CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

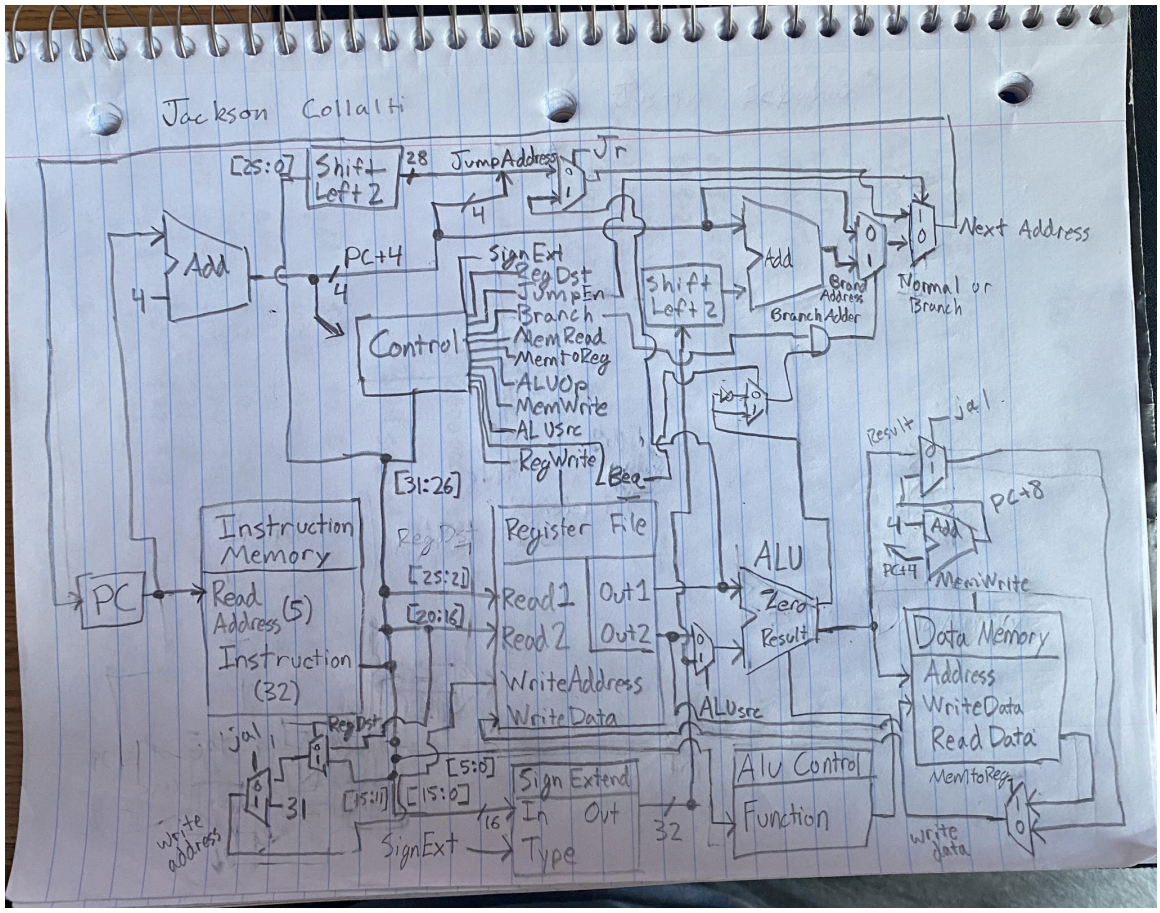
Team Members: Jackson Collalti

Justin Sebahar

Project Teams Group #: SecD_04

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.



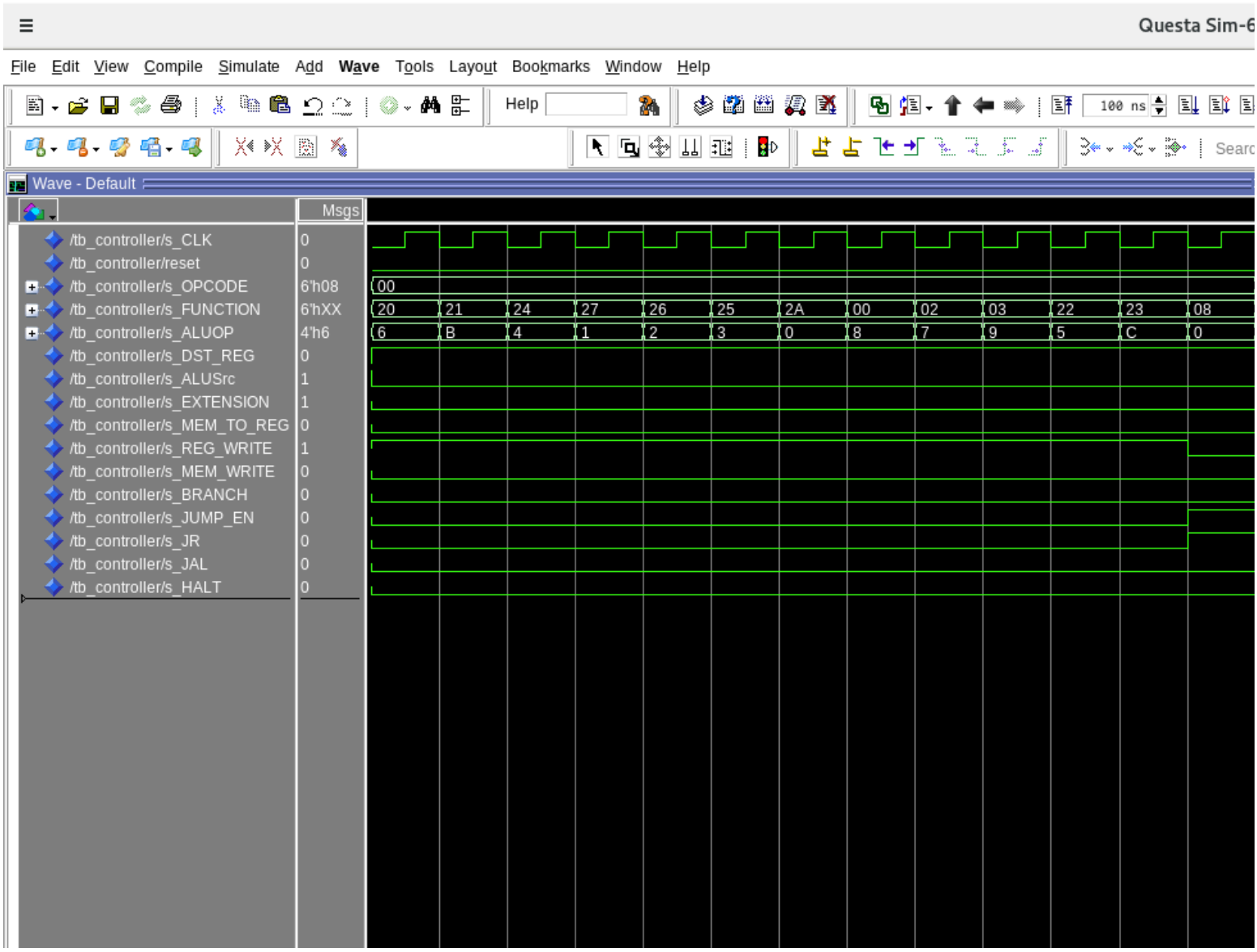
[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your 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/fui	ALUOp	RegDST	ALUSrc	Extension	MemToReg	RegWrite	MemWrite	Branch	BranchType	JumpEN	jr	jal	halt
R-type														
add	000000...10000	6	1	0	0	0	1	0	0	0	0	0	0	0
addu	000000...10000	11	1	0	0	0	1	0	0	0	0	0	0	0
and	000000...10010	4	1	0	0	0	1	0	0	0	0	0	0	0
nor	000000...10011	1	1	0	0	0	1	0	0	0	0	0	0	0
xor	000000...10011	2	1	0	0	0	1	0	0	0	0	0	0	0
or	000000...10010	3	1	0	0	0	1	0	0	0	0	0	0	0
slt	000000...10101	0	1	0	0	0	1	0	0	0	0	0	0	0
sll	000000...00000	8	1	0	0	0	1	0	0	0	0	0	0	0
srl	000000...00001	7	1	0	0	0	1	0	0	0	0	0	0	0
sra	000000...00011	9	1	0	0	0	1	0	0	0	0	0	0	0
sub	000000...10001	5	1	0	0	0	1	0	0	0	0	0	0	0
subu	000000...10001	12	1	0	0	0	1	0	0	0	0	0	0	0
jr	000000...00100	0	1	0	0	0	0	0	0	0	1	1	0	0
I-type														
addi	...001000	6	0	1	1	0	1	0	0	0	0	0	0	0
addiu	...001001	11	0	1	1	0	1	0	0	0	0	0	0	0
andi	...001100	4	0	1	0	0	1	0	0	0	0	0	0	0
lui	...001111	10	0	1	0	0	1	0	0	0	0	0	0	0
lw	...100011	6	0	1	1	1	1	0	0	0	0	0	0	0
xori	...001110	2	0	1	0	0	1	0	0	0	0	0	0	0
ori	...001101	3	0	1	0	0	1	0	0	0	0	0	0	0
slti	...001010	0	0	1	1	0	1	0	0	0	0	0	0	0
sw	...101011	6	0	1	1	0	0	1	0	0	0	0	0	0
beq	...000100	12	0	0	1	0	0	0	1	1	0	0	0	0
bne	...000101	12	0	0	1	0	0	0	1	0	0	0	0	0
J-type														
j	...000010	0	0	0	0	0	0	0	0	0	1	0	0	0
jal	...000011	0	0	0	0	0	1	0	0	0	1	0	1	0
Other														
halt	...010100	0	0	0	0	0	0	0	0	0	0	0	0	1

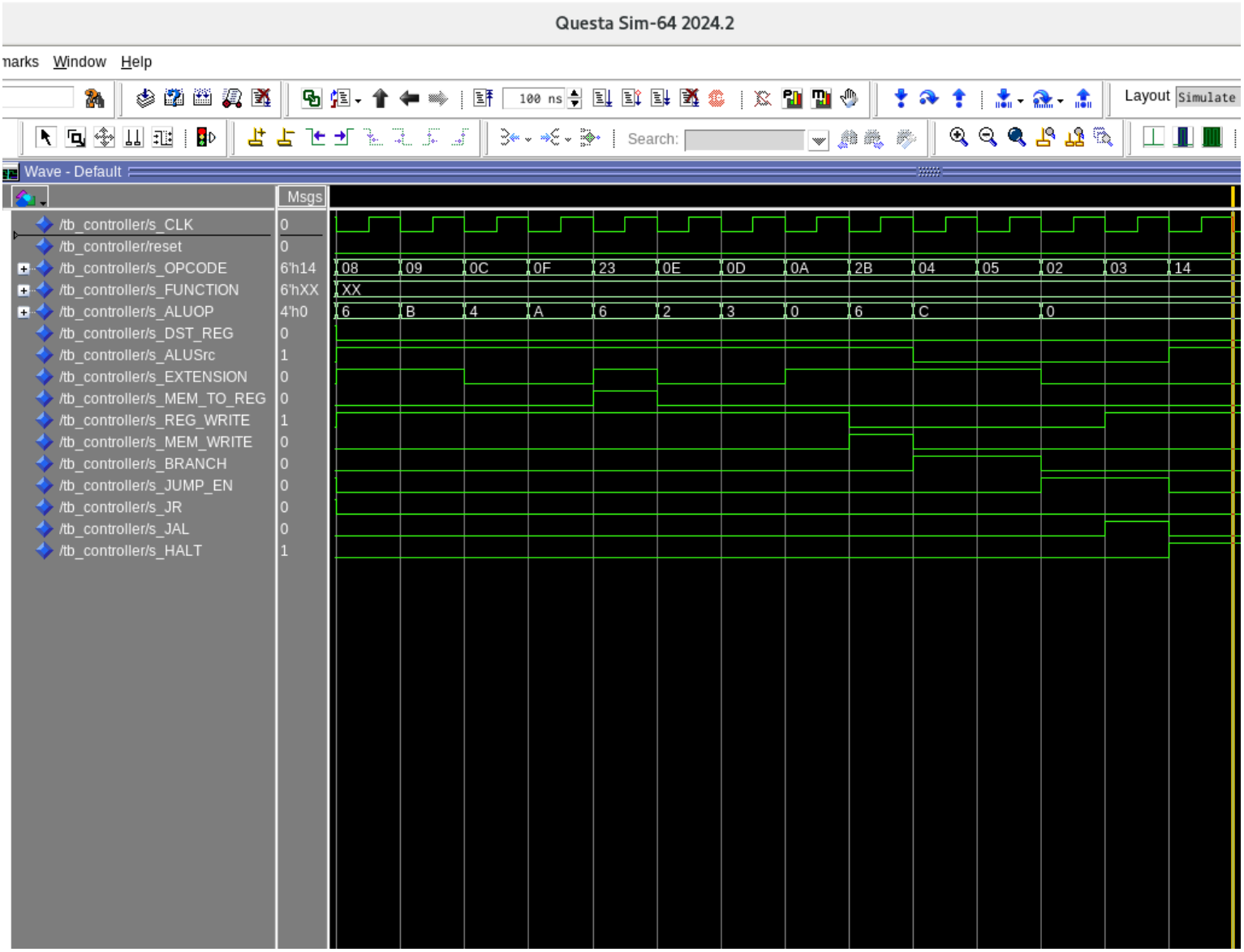
ALUOP	Function
load upper imm	10
shift righ arithm	9
shift left logical	8
shift right logica	7
add	6
subtract	5
and	4
or	3
xor	2
nor	1
slt	0

[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).

This wave demonstrates the control outputs for each R-type instruction. The wave holds true to the expected values in the spreadsheet.



This wave demonstrates the control outputs for each I-type and J-type instruction.



[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.

Jumps

- Standard Jumps (j)
- Jump Register (jr)
- Jump And Link (jal)

  *jal and jr both use register value addresses, so they can be implemented together

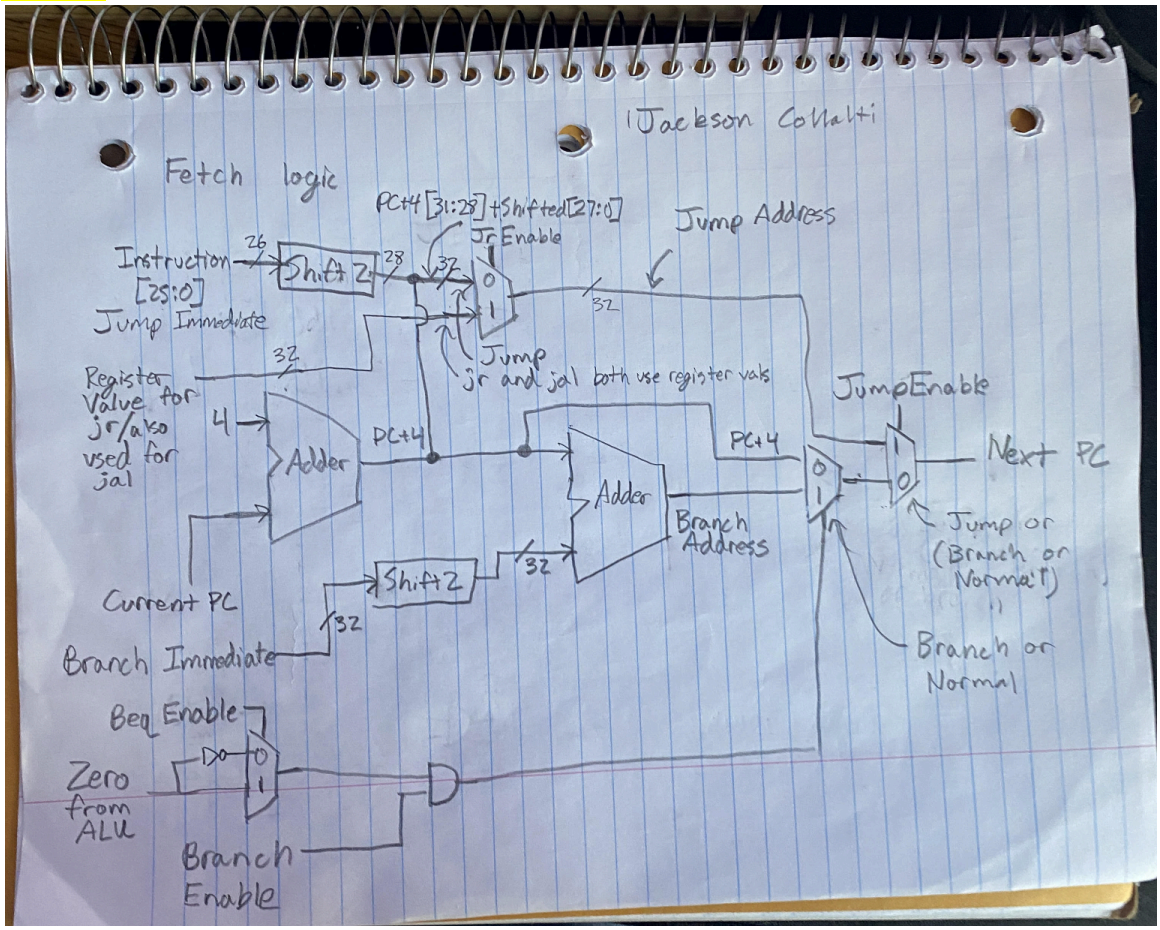
Branches

- Branch Equal (beq)
- Branch Not Equal (bne)

Normal

- PC + 4

[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?



