CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

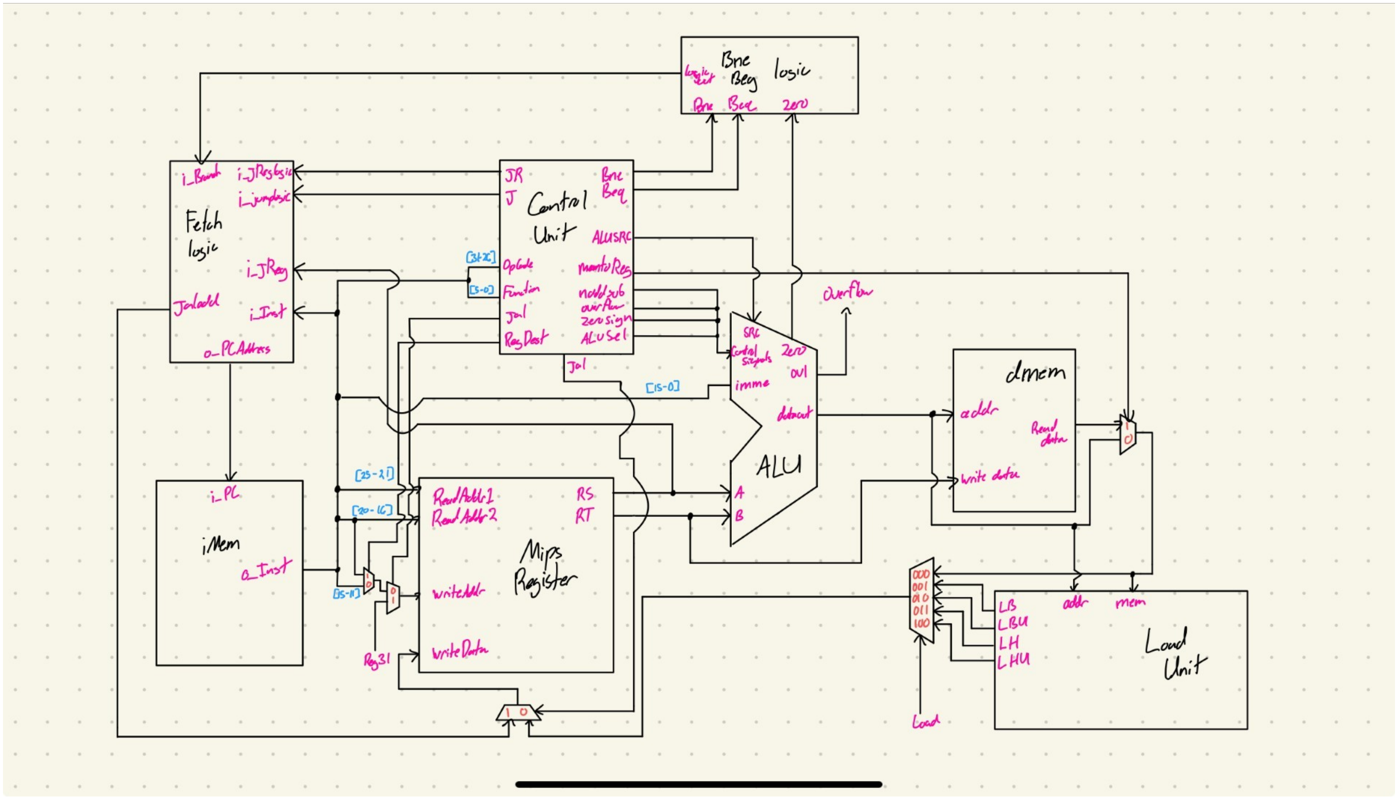
Team Members: __Camden Fergen__

__Emil Kosic__

Project Teams Group #: __Group 2__

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.

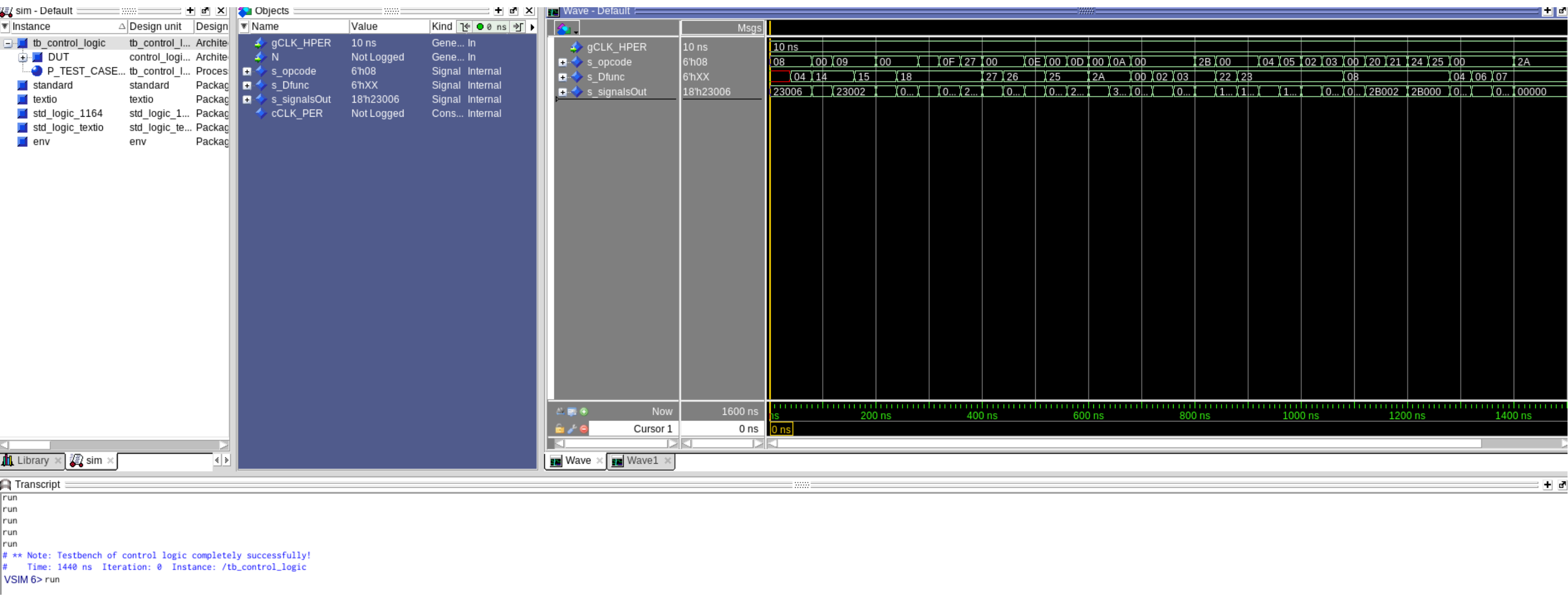


[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your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

Given in Proj1_control_signalsFinalNEW.xlsx

Control Logic

[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)



Instance	Design unit	Design
tb_control_logic	tb_control_l...	Archite
DUT	control_logi...	Archite
P_TEST_CASE...	tb_control_l...	Proces
standard	standard	Packag
textio	textio	Packag
std_logic_1164	std_logic_1...	Packag
std_logic_textio	std_logic_te...	Packag
env	env	Packag

Name	Value	Kind	
gCLK_HPER	10 ns	Gene...	In
N	Not Logged	Gene...	In
s_opcode	6'h08	Signal	Internal
s_Dfunc	6'hXX	Signal	Internal
s_signalsOut	18'h23006	Signal	Internal
cCLK_PER	Not Logged	Cons...	Internal

	Msgs
gCLK_HPER	10 ns
s_opcode	6'h08
s_Dfunc	6'hXX
s_signalsOut	18'h23006
Now	1600 ns
Cursor 1	0 ns

```
run
run
run
run
run
# ** Note: Testbench of control logic completely successfully!
#    Time: 1440 ns  Iteration: 0  Instance: /tb_control_logic
VSIM 6> run
```

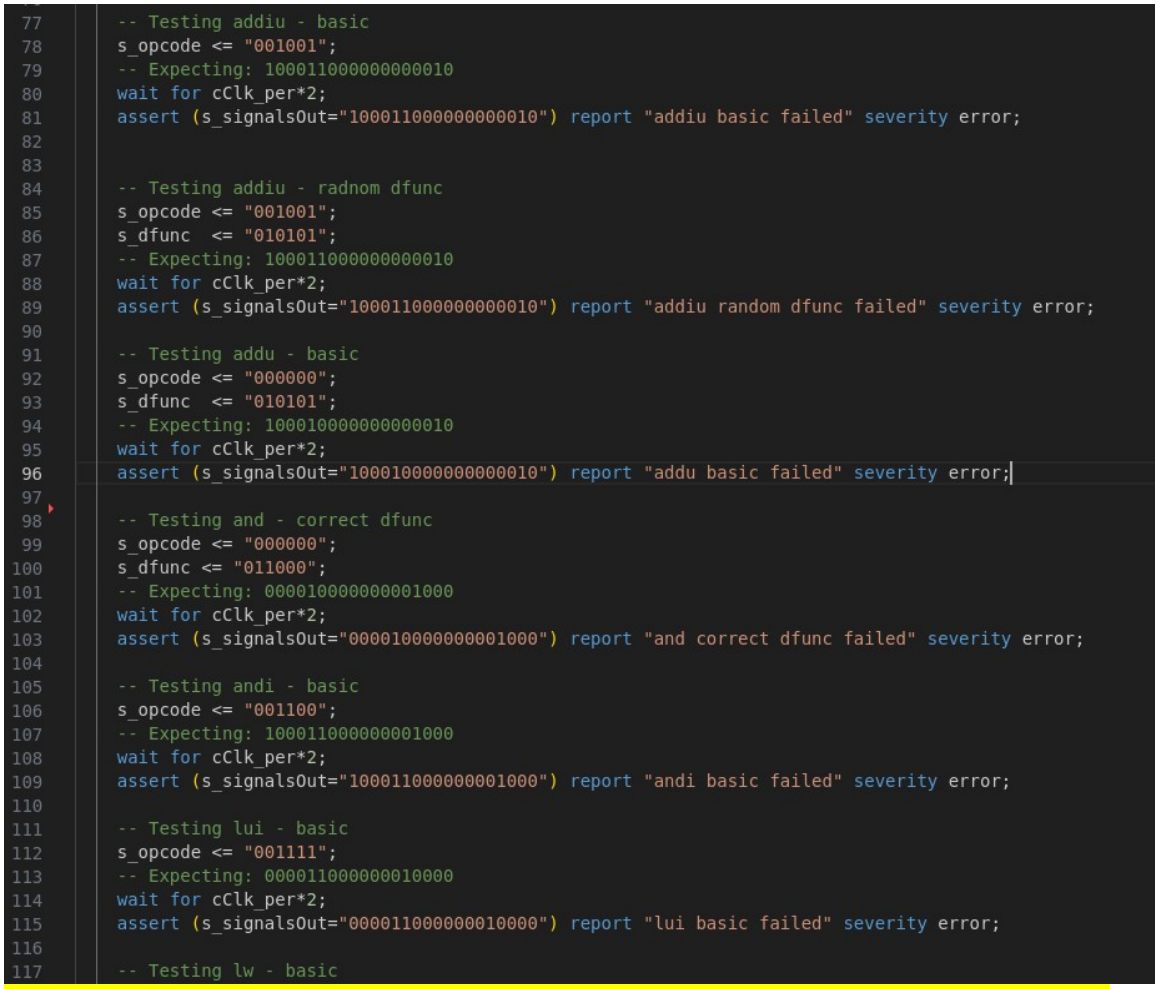
.
You can see in these two images that all tests did in fact pass, since the console did not report any errors when running through all the tests. The assert lines are the exact same as found in the control signals spreadsheet

Fetch Logic

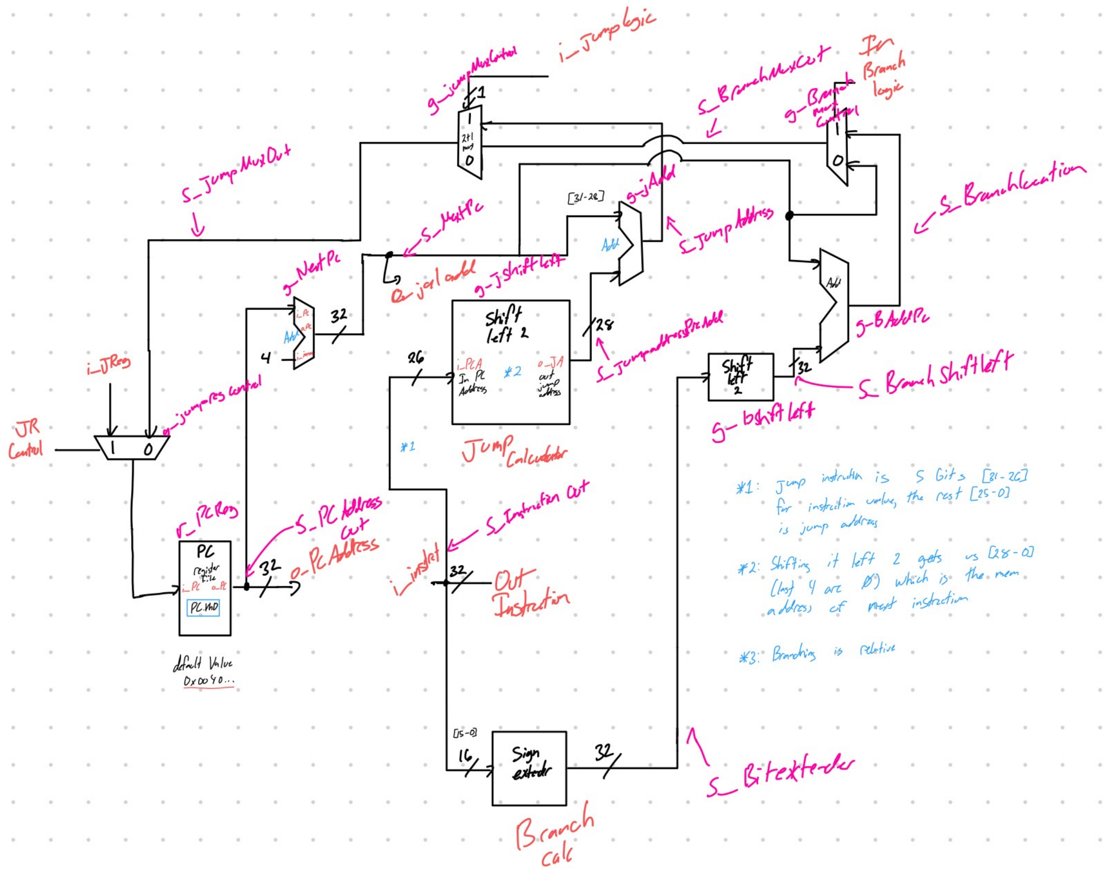
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

The control logic that the fetch must implement are as follows: inBranchLogic (calculated in a high level box depending on branch equals and branch not equals), jump logic, and jump register logic.

