CprE 381: Computer Organization and Assembly Level Programming, Spring 2019

Report – Project Part 3

Lab Partners ___________________Sean Gordon_________________

____

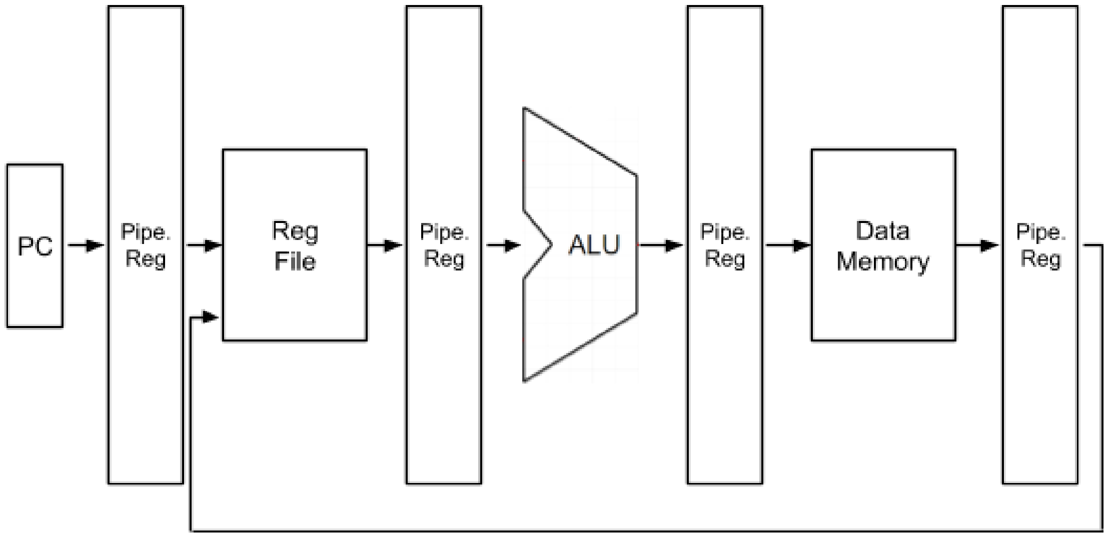
____

Section /Lab Time __________C/10:00__________

Canvas Group ID __________Team-8__________

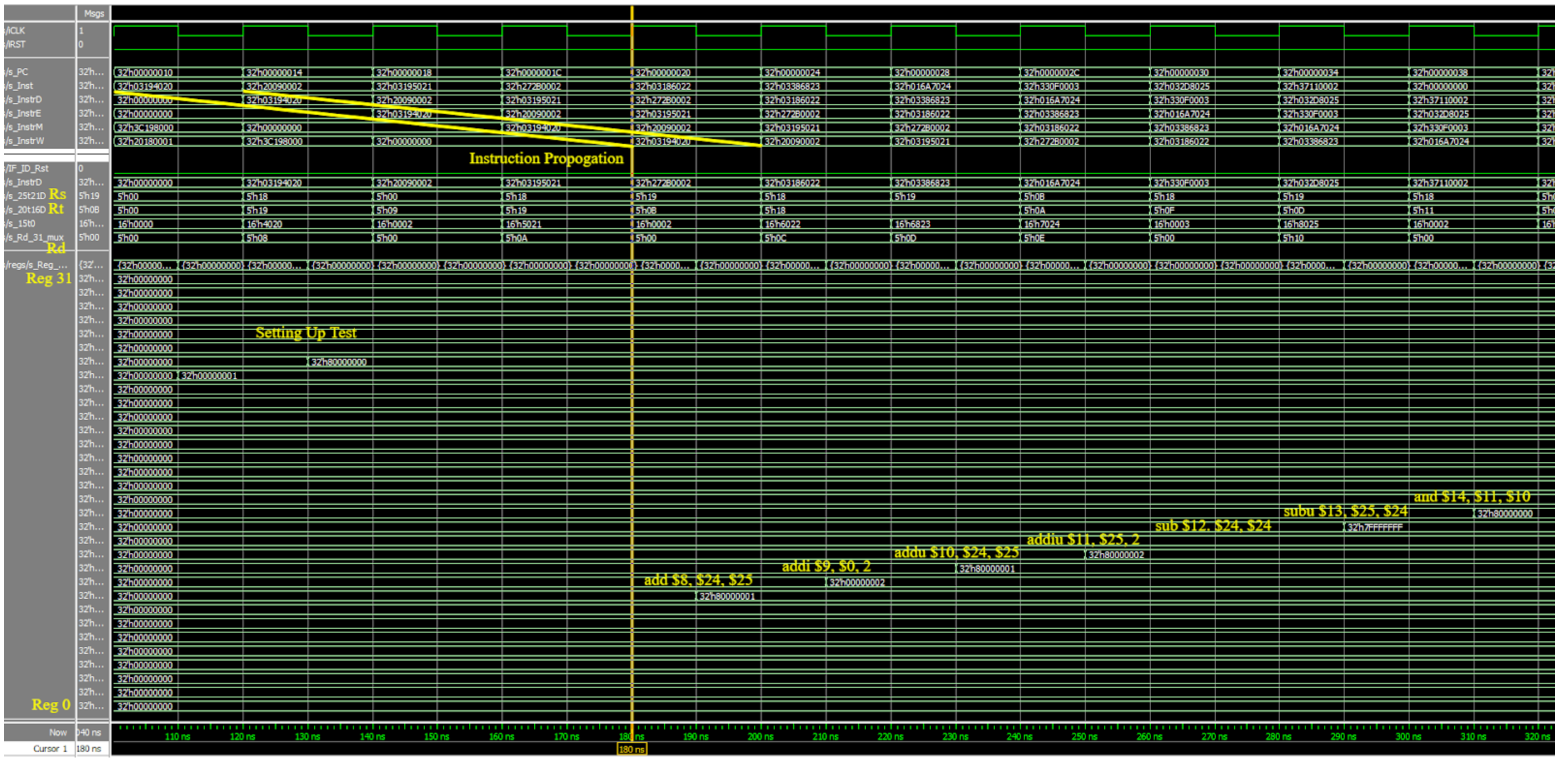
***Refer to the highlighted language in the Project 3 instruction for the context of the following questions*.**

a. [Part (a.2)] Provide a high-level schematic drawing of the interconnection between components.



b. [Part (a.3), Testing Program #1] Include an annotated waveform in your writeup and provide a short discussion of result correctness.

