CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 1 Report

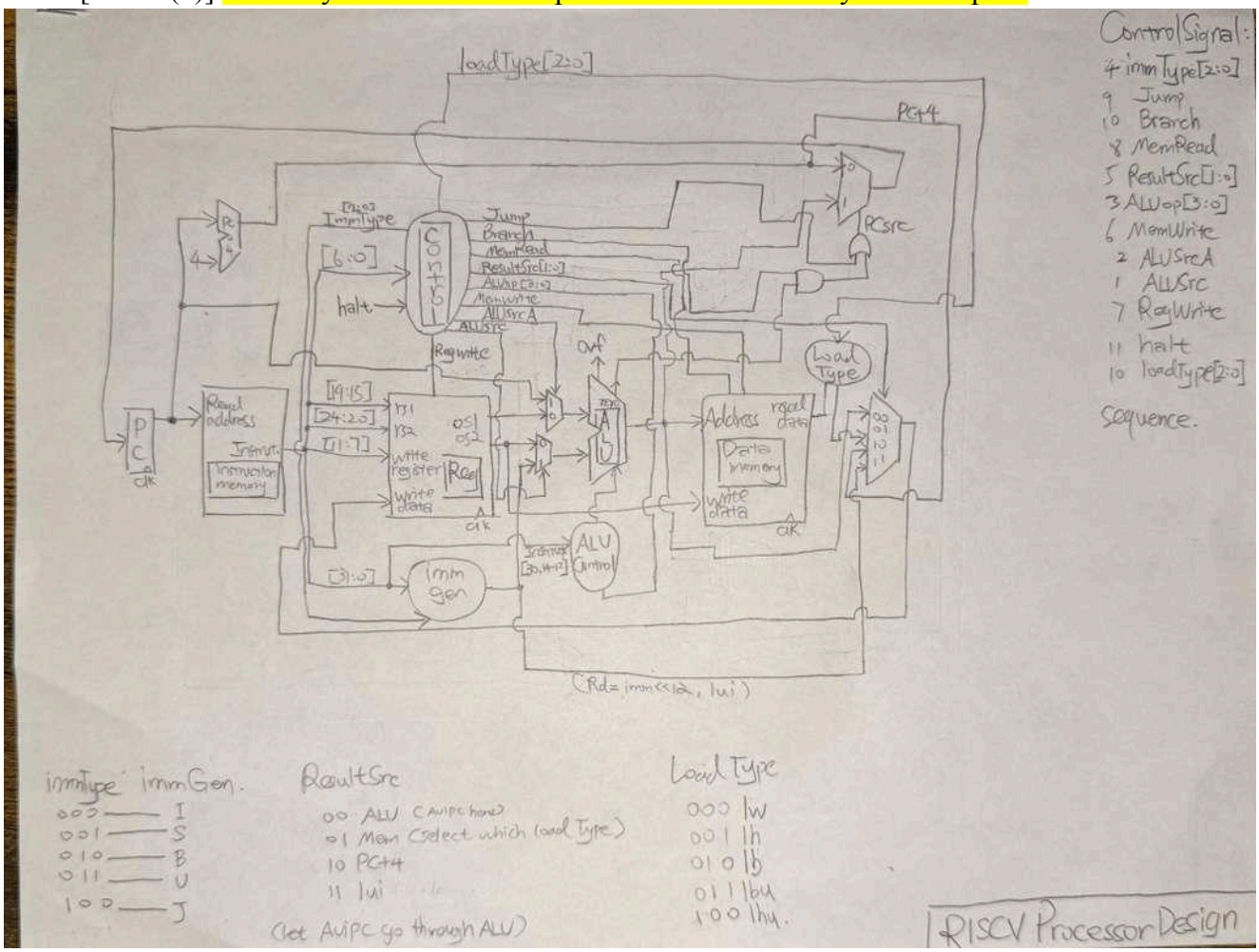
Team Members: Tian Jun Teoh

Austin Nguyen

Project Teams Group #:_____C_03_____

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

[Part 2 (d)] Include your final RISC-V processor schematic in your lab report.

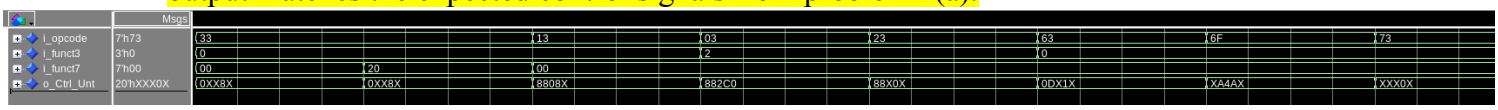


(ALU control unit is not implemented later in our project, we wired ALUOp straight to the ALU)

[Part 3.1.a.] 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.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Instruction	Opcode (Binary)	Funct3 (Binary)	Funct7 or Imm (Binary)	Control Signals											
2					ALUSrc	ALUSrcA	ALUOp	ImmType	ResultSrc	MemWrite	RegWrite	MemRead	Jump	Branch	loadType	halt
3	addi	"0010011"	"000"	imm[11:0]	1	0	0010	000	00	0	1	0	0	0	XXX	0
4	add	0110011	000	0000000	0	0	0010	XXX	00	0	1	0	0	0	XXX	0
5	and	0110011	111	0000000	0	0	0000	XXX	00	0	1	0	0	0	XXX	0
6	andi	0010011	111	imm[11:0]	1	0	0000	000	00	0	1	0	0	0	XXX	0
7	lui	0110111	XXX	imm[31:12]	X	X	XXXX	011	11	0	1	0	0	0	XXX	0
8	lw	0000011	010	imm[11:0]	1	0	0010	000	01	0	1	1	0	0	000	0
9	xor	0110011	100	0000000	0	0	0100	XXX	00	0	1	0	0	0	XXX	0
10	xori	0010011	100	imm[11:0]	1	0	0100	000	00	0	1	0	0	0	XXX	0
11	or	0110011	110	0000000	0	0	0001	XXX	00	0	1	0	0	0	XXX	0
12	ori	0010011	110	imm[11:0]	1	0	0001	000	00	0	1	0	0	0	XXX	0
13	slt	0110011	010	0000000	0	0	0111	XXX	00	0	1	0	0	0	XXX	0
14	slti	0010011	010	imm[11:0]	1	0	0111	000	00	0	1	0	0	0	XXX	0
15	sltiu	0010011	011	imm[11:0]	1	0	1000	000	00	0	1	0	0	0	XXX	0
16	sll	0110011	001	0000000	0	0	1001	XXX	00	0	1	0	0	0	XXX	0
17	srl	0110011	101	0000000	0	0	1010	XXX	00	0	1	0	0	0	XXX	0
18	sra	0110011	101	0100000	0	0	1011	XXX	00	0	1	0	0	0	XXX	0
19	sw	0100011	010	imm[11:0] (S-type: imm[11:5] ‖imm[4:0])	1	0	0010	001	XX	1	0	0	0	0	XXX	0
20	sub	0110011	000	0100000	0	0	0011	XXX	00	0	1	0	0	0	XXX	0
21	beq	1100011	000	imm[12‖10:5‖4:1 ‖11] (B-type)	0	0	0011	010	XX	0	0	0	0	1	XXX	0
22	bne	1100011	001	imm[12‖10:5‖4:1 ‖11] (B-type)	0	0	0011	010	XX	0	0	0	0	1	XXX	0
23	blt	1100011	100	imm[12‖10:5‖4:1 ‖11] (B-type)	0	0	0111	010	XX	0	0	0	0	1	XXX	0
24	bge	1100011	101	imm[12‖10:5‖4:1 ‖11] (B-type)	0	0	0111	010	XX	0	0	0	0	1	XXX	0
25	bltu	1100011	110	imm[12‖10:5‖4:1 ‖11] (B-type)	0	0	1000	010	XX	0	0	0	0	1	XXX	0
26	bgeu	1100011	111	imm[12‖10:5‖4:1 ‖11] (B-type)	0	0	1000	010	XX	0	0	0	0	1	XXX	0
27	jal	1101111	XXX	m[20‖10:1‖11‖19:12] (J-typ	X	1	0010	100	10	0	1	0	1	0	XXX	0
28	jalr	1100111	000	imm[11:0] (I-type)	1	0	0010	000	10	0	1	0	1		XXX	0
29	lb	0000011	000	imm[11:0] (I-type)	1	0	0010	000	01	0	1	1	0	0	010	0
30	lh	0000011	001	imm[11:0] (I-type)	1	0	0010	000	01	0	1	1	0	0	001	0
31	lbu	0000011	100	imm[11:0] (I-type)	1	0	0010	000	01	0	1	1	0	0	011	0
32	lhu	0000011	101	imm[11:0] (I-type)	1	0	0010	000	01	0	1	1	0	0	100	0
33	slli	0010011	001	0000000	1	0	1001	000	00	0	1	0	0	0	XXX	0
34	srli	0010011	101	0000000	1	0	1010	000	00	0	1	0	0	0	XXX	0
35	srai	0010011	101	0100000	1	0	1011	000	00	0	1	0	0	0	XXX	0
36	auipc	0010111	XXX	imm[31:12] (U-type)	1	1	0010	011	00	0	1	0	0	0	XXX	0
37	halt	1110011	XXX	X	X	X	XXXX	XXX	XX	0	0	0	0	0	XXX	1

[Part 3.1.(b)] 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).