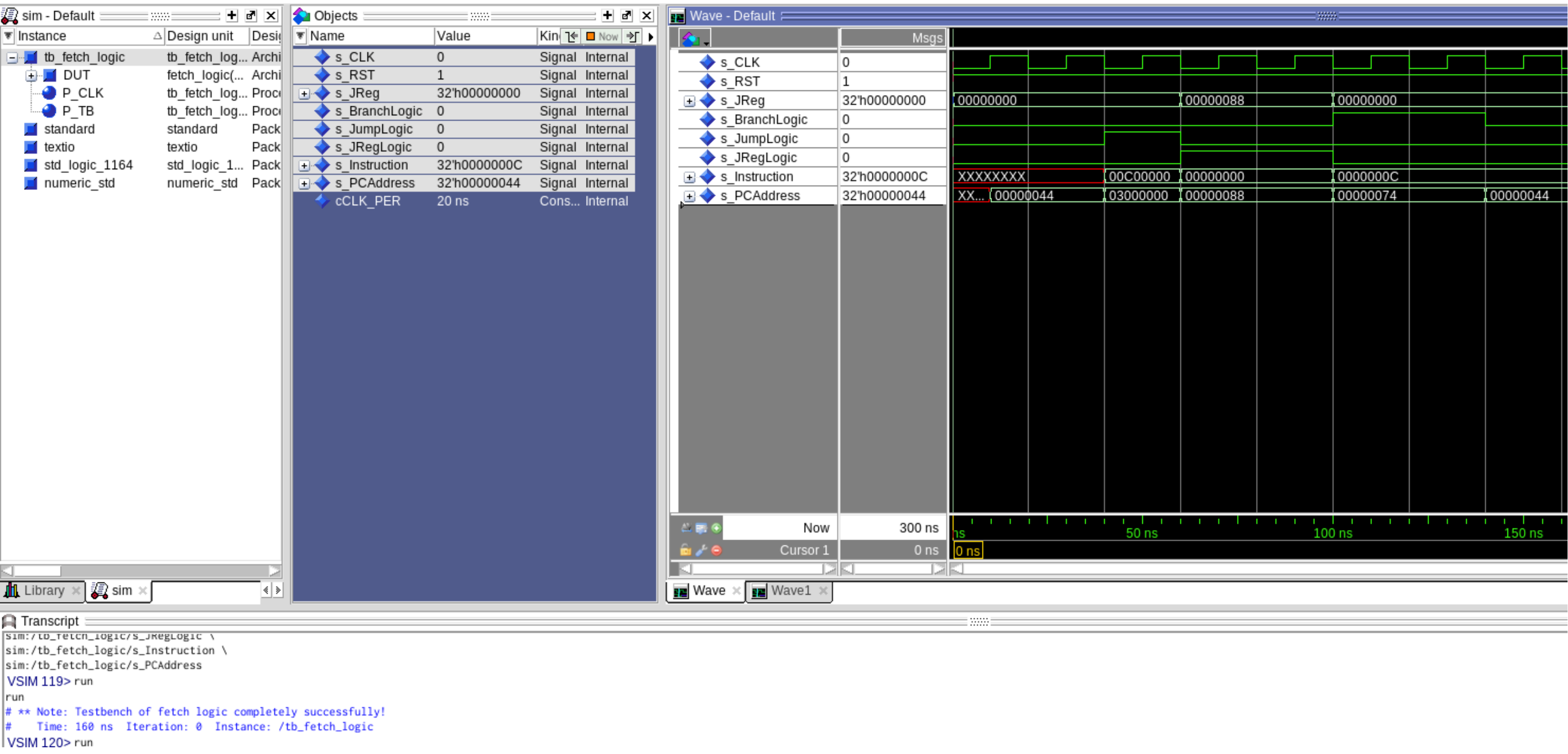
[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?



```vhdl
-- Testing addiu - basic
s_opcode <= "001001";
-- Expecting: 100011000000000010
wait for cClk_per*2;
assert (s_signalsOut="100011000000000010") report "addiu basic failed" severity error;


-- Testing addiu - radnom dfunc
s_opcode <= "001001";
s_dfunc  <= "010101";
-- Expecting: 100011000000000010
wait for cClk_per*2;
assert (s_signalsOut="100011000000000010") report "addiu random dfunc failed" severity error;

-- Testing addu - basic
s_opcode <= "000000";
s_dfunc  <= "010101";
-- Expecting: 100010000000000010
wait for cClk_per*2;
assert (s_signalsOut="100010000000000010") report "addu basic failed" severity error;

-- Testing and - correct dfunc
s_opcode <= "000000";
s_dfunc <= "011000";
-- Expecting: 000010000000001000
wait for cClk_per*2;
assert (s_signalsOut="000010000000001000") report "and correct dfunc failed" severity error;

-- Testing andi - basic
s_opcode <= "001100";
-- Expecting: 100011000000001000
wait for cClk_per*2;
assert (s_signalsOut="100011000000001000") report "andi basic failed" severity error;

-- Testing lui - basic
s_opcode <= "001111";
-- Expecting: 000011000000010000
wait for cClk_per*2;
assert (s_signalsOut="000011000000010000") report "lui basic failed" severity error;

-- Testing lw - basic
```



[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.



Instance	Design unit	Desi
tb_fetch_logic	tb_fetch_log...	Archi
DUT	fetch_logic(...	Archi
P_CLK	tb_fetch_log...	Proc
P_TB	tb_fetch_log...	Proc
standard	standard	Pack
textio	textio	Pack
std_logic_1164	std_logic_1...	Pack
numeric_std	numeric_std	Pack

Name	Value	Kin	
s_CLK	0	Signal	Internal
s_RST	1	Signal	Internal
s_JReg	32'h00000000	Signal	Internal
s_BranchLogic	0	Signal	Internal
s_JumpLogic	0	Signal	Internal
s_JRegLogic	0	Signal	Internal
s_Instruction	32'h0000000C	Signal	Internal
s_PCAddress	32'h00000044	Signal	Internal
cCLK_PER	20 ns	Cons...	Internal

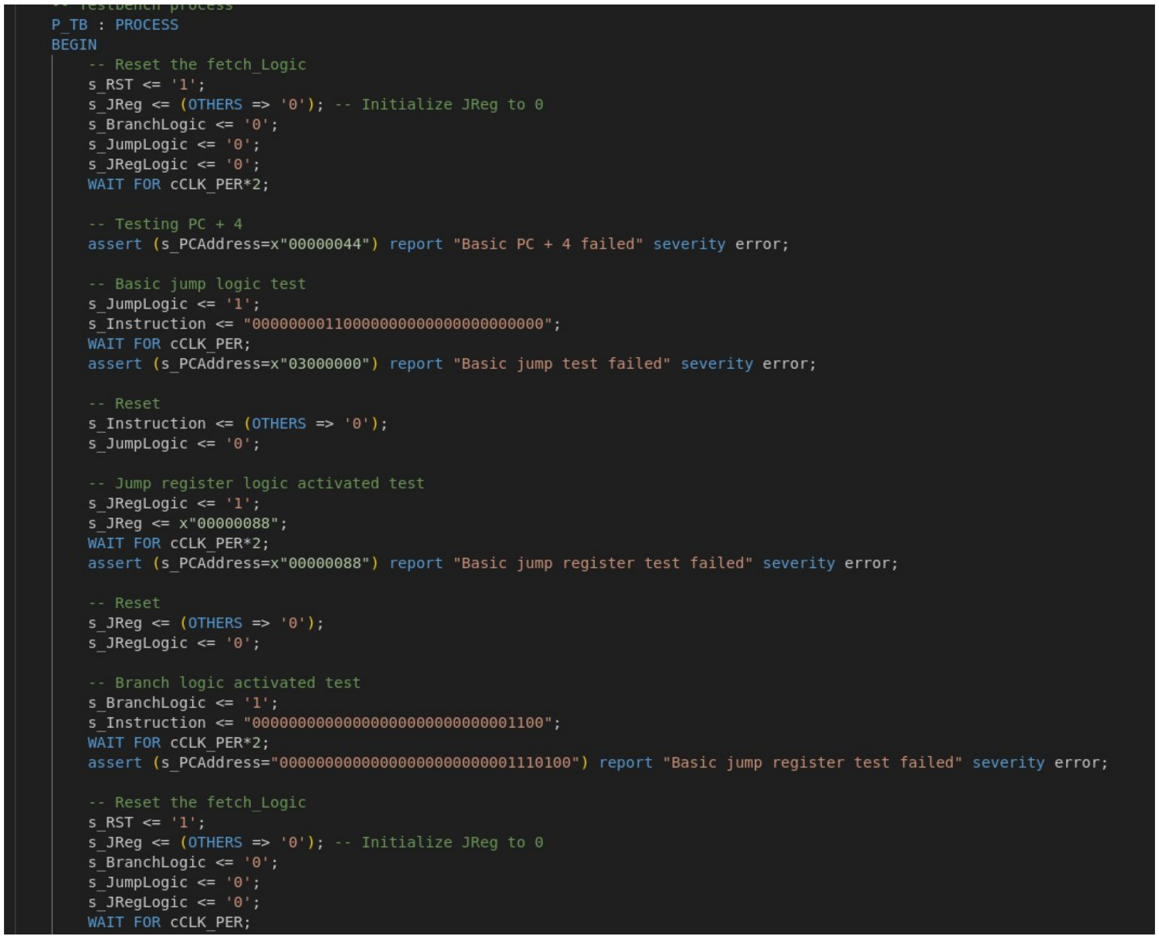
	Msgs	
s_CLK	0	
s_RST	1	
s_JReg	32'h00000000	00000000 · 00000088 · 00000000
s_BranchLogic	0	
s_JumpLogic	0	
s_JRegLogic	0	
s_Instruction	32'h0000000C	XXXXXXXX · 00C00000 · 00000000 · 0000000C
s_PCAddress	32'h00000044	XX... · 00000044 · 03000000 · 00000088 · 00000074 · 00000044

```
sim:/tb_fetch_logic/s_JRegLogic \
sim:/tb_fetch_logic/s_Instruction \
sim:/tb_fetch_logic/s_PCAddress
VSIM 119> run
run
# ** Note: Testbench of fetch logic completely successfully!
#    Time: 160 ns  Iteration: 0  Instance: /tb_fetch_logic
VSIM 120> run
```

In these two images you can see that the fetch logic is performing correctly since it does not report any errors when running the test bench file in the console. The instructions were calculated by hand using the diagram above to ensure that the output would match what was expected.
In the first image you can see the output of questasim
In the second image you can see a part of the test code used (the rest is about the same but with different values and is provided

ALU

[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 when doing srl, it shifts the amount to the right, adding a 0 to the left most side filling in empty spots. Sra shifts the number to the right, but wraps around, for example if you have 1001 and you sra it then you would get 1100. If you did 1001 with srl you would get 0100.
The reason that MIPS does not have sla is because you can instead use sra to shift it the same amount, by doing the max bit size (32) – the shift amount. This will result in the same answer.

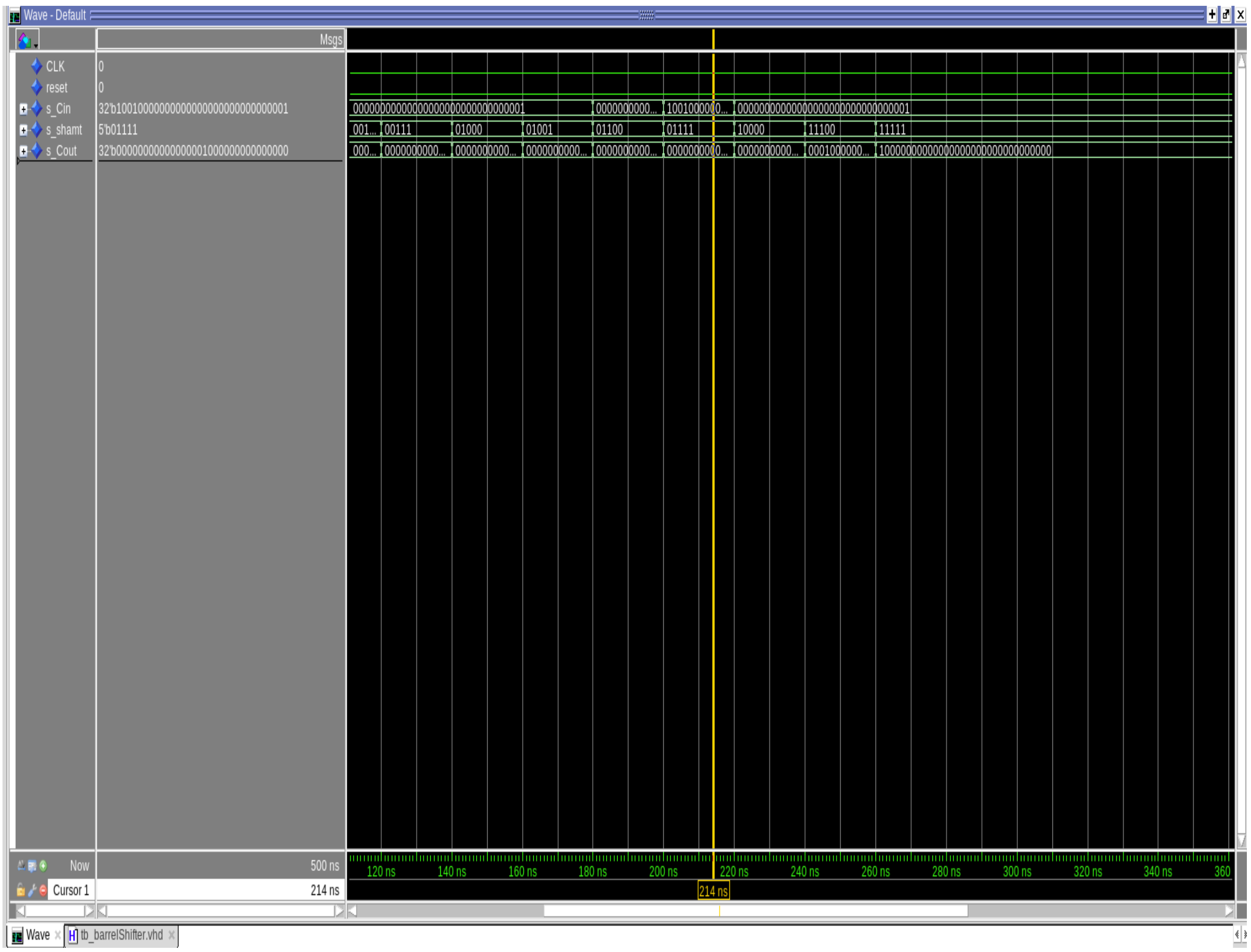


```
-- Testbench process
P_TB : PROCESS
BEGIN
    -- Reset the fetch_Logic
    s_RST <= '1';
    s_JReg <= (OTHERS => '0'); -- Initialize JReg to 0
    s_BranchLogic <= '0';
    s_JumpLogic <= '0';
    s_JRegLogic <= '0';
    WAIT FOR cCLK_PER*2;

    -- Testing PC + 4
    assert (s_PCAddress=x"00000044") report "Basic PC + 4 failed" severity error;

    -- Basic jump logic test
    s_JumpLogic <= '1';
    s_Instruction <= "00000000110000000000000000000000";
    WAIT FOR cCLK_PER;
    assert (s_PCAddress=x"03000000") report "Basic jump test failed" severity error;

    -- Reset
    s_Instruction <= (OTHERS => '0');
    s_JumpLogic <= '0';

    -- Jump register logic activated test
    s_JRegLogic <= '1';
    s_JReg <= x"00000088";
    WAIT FOR cCLK_PER*2;
    assert (s_PCAddress=x"00000088") report "Basic jump register test failed" severity error;

    -- Reset
    s_JReg <= (OTHERS => '0');
    s_JRegLogic <= '0';

    -- Branch logic activated test
    s_BranchLogic <= '1';
    s_Instruction <= "00000000000000000000000000001100";
    WAIT FOR cCLK_PER*2;
    assert (s_PCAddress="00000000000000000000000001110100") report "Basic jump register test failed" severity error;

    -- Reset the fetch_Logic
    s_RST <= '1';
    s_JReg <= (OTHERS => '0'); -- Initialize JReg to 0
    s_BranchLogic <= '0';
    s_JumpLogic <= '0';
    s_JRegLogic <= '0';
    WAIT FOR cCLK_PER;
```

[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.
The implementation of logical shift operations was quite easy, we essentially hard coded the input value of the barrel shifter to zero, so giving the new bits a value of zero every single time it is being shifted. For arithmetic shifts, we used a rather nifty method to determine the new bits going in the shift, we take the MSB of the shift right shift MSB being bit 1, since we are shifting to the right. If it were a left shift we would the MSB would be the 32nd bit. This bit either 1st or 32nd bit being chosen, determines what value we right in, so if this bit was 1, then a 1 would be written in for the newly shifted bits.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.
We actually first built a “left” barrel shifter first, and the quickly realizing that the inputs and outputs of the same barrel shifter can be changed, we can essentially create a right barrel shifter. The input bits of the barrel shifter are flipped first using a generate statement, then after shifting, these output bits are again flipped using another generate statement. The output is the a right shift.

