**CprE 381 – Computer Organization and**

**Assembly-Level Programming**

**Proj-C Report**

Lab Partners Curt Lengemann

David Quan

Section / Lab Time Section 1 (8:00-10am Thursday)

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

1. [Part 0] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

IF/ID

* + register Pc+4 value
  + 32bit instruction ID/EX
  + values from register rs+rt
  + zero/sign extended immediate
  + Value of address of rt rd
  + RegWriteE, MemtoReg, AluControl, logic arithmatics, Left\_right, alushifter, dmemWE, AluSrc, RegDst, LoadUpperImmidiete, VarShift

EX/Mem

* + AluOutput,
  + Value of register Rt
  + Address of rd rt
  + RegWriteE, MemToReg, dmemWE, RegDst

Mem/WB

* + RegDst, RegWE, MemtoReg
  + Address of rt rd
  + ALUOut
  + Dmem Out

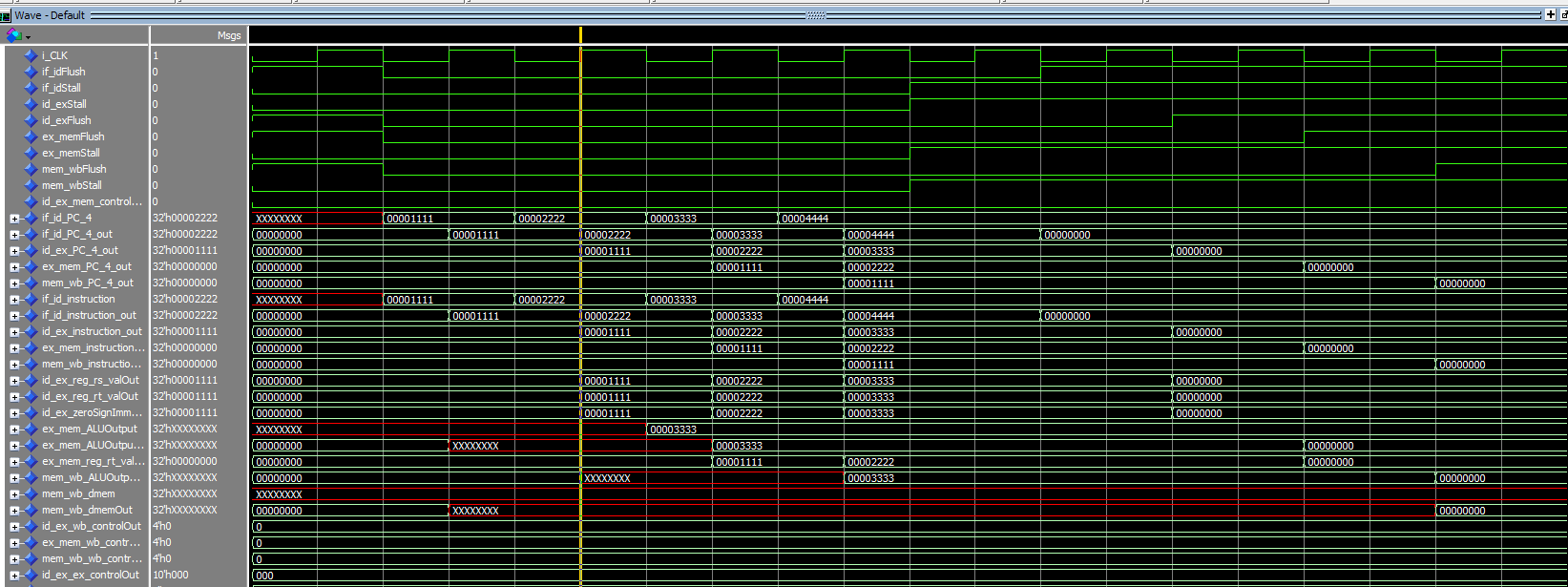
1. [Part 1 (a)] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

Graphical user interface, application

Description automatically generated

Flushing will reset the register. The stall is negated so it is active high. If stall is 1 then write enable will be 0. This means the register will be stalled because nothing can write to it.

