CprE 381 – Computer Organization and Assembly-Level Programming

Proj-B Report

WORKS: MarsWork/Examples/addiSeq.asm
WORKS: MarsWork/Examples/lab3Seq.asm
WORKS: MarsWork/Examples/fibonacci.asm
WORKS: MarsWork/Examples/Proj-B_test1.s
WORKS: MarsWork/Examples/Proj-B_test2.s
WORKS: MarsWork/Examples/Proj-B_test3.s

Lab Partners	Jonathon Schnell
	Nicholas Krabbenhoft
	Constantine Mantas
Section / Lab Time	7 Thursday 2-4pm

Submit a typeset pdf version of this on Canvas by the due date. Refer to the highlighted language in the Proj-B instructions for the context of the following questions.

a. [Part 0] How are instruction and data memory initialized in the simulation? How does MARS interface with ModelSim?

instructions are synthesised and initialized from a mips program with the test_framework python script. Mars is not directly interfacing with model sim, instead the script creates a log that represents the state of the data memory cycle by cycle, these logs are compared to look for errors in the VHDL implementation.

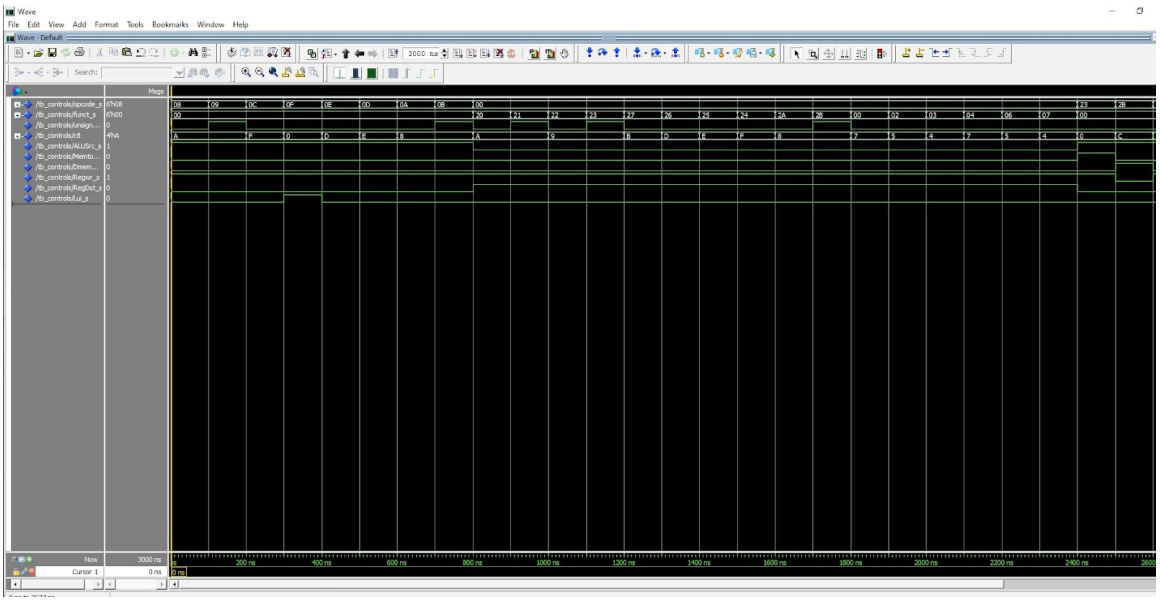
b. [Part 0] In the MIPS skeleton VHDL, how is a halting / termination condition detected? What MIPS assembly instruction does this correspond to?

mips has a nop instruction but to halt/terminate but it does have a NOP instruction that does nothing for one clock cycle. In mars we terminate a program by logging the value 10 into register $v0 and doing a syscall. The same approach is taken in the testbed where we ser a signal $v0 to 10 and then run the syscall instruction.

c. [Part 1 (a)] Modify the provided spreadsheet to include the list of *M* instructions to be supported in this phase alongside their binary Instruction OPcodes and Funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed in your single-cycle processor implementation. The end result should be an *M***N* table where each row corresponds to the output of the control logic module for a given instruction. *[You can attach the spreadsheet as a separate file, with a single spreadsheet for all Phase I and Phase II instructions.]*