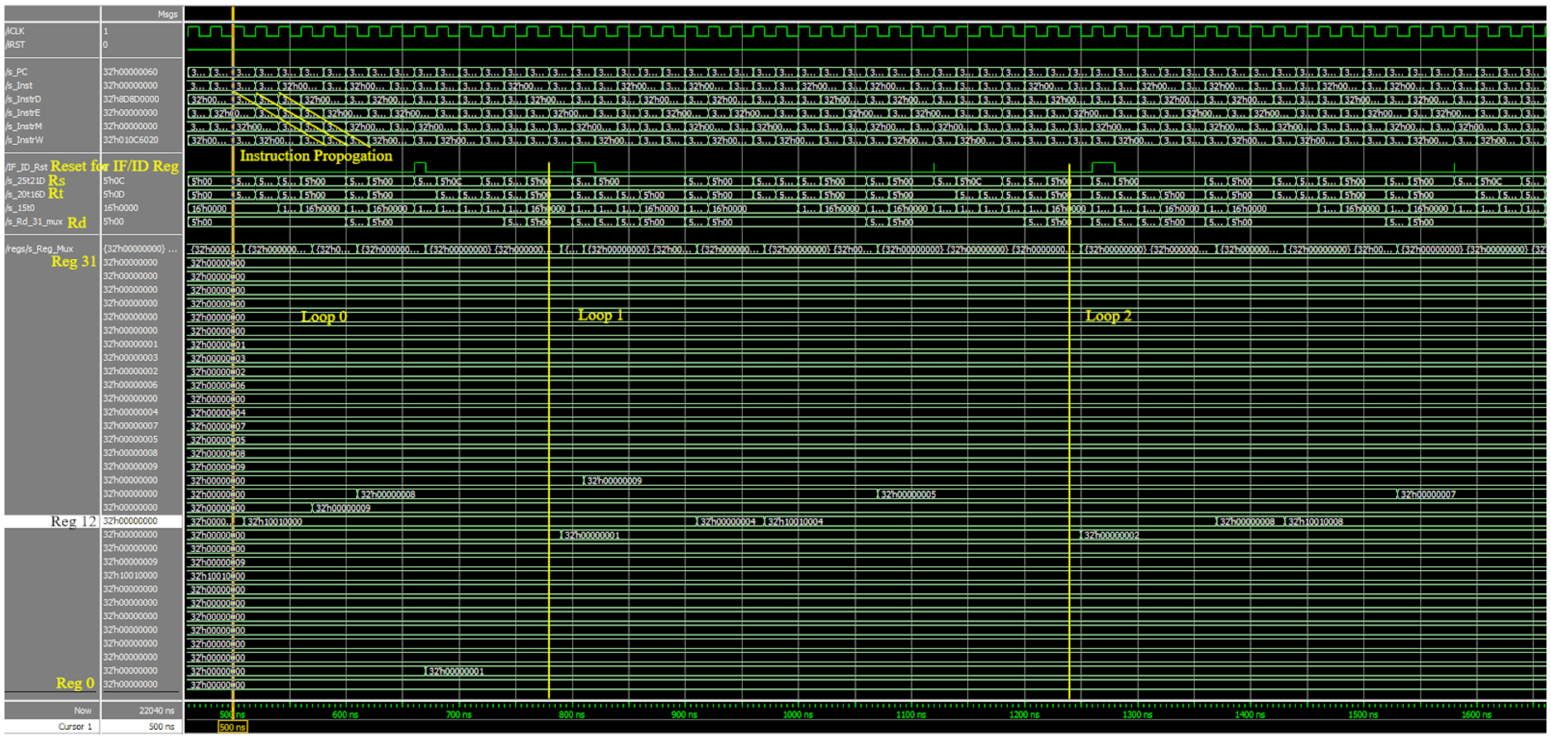
The file run here is "TestAllInstrNops.s"



As, throughout the running of the test, every instruction deposited the correct value into the correct register and every instruction that did not write to the RegFile performed its required function, I decree this test successful and the processor functional.

c. [Part (a.3), Testing Program #2-Bubble Sort] 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.

The file run here is “BubbleSortFancyNops.s”



Every loop annotated above is one run of the inner section of code in BubbleSortFancyNops.s. As when the final array was printed into registers 16-25 at the end of the program the numbers were in correct order, I decree this test successful and the processor functional.

d. [Part (a.3), not to be graded] How did you modify the bubble and merge sort so that they avoid all control and data hazard? By inserting NOPs? Or, by rearranging some instruction?
Inserted nops, it requires less movement.