1. [Part 1 (b)] 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. *[Please include waveforms and explanations.]*



You can see values like if\_id\_PC\_4 are inserted in on the second cycle. It’s then rippled through the id\_ex\_PC\_4\_out => ex\_mem\_PC\_4\_out => mem\_wb\_PC\_4\_out. This can be seen in our instructions as well. If\_id\_instruction\_out -> id\_ex => ex\_mem all ripple the same same values. This is evidence that our pipeline is pipelining correctly. It also fetches different values every cycle. You can see this in if\_id\_pc\_4 where it fetches 00001111 in 1 clock cycle. This proves our pipeline is working as it 1.ripples values correctly 2. Fetches values in 1 clock cycle

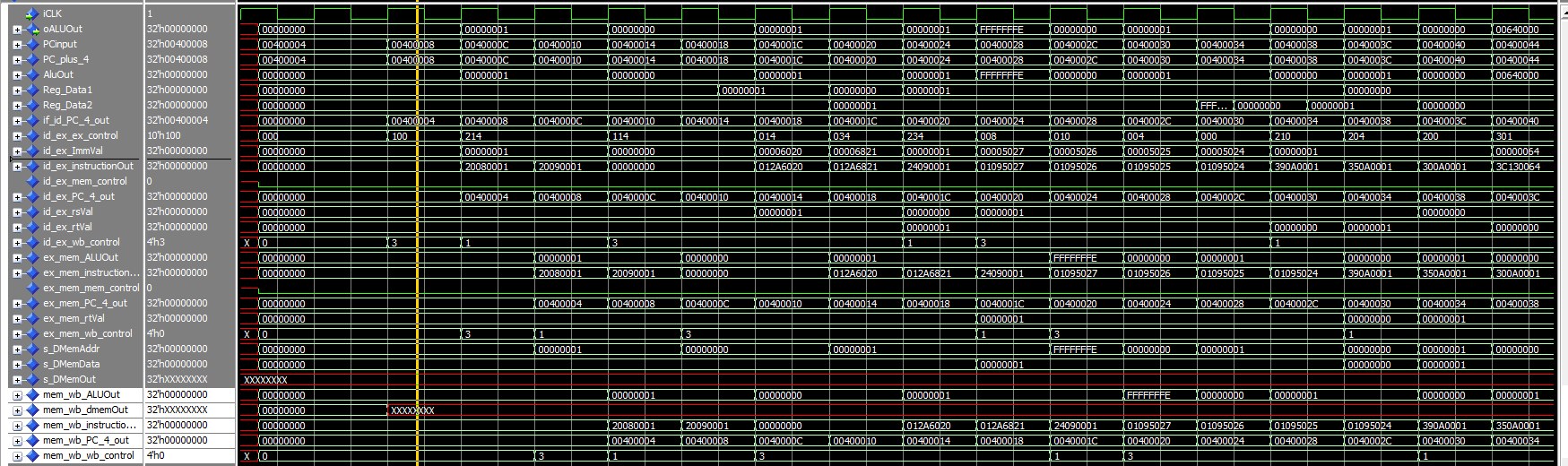
1. [Part 2] In your writeup, provide your schematic for this part, describe what challenges (if any) you faced in implementing this module.

The schematic is located below and in the zip for a better view. We faced a couple challenges in implementing this. First, we struggled to make the processor halt. We learned we had to carry the instruction through all of the registers and compare the instruction to the halt condition during the write-back stage instead of in the decode stage otherwise not all of the instructions would finish executing. Second, we had to modify our register file to be able to write in the first half of the cycle and read in the second half. This required a modified dff for the register file that took input on the falling edge.

