CprE 381: Computer Organization and Assembly-Level Programming

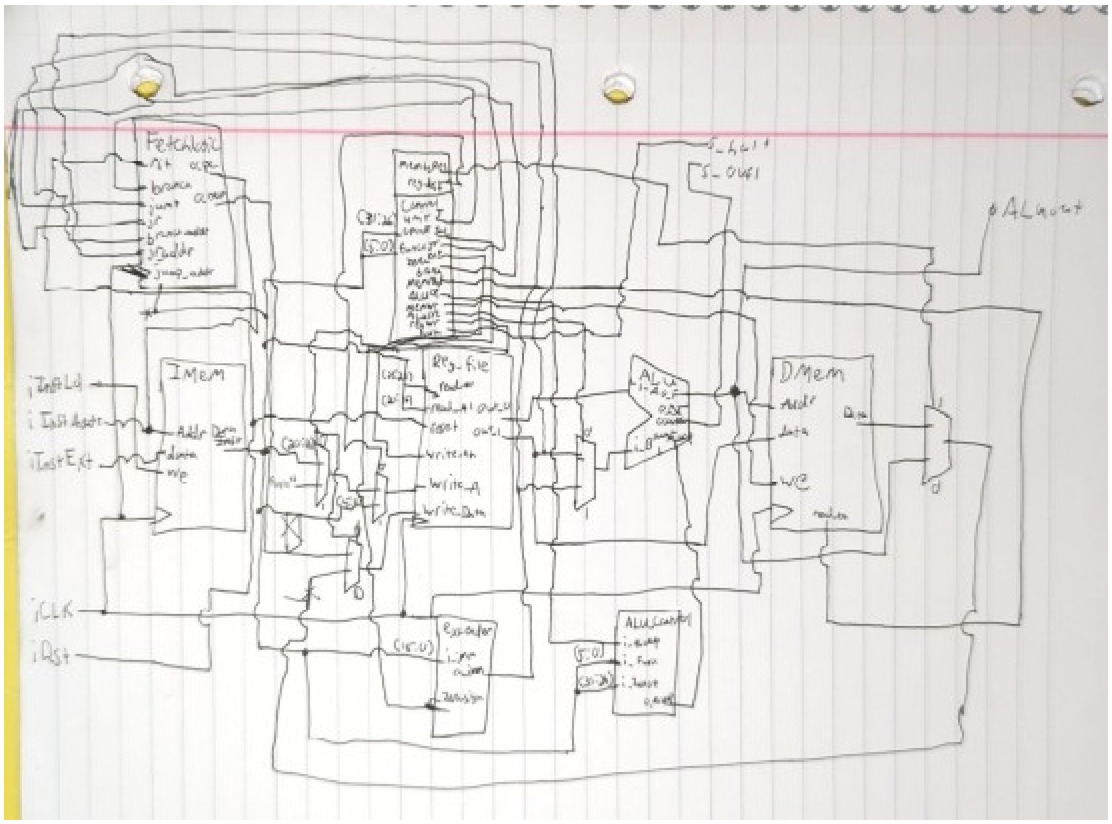
Project Part 1 Report

Team Members: **Jayson Acosta, Parnika Dasgupta, Nakota Clark**

Project Teams Group #: **Section D_01**

Refer to the highlighted language in the project 1 instruction for the context of the following questions.

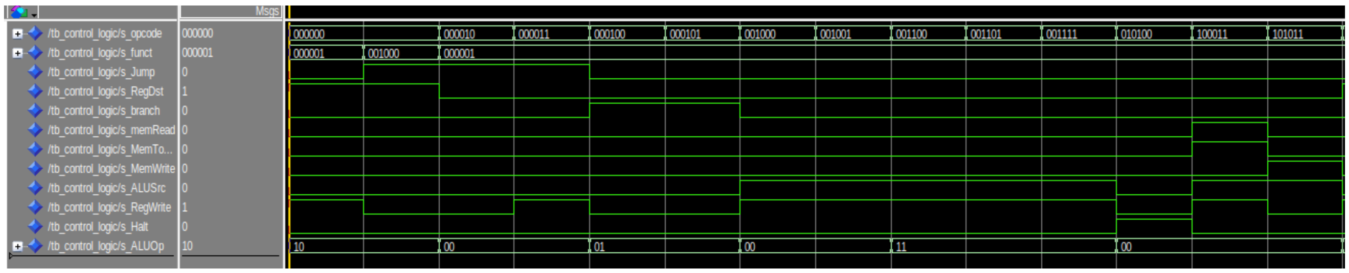
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary 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 by your datapath implementation. The end result should be an *N***M* table where each row corresponds to the output of the control logic module for a given instruction.

