CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

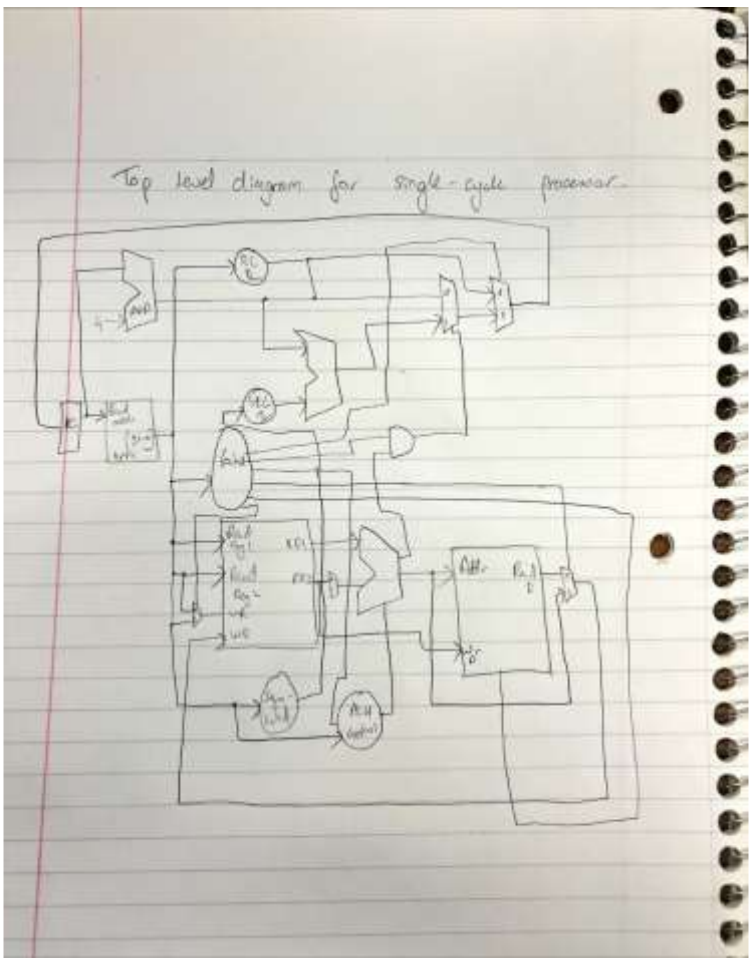
Team Members: Varun Advani

Kalyan Thapaliya

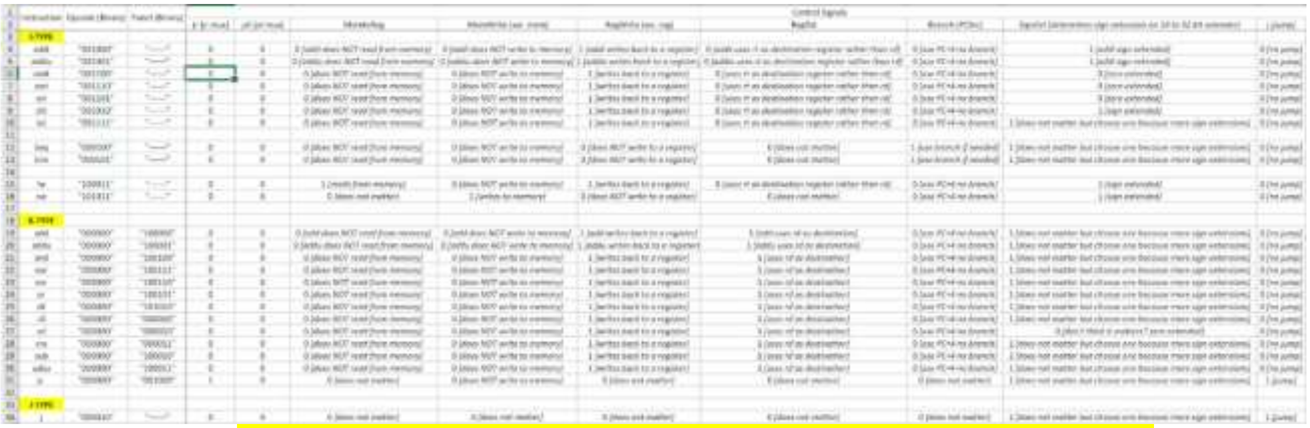
Project Teams Group #: 6

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.

