CECS 341 – Computer Organization and Design

Lorenzo Murillo IV, 028112355

Matthew Eskridge, 027706989

Professor: Mandy He

California State University, Long Beach

College of Engineering

1250 Bellflower Blvd, Long Beach, CA 90840

Lab Due Date

Design Adder Subtractor

Goal/Objective:

Design a 4-bit binary Adder-Subtractor. We will create two functioning circuits and a testbench to demonstrate their truth tables. After, we will program a simple Adder-Subtractor and create a testbench demonstrating the proper outputs.

Technical Description/Steps:

Part I:

To complete this lab we will be utilizing Xilinx's Vivado software. Over two parts we will recreate two provided digital circuits using behavioral Verilog and design a simple 4-bit adder/subtractor. For part one, we developed two separate behavioral Verilog modules using the provided imagery and truth tables. We then made a testbench to verify each circuit's design and made sure they were constructed correctly.

Part 2:

First we upload the Adder that was developed in Lab 1, this is done by adding the FA.v file as a source. We will then create a new file for our Adder Subtractor, here ours is called "add_sub2bit", it should be called "add_sub4bit". Upon the implementation of add_sub2bit we will need to declare a few inputs and outputs and they are as follows:

1) Inputs: A, B, k
2) Outputs: cout, S

Once the file is created we can begin modification. Here we will initialize our 3 registers and ensure they have appropriate sizing. Following the registers, we will create our wires for our outputs. One output will require the declaration of bit size.

Next we initialize our counting variable, and instantiate the module by implementing the unit under test or "uut". Within our uut we will call the appropriate functions and use the previously declared registers and wires.

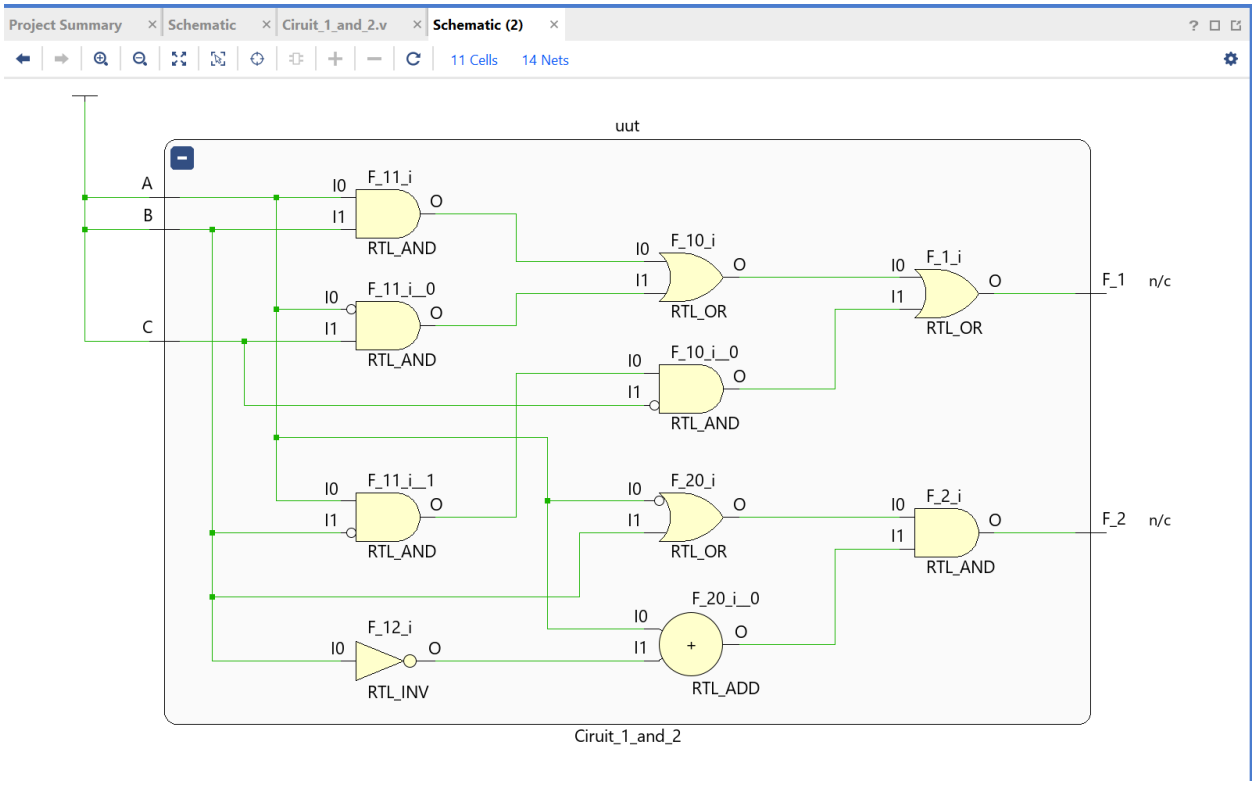
After the uut declaration we are able to use "initial begin". Within this portion we will have our display statements and our for loop to iterate over our adder subtractor and display the appropriate outputs.

Once the Adder Subtractor is complete we can run the waveform test and verify our schematic design.

Results:

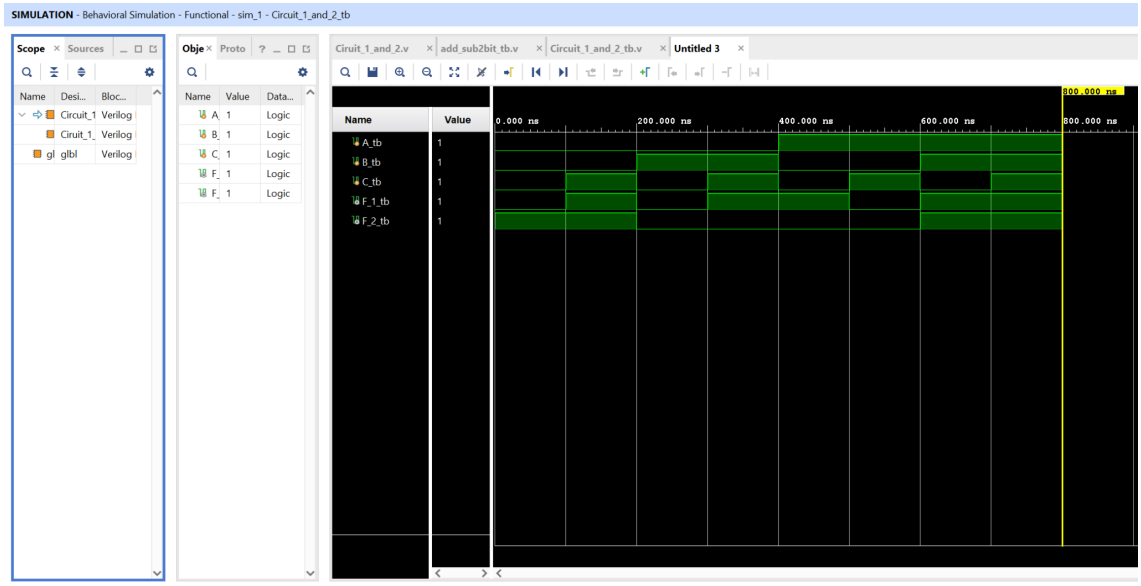
Describe the results you obtained. You can include diagrams, schematics, and images here to aid in your description. This section should be the most descriptive.

Part 1 Schematic:

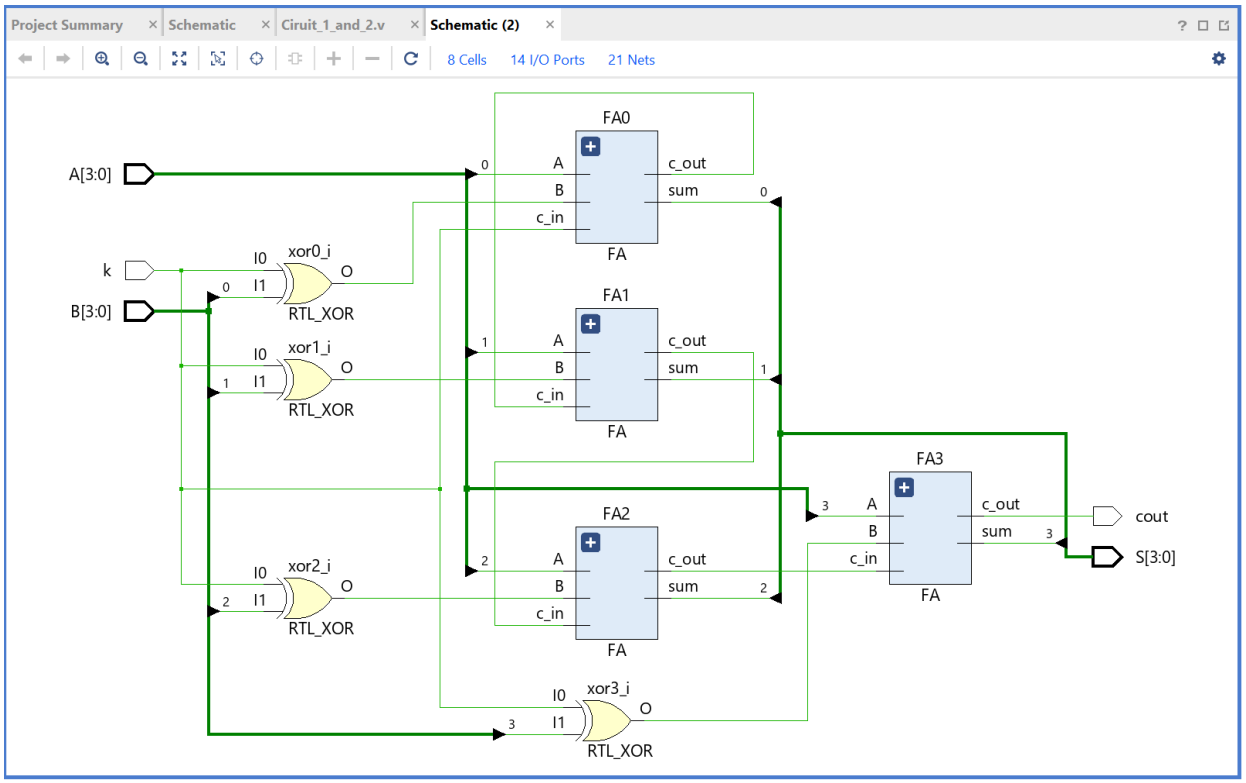


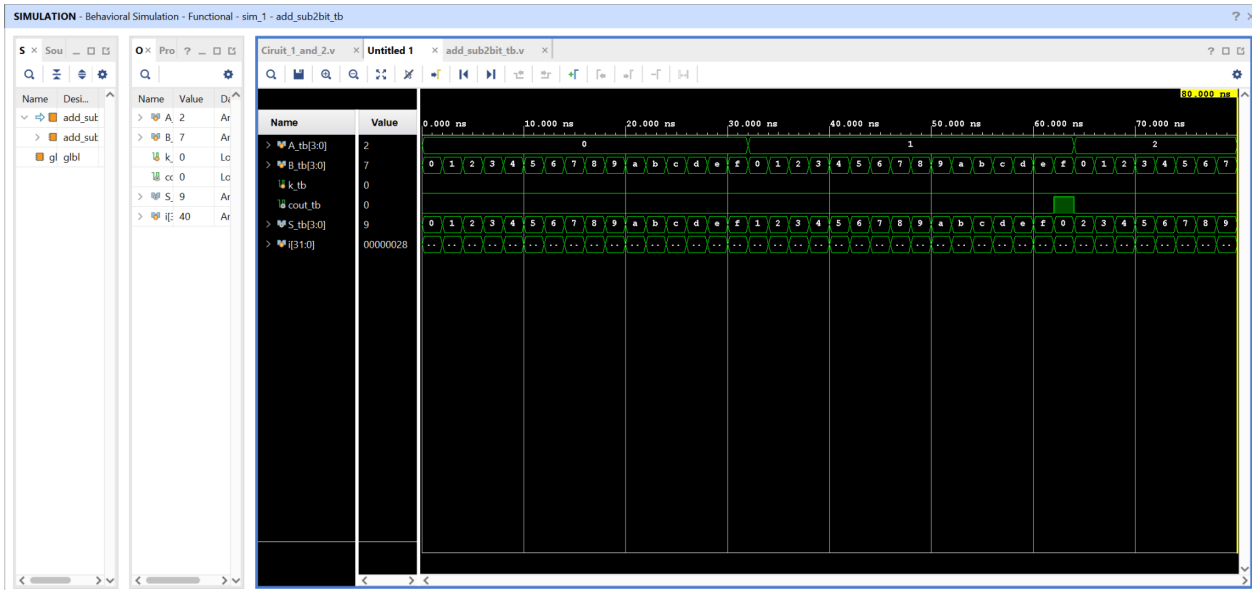
In the schematic above, we can see both of our circuits. This shows the logic gates behind both of our given circuits.