Instruction	Opcode	Funct	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
add	0	100000	1	0	0	0	0	10	0	0	1
addi	1000	N/A	0	0	0	0	0	0	0	1	1
addiu	1001	N/A	0	0	0	0	0	0	0	1	1
addu	0	100001	1	0	0	0	0	10	0	0	1
and	0	100100	1	0	0	0	0	10	0	0	1
andi	1100	N/A	0	0	0	0	0	0	0	1	1
beq	100	N/A	X	0	1	0	X	1	0	0	0
bne	101	N/A	X	0	1	0	X	1	0	0	0
j	10	N/A	X	1	0	0	X	XX	0	X	0
jal	11	N/A	X	1	0	0	X	XX	0	X	1
jr	0	1000	X	1	0	0	X	XX	0	X	0
lui	1111	N/A	0	0	0	0	0	0	0	1	1
lw	100011	N/A	0	0	0	1	1	0	0	1	1
nor	0	100111	1	0	0	0	0	10	0	0	1
or	0	100101	1	0	0	0	0	10	0	0	1
ori	1101	N/A	0	0	0	0	0	0	0	1	1
slt	0	101010	1	0	0	0	0	10	0	0	1
slti	1010	N/A	0	0	0	0	0	0	0	1	1
sll	0	0	1	0	0	0	0	10	0	0	1
srl	0	10	1	0	0	0	0	10	0	0	1
sra	0	11	1	0	0	0	0	10	0	0	1
sw	101011	N/A	X	0	0	0	X	0	1	1	0
sub	0	100010	1	0	0	0	0	10	0	0	1
subu	0	100011	1	0	0	0	0	10	0	0	1
xor	0	100110	1	0	0	0	0	10	0	0	1
xori	1110	N/A	0	0	0	0	0	0	0	1	1

[Part 2 (a.ii)] 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 problem 1(a).


Msgs
/tb_control_logic/s_opcode 000000
/tb_control_logic/s_funct 000001
/tb_control_logic/s_Jump 0
/tb_control_logic/s_RegDst 1
/tb_control_logic/s_branch 0
/tb_control_logic/s_memRead 0
/tb_control_logic/s_MemTo... 0
/tb_control_logic/s_MemWrite 0
/tb_control_logic/s_ALUSrc 0
/tb_control_logic/s_RegWrite 1
/tb_control_logic/s_Halt 0
/tb_control_logic/s_ALUOp 10
000000
000010
000011
000100
000101
001000
001001
001100
001101
001111
010100
100011
101011
000001
001000
000001
10
00
01
00
11
00

This testbench output shows that the control logic module is operating exactly as it should be based on our expected control signals. The test bench iterates through each of the opcodes anb functions. The output for all of these test cases match what they should be.

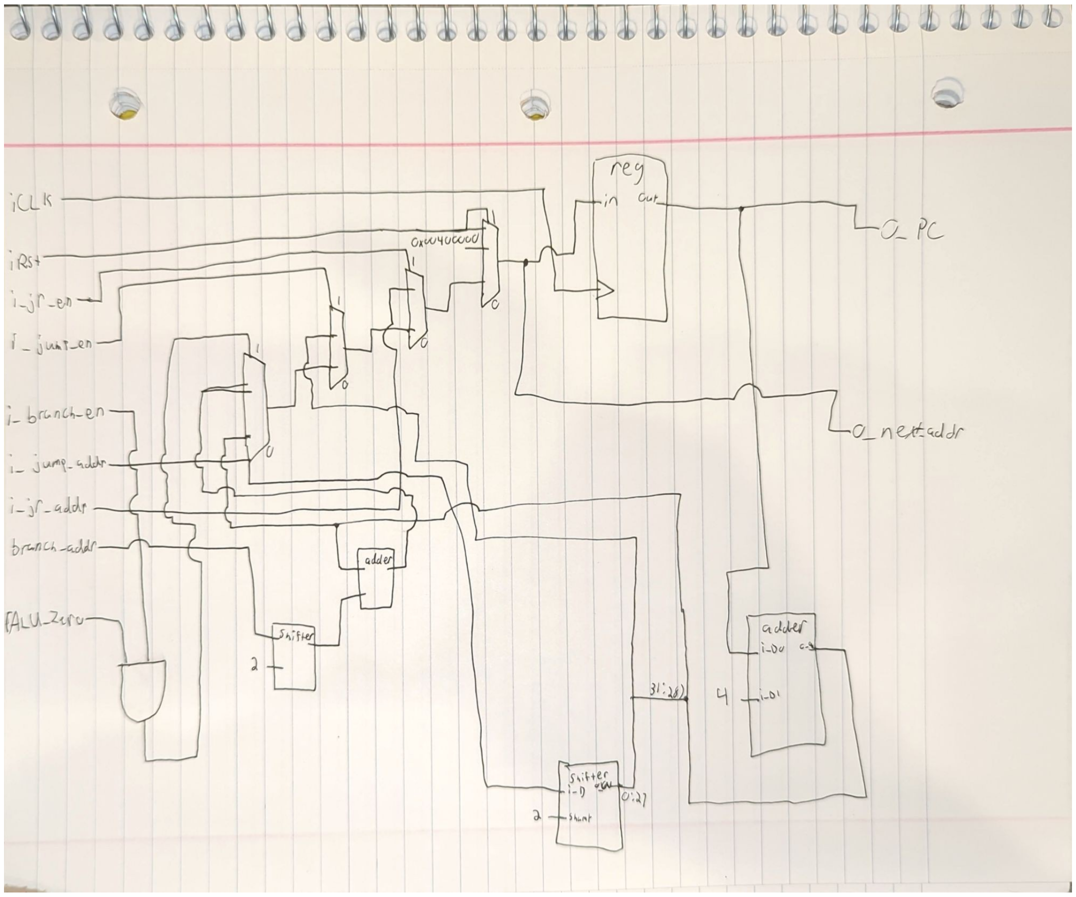
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

Our fetch logic supports the following control flow possibilities:

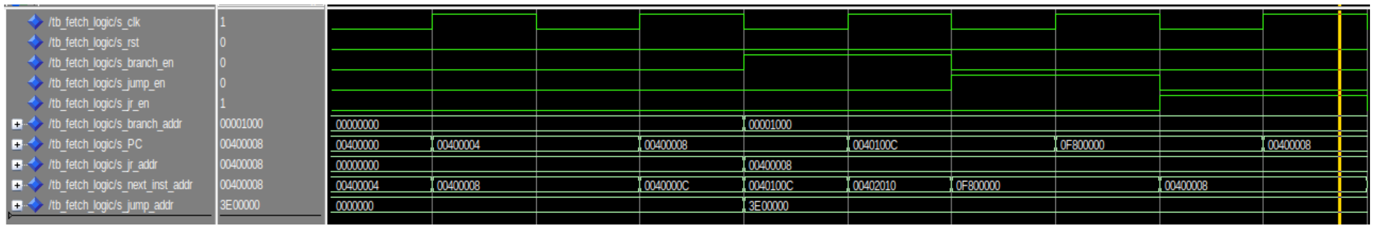
- **Sequential execution: PC = PC + 4**
- **Branch instructions (beq, bne): PC = PC + 4 + (sign-extended immediate << 2)**
- **Jump instructions (j, jal): PC = (PC + 4)[31:28] concatenated with (address << 2)**
- **Jump Register (jr): PC = register value**
- **These correspond to the MIPS branch (beq, bne), jump (j, jal), and jump register (jr) instructions.**

[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

We needed to add a reset signal to set the PC register to the proper initial value. We also needed to add signals to handle Jump Return logic. We added a Jump Return enable and a Jump Return Address.



[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use Modelsim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the Modelsim waveforms in your writeup.


/tb_fetch_logic/s_clk 1
/tb_fetch_logic/s_rst 0
/tb_fetch_logic/s_branch_en 0
/tb_fetch_logic/s_jump_en 0
/tb_fetch_logic/s_jr_en 1
/tb_fetch_logic/s_branch_addr 00001000 | 00000000 | 00001000
/tb_fetch_logic/s_PC 00400008 | 00400000 | 00400004 | 00400008 | 0040100C | 0F800000 | 00400008
/tb_fetch_logic/s_jr_addr 00400008 | 00000000 | 00400008
/tb_fetch_logic/s_next_inst_addr 00400008 | 00400004 | 00400008 | 0040000C | 0040100C | 00402010 | 0F800000 | 00400008
/tb_fetch_logic/s_jump_addr 3E00000 | 0000000 | 3E00000

This waveform shows that the fetch logic is working as expected. It was first reset and the value of PC went to 0x004000000 as expected. hen it went through 2 clock cycles without branching or jumping. The value increased by 4 each time as expected. Then was a branch instruction. This worked properly as 0x0040008 + 0x1000 + 0x4 = 0x0040100C. Next was a jump instruction with the jump address being 0b11111000000000000000000000. Shifted left by 2 it becomes 0b1111100000000000000000000000. Added with the 4 most significant bits of PC + 4 it becomes, 0b00001111100000000000000000000000. Which is 0x0F800000.

[Part 2 (c.i.1)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?

The difference between srl and sra is how it decides whether to shift in 1's or 0's. Logical shifts don't care about the numerical value, and so it shifts all the bits and place 0's in the new bits. Arithmetic shifts treat the values as numbers, and look at the sign bit to decide whether to shift in 0's or 1's. Sla does not exist because shifting to the left shifts in values to the least significant bits. Looking at the sign bit to decide which value to shift into these bits will not preserve the numerical value, and instead will make the number entirely different than expected. This means you should always shift in 0's, so sla would be the same as sll, and would be redundant to add separately.

Example of sla being incorrect:

-1 -4

sll 0b1111, 2 => 0b1100

-1 -1

sla 0b1111,2 => 0b1111

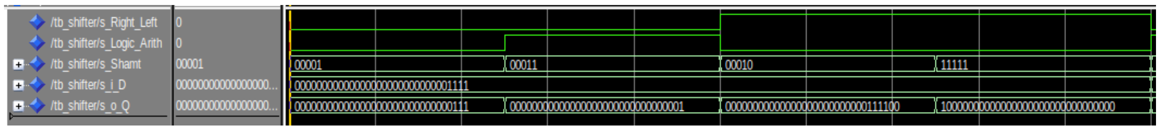
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

My code implements both arithmetic and logical operations by using a control signal and if/elsif statements. The control bit is Logic_Arith; 0 means logical and 1 means arithmetic. This value decides which branch of the if/elsif to take. For logical shifts, I shift the data and treat it as an unsigned number, so that it shifts in 0's. Arithmetic treats the data as a signed number, so it will shift in 0's or 1's depending on the sign bit.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

In order to support left shift operations, I will just add a new control signal Right_Left, and this will decide whether to do a left or right shift. I will concatenate this bit with the Logic_Arith bit and use that combined signal as the control for the if/elsif. If the Right_Left bit is 1, it doesn't matter what the Logic_Arith bit is. I'll only need to add 1 more branch to shift the data to the left.

[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the Modelsim waveforms in your writeup.

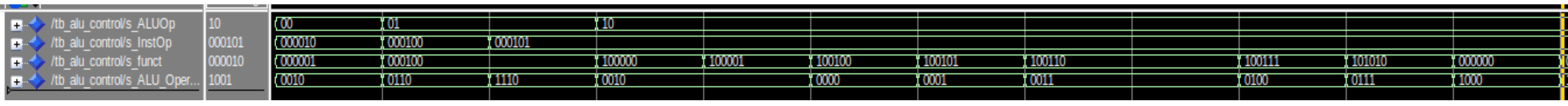


This image shows the shifter working properly for basic cases. I kept the input the same for all of them to make it easier for me to compare the results. The data in was always 0b00000000000000000000000000001111. The first operation was a srl operation with a shamt of 1. The result matches the expected result of 0b00000000000000000000000000000111. The second was a sra operation with a shamt of 3. It gave the expected result of 0b00000000000000000000000000000001. Then came a sll with a shamt of 2. It gave the expected result of 0b00000000000000000000000000111100. Finally there is a sll with a shamt of 31. It gave the expected result of 0b10000000000000000000000000000000. This shows that the operations are working as expected.

