CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: ________Alek Norris __________

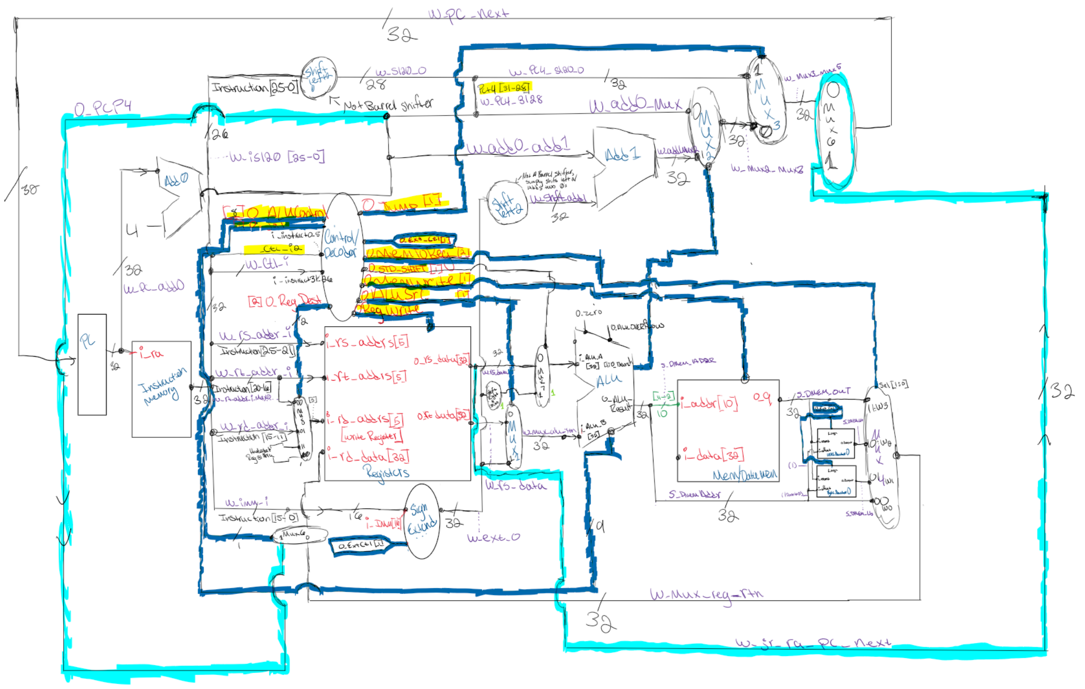
________Drew Kearns _____ ______________

____

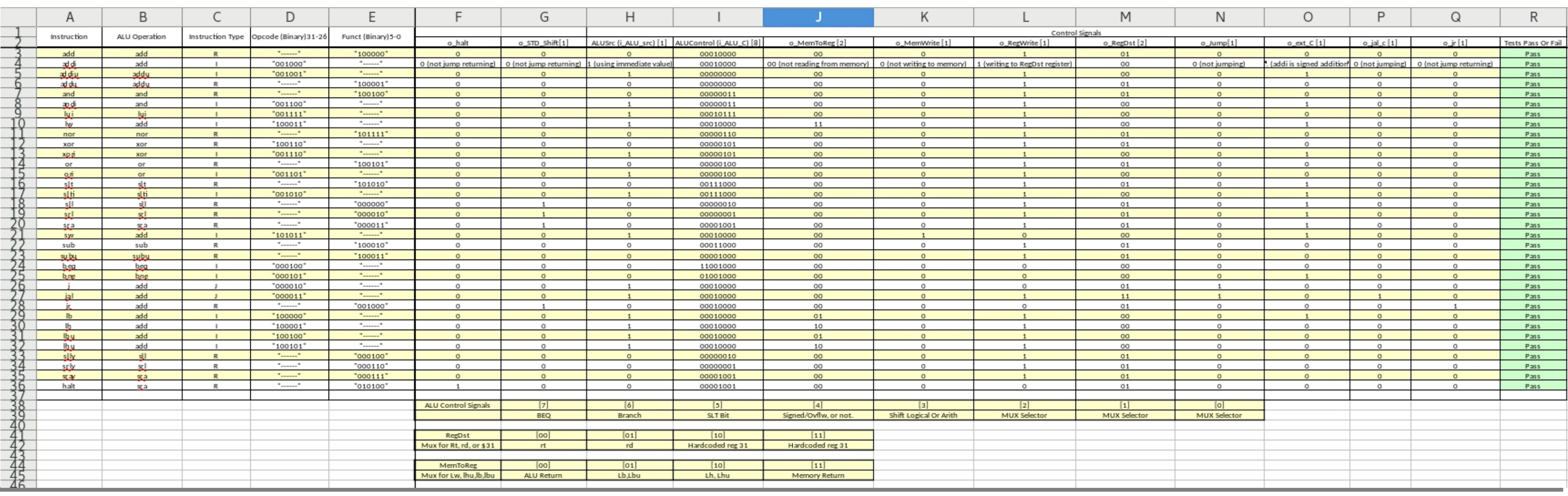
Project Teams Group #:_____Term Proj1_2_07___________________

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.