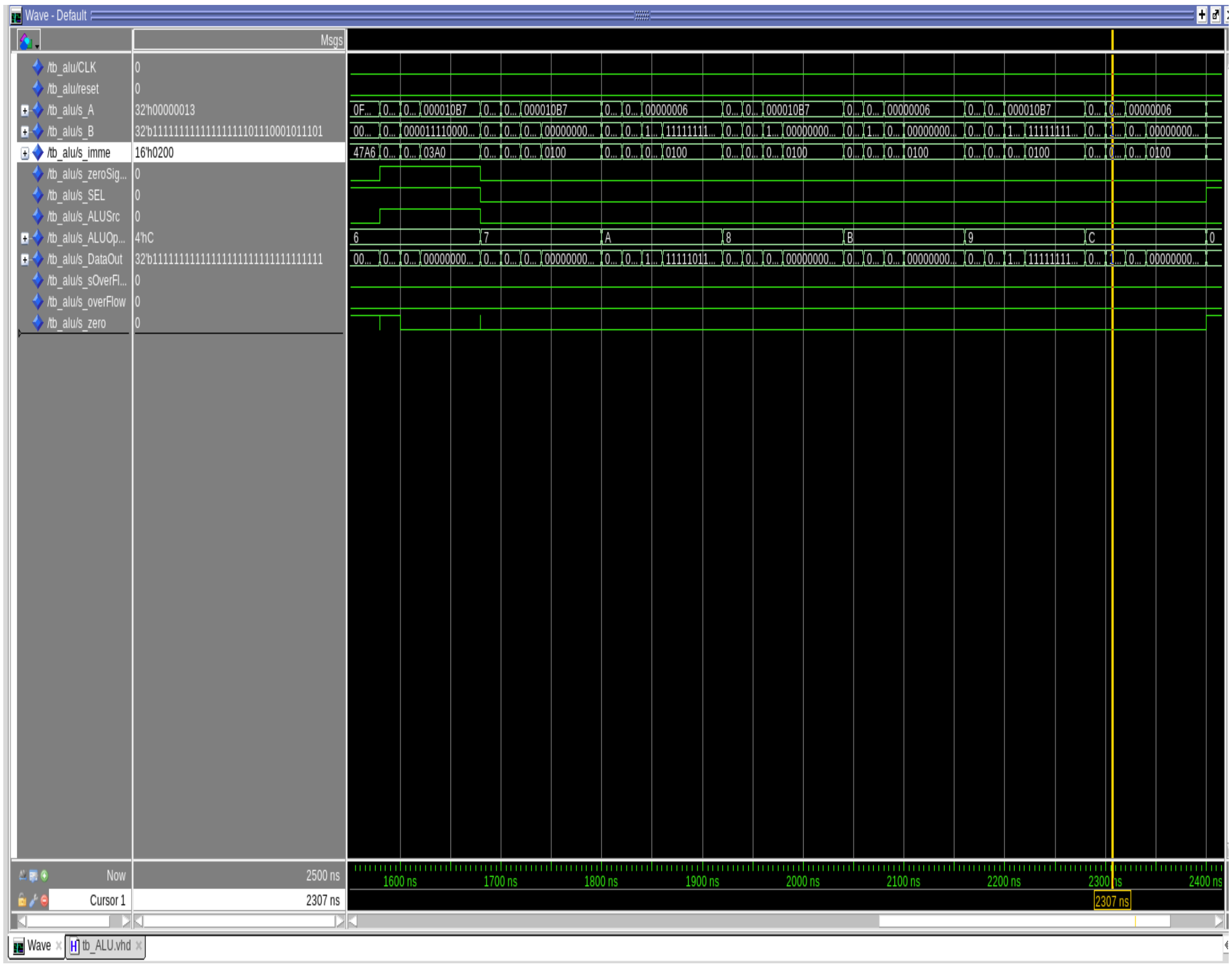
[Part 2 (c.i.4)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.



```
Wave - Default
                                             Msgs
CLK        0
reset      0
s_Cin      32'b10010000000000000000000000000001
s_shamt    5'b01111
s_Cout     32'b00000000000000001000000000000000

Now        500 ns
Cursor 1   214 ns
```

```
s_Cin    00000000000000000000000000000001 | 0000000000... | 1001000000... | 00000000000000000000000000000001
s_shamt  001... | 00111 | 01000 | 01001 | 01100 | 01111 | 10000 | 11100 | 11111
s_Cout   000... | 0000000000... | 0000000000... | 0000000000... | 0000000000... | 0000000000... | 0000000000... | 0001000000... | 10000000000000000000000000000000
```

Above is the waveform for the test bench of the Barrel shifter itself with an extensive amount of test cases. These test cases were also tested with Mars to confirm basic shifting to the left (SLL). Once we confirmed the barrel shifter works as intended, we then modified it for all other shifting instructions.



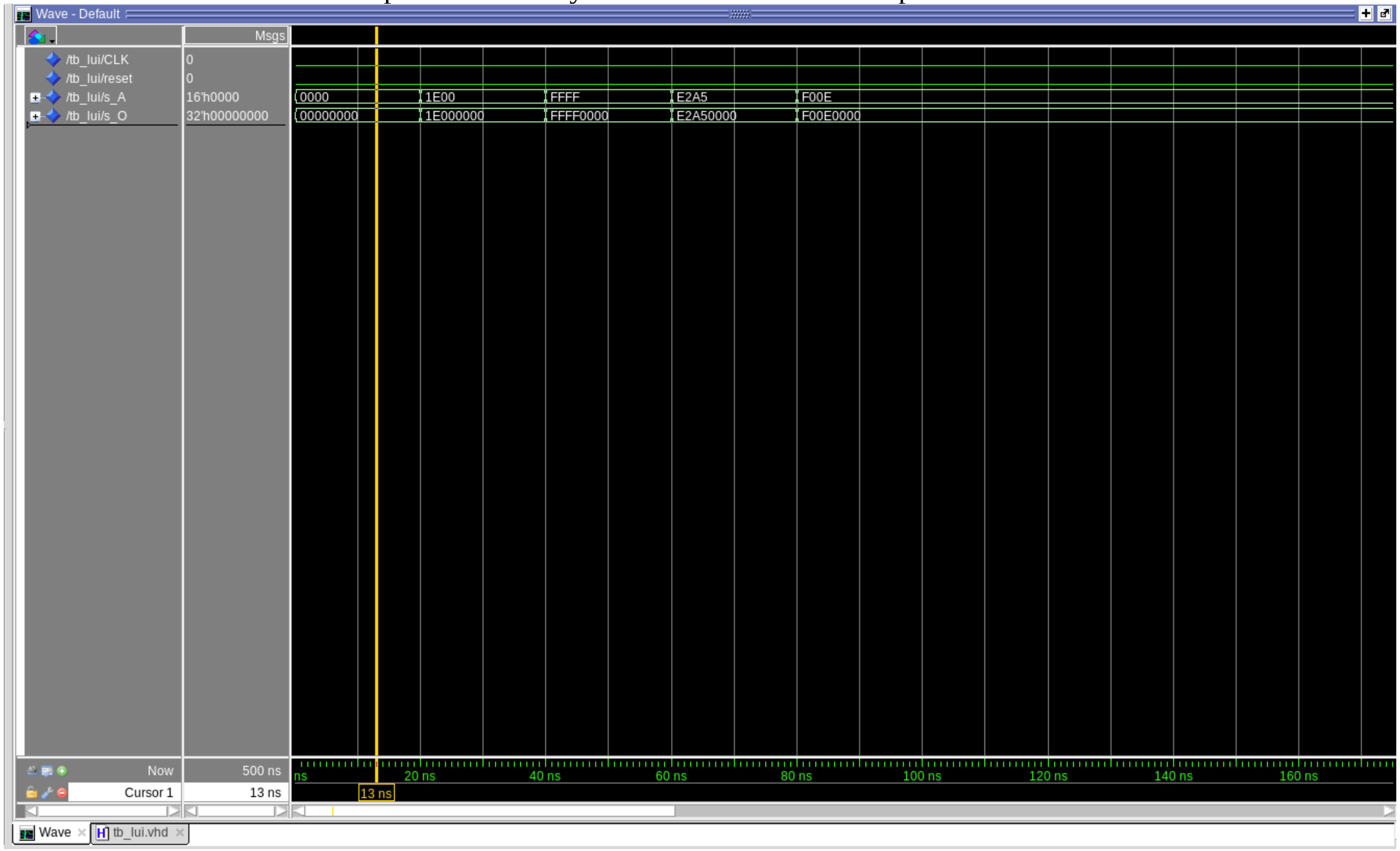
```
/tb_alu/CLK          0
/tb_alu/reset        0
/tb_alu/s_A          32'h00000013
/tb_alu/s_B          32'b11111111111111111111011100011011101
/tb_alu/s_imme       16'h0200
/tb_alu/s_zeroSig... 0
/tb_alu/s_SEL        0
/tb_alu/s_ALUSrc     0
/tb_alu/s_ALUOp...   4'hC
/tb_alu/s_DataOut    32'b11111111111111111111111111111111
/tb_alu/s_sOverFl... 0
/tb_alu/s_overFlow   0
/tb_alu/s_zero       0
Now       2500 ns
Cursor 1  2307 ns
```

Above is the waveform of a test case for SRAV, which is located inside the ALU testbench file. This shift instruction, as for all other shift instructions is tested inside of the ALU unit, as we confirmed that the barrel shifter unit alone works above. There are approximately 4 test cases for each shift instruction, as this one above tests the arithmetic portion edge case of the SRAV instruction. All of the test cases were confirmed working with MARs replicating the same instructions.

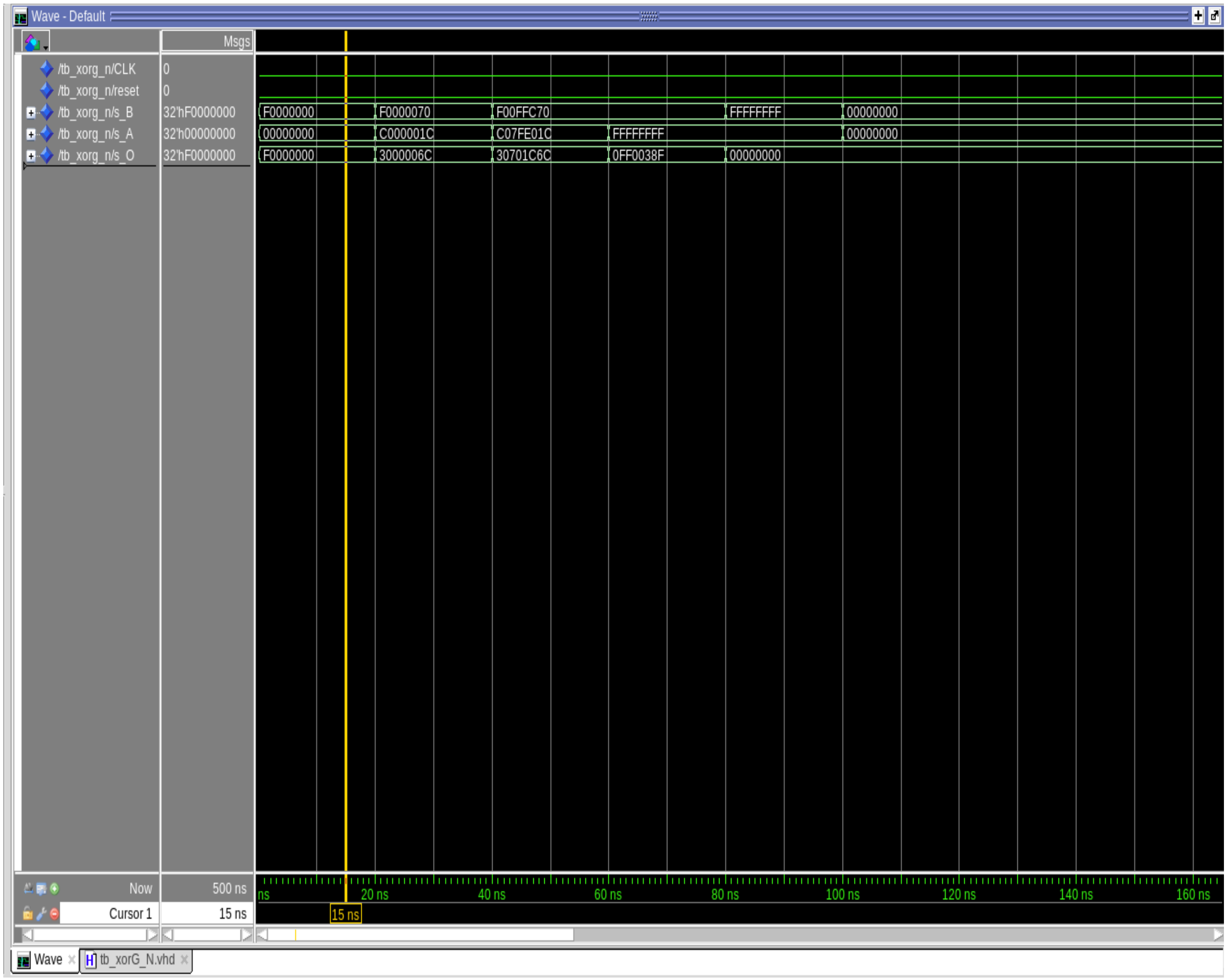
[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

My design approach for the ALU was to create a well thought out design of the ALU in paper, including what signals would be needed and approximately how many, after studying the green sheet. Other than the green sheet being used, and looking at how the ALU signals were used in the Zybook, I created the rest of it without any other resources. One design decision to used in making the ALU was to create everything using as much structural VHDL as possible, this is because if I can draw the ALU out on paper it would be extremely easy to understand and keep track of the many operations and eliminate future issues. The payoff of doing this was extremely high, once the ALU was added inside, with extensive test benches as a safe measure, no issues were found with it and all operations performed their needed tasks. Other than a one or 2 data flow and behavioral sub components, the entire ALU was created using structural VHDL.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.
To test the functionality of components inside of the ALU we first created each component in a separate VHDL file, then we created a separate test bench for each component to test the functionality of each component to confirm that it works. Below shows a few examples of our many test benches for each component.