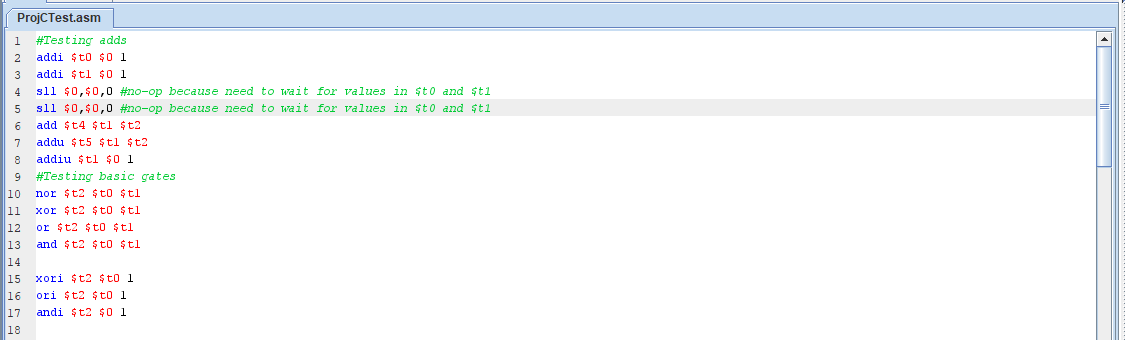
A picture containing holding, sign, standing

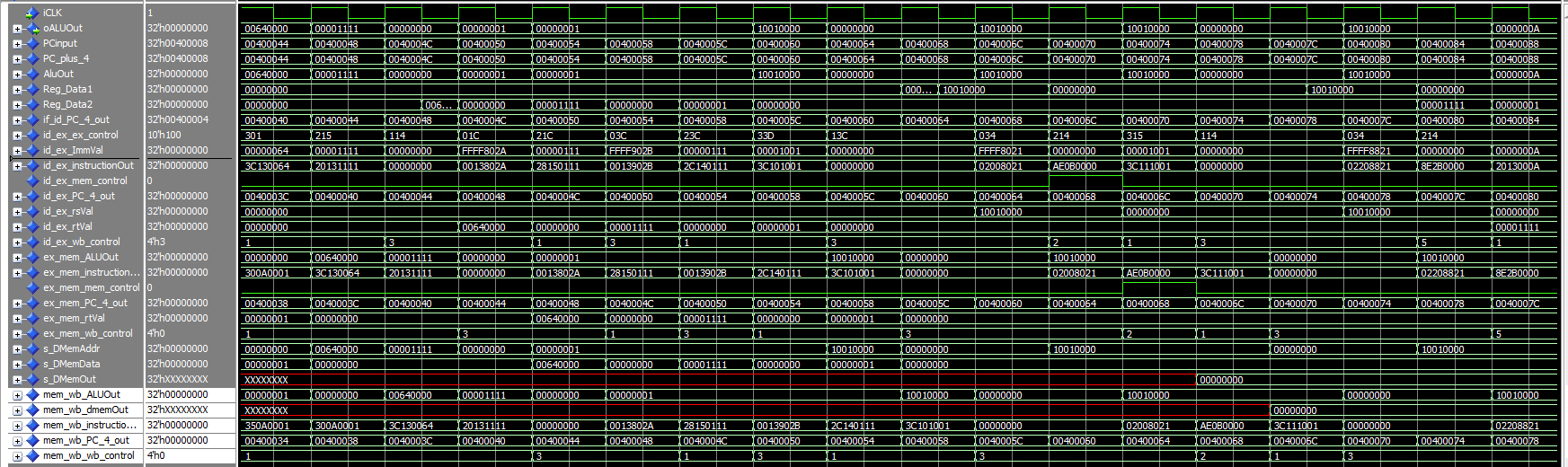
Description automatically generated

1. [Part 3 (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.