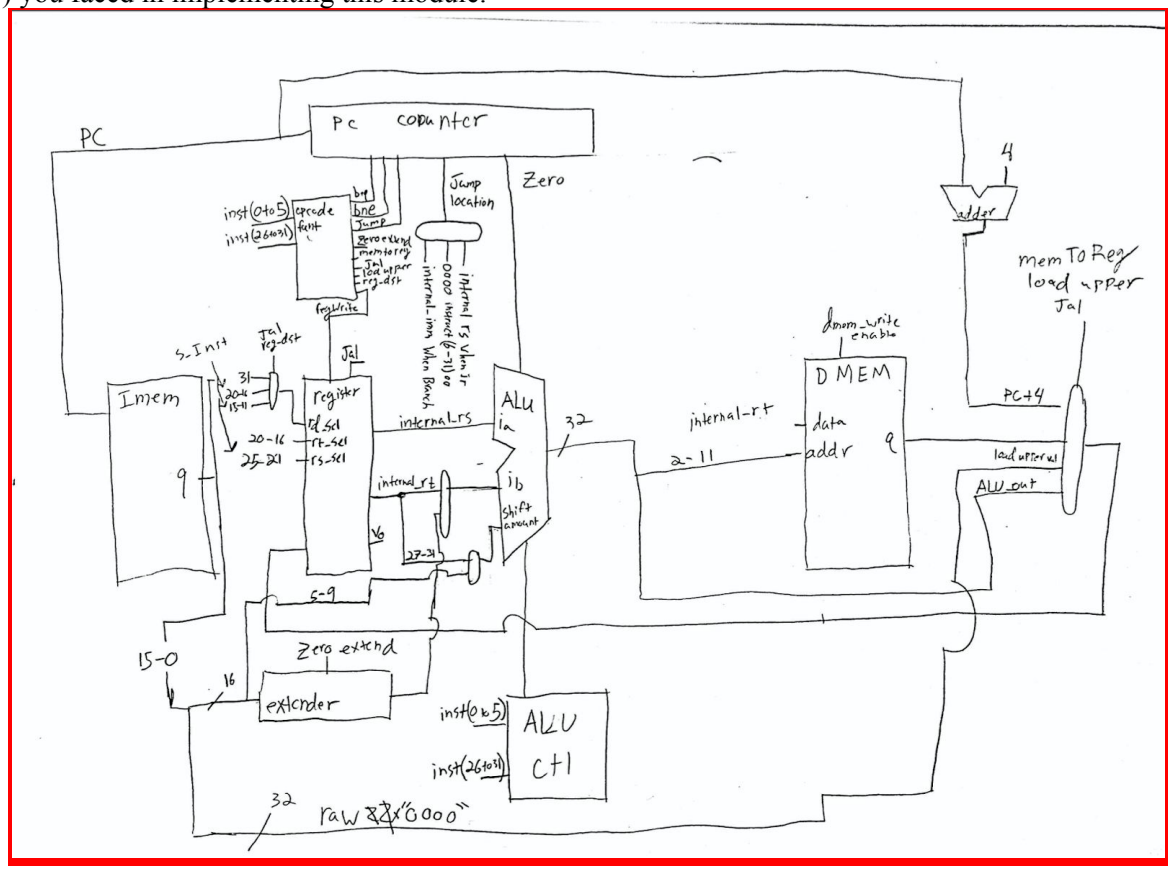
https://docs.google.com/spreadsheets/d/1XykQ-zra1M6vLSZCR4-eODjaHXZTPZKX/edit#gid=495437357

d. [Part 1 (b)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually, and show that your output matches the expected control signals from part 1a). *[Please include waveforms and explanations.]*



This waveform demonstrates the outputs of the controls when given certain inputs for opcode and funct code. Each case is associated with one of the instructions on the datatable to ensure that each combination works as intended.

e. [Part 2] In your writeup, provide your schematic for this part, describe what challenges (if any) you faced in implementing this module.

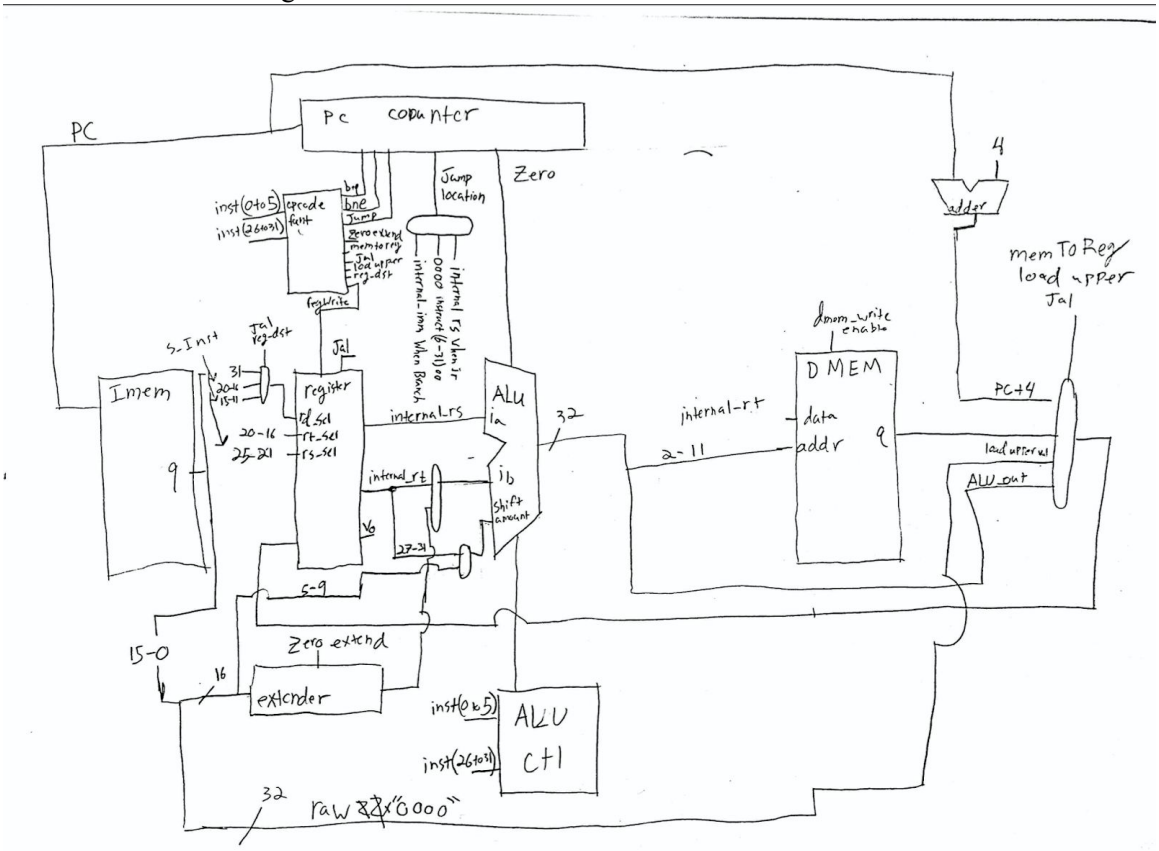


There were no major problems with designing the logic for this part. Most of the problems come from mistakes in writing out the code and assigning signals.