Part 1 Waveform:



Here we show the results obtained by our schematic. The waveform matches our given truth tables.





Conclusion:

What did you learn, and what were the challenges. What could you do differently in hindsight or what changes would you make.

In part one we were able to understand a truth table that was provided to us at a deeper level. We built a program which allowed us to view the gates and waveform. In part one we did both circuits in the same file and this showed us an overall picture of the logic gates. It may have been better to do them separately, but after examining their schematic and ensuring their accuracy it seems that this way was efficient.

In part two we learned how a 4-bit adder/subtractor works and that it requires 4 Full Adders to accomplish. Our test bench was a bit difficult to implement but after some time we were able to view the proper schematic and outputs. After the completion, we can see an iteration happening within the code and is shown on the waveform. This code is presenting us with a series of hexadecimal values and each iteration we add 1 to each provided value, one through fifteen.

R-Type Datapath

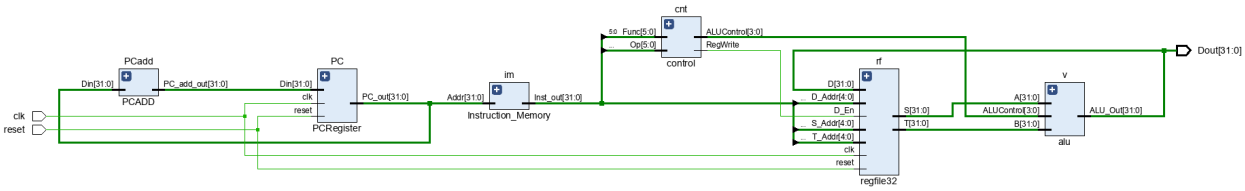
Goal/Objective:

Design a MIPS datapath for R-type instructions using Vivado. To do this, we used resources from previous labs, such as our 32 bit ALU.

Technical Description/Steps:

We designed our Control.v file to take input from instruction memory and outputs from ALU Control. We used our old ALU file and added SLT instructions. With all that, we ended up getting our simulated values for our final hex values.

Schematic



- The schematic shows input clk and reset. These two go through all the different operations(PC, PCADD, IMEM, Control, etc.) and spit out a 32 bit result in the form of Dout.

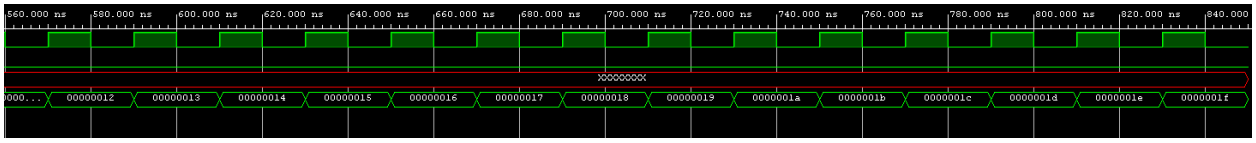
Console Output

```
# run 1000ns
t=t                20000 Dout=00000015
t=t                40000 Dout=00000017
t=t                60000 Dout=0000000c
t=t                80000 Dout=fffffff0
t=t               100000 Dout=0000000f
t=t               120000 Dout=0000001f
t=t               140000 Dout=ffffffff
t=t               160000 Dout=ffffffff
t=t               180000 Dout=00000001
t=t               200000 Dout=00000001
t=t               220000 Dout=xxxxxxxx
```

```
t=230.000000 ns rf[0]:  00000000
t=250.000000 ns rf[1]:  00000000
t=270.000000 ns rf[2]:  00000000
t=290.000000 ns rf[3]:  00000000
t=310.000000 ns rf[4]:  00000000
t=330.000000 ns rf[5]:  00000000
t=350.000000 ns rf[6]:  00000000
t=370.000000 ns rf[7]:  00000000
t=390.000000 ns rf[8]:  00000015
t=410.000000 ns rf[9]:  00000017
t=430.000000 ns rf[10]:  0000000c
t=450.000000 ns rf[11]:  fffffff0
t=470.000000 ns rf[12]:  0000000f
t=490.000000 ns rf[13]:  0000001f
t=510.000000 ns rf[14]:  0000000f
t=530.000000 ns rf[15]:  00000010
t=550.000000 ns rf[16]:  ffffffff
t=570.000000 ns rf[17]:  ffffffff
t=590.000000 ns rf[18]:  00000001
t=610.000000 ns rf[19]:  00000001
t=630.000000 ns rf[20]:  00000015
t=650.000000 ns rf[21]:  00000016
t=670.000000 ns rf[22]:  00000017
t=690.000000 ns rf[23]:  00000018
t=710.000000 ns rf[24]:  00000019
t=730.000000 ns rf[25]:  0000001a
t=750.000000 ns rf[26]:  00000000
t=770.000000 ns rf[27]:  00000000
t=790.000000 ns rf[28]:  00000000
t=810.000000 ns rf[29]:  00000000
t=830.000000 ns rf[30]:  00000000
t=850.000000 ns rf[31]:  00000000
```

The console output just verifies the waveforms in the form of written numbers. These values and our hand calculated values are equivalent.

Waveform:



In the waveforms above, we can see clk, reset, dout, and i. CLK seems to have constant variation between 0 and 1. Reset starts at 1 then becomes off to 0. Dout is our result, which is red because we only ran 11 clock cycles. i represents the loop within the testbench.

Describe the results you obtained. You can include diagrams, schematics, and images here to aid in your description. This section should be the most descriptive.