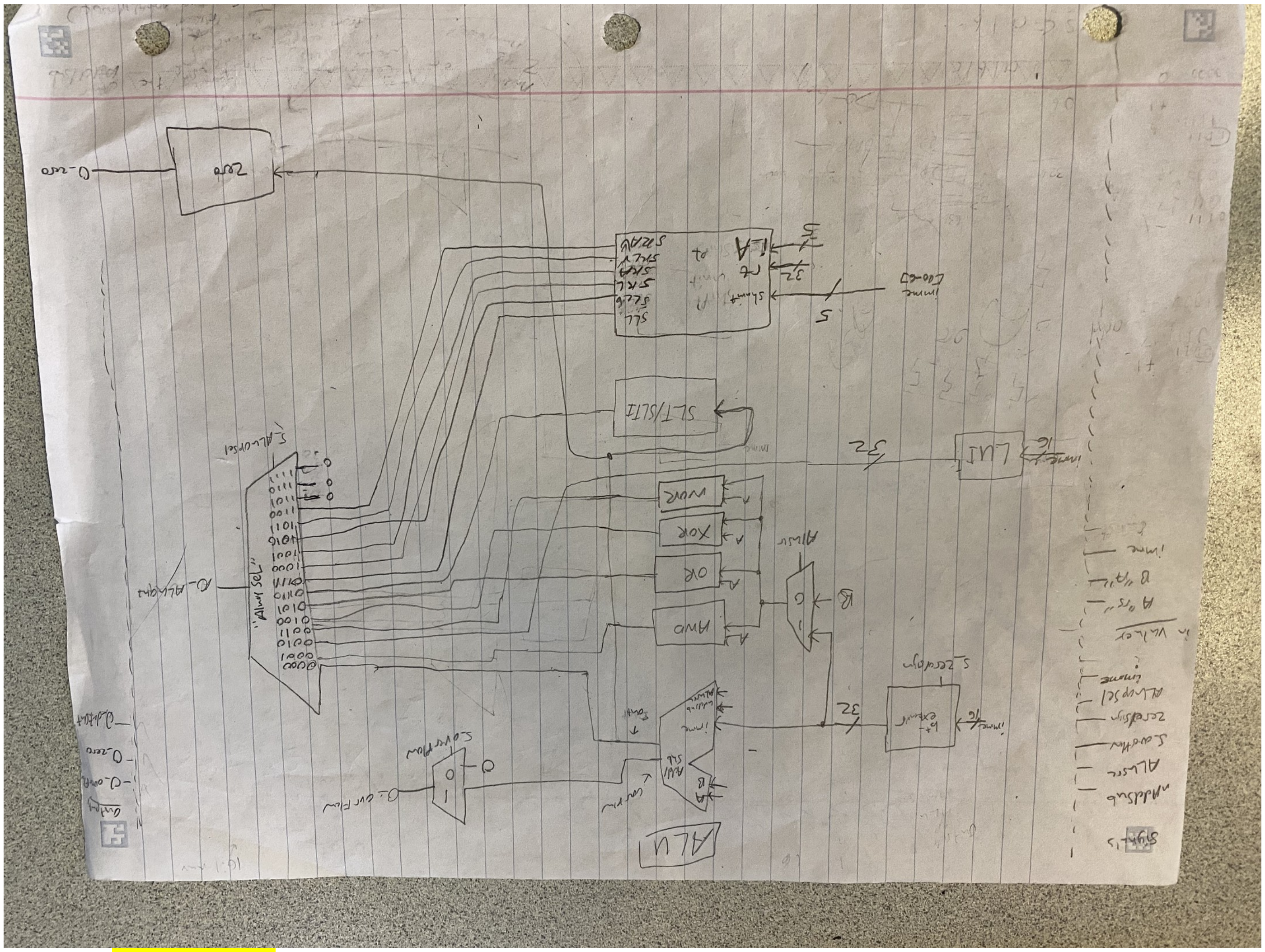


Above is the waveform of the test bench for the LUI component. As we can see, with each input the values are successfully being shifted up to the upper 16 bits of a 32 bit number. This test bench tested 5 completely different values to make sure any possible edge cases are ruled out.



Above is the waveform for the XOR component of the ALU, called “tb_xorG_N”. Right now we can see 6 separate tests on the XOR component. If we compare these tests to the same values in mars, we can see the exact same outputs, confirming the values. Also, the expected results, which do match the waveforms, are inside of the test bench VHDL file itself.

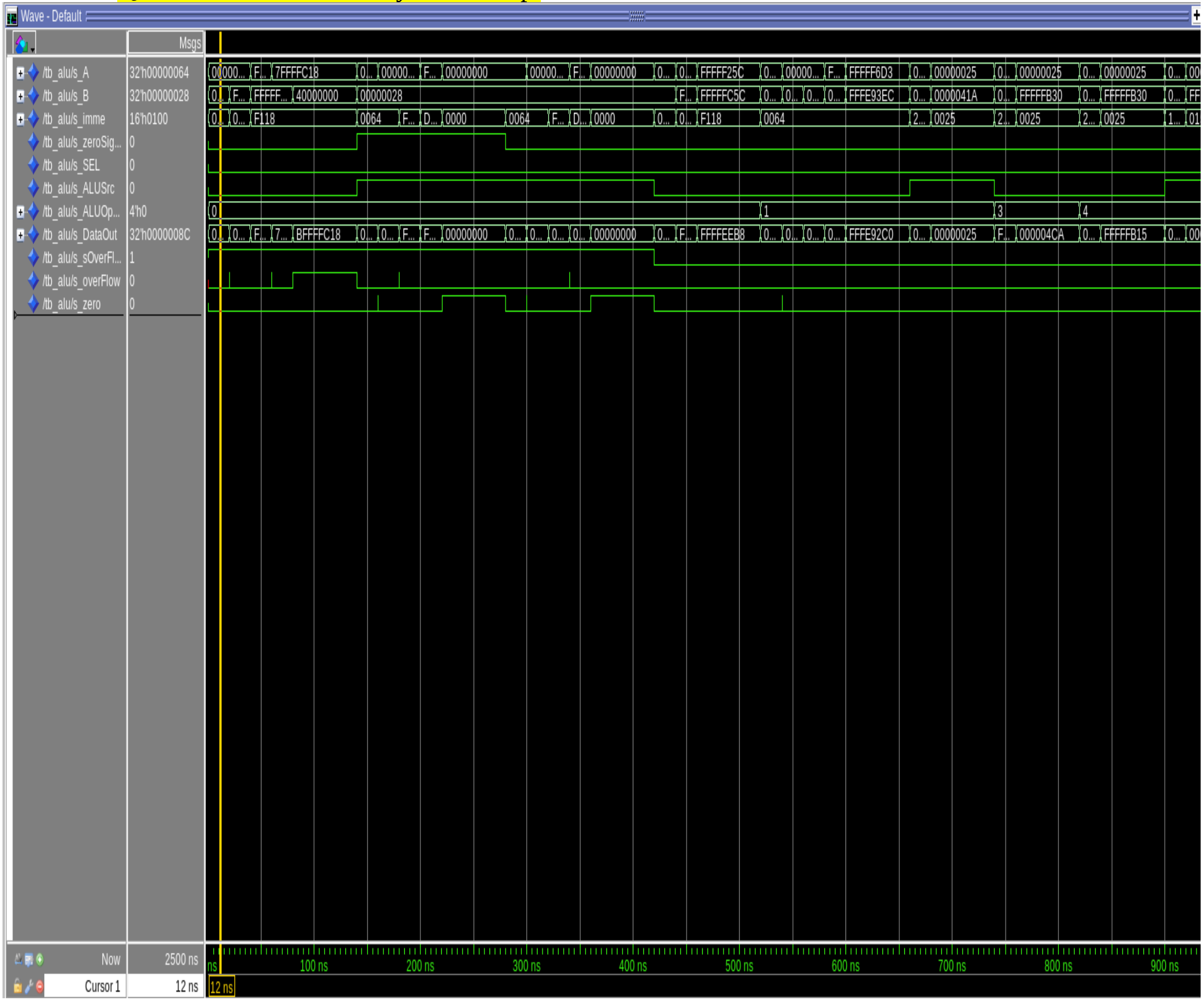
[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is slt



implemented?

Overflow is calculated using Cin of the last bit of the ripple carry adder and the final carry out of the same carry ripper adder. These 2 signals are then put into an xor gate, if the output is 1 then there is overflow, if not there is no overflow, a separate mux after this output is used to enable and disable overflow detection, which is also located inside the ALU. Zero is calculated by taking the output of the adder/subtractor unit in the ALU and then putting it inside a 32 bit or gate, then the output of this OR gate gets put inside an inverter gate. The output of the inverter gate is then zero value. SLT is implemented using behavioral vhdl by taking the data output of the adder/subtractor unit. This output is then compared, if the output is greater than or equal 0 and if the number is not negative then SLT will output 0, else it will output 1.

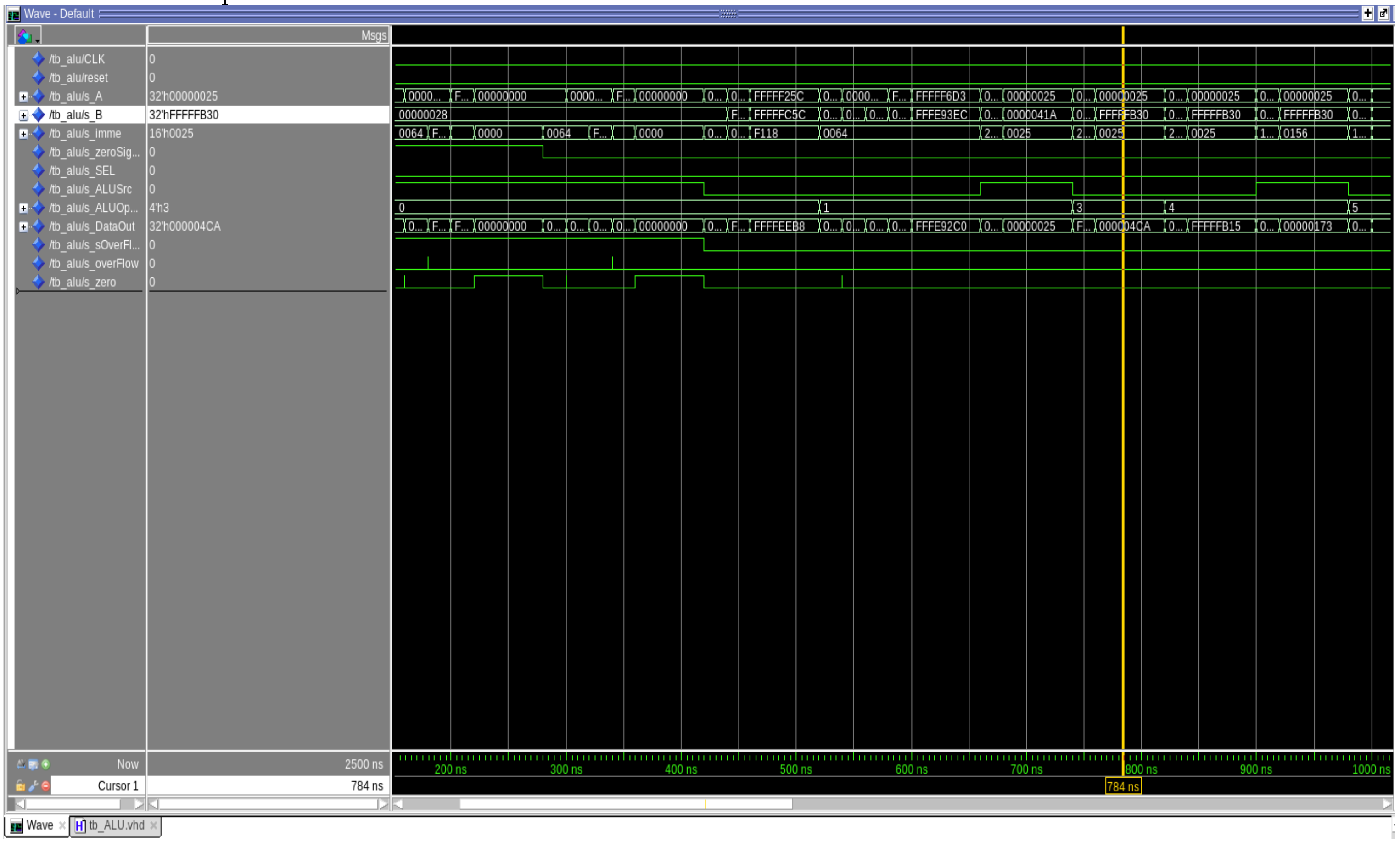
[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.



In the waveform shown above is just one portion of the entire test bench for the ALU. To make it a little easier to distinguish which operation is being tested, I created a longer delay for the last test of each instruction. As You can see above some values has a longer clock delay, these mean what I just said above, that at the end of the long delay, a new instruction will be tested, and so forth. To know which instructions are being tested, inside of the test bench, each instruction is formatted in a list of tests to make it easier to keep track of which instruction is being tested. You can also see expected output values of the registers after each test at the end of the test cases inside of the test bench VHDL code itself.

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

Our test plan for the ALU unit is extremely comprehensive, each instruction has roughly atleast 4 separate test cases, with at least 2 of these acting as edge cases. To make the ALU test even more comprehensive we compared each instruction test case with a MARs program. We first wrote the MARs program, then wrote the same input values into the ALU to simulate the program in use. After this we compared the output values of each register in order with the outputs of the ALU to confirm that the values of both of these were the same. If they were the same, then the ALU successfully simulated a few tests of that specific instruction, if not there was an error with the specific component of the ALU. This streamlined our testing of the ALU and confirmed that is was 100% functional before putting all of the components together into a single cycle processor, eliminating unit test errors coming from the ALU. All MARs unit tests of the ALU instructions is inside a separate folder with the test benches. Below are an example of one of the many test bench waveforms as we already discussed a few testbench waveforms in the questions above.