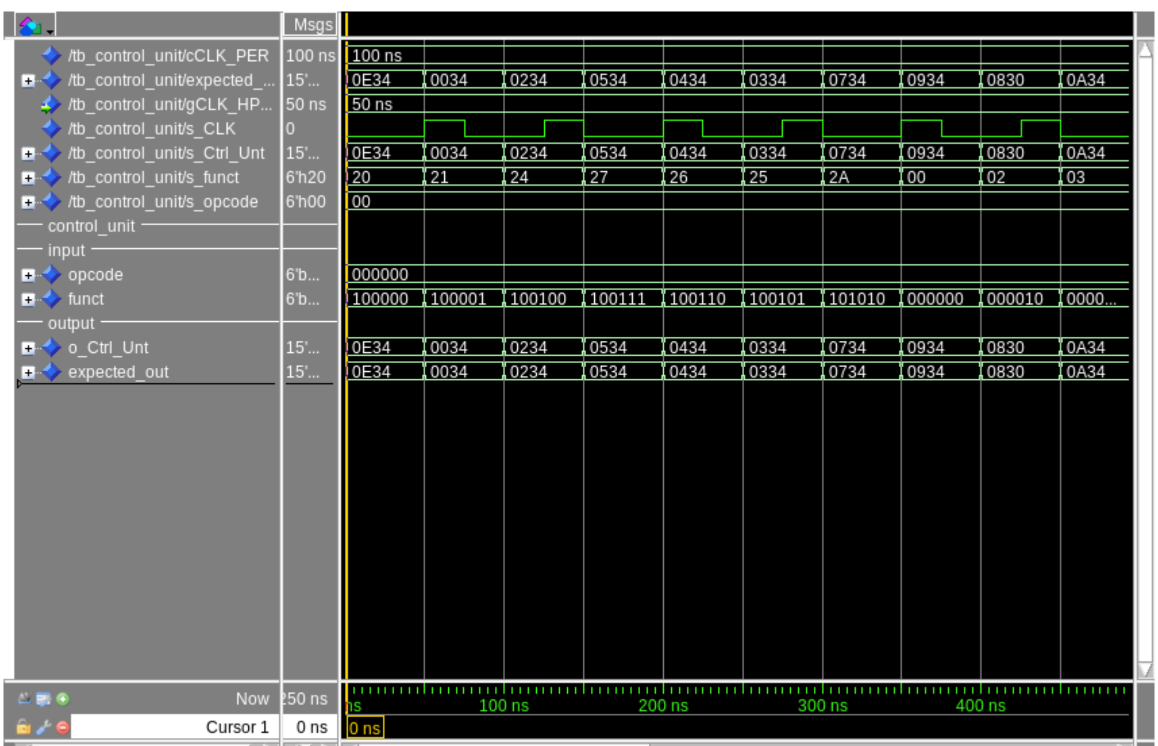


[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your 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.



[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).

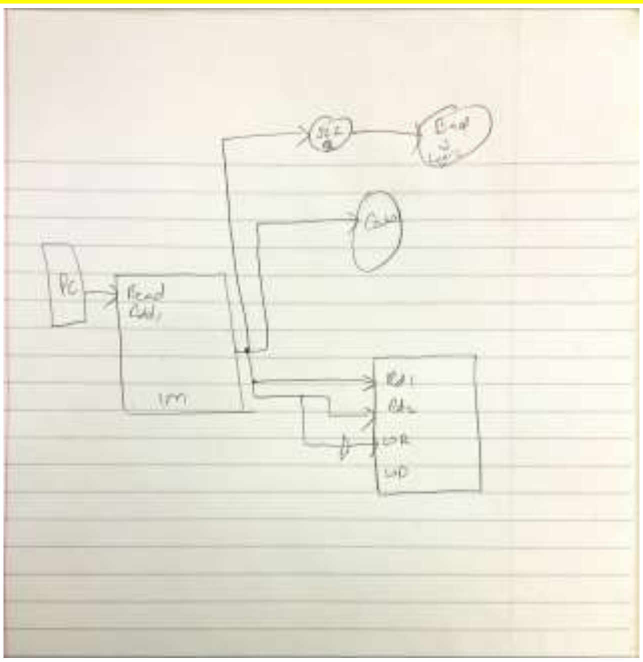
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity control_unit is
        port(i_opcode           : in std_logic_vector(5 downto 0);
             i_funct            : in std_logic_vector(5 downto 0);
             o_Ctrl_Unt         : out std_logic_vector(14 downto 0));
end control_unit;


architecture dataflow of control_unit is
signal s_RTYPE : std_logic_vector(14 downto 0);
begin


with i_funct select s_RTYPE <=
    "000111000110100"  when "100000",
    "000000000110100"  when "100001",
    "000001000110100"  when "100100",
    "000010100110100"  when "100111",
    "000010000110100"  when "100110",
    "000001100110100"  when "100101",
    "000011100110100"  when "101010",
    "000011100110100"  when "101011",
    "000100100110100"  when "000000",
    "000100000110000"  when "000010",
    "000101000110100"  when "000011",
    "000111100110100"  when "100010",
    "000000100110100"  when "100011",
    "100000000000110"  when "001000",
    "000000000000000"  when others;

with i_opcode select o_Ctrl_Unt <=
    s_RTYPE            when "000000",
    "001111000100100"  when "001000",

    "000000000000001"  when "010100",

    "001000000100100"  when "001001",
    "001001000100000"  when "001100",
    "001010000100000"  when "001110",
    "001001100100000"  when "001101",
    "001011100100100"  when "001010",
    "001011100100100"  when "001011",
    "001011000100100"  when "001111",
    "000101100001100"  when "000100",
    "000110000001100"  when "000101",
    "001000010100100"  when "100011",
    "001000001000100"  when "101011",
    "000000000000110"  when "000010",
    "010000000100110"  when "000011",
    "000000000000000"  when others;

end dataflow;
```



[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.