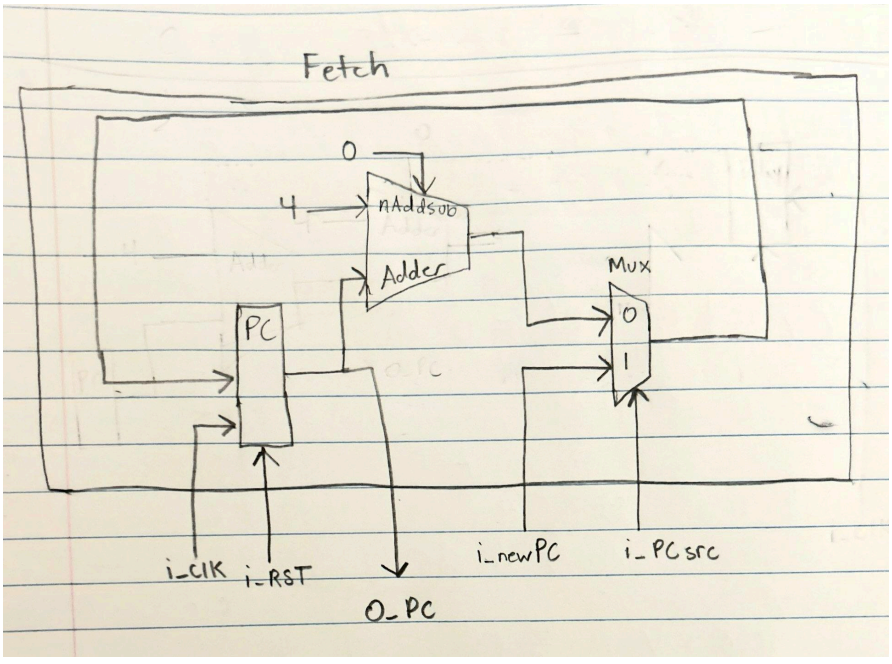
Simulation waveform for the control_unit module using testbench tb_control_unit.vhd. (Dataflow)

We picked one main instruction from each type to test: add, sub, addi, lw, sw, beq, jal, halt. (using 1110011 as opcode)
After decoding the hex from the simulation is all matched up with our spreadsheet control signal value.

[Part 3.2. (a)] 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.

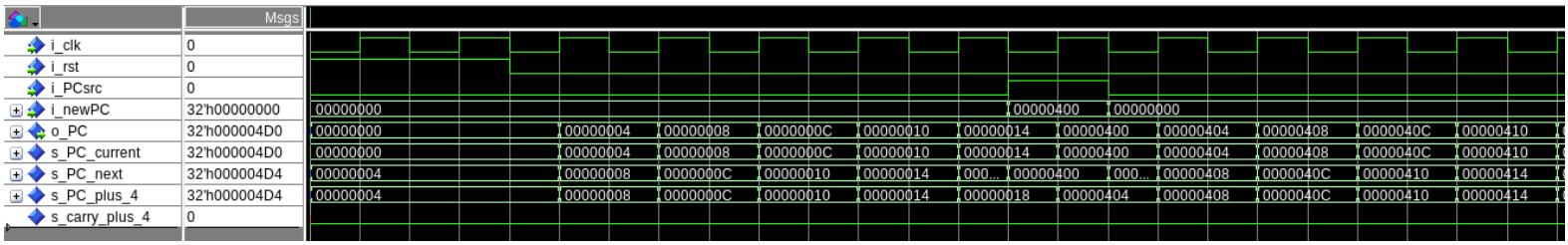
The instruction fetch logic in our RISC-V processor must handle several control flow situations that determine how the program counter (PC) is updated each cycle. For most instructions such as arithmetic, logic, load, and store operations, the processor executes sequentially by fetching the next instruction from PC + 4. For conditional branch instructions (beq, bne, blt, bge, bltu, and bgeu), the ALU comparison flags determine whether the branch condition is true. If the condition is satisfied, the PC is updated to PC + immediate offset; otherwise, it continues sequentially. For unconditional jumps (jal) and indirect jumps (jalr), the control unit signals the fetch logic to redirect the PC to a new target address. In the case of jal, the target address is computed as PC + immediate, while for jalr, it is (rs1 + immediate) & 0xFFFFFFFE, ensuring proper alignment.

[Part 3.2. (b)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?



Additional control signals that were used are i_newPC, which is used when we are performing a branch or jump instruction. We also have a i_PCsrc signal that allows us to select if we want the PC to increment by 4 or change to a specified PC value. We have a clock and reset control signal, for when we want to update the PC and when we want to reset the PC as well.

[Part 3.2.(c)] 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.



In the waveform, we can see that by setting the i_PCsrc to 0, the PC value increments by 4 every clock cycle. This resembles going through the instructions as is without any jumps or branches. We can also see if i_PCsrc is set to 1, we can overwrite the value of the PC with i_newPC. This allows us to resemble a jump or branch for when we need it in the control flow.

[Part 3.3.1.(a)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does RISC-V not have a `sla` instruction?

In RISC-V, both logical right shift (SRL) and arithmetic right shift (SRA) move bits to the right, but they differ in how they handle the sign bit. A logical right shift (SRL) fills the most significant bits (MSBs) with zeros. It is used for unsigned numbers, since it does not preserve any sign information. An arithmetic right shift (SRA) copies the original sign bit (bit 31) into the new MSBs as the bits shift right, preserving the sign of a signed two's-complement number. This maintains the correct mathematical result when dividing signed integers by powers of two. RISC-V does not include an SLA instruction because a left shift affects all bits equally and does not depend on the sign. Whether the number is signed or unsigned, the vacated least significant bits are always filled with zeros.

[Part 3.3.1.(b)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

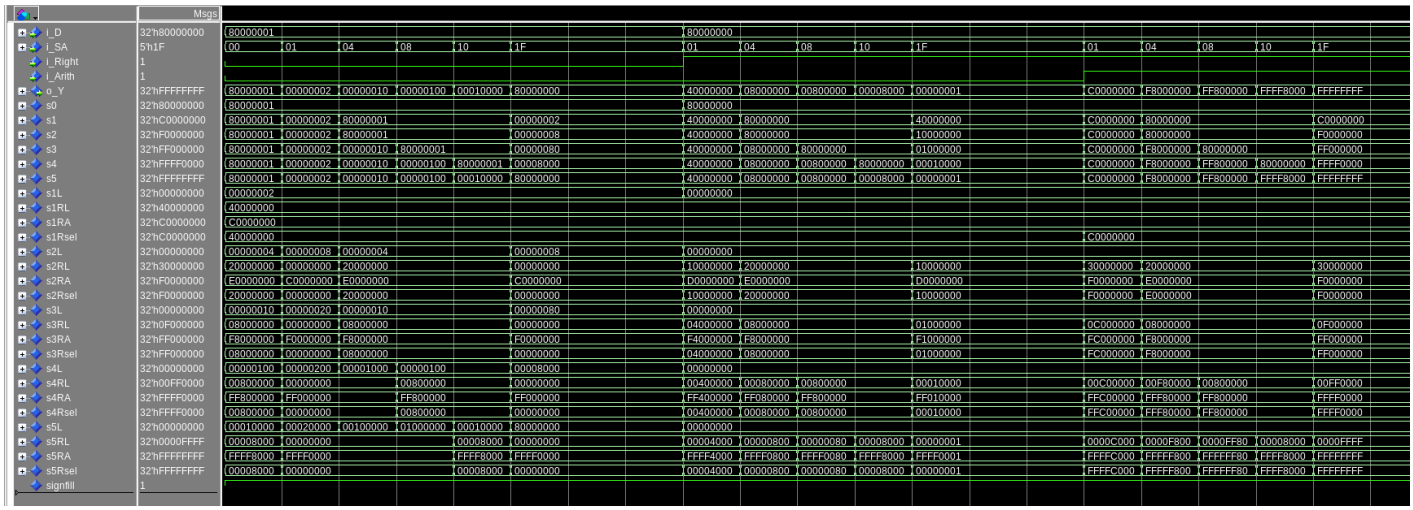
In our design, both logical and arithmetic shifts are implemented structurally in the BarrelShifter32.vhd component. The shifter uses five stages to shift the 32-bit input by 1, 2, 4, 8, or 16 bits depending on the shift-amount input (i_SA). The control signal i_Right selects the shift direction (left or right), while i_Arith determines whether the

operation is logical or arithmetic. For logical right shifts (SRL), zeros are inserted into the vacated most-significant bits, whereas for arithmetic right shifts (SRA), the shifter replicates the sign bit (i_D(31)) into the new bits to preserve the sign of signed values. This conditional behavior is achieved using multiplexers that choose between the zero-filled and sign-extended versions at each stage, allowing the same hardware structure to perform SLL, SRL, and SRA efficiently.

[Part 3.3.1.(c)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

The right barrel shifter can be improved to also support left shifting by introducing a direction control signal (for example, i_Right). This signal determines whether the shifting operation proceeds toward the most significant bit (right shift) or toward the least significant bit (left shift). In the implementation, each stage of the barrel shifter already computes both left-shifted (sXL) and right-shifted (sXR) versions of the data. By using multiplexers controlled by i_Right, the circuit can select which version to propagate to the next stage. When i_Right = '0', the design performs SLL (shift left logical) by inserting zeros into the least significant bits; when i_Right = '1', it performs SRL or SRA depending on the arithmetic control signal (i_Arith). This adjustment allows a single unified hardware block to handle all three RISC-V shift operations: SLL, SRL, and SRA without duplicating logic.

[Part 3.3.1.(d)] Describe how the execution of the different shifting operations corresponds to the QuestaSim waveforms in your writeup.



Simulation waveform for the BarrelShifter32 module using testbench tb_BarrelShifter.vhd.

The simulation waveform verifies the correct functionality of the BarrelShifter32 module for all three shift operations: sll, srl, sra. Each test case in the waveform changes the shift amount

(i_SA), direction (i_Right), and arithmetic control (i_Arith), allowing clear observation of bit movement and fill behavior.

When i_Right = '0', the waveform shows left logical shifts (SLL) , bits move toward the most significant side, and zeros fill the vacated least significant bits. As i_SA increases, the output o_Y shifts left by the corresponding number of bits, matching the expected pattern of power-of-two multiplication.

