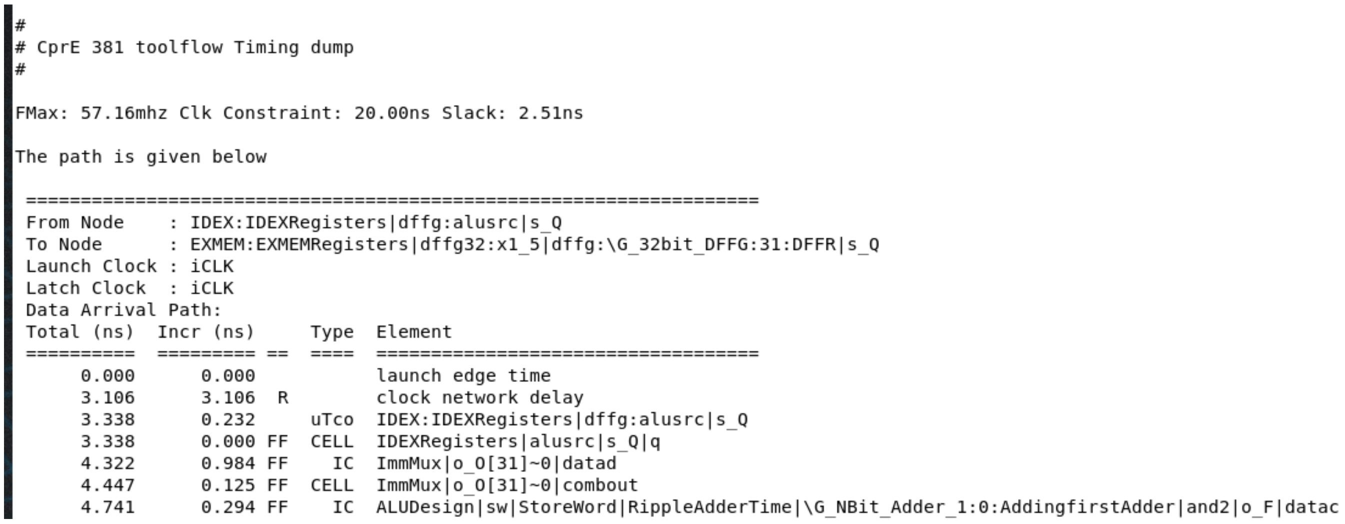
| | A | B | C |
|---|---|---|---|
| 3 | Branch Instructions: | | |
| 4 | BEQ and BNE | | Check to see if it actually branches, If it does, the stall the pipleline and clear the flush the ID instruction. If not, then continue without affecting or changing the pipeline. |
| 5 | Jump Instructions: | | |
| 6 | Jump, JR and JAL | | These will always happen compared to branch instructions but have a similar pipeline effect. Stall the PC and move the pervious instructions through before jumping. This also allows for JAL to write the correct PC address to register #31 |
| 7 | | | |
| 8 | | | |