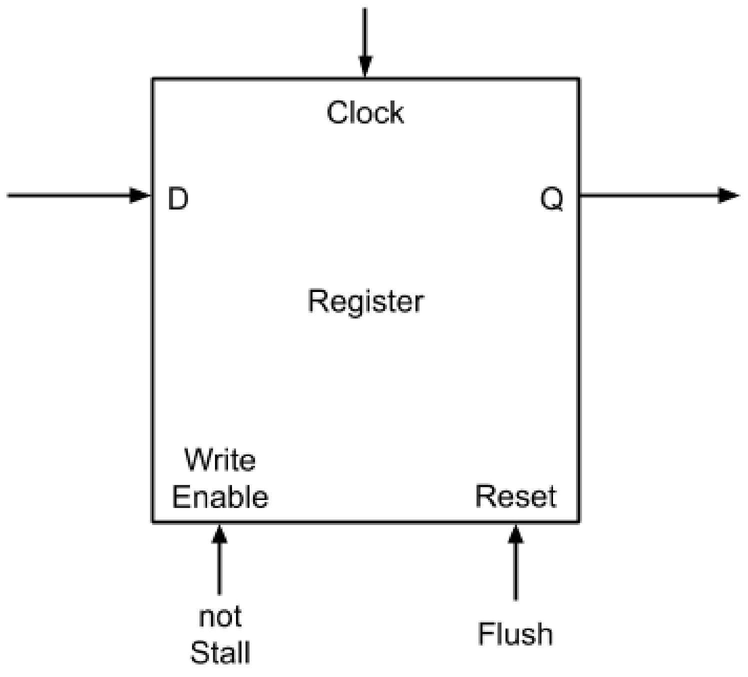
e. [Part (a.4)] Report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).
The maximum frequency the 3a processor can run on is 38.79 mHz. The critical path is the 'branch detection → IF/ID reset' path, as the branch detection logic can only run after the falling edge of the clock due to the register file. The output is as follows:

```
Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
    10.000    10.000          launch edge time
    19.731     9.731  F        clock network delay
    19.963     0.232     uTco  RegFile:regs|NBitReg:\G1:25:regs|o_Q[10]            Register File
    19.963     0.000 FF  CELL  regs|\G1:25:regs|o_Q[10]|q                           and register
    20.308     0.345 FF    IC  regs|multiplexor2|Mux21~2|datac                      multiplexor,
    20.589     0.281 FF  CELL  regs|multiplexor2|Mux21~2|combout                    using the output
    20.954     0.365 FF    IC  regs|multiplexor2|Mux21~3|datad                      of the regFile
    21.104     0.150 FR  CELL  regs|multiplexor2|Mux21~3|combout                    decoder
    22.161     1.057 RR    IC  regs|multiplexor2|Mux21~6|datac
    22.448     0.287 RR  CELL  regs|multiplexor2|Mux21~6|combout
    22.653     0.205 RR    IC  regs|multiplexor2|Mux21~9|datad
    22.808     0.155 RR  CELL  regs|multiplexor2|Mux21~9|combout
    23.012     0.204 RR    IC  regs|multiplexor2|Mux21~19|datad
    23.167     0.155 RR  CELL  regs|multiplexor2|Mux21~19|combout
    24.232     1.065 RR    IC  Equal0~6|dataa                                       Comparing
    24.661     0.429 RF  CELL  Equal0~6|combout                                     outputs from
    25.099     0.438 FF    IC  Equal0~9|dataa                                       register file
    25.452     0.353 FF  CELL  Equal0~9|combout                                     for branch
    25.820     0.368 FF    IC  Equal0~20|datac                                      logic
    26.101     0.281 FF  CELL  Equal0~20|combout
    26.339     0.238 FF    IC  IF_ID_Rst~0|datad                                    IF/ID reset
    26.464     0.125 FF  CELL  IF_ID_Rst~0|combout                                  combination
    28.950     2.486 FF    IC  IF_IDReg|InstrReg|o_Q~14|datad                       logic
    29.100     0.150 FR  CELL  IF_IDReg|InstrReg|o_Q~14|combout
    29.100     0.000 RR    IC  IF_IDReg|InstrReg|o_Q[28]|d
    29.187     0.087 RR  CELL  PipRegIF_ID:IF_IDReg|Reg32BitIF_ID:InstrReg|o_Q[28]
                                                                                    ^^ Final reset
                                                                                    signal, delayed
                                                                                    with a reg to
                                                                                    avoid issues with
                                                                                    delta cycle delay
```

f. [Part (b.5)] Update your global list of the datapath values and control signals that are required during each pipeline stage.

Incoming Signals per Section				
IF	**ID**	**EXE**	**MEM**	**WB**
JumpD	PC+4	ALUCtrl	MemWrite	RegWrite
JumpAddrD	Instr	ALUSrc	ALUOut	MemToReg
BranchCalcD	RegAddrWB	ReadData1	WriteData	ALUOut
BranchAddrD	WriteValWB	ReadData2	RegAddr	InstrRead
		SextImm	Lui	Lui
		Rs	Imm	RegAddr
		Rt		Imm
		Rd		
		RegDst		
		Imm		

g. [Part (b.6.a)] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.



h. [Part (b.6.b)] 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.

    Will probably do later. For now, the correct output of a fully functional pipelined processor below should suffice.

i. [Part (b.7.a)] List which instructions produce values, and what signals in the pipeline these correspond to.

Instr	Produce		Consumes		Consumes	
add	☑	(Rd)	☑	(Rs)	☑	(Rt)
addi	☑	(Rt)	☑	(Rs)	☐	
addu	☑	(Rd)	☑	(Rs)	☑	(Rt)
addiu	☑	(Rt)	☑	(Rs)	☐	
sub	☑	(Rd)	☑	(Rs)	☑	(Rt)
subu	☑	(Rd)	☑	(Rs)	☑	(Rt)
and	☑	(Rd)	☑	(Rs)	☑	(Rt)
andi	☑	(Rt)	☑	(Rs)	☐	
or	☑	(Rd)	☑	(Rs)	☑	(Rt)
ori	☑	(Rt)	☑	(Rs)	☐	
nor	☑	(Rd)	☑	(Rs)	☑	(Rt)
xor	☑	(Rd)	☑	(Rs)	☑	(Rt)
sll	☑	(Rd)	☑	(Rt)	☐	
srl	☑	(Rd)	☑	(Rt)	☐	
sra	☑	(Rd)	☑	(Rt)	☐	
slt	☑	(Rd)	☑	(Rs)	☑	(Rt)
sltu	☑	(Rd)	☑	(Rs)	☑	(Rt)
slti	☑	(Rt)	☑	(Rs)	☐	
sltiu	☑	(Rt)	☑	(Rs)	☐	
lw	☑	(Rt)	☑	(Rs)	☐	
lui	☑	(Rt)	☐		☐	
sw	☐		☑	(Rs)	☐	
beq	☐		☑	(Rs)	☐	
bne	☐		☑	(Rs)	☐	
j	☐		☐		☐	
jal	☑	($31)	☐		☐	
jr	☐		☑	(Rs)	☐	

j. [Part (b.7.b)] List which of these same instructions consume values, and what signals in the pipeline these correspond to
All immediates are sent to “Imm”, others are passed through Rs and Rt.

k. [Part (b.7.c)] Come up with a generalized list of potential data dependencies. From this generalized list, select those dependencies that will require forwarding (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.
Any instruction that consumes a value produced by an instruction one or two prior will have a data dependency.