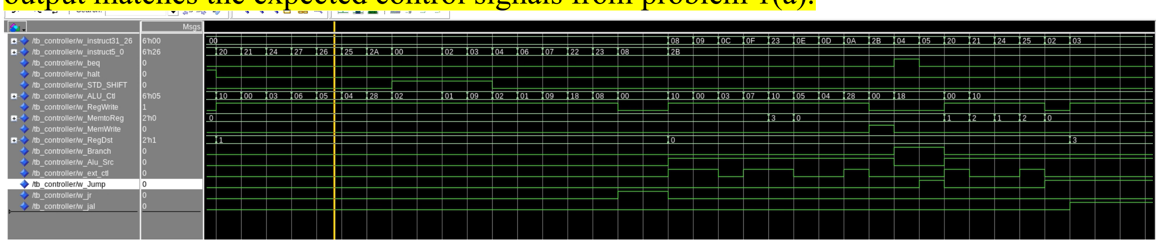
											Control Signals						
Instruction	ALU Operation	Instruction Type	Opcode (Binary)31-26	Funct (Binary)5-0	o_halt	o_STD_Shift[1]	ALUSrc (i_ALU_src) [1]	ALUControl (i_ALU_C) [8]	o_MemToReg [2]	o_MemWrite [1]	o_RegWrite [1]	o_RegDst [2]	o_Jump[1]	o_ext_C [1]	o_jal_c [1]	o_jr [1]	Tests Pass Or Fail
add	add	R	"------"	"100000"	0	0	0	00010000	00	0	1	01	0	0	0	0	Pass
addi	add	I	"001000"	"------"	0 (not jump returning)	0 (not jump returning)	1 (using immediate value)	00010000	00 (not reading from memory)	0 (not writing to memory)	1 (writing to RegDst register)	00	0 (not jumping)	(addi is signed addition	0 (not jumping)	0 (not jump returning)	Pass
addiu	addu	I	"001001"	"------"	0	0	1	00000000	00	0	1	00	0	1	0	0	Pass
addu	addu	R	"------"	"100001"	0	0	0	00000000	00	0	1	01	0	0	0	0	Pass
and	and	R	"------"	"100100"	0	0	0	00000011	00	0	1	01	0	0	0	0	Pass
andi	and	I	"001100"	"------"	0	0	1	00000011	00	0	1	00	0	1	0	0	Pass
lui	lui	I	"001111"	"------"	0	0	1	00010111	00	0	1	00	0	0	0	0	Pass
lw	add	I	"100011"	"------"	0	0	1	00010000	11	0	1	00	0	1	0	0	Pass
nor	nor	R	"------"	"101111"	0	0	0	00000110	00	0	1	01	0	0	0	0	Pass
xor	xor	R	"100110"	"------"	0	0	0	00000101	00	0	1	01	0	0	0	0	Pass
xori	xor	I	"001110"	"------"	0	0	1	00000101	00	0	1	00	0	1	0	0	Pass
or	or	R	"------"	"100101"	0	0	0	00000100	00	0	1	01	0	0	0	0	Pass
ori	or	I	"001101"	"------"	0	0	1	00000100	00	0	1	00	0	0	0	0	Pass
slt	slt	R	"------"	"101010"	0	0	0	00111000	00	0	1	01	0	1	0	0	Pass
slti	slti	I	"001010"	"------"	0	0	1	00111000	00	0	1	00	0	1	0	0	Pass
sll	sll	R	"------"	"000000"	0	1	0	00000010	00	0	1	01	0	1	0	0	Pass
srl	srl	R	"------"	"000010"	0	1	0	00000001	00	0	1	01	0	1	0	0	Pass
sra	sra	R	"------"	"000011"	0	1	0	00001001	00	0	1	01	0	1	0	0	Pass
sw	add	I	"101011"	"------"	0	0	1	00010000	00	1	0	00	0	1	0	0	Pass
sub	sub	R	"------"	"100010"	0	0	0	00011000	00	0	1	01	0	0	0	0	Pass
subu	subu	R	"------"	"100011"	0	0	0	00001000	00	0	1	01	0	0	0	0	Pass
beq	beq	I	"000100"	"------"	0	0	0	11001000	00	0	0	00	0	0	0	0	Pass
bne	bne	I	"000101"	"------"	0	0	0	01001000	00	0	0	00	0	1	0	0	Pass
j	add	J	"000010"	"------"	0	0	1	00010000	00	0	0	01	1	0	0	0	Pass
jal	add	J	"000011"	"------"	0	0	1	00010000	00	0	1	11	1	0	1	0	Pass
jr	add	R	"------"	"001000"	0	1	0	00010000	00	0	0	01	0	0	0	1	Pass
lb	add	I	"100000"	"------"	0	0	1	00010000	01	0	1	00	0	1	0	0	Pass
lh	add	I	"100001"	"------"	0	0	1	00010000	10	0	1	00	0	0	0	0	Pass
lbu	add	I	"100100"	"------"	0	0	1	00010000	01	0	1	00	0	0	0	0	Pass
lhu	add	I	"100101"	"------"	0	0	1	00010000	10	0	1	00	0	0	0	0	Pass
sllv	sll	R	"------"	"000100"	0	0	0	00000010	00	0	1	01	0	0	0	0	Pass
srlv	srl	R	"------"	"000110"	0	0	0	00000001	00	0	1	01	0	0	0	0	Pass
srav	sra	R	"------"	"000111"	0	0	0	00001001	00	0	1	01	0	0	0	0	Pass
halt	sra	R	"------"	"010100"	1	0	0	00001001	00	0	0	01	0	0	0	0	Pass