f. [Part 4 (a)] Describe these possibilities as a function of the different control flow-related instructions.

Branch: PC += offset *4
Jump: pc=pc_upper|(target<<2)
JAL: r31=pc; pc=target<<2
JR: pc=rs

g. [Part 4 (c)] Update your processor schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. In your writeup, describe what additional control signals are needed.



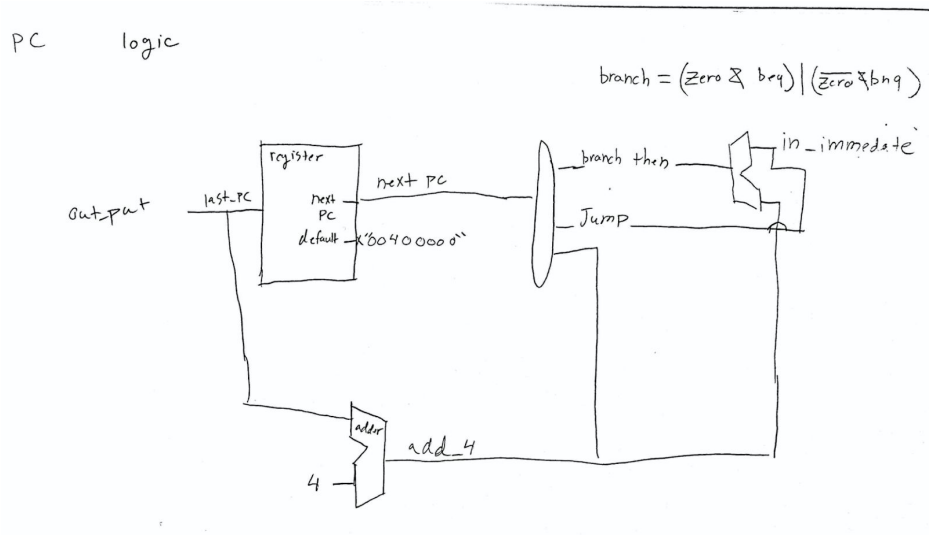
Jump- signal to indicate to PC if the instruction is j, jr, jal so that the pc can jump to that offset

Branch- Two signals for the different types of branch instructions to allow the PC controller to combine both with the zero input to figure out if it should execute the branch and let the jump location mux to know what immediate value to input

Jal - signal to let the register file to know to store the current counter plus 4.

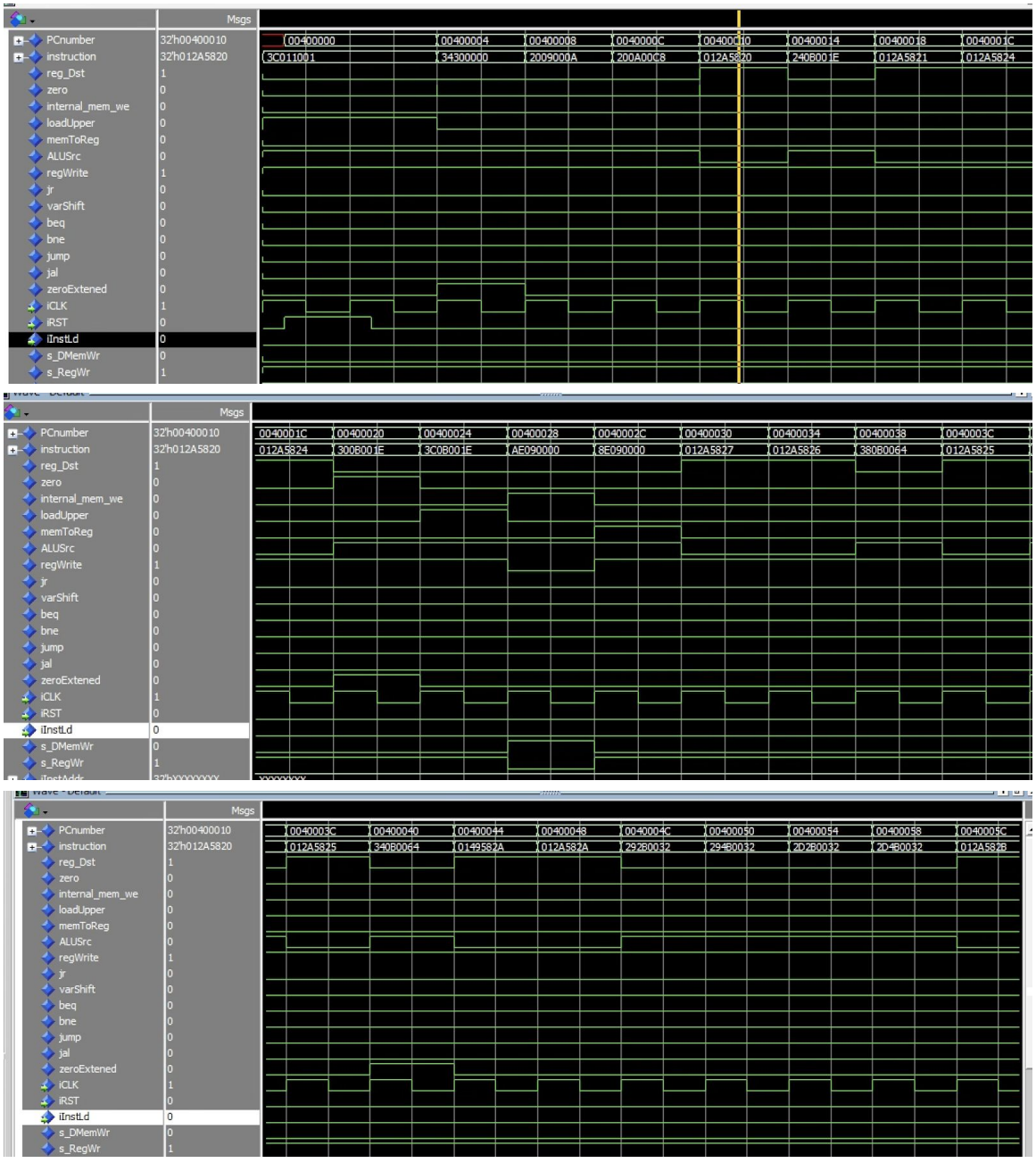
Jr allows the jump location mux to know what immediate value to input