The given instructions that need to be simulate would all have unique control unit values depending on the path. . In the I-type instructions, the ALUSrc receives an immediate value instead of a value read from the register.
There are two control signals that control the addressing of the Jump and branch instructions. The branch instruction needs a ALU output The jump register can receive an address from a register or a 26 bit immediate value. The control signal for the Jump and Link writes the address +4 that is computed the ALU.
There are two control signal for the write enable, and two control signals that control the write back data of the rs and rd registers from the data memory or the ALU.
The sign extend control signal determines whether the immediate value would be zero or sign extended.

[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?

Control Signal is roughly 15 bits for the fetch instruction which includes, jr, jal, ALUSrc, MemtoReg, MemWrite, RegWrite, RegDst, Branch, SignExt, j, and the 4 bit ALU opcode.

[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.
I've tested out each instruction to see if it fetches correctly, did not have time to make a testbench for it.

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

Srl is a logical shift right which means that it shifts the bits from left to rights, and the bits added would be zero, irrespective of the most significant bit. An arithmetic shift would account for the the Most Significant bit and hence is useful for shifting negative numbers whilst holding its value.

SLA shifts from right to left however, the Least Significant bit won't impact the sign and hence an arithmetic instruction for shift left is not required.

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

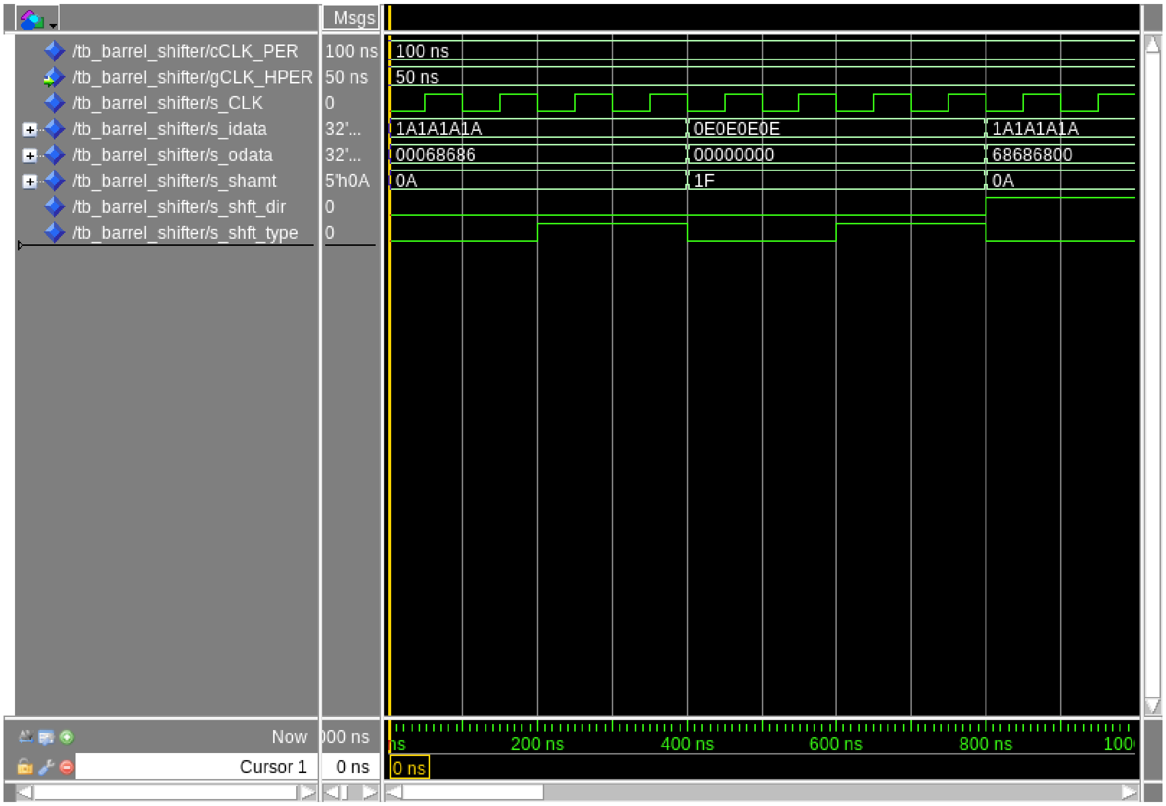
We added a 0 or the binary value of the MSB, with the help of a 2 to 1 MUX that is controlled by a control signal of the control unit.