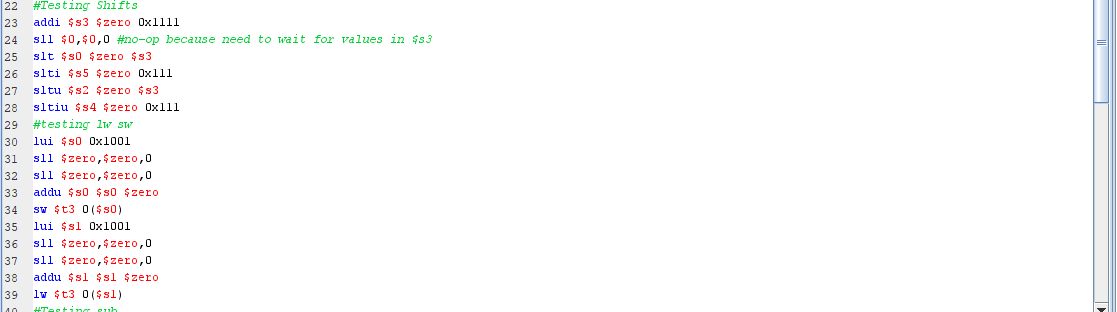


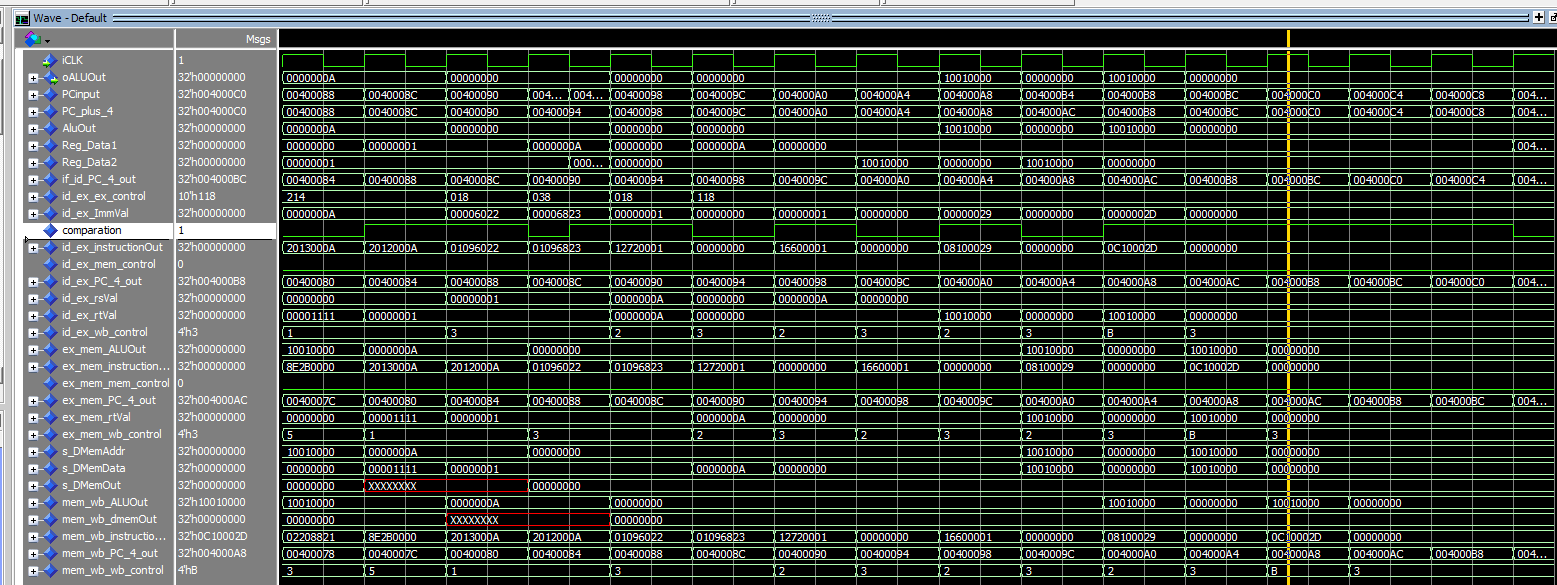
You can tell these are right because we start with 2 addi instructions and aluOut outputs 1 when it’s done with the if/id register then writes to the register which you can see if reg data 1/reg data 2. Then we tests the basic gates and the same pattern shows. Below is the code this waveform corresponds to