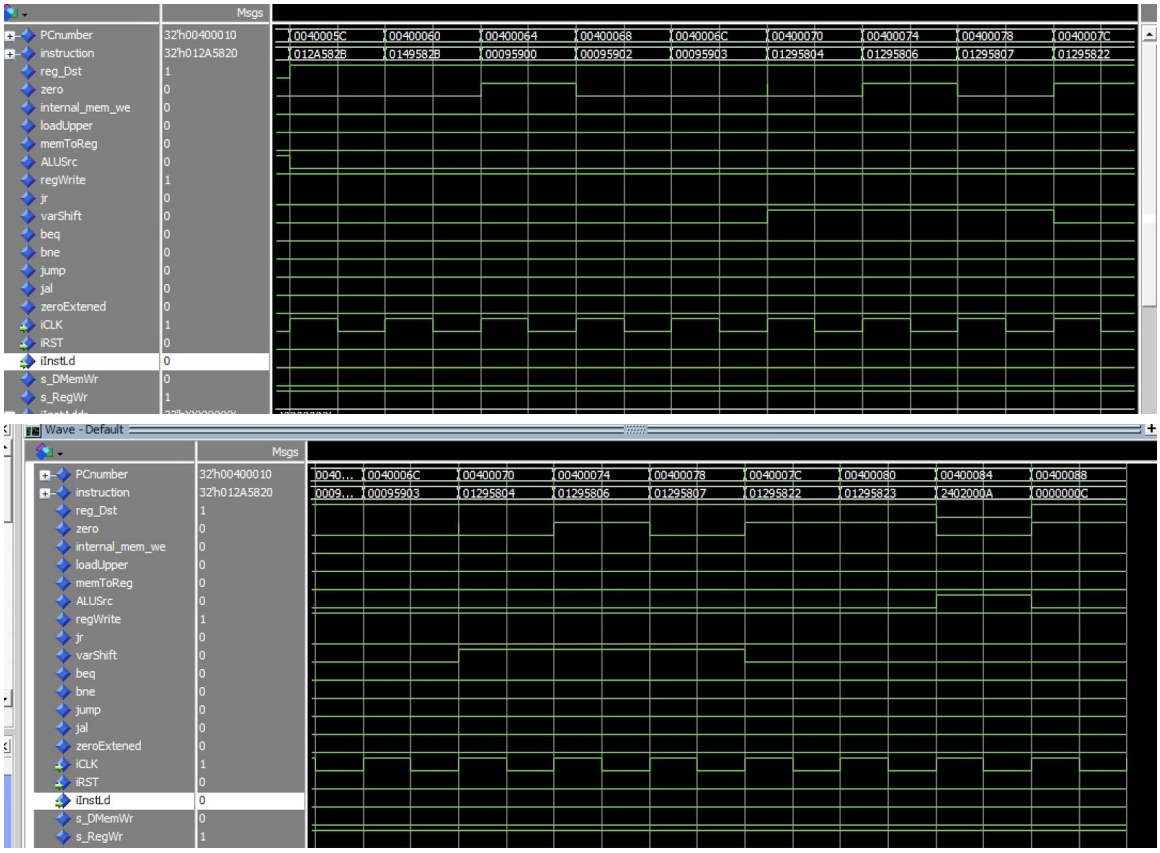
Inside the PC counter, it uses the branch and jump signals to determine how to treat the immediate value passed into it.