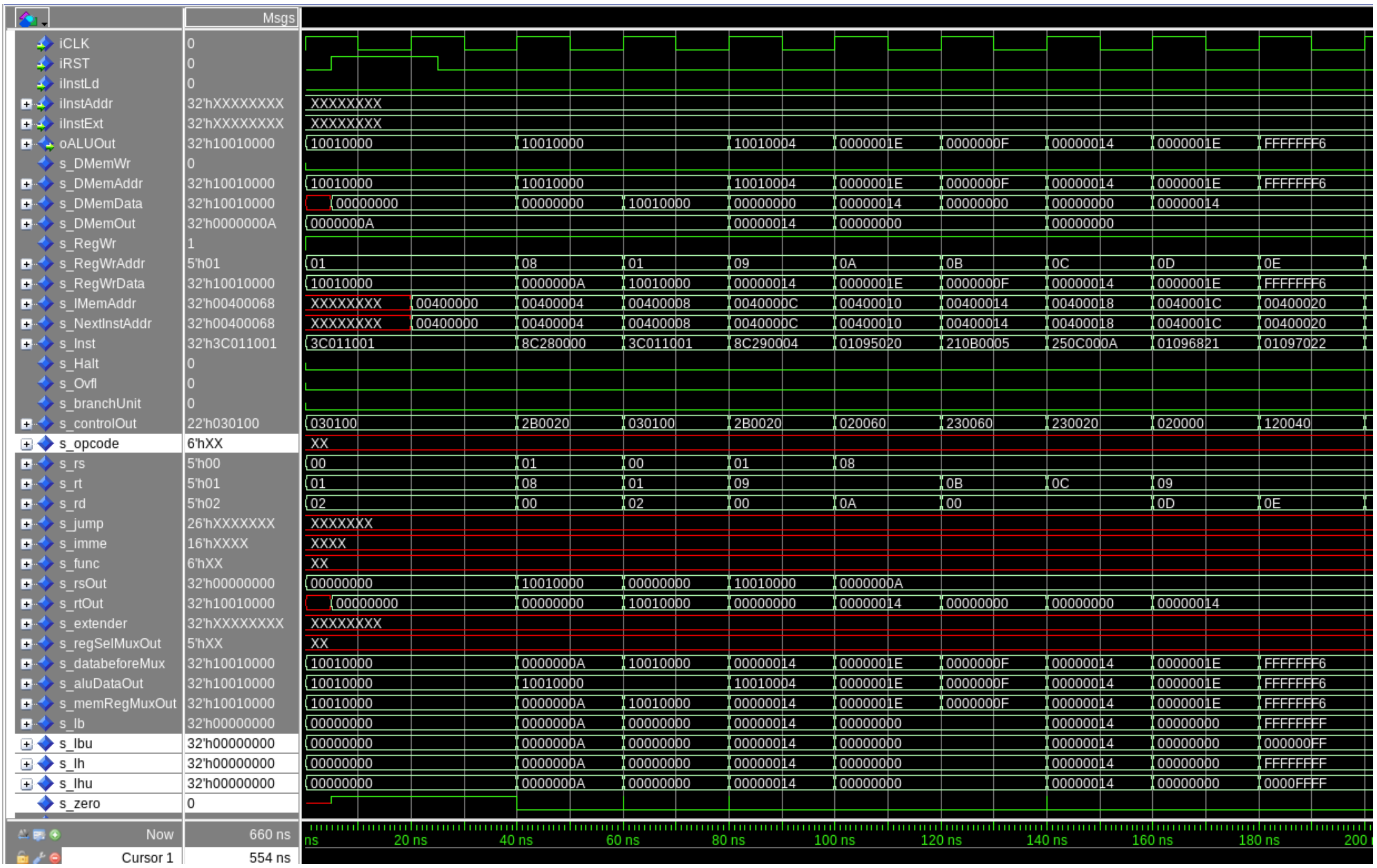
```
/tb_alu/CLK         0
/tb_alu/reset       0
/tb_alu/s_A         32'h00000025
/tb_alu/s_B         32'hFFFFFB30
/tb_alu/s_imme      16'h0025
/tb_alu/s_zeroSig...  0
/tb_alu/s_SEL       0
/tb_alu/s_ALUSrc    0
/tb_alu/s_ALUOp...  4'h3
/tb_alu/s_DataOut   32'h000004CA
/tb_alu/s_sOverFl...  0
/tb_alu/s_overFlow  0
/tb_alu/s_zero      0
```

Above is the test bench for the NOR component of the ALU. As you can see we have multiply test cases for this rather simple component. The output values are confirmed to be correct with MARs. You can see all expected outputs inside of the ALU VHDL file itself.

Testing

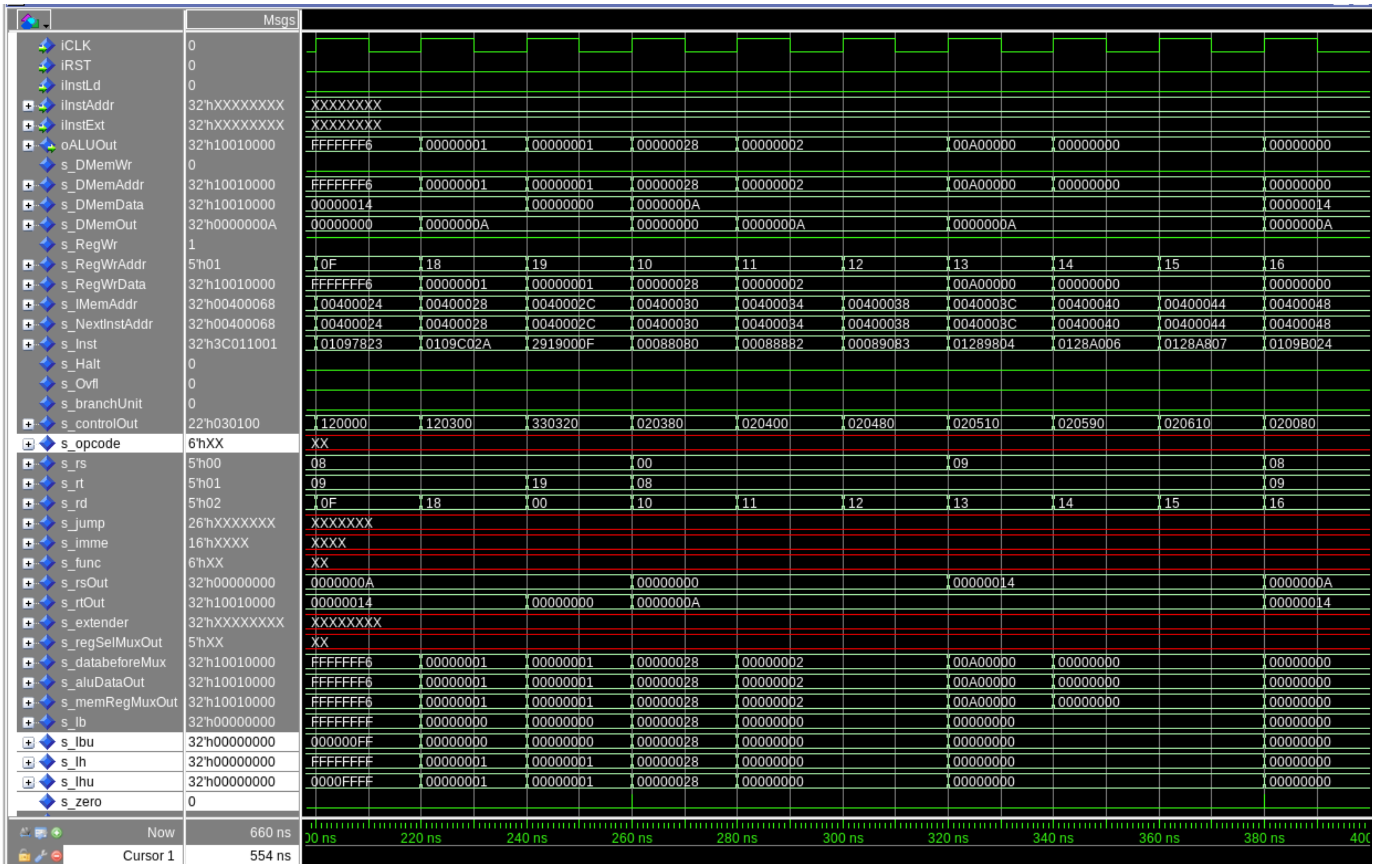
[Part 3] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[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.



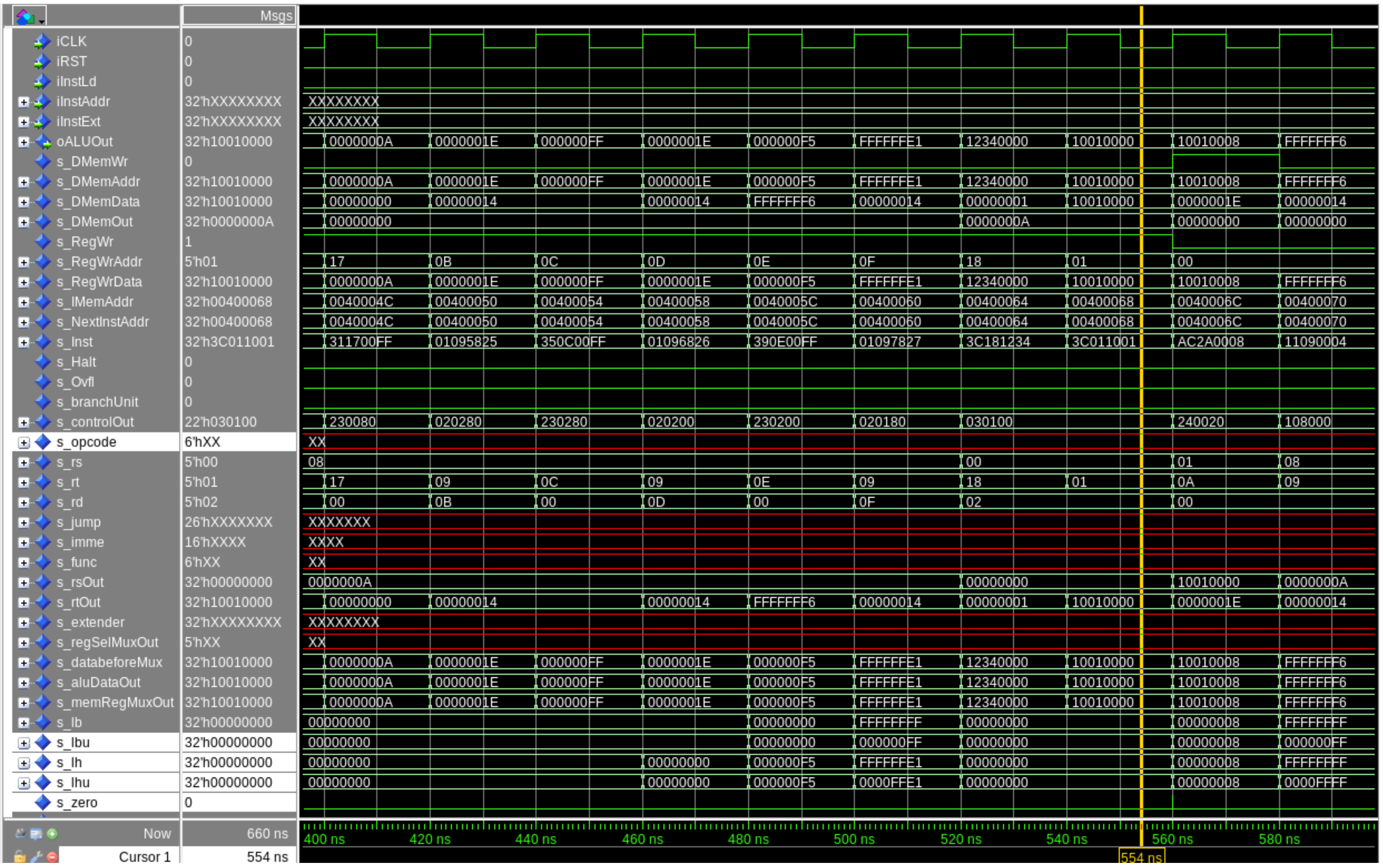
	Msgs
iCLK	0
iRST	0
iInstLd	0
iInstAddr	32'hXXXXXXXX
iInstExt	32'hXXXXXXXX
oALUOut	32'h10010000
s_DMemWr	0
s_DMemAddr	32'h10010000
s_DMemData	32'h10010000
s_DMemOut	32'h0000000A
s_RegWr	1
s_RegWrAddr	5'h01
s_RegWrData	32'h10010000
s_IMemAddr	32'h00400068
s_NextInstAddr	32'h00400068
s_Inst	32'h3C011001
s_Halt	0
s_Ovfl	0
s_branchUnit	0
s_controlOut	22'h030100
s_opcode	6'hXX
s_rs	5'h00
s_rt	5'h01
s_rd	5'h02
s_jump	26'hXXXXXXX
s_imme	16'hXXXX
s_func	6'hXX
s_rsOut	32'h00000000
s_rtOut	32'h10010000
s_extender	32'hXXXXXXXX
s_regSelMuxOut	5'hXX
s_databeforeMux	32'h10010000
s_aluDataOut	32'h10010000
s_memRegMuxOut	32'h10010000
s_lb	32'h00000000
s_lbu	32'h00000000
s_lh	32'h00000000
s_lhu	32'h00000000
s_zero	0
Now	660 ns
Cursor 1	554 ns

In this screenshot you can see it starts by loading data into the two registers, which are correctly showing in rs as well as the data in. It then goes on to use add, addi, addiu, addu, sub, subu, and slt. The result of add 10+20 in $t2, then 10+5 into $t3, then 10_10 into $t4, then 10-20 into $t6, and again with the unsigned into $t7. Throughout this you can see the instruction address is going up in the correct amounts as well as setting the right control logic as well as register write address.