l. [Part (b.8)] Write a more generalized series of data forwarding and hazard detection logic equations based on the result from the previous part. These should be of the format of those found in P&H 4.7, but there will be several more types of dependencies to consider.
Pulled from my main schematic, the numbers refer to hazard signal superscripts:

Hazard Reference List:

1. These signals are used to forward ALU output, detected by:
   ID_Rs == MEM_RegAddr <-- A (Same order for below)
   ID_Rt == MEM_RegAddr <-- B (Same order for below)
   Operates iff MEM_RegWrite == 1 and MEM_RegAddr != $0

2. These signals are used to forward ALU output, detected by:
   EXE_Rs == MEM_RegAddr
   EXE_Rt == MEM_RegAddr
   Operates iff MEM_RegWrite == 1 and MEM_RegAddr != $0

3. These signals are used to forward DMem output, detected by:
   EXE_Rs == WB_RegAddr
   EXE_Rt == WB_RegAddr
   Operates iff WB_RegWrite == 1 and WB_RegAddr != $0

4. These signals are used to stall the pipeline. Used together, this holds current instr. in ID and lets EXE, MEM, & WB propagate. There are many situations where this is used, logic listed below:

   Branch/Jump needs ALU or DMem of prev instr. or
   ID_Rs == EXE_RegAddr
   ID_Rt == EXE_RegAddr
   Operates iff EXE_RegWrite == 1 and EXE_RegAddr != $0
   (Checking EXE_MemtoReg == 1 is redundant here)

   Branch/Jump needs DMem of instr. 2 prev
   ID_Rs == MEM_RegAddr
   ID_Rt == MEM_Reg Addr
   Operates iff MEM_RegWrite == 1 and MEM_RegAddr != $0
   and MEM_MemtoReg == 1
   and (not (ForwardA or ForwardB)) (^{1})

   ALU needs DMem of prev instr.
   (ALU <- DMem calculation must be done w/ ID, early enough to stall)
   ID_Rs == EXE_RegAddr
   ID_Rt == EXE_RegAddr
   Operates iff EXE_RegWrite == 1 and EXE_RegAddr != $0
   and EXE_MemtoReg == 1
   and (not (ForwardA2 or ForwardB2)) (^{2})

5. Beware, if stalling to forward for a branch or jump, PC will be stalled, so the new address will not take unless PC stall is changed:
   PCStall = (Stall) or (Jump/Branch IF/ID reset signal)

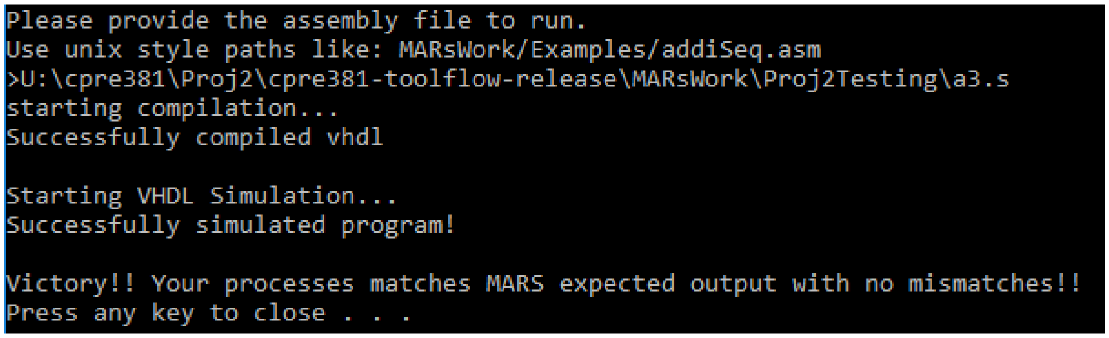
m. [Part (b.9)] Provide a high-level schematic drawing of the interconnection between components. Refer to main schematic below.

n. [Part (b.10)] In your writeup, show the ModelSim output for each of the following tests, and provide a discussion of results correctness. It may be helpful to also annotate the waveform directly

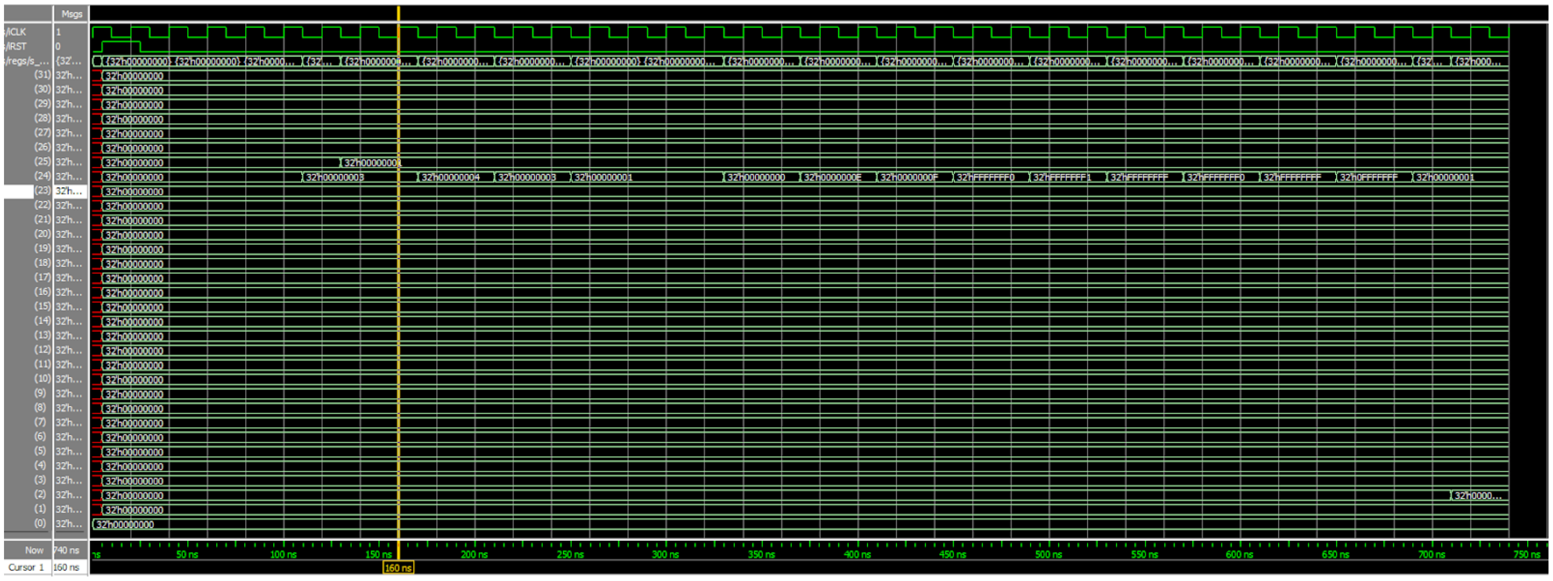
   a. Verify that your three test applications from Project Part 2 work on this processor without being modified.