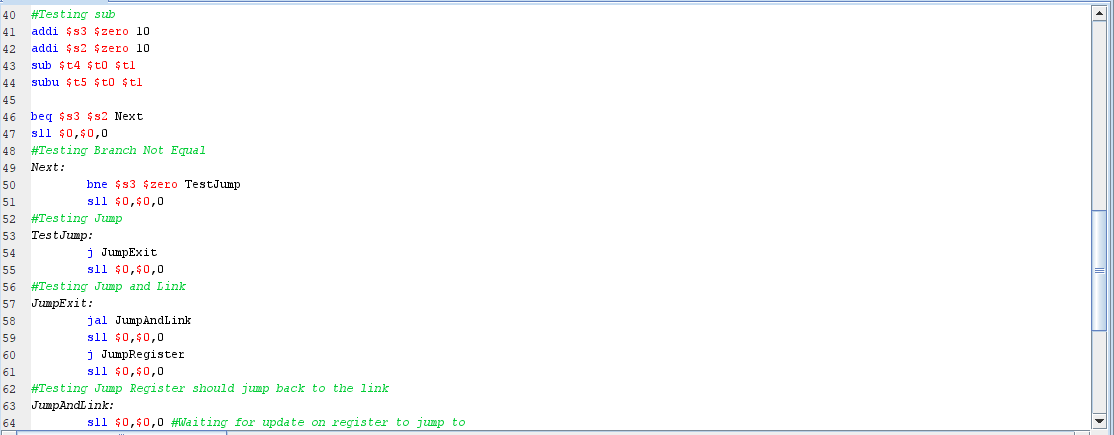


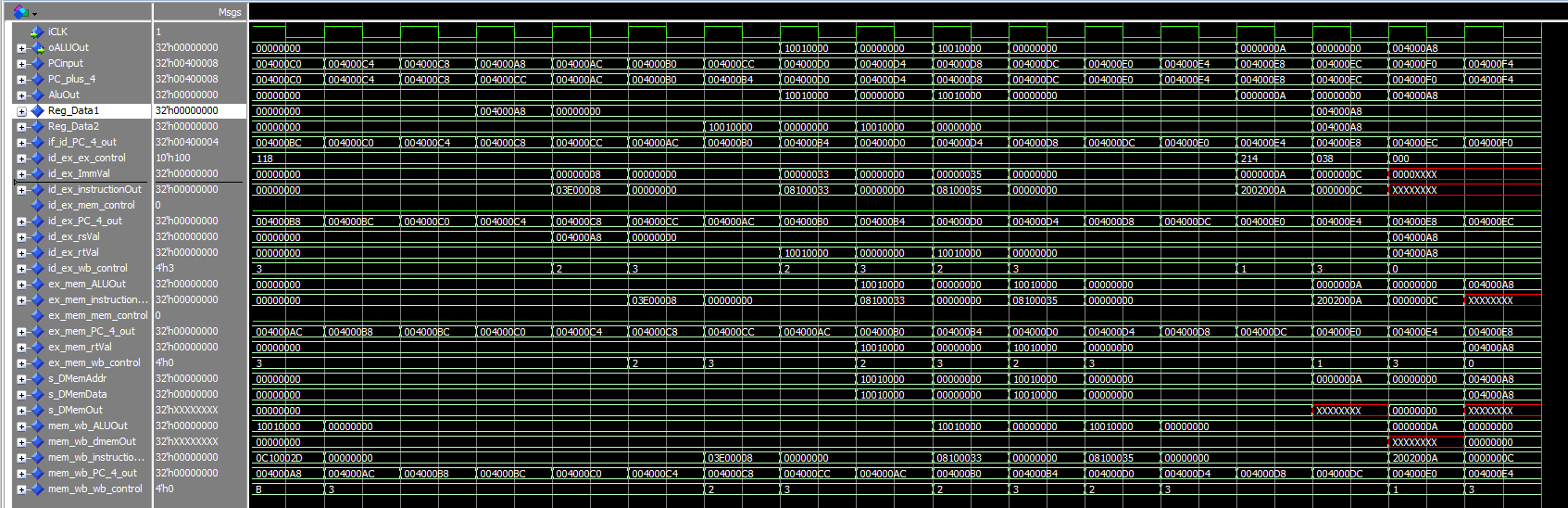
Then we go on to test lui, slt, lw, and sw. You can tell the slt work because it outputs in aluOut and outputs the reg data on the correct cycle (after the 00001111). The sw works because it writes to sdmem on the correct clock cycle. Lw works because it inputs it into the register correctly. The code this waveform tests it below.