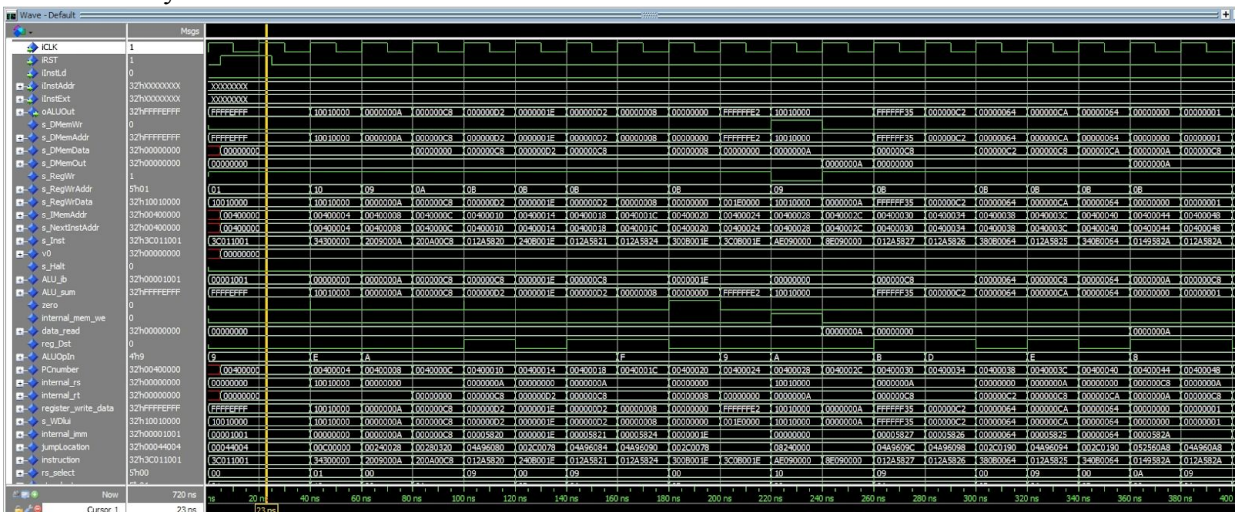
h. [Part 4 (d)] Include ModelSim waveforms, and describe how the execution of the control flow possibilities corresponds to the waveforms in your writeup.

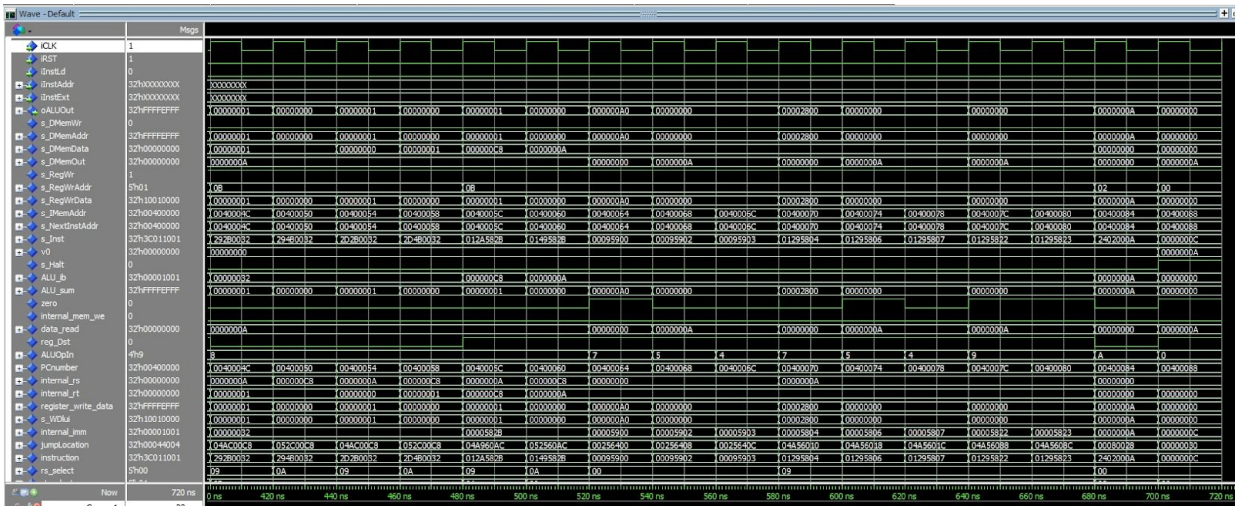
Here are a set of screenshots from running the test1 program. All of the control signals are on the top of the diagram right below the instruction number and the hex value of the instruction. The program resulted in no miss matches with MARS and every spot check that I have done shows the signals corresponding to the correct values for the instructions.


Msgs
PCnumber 32'h00400010
instruction 32'h012A5820
reg_Dst 1
zero 0
internal_mem_we 0
loadUpper 0
memToReg 0
ALUSrc 0
regWrite 1
jr 0
varShift 0
beq 0
bne 0
jump 0
jal 0
zeroExtened 0
iCLK 1
iRST 0
iInstLd 0
s_DMemWr 0
s_RegWr 1
00400000 00400004 00400008 0040000C 00400010 00400014 00400018 0040001C
3C011001 34300000 2009000A 200A00C8 012A5820 240B001E 012A5821 012A5824
Msgs
PCnumber 32'h00400010
instruction 32'h012A5820
reg_Dst 1
zero 0
internal_mem_we 0
loadUpper 0
memToReg 0
ALUSrc 0
regWrite 1
jr 0
varShift 0
beq 0
bne 0
jump 0
jal 0
zeroExtened 0
iCLK 1
iRST 0
iInstLd 0
s_DMemWr 0
s_RegWr 1
0040001C 00400020 00400024 00400028 0040002C 00400030 00400034 00400038 0040003C
012A5824 300B001E 3C0B001E AE090000 8E090000 012A5827 012A5826 380B0064 012A5825
Msgs
PCnumber 32'h00400010
instruction 32'h012A5820
reg_Dst 1
zero 0
internal_mem_we 0
loadUpper 0
memToReg 0
ALUSrc 0
regWrite 1
jr 0
varShift 0
beq 0
bne 0
jump 0
jal 0
zeroExtened 0
iCLK 1
iRST 0
iInstLd 0
s_DMemWr 0
s_RegWr 1
0040003C 00400040 00400044 00400048 0040004C 00400050 00400054 00400058 0040005C
012A5825 340B0064 0149582A 012A582A 292B0032 294B0032 2D2B0032 2D4B0032 012A582B