Output and Calculation table:

No	Register	Calculated		Simulated	
		Initial Val.	Final Val.	Initial Val.	Final Val.
0	$zero	0000 0000	0000 0000	0000 0000	0000 0000
1	$at	0000 0000	0000 0000	0000 0000	0000 0000
2	$v0	0000 0000	0000 0000	0000 0000	0000 0000
3	$v1	0000 0000	0000 0000	0000 0000	0000 0000
4	$a0	0000 0000	0000 0000	0000 0000	0000 0000
5	$a1	0000 0000	0000 0000	0000 0000	0000 0000
6	$a2	0000 0000	0000 0000	0000 0000	0000 0000
7	$a3	0000 0000	0000 0000	0000 0000	0000 0000
8	$t0	0000 0009	0000 0015	0000 0009	0000 0015
9	$t1	0000 000A	0000 0017	0000 000A	0000 0017
10	$t2	0000 000B	0000 000C	0000 000B	0000 000C
11	$t3	0000 000C	FFFF FFF0	0000 000C	FFFF FFF0
12	$t4	0000 000D	0000 000F	0000 000D	0000 000F
13	$t5	0000 000E	0000 001F	0000 000E	0000 001F
14	$t6	0000 000F	0000 000F	0000 000F	0000 000F
15	$t7	0000 0010	0000 0010	0000 0010	0000 0010
16	$s0	0000 0011	FFFF FFFF	0000 0011	FFFF FFFF
17	$s1	0000 0012	FFFF FFFF	0000 0012	FFFF FFFF

18	$s2	0000 0013	0000 0001	0000 0013	0000 0001
19	$s3	0000 0014	0000 0001	0000 0014	0000 0001
20	$s4	0000 0015	0000 0015	0000 0015	0000 0015
21	$s5	0000 0016	0000 0016	0000 0016	0000 0016
22	$s6	0000 0017	0000 0017	0000 0017	0000 0017
23	$s7	0000 0018	0000 0018	0000 0018	0000 0018
24	$t8	0000 0019	0000 0019	0000 0019	0000 0019
25	$t9	0000 001A	0000 001A	0000 001A	0000 001A
26	$k0	0000 0000	0000 0000	0000 0000	0000 0000
27	$k1	0000 0000	0000 0000	0000 0000	0000 0000
28	$gp	0000 0000	0000 0000	0000 0000	0000 0000
29	$sp	0000 0000	0000 0000	0000 0000	0000 0000
30	$fp	0000 0000	0000 0000	0000 0000	0000 0000
31	$ra	0000 0000	0000 0000	0000 0000	0000 0000

Hand Calculated Values

1 add $t0, $t1, $t2

⇒ $t0 = $t1 + $t2 where $t1 = 0000 000A and $t2 = 0000 000B

⇒ $t0 = (0000 000A + 0000 000B)$_{hex}$

⇒ $t0 = $10_{dec.} + 11_{dec.}$

⇒ $t0 = $21_{dec.}$ ⇒ 0001 0101$_{bin.}$ ⇒ 0x15 ⇒ 0000 0015

∴ $t0 has 15_{hex} saved into it

2 addu $t1, $t2, $t3 (add unsigned)

⇒ $t1 = $t2 + $t3 where $t2 = 0000 000B and $t3 = 0000 000C

⇒ $t1 = (0000 000B + 0000 000C)$_{hex}$

⇒ $t1 = $11_{dec.} + 12_{dec.}$

⇒ $t1 = $23_{dec.}$ ⇒ 0001 0111$_{bin.}$ ⇒ 0x17 ⇒ 0000 0017

∴ now $t1 = 0000 0017 $t1 changed from 0000 000A to 0000 0017$_{hex.}$

3 and $t2, $t3, $t4

recall: AND truth table

A	B	
0	0	0
0	1	0
1	0	0
1	1	1

⇒ $t2 = ($t3) • ($t4) where $t3 = 0000 000C and $t4 = 0000 000D

$t2 = $(12_{dec.}) • (13_{dec.})$

$t2 = 0000 1100$_{bin.}$

AND 0000 1101$_{bin.}$

0000 1100$_{bin.}$ ⇒ 0x C ⇒ 0000 000C

∴ $t2 = 0000 000C$_{hex.}$ $t2 changed from 0000 000B to 0000 000C

4 nor $t3, $t4, $t5

recall: NOR truth table

A	B	
0	0	1
0	1	0
1	0	0
1	1	0

⇒ $t3 = $\overline{(\$t4 + \$t5)}$ where $t4 = 0000 000D and $t5 = 0000 000E

⇒ $t3 = $\overline{((13)_{dec.} + (14)_{dec.})}$

⇒ $t3 = 0000 1101$_{bin}$

NOR 0000 1110$_{bin}$

⇒ $t3 = 1111 0000$_{bin}$ ⇒ 0xF0 ⇒ FFFF FFF0

because preceding bytes of $t4 and $t5 are Zero: 0000 ... 0000 1101 NOR 0000 ... 0000 1110