When i_Right = '1' and i_Arith = '0', the waveform confirms right logical shifts (SRL), bits shift right, and zeros fill the vacated most significant bits. This behavior is visible in the transitions where higher-order bits become zero as the shift amount increases.

Finally, when both i_Right = '1' and i_Arith = '1', the waveform displays right arithmetic shifts (SRA). Here, the sign bit (signfill = i_D(31)) is replicated into the vacant high-order bits. This is evident in the waveform when the input has its MSB set (e.g., i_D = 0x80000000), and the output maintains leading ones after shifting (e.g., 0xF8000000, 0xFF800000, 0xFFFF8000).

[Part 3.3.2.(a)] 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 approach divided the design into reusable components, ControlUnit, ALU, LoadType, Fetch, and RegFile with control signals directing data flow between them. One major design decision we made was to allow the ALU to perform both arithmetic operations and address calculations for branch and jump instructions, eliminating the need for a separate PC adder. This simplification made the datapath cleaner and more flexible.

For control flow, we implemented a branch decision process that uses the ALU's Zero, LT, and LTU flags to determine whether a branch condition is satisfied. The PCSrc signal is generated by an OR gate combining the branch condition (Branch AND Zero) and the Jump signal, ensuring that the PC updates correctly for both branch and jump instructions.

We also designed the write-back select multiplexer (ResultSrc) to choose between four possible data sources: the ALU result, data memory output, PC+4, or the immediate value (for LUI). This ensures that the register file receives the correct data for each instruction type.

The LoadType module extends data read from memory based on the instruction's funct3 field, supporting byte, half-word, and word loads with both signed and unsigned extensions. Together, these design choices made our datapath compact, flexible, and easily testable, while still following the RISC-V ISA behavior precisely.

We also introduced the ALUSrcA signal to select between the PC and RS1 as the first ALU input, enabling support for JALR, JAL, and arithmetic instructions using a shared hardware path.

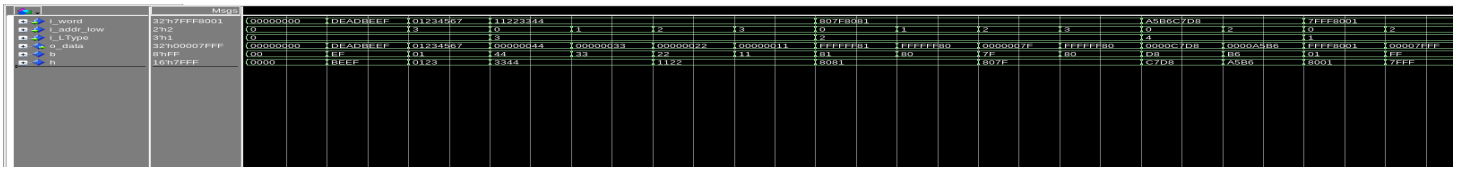
We included an ImmGen module which allows the functionality of performing instructions with immediate values. Since the immediate can be of different sizes

depending on instruction, ImmGen takes an input signal of i_immType, which specifies what instruction type the input instruction is and then converts the 32 bit instruction to a 32 bit signed extended immediate output based on the instruction type.

We used a combination of course-provided materials and external learning resources. We referred to VHDL examples provided by the lectures, quiz and previous labs, which helped us understand module structures such as the register file, ALU, and memory interfaces. ZyBooks has a lot of useful sketches that give us a lot of ideas and tips. We also browsed several online tutorials and YouTube videos on VHDL design concepts, including hierarchical design, multiplexing, and structural design, which helped us visualize and debug our own implementation. We also found one very helpful Powerpoint document on google that help us visualize the datapath for several key instructions. In addition, we consulted the green sheet and lecture slides to confirm instruction formats and control signal mappings.

[Part 3.3.2.(b)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

loadType: Besides fetch, control unit, BarrelShifter that is mentioned in the project instructions. The new component we implemented: loadType as 3.3.2 part A mentioned, is meant to support different load word type instructions like lb, lh, lw, lhu, and lbu.



Simulation waveform for the loadType module using testbench tb_loadType.vhd.

First Test Case: lw
s_word = DEADBEEF, s_addr_low = 00, s_ltype = "000".
For a load word, the entire 32-bit value (DEADBEEF) should be passed directly as s_data.
Expected Output: s_data = DEADBEEF. This is shown correctly in the waveform.

Second Test Case: lbu
s_word = 11223344, s_addr_low = 00, s_ltype = "011".
For lbu, the byte at the selected address (0x44 at addr_low = 00) should be zero-extended to 32 bits.
Expected Output: s_data = 00000044. This is seen in the waveform.

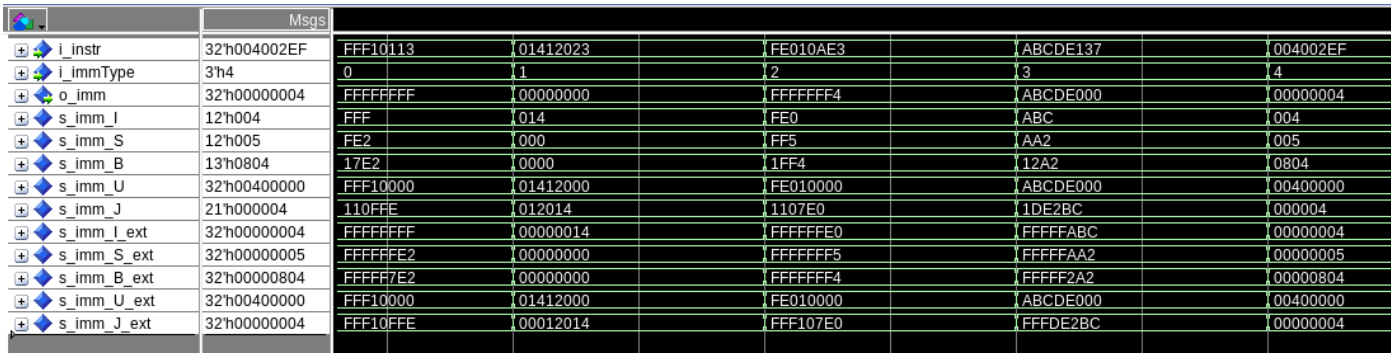
Third Test Case: lb
s_word = 807F8081, s_addr_low = 00, s_ltype = "010".
For lb, the byte at addr_low = 00 is 0x81. Since LB is signed, the byte 0x81 (which is -127 in two's complement) should be sign-extended to 32 bits.
Expected Output: s_data = FFFFFF81. This is correct in the waveform.

Fourth Test Case: lhu
s_word = A5B6C7D8, s_addr_low = 00, s_ltype = "100".
For lhu, the lower half of the word (0xC7D8) should be zero-extended to 32 bits.
Expected Output: s_data = 0000C7D8. This is visible in the waveform.

Fifth Test Case: LH (Load Halfword Signed)
s_word = 7FFF8001, s_addr_low = 00, s_ltype = "001".
For lh, the lower half of the word (0x8001) should be sign-extended. Since 0x8001 is negative, the extension should result in FFFF8001.
Expected Output: s_data = FFFF8001. This matches the waveform.

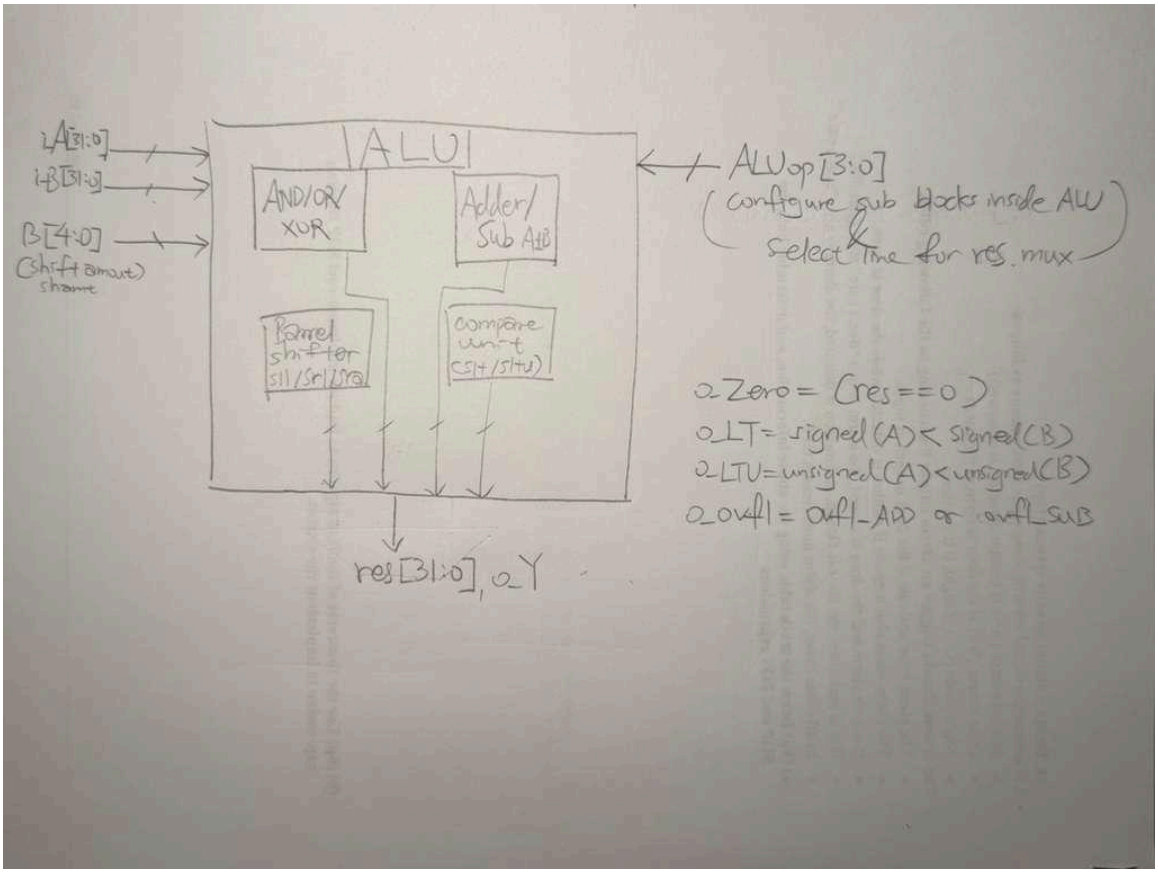
Thus, it proves that the loadType is having the expected behavior as intended and can be safely integrated into the TopLevel.