ALU Control Signals	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	BEQ	Branch	SLT Bit	Signed/Ovflw, or not.	Shift Logical Or Arith	MUX Selector	MUX Selector	MUX Selector

RegDst	[00]	[01]	[10]	[11]
Mux for Rt, rd, or $31	rt	rd	Hardcoded reg 31	Hardcoded reg 31

MemToReg	[00]	[01]	[10]	[11]
Mux for Lw, lhu,lb,lbu	ALU Return	Lb,Lbu	Lh, Lhu	Memory Return

[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_controller/w_instruct31_26	6'h00
/tb_controller/w_instruct5_0	6'h26
/tb_controller/w_beq	0
/tb_controller/w_halt	0
/tb_controller/w_STD_SHIFT	0
/tb_controller/w_ALU_Ctl	6'h05
/tb_controller/w_RegWrite	1
/tb_controller/w_MemtoReg	2'h0
/tb_controller/w_MemWrite	0
/tb_controller/w_RegDst	2'h1
/tb_controller/w_Branch	0
/tb_controller/w_Alu_Src	0
/tb_controller/w_ext_ctl	0
/tb_controller/w_Jump	0
/tb_controller/w_jr	0
/tb_controller/w_jal	0

00 08 09 0C 0F 23 0E 0D 0A 2B 04 05 20 21 24 25 02 03
20 21 24 27 26 25 2A 00 02 03 04 06 07 22 23 08 2B
10 00 03 06 05 04 28 02 01 09 02 01 09 18 08 00 10 00 03 07 10 05 04 28 00 18 00 10
0 3 0 1 2 1 2 0
1 0 3

For this part to verify everything was working, I just simply set up a test bench to push every single instruction code, in the image you can see the first half using changing in instructions [5:0] or the funct field for R types. For he second half using changing in instructions corresponding to [31:26] or the opcode for I and J type instructions. I then went through and verified I was seeing the correct output on each line for our excel sheet.

[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.

TODO

[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?

TODO

[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use QuestaSim 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 QuestaSim waveforms in your writeup.

TODO

[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 logical (SRL) and arithmetic (SRA) right shifts lies in how the vacated positions are filled. In a logical shift right (SRL), the vacated positions are padded with zeros. Conversely, in an arithmetic shift right (SRA), the vacated positions are filled with the value of the most significant bit (MSB) to preserve the sign of the original number; this means if the MSB is 1 (indicating a negative number in two's complement), the shift operation pads with 1s. MIPS does not include a Shift Left Arithmetic (SLA) instruction primarily because arithmetic left shifts do not require sign bit management—left shifts by their nature multiply the number by two for each shift position, preserving the sign. Furthermore, adding an SLA instruction would complicate the instruction set without offering a significant benefit, as the logical shift left operation (SLL) already achieves the desired outcome without altering the number's sign.

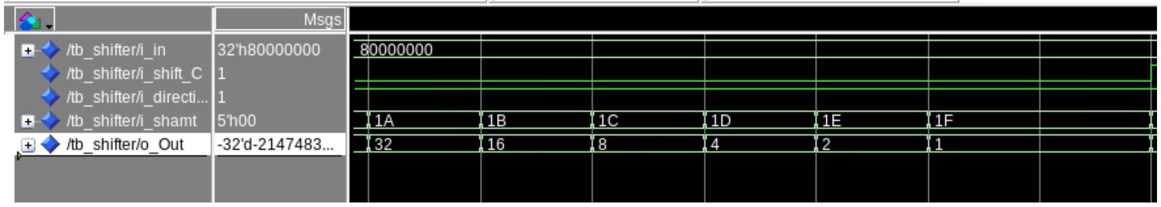
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

It utilizes a hierarchical structure of 2-to-1 multiplexers (muxes) organized into five stages, each stage doubling the shifting ability from 1-bit shifts up to 16-bit shifts, allowing for shift amounts ranging from 0 to 31 bits. The type of shift (logical or arithmetic) is determined by a shift type signal, and the amount of shift is controlled by a 5-bit shift amount input. This design efficiently achieves variable bit shifting by selecting the appropriate path through the muxes based on the shift amount, with the fill bit for arithmetic shifts dynamically determined by the input's most significant bit (MSB).

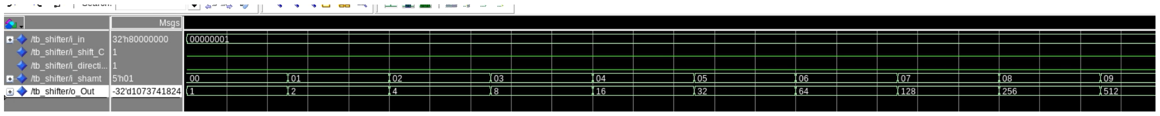
[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

To make a right barrel shifter work both ways, we add a simple trick: flipping the bits around before and after shifting. There's a new input called "shift direction" that lets us choose which way to shift. If we want to shift right, it just does its normal thing. But if we want to shift left, it first flips all the bits around, does a normal right shift, and then flips the bits back. This way, we end up with a shifter that can go both left and right.