Inputs:

Control Signals

- Branch Enable (1= branch instruction (beq or bne), 0 = else)
- Beq Enable (0 = BNE instruction, 1= BEQ instruction, and other = don't care)
- Jump Enable (1= jump, jr, or jal, 0 = else)
- Jump Register Enable = JrEnable (1= jr or jal instruction, 0 = else)

Inputs from design

- Jump Immediate (Instruction [25:0])
- Current PC value
- Register value for jr and jal (R[rs])
- Branch Immediate (Sign extended immediate)

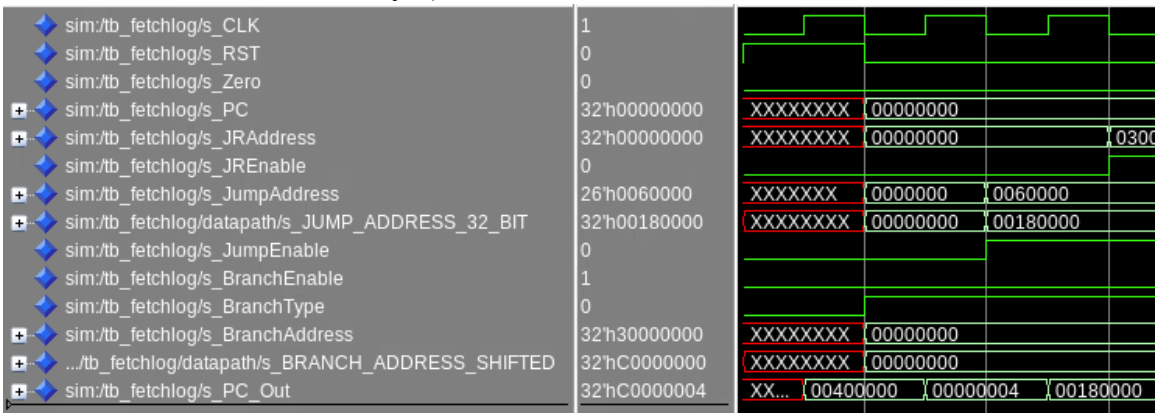
- Zero from ALU (1 = ALU inputs are equal, 0 = ALU inputs aren't equal)

Output

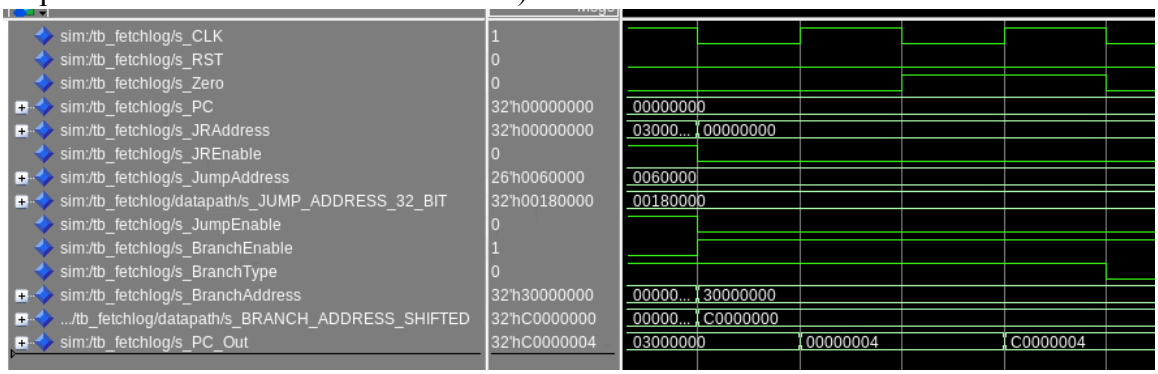
- Next PC value

[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.

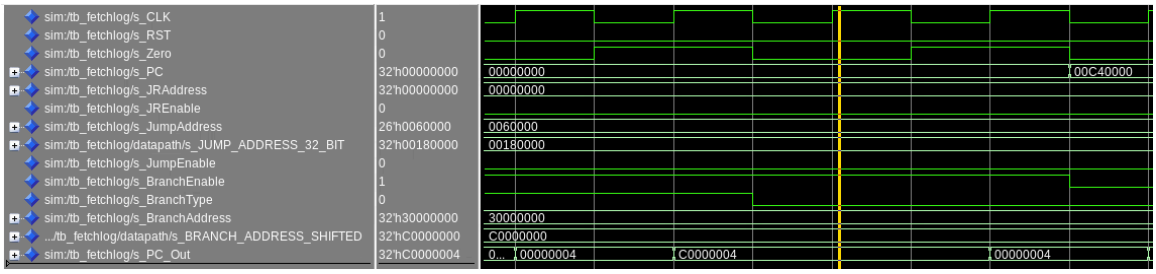
Cycle 1-Reset (resets to 0x00400000), Cycle 2-PC+4 (0x00000000 + 4), Cycle 3-Jump (with current address of 0x00000000 with immediate of 0x0600000 goes to 4*0x0600000 because immediate is shifted by 2)



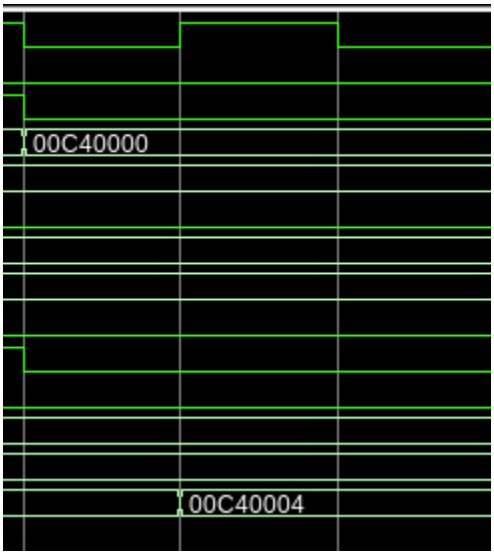
Cycle 4- Jump Register (Register Address - 0x03000000), Cycle 5-BEQ where zero isn't true (since branch condition isn't met, it does PC+4 in this case, previous PC = 0x00000000). Cycle 6 - is the same as Cycle 5, and only the branch condition is true (the output is branch address shifted then + 4)



Cycles 7 and 8 are the same as 5 and 6, only now testing BNE. Cycle 7 - BNE condition met (Zero != 1, the output is branch address shifted then + 4). Cycle 8-BNE where zero is true (since branch condition isn't met, it does PC+4 in this case, previous PC = 0x00000000)



PC+4 where PC != 0x00000000. Current PC = 0x00C40000 next PC = 0x00C40004



[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 that srl loads in 0's while sra loads in 1's or 0's, depending on the most significant bit (used for signed numbers). MIPS doesn't have sla because sll already does its job, and sla would be redundant. For sla and sll signed vs. unsigned doesn't matter since if the sign changes, you passed either the max positive or max negative value with the shift/multiplication, which would trigger an exception if implemented. Due to the only issue with sign-changing relating to an exception sll can be used instead of sla ultimately resulting in only sll needing to be implemented

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

For each row of the shifter, some number of the muxes will take in the "filler" value as the D1 input. This value determines what will replace all the values that have been shifted.
So, to implement this, I used a mux to determine what to feed into this filler value for the corresponding MUXs; if logical shift, choose 0, and if arithmetic shift, choose the 31st bit of the input vector.

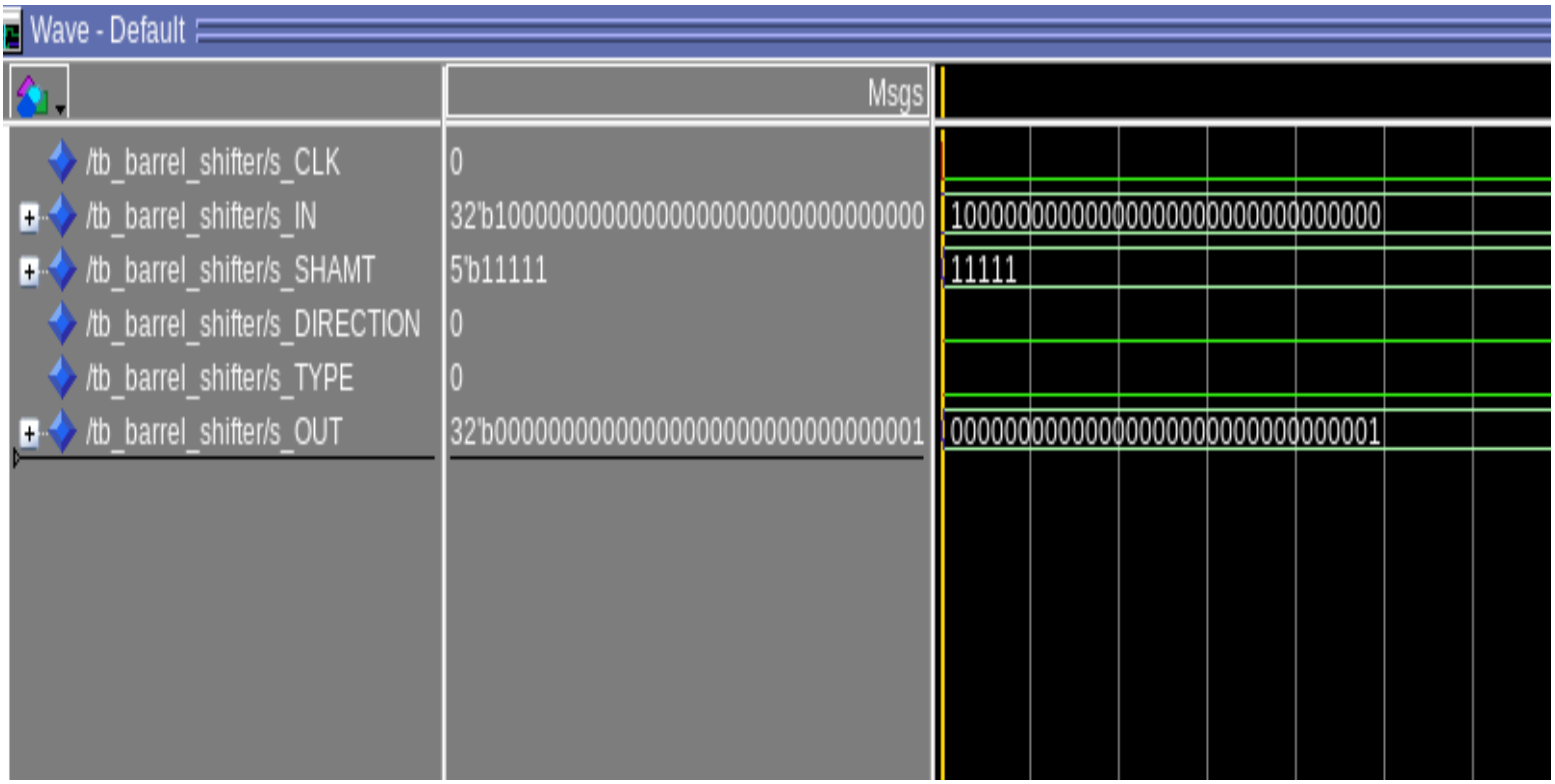
[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

To shift left, we can just reverse the input vector, shift it right, and then reverse the output again. To implement this, I create a signal that is a reversed version of the input (using a for loop), and then use a mux (given a control input "type") to decide if the normal input or reversed input should be processed through the shifter. Similarly, in the end, I reverse the output and save it as a separate signal, then feed the two through another MUX to determine the desired output.

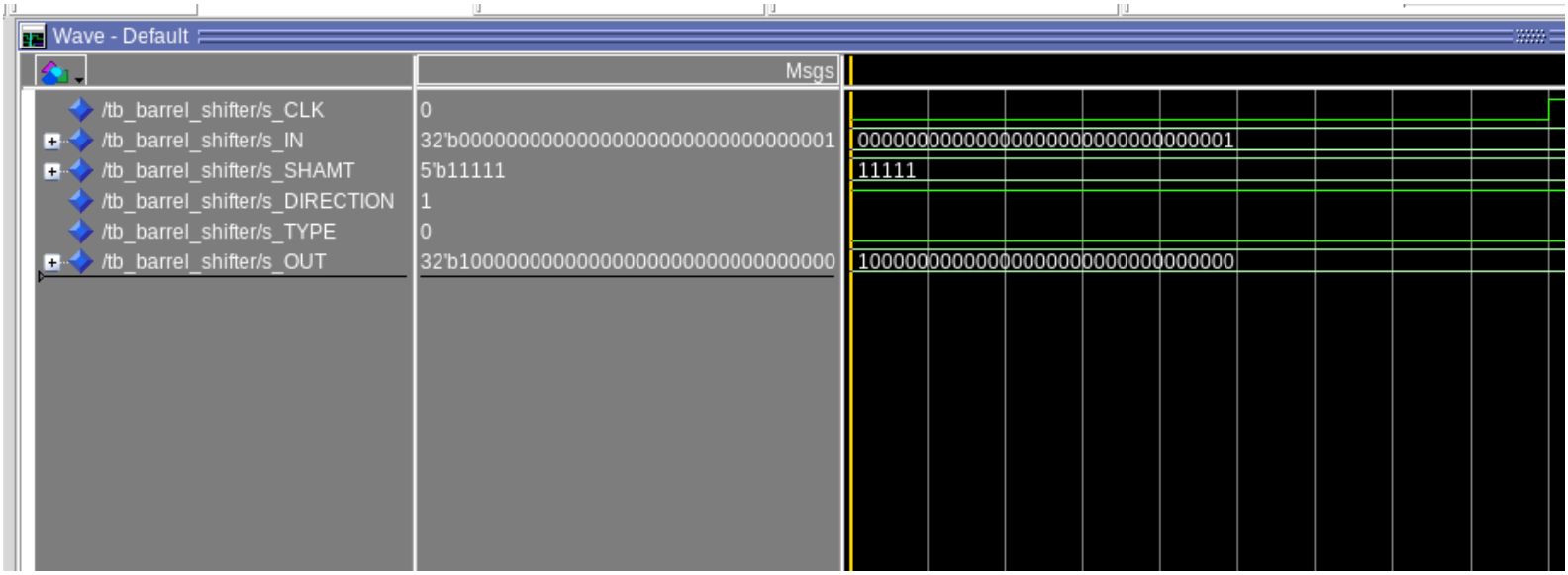
[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the Modelsim waveforms in your writeup.

For the following waves, note that these are only a select few of the test cases I executed. I can't show them all, but for demonstration purposes, I shift by the max shamt (31 bits).