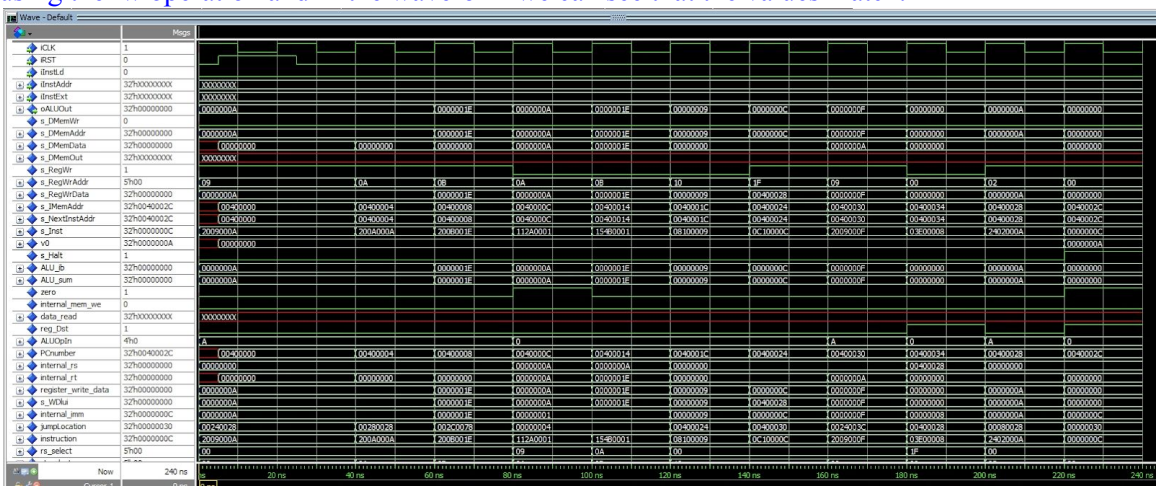
i. [Part 5 (a)] In your writeup, show the ModelSim output for the individual instruction tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.





the above screenshots are from test1.s
there are many tests here so I will go over a couple of them:

1. addi $1, $zero, 10 this is used to test I-type instruction. we see that the RS and RT fields have the proper addresses of 0 and 10 we see the write enable go high and we also see the write signal pointing to the correct port on the reg file and ultimately the output of the alu is correct and therefore the value of 10 is now stored in register 10.
2. add $t3, $t1, $t2 for this instruction we are testing the R-type instruction.we can see a similar situation as previously except we are no longer taking an immediate value out of the instruction. the previously stored values (10,20) are read from the regfile and the ALU successfully writes back the value of 30 to t3.
3. sw $1, 0 ($s0) && lw $1, 0 ($s0) to test functionality of the memory, in these instructions we are storing a loading the pointer to the head of an arbitrary array, in the waveform we can see the when storing the word the address is not written back to the regfile, instead it is stored in the memory, we are able to demonstrate that this value is properly stored by using the lw operation and in the waveform we can see that the values match.