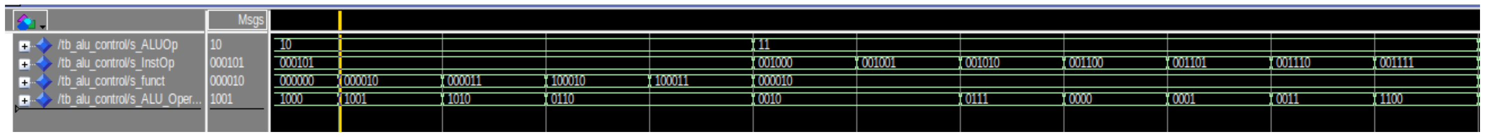
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

We didn't really have to design any other components to complete the ALU. We primarily used Behavioral instead of Structural. One component we did create was a separate ALU Controller to handle the logic that controls the ALU's output. This component was made to offload some of the control logic from the Control Logic module. This was done to simplify the code to prevent potential issues.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



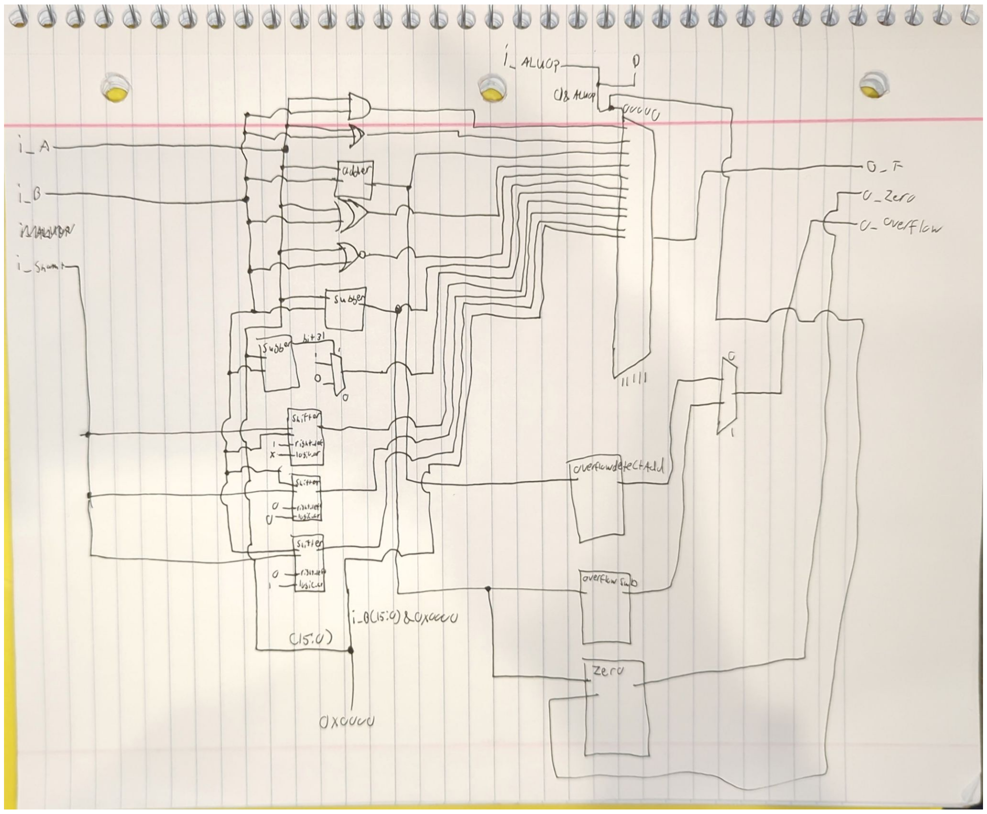
```
/tb_alu_control/s_ALUOp      10      00     01                    10
/tb_alu_control/s_InstOp     000101  000010 000100 000101
/tb_alu_control/s_funct      000010  000001 000100                100000 100001 100100 100101 100110        100111 101010 000000
/tb_alu_control/s_ALU_Oper...1001    0010   0110   1110           0010          0000   0001   0011          0100   0111   1000
```



```
                              Msgs
/tb_alu_control/s_ALUOp      10      10                                   11
/tb_alu_control/s_InstOp     000101  000101                               001000 001001 001010 001100 001101 001110 001111
/tb_alu_control/s_funct      000010  000000 000010 000011 100010 100011  000010
/tb_alu_control/s_ALU_Oper...1001    1000   1001   1010   0110           0010          0111   0000   0001   0011   1100
```

