CprE 381 – Computer Organization and Assembly-Level Programming

Proj-B Report

Lab Partners	Colby McKinley
	Jordan Silvers
	Tyler Dawson
Section / Lab Time	Section 5 / Wednesday 6:10p – 8:00p

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?

   This can be found in /prelab.pdf

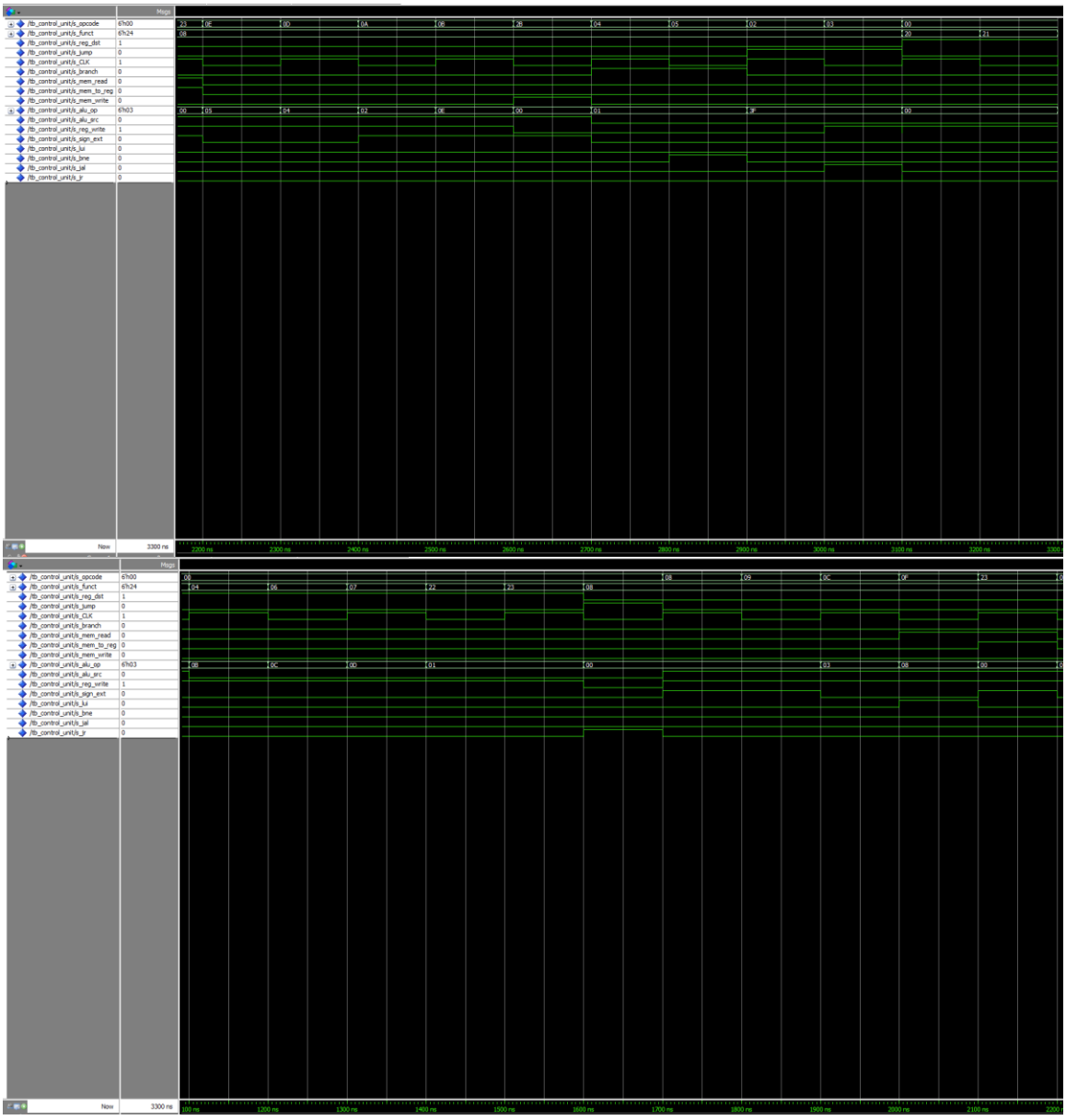
b. [Part 0] In the MIPS skeleton VHDL, how is a halting / termination condition detected? What MIPS assembly instruction does this correspond to?