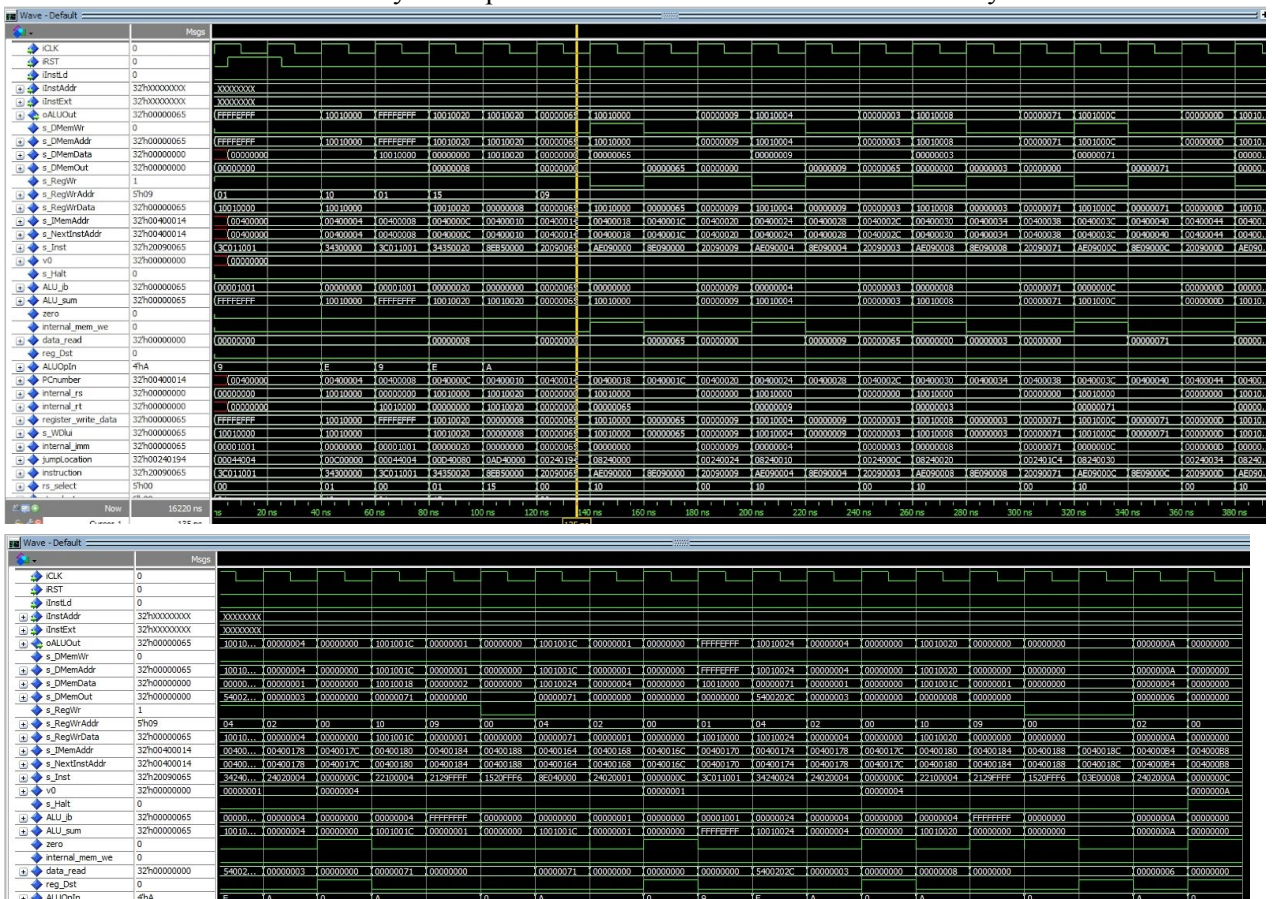
the above screenshot shows the test2 being run
again i'll elaborate on a couple of these

1. beq $t1, $t2, A for this test we expect to see the program counter branch to the address of A in the instruction memory because $t1 and $t2 have both been loaded with the value of 10. we see that the program successfully branches because the pc updates and a test add instruction after this beq instruction is

skipped and cannot be seen in the waveform. The same testing methodology is used for the bne except we pass different registers that we know are not equal.

2. j c, to test this we follow a similar method as before we should see the program counter jump to skip an instruction because the following instruction should not be executed. this behavior can be seen in the waveform not only because the following instruction does not get executed but also the PC jumps 8 instead of 4 to skip the following instruction.

j. [Part 5 (b)] In your writeup, show the ModelSim output for the Bubblesort test, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.



above is a portion of the test3 or bubblesort application, only a portion because the dynamic instruction count is over 800 instructions for this program with an array of 10 elements. Here we can see the first few instructions and the last few instructions. overall we have a good idea that everything is running alright because there are no undriven signals. we can confirm correctness by looking at the last few instructions that are printing out each value in the array followed by a comma. If we inspect these last few cycles we can see that the values output by the ALU and therefore being printed are printed in ascending order meaning that all the previous instructions properly did what they were supposed to.

k. [Part 5 (c) BONUS] In your writeup, show the ModelSim output for the Mergesort test, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

N/A

l. [Part 7] Report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematic. In your writeup, briefly discuss your critical path results. What components would you focus on to improve the frequency?