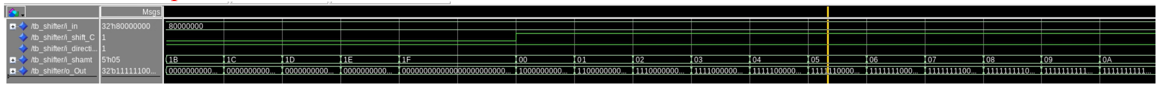
[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.


Msgs
/tb_shifter/i_in 32'h80000000
/tb_shifter/i_shift_C 1
/tb_shifter/i_directi... 1
/tb_shifter/i_shamt 5'h00
/tb_shifter/o_Out -32'd-2147483...
80000000
1A 1B 1C 1D 1E 1F
32 16 8 4 2 1

You can see here that when direction is 1(right) and the Control bit (logical), is set to 0 the the value decreases by dividing by 2 each time, until we get to 0. This is expected.


Msgs
/tb_shifter/i_in 32'h80000000
/tb_shifter/i_shift_C 1
/tb_shifter/i_directi... 1
/tb_shifter/i_shamt 5'h01
/tb_shifter/o_Out -32'd1073741824
00000001
00 01 02 03 04 05 06 07 08 09
1 2 4 8 16 32 64 128 256 512

Here you can see that when the direction is set to 0, and the shift type is set to 1(logical), a normal bit shift occures, shifting the bit to the left by 1 place, effectivly doubling the value at every shift, the opposite of the above example.



In this example you can the right shift at work with the control bit set to 1 for arithmetic, in this setting the shifter adds a bit to the msb slot and removes one from the lsb slot each time. When adding bits, because a 1 is set in the msb at start, the shifter adds a 1 bit every time.

[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.

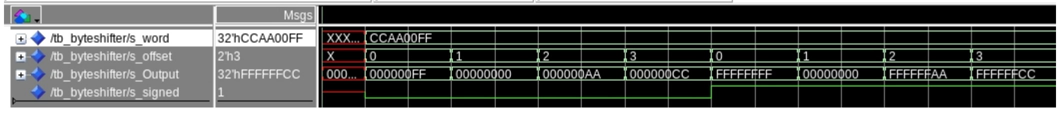
The Design approach we took was to first figure out what all the major things we needed to finish the design were. We then broke them down into smaller parts such as the ALU, control, Fetch logic, and register. Anything outside these were considered extra things needing to be implemented on the overall processor design.

We then go to work making and testing the individual parts separately with testing, and building up to throwing it all together. Once thrown together we were able to start running some real test benches with the provided tests and verify what we had working and what we didn't have working. The following is a list of additional things we needed to implement.

Mux4t1_N, Mux8t1_N,Nor_N,Or_N,Xor_N,And_N, mux32t1, ByteShifter, and WordShifter … as well as I'm sure there's some missing.

One of the design choices we decided to make was implementing a ByteShifter, and a WordShifter, these both work similarly, but do different things. Because of once everything was assembled, and all the basic ALU components passed their test, as well as LW, we needed a way to be able to implement a Lb, lbu, lh, and lhu. After looking at the outputs it became clear the we had the right chunk of memory, but not the right piece we were looking for, and because we knew that out CPU was word addressable there must be another way, outside of the ALU and DMEM to get the desired output. So that's why we came up with these decoders. These decodes sit just between the mux used for Mux2reg(mux4 in our design) and the DMEM. These decoders then, in parallel hook up to the dmem and the alu output, and then into the mux4t1_N we implemented. Then depending on a ctl signal from the controller, pics between the decoders or the regular outputs etc. if Byte decoder is selected, it pulls off the desired byte, with offset. Then pads depending on the MSB and an ext type control signal. The word decoder works the same, but instead of having 4 offset possibilities, it only has 2. It also relies on bit (1) of the alu out to determine the shift amount, while the byte decoder relies on bits [1:0]

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

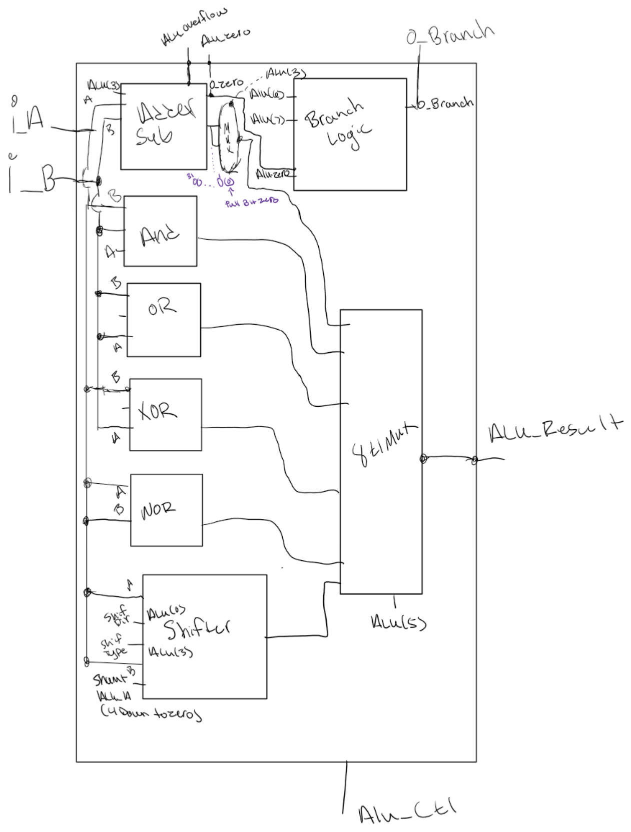


```
                              Msgs
/tb_byteshifter/s_word        32'hCCAA00FF    XXX...  CCAA00FF
/tb_byteshifter/s_offset      2'h3            X       0         1         2         3         0         1         2         3
/tb_byteshifter/s_Output      32'hFFFFFFCC    000...  000000FF  00000000  000000AA  000000CC  FFFFFFFF  00000000  FFFFFFAA  FFFFFFCC
/tb_byteshifter/s_signed      1
```