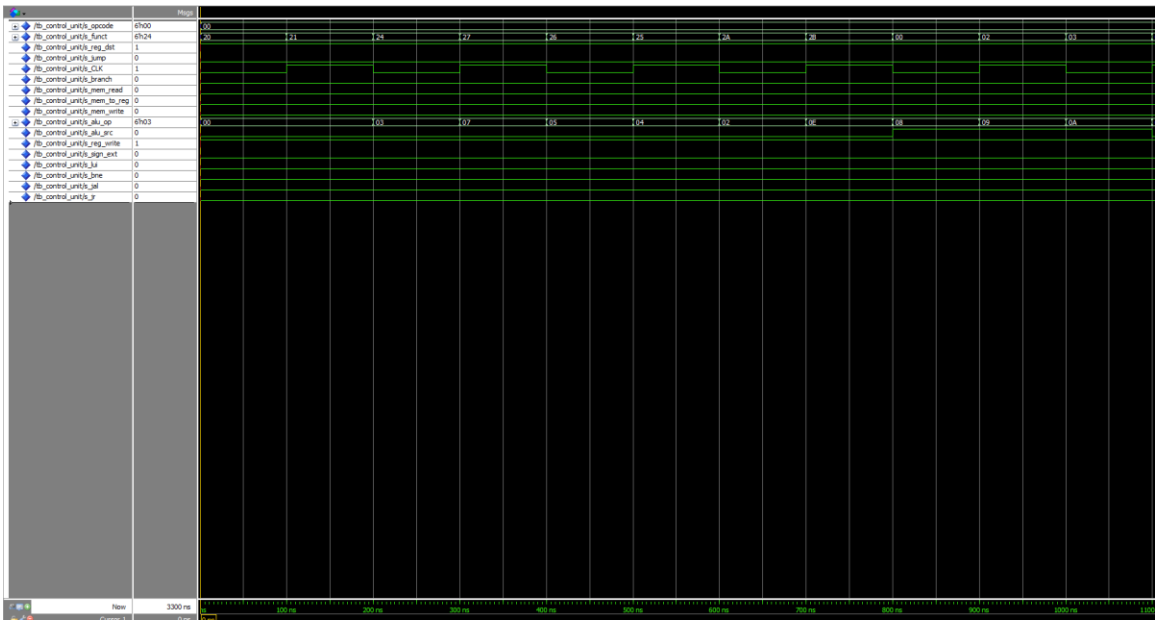
   This can be found in /prelab.pdf

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.]*

JR	JAL	BNE	LUI	Sign	RGDT	JMP	BR	MRd	MW	MRg	ALU_CTRL	ASRC	RW	Instruction	Opcode	Funct
0	0	0	0	0	1	0	0	0	0	0	ADD_ALU_OP	0	1	ADD	000000	100000
0	0	0	0	0	1	0	0	0	0	0	ADD_ALU_OP	0	1	ADDU	000000	100001
0	0	0	0	0	1	0	0	0	0	0	AND_ALU_OP	0	1	AND	000000	100100
0	0	0	0	0	1	0	0	0	0	0	NOR_ALU_OP	0	1	NOR	000000	100111
0	0	0	0	0	1	0	0	0	0	0	XOR_ALU_OP	0	1	XOR	000000	100110
0	0	0	0	0	1	0	0	0	0	0	OR_ALU_OP	0	1	OR	000000	100101
0	0	0	0	0	1	0	0	0	0	0	SLT_ALU_OP	0	1	SLT	000000	101010
0	0	0	0	0	1	0	0	0	0	0	SLTU_ALU_OP	0	1	SLTU	000000	101011
0	0	0	0	0	1	0	0	0	0	0	SLL_ALU_OP	1	1	SLL	000000	000000
0	0	0	0	0	1	0	0	0	0	0	SRL_ALU_OP	1	1	SRL	000000	100000
0	0	0	0	0	1	0	0	0	0	0	SRA_ALU_OP	1	1	SRA	000000	000011
0	0	0	0	0	1	0	0	0	0	0	SLLV_ALU_OP	0	1	SLLV	000000	000100
0	0	0	0	0	1	0	0	0	0	0	SRLV_ALU_OP	0	1	SRLV	000000	000110
0	0	0	0	0	1	0	0	0	0	0	SRAV_ALU_OP	0	1	SRAV	000000	000111
0	0	0	0	0	1	0	0	0	0	0	SUB_ALU_OP	0	1	SUB	000000	100010
0	0	0	0	0	1	0	0	0	0	0	SUB_ALU_OP	0	1	SUBU	000000	100011
1	0	0	0	0	0	1	0	0	0	0	ADD_ALU_OP	0	0	JR	000000	001000
0	0	0	0	1	0	0	0	0	0	0	ADD_ALU_OP	1	1	ADDI	001000	NA
0	0	0	0	1	0	0	0	0	0	0	ADD_ALU_OP	1	1	ADDIU	001001	NA
0	0	0	0	0	0	0	0	0	0	0	AND_ALU_OP	1	1	ANDI	001100	NA
0	0	0	1	0	0	0	0	1	0	0	SLL_ALU_OP	1	1	LUI	001111	NA
0	0	0	0	1	0	0	0	1	0	1	ADD_ALU_OP	1	1	LW	100011	NA
0	0	0	0	0	0	0	0	0	0	0	XOR_ALU_OP	1	1	XORI	001110	NA
0	0	0	0	0	0	0	0	0	0	0	OR_ALU_OP	1	1	ORI	001101	NA
0	0	0	0	1	0	0	0	0	0	0	SLT_ALU_OP	1	1	SLTI	001010	NA
0	0	0	0	1	0	0	0	0	0	0	SLTU_ALU_OP	1	1	SLTIU	001011	NA
0	0	0	0	1	0	0	0	0	1	0	ADD_ALU_OP	1	0	SW	101011	NA
0	0	0	0	0	0	0	1	0	0	0	SUB_ALU_OP	0	0	BEQ	000100	NA
0	0	1	0	0	0	0	1	0	0	0	SUB_ALU_OP	0	0	BNE	000101	NA
0	0	0	0	0	0	1	0	0	0	0	NO_ALU_OP	0	0	J	000010	NA
0	1	0	0	0	0	1	0	0	0	0	NO_ALU_OP	0	1	JAL	000011	NA