ImmGen Waveform:



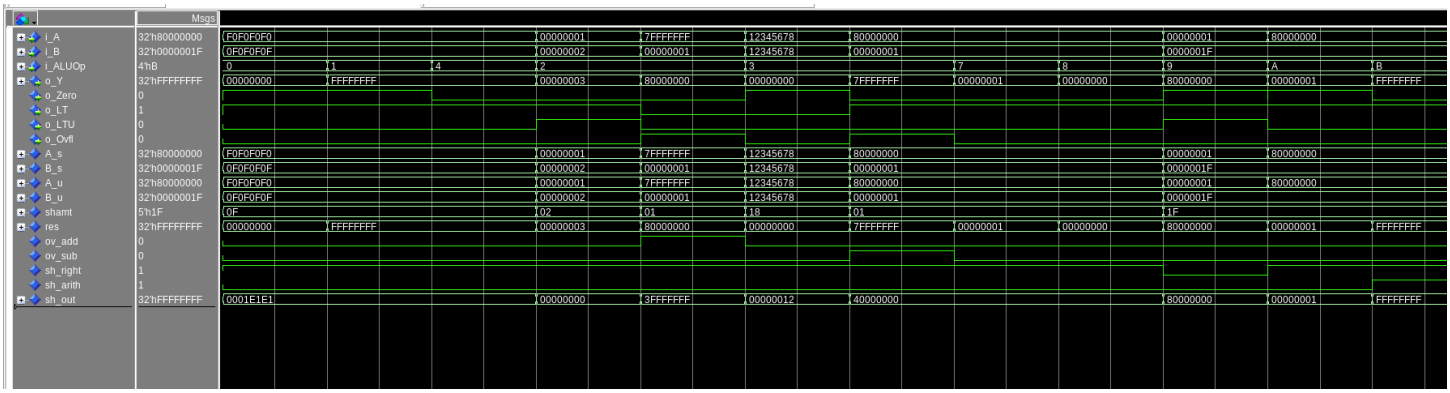
Msgs						
i_instr	32'h004002EF	FFF10113	01412023	FE010AE3	ABCDE137	004002EF
i_immType	3'h4	0	1	2	3	4
o_imm	32'h00000004	FFFFFFFF	00000000	FFFFFFF4	ABCDE000	00000004
s_imm_I	12'h004	FFF	014	FE0	ABC	004
s_imm_S	12'h005	FE2	000	FF5	AA2	005
s_imm_B	13'h0804	17E2	0000	1FF4	12A2	0804
s_imm_U	32'h00400000	FFF10000	01412000	FE010000	ABCDE000	00400000
s_imm_J	21'h000004	110FFE	012014	1107E0	1DE2BC	000004
s_imm_I_ext	32'h00000004	FFFFFFFF	00000014	FFFFFFE0	FFFFFABC	00000004
s_imm_S_ext	32'h00000005	FFFFFFE2	00000000	FFFFFFF5	FFFFFAA2	00000005
s_imm_B_ext	32'h00000804	FFFFF7E2	00000000	FFFFFFF4	FFFFF2A2	00000804
s_imm_U_ext	32'h00400000	FFF10000	01412000	FE010000	ABCDE000	00400000
s_imm_J_ext	32'h00000004	FFF10FFE	00012014	FFF107E0	FFFDE2BC	00000004

[Part 3.3.3] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: How is Zero calculated? How is `slt` implemented?



In our ALU, we compute Zero as a reduction-NOR of the selected 32-bit result: after the result MUX chooses the active operation's output, we set Zero = 1 iff all 32 bits are 0 (implemented as res == 0). This makes branches like beq/bne correct when we drive the ALU with a SUB for comparison (A − B == 0 means beq). For slt (signed less-than), we generate a full 32-bit result where bit 0 reflects the comparison and all upper bits are zero: we set the result to …0001 when signed(A) < signed(B) and …0000 otherwise. Equivalently, this matches the textbook subtract-based form where we compute diff = A − B and form the SLT bit as sign(diff) XOR overflow using two's-complement overflow rules; both approaches synthesize to the same hardware. For sltu (unsigned), we use an unsigned comparison (unsigned(A) < unsigned(B)) and again produce …0001 or …0000. We got around i_Cin and i_Cout and the design also supports branch decisions (via Zero, LT, and LTU as flags) while writing architecturally correct word results for slt/sltu, we avoid a lot more extra components or duplicate ALU for the TopLevel design.

[Part 3.3.5] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.
[Part 3.3.8] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.



Simulation waveform for the ALU module using testbench tb_ALU.vhd.
The execution of the different operations in the provided testbench code for the ALU corresponds to different expected outputs. Here's how these operations reflect in the waveforms from the simulation:

AND operation (ALUOp = 0000):
Expected Output: Y = 00000000 and Zero = 1 because F0F0F0F0 AND 0F0F0F0F results in all zeros.
In the waveform, this should show all signals low (Y = 0) and Zero signal high.

OR operation (ALUOp = 0001):
Expected Output: Y = FFFFFFFF and Zero = 0 because ORing any value with another gives a non-zero result.
The waveform will show Y as all 1s and Zero as 0.

XOR operation (ALUOp = 0100):
Expected Output: Y = FFFFFFFF as the XOR of two non-equal values.
The waveform will reflect this by showing Y = FFFFFFFF.

ADD operation (normal case, ALUOp = 0010):
Expected Output: Y = 00000003 and Ovfl = 0 for normal addition.
The waveform should reflect Y = 00000003 with no overflow signal (Ovfl = 0).

ADD overflow (ALUOp = 0010):
Expected Output: Y = 80000000 and Ovfl = 1 since 7FFFFFFF + 1 overflows.
In the waveform, the overflow flag (Ovfl) will be high, showing the result 80000000.

SUB operation (equal case, ALUOp = 0011):
Expected Output: Y = 00000000 and Zero = 1 because 12345678 - 12345678 = 0.
The waveform will show the subtraction result Y = 00000000 and Zero = 1.

SUB overflow (ALUOp = 0011):
Expected Output: Y = 7FFFFFFF and Ovfl = 1 due to underflow.
The overflow signal will be high, showing the result as 7FFFFFFF.

SLT (signed less than, ALUOp = 0111):
Expected Output: Y = 00000001 as 80000000 < 00000001 is true in signed comparison.
The waveform will show Y = 00000001.

SLTU (unsigned less than, ALUOp = 1000):
Expected Output: Y = 00000000 as 80000000 > 00000001 in unsigned comparison.
The waveform will show Y = 00000000.

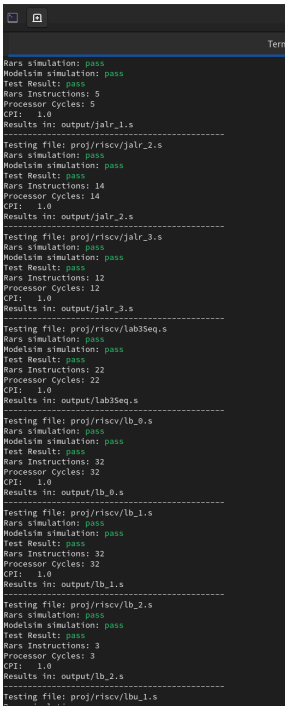
SLL (shift left logical, ALUOp = 1001):
Expected Output: Y = 80000000 because shifting 00000001 left by 31 bits results in 80000000.
The waveform will reflect Y = 80000000.

SRL (shift right logical, ALUOp = 1010):
Expected Output: Y = 00000001 because shifting 80000000 right by 31 bits results in 00000001.
The waveform will show Y = 00000001.

SRA (shift right arithmetic, ALUOp = 1011):
Expected Output: Y = FFFFFFFF due to arithmetic shift behavior on 80000000.
The waveform will reflect Y = FFFFFFFF

Waveform proves that the ALU is working and behavioring as intended by our design.

[Part 4] 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.