Showing the ByteShifter Above, and the test bench that was designed for it. The shifter works by having a decoder as the base, that is hooked up to an offset input. The offset input comes in, is decoded into an offset of 5 bits, then feeds into a left shifter depending on the decoder input. The shifter then outputs the fully left shifted data, and then a hardcoded shifter shifts the bits 24 times to the right. To determine sign, the 32nd bit is stripped in between the first output and the second shifter input, and is read at an and gate. If the second input to the and gate is a 1(meaning we want a signed extension) it will propagate the 1 bit to the second shifter if, and only if that 31st bit from the first output was a 1. The Word shifter, not show is the same, except with only two possible offsets, and 16 bit shifts only.

You can see in the testbench when the sign bit is 1, and the 32nd bit is 1, it sign extends, and you can also see, depending on the offset, 0,1,2,3 it will grab different bytes of the input data, 00 = byte 1, 01 = byte 2 etc. the shift amount is determined on the byte shifter form the alu out bits [1:0] and for the word shifter, it uses bit (1) only

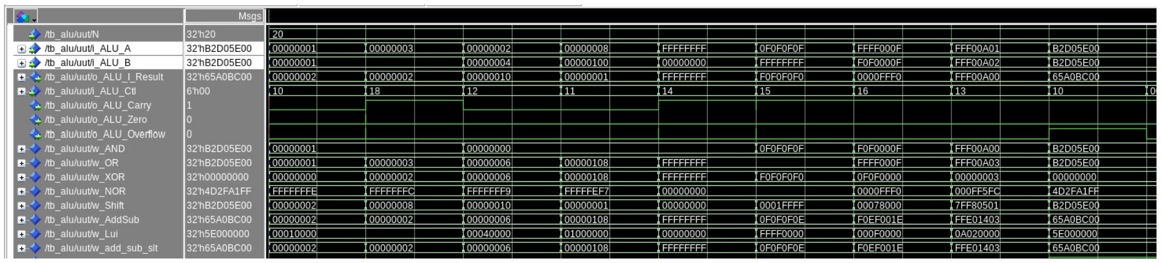
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is `slt` implemented?



Overflow – Overflow is calculated in the adder/sub component in the adder specifically. It it calculated by xoring the N bit and the N-1 bit, or the msb bits and the msb bit-1. The carry out of the msb bit must be equal to the carry in or its considered an overflow, and the msb has changed when it should not have, otherwise, we have surpassed the most representable number.

Zero – overflow is calculated in the adder subtractor part of the ALU, and a flag is thrown when the output is equal to 0 from here. Otherwise the output is 0 on the flag**.**

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



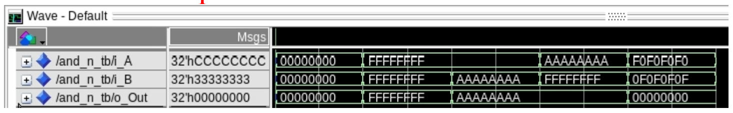
```
                             Msgs
/tb_alu/uut/N                32'h20        20
/tb_alu/uut/i_ALU_A          32'hB2D05E00  00000001  00000003  00000002  00000008  FFFFFFFF  0F0F0F0F  FFFF000F  FFF00A01  B2D05E00
/tb_alu/uut/i_ALU_B          32'hB2D05E00  00000001            00000004  00000100  00000000  FFFFFFFF  F0F0000F  FFF00A02  B2D05E00
/tb_alu/uut/o_ALU_I_Result   32'h65A0BC00  00000002  00000002  00000010  00000001  FFFFFFFF  F0F0F0F0  0000FFF0  FFF00A00  65A0BC00
/tb_alu/uut/i_ALU_Ctl        6'h00         10        18        12        11        14        15        16        13        10        0
/tb_alu/uut/o_ALU_Carry      1
/tb_alu/uut/o_ALU_Zero       0
/tb_alu/uut/o_ALU_Overflow   0
/tb_alu/uut/w_AND            32'hB2D05E00  00000001            00000000                      0F0F0F0F  F0F0000F  FFF00A00  B2D05E00
/tb_alu/uut/w_OR             32'hB2D05E00  00000001  00000003  00000006  00000108  FFFFFFFF            FFFF000F  FFF00A03  B2D05E00
/tb_alu/uut/w_XOR            32'h00000000  00000000  00000002  00000006  00000108  FFFFFFFF  F0F0F0F0  0F0F0000  00000003  00000000
/tb_alu/uut/w_NOR            32'h4D2FA1FF  FFFFFFFE  FFFFFFFC  FFFFFFF9  FFFFFEF7  00000000            0000FFF0  000FF5FC  4D2FA1FF
/tb_alu/uut/w_Shift          32'hB2D05E00  00000002  00000008  00000010  00000001  00000000  0001FFFF  00078000  7FF80501  B2D05E00
/tb_alu/uut/w_AddSub         32'h65A0BC00  00000002  00000002  00000006  00000108  FFFFFFFF  0F0F0F0E  F0EF001E  FFE01403  65A0BC00
/tb_alu/uut/w_Lui            32'h5E000000  00010000            00040000  01000000  00000000  FFFF0000  000F0000  0A020000  5E000000
/tb_alu/uut/w_add_sub_slt    32'h65A0BC00  00000002  00000002  00000006  00000108  FFFFFFFF  0F0F0F0E  F0EF001E  FFE01403  65A0BC00
```