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.]*
The testing module goes through and checks each output signal for every instruction to be the expected output. Below are the shown wave forms of tb_control_unit.vhd.


/tb_control_unit/s_opcode
/tb_control_unit/s_funct
/tb_control_unit/s_reg_dst
/tb_control_unit/s_jump
/tb_control_unit/s_CLK
/tb_control_unit/s_branch
/tb_control_unit/s_mem_read
/tb_control_unit/s_mem_to_reg
/tb_control_unit/s_mem_write
/tb_control_unit/s_alu_op
/tb_control_unit/s_alu_src
/tb_control_unit/s_reg_write
/tb_control_unit/s_sign_ext
/tb_control_unit/s_lui
/tb_control_unit/s_bne
/tb_control_unit/s_jal
/tb_control_unit/s_jr
Now
3300 ns



This module was just a pain and was the source of most of the bugs. The don't cares did not compile because of multiplexers elsewhere, so all the signals were driven high or low. If given more time this would be a good module to spend time to get the don't cares to work.

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

The updated schematic below should contain the logic.

The challenge I ran into here was not taking the sign extended value from the extender module, where it sometimes picked zero extend. To correct this issue I changed to use the result of the extender to the result of sign extender. An addition implementation I made was with beq and bne, which negated the alu_o_zero signal depending on if the instruction was beq, bne as the should_branch signal to select between PC + 4 or the computed branch address. The JAL and JR instructions added multiplexers to the control flow, choosing between a larger amount of potential jump values, and storing PC + 4.