What the code looks like is found below

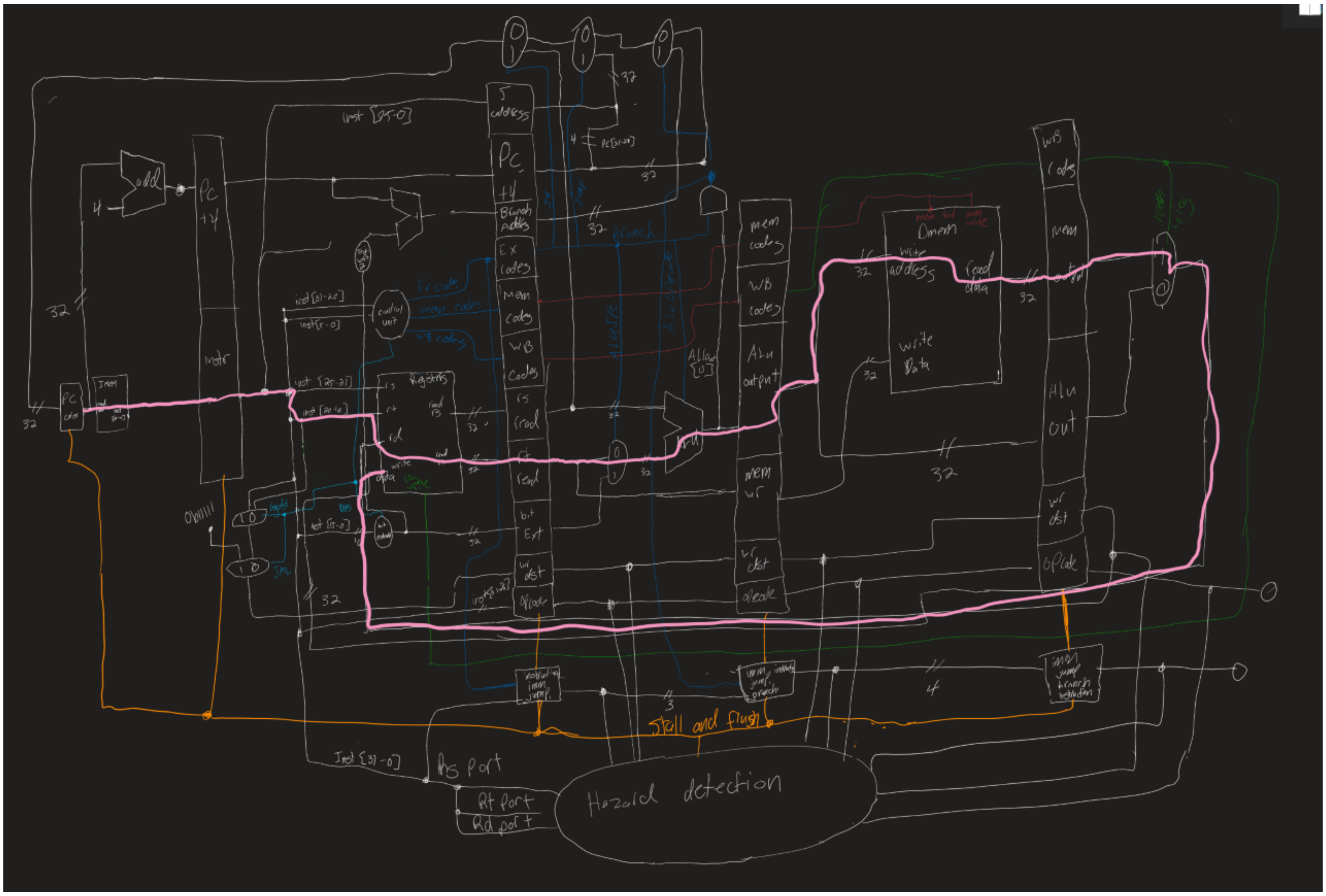
```
if (( i_jumpWB = '1')) then
                o_stall <= "00001";
                o_flush <= "01110";
                o_RegWr4WB <= i_RegWr4WB;
                o_RegWrMEM <= i_RegWrMEM;
                s1 <= x"10";

elsif (( i_jumpMEM = '1')) then
                o_stall <= "00011";
                o_flush <= "00000";
                o_RegWr4WB <= i_RegWr4WB;
                o_RegWrMEM <= i_RegWrMEM;
                s1 <= x"11";
elsif (( i_jumpEX = '1')) then
                o_stall <= "10111";
                o_flush <= "00000";
                o_RegWr4WB <= i_RegWr4WB;
                o_RegWrMEM <= i_RegWrMEM;
                s1 <= x"12";
elsif ((i_branchWB = '1')) then
                o_stall <= "11001";
                o_flush <= "00110";
                o_RegWr4WB <= i_RegWr4WB;
                o_RegWrMEM <= i_RegWrMEM;
                s1 <= x"00";

elsif ((i_branchMEM = '1')) then
                o_stall <= "11001";
                o_flush <= "00000";
                o_RegWr4WB <= i_RegWr4WB;
                o_RegWrMEM <= i_RegWrMEM;
                s1 <= x"01";
elsif ((i_branchEX = '1')) then
                o_stall <= "10011";
                o_flush <= "00000";
                o_RegWr4WB <= i_RegWr4WB;
                o_RegWrMEM <= i_RegWrMEM;
                s1 <= x"02";
```

[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).

Path on the last page



```
#
# CprE 381 toolflow Timing dump
#

FMax: 57.16mhz Clk Constraint: 20.00ns Slack: 2.51ns

The path is given below

 ===================================================================
 From Node    : IDEX:IDEXRegisters|dffg:alusrc|s_Q
 To Node      : EXMEM:EXMEMRegisters|dffg32:x1_5|dffg:\G_32bit_DFFG:31:DFFR|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
      0.000      0.000           launch edge time
      3.106      3.106  R        clock network delay
      3.338      0.232     uTco  IDEX:IDEXRegisters|dffg:alusrc|s_Q
      3.338      0.000 FF  CELL  IDEXRegisters|alusrc|s_Q|q
      4.322      0.984 FF    IC  ImmMux|o_O[31]~0|datad
      4.447      0.125 FF  CELL  ImmMux|o_O[31]~0|combout
      4.741      0.294 FF    IC  ALUDesign|sw|StoreWord|RippleAdderTime|\G_NBit_Adder_1:0:AddingfirstAdder|and2|o_F|datac
```



The critical path is seen below. This would be LW as it goes through all of the state registers and most of the components in the device. The path in order goes, Pc, then Imem, then IF/ID register, register file, ID/EX register, immediate mux, ALU, EX/MEM register, Dmem, MEM/WB register, data mux, then finally write back to the register.