Each of the files for part a was run directly from the Proj2 MARsWork testing directory.

The wlf form corresponding to this test is "Proj2_a3.wlf"

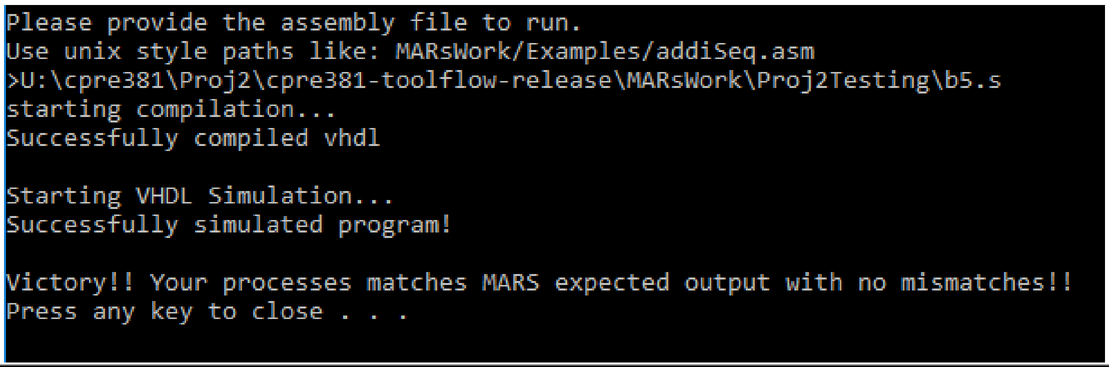


```
Please provide the assembly file to run.
Use unix style paths like: MARsWork/Examples/addiSeq.asm
>U:\cpre381\Proj2\cpre381-toolflow-release\MARsWork\Proj2Testing\a3.s
starting compilation...
Successfully compiled vhdl

Starting VHDL Simulation...
Successfully simulated program!

Victory!! Your processes matches MARS expected output with no mismatches!!
Press any key to close . . .
```