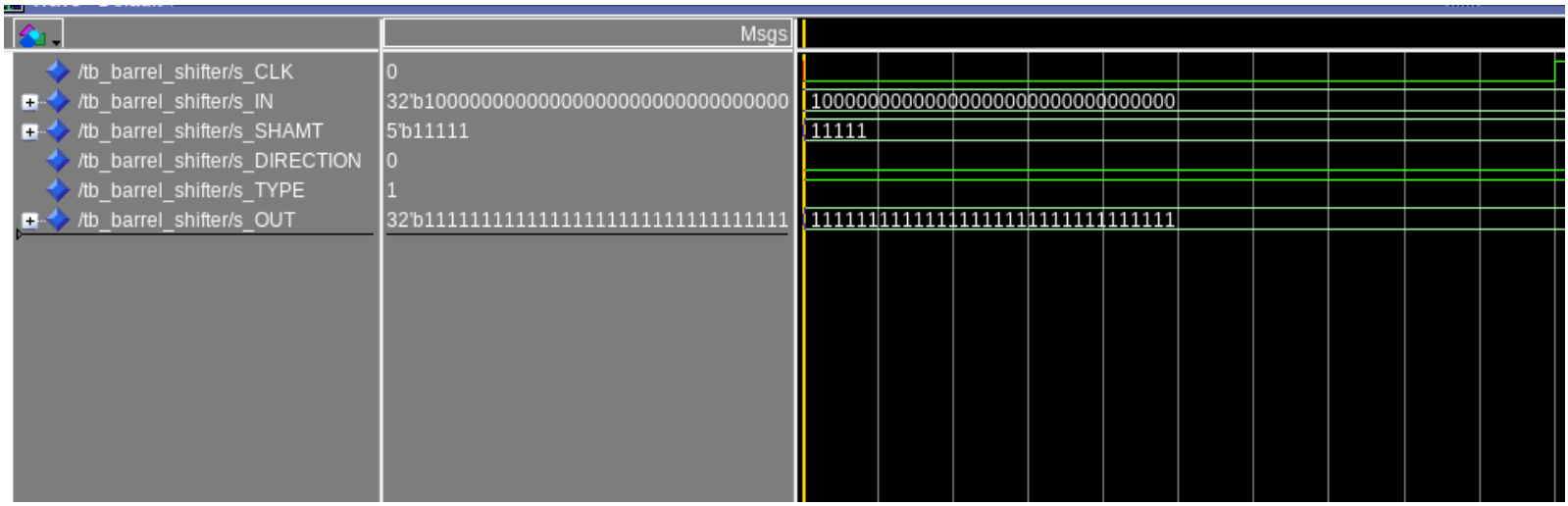
srl: We can see that the input, which is one in the 31st bit, is shifted right 31 bits so that now the 1 is in the 0th bit, and all other bits were replaced with 0.



sll: We can see in the wave that the input of 1 in the 0th bit is shifter left by 31 bits such that the 1 is now in the 31st bit. Everything else is replaced with 0.



sra: We can see that the input, which is one in the 31st bit, is shifted right 31 bits such that the 1 is now in the 0th bit. All other bits were replaced with 1.



[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.

Our design approach was made with simplicity in mind. We had all components run and do the calculations each clock cycle, which is probably not ideal for power efficiency or latency, but it allowed us to use a series of if statements to assign ALU out (i.e., ALUOP=6 would select the output of the adder as ALU_result). As a result of our design decision to have each component run and ALUOP select which output to use as the ALU_result, we had to have one component for each instruction inside of our ALU. This means we had 4 adder_subtractor units (One for normal add, One for unsigned add, One for normal sub, and finally one for unsigned sub). This also applied to shifts where we had a barrel shifter for sll, srl, lui, and sra. Then, we also had 32-bit logic gates for each logical operation. Finally, we had replicate, which was a created component that takes an 8-bit input and copies it 3 more times (repl(0x00000021) = 0x21212121). In total, we had 13 components plus assign statements for set less than instructions, which used the subtractor and also the 31st bit of input 1 and 2 to decide the output of set less than.

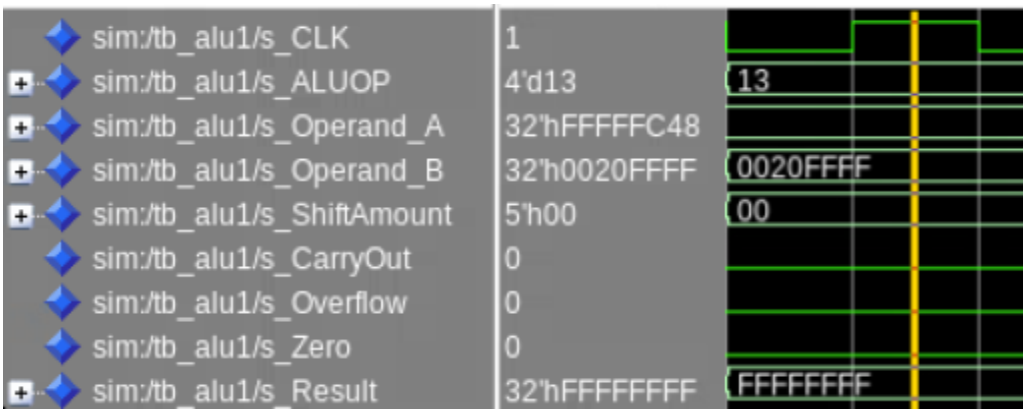
```
o_Result <= s_SET_LESS_THAN when (i_ALUOP = "0000") else
            s_NOR_OUT when (i_ALUOP = "0001") else
            s_XOR_OUT when (i_ALUOP = "0010") else
            s_OR_OUT when (i_ALUOP = "0011") else
            s_AND_OUT when (i_ALUOP = "0100") else
            s_SUB_OUT when (i_ALUOP = "0101") else
            s_SUM_OUT when (i_ALUOP = "0110") else
            s_BARRELRIGHT_OUT when (i_ALUOP = "0111") else
            s_BARRELLEFT_OUT when (i_ALUOP = "1000") else
            s_SRA_OUT when (i_ALUOP = "1001") else
            s_LUI_OUT when (i_ALUOP = "1010") else
            s_SUMU_OUT when (i_ALUOP = "1011") else
            s_SUBU_OUT when (i_ALUOP = "1100") else
            s_REPL_OUT when (i_ALUOP = "1101") else
            x"00000000";
```

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

Replicate:

```
REPL_IN: FOR i IN 7 DOWNTO 0 generate
    s_REPL_IN(i) <= i_Operand_B(i);
END generate REPL_IN;

--repl.qb
repl0: Repl
port MAP(
    i_A  => s_REPL_IN,
    o_F  => s_REPL_OUT
);
```

```
--Replicate, Expect 0xFFFFFFFF
s_Operand_A <= "11111111111111111111110001001000";
s_Operand_B <= "00000000001000001111111111111111";
s_ShiftAmount <= "00000";
s_ALUOP <= "1101";
WAIT FOR cCLK_PER;
```



[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 depends on the carryout bit xor'd with the last internal carry bit. Only detected when not using unsigned math

```
o_Overflow <= s_SUBOverflow when (i_ALUOP = "0101") else
              s_ADDOverflow when (i_ALUOP = "0110") else
             '0';
```

Zero- Is the result from subtracting input A and input B if A=B then the difference(A-B) =0

```
s_SET_EQUAL <= '1' when (s_SUB_OUT = x"00000000") else '0';

o_Zero <= s_SET_EQUAL;
```

Slt (A<B)- Four cases exist

Input A - Input B

Positive - Positive, slt is true when A minus B is negative

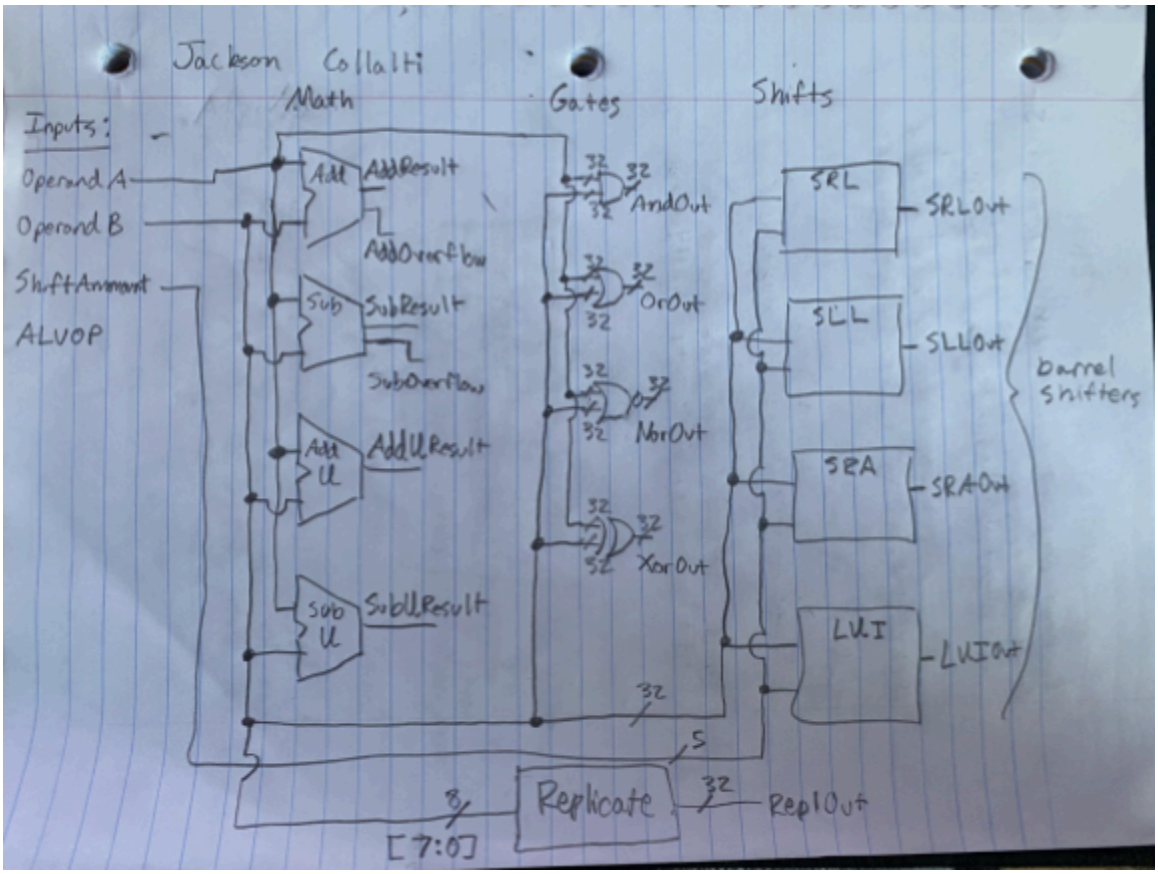
Positive - Negative, slt is false

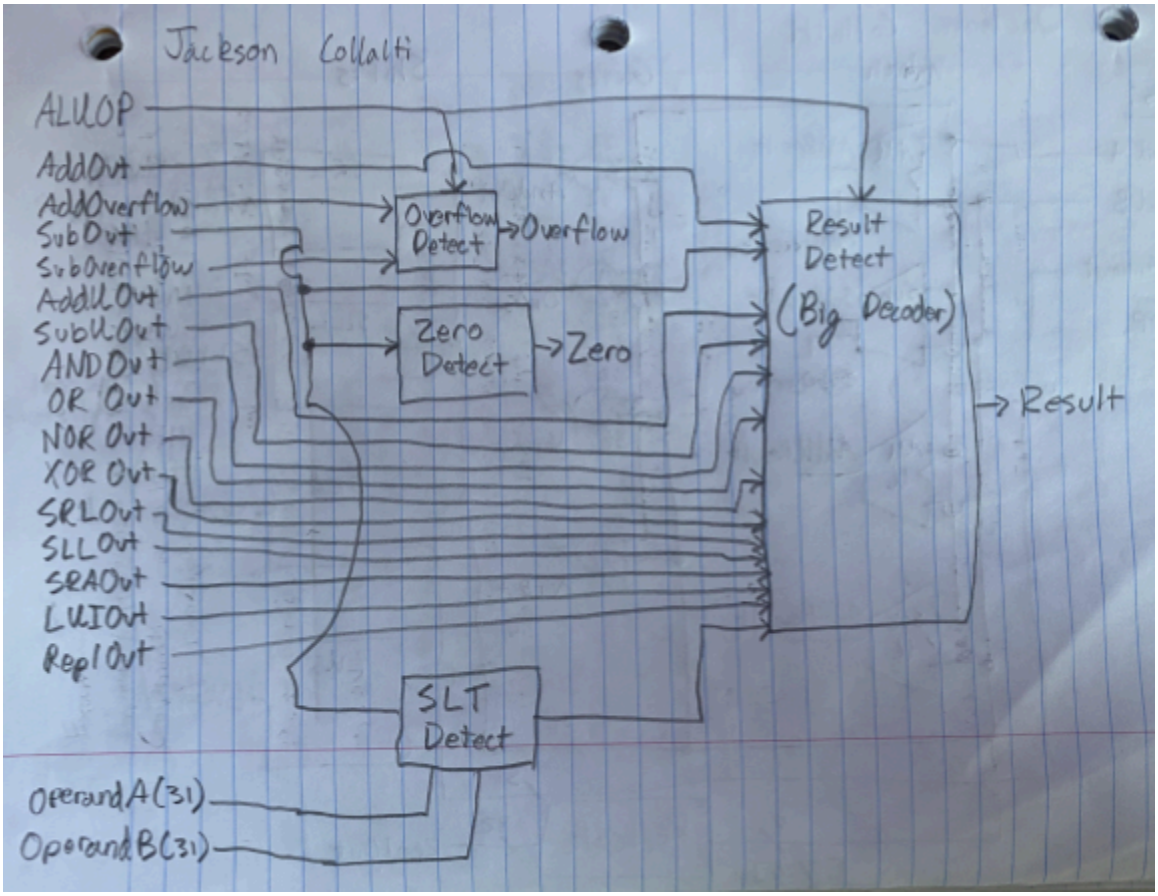
Negative - Positive, slt is true

Negative - Negative, slt is true when A minus B is negative

```
s_SET_LESS_THAN <= x"00000001" when (i_Operand_A(31) = '0' and i_Operand_B(31) = '0' and s_SUB_OUT > x"80000000") else
                   x"00000000" when (i_Operand_A(31) = '0' and i_Operand_B(31) = '1') else
                   x"00000001" when (i_Operand_A(31) = '1' and i_Operand_B(31) = '0') else
                   x"00000001" when (i_Operand_A(31) = '1' and i_Operand_B(31) = '1' and s_SUB_OUT > x"80000000") else
                   x"00000000";
```

We found it easier to do if cases instead of using a series of mux's, so the schematic will just reference these lines of code above, such as overflow detect, zero detect and SLT detect.

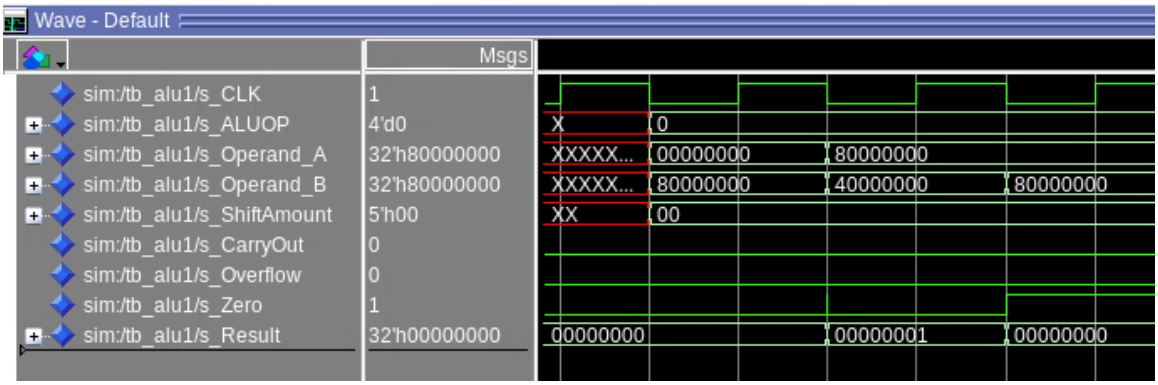

Jackson Collalti
Math
Gates
Shifts
Inputs:
Operand A
Operand B
ShiftAmount
ALUOP
Add
AddResult
AddOverflow
Sub
SubResult
SubOverflow
Add U
AddUResult
Sub U
SubUResult
32
AndOut
OrOut
NorOut
XorOut
SRL
SRLOut
SLL
SLLOut
SRA
SRAOut
LUI
LUIOut
Barrel Shifters
5
Replicate
ReplOut
8
[7:0]



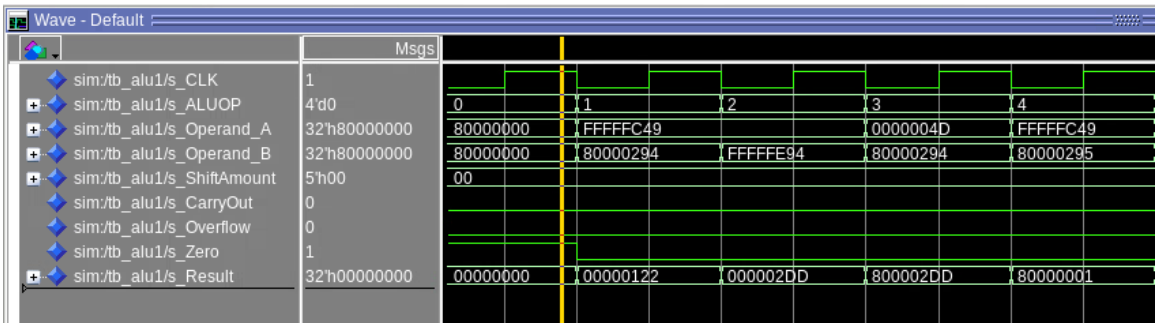
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