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

The left shifting would make the MSB the LSB, bit 30 bit 1 up until the shift amount. Therefore a 2 to 1 MUX canbe used to either re-shift the left shifted output, or use the right shift output. A second MUX may also be required, select between the right shift input and the left shift input. Control bit would be 0.

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

Type gives you the type of shift, 0 is logical and 1 is arithmetic. Dir gives direction, 0 is right and 1 is left, and shamt gives shift amount and data is the value being shifted.



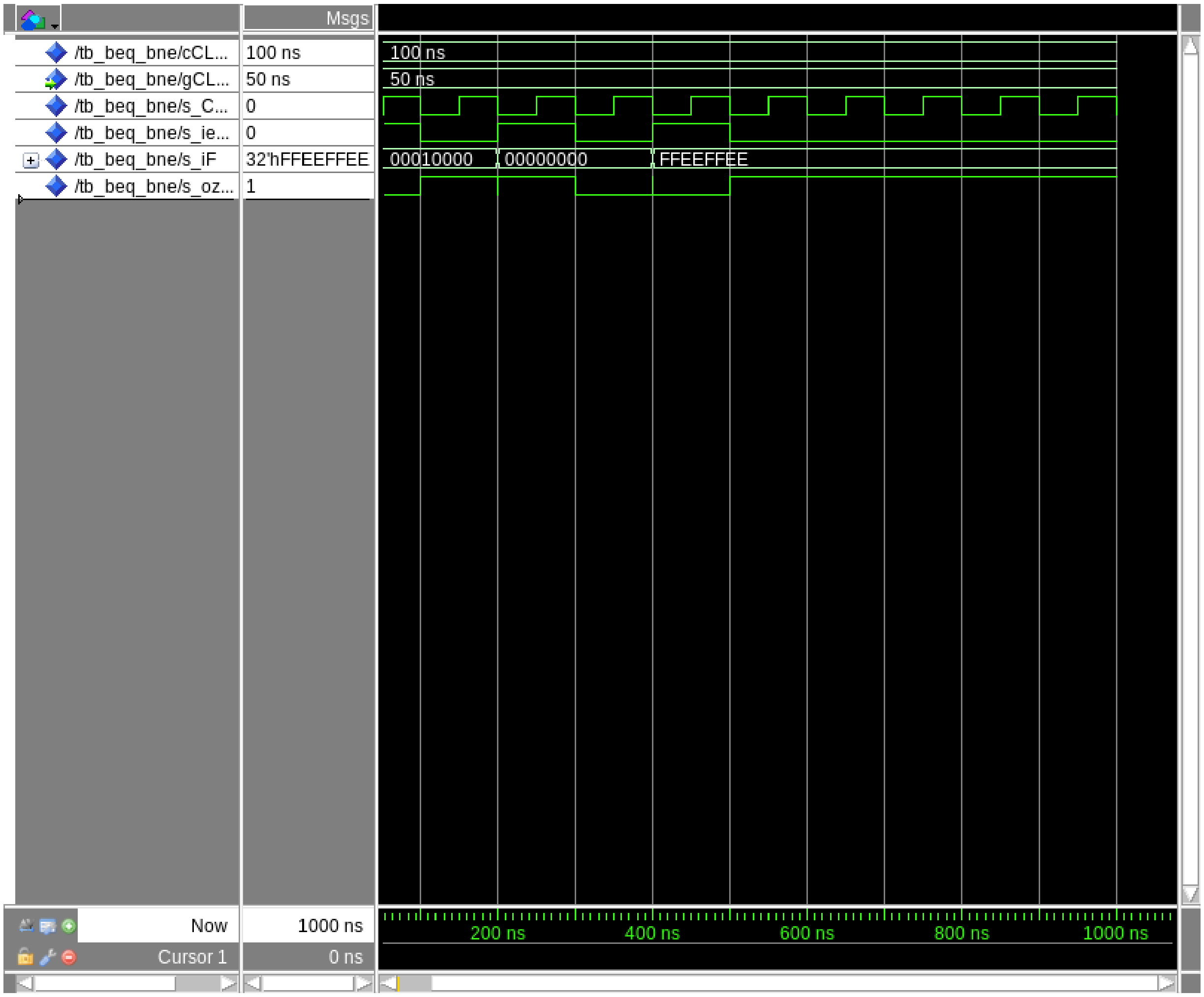
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