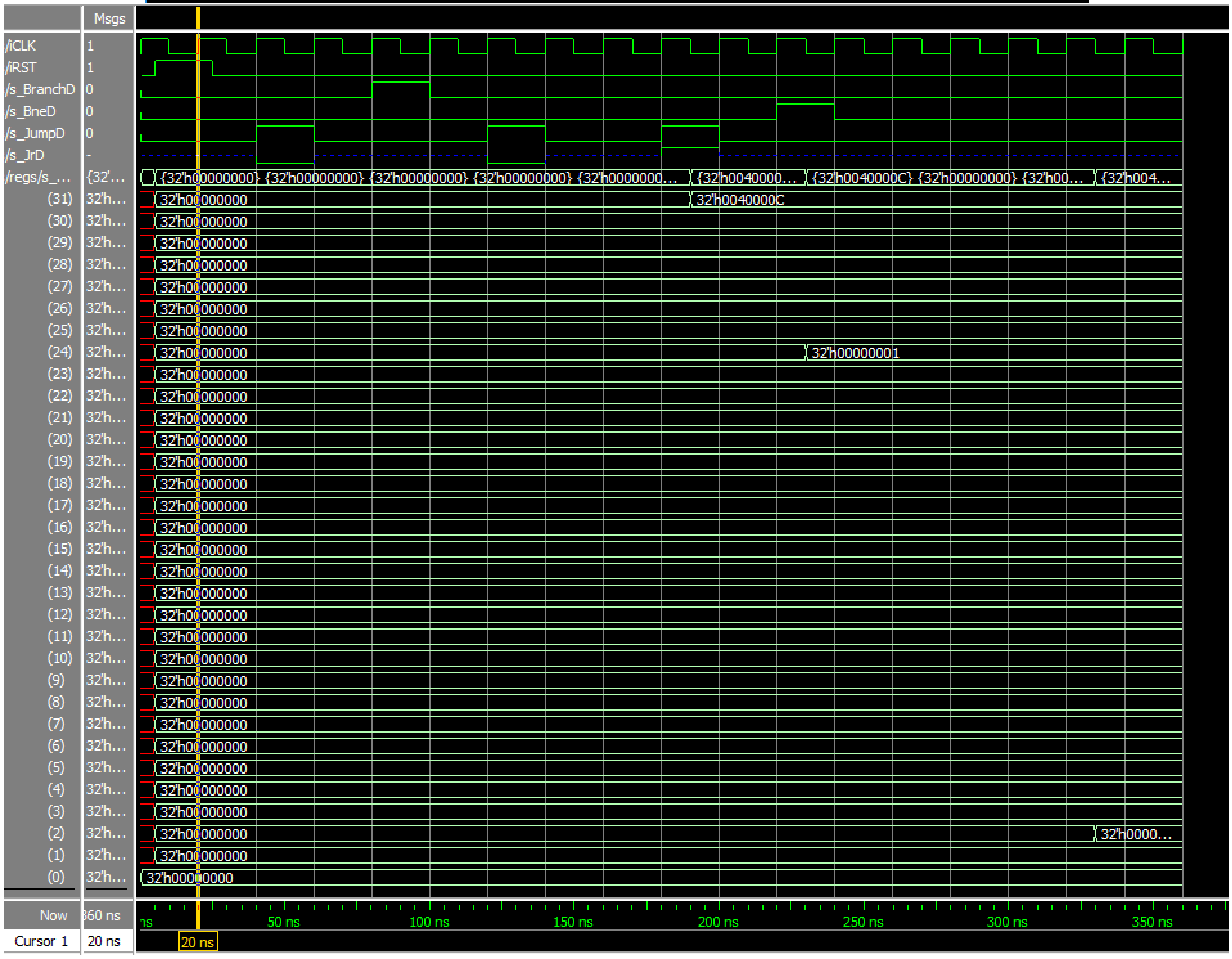

Msgs
/iCLK
/iRST
/regs/s_...
(31) 32'h...
(25) 32'h00000001
(24) 32'h00000000 32'h00000003 32'h00000004 32'h00000003 32'h00000001 32'h00000000 32'h0000000E 32'h0000000F 32'hFFFFFFF0 32'hFFFFFFF1 32'hFFFFFFFF 32'hFFFFFFF0 32'hFFFFFFFF 32'h0FFFFFFF 32'h00000001
(0) 32'h00000000
Now 740 ns
Cursor 1 160 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
350 ns
400 ns
450 ns
500 ns
550 ns
600 ns
650 ns
700 ns
750 ns

For every instruction, register 24 was updated with the correct value, so I would assume the processor was functioning correctly.

The wlf form corresponding to this test is “Proj2_b5.wlf”

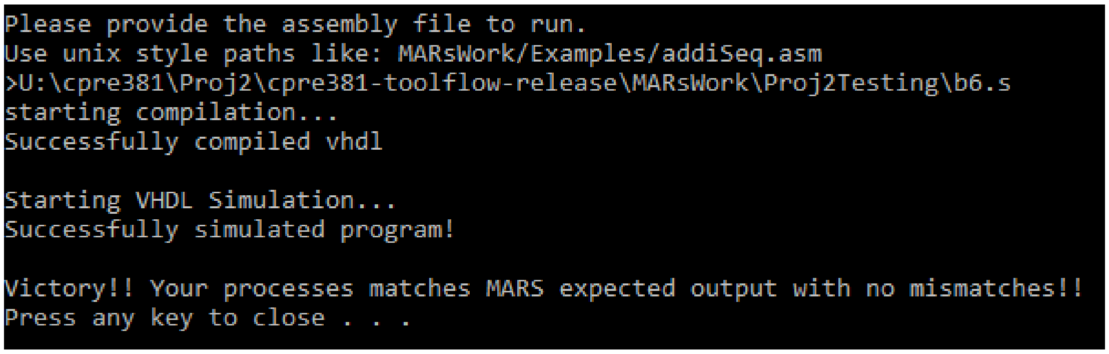


```
Please provide the assembly file to run.
Use unix style paths like: MARsWork/Examples/addiSeq.asm
>U:\cpre381\Proj2\cpre381-toolflow-release\MARsWork\Proj2Testing\b5.s
starting compilation...
Successfully compiled vhdl

Starting VHDL Simulation...
Successfully simulated program!

Victory!! Your processes matches MARS expected output with no mismatches!!
Press any key to close . . .
```

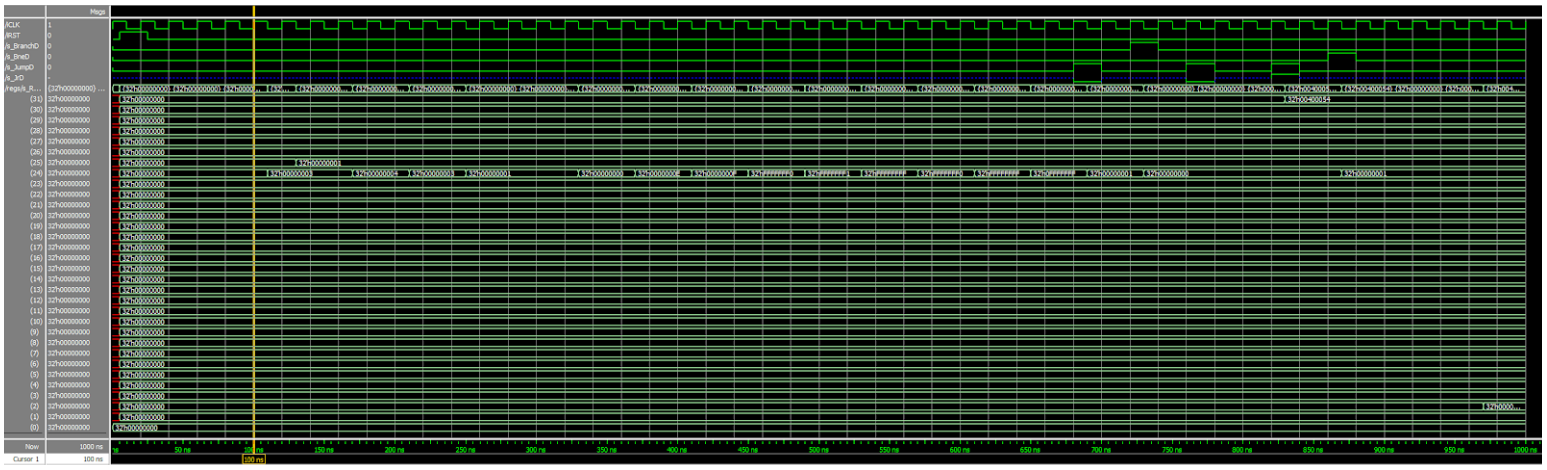


Every branch/jump cleared the IF/ID pipeline register as intended, and skipped the ‘got you’ instructions inserted after the branches in the code, while every other instruction executed as intended, so I presume the processor was operating correctly.