∴ $t3 = FFFF FFF0 $t3 has changed from 0000 000C to 0000 00F0

5 or $t4, $t5, $t6

recall: OR truth table

A	B	
0	0	0
0	1	1
1	0	1
1	1	1

⇒ $t4 = ($t5 + $t6) where $t5 = 0000 000E and $t6 = 0000 000F

⇒ $t4 = $(14_{dec.}$ (OR) $15_{dec.})$

⇒ $t4 = 0000 1110$_{bin}$

OR 0000 1111$_{bin}$

⇒ $t4 = 0000 1111$_{bin}$ ⇒ 0x0F ⇒ 0000 000F

∴ $t4 = 0000 000F $t4 has changed from 0000 000D to 0000 000F

6 XOR $t5, $t6, $t7

recall: XOR truth table

A	B	
0	0	0
0	1	1
1	0	1
1	1	0

⇒ $t5 = $t6 ⊕ $t7 where $t6 = 0000 000F and $t7 = 0000 ... 0001 0000

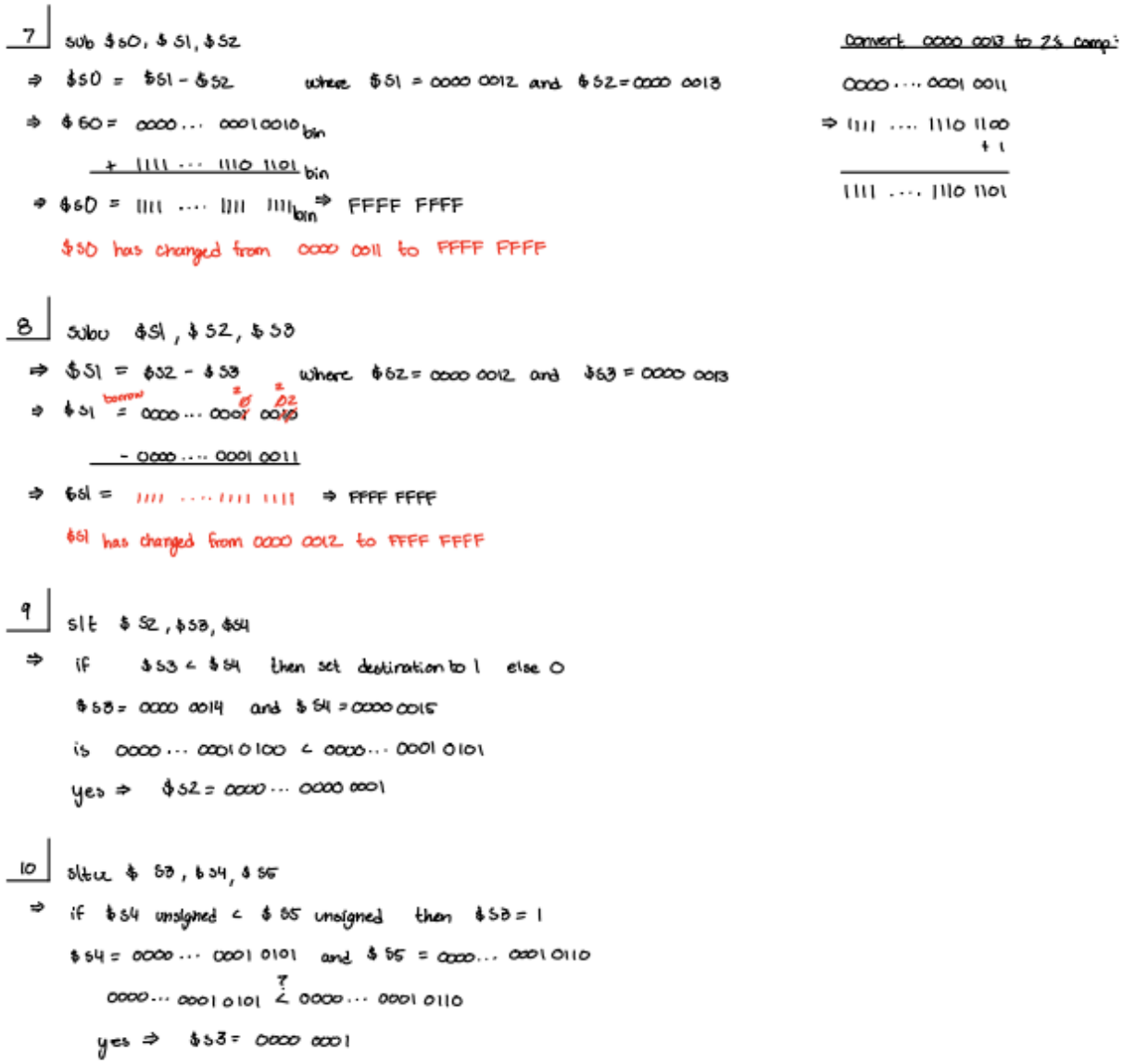
⇒ $t5 = $15_{dec.} ⊕ 16_{dec.}$

⇒ $t5 = 0000 ... 0000 1111$_{bin.}$

XOR 0000 ... 0001 0000$_{bin.}$

⇒ $t5 = 0000 ... 0001 1111$_{bin}$ ⇒ 0x1F ⇒ 0000001F

∴ $t5 = 0000 001F $t5 has changed from 0000 000E to 0000 001F



Hand calculated values for registers in imem.

Conclusion:

This lab was used to help us design a MIPS datapath with 2 inputs, multiple files, and 1 output. The most difficult part of this lab was designing the testbench and making sure that our calculated and simulated values were on the right track.

MIPS Datapath for I-type and R-type Instructions

Goal/Objective:

Modify existing MIPS datapath to allow for I-type instruction processing. The goal of this implementation is for our datapath to allow both R-type and I-type instructions.

Technical Description/Steps:

1) Create a new file and evaluate the given schematic and determine which components must be incorporated in our program to implement our I-type instruction. Upon examination we will notice that we must implement four multiplexers (A, B, C, D), along with a shift left and sign extension.
2) Additionally we will add the given files to our project for reading and writing purposes.
3) After evaluating the schematic we will implement our I-type instruction by using an else statement that uses Op-code from our given instruction. Here we will instantiate different cases for our Op-code to be executed properly. The needed adjustments for I-type instruction are: addi, addiu, andi, ori, lw, sw, beq, bne, slti, sltiu. Each of these operations has a different hexadecimal Opcode and most will use different ALUcontrols.

4) Once we have completed our given cases we must include a default case that outputs "x".

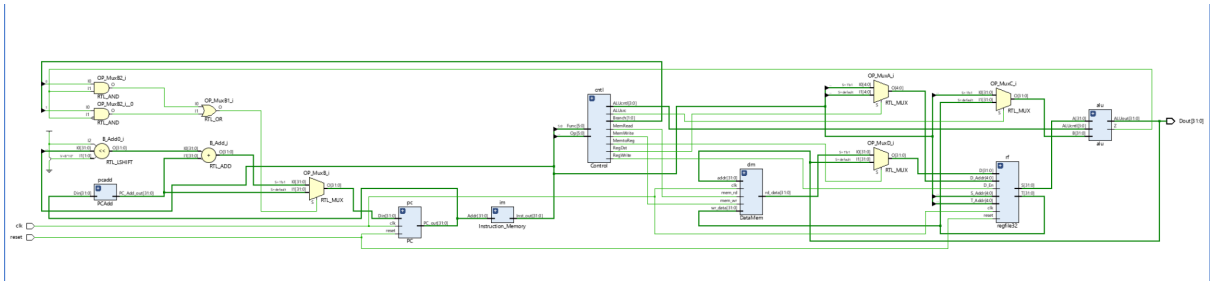
5) Following the competition of the Control.v file, we will now look to create our mux's and the sign extension on our datapath. This is done by modifying the Datapath.v file. We will initialize our functions and pass in values, here we will use: PC, PCAdd, Control, Instruction_Memory, alu, regfile32, and DataMem.