The way we thought of doing the ALU was to figure out how we got implement similar types of instructions into smaller blocks within the ALU. This would make it easier for testing and improve reusability to try and make the ALU as efficient as possible. Some of the instructions like the barrel shifter for the shifts were a little complicated.

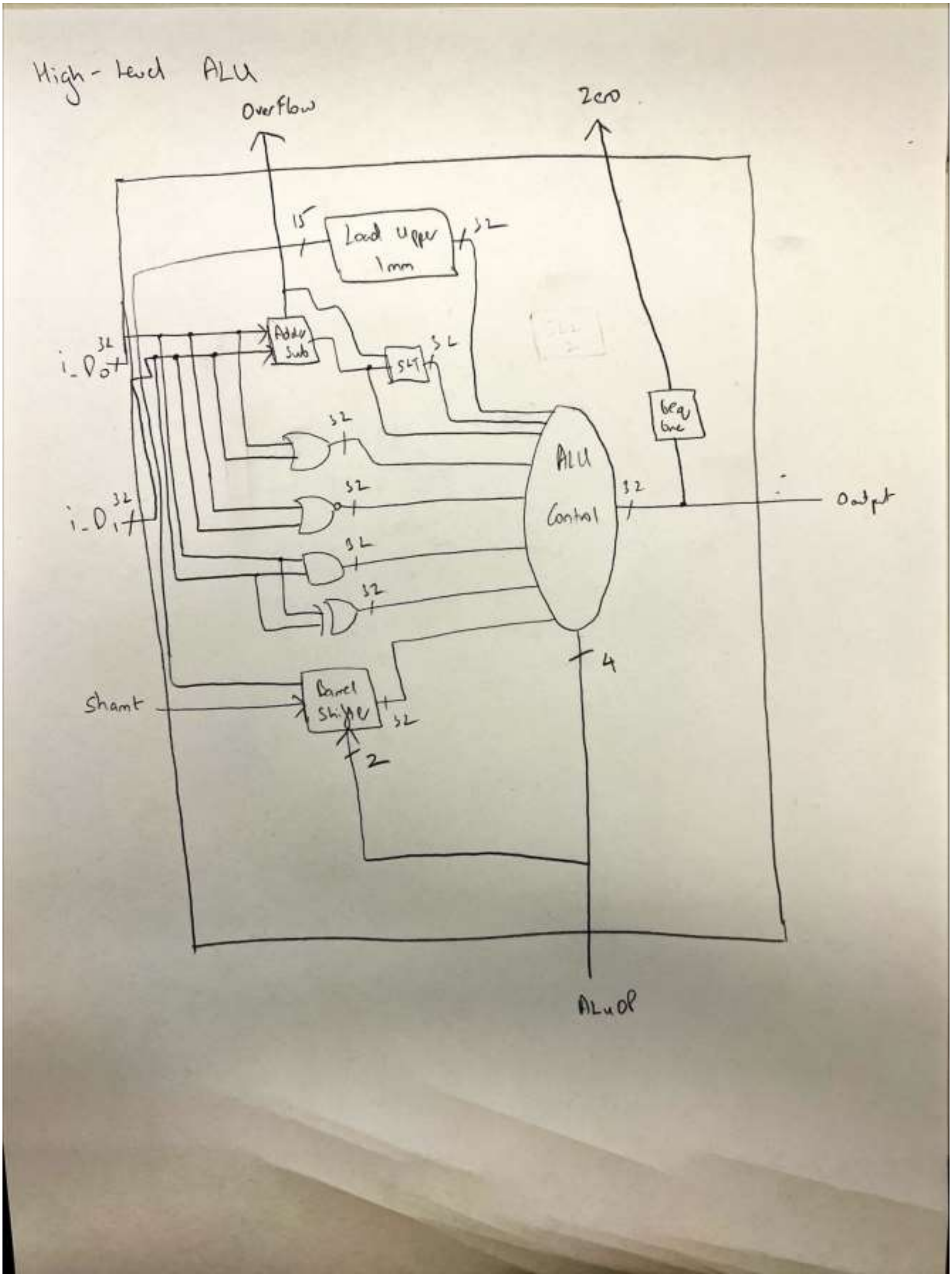
[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms

This question is repeated I think but I put in one of the components anyway

The execution of the different operations will be shown and annotated in the testbench simulations below. The other important components of the ALU that have been tested is the component for the BEQ and BNE logic the waveform for which is as attached below.