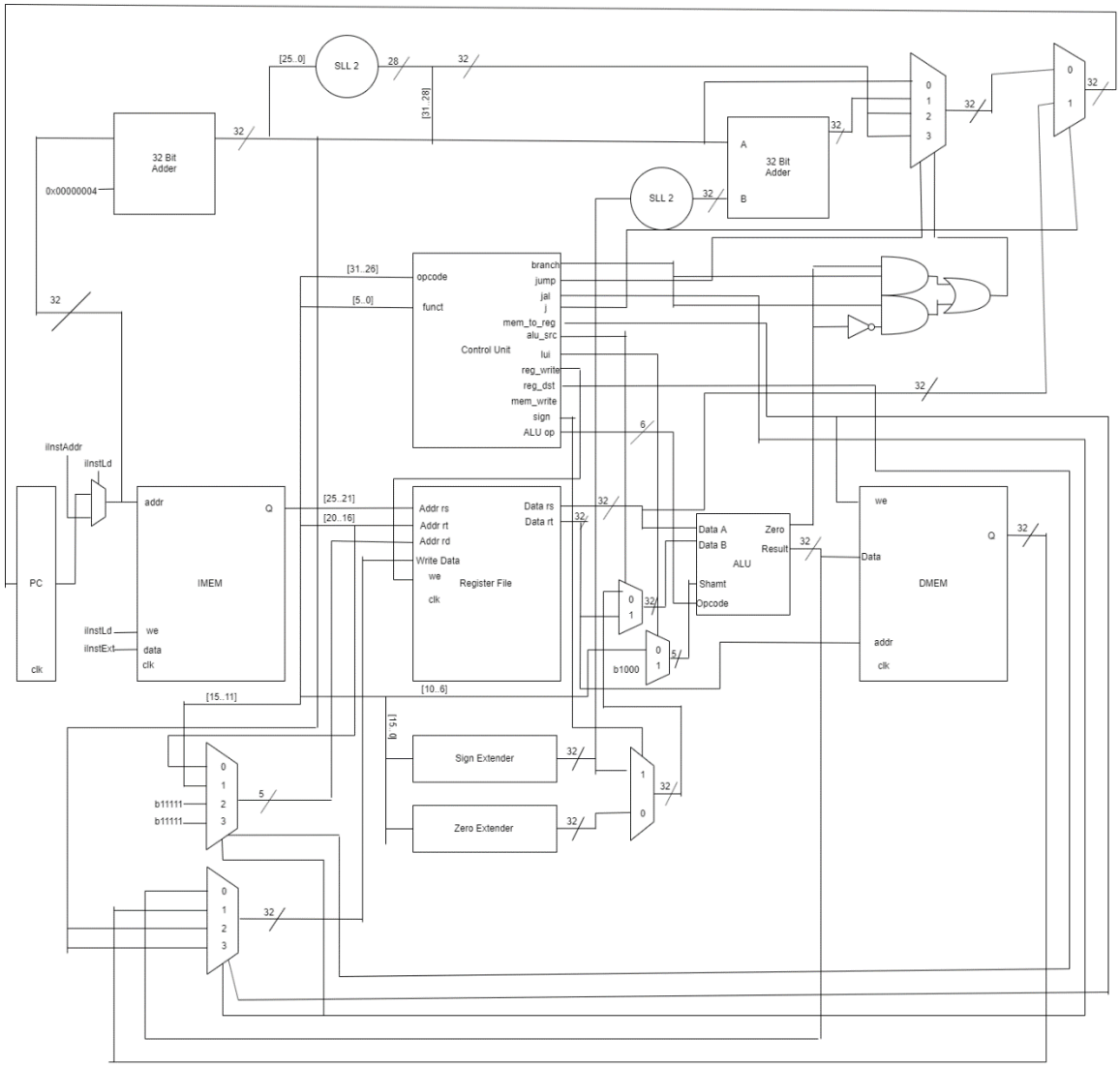
f. [Part 4 (a)] Describe these possibilities as a function of the different control flow-related instructions.

Normally, PC will increment by 4, PC = PC + 4. However, when a branch is to be taken, PC becomes PC = PC + 4 + BranchAddr. Under jump and jump and link PC transforms to PC = JumpAddr. In jr PC becomes PC = R[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.

The additional control signals were branch, jump, jr, jal, beq, bne. The instructions j, jr, jal, beq, bne were single bit signals indication that those instructions were currently being

executed. The signals branch and jump were signal indicating if the program should branch or should jump.



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

The wave forms can be found in /report.
Test1.wlf contains all the induvial tests we wrote, test2.wlf contains more tests.
The control flow possibilities happen through the bits jal, branch, jump and jr. The appropriate combination of these signals will select the correct computed result. This selection happens in the upper left corner of the diagram shown above.

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.
, and

```
Please provide the assembly file to run.
Use unix style paths like: MARsWork/Examples/addiSeq.asm
>MARsWork/Tests/N_queens.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 . . .
```

The above image says the MIPS processor is correct according to MARS, which Dr. Zambreno says is correct. By the transitive property my MIPS processor is correct according to Dr. Zambreno.