Set Less than (o_Result = 0x00000001 when slt is true, o_Result = 0x00000000 when slt is false)

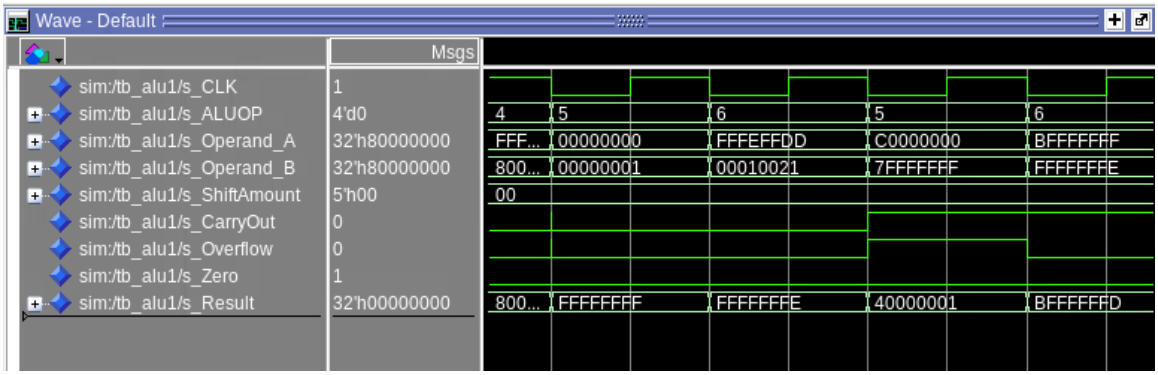
```
--slt false (A=0, B = -2147483648, A > B)
s_Operand_A <= "00000000000000000000000000000000";
s_Operand_B <= "10000000000000000000000000000000";
s_ShiftAmount <= "00000";
s_ALUOP <= "0000";
WAIT FOR cCLK_PER;

--slt true (A= -2147483648 B = 1073741824, A > B)
s_Operand_A <= "10000000000000000000000000000000";
s_Operand_B <= "01000000000000000000000000000000";
s_ShiftAmount <= "00000";
s_ALUOP <= "0000";
WAIT FOR cCLK_PER;

--slt false A = B, Zero Flag should also be triggered
s_Operand_A <= "10000000000000000000000000000000";
s_Operand_B <= "10000000000000000000000000000000";
s_ShiftAmount <= "00000";
s_ALUOP <= "0000";
WAIT FOR cCLK_PER;
```



```
---------
--Gates--
---------

    --NOR, Expected-00000000000000000000000100100010 = 0x00000122
    s_Operand_A <= "11111111111111111111110001001001";
    s_Operand_B <= "10000000000000000000001010010100";
    s_ShiftAmount <= "00000";
    s_ALUOP <= "0001";
    WAIT FOR cCLK_PER;

    --XOR, Expected-00000000000000000000001011011101 = 0x00002DD
    s_Operand_A <= "11111111111111111111110001001001";
    s_Operand_B <= "11111111111111111111111010010100";
    s_ShiftAmount <= "00000";
    s_ALUOP <= "0010";
    WAIT FOR cCLK_PER;

    --OR, Expected- 10000000000000000000001011011101 = 0x800002DD
    s_Operand_A <= "00000000000000000000000001001101";
    s_Operand_B <= "10000000000000000000001010010100";
    s_ShiftAmount <= "00000";
    s_ALUOP <= "0011";
    WAIT FOR cCLK_PER;

    --AND, Expected-10000000000000000000000000000001 = 0x80000001
    s_Operand_A <= "11111111111111111111110001001001";
    s_Operand_B <= "10000000000000000000001010010101";
    s_ShiftAmount <= "00000";
    s_ALUOP <= "0100";
    WAIT FOR cCLK_PER;
```



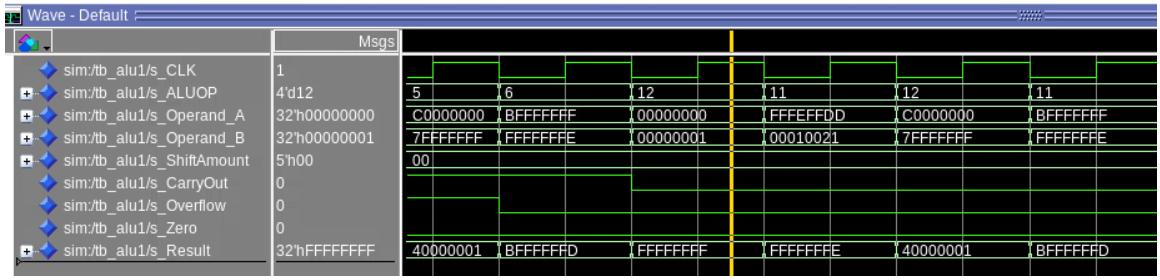
```
--------
--Math--
--------
        --SUB, Expected-11111111111111111111111111111111 = 0xFFFFFFFF
        s_Operand_A <= "00000000000000000000000000000000";
        s_Operand_B <= "00000000000000000000000000000001";
        s_ShiftAmount <= "00000";
        s_ALUOP <= "0101";
        WAIT FOR cCLK_PER;

        --ADD, Expected-11111111111111111111111111111110 = 0xFFFFFFFE
        s_Operand_A <= "11111111111111101111111111011101";
        s_Operand_B <= "00000000000000010000000000100001";
        s_ShiftAmount <= "00000";
        s_ALUOP <= "0110";
        WAIT FOR cCLK_PER;

        --SUB Overflow = 1, Expected-11111111111111111111111111111111 = 0x40000001
        s_Operand_A <= "11000000000000000000000000000000";
        s_Operand_B <= "01111111111111111111111111111111";
        s_ShiftAmount <= "00000";
        s_ALUOP <= "0101";
        WAIT FOR cCLK_PER;

        --ADD Overflow = 1, Expected-11111111111111111111011011110 = 0xBFFFFFFD
        s_Operand_A <= "10111111111111111111111111111111";
        s_Operand_B <= "11111111111111111111111111111110";
        s_ShiftAmount <= "00000";
        s_ALUOP <= "0110";
        WAIT FOR cCLK_PER;
---------------------------
```



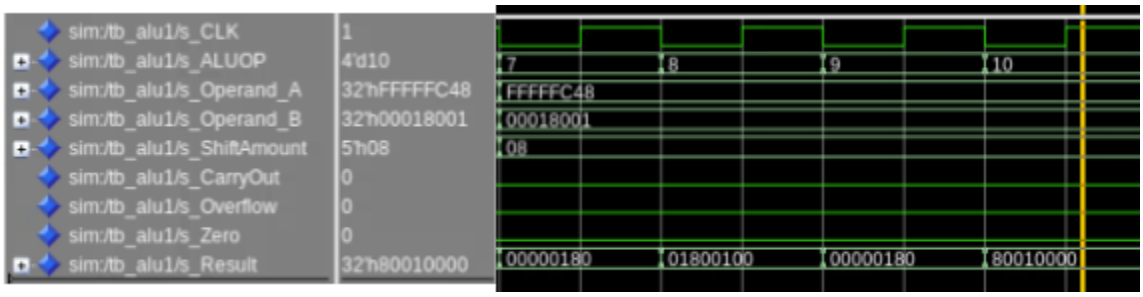
Math 2: Unsigned

```
--------------------
--SUBU, Expected-11111111111111111111111111111111 = 0xFFFFFFFF
s_Operand_A <= "00000000000000000000000000000000";
s_Operand_B <= "00000000000000000000000000000001";
s_ShiftAmount <= "00000";
s_ALUOP <= "1100";
WAIT FOR cCLK_PER;

--ADDU, Expected-11111111111111111111111111111111 = 0xFFFFFFFE
s_Operand_A <= "11111111111111101111111111011101";
s_Operand_B <= "00000000000000010000000000100001";
s_ShiftAmount <= "00000";
s_ALUOP <= "1011";
WAIT FOR cCLK_PER;

--SUBU Overflow = 0 (Not triggered, but is overflow), Expected-11111111111111111111111111111111 = 0x40000001
s_Operand_A <= "11000000000000000000000000000000";
s_Operand_B <= "01111111111111111111111111111111";
s_ShiftAmount <= "00000";
s_ALUOP <= "1100";
WAIT FOR cCLK_PER;

--ADDU Overflow = 0 (Not triggered, but is overflow), Expected-11111111111111111111011011110 = 0xBFFFFFFD
s_Operand_A <= "10111111111111111111111111111111";
s_Operand_B <= "11111111111111111111111111111110";
s_ShiftAmount <= "00000";
s_ALUOP <= "1011";
WAIT FOR cCLK_PER;
```

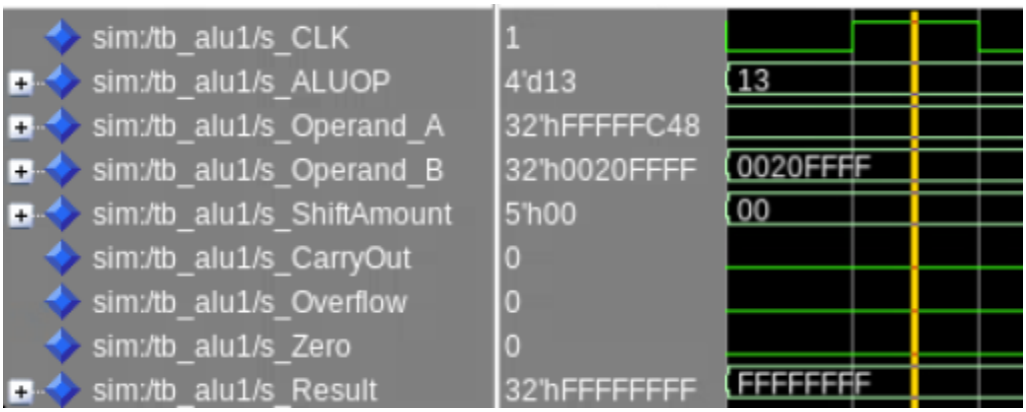
Note First 2 clocks when ALUOP =5,6 are from the testing of sub and add. Clocks 3,4,5 and 6 of this waveform match the tests in the above picture (ALUOP's 12 and 11)



```
----------
--Shift--
----------
        --SRL, Shift 8 Right Expect 0x00000180
        s_Operand_A <= "11111111111111111111110001001000";
        s_Operand_B <= "00000000000000011000000000000001";--0x00018001
        s_ShiftAmount <= "01000";
        s_ALUOP <= "0111";
        WAIT FOR cCLK_PER;

        --SLL, Shift 8 Left Expect 0x01800100
        s_Operand_A <= "11111111111111111111110001001000";
        s_Operand_B <= "00000000000000011000000000000001";--0x00018001
        s_ShiftAmount <= "01000";
        s_ALUOP <= "1000";
        WAIT FOR cCLK_PER;

        --SRA, Shift 8 Right Expect 0xFF000180
        s_Operand_A <= "11111111111111111111110001001000";
        s_Operand_B <= "00000000000000011000000000000001";--0x00018001
        s_ShiftAmount <= "01000";
        s_ALUOP <= "1001";
        WAIT FOR cCLK_PER;

        --LUI, Shift 16 Right Expect 0x80010000
        s_Operand_A <= "11111111111111111111110001001000";
        s_Operand_B <= "00000000000000011000000000000001";--0x00018001
        s_ShiftAmount <= "01000";
        s_ALUOP <= "1010";
        WAIT FOR cCLK_PER;
```

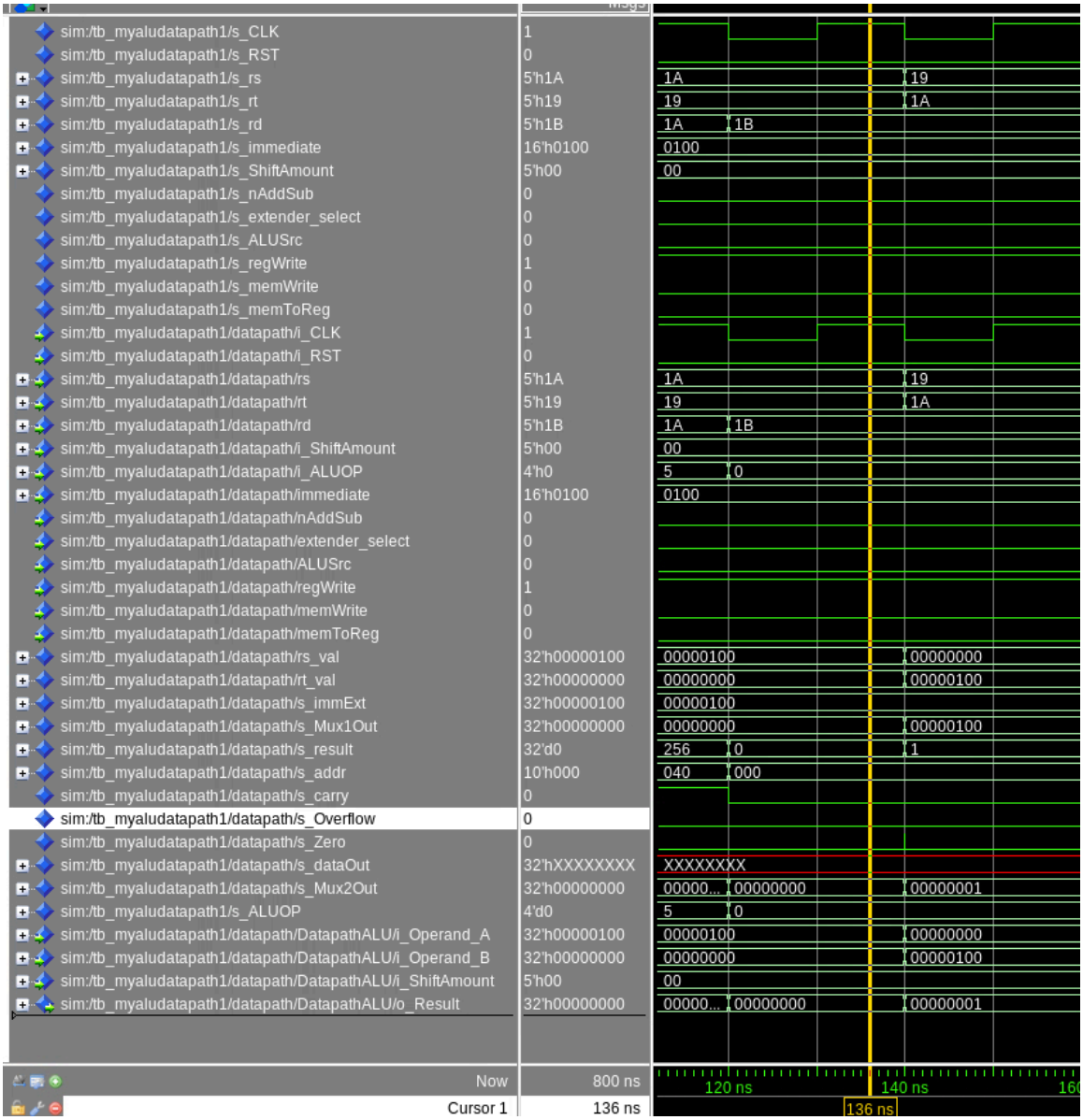



```
---------
--Other--
---------
        --Replicate, Expect 0xFFFFFFFF
        s_Operand_A <= "11111111111111111111110001001000";
        s_Operand_B <= "00000000010000011111111111111111";
        s_ShiftAmount <= "00000";
        s_ALUOP <= "1101";
        WAIT FOR cCLK_PER;
```

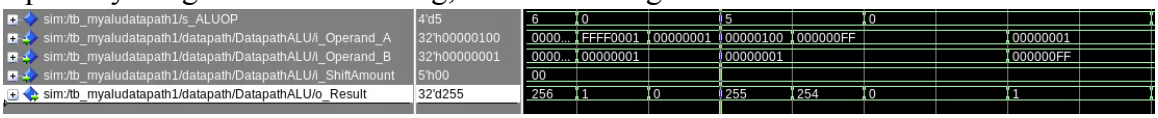


[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

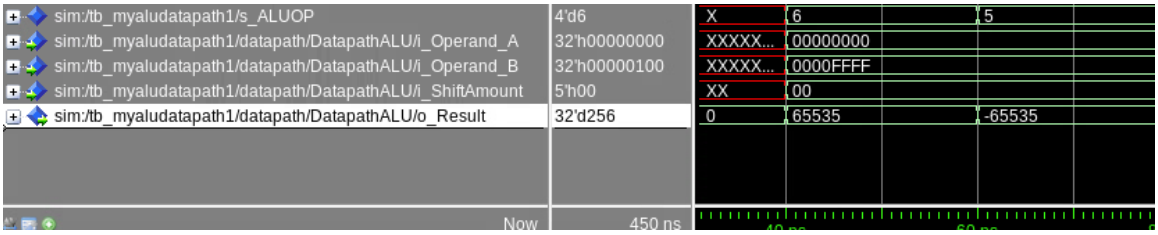
The test plan for implementing ALU with Datapath 2 was to add tests not tested in Part 2 (c.v) (i.e., slt was only tested with the opposite signs in Part 2 (c.v) so same signs were compared in the datapath test). Overall I felt my test cases for many of the functions in Part 2 (c.v) covered most scenarios.