[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?



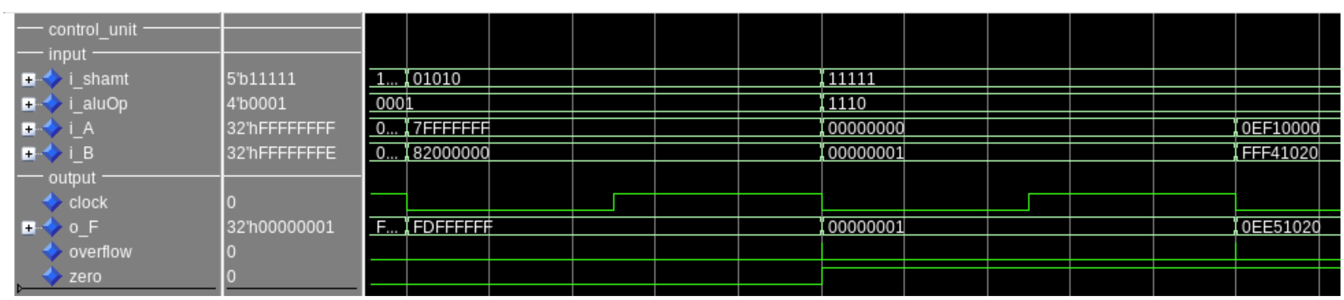
Slt logic was the computed by subtracting the D1 – D0 with the sign of the output being considered. The logic for the overflow was a simple AND gate. Overflow was computed inside the adder/subtracter. The zero was computed through the BEQ and BNE functional block which takes the ALU control output as its input, to compute the zero.

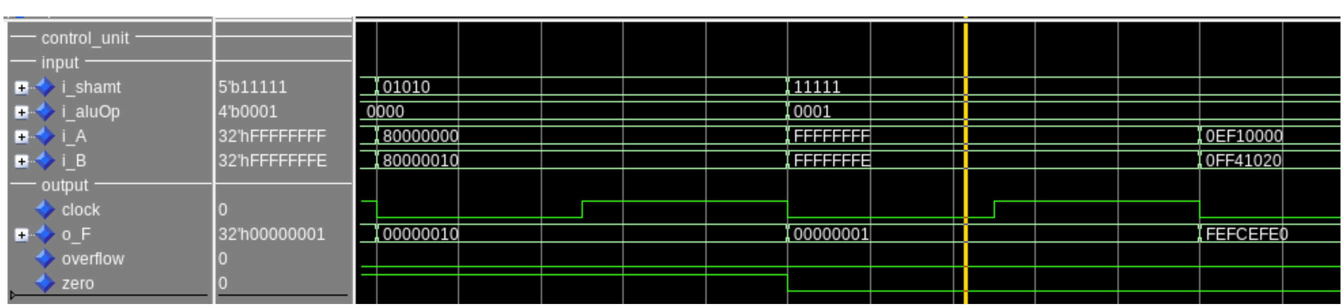
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

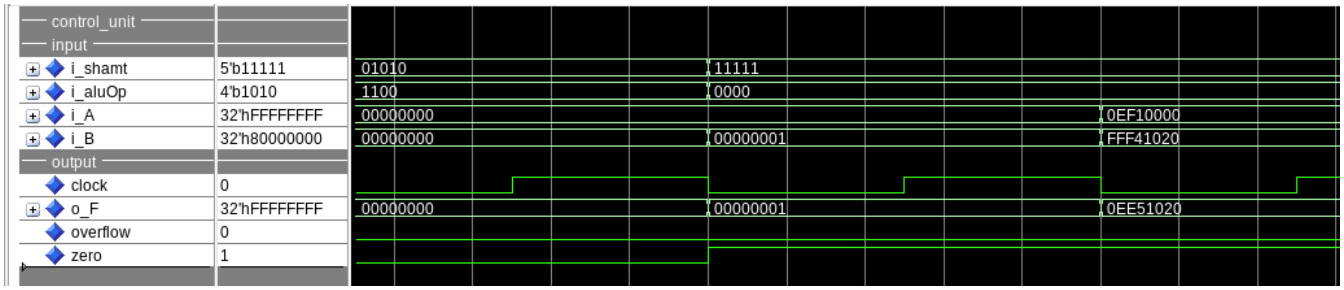