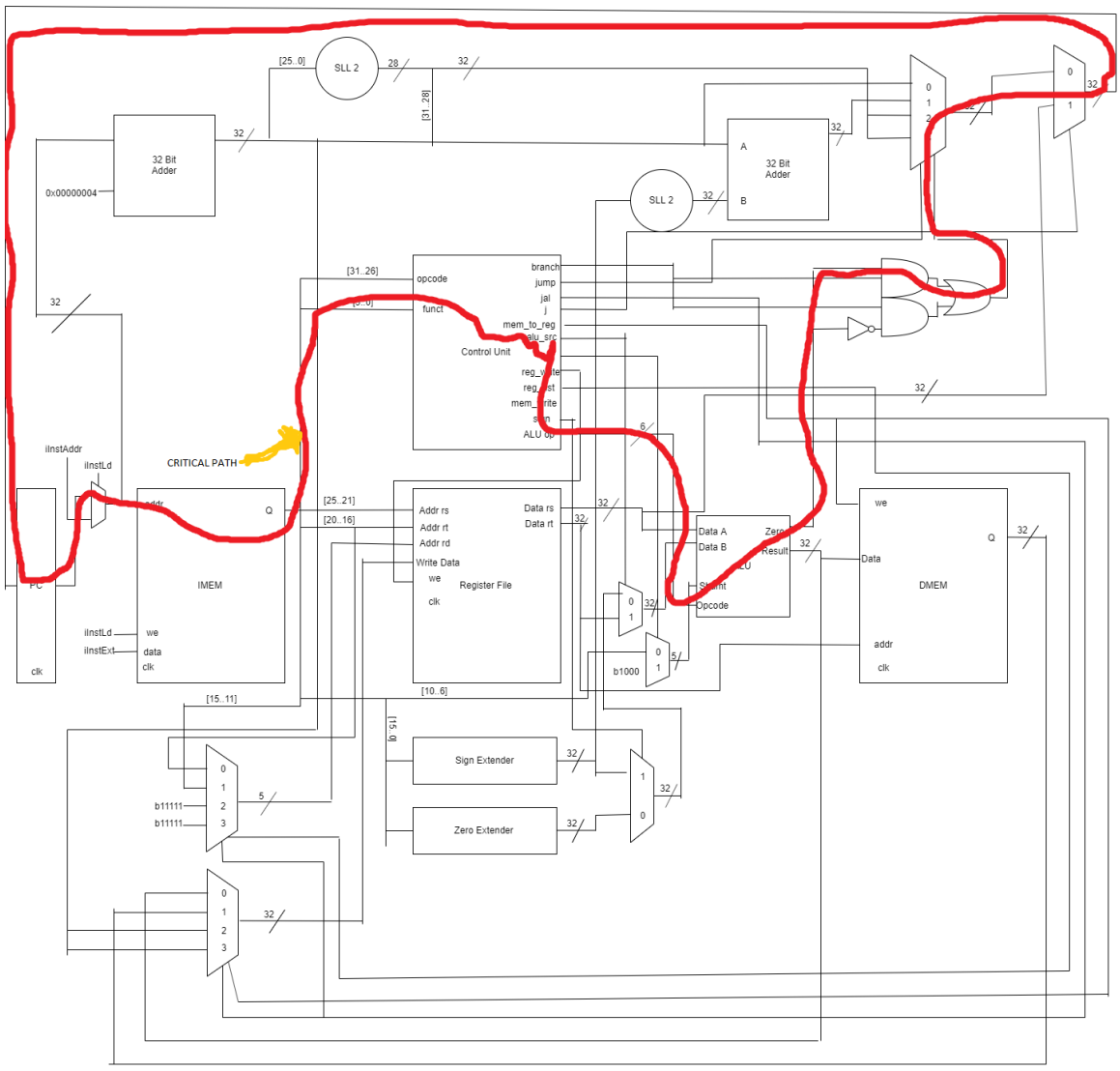
j. [Part 5 (b)] In your writeup, show the ModelSim output for the Bubblesort test, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

   In /report wave forms can be found for bubble sort in test3.wlf

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.
   In /report wave forms can be found for merge sort in test4.wlf.

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?

   The maximum frequency our processor can run at 27.00mhz. The critical path is PC → IMem → Control Unit → ALU (adder)→ Branch And → MUX → PC. The component I would improve is the ALU, more specifically I would choose to compute the 32 bit vectors in a more efficient way through either a carry look ahead adder or a carry select adder.



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	CM	JS	TD	CM	JS	TD
Reading lab	20 min	20 mins	20 min	15 min	15 min	15 min
Pencil/paper design	0	0	0	4 hr	0	0
VHDL design	6 hr	0	1 hr	20 hr	0	0
Assembly coding	0	3 hr	3 hr	0	12 hr	15 hr
Simulation	0	40 min	0	15 hr	3 hr	0
Debugging	1 hr 40	4 hr	4 hr	20 hr	0	0
Report writing	0	0	0	2 hr	4 hr	0

Other:						
Total	8 hr	8hr	8hr	61 hr	19 hr	15 hr

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

There was quite a few headaches wrapped around things I had no control over, such as MARs initializes PC at 0x00400000 and its JAL stores PC=PC + 4 not PC=PC + 8 described on the green sheet.

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

Until N_queens.s worked, I was very sure MIPS processors were an idea we wrote on paper and did not actually exists, like unicorns or something.

In order to get some of the more advance instructions to work, I started extending multiplexers, and it worked. The testing framework is very nice once we got it to work.