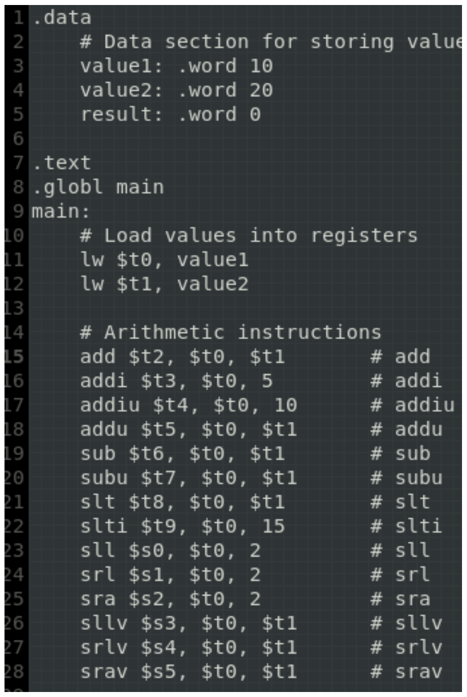


```
                     Msgs
iCLK                 0
iRST                 0
iInstLd              0
iInstAddr            32'hXXXXXXXX     XXXXXXXX
iInstExt             32'hXXXXXXXX     XXXXXXXX
oALUOut              32'h10010000     FFFFFFF6 00000001 00000001 00000028 00000002 00A00000 00000000 00000000
s_DMemWr             0
s_DMemAddr           32'h10010000     FFFFFFF6 00000001 00000001 00000028 00000002 00A00000 00000000 00000000
s_DMemData           32'h10010000     00000014 00000000 0000000A 00000014
s_DMemOut            32'h0000000A     00000000 0000000A 00000000 0000000A 0000000A 0000000A
s_RegWr              1
s_RegWrAddr          5'h01            0F 18 19 10 11 12 13 14 15 16
s_RegWrData          32'h10010000     FFFFFFF6 00000001 00000001 00000028 00000002 00A00000 00000000 00000000
s_IMemAddr           32'h00400068     00400024 00400028 0040002C 00400030 00400034 00400038 0040003C 00400040 00400044 00400048
s_NextInstAddr       32'h00400068     00400024 00400028 0040002C 00400030 00400034 00400038 0040003C 00400040 00400044 00400048
s_Inst               32'h3C011001     01097823 0109C02A 2919000F 00088080 00088882 00089083 01289804 0128A006 0128A807 0109B024
s_Halt               0
s_Ovfl               0
s_branchUnit         0
s_controlOut         22'h030100       120000 120300 330320 020380 020400 020480 020510 020590 020610 020080
s_opcode             6'hXX            XX
s_rs                 5'h00            08 00 09 08
s_rt                 5'h01            09 19 08 09
s_rd                 5'h02            0F 18 00 10 11 12 13 14 15 16
s_jump               26'hXXXXXXX      XXXXXXX
s_imme               16'hXXXX         XXXX
s_func               6'hXX            XX
s_rsOut              32'h00000000     0000000A 00000000 00000014 0000000A
s_rtOut              32'h10010000     00000014 00000000 0000000A 00000014
s_extender           32'hXXXXXXXX     XXXXXXXX
s_regSelMuxOut       5'hXX            XX
s_databeforeMux      32'h10010000     FFFFFFF6 00000001 00000001 00000028 00000002 00A00000 00000000 00000000
s_aluDataOut         32'h10010000     FFFFFFF6 00000001 00000001 00000028 00000002 00A00000 00000000 00000000
s_memRegMuxOut       32'h10010000     FFFFFFF6 00000001 00000001 00000028 00000002 00A00000 00000000 00000000
s_lb                 32'h00000000     FFFFFFFF 00000000 00000000 00000028 00000000 00000000 00000000
s_lbu                32'h00000000     000000FF 00000000 00000000 00000028 00000000 00000000 00000000
s_lh                 32'h00000000     FFFFFFFF 00000001 00000001 00000028 00000000 00000000 00000000
s_lhu                32'h00000000     0000FFFF 00000001 00000001 00000028 00000000 00000000 00000000
s_zero               0

Now                  660 ns           00 ns  220 ns  240 ns  260 ns  280 ns  300 ns  320 ns  340 ns  360 ns  380 ns  400
Cursor 1             554 ns
```

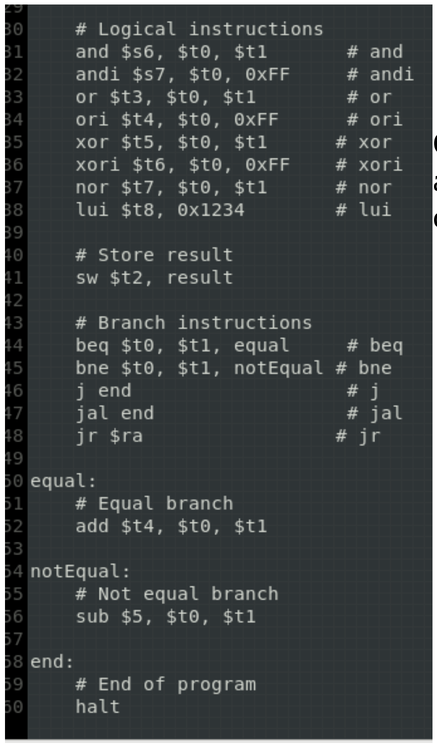
This continues to go on to execute the next commands in the assembly file, correctly going up in the instruction address and control logic output to match the instructions going in.



	Msgs
iCLK	0
iRST	0
iInstLd	0
iInstAddr	32'hXXXXXXXX
iInstExt	32'hXXXXXXXX
oALUOut	32'h10010000
s_DMemWr	0
s_DMemAddr	32'h10010000
s_DMemData	32'h10010000
s_DMemOut	32'h0000000A
s_RegWr	1
s_RegWrAddr	5'h01
s_RegWrData	32'h10010000
s_IMemAddr	32'h00400068
s_NextInstAddr	32'h00400068
s_Inst	32'h3C011001
s_Halt	0
s_Ovfl	0
s_branchUnit	0
s_controlOut	22'h030100
s_opcode	6'hXX
s_rs	5'h00
s_rt	5'h01
s_rd	5'h02
s_jump	26'hXXXXXXX
s_imme	16'hXXXX
s_func	6'hXX
s_rsOut	32'h00000000
s_rtOut	32'h10010000
s_extender	32'hXXXXXXXX
s_regSelMuxOut	5'hXX
s_databeforeMux	32'h10010000
s_aluDataOut	32'h10010000
s_memRegMuxOut	32'h10010000
s_lb	32'h00000000
s_lbu	32'h00000000
s_lh	32'h00000000
s_lhu	32'h00000000
s_zero	0



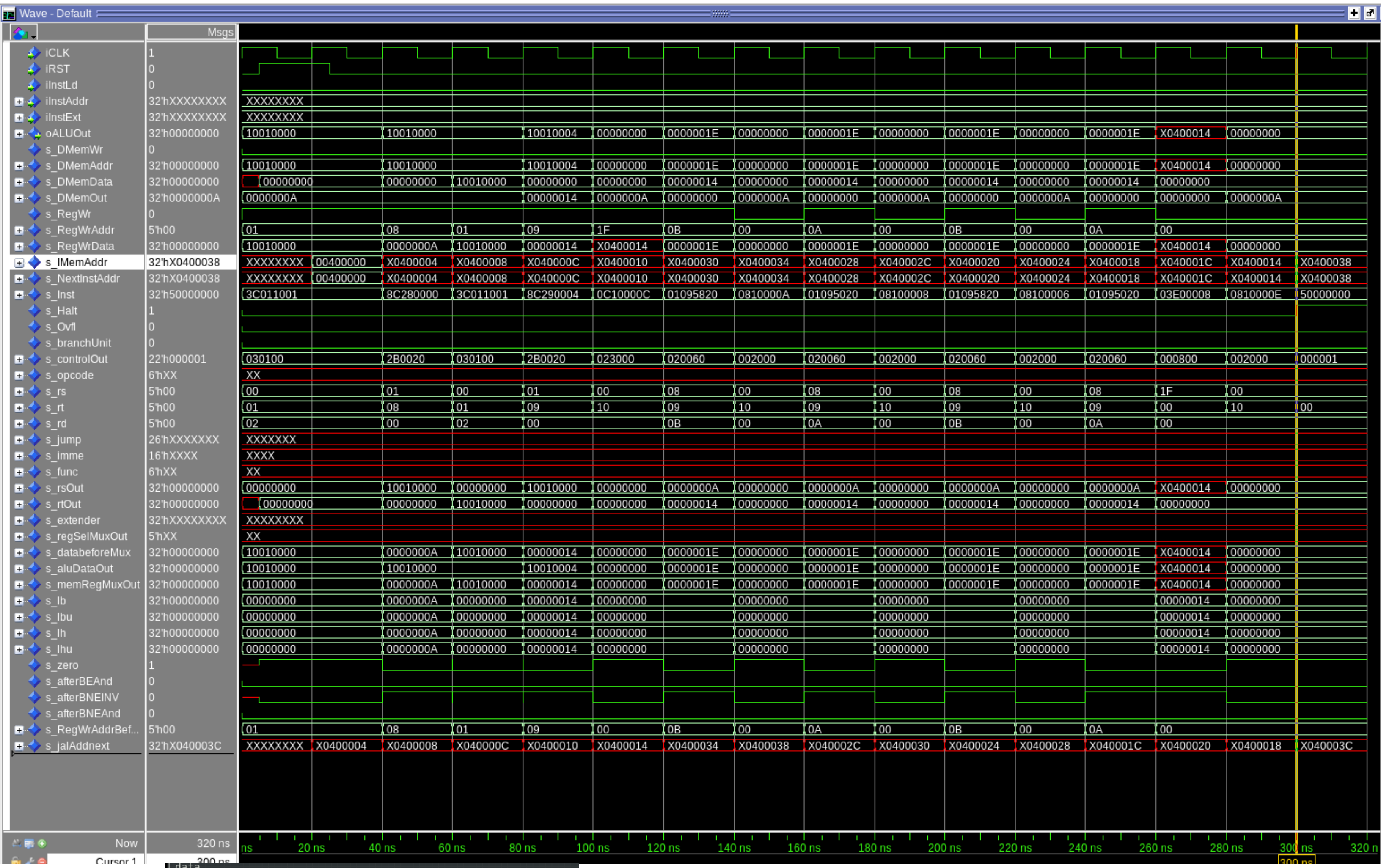
```
.data
    # Data section for storing value
    value1: .word 10
    value2: .word 20
    result: .word 0

.text
.globl main
main:
    # Load values into registers
    lw $t0, value1
    lw $t1, value2

    # Arithmetic instructions
    add $t2, $t0, $t1        # add
    addi $t3, $t0, 5         # addi
    addiu $t4, $t0, 10       # addiu
    addu $t5, $t0, $t1       # addu
    sub $t6, $t0, $t1        # sub
    subu $t7, $t0, $t1       # subu
    slt $t8, $t0, $t1        # slt
    slti $t9, $t0, 15        # slti
    sll $s0, $t0, 2          # sll
    srl $s1, $t0, 2          # srl
    sra $s2, $t0, 2          # sra
    sllv $s3, $t0, $t1       # sllv
    srlv $s4, $t0, $t1       # srlv
    srav $s5, $t0, $t1       # srav
```



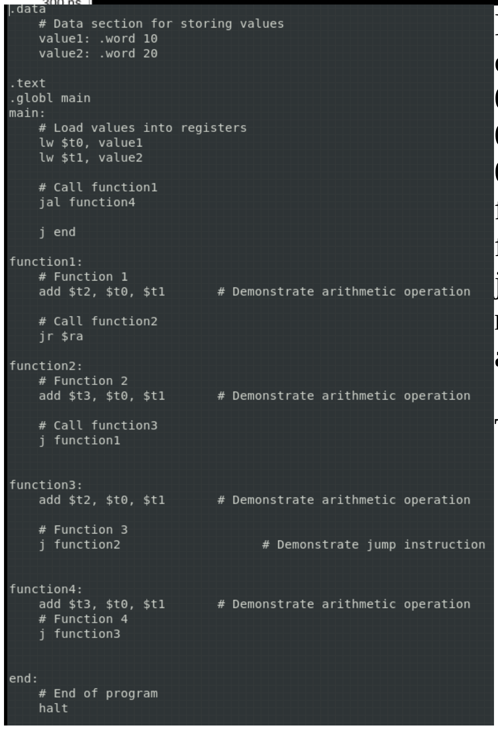
```

    # Logical instructions
    and $s6, $t0, $t1       # and
    andi $s7, $t0, 0xFF     # andi
    or $t3, $t0, $t1        # or
    ori $t4, $t0, 0xFF      # ori
    xor $t5, $t0, $t1      # xor
    xori $t6, $t0, 0xFF    # xori
    nor $t7, $t0, $t1      # nor
    lui $t8, 0x1234        # lui

    # Store result
    sw $t2, result

    # Branch instructions
    beq $t0, $t1, equal     # beq
    bne $t0, $t1, notEqual # bne
    j end                   # j
    jal end                 # jal
    jr $ra                 # jr

equal:
    # Equal branch
    add $t4, $t0, $t1

notEqual:
    # Not equal branch
    sub $5, $t0, $t1

end:
    # End of program
    halt
```

On the left you can see the full assembly code used to test all the commands.

[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.



	Msgs
iCLK	1
iRST	0
iInstLd	0
iInstAddr	32'hXXXXXXXX
iInstExt	32'hXXXXXXXX
oALUOut	32'h00000000
s_DMemWr	0
s_DMemAddr	32'h00000000
s_DMemData	32'h00000000
s_DMemOut	32'h0000000A
s_RegWr	0
s_RegWrAddr	5'h00
s_RegWrData	32'h00000000
s_IMemAddr	32'hX0400038
s_NextInstAddr	32'hX0400038
s_Inst	32'h50000000
s_Halt	1
s_Ovfl	0
s_branchUnit	0
s_controlOut	22'h000001
s_opcode	6'hXX
s_rs	5'h00
s_rt	5'h00
s_rd	5'h00
s_jump	26'hXXXXXXX
s_imme	16'hXXXX
s_func	6'hXX
s_rsOut	32'h00000000
s_rtOut	32'h00000000
s_extender	32'hXXXXXXXX
s_regSelMuxOut	5'hXX
s_databeforeMux	32'h00000000
s_aluDataOut	32'h00000000
s_memRegMuxOut	32'h00000000
s_lb	32'h00000000
s_lbu	32'h00000000
s_lh	32'h00000000
s_lhu	32'h00000000
s_zero	1
s_afterBEAnd	0
s_afterBNEINV	0
s_afterBNEAnd	0
s_RegWrAddrBef...	5'h00
s_jalAddnext	32'hX040003C