We ran a testbench with all the possible arithmetic instructions supported by the ALU, as well as edge cases to make sure the ALU works as expected. The testbench and the do file is in the test folder and attached below is a screenshot of the waveforms:

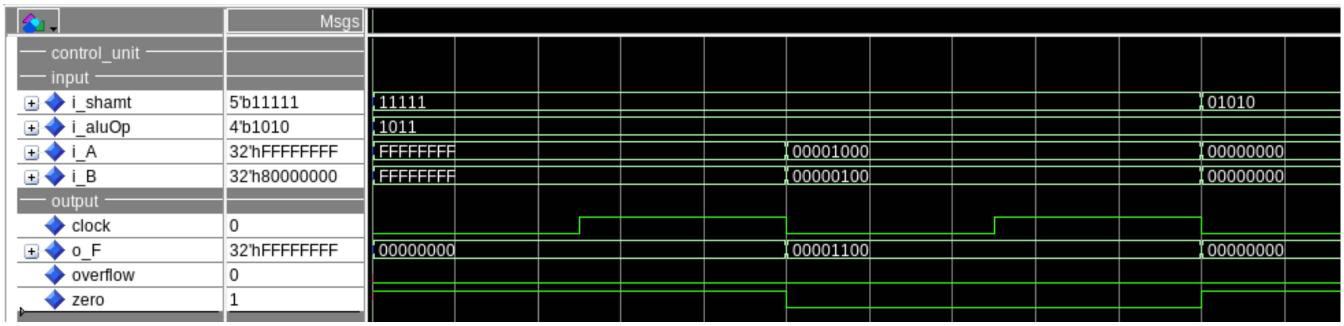
Tests conducted were:
1. Test for beq
2. Test for bne
3. Test for addu
4. Test for subu
5. Test for add
6. Test for sub
7. Test for and
8. Test for or
9. Test for xor
10. Test for nor
11. Test for lui
12. Test for slt
13. Test for sll
14. Test for srl
15. Test for sra

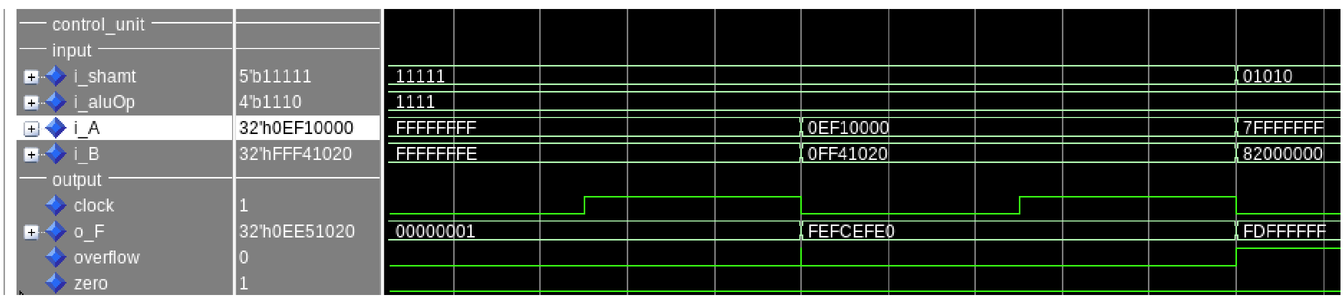
Could not attached screenshots for all but here is some of them.