To keep things simple, I just did a simple test for everything put together. More extensive tests were done on individual parts, So all this was is a matter of throwing everything together with a mux selector and then watching the magic happen. You can see from left to Right, using ALU A, B, then result to see what came out. From left to right, each change of number, signals a change of operation. Add, subtract, bit shift B by A left, B by A right, OR, XOR, NOR,AND, AND, Add with overflow, and ADDU without overflow with the same numbers. You can see they are controlled by the ALU_CTL bits, which for the most part just handle the switching of the mux. So everyhting is used every cycle, however, only what we want leaves the ALU. You can also see here that the Zero is never triggered, and the ALU overflow is only triggered when the regular add is done and not the addu.

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

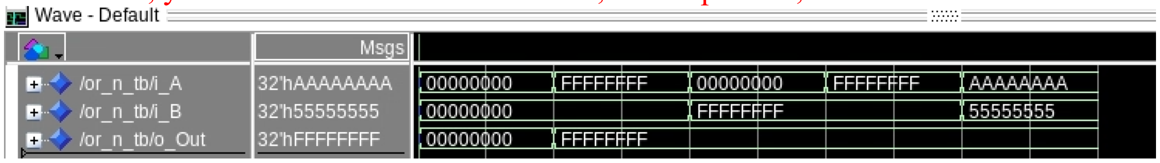
Please let me know if we need to make another test bench for the ALU, but for this, we are just showing you again, what we had done earlier. The reason no additional Testbench for the ALU were done was because additional Testing on all the individual components added were done. The Adders had been tested for overflow, and the adder subtractor has been tested and confirmed to work with the zero flag. Because the adder subtractor uses addition to subtract, and the internal components to get to the adder overflow had been tested quite well, with edge cases prior to this lab. So, we had come to the conclusion that the adder and subtractor were working good, and no unexpected behavior has or had arisen during this lab. The following are test cases from the AND gate, or gate, nor gate, and xor gate, mux8t1, and the shifter has already been mentioned above so I Will leave that out. All testbenches have been included in the report in a file called ALU, there are additional sub sections for the test benches to be tried and ran if needed. Let me know if you'd like us to resubmit with a more comprehensive test bench, however, because of the time of writing this, we have already finished all the provided tests, I don't see the need to.

AND here is an and gate at working being test on every bit, both bits from A and B must be 1 for the output to be a 0.



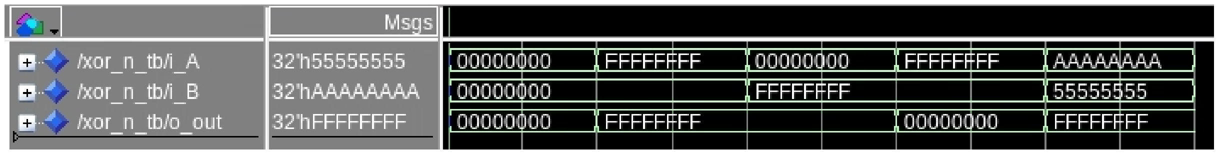
```
Wave - Default
                  Msgs
/and_n_tb/i_A     32'hCCCCCCCC  00000000  FFFFFFFF            AAAAAAAA  F0F0F0F0
/and_n_tb/i_B     32'h33333333  00000000  FFFFFFFF  AAAAAAAA  FFFFFFFF  0F0F0F0F
/and_n_tb/o_Out   32'h00000000  00000000  FFFFFFFF  AAAAAAAA            00000000
```

OR here you can see the test benches for the OR gate being tested on every bit, for correctness, you can see if 1 or both bits are 1, the output is 1, else 0.



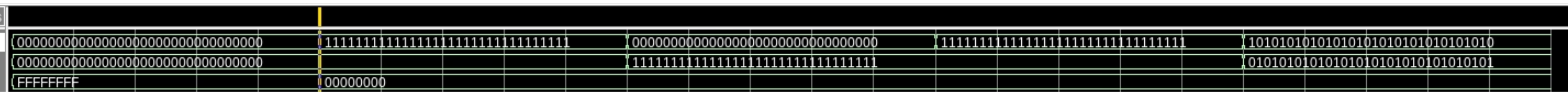
```
Wave - Default
                    Msgs
/or_n_tb/i_A     32'hAAAAAAAA   00000000  FFFFFFFF  00000000  FFFFFFFF  AAAAAAAA
/or_n_tb/i_B     32'h55555555   00000000            FFFFFFFF            55555555
/or_n_tb/o_Out   32'hFFFFFFFF   00000000  FFFFFFFF
```

XOR – here you can see a basic xor gate at work, one or the other but not both will trigger an output of 1.