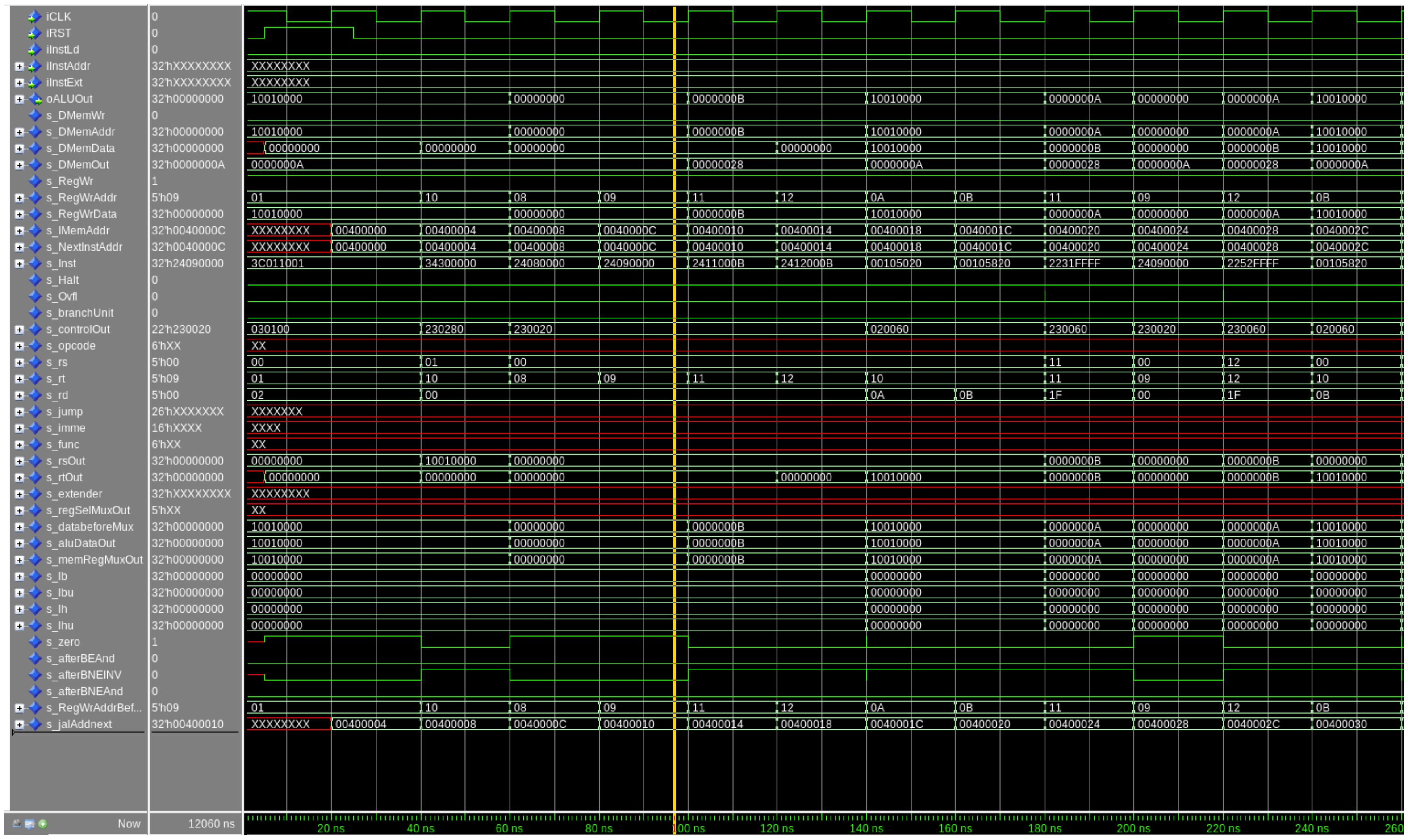


```
.data
    # Data section for storing values
    value1: .word 10
    value2: .word 20

.text
.globl main
main:
    # Load values into registers
    lw $t0, value1
    lw $t1, value2

    # Call function1
    jal function4

    j end

function1:
    # Function 1
    add $t2, $t0, $t1       # Demonstrate arithmetic operation

    # Call function2
    jr $ra

function2:
    # Function 2
    add $t3, $t0, $t1       # Demonstrate arithmetic operation

    # Call function3
    j function1


function3:
    add $t2, $t0, $t1       # Demonstrate arithmetic operation

    # Function 3
    j function2                   # Demonstrate jump instruction


function4:
    add $t3, $t0, $t1       # Demonstrate arithmetic operation
    # Function 4
    j function3


end:
    # End of program
    halt
```

In this waveform you can see that the processor is correctly jumping to the last function (address 0400030) and then does an add, jumps back to 0400028, then does another add, then jumps back to 0400020, then back one more time to 0400018, then finally returning to the jal command, and jumping one final time to halt. This shows that the jumping is jumping deep into the call stack (4 times) and is also retaining the return address and returning to the correct address.

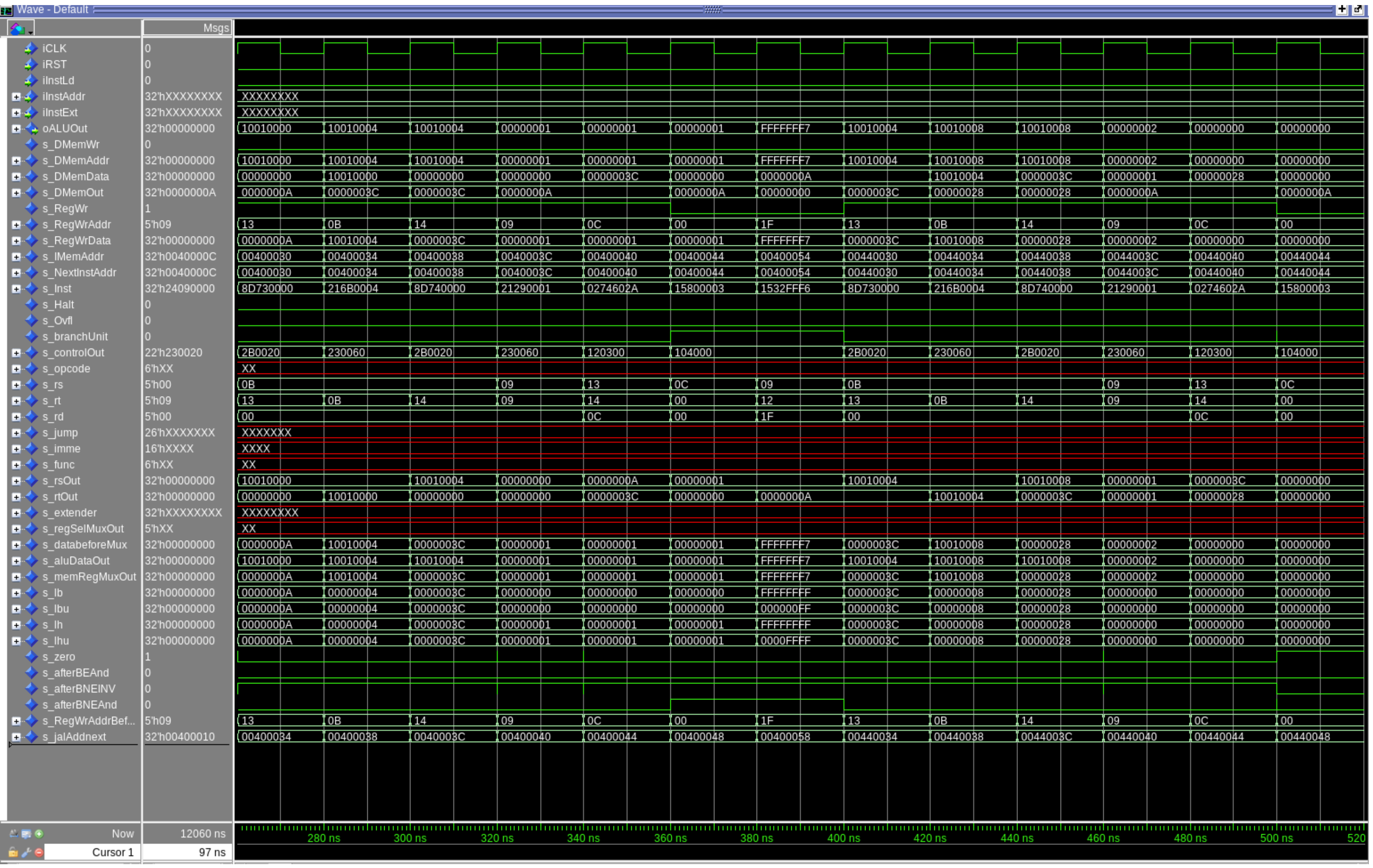
This also matches the assembly code on the left side.

[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.



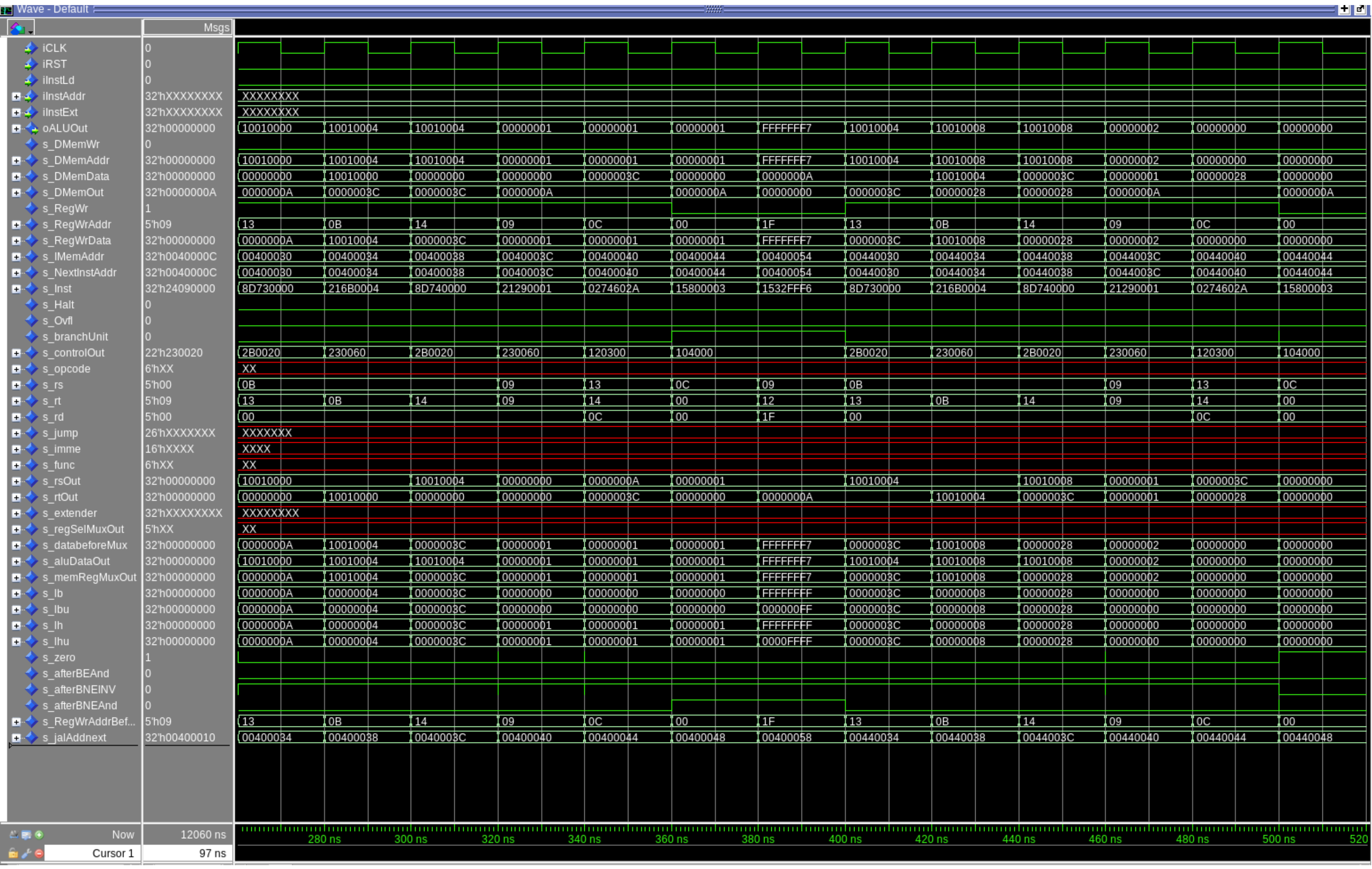
iCLK	0
iRST	0
iInstLd	0
iInstAddr	32'hXXXXXXXX
iInstExt	32'hXXXXXXXX
oALUOut	32'h00000000
s_DMemWr	0
s_DMemAddr	32'h00000000
s_DMemData	32'h00000000
s_DMemOut	32'h0000000A
s_RegWr	1
s_RegWrAddr	5'h09
s_RegWrData	32'h00000000
s_IMemAddr	32'h0040000C
s_NextInstAddr	32'h0040000C
s_Inst	32'h24090000
s_Halt	0
s_Ovfl	0
s_branchUnit	0
s_controlOut	22'h230020
s_opcode	6'hXX
s_rs	5'h00
s_rt	5'h09
s_rd	5'h00
s_jump	26'hXXXXXXX
s_imme	16'hXXXX
s_func	6'hXX
s_rsOut	32'h00000000
s_rtOut	32'h00000000
s_extender	32'hXXXXXXXX
s_regSelMuxOut	5'hXX
s_databeforeMux	32'h00000000
s_aluDataOut	32'h00000000
s_memRegMuxOut	32'h00000000
s_lb	32'h00000000
s_lbu	32'h00000000
s_lh	32'h00000000
s_lhu	32'h00000000
s_zero	1
s_afterBEAnd	0
s_afterBNEINV	0
s_afterBNEAnd	0
s_RegWrAddrBef...	5'h09
s_jalAddnext	32'h00400010
Now	12060 ns