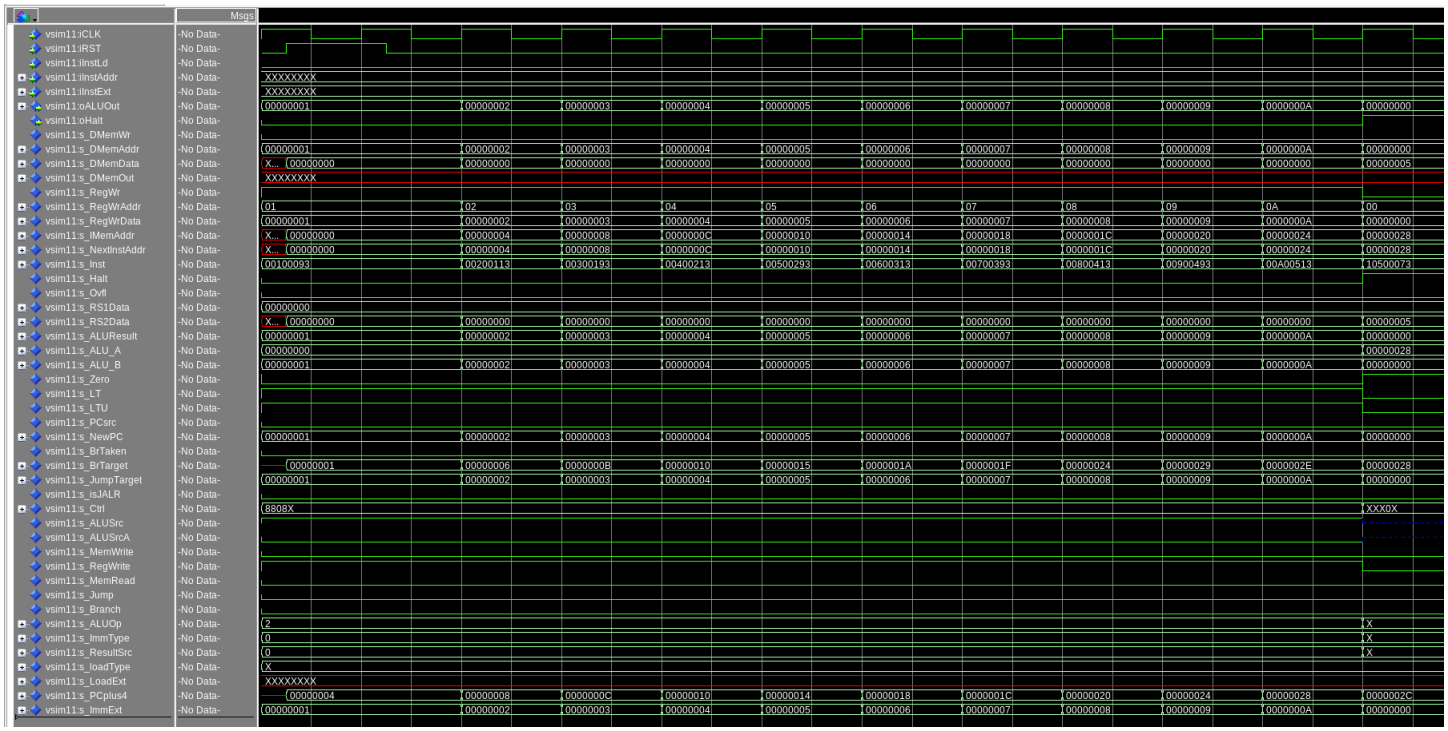


We stress tested with all the test programs consists of all the required instructions in cpre3810_test_assembly_program_suite.zip.
At first we found out that auipc, bne, beq, and sw is not passing the test and not working properly. We debug by checking out the error log, traces and vsim.wlf to observe the waveform (output folder). We found out for auipc the starting PC address is expected to start at 0x00400000 instead of 0x00, so we fix that in the fetch. For bne and beq, the error is caused by the zero flag handling and we fixed that by moving the zero flag update in ALU at the end of the code to let it get the correct updated zero flag. As for sw, the error was caused by a small typo in the control signal and it was fixed by simply changing the DMemWr bit in control to high for sw.
All the tests are passed after fixing all the problems, proving our RISCVProcessor is fully functional and working correctly!

After stress testing, we tested all the five given programs in the riscv folder, all five tests passed as well.

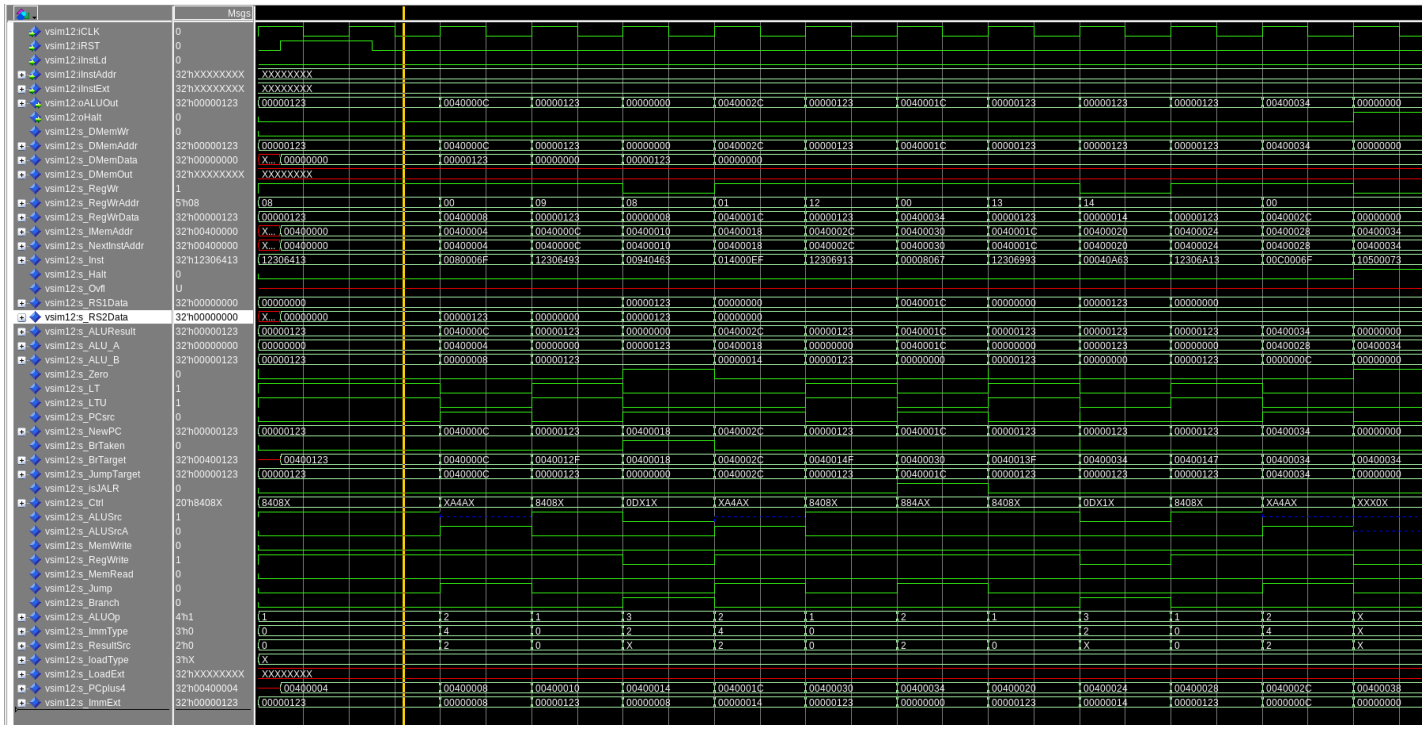
Addiseq.s

```
bash-5.1$ ./3810_tf.sh test proj/riscv/addiseq.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid,
All VHDL src files compiled successfully
Testing file: proj/riscv/addiseq.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 11
Processor Cycles: 11
CPI:   1.0
Results in: output/addiseq.s
------------------------------------------------
```



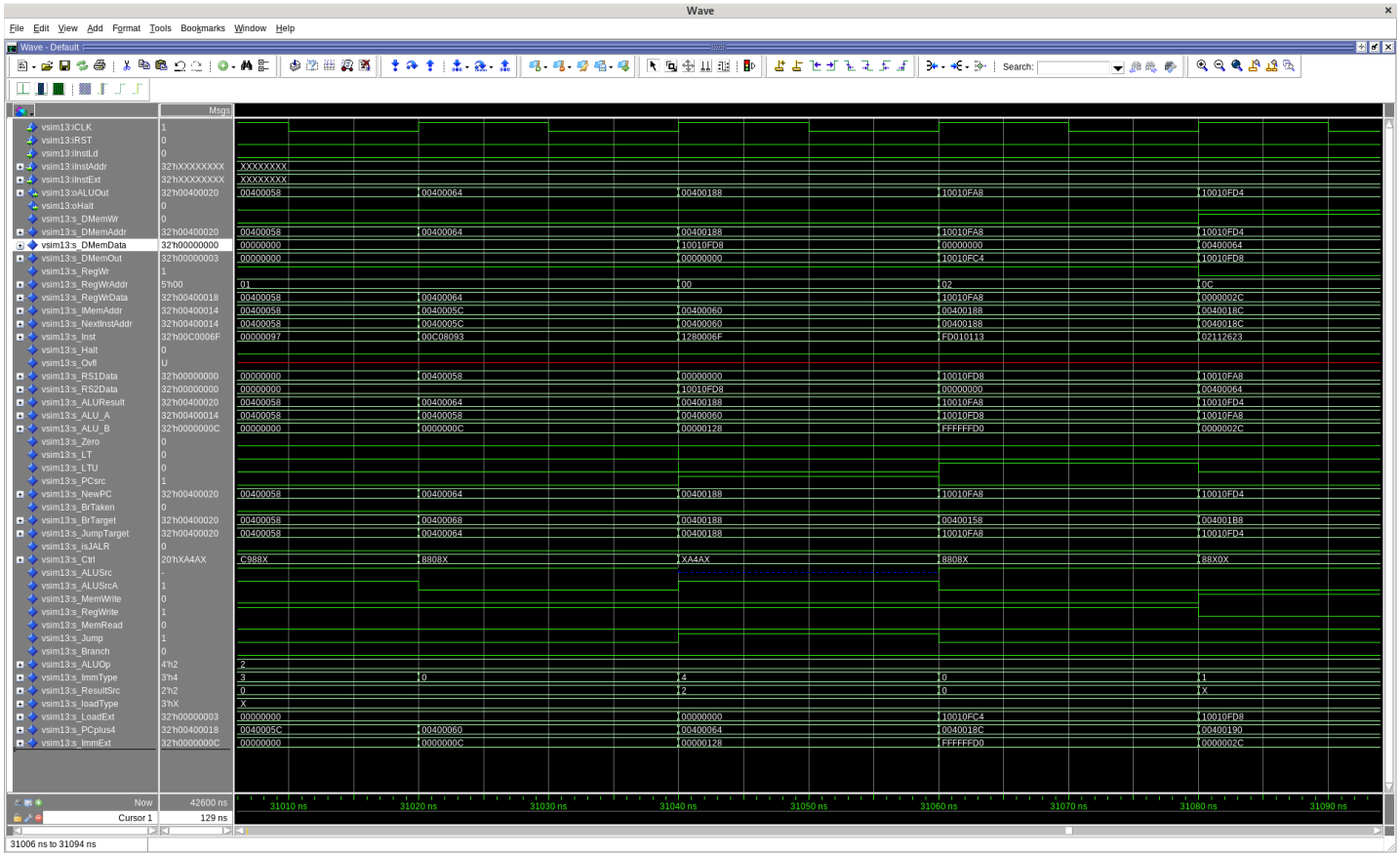
Simplebranch.s

```
bash-5.1$ ./3810_tf.sh test proj/riscv/simplebranch.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, us
All VHDL src files compiled successfully
Testing file: proj/riscv/simplebranch.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 12
Processor Cycles: 12
CPI:    1.0
Results in: output/simplebranch.s
```