6) With our datapath complete we can now begin the testbench. We must create a task function that iterates over a provided amount and adds four to each iteration. Here we will store our calculated values by "dumping" or storing the calculated values into registers while displaying the time and storage location in the register file.

7) Finally, we can access if the given output matches our hand calculations. Once complete we can then examine the schematic and waveform.

Results:

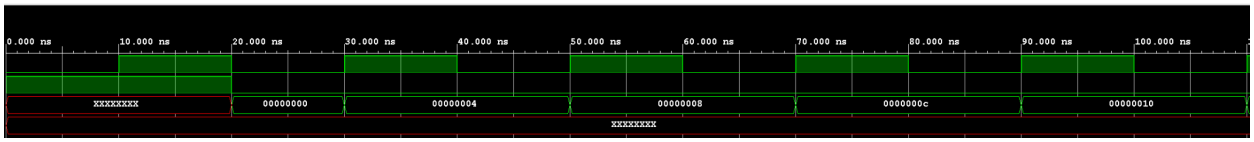
Schematic:

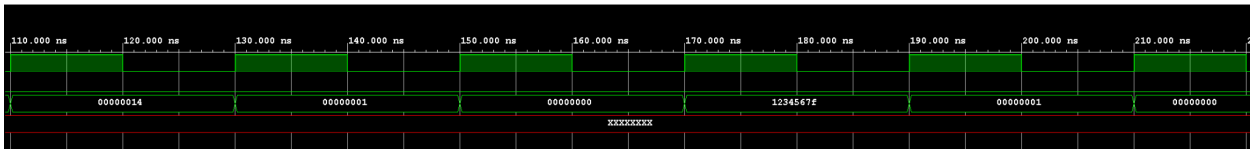


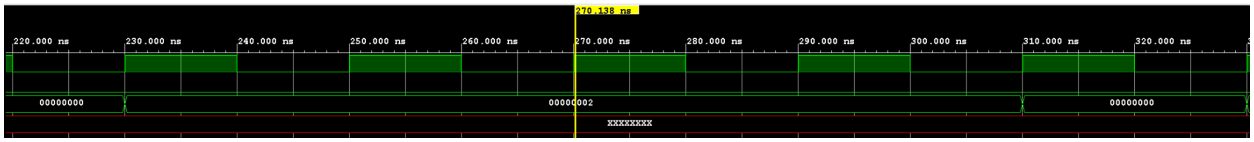
The schematic of our complete datapath shows that our control has 8 outputs as intended.

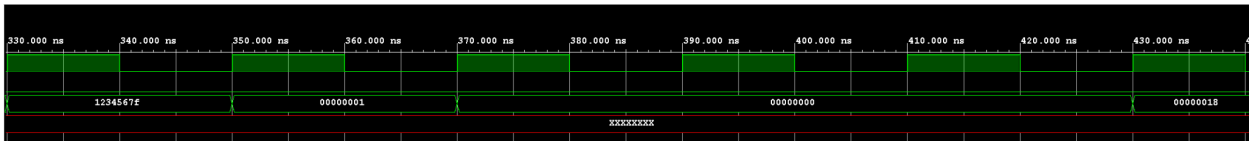
Waveform:

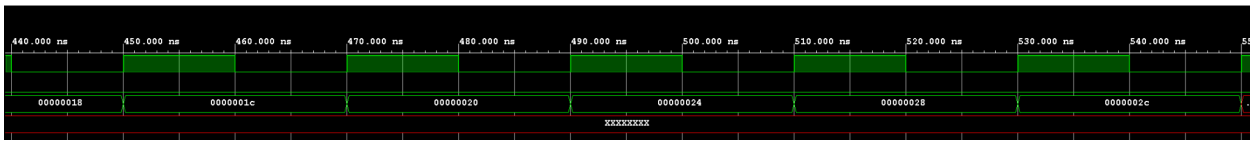
Waveform that shows all clock cycles and inputs. The waveform ranges from 0.00ns - 730.00ns

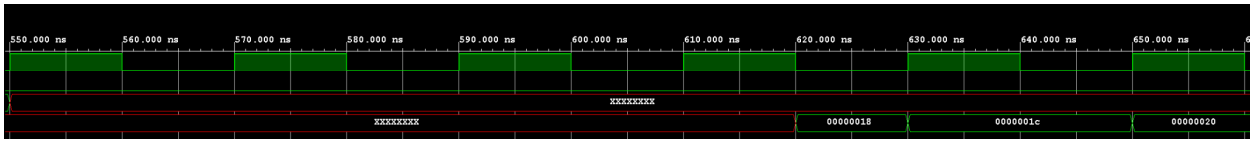


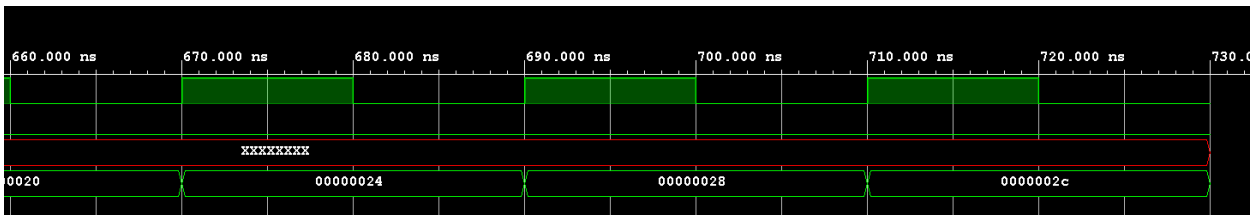












Given Test Cases and Written Work:

Table 2 : Initial Contents of Data Memory

ML	IV	ML	IV	ML	IV	ML	IV	ML	IV	ML	IV
0	00	4	FF	8	12	12	00	16	00	20	00
1	00	5	FF	9	34	13	00	17	00	21	00
2	00	6	FF	10	56	14	00	18	00	22	00
3	00	7	FF	11	78	15	08	19	01	23	03

instruction :

lw $t0, 0($zero) ⇒ t0: 0000 0000
lw $t1, 4($zero) ⇒ t1: FFFF FFFF
lw $t2, 8($zero) ⇒ t2: 1234 5678
lw $t3, 12($zero) ⇒ t3: 0000 0008
lw $t4, 16($zero) ⇒ t4: 0000 0001
lw $t5, 20($zero) ⇒ t5: 0000 0003

addi $t0, $t0, 1 ⇒ t0 : 0000 0001
andi $t1, $t1, 0 ⇒ t1 : 0000 0000
ori $t2, $t2, 7 ⇒ t2 : 0001 0010 0011 0100 0101 0110 0111 1000
0000 0000 0000 0000 0000 0000 0000 0111
0001 0010 0011 0100 0101 0110 0111 1111 ⇒ 1234 567F
slti $t3, $t3, 9 ⇒ t3: 0000 0001
sltiu $t4, $t4, 0 ⇒ t4 : 0000 0000
sub $t5, $t5, $t0 ⇒ t5: 0000 0001
beq $t5, $zero, store ⇒ do not take
bne $t5, $zero, inst ⇒ take branch
⇒ addi $t0, $t0, 1 ⇒ t0: 0000 0010
sw $t0, 24($zero) ⇒ t0: 0000 0002
sw $t1, 28($zero) ⇒ t1 : 0000 0000
sw $t2, 32($zero) ⇒ t2: 1234 567F
sw $t3, 36($zero) ⇒ t3: 0000 0001
sw $t4, 40($zero) t4: 0000 0000
sw $t5, 44($zero) t5: 0000 0000

result :

0	00	4	FF	8	12	12	00	16	00	20	00	24	00	28	00	32	12	36	00	40	00	44	00
1	00	5	FF	9	34	13	00	17	00	21	00	25	00	29	00	33	34	37	00	41	00	45	00
2	00	6	FF	10	56	14	00	18	00	22	00	26	00	30	00	34	56	38	00	42	00	46	00
3	00	7	FF	11	78	15	08	19	01	23	03	27	02	31	00	35	7F	39	01	43	00	47	00

Vivado Output:

```
t=  630.0ns rf[24] 00000002
t=  650.0ns rf[28] 00000000
t=  670.0ns rf[32] 1234567f
t=  690.0ns rf[36] 00000001
t=  710.0ns rf[40] 00000000
t=  730.0ns rf[44] 00000000
$finish called at time : 730 ns : File "C:/Users/028112355/Downloads/Datapath_tb.v" Line 43
```

Conclusion:

This lab allowed us to elaborate on our simple MIPS architecture design. By expanding the instruction type and introducing I-type we have allowed the design to accept more complex instructions. Combining these two types helps us understand the MIPS architecture and see the more complicated design come to fruition. J-type is now the only missing instruction type. This lab was extremely helpful in many ways and further strengthened our knowledge around the MIPS architecture and how data is computed and distributed.

Datapath for I-type, R-type, and J-type Instructions

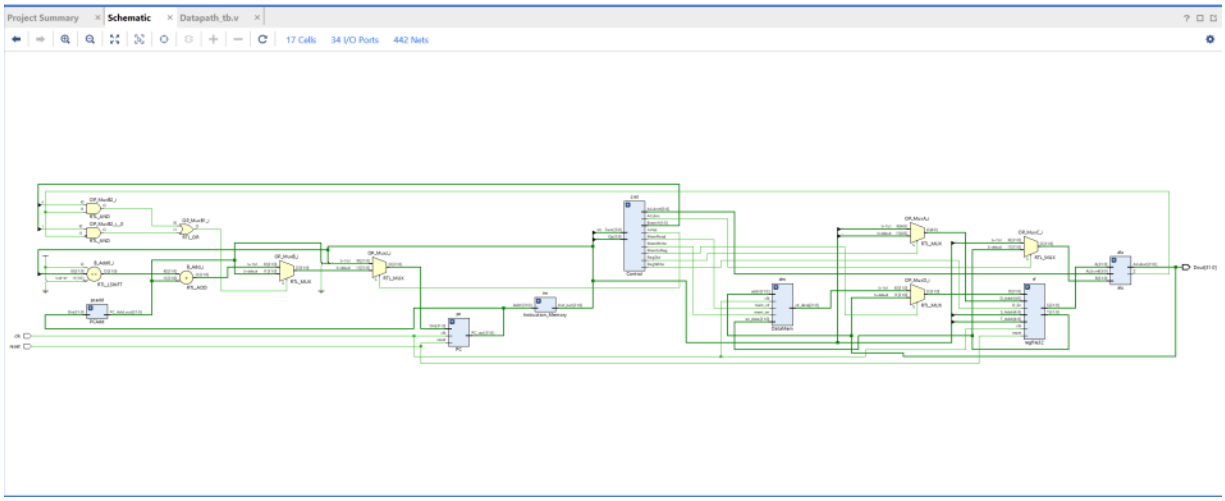
Goal/Objective:

Goal of this lab was to add jump instructions to our MIPS datapath from previous labs. We also created 2 .dat files, one for instruction memory and one for data memory.

Technical Description/Steps:

In this lab, we are modifying our existing datapath for R type and I type instructions to create a J type instruction. To do this, we must use the datapath from Lab 5. Then, we modify it by adding the new component needed for the j type instruction..