0-260



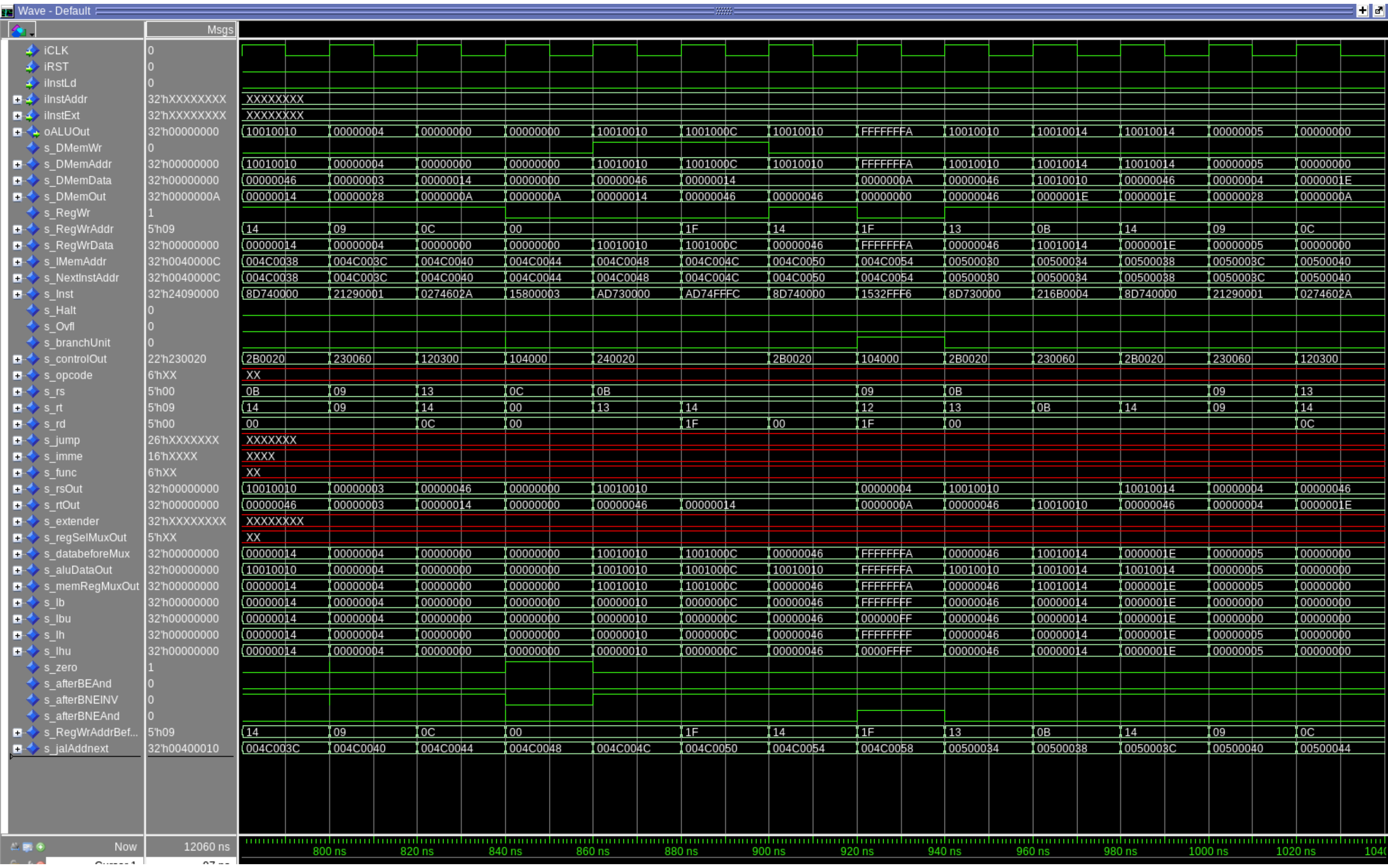
```
Wave - Default
                              Msgs
iCLK             0
iRST             0
iInstLd          0
iInstAddr        32'hXXXXXXXX      XXXXXXXX
iInstExt         32'hXXXXXXXX      XXXXXXXX
oALUOut          32'h00000000      10010000 10010004 10010004 00000001 00000001 00000001 FFFFFFF7 10010004 10010008 10010008 00000002 00000000 00000000
s_DMemWr         0
s_DMemAddr       32'h00000000      10010000 10010004 10010004 00000001 00000001 00000001 FFFFFFF7 10010004 10010008 10010008 00000002 00000000 00000000
s_DMemData       32'h00000000      00000000 10010000 00000000 00000000 0000003C 00000000 0000000A 10010004 0000003C 00000001 00000028 00000000
s_DMemOut        32'h0000000A      0000000A 0000003C 0000003C 0000000A 0000000A 00000000 0000003C 00000028 00000028 0000000A 0000000A
s_RegWr          1
s_RegWrAddr      5'h09             13 0B 14 09 0C 00 1F 13 0B 14 09 0C 00
s_RegWrData      32'h00000000      0000000A 10010004 0000003C 00000001 00000001 00000001 FFFFFFF7 0000003C 10010008 00000028 00000002 00000000 00000000
s_IMemAddr       32'h0040000C      00400030 00400034 00400038 0040003C 00400040 00400044 00400054 00440030 00440034 00440038 0044003C 00440040 00440044
s_NextInstAddr   32'h0040000C      00400030 00400034 00400038 0040003C 00400040 00400044 00400054 00440030 00440034 00440038 0044003C 00440040 00440044
s_Inst           32'h24090000      8D730000 216B0004 8D740000 21290001 0274602A 15800003 1532FFF6 8D730000 216B0004 8D740000 21290001 0274602A 15800003
s_Halt           0
s_Ovfl           0
s_branchUnit     0
s_controlOut     22'h230020        2B0020 230060 2B0020 230060 120300 104000 2B0020 230060 2B0020 230060 120300 104000
s_opcode         6'hXX             XX
s_rs             5'h00             0B 09 13 0C 09 0B 09 13 0C
s_rt             5'h09             13 0B 14 09 14 00 12 13 0B 14 09 14 00
s_rd             5'h00             00 0C 00 1F 00 0C 00
s_jump           26'hXXXXXXX       XXXXXXX
s_imme           16'hXXXX          XXXX
s_func           6'hXX             XX
s_rsOut          32'h00000000      10010000 10010004 00000000 0000000A 00000001 10010004 10010008 00000001 0000003C 00000000
s_rtOut          32'h00000000      00000000 10010000 00000000 00000000 0000003C 00000000 0000000A 10010004 0000003C 00000001 00000028 00000000
s_extender       32'hXXXXXXXX      XXXXXXXX
s_regSelMuxOut   5'hXX             XX
s_databeforeMux  32'h00000000      0000000A 10010004 0000003C 00000001 00000001 00000001 FFFFFFF7 0000003C 10010008 00000028 00000002 00000000 00000000
s_aluDataOut     32'h00000000      10010000 10010004 10010004 00000001 00000001 00000001 FFFFFFF7 10010004 10010008 10010008 00000002 00000000 00000000
s_memRegMuxOut   32'h00000000      0000000A 10010004 0000003C 00000001 00000001 00000001 FFFFFFF7 0000003C 10010008 00000028 00000002 00000000 00000000
s_lb             32'h00000000      0000000A 00000004 0000003C 00000000 00000000 00000000 FFFFFFFF 0000003C 00000008 00000028 00000000 00000000 00000000
s_lbu            32'h00000000      0000000A 00000004 0000003C 00000000 00000000 00000000 000000FF 0000003C 00000008 00000028 00000000 00000000 00000000
s_lh             32'h00000000      0000000A 00000004 0000003C 00000001 00000001 00000001 FFFFFFFF 0000003C 00000008 00000028 00000000 00000000 00000000
s_lhu            32'h00000000      0000000A 00000004 0000003C 00000001 00000001 00000001 0000FFFF 0000003C 00000008 00000028 00000000 00000000 00000000
s_zero           1
s_afterBEAnd     0
s_afterBNEINV    0
s_afterBNEAnd    0
s_RegWrAddrBef...5'h09             13 0B 14 09 0C 00 1F 13 0B 14 09 0C 00
s_jalAddnext     32'h00400010      00400034 00400038 0040003C 00400040 00400044 00400048 00400058 00440034 00440038 0044003C 00440040 00440044 00440048

Now       12060 ns      280 ns  300 ns  320 ns  340 ns  360 ns  380 ns  400 ns  420 ns  440 ns  460 ns  480 ns  500 ns  520
Cursor 1  97 ns
```

260-520



```
iCLK                 0
iRST                 0
iInstLd              0
iInstAddr            32'hXXXXXXXX  XXXXXXXX
iInstExt             32'hXXXXXXXX  XXXXXXXX
oALUOut              32'h00000000  10010000 10010004 10010004 00000001 00000001 00000001 FFFFFFF7 10010004 10010008 10010008 00000002 00000000 00000000
s_DMemWr             0
s_DMemAddr           32'h00000000  10010000 10010004 10010004 00000001 00000001 00000001 FFFFFFF7 10010004 10010008 10010008 00000002 00000000 00000000
s_DMemData           32'h00000000  00000000 10010000 00000000 00000000 0000003C 00000000 0000000A 10010004 0000003C 00000001 00000028 00000000
s_DMemOut            32'h0000000A  0000000A 0000003C 0000003C 0000000A 0000000A 00000000 0000003C 00000028 00000028 0000000A 0000000A
s_RegWr              1
s_RegWrAddr          5'h09         13 0B 14 09 0C 00 1F 13 0B 14 09 0C 00
s_RegWrData          32'h00000000  0000000A 10010004 0000003C 00000001 00000001 00000001 FFFFFFF7 0000003C 10010008 00000028 00000002 00000000 00000000
s_IMemAddr           32'h0040000C  00400030 00400034 00400038 0040003C 00400040 00400044 00400054 00440030 00440034 00440038 0044003C 00440040 00440044
s_NextInstAddr       32'h0040000C  00400030 00400034 00400038 0040003C 00400040 00400044 00400054 00440030 00440034 00440038 0044003C 00440040 00440044
s_Inst               32'h24090000  8D730000 216B0004 8D740000 21290001 0274602A 15800003 1532FFF6 8D730000 216B0004 8D740000 21290001 0274602A 15800003
s_Halt               0
s_Ovfl               0
s_branchUnit         0
s_controlOut         22'h230020    2B0020 230060 2B0020 230060 120300 104000 2B0020 230060 2B0020 230060 120300 104000
s_opcode             6'hXX         XX
s_rs                 5'h00         0B 09 13 0C 09 0B 09 13 0C
s_rt                 5'h09         13 0B 14 09 14 00 12 13 0B 14 09 14 00
s_rd                 5'h00         00 0C 00 1F 00 0C 00
s_jump               26'hXXXXXXX   XXXXXXX
s_imme               16'hXXXX      XXXX
s_func               6'hXX         XX
s_rsOut              32'h00000000  10010000 10010004 00000000 0000000A 00000001 10010004 10010008 00000001 0000003C 00000000
s_rtOut              32'h00000000  00000000 10010000 00000000 00000000 0000003C 00000000 0000000A 10010004 0000003C 00000001 00000028 00000000
s_extender           32'hXXXXXXXX  XXXXXXXX
s_regSelMuxOut       5'hXX         XX
s_databeforeMux      32'h00000000  0000000A 10010004 0000003C 00000001 00000001 00000001 FFFFFFF7 0000003C 10010008 00000028 00000002 00000000 00000000
s_aluDataOut         32'h00000000  10010000 10010004 10010004 00000001 00000001 00000001 FFFFFFF7 10010004 10010008 10010008 00000002 00000000 00000000
s_memRegMuxOut       32'h00000000  0000000A 10010004 0000003C 00000001 00000001 00000001 FFFFFFF7 0000003C 10010008 00000028 00000002 00000000 00000000
s_lb                 32'h00000000  0000000A 00000004 0000003C 00000000 00000000 00000000 FFFFFFFF 0000003C 00000008 00000028 00000000 00000000 00000000
s_lbu                32'h00000000  0000000A 00000004 0000003C 00000000 00000000 00000000 000000FF 0000003C 00000008 00000028 00000000 00000000 00000000
s_lh                 32'h00000000  0000000A 00000004 0000003C 00000001 00000001 00000001 FFFFFFFF 0000003C 00000008 00000028 00000000 00000000 00000000
s_lhu                32'h00000000  0000000A 00000004 0000003C 00000001 00000001 00000001 0000FFFF 0000003C 00000008 00000028 00000000 00000000 00000000
s_zero               1
s_afterBEAnd         0
s_afterBNEINV        0
s_afterBNEAnd        0
s_RegWrAddrBef...    5'h09         13 0B 14 09 0C 00 1F 13 0B 14 09 0C 00
s_jalAddnext         32'h00400010  00400034 00400038 0040003C 00400040 00400044 00400048 00400058 00440034 00440038 0044003C 00440040 00440044 00440048

Now        12060 ns   280 ns  300 ns  320 ns  340 ns  360 ns  380 ns  400 ns  420 ns  440 ns  460 ns  480 ns  500 ns  520
Cursor 1   97 ns
```

520-760



	Msgs
iCLK	0
iRST	0
iInstLd	0
iInstAddr	32'hXXXXXXXX
iInstExt	32'hXXXXXXXX
oALUOut	32'h00000000
s_DMemWr	0
s_DMemAddr	32'h00000000
s_DMemData	32'h00000000
s_DMemOut	32'h0000000A
s_RegWr	1
s_RegWrAddr	5'h09
s_RegWrData	32'h00000000
s_IMemAddr	32'h0040000C
s_NextInstAddr	32'h0040000C
s_Inst	32'h24090000
s_Halt	0
s_Ovfl	0
s_branchUnit	0
s_controlOut	22'h230020
s_opcode	6'hXX
s_rs	5'h00
s_rt	5'h09
s_rd	5'h00
s_jump	26'hXXXXXXX
s_imme	16'hXXXX
s_func	6'hXX
s_rsOut	32'h00000000
s_rtOut	32'h00000000
s_extender	32'hXXXXXXXX
s_regSelMuxOut	5'hXX
s_databeforeMux	32'h00000000
s_aluDataOut	32'h00000000
s_memRegMuxOut	32'h00000000
s_lb	32'h00000000
s_lbu	32'h00000000
s_lh	32'h00000000
s_lhu	32'h00000000
s_zero	1
s_afterBEAnd	0
s_afterBNEINV	0
s_afterBNEAnd	0
s_RegWrAddrBef...	5'h09
s_jalAddnext	32'h00400010

780-1040

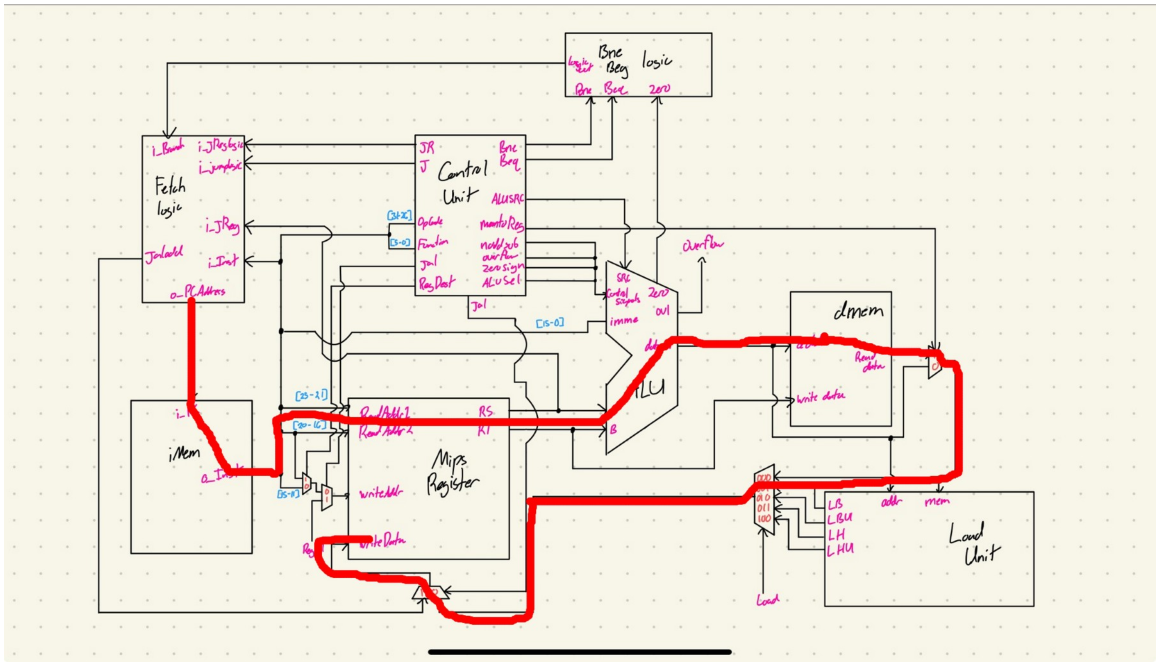
In the above waveforms are the first 1000 ish ns of the bubble sort program running. You can see that its correctly jumping as well as comparing the different values of the (set as 11) array size. It correctly works in MARS as well as on the CPU. I also will include the wave form file since it rand for about 12000ns, and I didnt think it would be helpful to screen shot every single part of it.

Synthesis

[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?

Max frequency: 25.21mhz   Constraint: 20.00ns   Slack: -19.66ns

Also including the timing.txt in case its needed



The critical path of the processor is fetch logic, imem, mips register, alu, dmem, then back to mips register. Some components that could be improved would be the ALU as it spent a lot of time in the ALU running the sllv commands as well as getting to the output of the ALU, almost 10ns