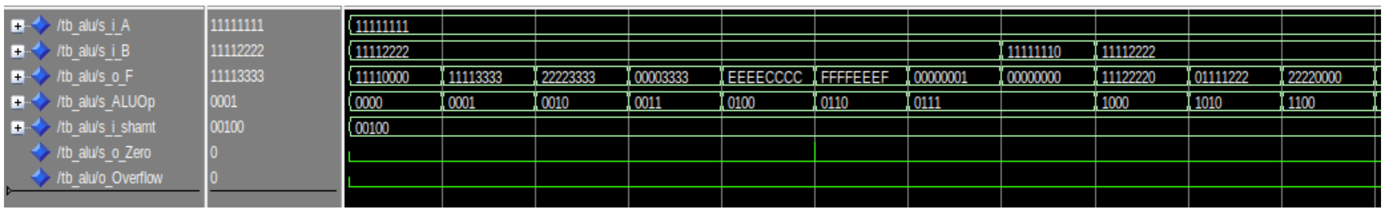
When executing instructions, we needed a way to tell what instruction type we were going to use. For this problem, we created alu_control.vhd. This file took in 3 parameters, i_ALUOp (from the main control), i_Funct (from instruction function field) and i_InstOp. ALUOp would be 00 for memory references or immediate instructions, 01 when it is a branch instruction, 10 when it is a R-type instruction and 11 when it is an immediate instruction. From this it would assign o_ALU_Operation with the appropriate instruction based on the instruction type and function code / op code for the instructions based on i_Funct (R-type) or i_InstOp (I-type and Branch)

[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?



Overflow is handled with a case statement that takes the most significant bit of the 2 inputs and the output. These 3 bits are concatenated together in a signal called s_overflow_detect. If the operation is addition if the signal is 001 or 110 then overflow occurred. If the operation is subtraction and the signal is 011 or 100 then overflow happened. Zero is calculated using a case statement as well. If the ALUOP is for beq and the result is 0, then Zero becomes 1. Otherwise it is zero. If the ALUOP for bne then if the result != 0, then Zero becomes 1. Slt is implemented using dataflow and simply uses < to compare the inputs and if it is true it outputs 1, otherwise it outputs 0.

[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.



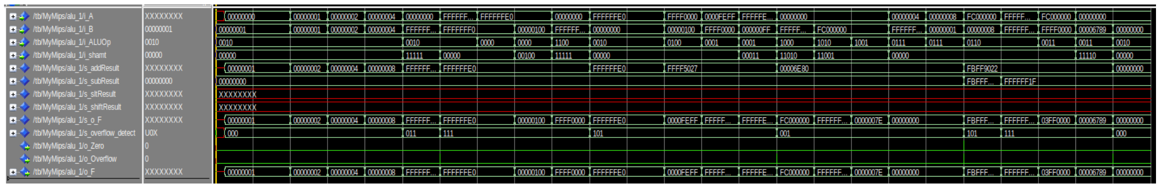
This was the output from a basic testbench that was made to simply ensure that the basic functionality worked properly. The testbench iterates through each of the basic ALU functions so we can verify the right operation happens. First the ALUOP is 0000 which is AND. inputs were 0x11111111 and 0x11112222. The output was 0x11110000. This is correct. Another example is Subtraction. The ALUOP is 0110. The inputs were 0x11111111 and 0x11112222. The output was 0xFFFFEEEF wich is also correct. This testbench was just to test basic functionality. We planned to do the real testing once it was integrated into the processor.

[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

Our test plan is comprehensive due to the amount of different operations and combinations in it. We received a lot of tests from the professor. There are a ton of unit tests for each of the components in the processor. There are also some mixed tests, such as one that does the Fibonacci sequence. We will also create our own extensive tests to verify each of our functions.

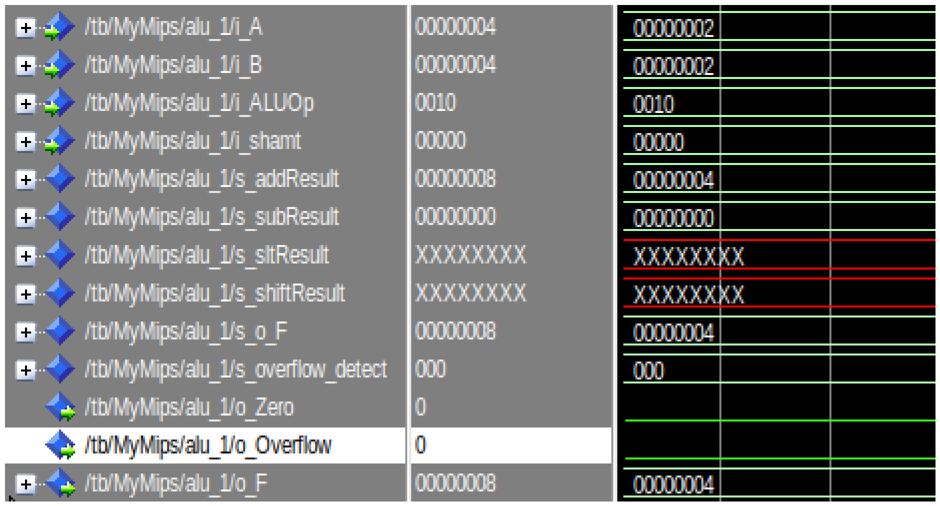
[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.