Most of the test cases were written such that a series of instructions were run on one register since in Part 2 (c.v). Most of the cases used written values. I spent most of my time testing overflow (shown below) and slt (slt shown above). Most operations, especially the gate and shift testing, didn't have edge cases.



Subi and Addi largest immediate values



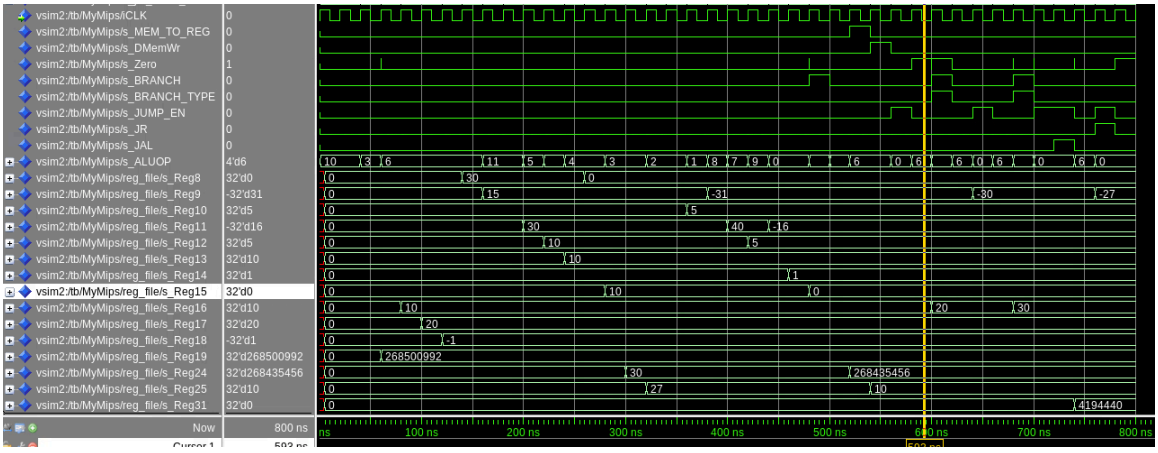
Shifts, Test of sra filling with 1's vs filling with 0's.

[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.

```
[collalti@linuxvdi-09 cpre381-toolflow]$ ./381_tf.sh test proj/mips/Proj1_base_test.s
/usr/bin/id: cannot find name for group ID 101
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Proj1_base_test.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 39
Processor Cycles: 39
CPI:   1.0
Results in: output/Proj1_base_test.s
-----------------------------------------------
```

Full Simulation:

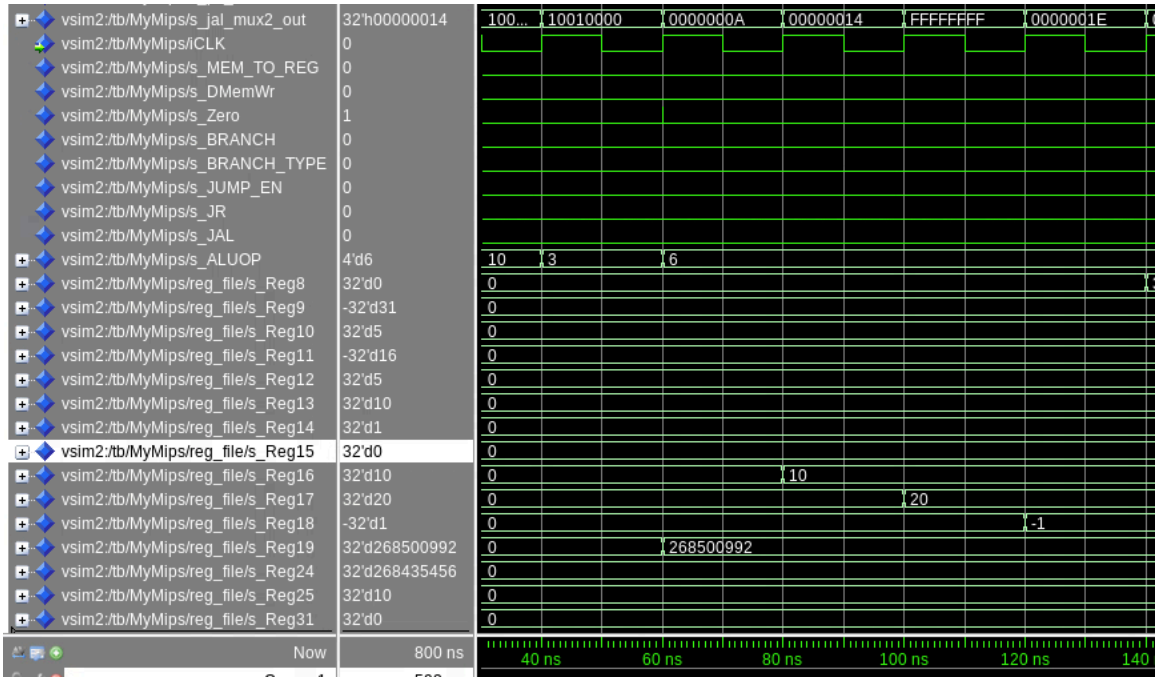


Initialization/Add Instructions: First instruction ALUOP=3 is used for load address, ALUOP=6 is add instruction and Cycle 1-3 after ALUOP set to 6 are addi, Cycle 4 is add, cycle 5 is addi to a non $0 register, ALUOP 11 is addiu first then addu second

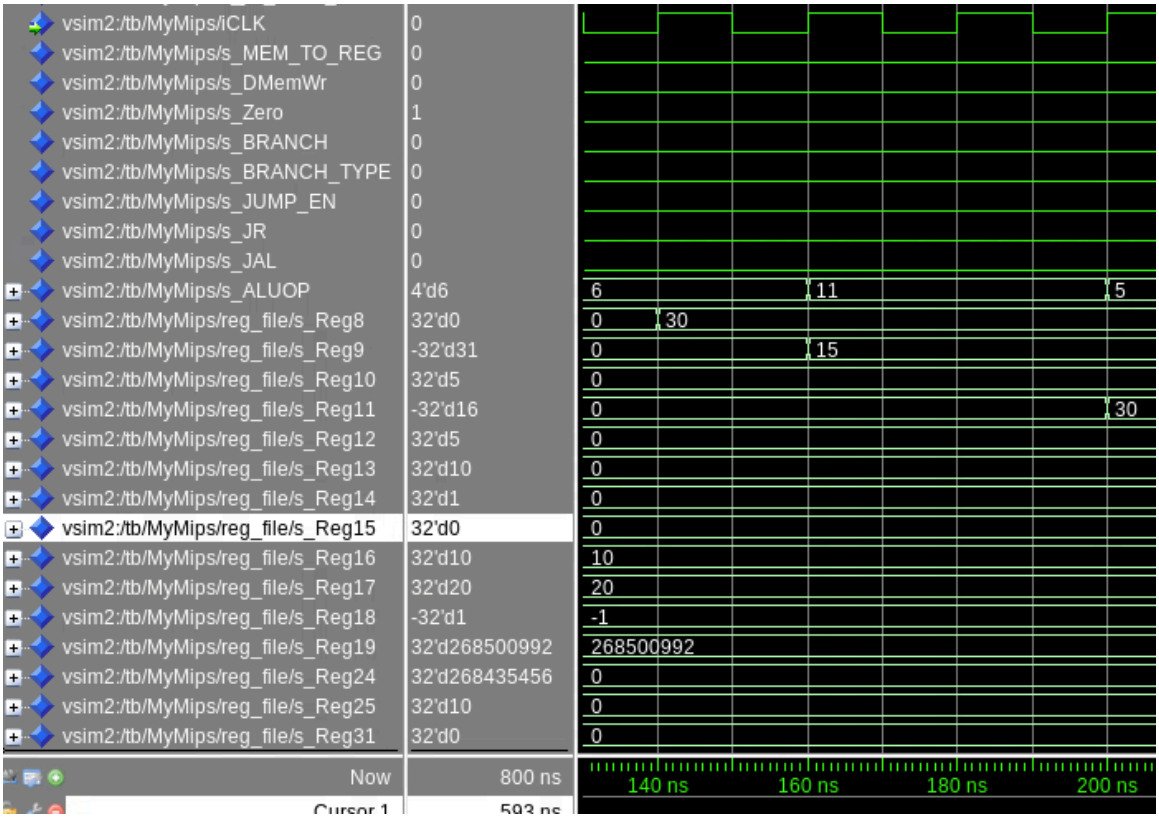
ALUOP cycle (Shown in the second screenshot). Sub Instructions: ALUOP =5 is sub, ALUOP =12 is subu (3rd screenshot)

```
.data
    array: .word 10, 20, 30, 40, 50   # Array for load/store tests
.text
.globl main

main:
    # Initialize some registers with values
    la   $s3, array              # Load base address of the array into $s3
    addi $s0, $0, 10            # $s0 = 10
    addi $s1, $0, 20            # $s1 = 20
    addi $s2, $0, -1            # $s2 = -1

    # Arithmetic operations
    add $t0, $s0, $s1     # $t0 = $s0 + $s1 = 10 + 20 = 30
    addi $t1, $s0, 5      # $t1 = $s0 + 5 = 10 + 5 = 15
    addiu $t2, $s2, 1     # $t2 = $s2 + 1 = -1 + 1 = 0
    addu $t3, $s0, $s1    # $t3 = $s0 + $s1 = 10 + 20 = 30
    sub $t4, $s1, $s0     # $t4 = $s1 - $s0 = 20 - 10 = 10
    subu $t5, $s1, $s0    # $t5 = $s1 - $s0 = 20 - 10 = 10
```

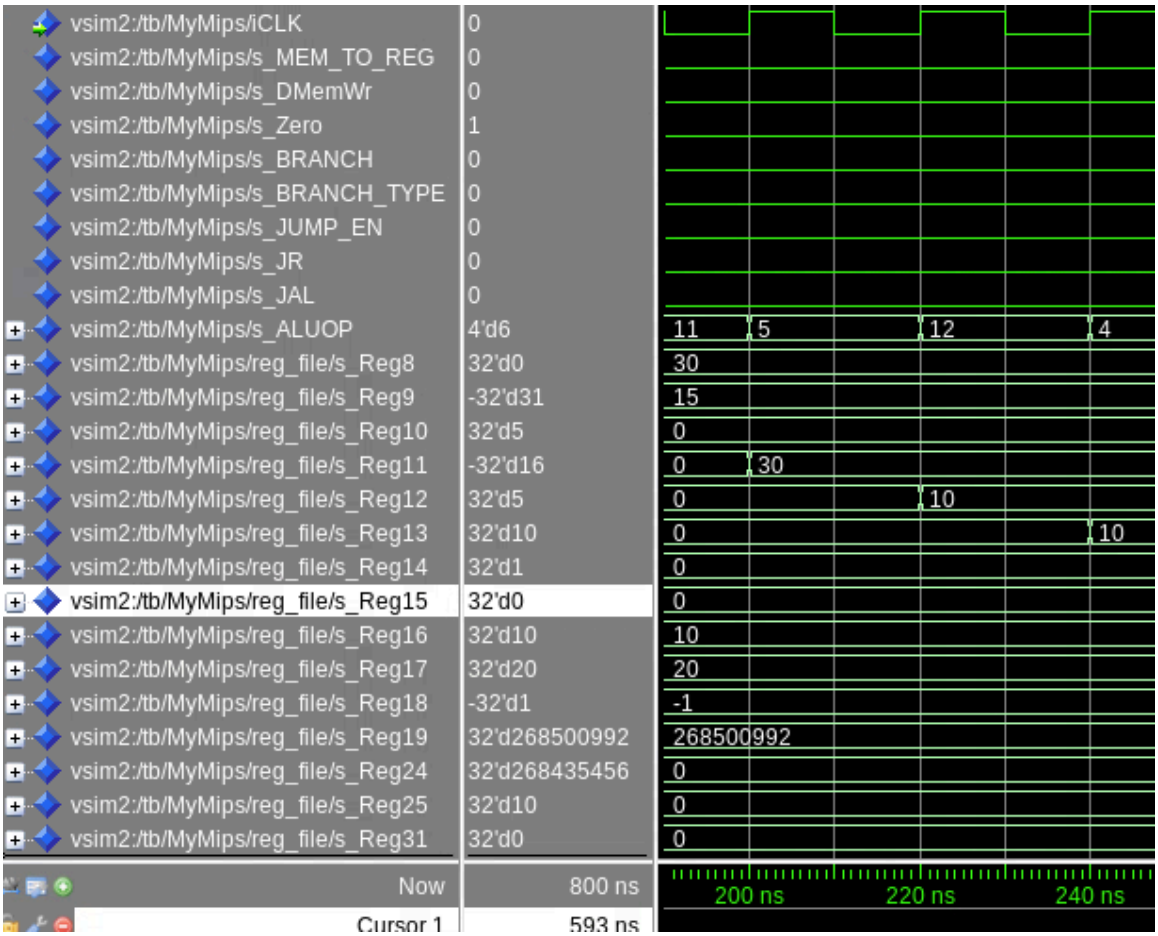
Initialization:



More Add's (add,addi,addiu,addu)


vsim2:/tb/MyMips/iCLK
vsim2:/tb/MyMips/s_MEM_TO_REG
vsim2:/tb/MyMips/s_DMemWr
vsim2:/tb/MyMips/s_Zero
vsim2:/tb/MyMips/s_BRANCH
vsim2:/tb/MyMips/s_BRANCH_TYPE
vsim2:/tb/MyMips/s_JUMP_EN
vsim2:/tb/MyMips/s_JR
vsim2:/tb/MyMips/s_JAL
vsim2:/tb/MyMips/s_ALUOP
vsim2:/tb/MyMips/reg_file/s_Reg8
vsim2:/tb/MyMips/reg_file/s_Reg9
vsim2:/tb/MyMips/reg_file/s_Reg10
vsim2:/tb/MyMips/reg_file/s_Reg11
vsim2:/tb/MyMips/reg_file/s_Reg12
vsim2:/tb/MyMips/reg_file/s_Reg13
vsim2:/tb/MyMips/reg_file/s_Reg14
vsim2:/tb/MyMips/reg_file/s_Reg15
vsim2:/tb/MyMips/reg_file/s_Reg16
vsim2:/tb/MyMips/reg_file/s_Reg17
vsim2:/tb/MyMips/reg_file/s_Reg18
vsim2:/tb/MyMips/reg_file/s_Reg19
vsim2:/tb/MyMips/reg_file/s_Reg24
vsim2:/tb/MyMips/reg_file/s_Reg25
vsim2:/tb/MyMips/reg_file/s_Reg31
0
0
0
1
0
0
0
0
0
4'd6
32'd0
-32'd31
32'd5
-32'd16
32'd5
32'd10
32'd1
32'd0
32'd10
32'd20
-32'd1
32'd268500992
32'd268435456
32'd10
32'd0
6
11
5
0
30
0
15
0
0
30
0
0
0
0
10
20
-1
268500992
0
0
0
Now
800 ns
140 ns
160 ns
180 ns
200 ns