grendel.s

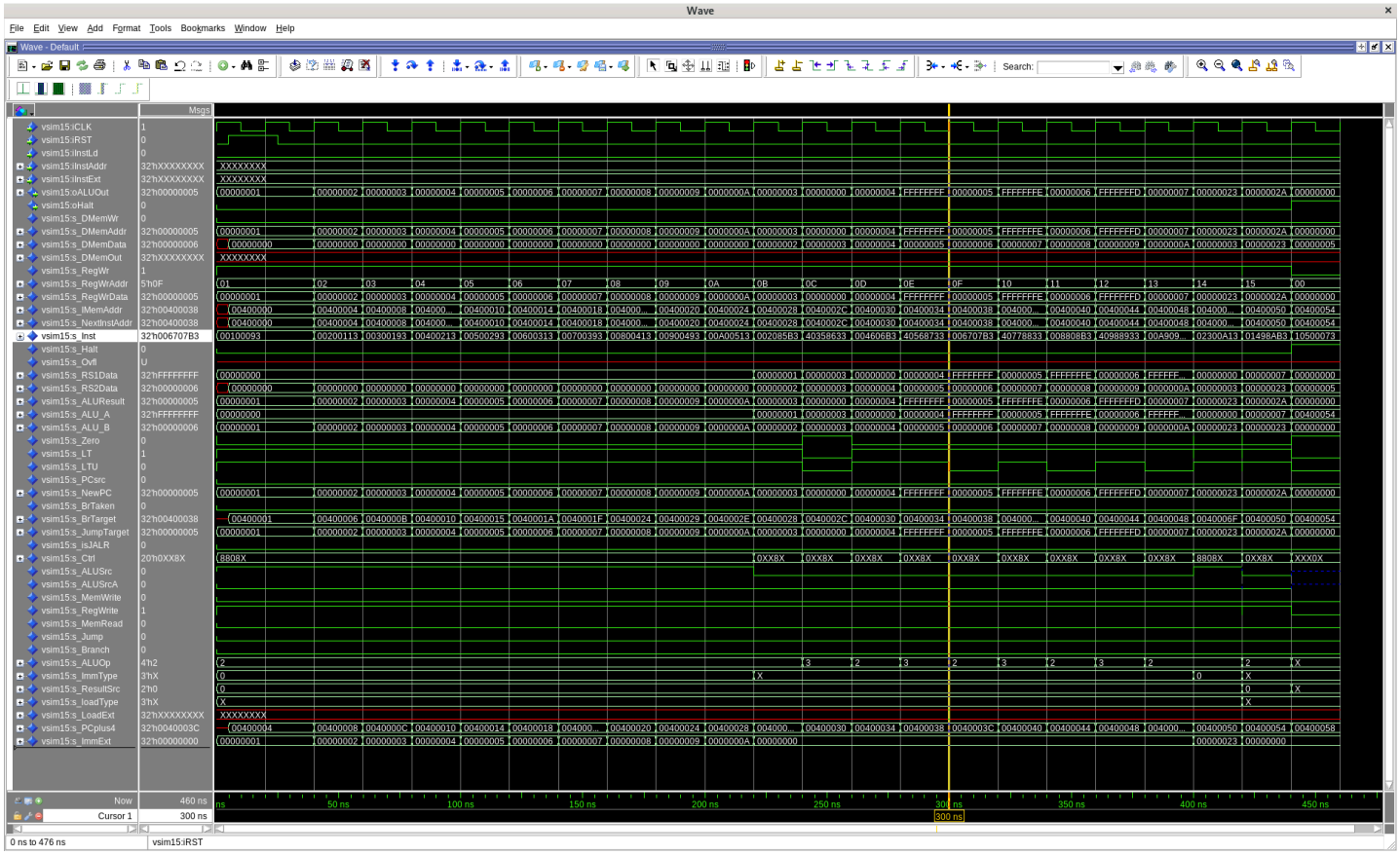
```
bash-5.1$ ./3810_tf.sh test proj/riscv/grendel.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invali
All VHDL src files compiled successfully
Testing file: proj/riscv/grendel.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 2129
Processor Cycles: 2129
CPI:    1.0
Results in: output/grendel.s
```



(This program is too long to get full screenshot of the full waveform)

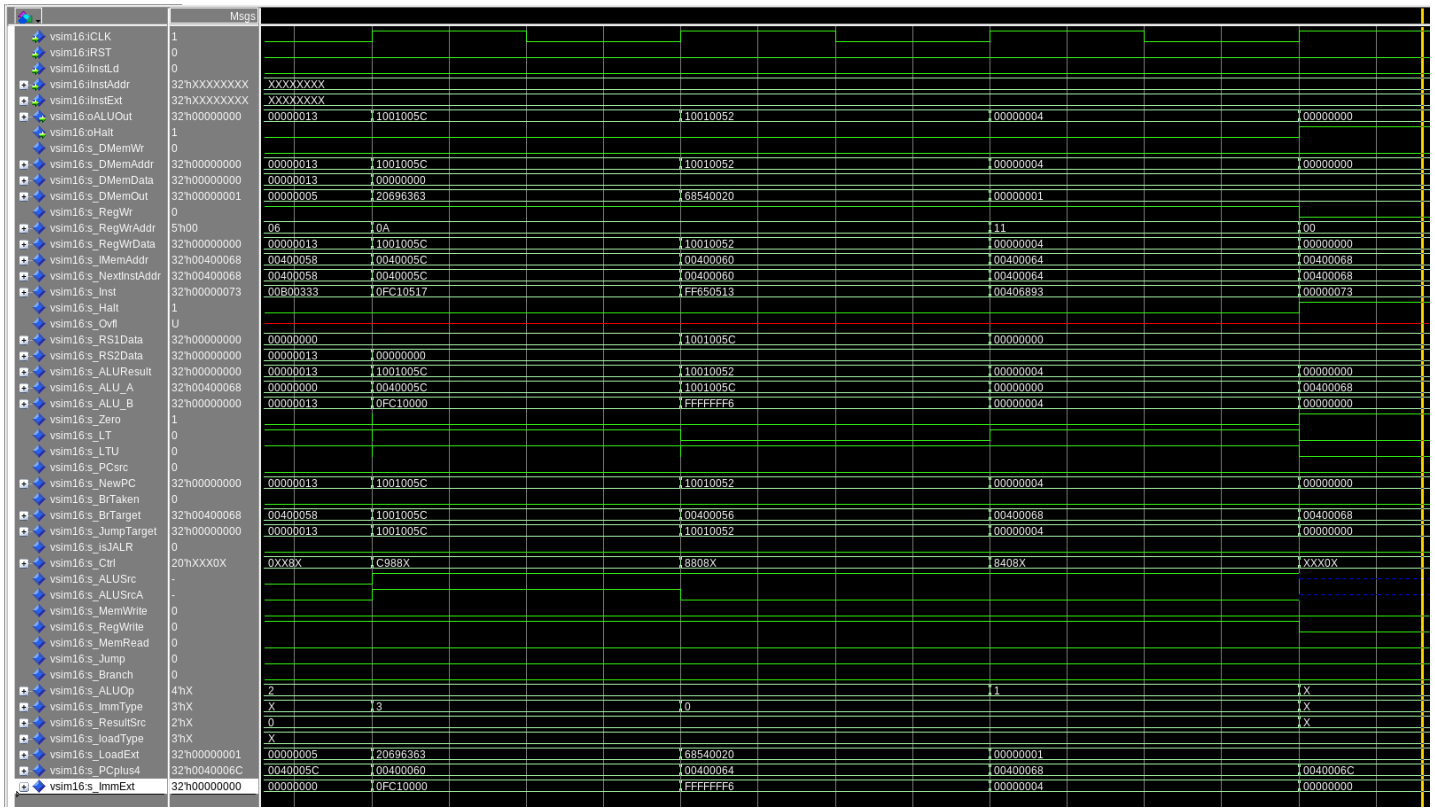
lab3Seq.s

```
bash-5.1$ ./3810_tf.sh test proj/riscv/lab3Seq.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invali
All VHDL src files compiled successfully
Testing file: proj/riscv/lab3Seq.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 22
Processor Cycles: 22
CPI:    1.0
Results in: output/lab3Seq.s
------------------------------------------------
```