Results:

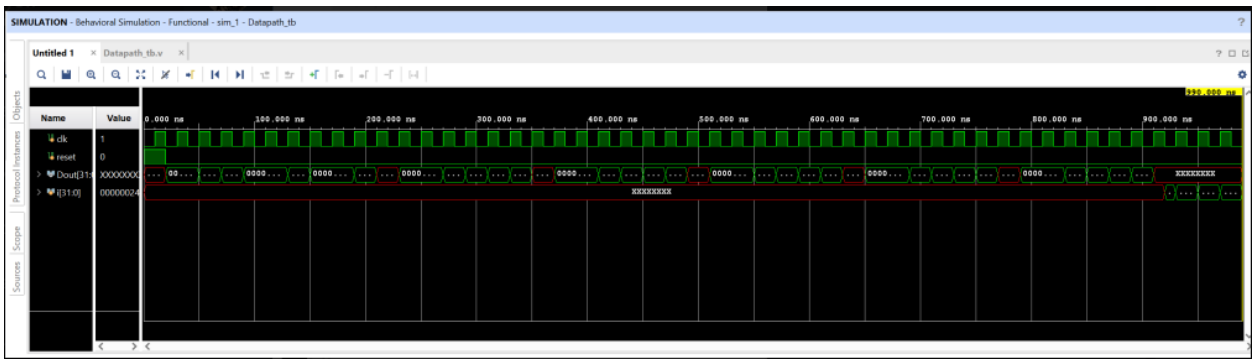


In this schematic, you can see that we were able to add a jump instruction with the MUX. Our Mux for jump added a wire that routes to the PC based on one of two inputs from either shift left 2 or BPC adder. Inputs are both 32bit wires and controlled by a jump control signal from the module.

```
Tcl Console × Messages Log

source Datapath_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If y
#   }
# }
# run 1000ns
t=  930.0ns dm[20] 00000005
t=  950.0ns dm[24] 0000000f
t=  970.0ns dm[28] 00000005
t=  990.0ns dm[32] 00000014
$finish called at time : 990 ns : File "C:/Users/028112355/Lab_5/Lab_5.srcs/Datapath_tb.v" Line 57
INFO: [USF-XSim-96] XSim completed. Design snapshot 'Datapath_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:11 ; elapsed = 00:00:21 . Memory (MB): peak = 1319.707 ; gain = 36.641
```

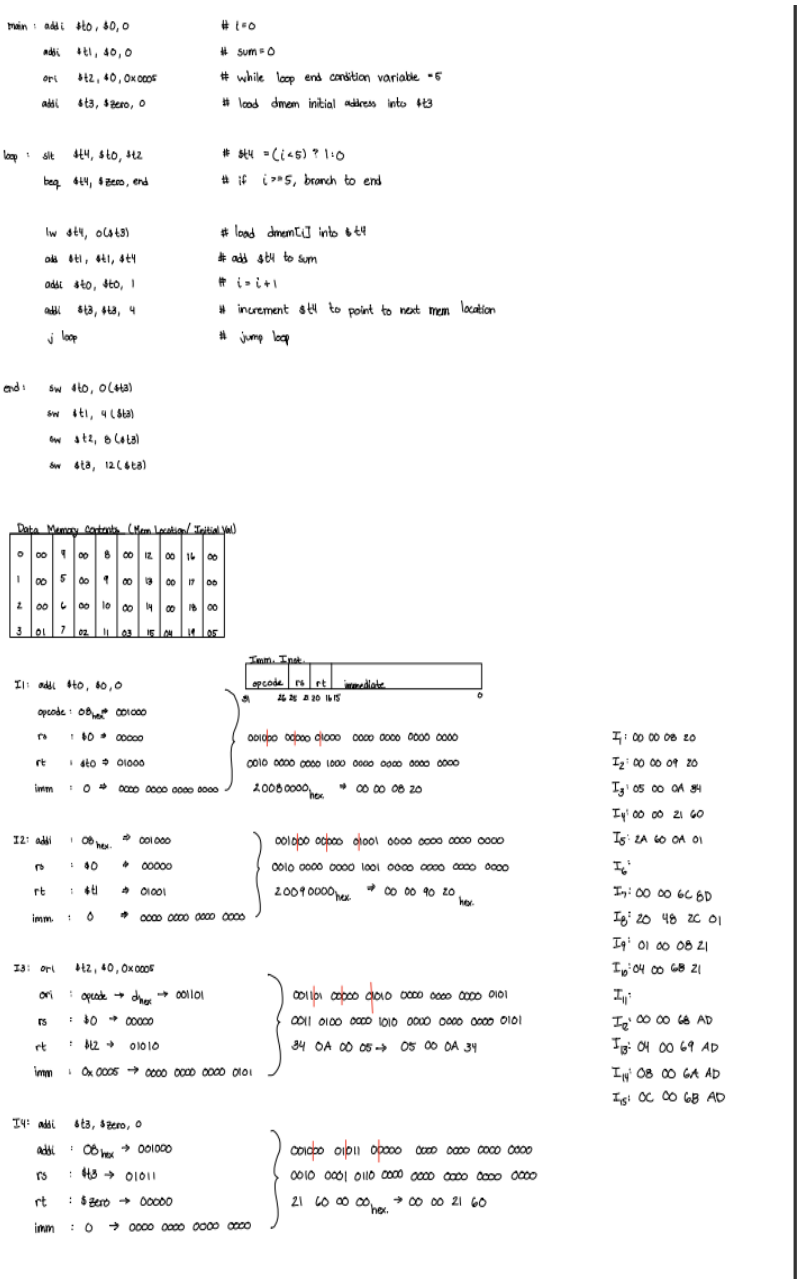
Our 4 results are from 4 different memory locations, which are 20, 24, 28, and 32 respectfully. The values shown are the combined initial values from the memory locations.



Our waveform confirms the result of the console output that is displayed above.

```
00 00 08 20
00 00 09 20
05 00 0a 34
00 00 0b 20
2a 60 0a 01
05 00 80 11
00 00 6c 8d
20 48 2c 01
01 00 08 21
04 00 6b 21
04 00 00 08
00 00 68 ad
04 00 69 ad
08 00 6a ad
0c 00 6b ad
```

Imem.dat - We used the MIPs reference card to interpret what each instruction meant from the instruction file provided. For this we needed to pay attention to which type of instruction was given and carry out the appropriate calculations.



Conclusion:

In this lab, we learned how to implement the jump instruction into our existing datapath. We also created two new .dat files, one for instruction memory and one for data memory. The hardest part of this new lab was definitely the .dat files, having to write them in big endian (data memory) and little endian (instruction memory).