control_unit
input
i_shamt 5'b11111
i_aluOp 4'b1010
i_A 32'hFFFFFFFF
i_B 32'h80000000
output
clock 0
o_F 32'hFFFFFFFF
overflow 0
zero 1
01010
11111
1100
0000
00000000
0EF10000
00000000
00000001
FFF41020
00000000
00000001
0EE51020


Msgs
control_unit
input
i_shamt 5'b11111
i_aluOp 4'b1010
i_A 32'hFFFFFFFF
i_B 32'h80000000
output
clock 0
o_F 32'hFFFFFFFF
overflow 0
zero 1
11111
01010
1011
FFFFFFFF
00001000
00000000
FFFFFFFF
00000100
00000000
00000000
00001100
00000000


control_unit
input
i_shamt 5'b11111
i_aluOp 4'b1110
i_A 32'h0EF10000
i_B 32'hFFF41020
output
clock 1
o_F 32'h0EE51020
overflow 0
zero 1
11111
01010
1111
FFFFFFFF
0EF10000
7FFFFFFF
FFFFFFFE
0FF41020
82000000
00000001
FEFCEFE0
FDFFFFFF

[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.

No time to annotate waveforms but all tests ran on the testing framework with no issues.

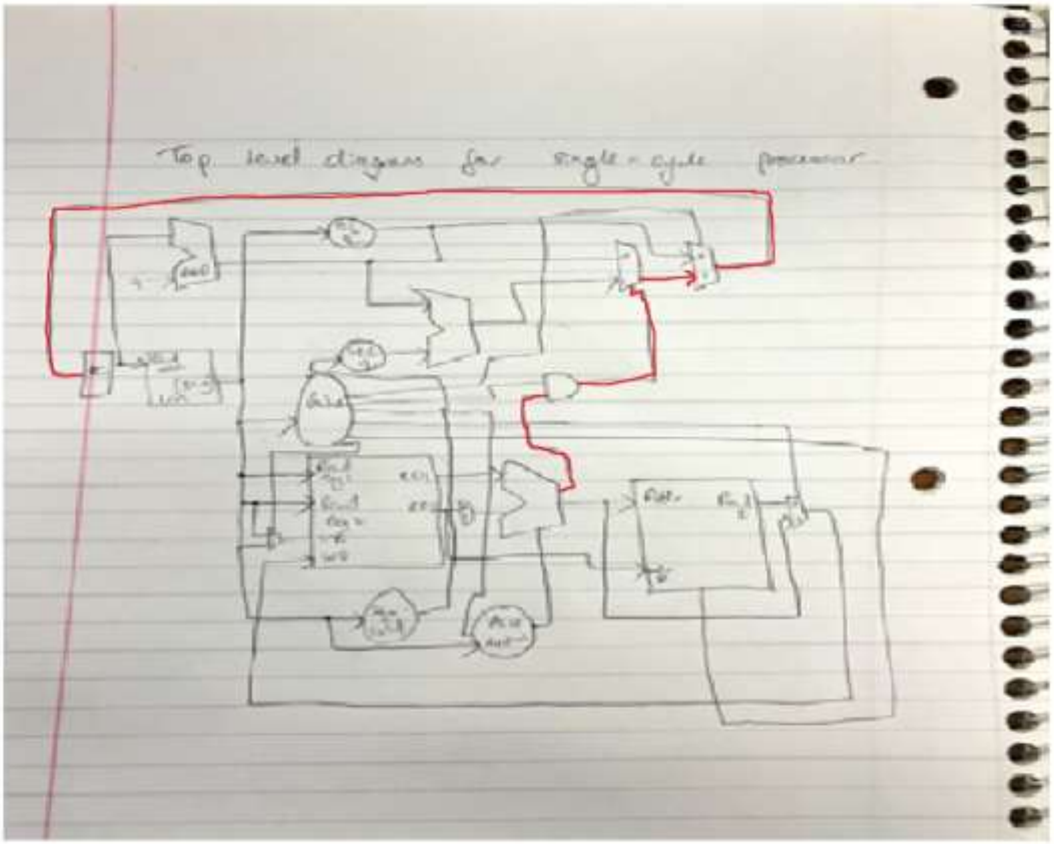
[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially,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

[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.

[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.

[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?

Fmax is 26.3 mhz with a slack of -17.67 ns
Critical Path seems to be the branch instruction from the ALU, without going through the Data Memory, which comes to approximately 43 ns
I think the ALU itself could definitely be optimized because the ALU must make a redundant comparison to 0 based on the output of ALU output signal, which makes the branch instruction's path the longest one in the processor. I feel like we could've come up with a better logic flow by assigning a sign bit in the control unit and better usage of logic gates to improve our processor, but we just ran out of time!


Top level diagram for single-cycle processor