Next we test the sub instructions which output 0 correctly because we were subtraction 1 from 1. ALUOut shows it outputs 0 and it writes it to the register. Then it tests the branch instructions. You can tell it works because our comparation (which compares the read data1 and read data2 and sees if they’re equal) jumps to 1 which means our beq is compared correctly. It also jumps to 004000AC to 00400B8 which is +8 instead of the usual +4. You can also see something similar when a jump is called.





This is a continuation of the program above. The last part is to show that jal and jr work. You can see it works because the registers gets the value then later on jumps to that value. You can tell because jal saves the register and jr jumps to the $ra value which is what jal pc\_plus\_4 output was +4.

1. [Part 3 (b)] In your writeup, show the ModelSim output for the modified Bubblesort test, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

1. [Part 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).

1. [Part 5 (a)] Of the MIPS instructions supported for Project Part B, list which instructions produce values, and what signals in the pipeline these correspond to.

1. [Part 5 (b)] List which of these same instructions consume values, and what signals in the pipeline these correspond to.

1. [Part 5 (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.

1. [Part 6] Write a more generalized series of data forwarding and hazard detection logic equations based on the result from part 5).

1. [Part 7] Provide a high-level schematic drawing of the interconnection between components for the MIPS Hardware-scheduled pipelined Processor.

1. [Part 8 (a)] In your writeup, show the ModelSim output for the individual instruction tests and Bubblesort test (the original from Project Part B), and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

1. [Part 8 (b)] In your writeup, show the ModelSim output an application that attempts to exhaustively test the forwarding and detection logic in your pipeline, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

1. [Part 10] 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?

1. [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).

* + 1. How many hours did you spend on this lab?

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Task** | **During lab time** | | | **Outside of lab time** | | |
| **Team Initials** | CL | DQ |  | CL | DQ |  |
| Reading lab |  | 1 |  |  | 1 |  |
| Pencil/paper design |  | 0 |  |  | 3 |  |
| VHDL design |  | 0 |  |  | 3 |  |
| Assembly coding |  | 3 |  |  | 2 |  |
| Simulation |  | 0 |  |  | 3 |  |
| Debugging |  | 0 |  |  | 2 |  |
| Report writing |  | 0 |  |  | 2 |  |
| Other: |  |  |  |  |  |  |
| Total |  | 4 |  |  | 16 |  |

* + 1. If you could change one thing about the lab experience, what would it be? Why?

I think the designs took a very long time to construct. Especially because our designs had multiple iterations. Also, you should tell students to keep lots of space between components from Part B so that we can easily input the pipeline without redoing our Part B design.

* + 1. What was the most interesting part of the lab?

The most interesting part of the lab was learning how to write a software scheduled MIPS program. It is hard to realize how many times that the programmer writes code that depends on each other. It was an interesting and difficult challenge to try to keep the no-ops to a minimum.