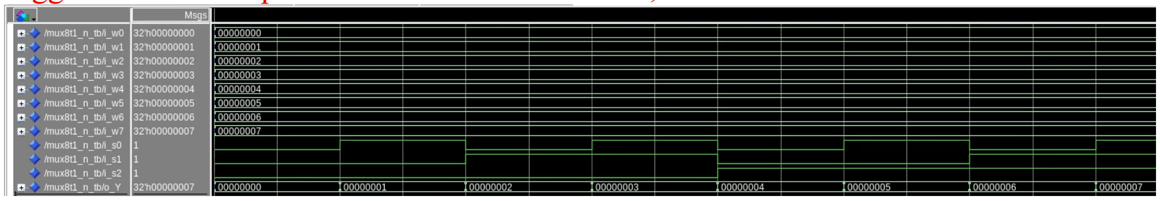
```
                    Msgs
/xor_n_tb/i_A    32'h55555555   00000000  FFFFFFFF  00000000  FFFFFFFF  AAAAAAAA
/xor_n_tb/i_B    32'hAAAAAAAA   00000000            FFFFFFFF            55555555
/xor_n_tb/o_out  32'hFFFFFFFF   00000000  FFFFFFFF            00000000  FFFFFFFF
```

NOR - You can see here, line 1 = A, line 2 = input b, line 3 = output, only a 0, 0 will trigger a 1 output.



```
00000000000000000000000000000000  11111111111111111111111111111111  00000000000000000000000000000000  11111111111111111111111111111111  10101010101010101010101010101010
00000000000000000000000000000000                                    11111111111111111111111111111111                                    01010101010101010101010101010101
FFFFFFFF                          00000000
```

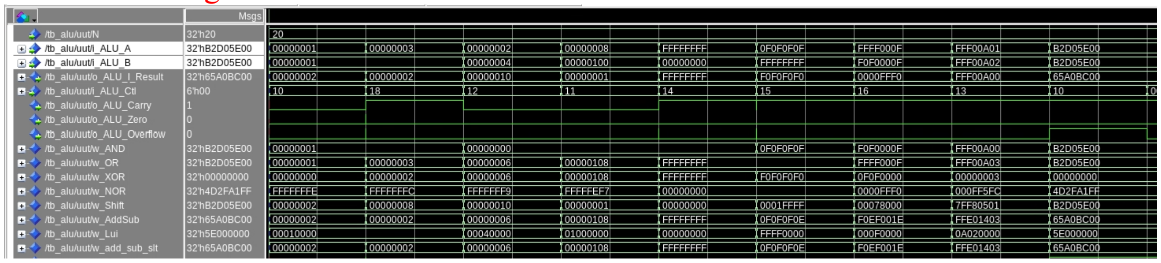
Mux8t1_N – Here you can see the 8 to 1 being tested for 32 bits, each select line is triggeredand the output can be seen below. 0 = w0, 1=w1….



```
                        Msgs
/mux8t1_n_tb/i_w0    32'h00000000   00000000
/mux8t1_n_tb/i_w1    32'h00000001   00000001
/mux8t1_n_tb/i_w2    32'h00000002   00000002
/mux8t1_n_tb/i_w3    32'h00000003   00000003
/mux8t1_n_tb/i_w4    32'h00000004   00000004
/mux8t1_n_tb/i_w5    32'h00000005   00000005
/mux8t1_n_tb/i_w6    32'h00000006   00000006
/mux8t1_n_tb/i_w7    32'h00000007   00000007
/mux8t1_n_tb/i_s0    1
/mux8t1_n_tb/i_s1    1
/mux8t1_n_tb/i_s2    1
/mux8t1_n_tb/o_Y     32'h00000007   00000000  00000001  00000002  00000003  00000004  00000005  00000006  00000007
```

SHIFTER – Testbench can be seen above, in earlier section of report.