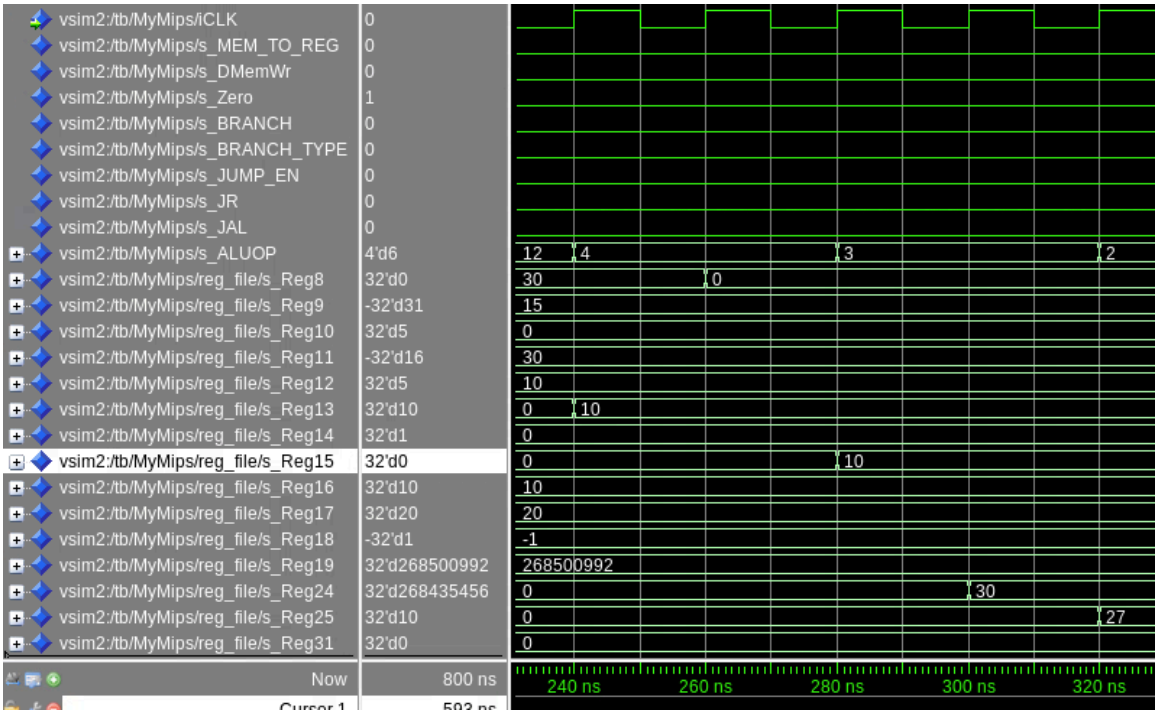
Sub’s



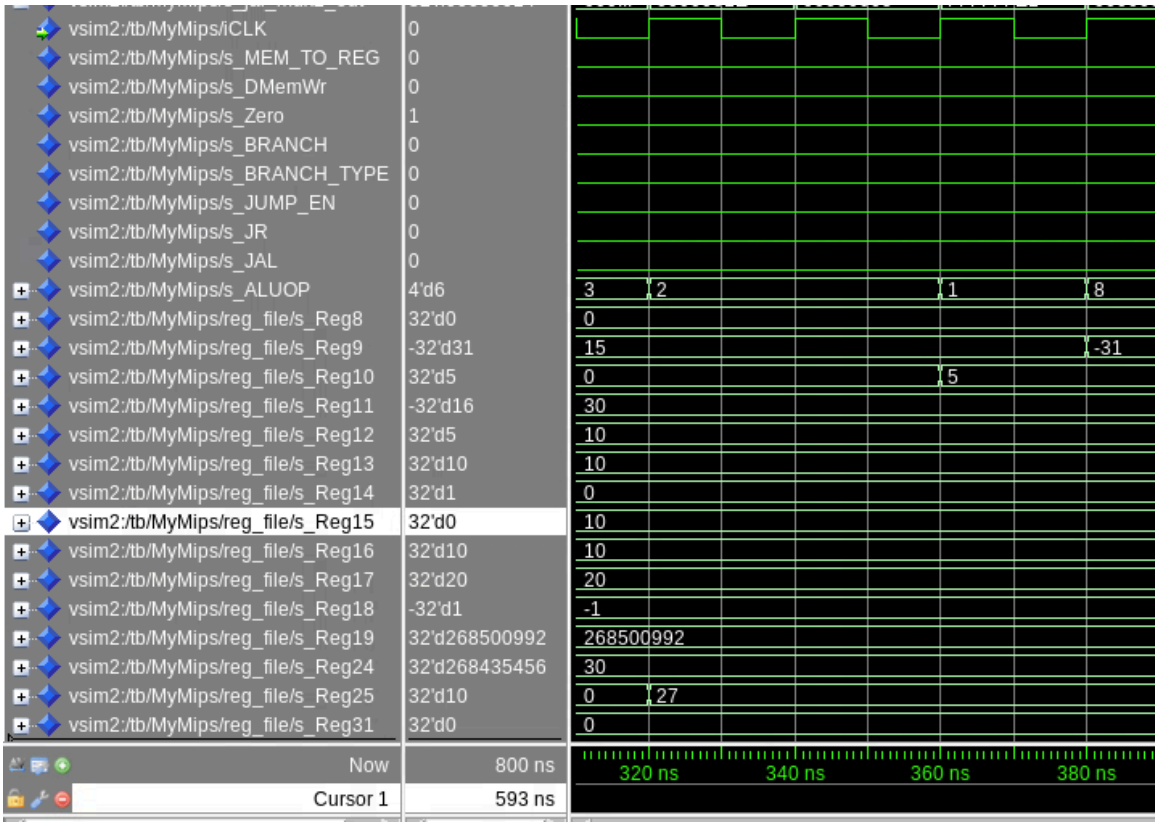
Logical operations: ALUOP = 4 = AND, ALUOP = 3 = OR, ALUOP = 2 = XOR, ALUOP = 1 = NOR

```
# Logical operations
and $t0, $s0, $s1     # $t0 = $s0 & $s1 = 0
andi $t7, $s0, 15     # $t7 = $s0 & 15 = 10
or $t8, $s0, $s1      # $t8 = $s0 | $s1 = 30
ori $t9, $s0, 25      # $t9 = $s0 | 25 = 27
xor $t3, $s0, $s1     # $t8 = $s0 ^ $s1 = 30
xori $t2, $s0, 15     # $t1 = $s0 ^ 15 = 5
nor $t1, $s0, $s1     # $t2 = ~($s0 | $s1) = FFFFFFE1 = -31
```

AND’s and OR’s

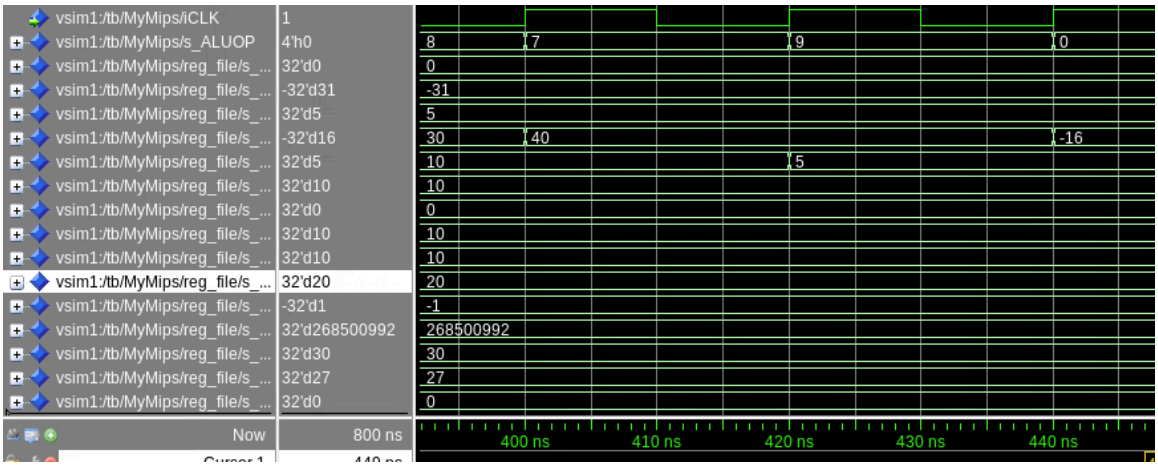


XOR's and NOR



Shift Operations: ALUOP = 8 = sll, ALUOP = 7 = srl, ALUOP = 9 = sra

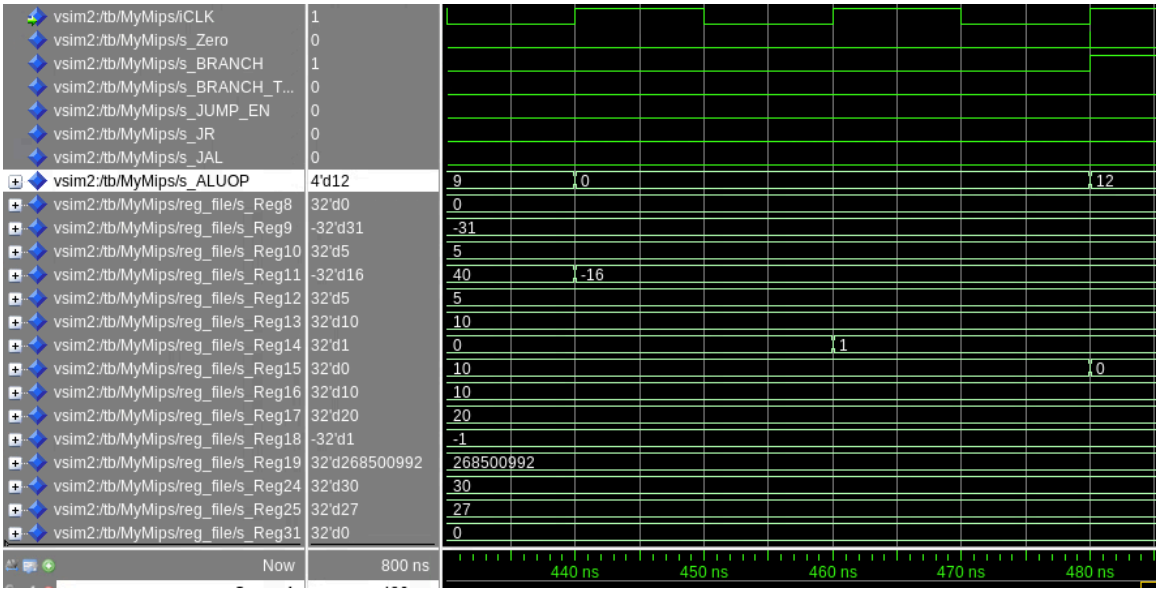
```
# Shift operations
sll $t3, $s0, 2      # $t3 = $s0 << 2 = 40
srl $t4, $s1, 2      # $t4 = $s1 >> 2 (logical shift) = 5
sra $t3, $t1, 1      # $t3 = $t1 >> 2 (arithmetic shift) = FFFFFFF0 = -16
```



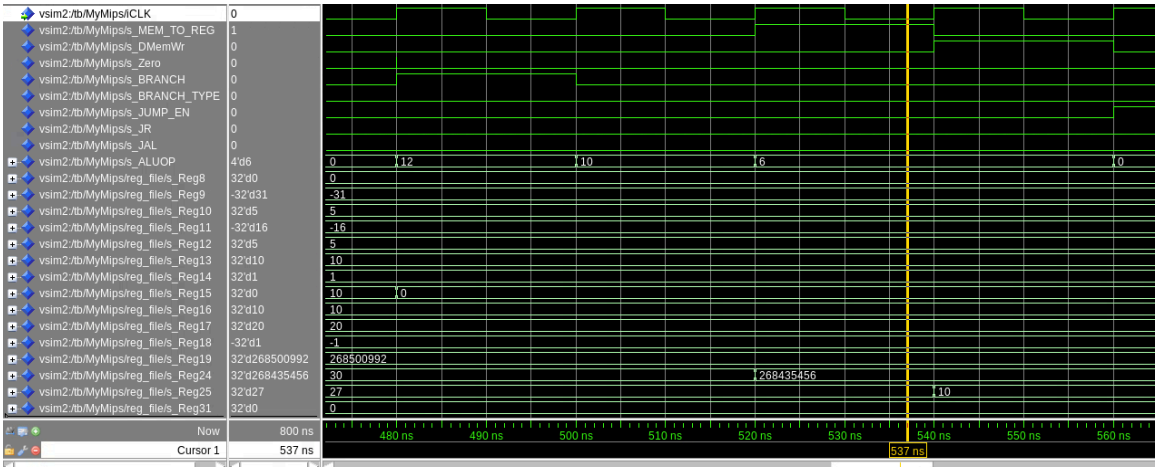
Comparisons, Control Flow, Loads/Stores:
ALUOP = 0 = slt
ALUOP = 12 = bne (BranchType = 0),
ALUOP = 12 = beq (BranchType = 1),
ALUOP = 0 = j (JumpEn = 1, jr =0, jal=0),
ALUOP = 10 = lui
ALUOP = 6 = lw (MemToReg =1, MemWrite = 0),
ALUOP = 6 = sw (MemToReg =0, MemWrite = 1),
ALUOP = 1 = jal (JumpEn = 1, jr =0, jal=1),
ALUOP = 1 = jr (JumpEn = 1, jr =1, jal=0)

```
    # Comparison operations
    slt $t6, $s0, $s1    # $t6 = ($s0 < $s1) true
    slti $t7, $s0, 5    # $t7 = ($s0 < 5) false


    # Branch and Jump instructions
    bne $s0, $s1, label1 # If $s0 != $s1, jump to label1 true
Loops:
    addi $s0, $s0, 10           # $s0 = 20
    beq $s0, $s1, label2 # If $s0 == $s1, jump to label2 true
    j end

label1:
    lui $t8, 0x1000      # $t8 = 0x10000000 (upper immediate)
    lw $t9, 0($s3)       # Load word from array (10)
    sw $t4, 4($s3)       # Store 5 in memory at $t3 + 4
    j Loops

label2:
    addi $t1, $t1, 1     # Increment $t1 if branch taken
    j Loops

end:
    jal JumpAndLink
    halt

JumpAndLink:
    addi $t1, $t1, 3     # Increment $t1 by 3 if jal taken
    jr $ra
```