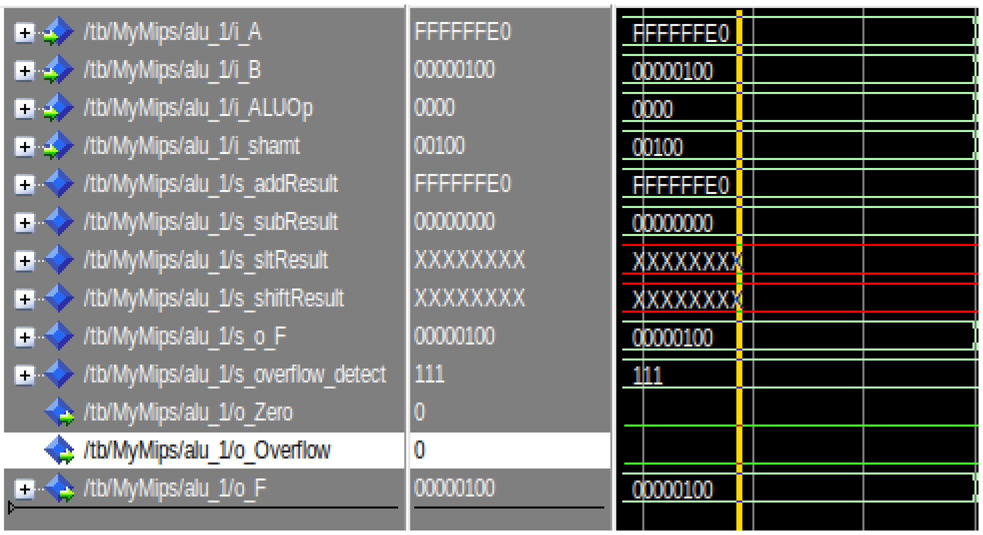
```
/tb/MyMips/alu_1/i_A              XXXXXXXX   00000000 00000001 00000002 00000004 00000000 FFFFFF... FFFFFFE0 00000000 FFFFFFE0 FFFF0000 0000FEFF FFFFFE... 00000000 00000004 00000008 FC000000 FFFFF... FC000000 00000000
/tb/MyMips/alu_1/i_B              00000001   00000001 00000001 00000002 00000004 FFFFFF... FFFFFFF0 00000100 FFFFFF... 00000000 00000100 FFFF0000 000000FF FFFFF... FC000000 FFFFFF... 00000001 00000008 FFFFFF... FFFF0000 00006789 00000000
/tb/MyMips/alu_1/i_ALUOp          0010       0010 0010 0000 0000 1100 0010 0100 0001 0001 1000 1010 1001 0111 0111 0110 0011 0011 0010
/tb/MyMips/alu_1/i_shamt          00000      00000 11111 00000 00100 11111 00000 00011 11010 11001 00000 11110 00000
/tb/MyMips/alu_1/s_addResult      XXXXXXXX   00000001 00000002 00000004 00000008 FFFFFF... FFFFFFE0 FFFFFFE0 FFFF5027 00006E80 FBFF9022 00000000
/tb/MyMips/alu_1/s_subResult      00000000   00000000 FBFFF... FFFFFF1F
/tb/MyMips/alu_1/s_sltResult      XXXXXXXX   XXXXXXXX
/tb/MyMips/alu_1/s_shiftResult    XXXXXXXX   XXXXXXXX
/tb/MyMips/alu_1/s_o_F            XXXXXXXX   00000001 00000002 00000004 00000008 FFFFFF... FFFFFFE0 00000100 FFFF0000 FFFFFFE0 0000FEFF FFFFF... FFFFFE... FC000000 FFFFFF... 0000007E 00000000 FBFFF... FFFFFF... 03FF0000 00006789 00000000
/tb/MyMips/alu_1/s_overflow_detect U0X       000 011 111 101 001 101 111 000
/tb/MyMips/alu_1/o_Zero           0
/tb/MyMips/alu_1/o_Overflow       0
/tb/MyMips/alu_1/o_F              XXXXXXXX   00000001 00000002 00000004 00000008 FFFFFF... FFFFFFE0 00000100 FFFF0000 FFFFFFE0 0000FEFF FFFFF... FFFFFE... FC000000 FFFFFF... 0000007E 00000000 FBFFF... FFFFFF... 03FF0000 00006789 00000000
```

This waveform demonstrates our processor successfully completing our test program for all of the reqauiired arithmetic/logical operations. This is a waveform of the ALU in poarticular, because it handles all of these instructions, so it can tell you if it is working properly without having all of the other signals to sift through. All of the tests completed successfully, but i'll give some examples.



```
/tb/MyMips/alu_1/i_A               00000004   00000002
/tb/MyMips/alu_1/i_B               00000004   00000002
/tb/MyMips/alu_1/i_ALUOp           0010       0010
/tb/MyMips/alu_1/i_shamt           00000      00000
/tb/MyMips/alu_1/s_addResult       00000008   00000004
/tb/MyMips/alu_1/s_subResult       00000000   00000000
/tb/MyMips/alu_1/s_sltResult       XXXXXXXX   XXXXXXXX
/tb/MyMips/alu_1/s_shiftResult     XXXXXXXX   XXXXXXXX
/tb/MyMips/alu_1/s_o_F             00000008   00000004
/tb/MyMips/alu_1/s_overflow_detect 000        000
/tb/MyMips/alu_1/o_Zero            0
/tb/MyMips/alu_1/o_Overflow        0
/tb/MyMips/alu_1/o_F               00000008   00000004
```