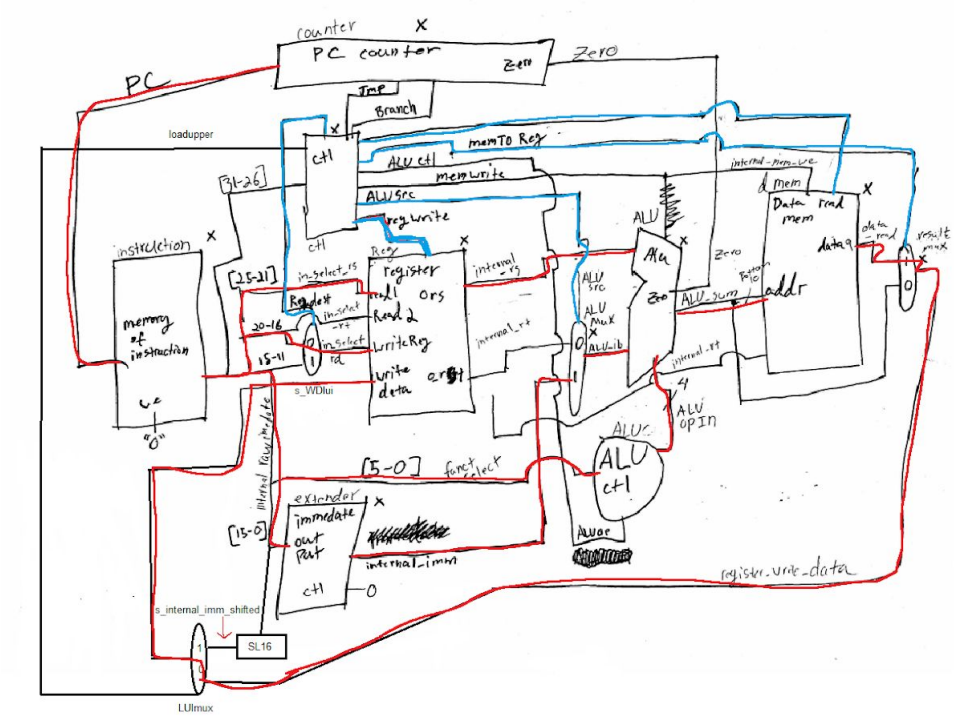
Fmax = 24.48mhz

critical path (lw)

```
FMax: 24.48mhz Clk Constraint: 20.00ns Slack: -20.85ns

The path is given below

 ====================================================================
 From Node    : pc:counter|register_nbit_struct:reg|dffg:\G1:25:dff_i|s_Q
 To Node      : registerFile_nbit_struct:reg|register_nbit_struct:\G0:18:register_j|dffg:\G1:0:dff_i|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ===================================
      0.000     0.000           launch edge time
      3.083     3.083  R        clock network delay
      3.315     0.232     uTco  pc:counter|register_nbit_struct:reg|dffg:\G1:25:dff_i|s_Q
      3.315     0.000 FF  CELL  counter|reg|\G1:25:dff_i|s_Q|q
      3.662     0.347 FF    IC  s_IMemAddr[6]~0|datad
      3.787     0.125 FF  CELL  s_IMemAddr[6]~0|combout
      6.119     2.332 FF    IC  IMem|ram~37630|dataa
      6.543     0.424 FF  CELL  IMem|ram~37630|combout
```



The critical path of our processor is that of many if not all MIPS processors, that is the load word path. This is because to load a word we have to use every major component of

the processor. above i've labeled active control signals in blue and the datapath itself in red.

looking at the data arrival path time we can see a couple components that take up a good percentage more than other components. For example in the barrel shifter the stage 5 shift or shift by 16 takes almost 5ns while the stage one shift takes about 2ns, I think this is because to get to this stage the data must first go through the other 4 stages of multiplexers. The shifter however is not part of the critical path.

My next thought for optimization is the multiplexers because there are 4 of them in the critical path, meaning that if we can speed them up we could see noticeable gains in performance of our critical path. However the multiplexer is quite a simple component with few gates, so it will likely be hard to squeeze more speed out of them.

Finally I might consider optimizing would be the ALU control, this is because this component is used for all instruction types meaning if we can speed it up we should see gains across the board. This component takes about 3ns/cycle, I'm positive that we could speed this up by simplifying the logic to reduce the total number of gates and therefore gate delays that we are seeing across this component.

m. [Feedback] You must complete this section for your lab to be graded. Please complete each column **separately** for each team member; I expect it to take roughly 10 minutes (do not take more than 20 minutes).

i. How many hours did you spend on this lab?

Task	**During lab time**			**Outside of lab time**		
Team Initials	js	cm	nk	js	cm	
Reading lab	1	1	2	1	1	0
Pencil/paper design	0	0	1	5	0	2
VHDL design	0	0	3	16	10	15
Assembly coding	1	2	0	8	6	0
Simulation	0	0	0	2	2	4
Debugging	4	3	0	24	10	30
Report writing	0	0	0	2	1	1.5
Other:	0	0		0	0	
Total	6	6	6	58	30	53

ii. If you could change one thing about the lab experience, what would it be? Why?

cm: Being able to work in person so that communication is easier and more efficient.

js: I Think at least as online semesters sit, projb should be the final project, this way we don't have to rush anything. we would get an extra week or two. or even a week with no lab similar to what we might get during a break week, to get caught up, that we obviously don't have the luxury of this semester. I also have fears of dead week being very painful with not much time to actually study which is the main goal of dead week. Just because something is technically compliant

with the dead week policies does not mean that it achieves the goal. In the universities words “The intent of this policy (prep week) is to establish a one-week period of **substantial** and **predictable** study time for undergraduate students.” I just don't see how two more weeks of debugging vhd is going to help me succeed on the exam or for that matter have a better understanding of the MIPS ISA. On top of all of that we are expected to stick around until thanksgiving day to try to scramble for as many points as possible. I would like to be home and spend time with my family on Thanksgiving.
In my opinion the documentation for this lab needs a bit of work. The toolflow manual really only gives the bare minimum, once you run through the 4 steps and nothing works you just have to figure the rest out on your own. I found that the FAQ section only had a couple useful FAQ’s. there should definitely be an FAQ for leaf names, that took me a couple weeks to discover.

NK: The biggest annoyance for me was realizing that the test bench used the down to notation. Me and several other students remember being told at the beginning of the semester that either method was fine and that caused a lot more work for us. **Tell students they have to use downto for arrays from the start.** Other then that, supplying useful modelsim tips to help debugging would be very nice. Possibly have example debug sessions for class spaced throught the semester. Some thoughts would be to tell students about the view leaf names early on and give ways to navigate to a specific cycle. I had several problems where an error would only occur on cycle 500 something. Being able to jump to that point immediately would be nice. Adding a pc counter comparison to the test bench would also be helpful for debugging jumps and branches.

iii. What was the most interesting part of the lab?

cm: Getting Bubblesort to work.

js: agrees that it is cool to see a real program run and do something meaningful.

Nk: seeing something built from scratch actually execute code.