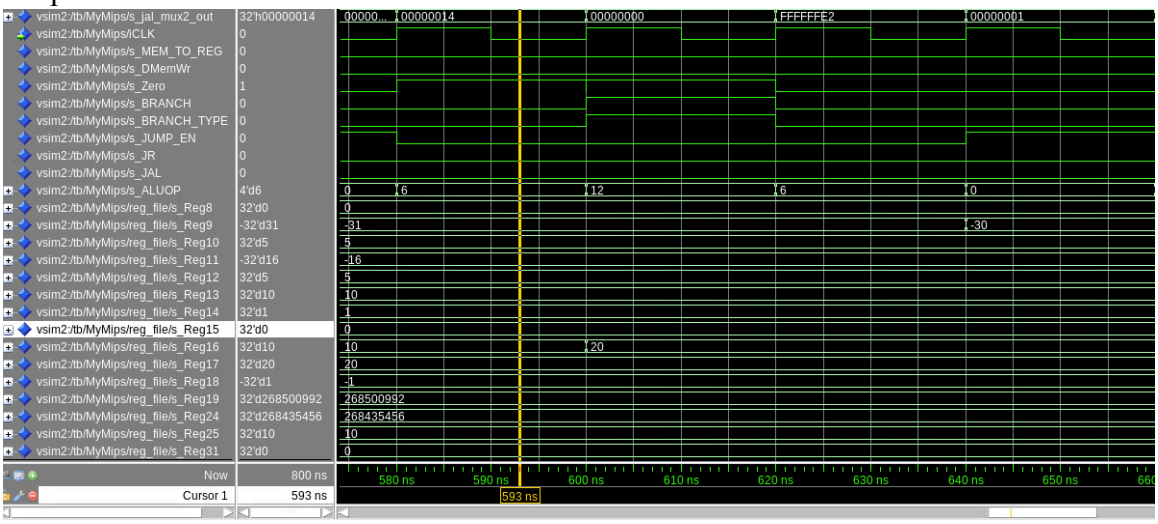
set less than’s



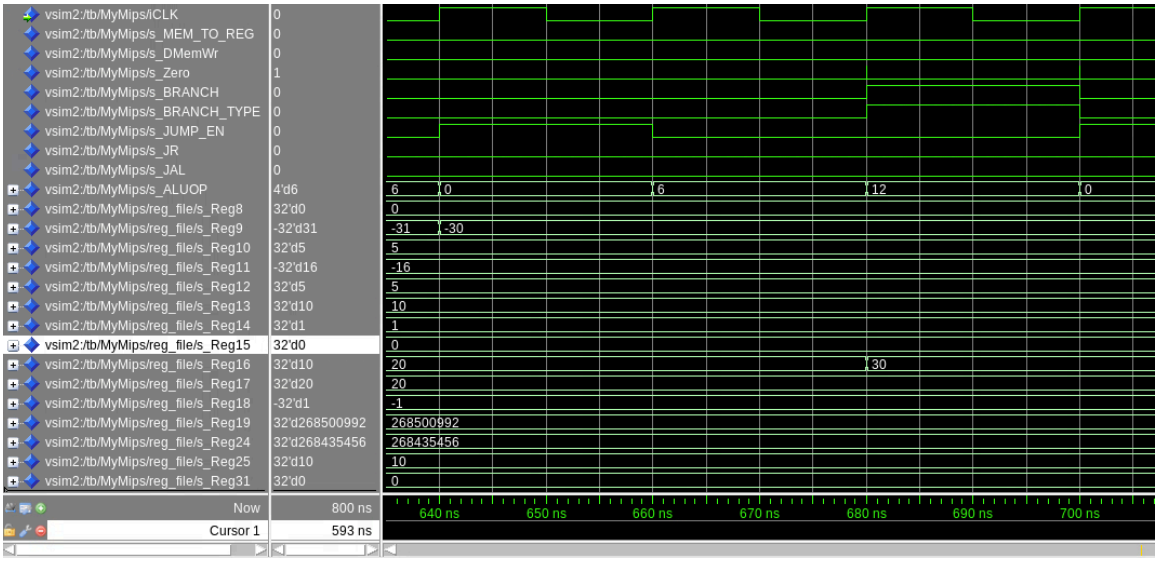
BNE, Label 1

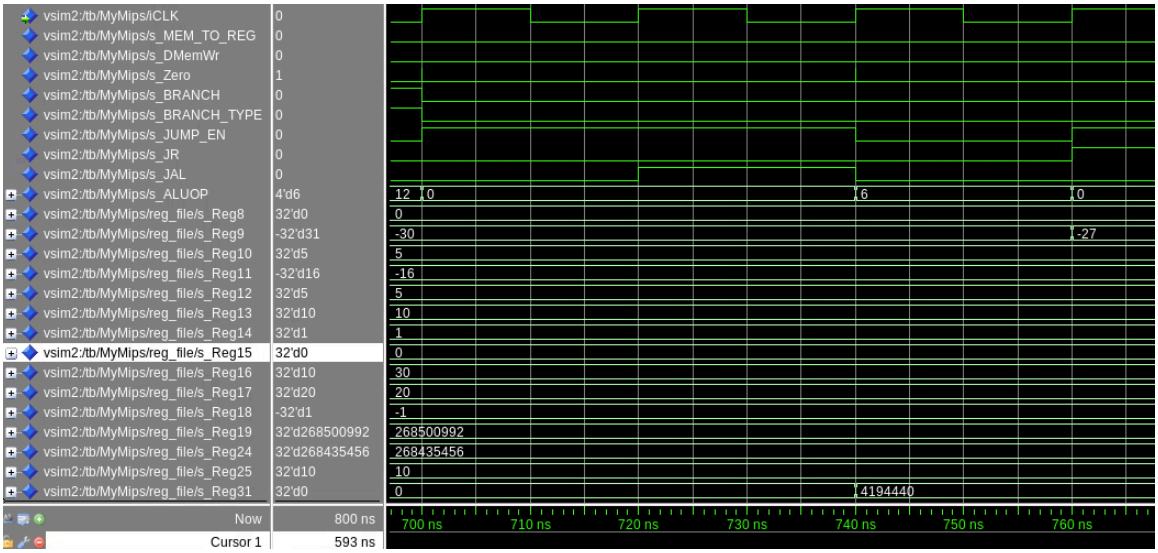


Loops and Label 2

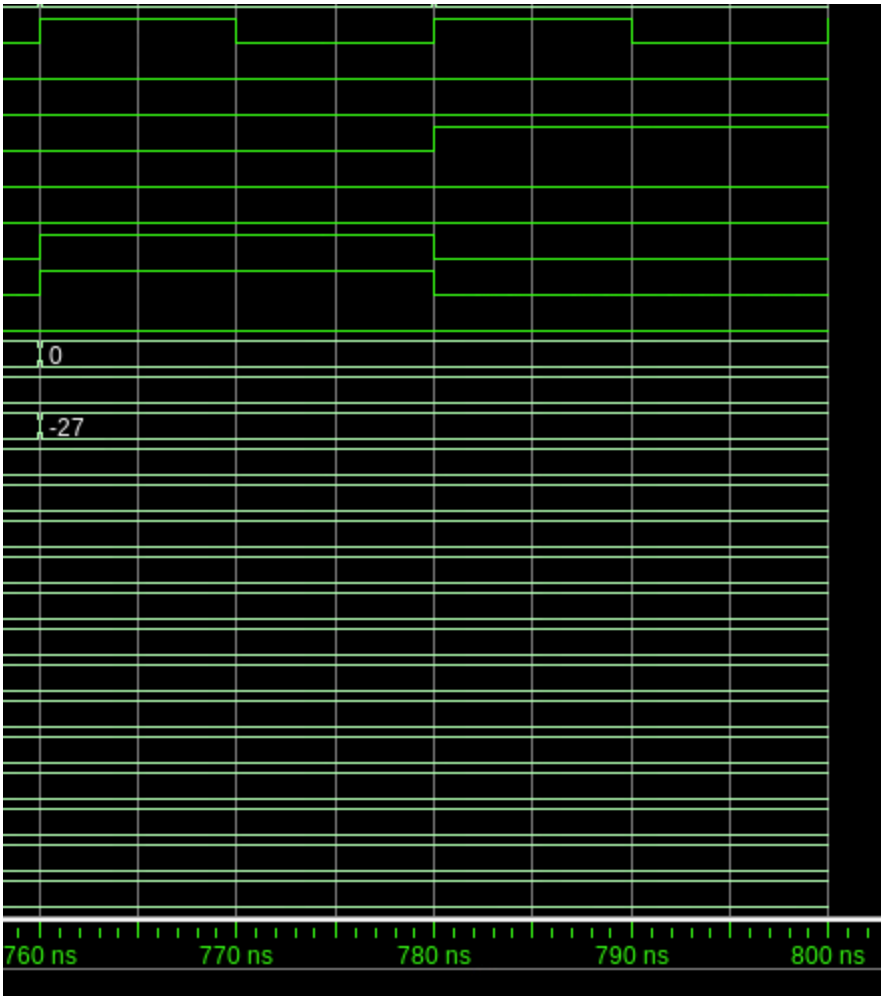


Loops Iteration 2, end, and end2





Last instruction

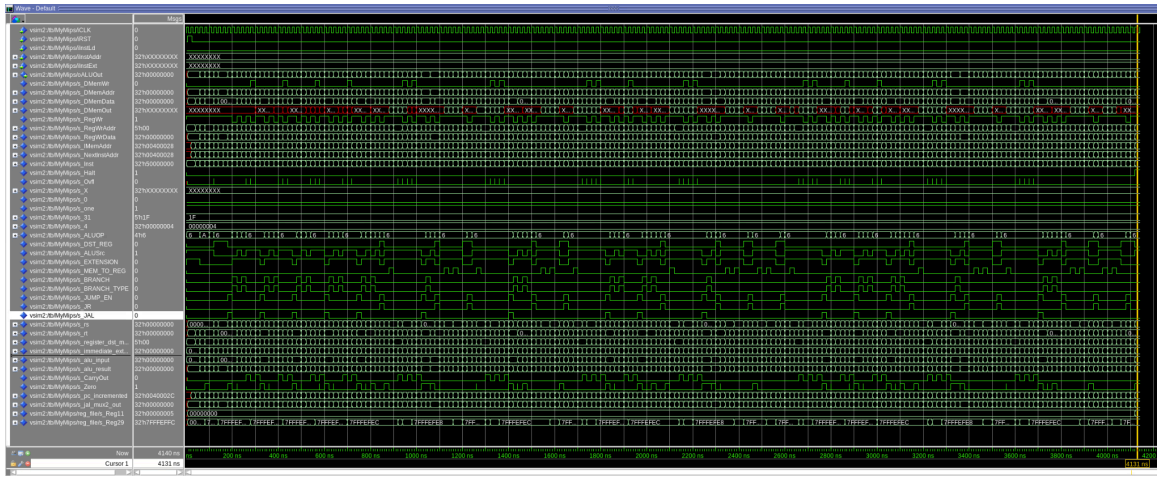


[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

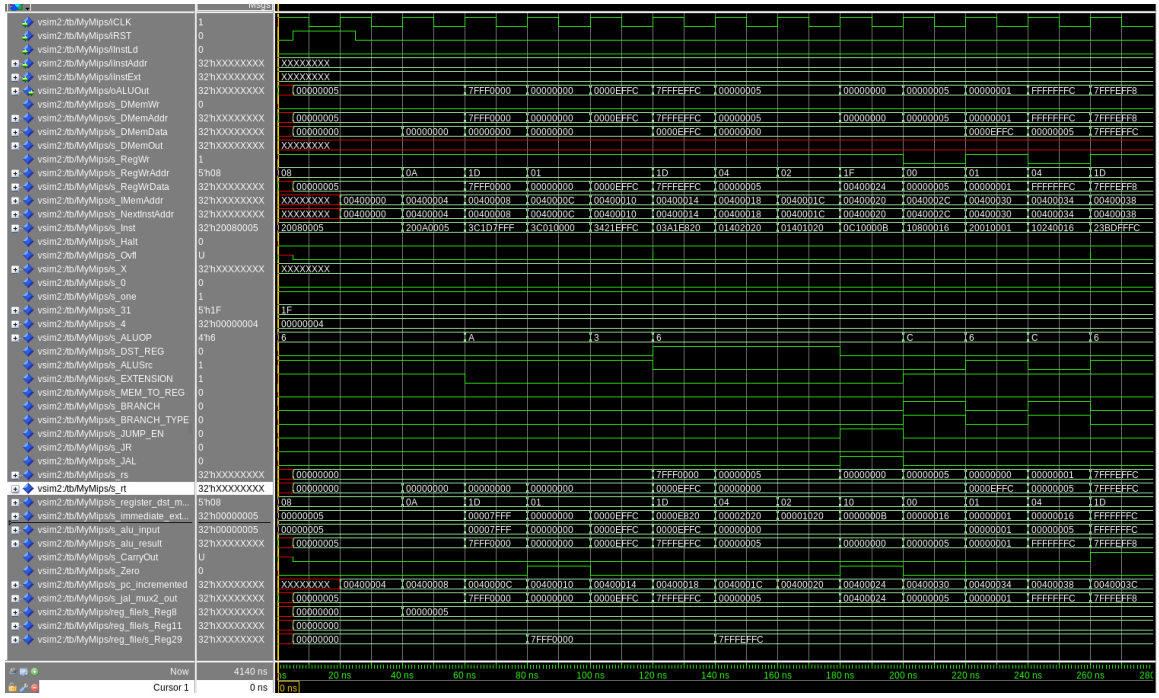
Application is Recursive Fibonacci in this case, the input is 5, so the call stack will have fib(5), fib(4), fib(3), fib(2), fib(1), and fib(0) with an expected output of 5

```
-----------------------------------------------
[collalti@linuxvdi-08 cpre381-toolflow]$ ./381_tf.sh test proj/mips/Proj1_cf_test.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Proj1_cf_test.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 206
Processor Cycles: 206
CPI:   1.0
Results in: output/Proj1_cf_test.s
-----------------------------------------------
```

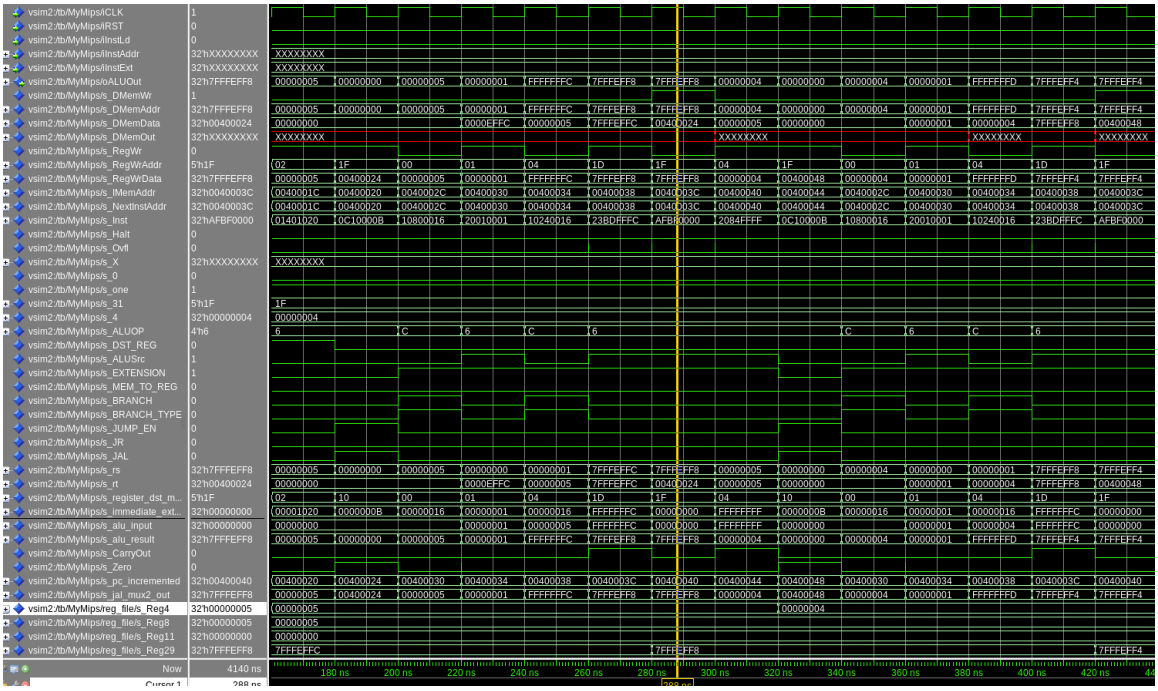
Full Multisim Simulation:



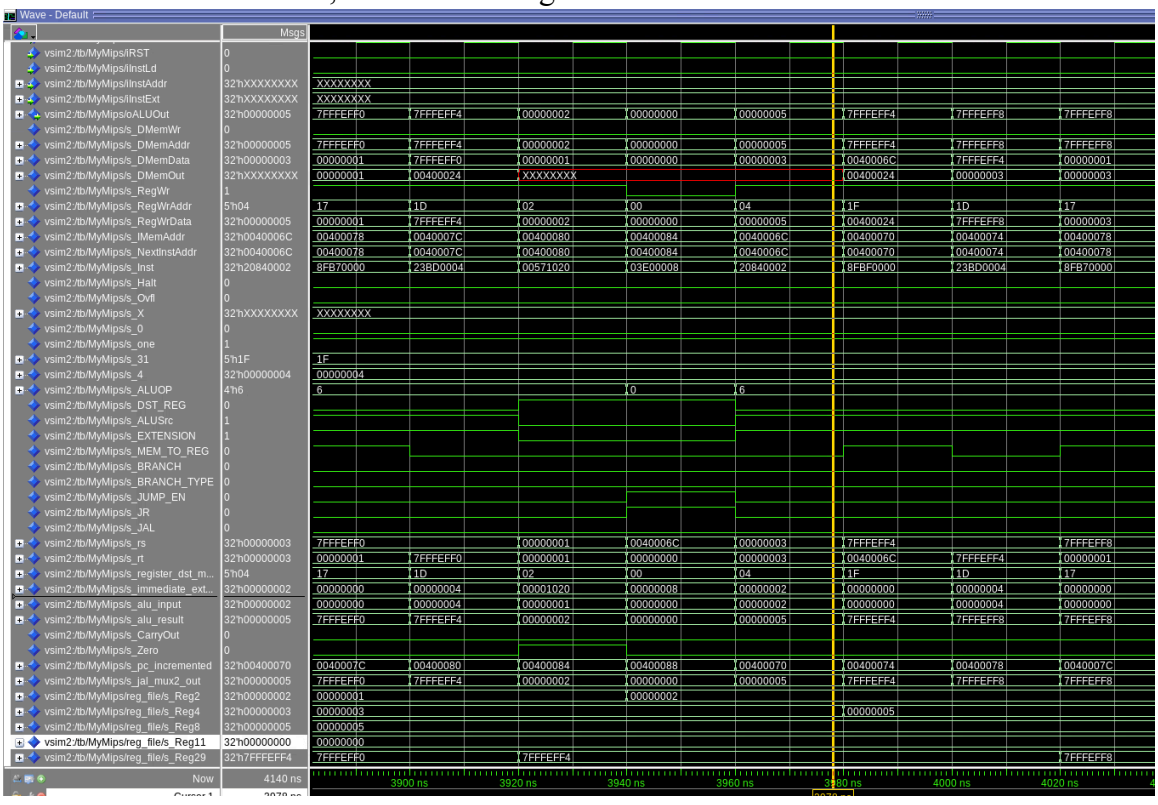
Start of Simulation: Reg29 ($sp) is initialized to it's proper location. Reg8($t0) is the input to Fibonacci, which is 5. Reg11($t3) is the output location which has the value 0 at the start of the program.