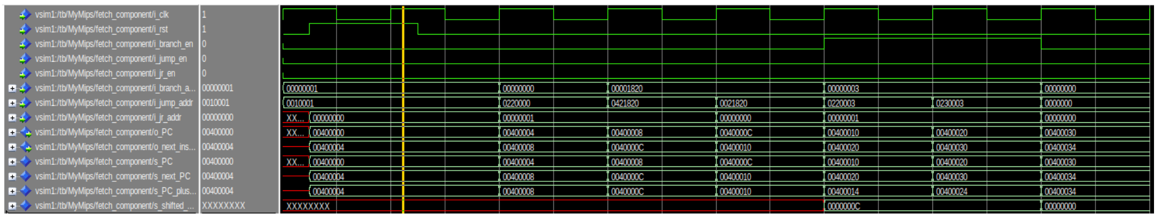
This image shows that addition is working, as 2+2 =4 as the output demonstrates.



```
/tb/MyMips/alu_1/i_A                FFFFFFE0
/tb/MyMips/alu_1/i_B                00000100
/tb/MyMips/alu_1/i_ALUOp            0000
/tb/MyMips/alu_1/i_shamt            00100
/tb/MyMips/alu_1/s_addResult        FFFFFFE0
/tb/MyMips/alu_1/s_subResult        00000000
/tb/MyMips/alu_1/s_sltResult        XXXXXXXX
/tb/MyMips/alu_1/s_shiftResult      XXXXXXXX
/tb/MyMips/alu_1/s_o_F              00000100
/tb/MyMips/alu_1/s_overflow_detect  111
/tb/MyMips/alu_1/o_Zero             0
/tb/MyMips/alu_1/o_Overflow         0
/tb/MyMips/alu_1/o_F                00000100
```

This image shows and ANDI operations completing successfully. it got 0xFFFFFFE0 from a register and 0x00000100 is the immediate from the instructions. The result of 0x00000100 is the correct output.

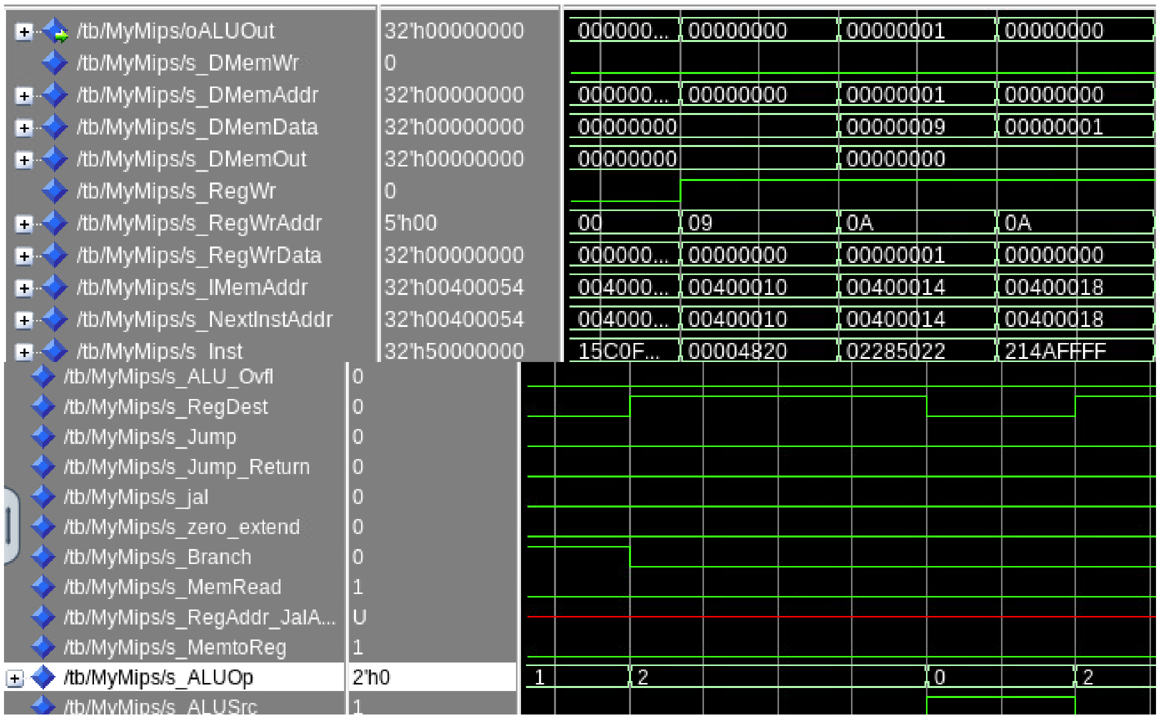
[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.



```
vsim1:/tb/MyMips/fetch_component/i_clk          1
vsim1:/tb/MyMips/fetch_component/i_rst          1
vsim1:/tb/MyMips/fetch_component/i_branch_en    0
vsim1:/tb/MyMips/fetch_component/i_jump_en      0
vsim1:/tb/MyMips/fetch_component/i_jr_en        0
vsim1:/tb/MyMips/fetch_component/i_branch_a...  00000001
vsim1:/tb/MyMips/fetch_component/i_jump_addr    0010001
vsim1:/tb/MyMips/fetch_component/i_jr_addr      00000000
vsim1:/tb/MyMips/fetch_component/o_PC           00400000
vsim1:/tb/MyMips/fetch_component/o_next_ins...  00400004
vsim1:/tb/MyMips/fetch_component/s_PC           00400000
vsim1:/tb/MyMips/fetch_component/s_next_PC      00400004
vsim1:/tb/MyMips/fetch_component/s_PC_plus...   00400004
vsim1:/tb/MyMips/fetch_component/s_shifted_... XXXXXXXX
```

This waveform demonstrates the branch and jump instructions operate properly and the PC is properly updated. The value for PC after each of the instructions match the expected results for each of the cases.