ALU- Below is the test bench for the ALU, as I mentioned above, because of our design approach to test everyhting individually, and because of the simple construction of our ALU, and armed with the knowneledge of more gueling test benches coming up once everything is assembled we choose to wait to test it more exetesivly, as It would take a lot of time, and the basic component testing has already been done quite well, so, since evrything was able to be seen clearly switching between eachother, we stopped here with the ALU Testing.



```
                            Msgs
/tb_alu/uut/N               32'h20         20
/tb_alu/uut/i_ALU_A         32'hB2D05E00   00000001  00000003  00000002  00000008  FFFFFFFF  0F0F0F0F  FFFF000F  FFF00A01  B2D05E00
/tb_alu/uut/i_ALU_B         32'hB2D05E00   00000001            00000004  00000100  00000000  FFFFFFFF  F0F0000F  FFF00A02  B2D05E00
/tb_alu/uut/o_ALU_I_Result  32'h65A0BC00   00000002  00000002  00000010  00000001  FFFFFFFF  F0F0F0F0  0000FFF0  FFF00A00  65A0BC00
/tb_alu/uut/i_ALU_Ctl       6'h00          10        18        12        11        14        15        16        13        10        0
/tb_alu/uut/o_ALU_Carry     1
/tb_alu/uut/o_ALU_Zero      0
/tb_alu/uut/o_ALU_Overflow  0
/tb_alu/uut/w_AND           32'hB2D05E00   00000001            00000000                      0F0F0F0F  F0F0000F  FFF00A00  B2D05E00
/tb_alu/uut/w_OR            32'hB2D05E00   00000001  00000003  00000006  00000108  FFFFFFFF            FFFF000F  FFF00A03  B2D05E00
/tb_alu/uut/w_XOR           32'h00000000   00000000  00000002  00000006  00000108  FFFFFFFF  F0F0F0F0  0F0F0000  00000003  00000000
/tb_alu/uut/w_NOR           32'h4D2FA1FF   FFFFFFFE  FFFFFFFC  FFFFFFF9  FFFFFEF7  00000000            0000FFF0  000FF5FC  4D2FA1FF
/tb_alu/uut/w_Shift         32'hB2D05E00   00000002  00000008  00000010  00000001  00000000  0001FFFF  00078000  7FF80501  B2D05E00
/tb_alu/uut/w_AddSub        32'h65A0BC00   00000002  00000002  00000006  00000108  FFFFFFFF  0F0F0F0E  F0EF001E  FFE01403  65A0BC00
/tb_alu/uut/w_Lui           32'h5E000000   00010000            00040000  01000000  00000000  FFFF0000  000F0000  0A020000  5E000000
/tb_alu/uut/w_add_sub_slt   32'h65A0BC00   00000002  00000002  00000006  00000108  FFFFFFFF  0F0F0F0E  F0EF001E  FFE01403  65A0BC00
```

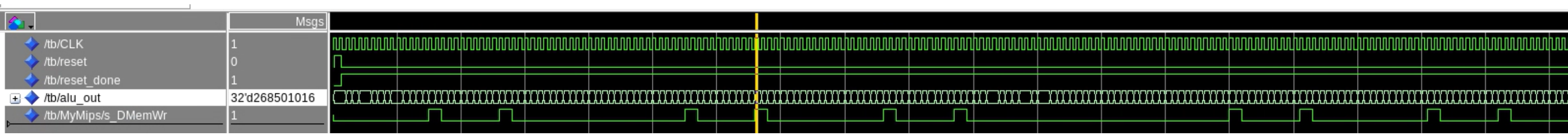
[Part 3] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[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.

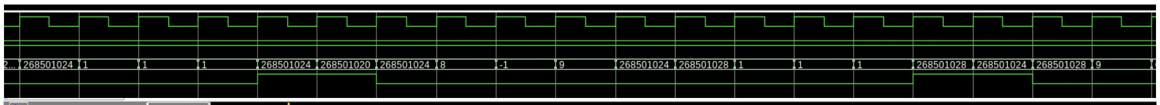
TODO:

[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.
TODO:

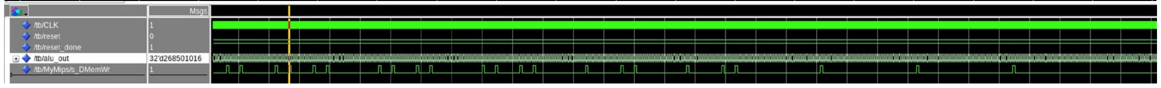
[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.



	Msgs
/tb/CLK	1
/tb/reset	0
/tb/reset_done	1
/tb/alu_out	32'd268501016
/tb/MyMips/s_DMemWr	1



2... 268501024 1 1 1 268501024 268501020 268501024 8 -1 9 268501024 268501028 1 1 1 268501028 268501024 268501028 9



	Msgs
/tb/CLK	1
/tb/reset	0
/tb/reset_done	1
/tb/alu_out	32'd268501016
/tb/MyMips/s_DMemWr	1

While it was kind of hard to show this in an easy to understand way while describing it in terms of wave forms, you can see that every time the DmemWr is enabled, we give 2 memory addresses, each 4 apart. This is done during the Swap part of the code, this indicates that a swap of two numbers was needed and we wrote the swap to memory, the new address. Then, as we get further and further down the line of swaps, the need to swap becomes less and less, as seen above until it sends with no swaps. This was further confirmed to have worked correctly as it was given a pass by running a test, and successfully complete.

[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 schematic from part 1. What components would you focus on to improve the frequency?

Might be wrong, just a quick guess, can update in final report.

Total (ns) Incr (ns) Type Element=====================================

20.000 20.000 latch edge time

23.399 3.399 R clock network delay

23.407 0.008 clock pessimism removed

23.387 -0.020 clock uncertainty 23.405 0.018 uTsu

reg:regist|dffg_N:\G_dffg_Nbit:23:dffg_i|dffg:\Nbit_dffg:22:DFFGG|s_Q Data Arrival Time : 43.823 Data Required Time : 23.405 Slack : -20.418 (VIOLATED)

Maximum Frequency=Data Required Time1/ Data Required Time=23.405ns=23.405×1

Data Required Time=23.405 ns=23.405×10−9 s0−9s

Maximum Frequency=1/23.405×10−9 HzMaximum Frequency=23.405×10−91Hz

42.73 MHZ?