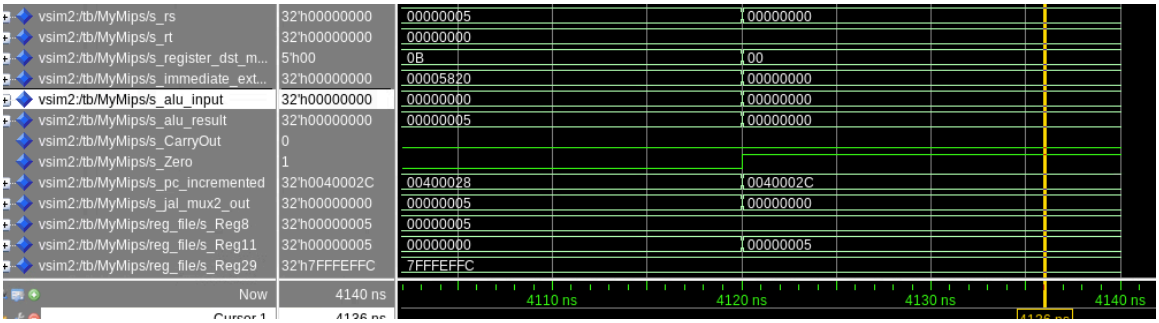
Example push onto the stack: Reg29($sp) is being changed to signify a new fibonacci call has been added. Since the stack pointer is decreased to signify a push onto the stack, Reg29($sp) goes from 7FFFEFFC (fib(5)) to 7FFFEFF8 (fib(4)) and then to 7FFFEFF4 (fib(3)), showing two fib calls in the below screenshot. Also, Reg4($a0) used as the fib function parameter one cycle after the first call 7FFFEFF8 ($a0) decreases from 5 to 4, meaning the current call is fib(4)



Example pop on the stack: Reg29($sp) is being changed to signify a Fibonacci call has been removed. Since the stack pointer is increased to signify a pop from the stack, Reg29($sp) goes from 7FFFEFF4 (fib(3)) to 7FFFEFF8 (fib(4)) and then to 7FFFEFFC (fib(5)), showing two fib pops in the below screenshot. At the cursor, fib(3) is being evaluated in this case to 2, as shown in Reg2.



End of Simulation: Reg11($t3) has the output, which is 5 as expected, and Reg8($t0) has the input, which is 5. Reg($29) is returned to the initialized value, meaning the stack is empty/ all Fibonacci calls have been popped.



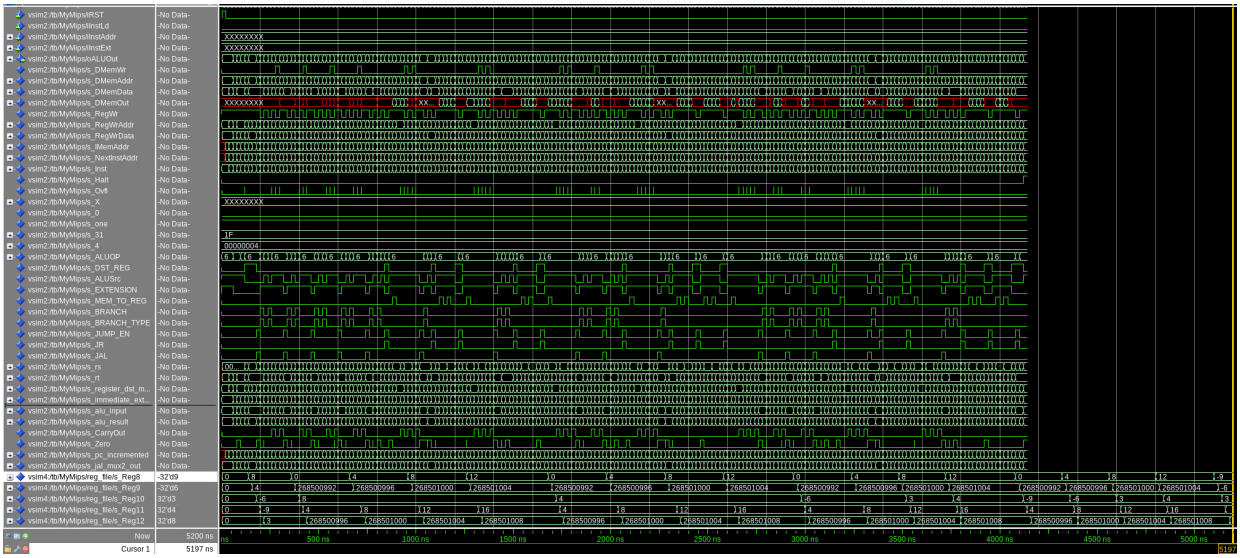
[Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

```
[collalti@linuxvdi-08 cpre381-toolflow]$ ./381_tf.sh test proj/mips/Proj1_bubblesort.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Proj1_bubblesort.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 259
Processor Cycles: 259
CPI:   1.0
Results in: output/Proj1_bubblesort.s
-----------------------------------------------
[collalti@linuxvdi-08 cpre381-toolflow]$
```

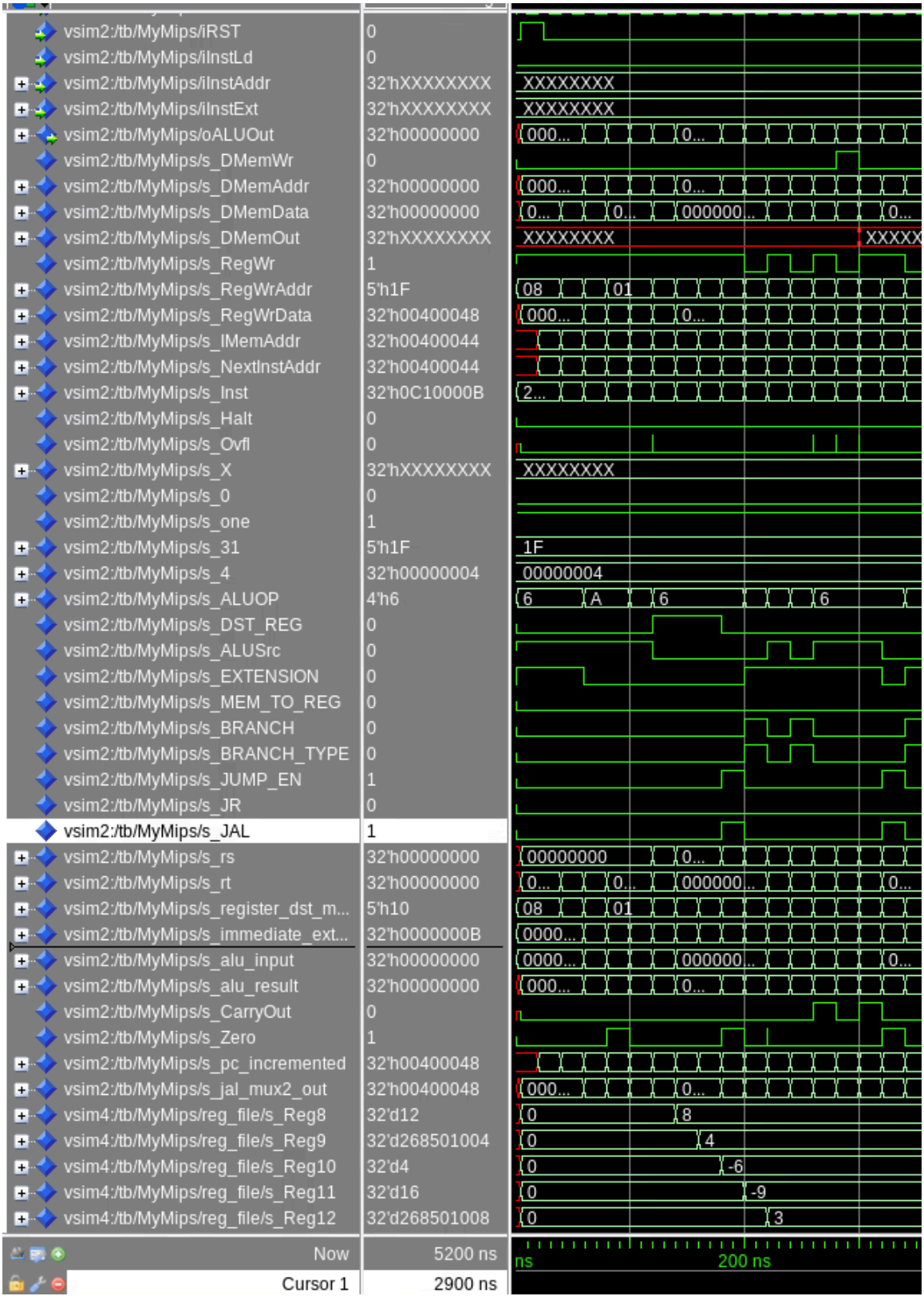
Starting Array in Code

```
array:   .word 8, 4, -6, -9, 3    # Array to be sorted
size:    .word 5                  # Size of the array
```

Full Simulation: The “overhang” on the bottom is just a series of load instructions to show the array being sorted

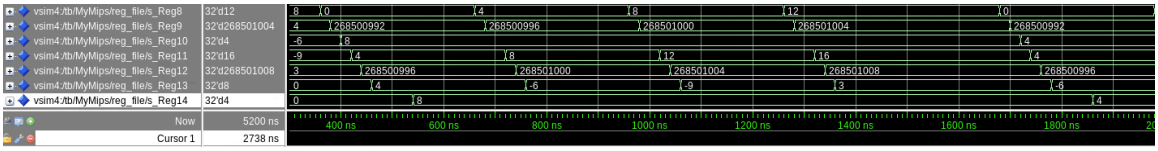


Start of Simulation: The first two cycles load the address of the array, and the second loads the size into a register. Reg8=array[0], Reg9=array[1], Reg10=array[2], Reg11=array[3], Reg12=array[4]

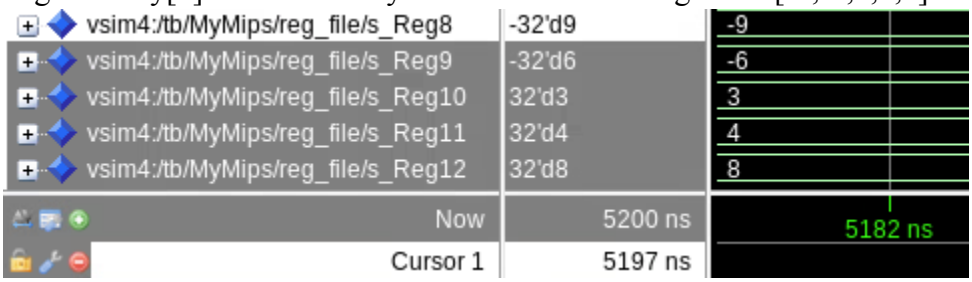


Sorting of 8 (First Element): Reg14 holds the value currently being sorted in the image, 8, and Reg13 holds the value being compared to Reg14. So the Image shows 8 being

bubbled up 8>4, 8>-6, 8>-9, 8>3. Then the next element begins its sorting, in this case 4. Reg9 keeps the address 8 is currently at, and as 8 gets bubbled up, Reg9 increases by 4. Then, Reg12 holds the address of the next element in the array being compared. In the case 8>4 the 8 and 4 swap addresses.



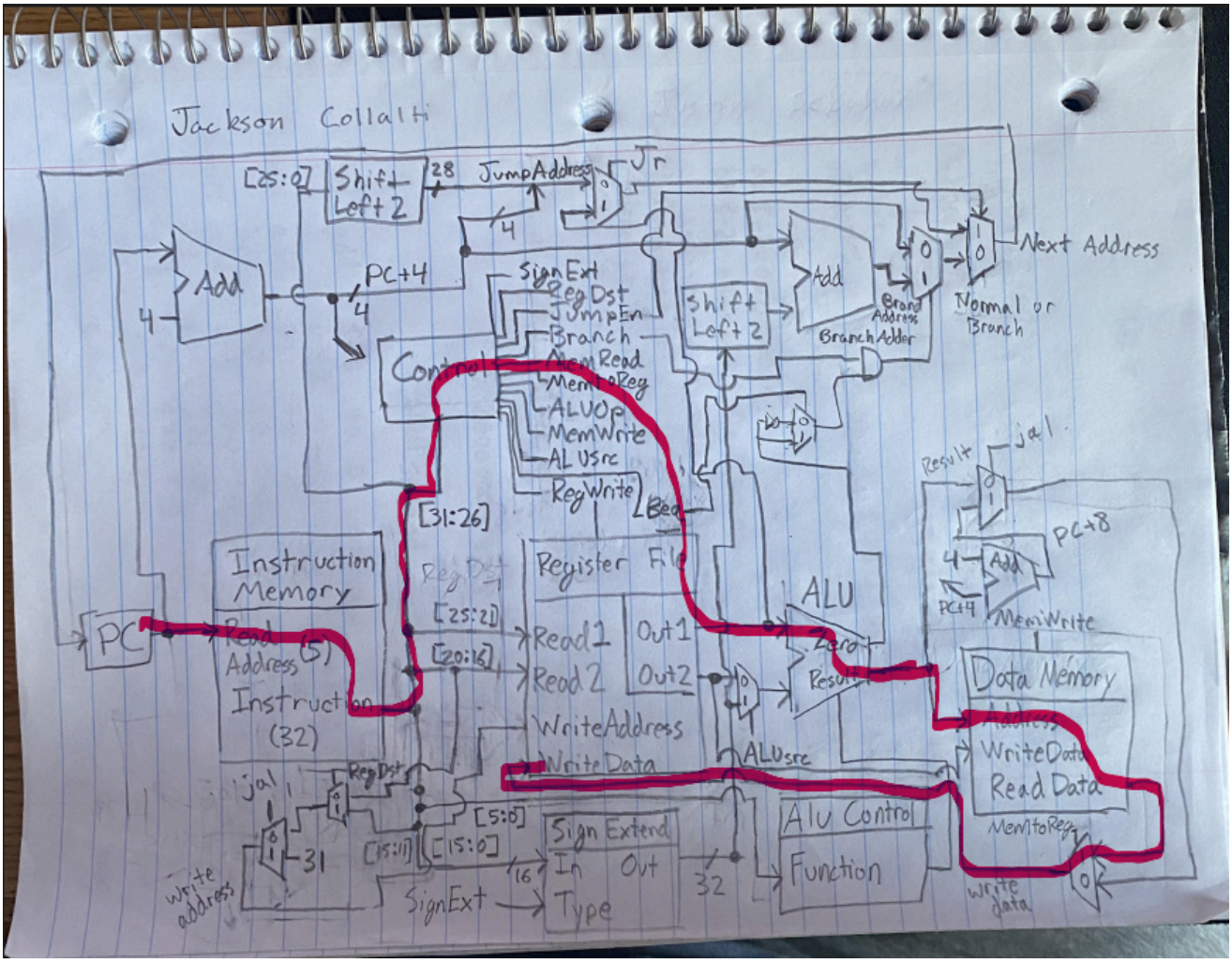
End of Simulation: Reg8=array[0], Reg9=array[1], Reg10=array[2], Reg11=array[3], Reg12=array[4]. Now the array is sorted in increasing order [-9,-6,3,4,8].



[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: 27.94 mhz

Critical path:
Fetch logic (PC) -> IMEM -> Control -> ALU -> DMEM -> RegWrite



To improve our cycle time, and increase frequency, one place to start would be optimizing the control unit. For this project, we did not make use of ‘don’t care’ values, and it seems to have impacted the performance of the control logic; that is, it is on the critical path when ideally it would pass through the RegRead instead. The ALU could also use some optimizations to bring it’s time down. We could potentially do this by reducing the number of internal components and optimizing the flow of logic within those subsystems.

Other than that, the time spent between the two memories is quite high, but there is not much we can do because the memory is provided and can’t necessarily be optimized.