Fibonacci.s

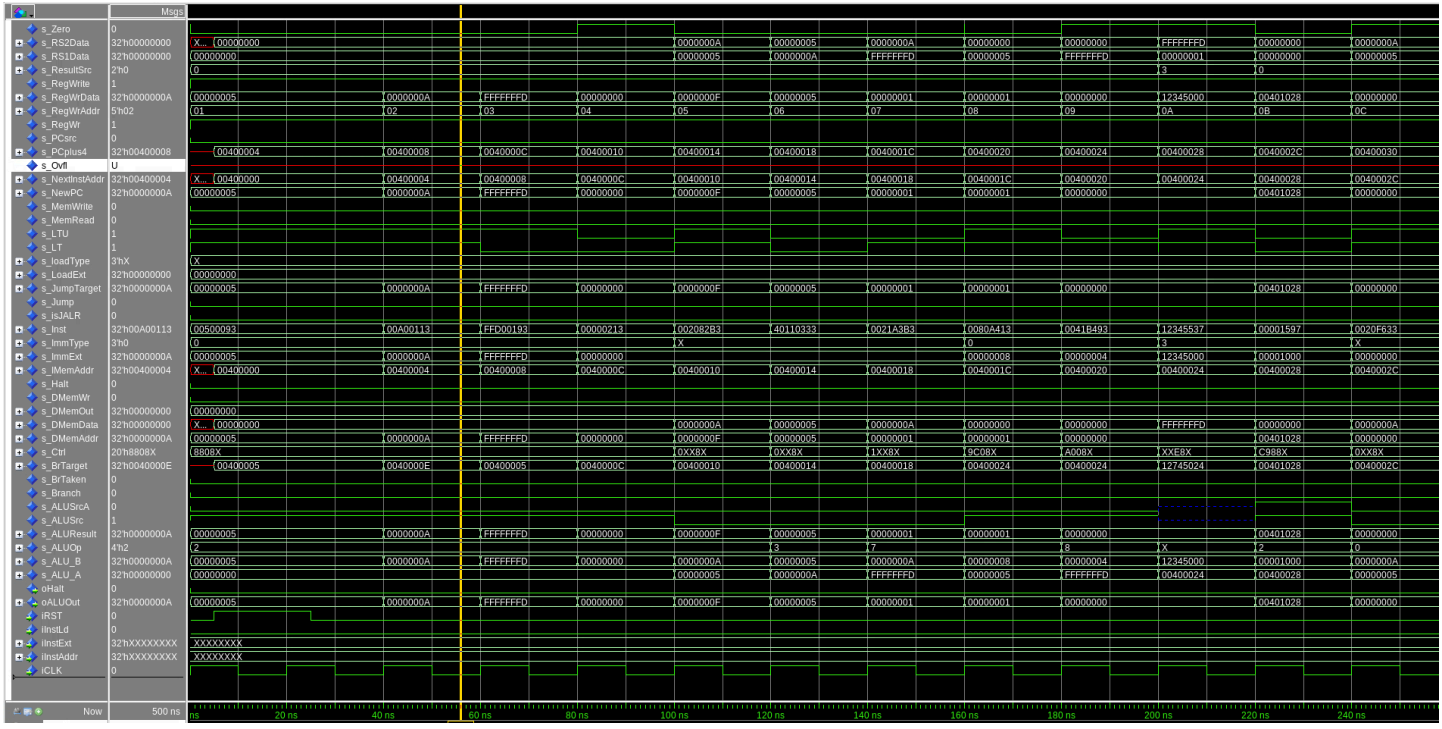
```
-----------------------------------------------
Testing file: proj/riscv/fibonacci.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 129
Processor Cycles: 129
CPI:   1.0
Results in: output/fibonacci.s
```



[Part 4.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.

```
bash-5.1$ ./3810_tf.sh test proj/riscv/Proj1_base_test.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using
All VHDL src files compiled successfully
Testing file: proj/riscv/Proj1_base_test.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 24
Processor Cycles: 24
CPI:    1.0
Results in: output/Proj1_base_test.s
```

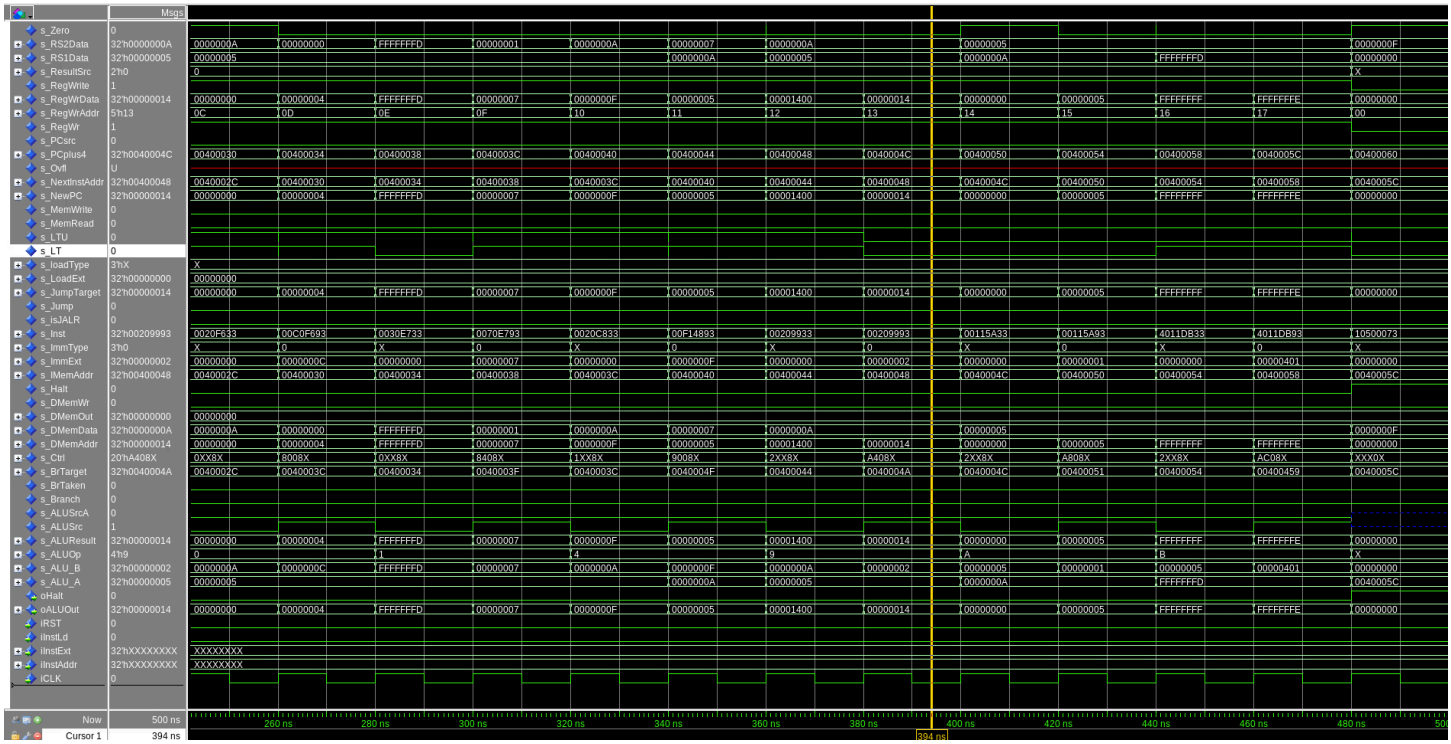
Proj1_base_test.s Created (proj/riscv) and tested successfully with our own RISCVProcessor!



0~250ns

Take one point for example, can see at yellow line point, instruction is 0x00A00113, this is doing 0000 0000 1010 0000 0000 0001 0001 0011, opcode 0010011, func3 is 000, which is ‘addi’, immediate is A which is 10, rd is 0010 which is x2, rs1 is

00000 which is x0, so it is doing addi x2, x0, 10, expected result is 10, on the waveform oALUOut which is the ALU result is A which is 10, indicating correct behavior and result.



250ns~500ns

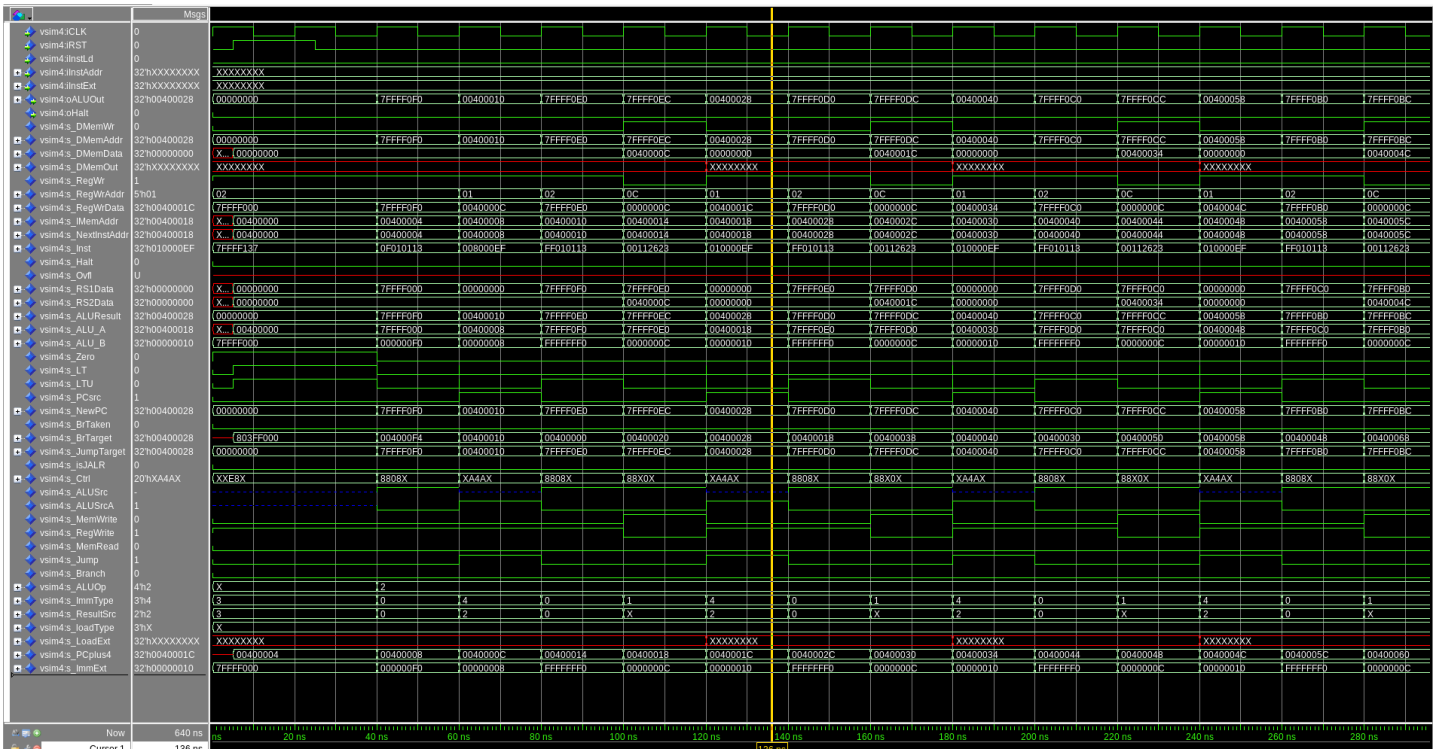
Take yellow line point for example, instruction is 0x00209993, this is doing 0000 0000 0010 0000 1001 1001 1001 0011, opcode 0010011, func3 is 001, which is ‘slli’, immediate is 2, rd is 10011 which is x19, rs1 is 00001 which is x1 current value is 5 indicating by s_RS1Data, so it is doing slli x19, x1, 2, in our program it is doing x19= 5 << 2, expected output is 20, on the waveform oALUOut, which is the ALU result is 0x14 which is 20, flags and control signal matching expected values, all indicating correct behavior and result.