[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

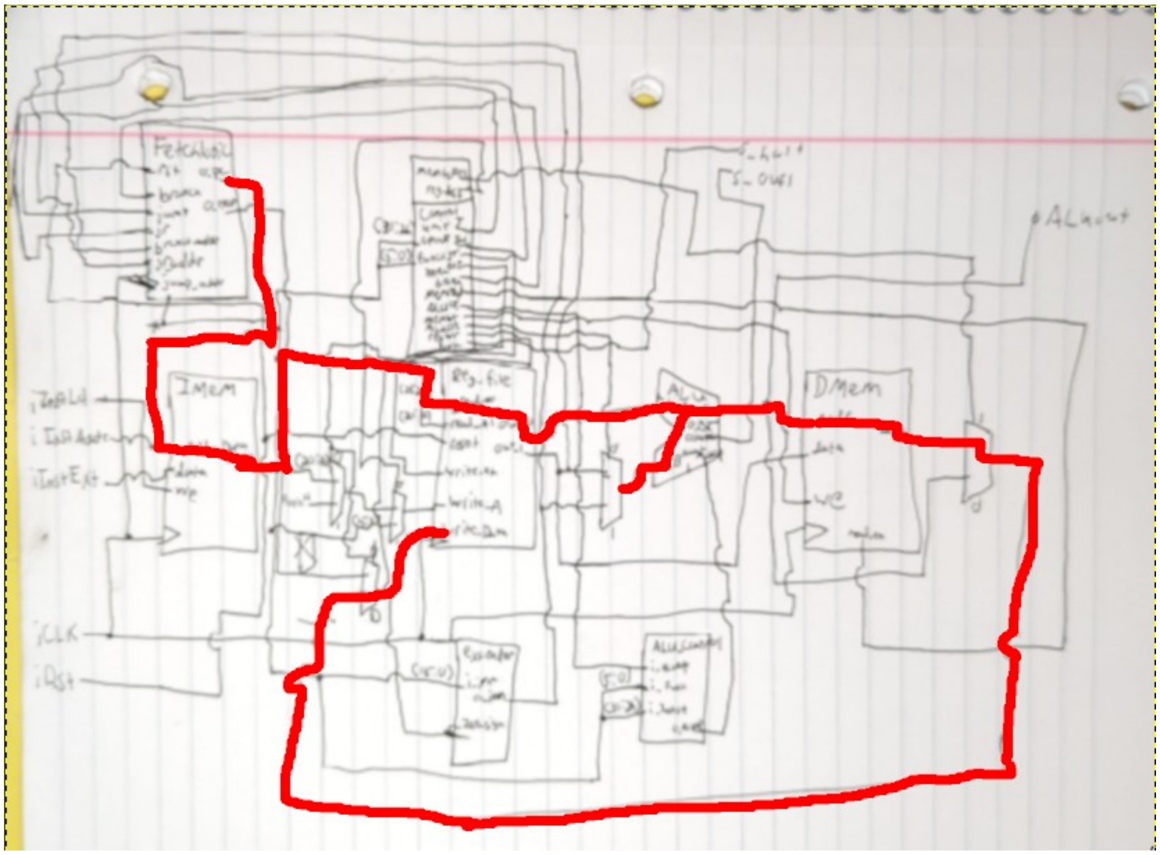


```
/tb/MyMips/oALUOut          32'h00000000   000000...  00000000  00000001  00000000
/tb/MyMips/s_DMemWr         0
/tb/MyMips/s_DMemAddr       32'h00000000   000000...  00000000  00000001  00000000
/tb/MyMips/s_DMemData       32'h00000000   00000000             00000009  00000001
/tb/MyMips/s_DMemOut        32'h00000000   00000000             00000000
/tb/MyMips/s_RegWr          0
/tb/MyMips/s_RegWrAddr      5'h00          00         09        0A        0A
/tb/MyMips/s_RegWrData      32'h00000000   000000...  00000000  00000001  00000000
/tb/MyMips/s_IMemAddr       32'h00400054   004000...  00400010  00400014  00400018
/tb/MyMips/s_NextInstAddr   32'h00400054   004000...  00400010  00400014  00400018
/tb/MyMips/s_Inst           32'h50000000   15C0F...   00004820  02285022  214AFFFF
/tb/MyMips/s_ALU_Ovfl       0
/tb/MyMips/s_RegDest        0
/tb/MyMips/s_Jump           0
/tb/MyMips/s_Jump_Return    0
/tb/MyMips/s_jal            0
/tb/MyMips/s_zero_extend    0
/tb/MyMips/s_Branch         0
/tb/MyMips/s_MemRead        1
/tb/MyMips/s_RegAddr_JalA...  U
/tb/MyMips/s_MemtoReg       1
/tb/MyMips/s_ALUOp          2'h0           1    2                   0         2
/tb/MyMips/s_ALUSrc         1
```

This waveform demonstrates the bubble sort algorithm, showing the comparisons, swaps, ALU results and the loops in the MIPS processor. s_DMemData shows the unsorted array, and when s_DMemWr goes high, a swap is occuring while s_DMemAddr shows which element is being accessed. The loops can be seen occuring with the branch instruction going from low to high.

[Part 4] 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 schematics. What components would you focus on to improve the frequency?

The reported FMax in the timing output is 26.76mhz. The critical path is the path taken by lw. The critical path involves reading a value from a register, operating on it in the ALU, and using this value as an input to the DMEM. Then the output of DMEM goes into the registers write_data port. This would be a lw instruction.



In order to improve the frequency of our processor we should focus on the register file and ALU. Those were the components that we made that had the highest latency. They took about 4ns and 5ns respectively. If we had a way to improve the memory that would likely help the most. There are 2 memory modules and they have the longest latency at about 8ns.