The wlf form corresponding to this test is “Proj2_b6.wlf”



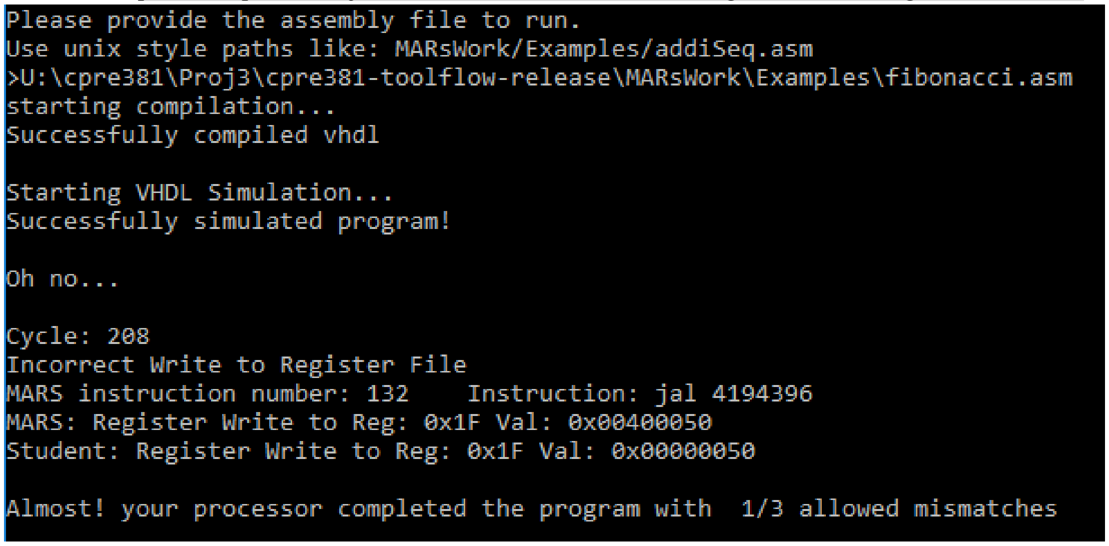
```
Please provide the assembly file to run.
Use unix style paths like: MARsWork/Examples/addiSeq.asm
>U:\cpre381\Proj2\cpre381-toolflow-release\MARsWork\Proj2Testing\b6.s
starting compilation...
Successfully compiled vhdl

Starting VHDL Simulation...
Successfully simulated program!

Victory!! Your processes matches MARS expected output with no mismatches!!
Press any key to close . . .
```



Every instruction that was meant to run deposited the correct value into the correct register, while the jumps and branches ran at the correct times and reset the IF/ID register correctly, so I deem this test successful and this processor functional.

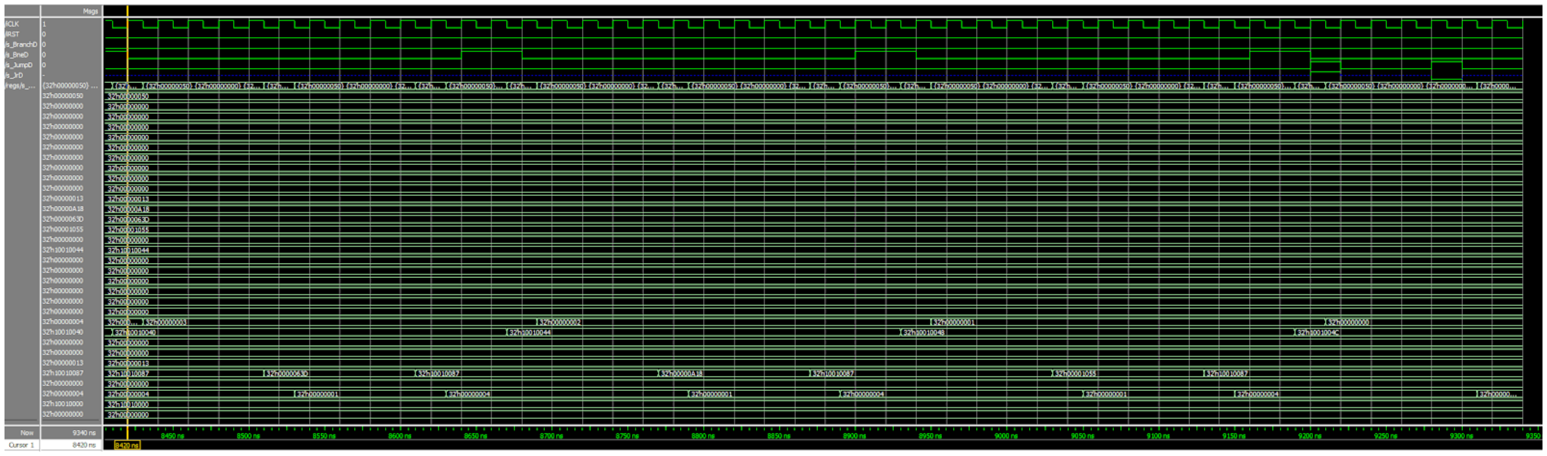
b. Create an application that tries to exhaustively test the forwarding and hazard detection capabilities of your pipeline.
Creating the extra tests for this writeup takes longer than it would take to physically run from the sun to the moon, and I have other things I need to do, so I just used fibonacci.asm for this section as that should *mostly* cover this requirement.

***Note:** MARS starts the program at instruction memory location **0x0040_0000**, while this processor starts the program at instruction memory location **0x0000_0000**, so when the processor performs '**jal**', different values are written to register $31, causing the error below.



```
Please provide the assembly file to run.
Use unix style paths like: MARsWork/Examples/addiSeq.asm
>U:\cpre381\Proj3\cpre381-toolflow-release\MARsWork\Examples\fibonacci.asm
starting compilation...
Successfully compiled vhdl

Starting VHDL Simulation...
Successfully simulated program!

Oh no...

Cycle: 208
Incorrect Write to Register File
MARS instruction number: 132    Instruction: jal 4194396
MARS: Register Write to Reg: 0x1F Val: 0x00400050
Student: Register Write to Reg: 0x1F Val: 0x00000050

Almost! your processor completed the program with  1/3 allowed mismatches
```

Final section of "fibonacci.asm" displayed below:



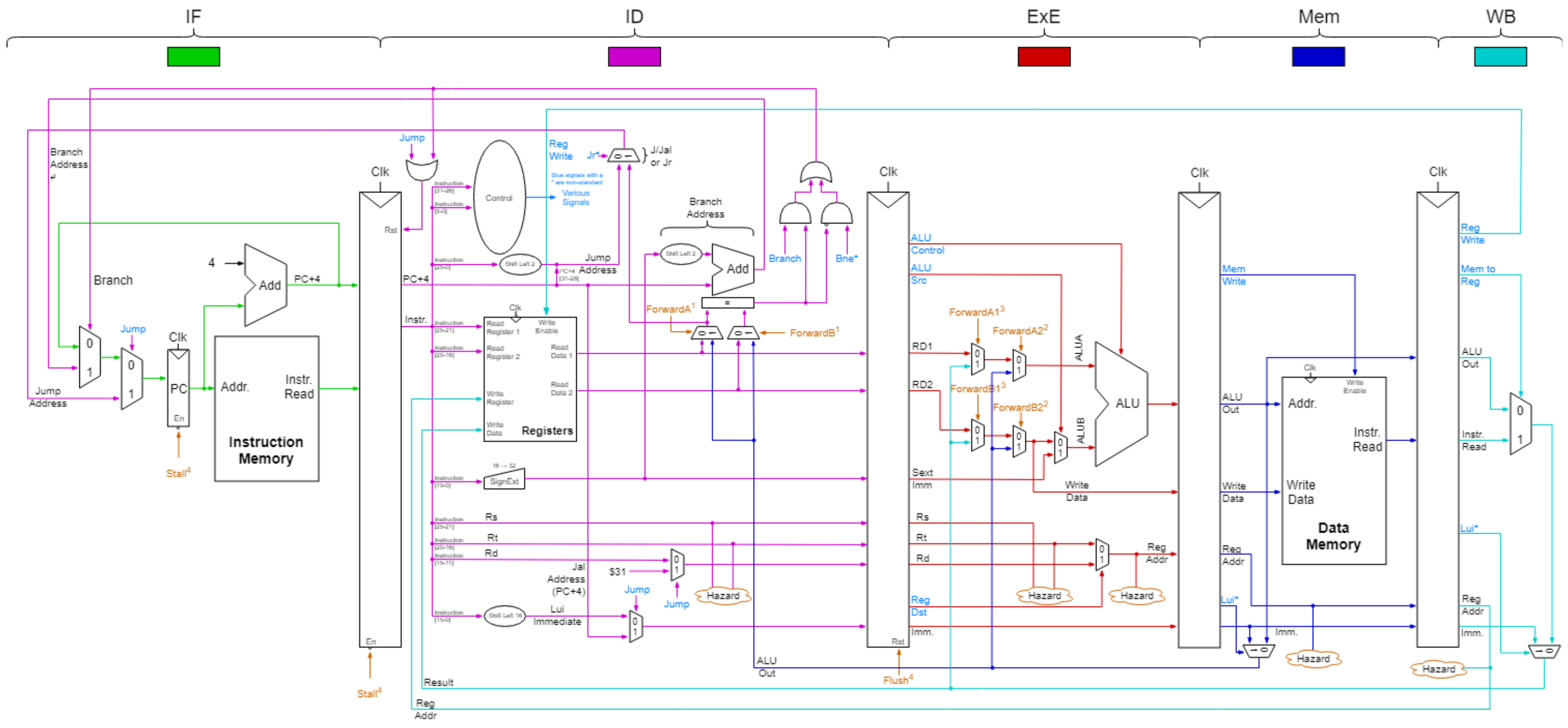
The program completed with no real errors, meaning every jump, branch, register write, and otherwise was completed successfully, I deem this processor fully functional.

o. [Part (b.12)] 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).

The maximum frequency the 3b processor can run on is 39.40 mHz. The critical path is the 'branch detection → PC enable' path, as the branch detection logic can only run after the falling edge of the clock due to the register file. This PC logic is necessary so as not to stall the PC register during a forward if a branch or a jump needs to update its value (number 5 in hazard logic described above). The output is as follows:

```
Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
    10.000     10.000          launch edge time
    19.315      9.315  F       clock network delay
    19.547      0.232     uTco  RegFile:regs|NBitReg:\G1:23:regs|o_Q[23]          Register File
    19.547      0.000 FF  CELL  regs|\G1:23:regs|o_Q[23]|q                        and register
    20.070      0.523 FF    IC  regs|multiplexor1|Mux8~9|datad                    multiplexor,
    20.195      0.125 FF  CELL  regs|multiplexor1|Mux8~9|combout                  using the output
    20.427      0.232 FF    IC  regs|multiplexor1|Mux8~10|datac                   of the RegFile
    20.708      0.281 FF  CELL  regs|multiplexor1|Mux8~10|combout                 decoder
    22.018      1.310 FF    IC  regs|multiplexor1|Mux8~11|datac
    22.299      0.281 FF  CELL  regs|multiplexor1|Mux8~11|combout
    22.527      0.228 FF    IC  regs|multiplexor1|Mux8~16|datad
    22.652      0.125 FF  CELL  regs|multiplexor1|Mux8~16|combout
    22.882      0.230 FF    IC  regs|multiplexor1|Mux8~19|datad
    23.032      0.150 FR  CELL  regs|multiplexor1|Mux8~19|combout
    23.260      0.228 RR    IC  ForA_MuxD|o_F[23]~22|datad                        Forwarding mux
    23.415      0.155 RR  CELL  ForA_MuxD|o_F[23]~22|combout                      for branch logic
    23.641      0.226 RR    IC  Equal0~31|datac                                   Comparing
    23.928      0.287 RR  CELL  Equal0~31|combout                                 outputs from
    24.627      0.699 RR    IC  Equal0~32|datac                                   register file
    24.894      0.267 RF  CELL  Equal0~32|comboutt                                for branch
    25.165      0.271 FF    IC  Equal0~38|datab                                   logic
    25.515      0.350 FF  CELL  Equal0~38|combout
    25.761      0.246 FF    IC  s_BranchOrD~0|datac                               Branch OR
    26.042      0.281 FF  CELL  s_BranchOrD~0|combout                             logic
    27.070      1.028 FF    IC  IF_ID_Rst~0|datac                                 IF/ID reset
    27.350      0.280 FF  CELL  IF_ID_Rst~0|combout                               logic
    27.620      0.270 FF    IC  PCReg|o_Q[0]|ena                                  PC register
28.299      0.679 FF  CELL  NBitReg:PCReg|o_Q[0]                                  enable logic
```

Main Schematic:



*Note: In the above schematic, I display the pipeline registers as updating on the falling edge of the clock, when in fact they update on the rising edge. This was simply something I forgot to change.

Link to schematic in draw.io:
https://drive.google.com/file/d/19Vn-4LoZGcJRO0gRVSQViPrvZdb4JGhl/view?usp=sharing