All other instructions tested by our created application can be observed on the waveform, indicating our processor fully supports the arithmetic and logic instructions required by the project.

[Part 4.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.

```
bash-5.1$ ./3810_tf.sh test proj/riscv/Proj1_cf_test.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, usi
entor/calibre
All VHDL src files compiled successfully
Failed to remove output/Proj1_cf_test.s. Is questasim o
Saving instead to "output/Proj1_cf_test.s_1"
Failed to remove output/Proj1_cf_test.s. Is questasim o
Saving instead to "output/Proj1_cf_test.s_2"
Testing file: proj/riscv/Proj1_cf_test.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 31
Processor Cycles: 31
CPI:    1.0
Results in: output/Proj1_cf_test.s_2
```

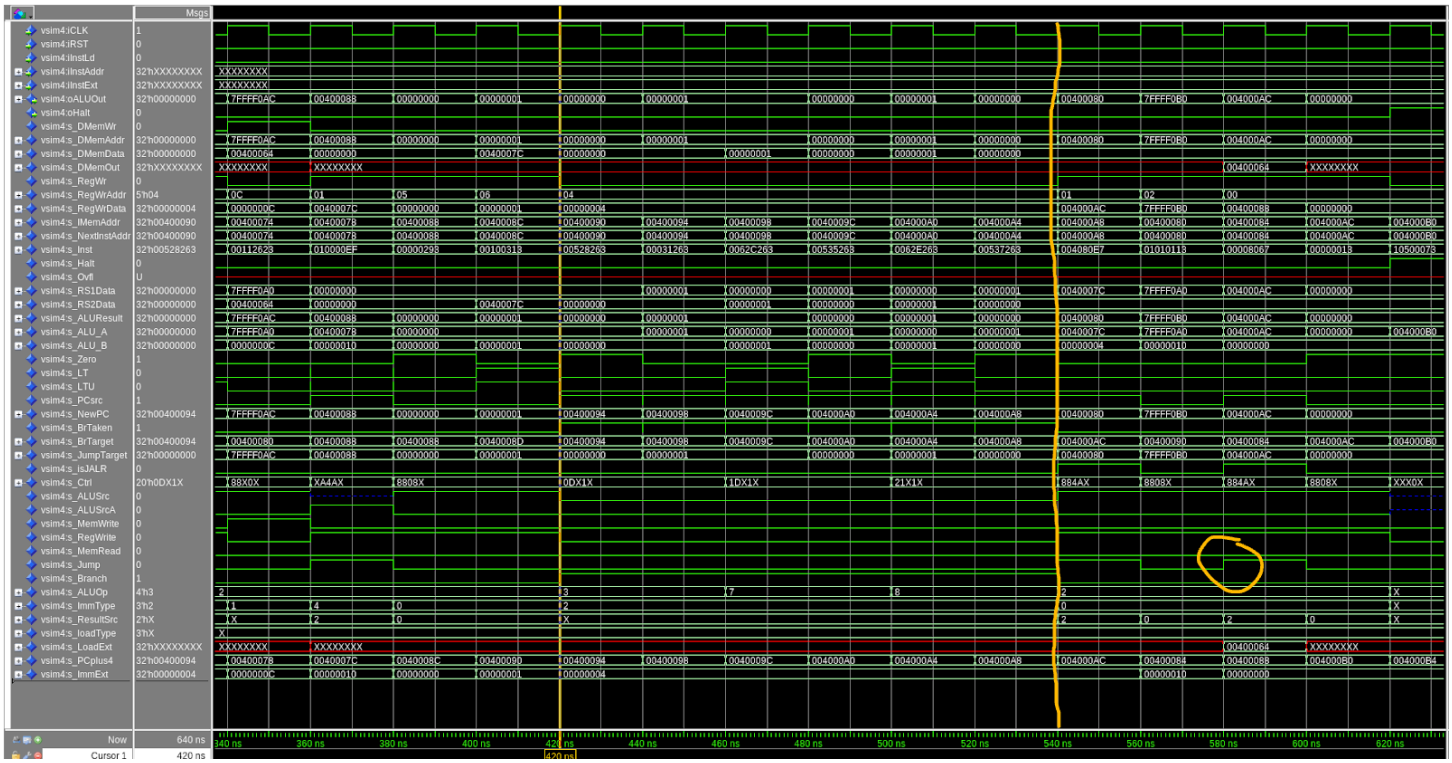
Proj1_cf_test.s Created (proj/riscv) and tested successfully with our own RISCVProcessor



0ns~290ns

This section of our application tests the jal instruction, simulating a sequence of jumps that eventually lead to the branch_test.You can find those jump by looking at the jump signal when it is high. Take the yellow line point as an example. Jump signal is high, instruction is 0x010000EF, Jump PC-relative by +16 bytes and store the return address

(PC + 4) into x1 (ra). The register write signal shows activity on x1, meaning the return address (current PC is 0x00400018 PC + 4= 0x0040001C) is properly stored. Simultaneously, the program counter updates to the new target address shown by signal NewPC (PC + 16=0x00400028). These combined observations verify that the jal instruction successfully redirects program execution to the correct jump target, demonstrating proper handling of PC-relative jumps.



340ns~640ns

This waveform is showing the later part of the application, where the branch test is being conducted starting at the yellow line when the branch signal becomes high (before branching we set the t0 to 0 and t1 to 1).
Branch test we wrote:

```
Branch_test:
addi t0, x0, 0
addi t1, x0, 1

beq t0, t0, L1
L1:
bne t1, x0, L2
L2:
blt t0, t1, L3
```

```
L3:
bge t1, t0, L4
L4:
bltu t0, t1, L5
L5:
bgeu t1, t0, L6
L6:
jalr ra, ra, 4 # link to next addr +4 go L7
L7:
addi x0, x0, 0
wfi
```

It is very straightforward, if a branch is taken it would carry on to fall onto the next branch instruction. The jalr in L6 would be jumping to L7 by incrementing 4 to the address. The result can be proven by the waveform (between two yellow lines and circled by the yellow circle) that our Processor successfully passes through all branch instructions and jalr all the way to the end of the program, indicating our processor fully supports the control flow required by the project.

[Part 4.c] Create and test an application that sorts an array with *N* elements using the MergeSort algorithm (link). Name this file Proj1_mergesort.s.

home > teoh > Documents > C mergesort.c > main()

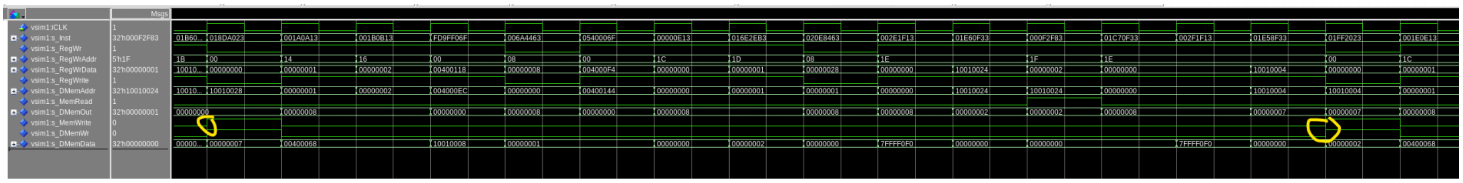
```c
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temporary arrays
    int L[n1], R[n2];

    // Copy data into temp arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    // Merge the temp arrays back into arr[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void iterativeMergeSort(int arr[], int n) {
    int curr_size;  // current size of subarrays to be merged
    int left_start; // Starting index of left subarray
    // Merge subarrays in bottom-up manner
    for (curr_size = 1; curr_size < n; curr_size = 2 * curr_size) {
        // Pick starting point for each subarray to merge
        for (left_start = 0; left_start < n - 1; left_start += 2 * curr_size) {
            // Find the end point of the right subarray
            int mid = (left_start + curr_size - 1);
            int right_end = (left_start + 2 * curr_size - 1) < (n - 1) ? (left_start + 2 * curr_size - 1) : (n - 1);

            // Merge the subarrays
            if (mid < right_end)
                merge(arr, left_start, mid, right_end);
        }
    }
}

int main() {
    int arr[] = {7, 2, 9, 1, 6, 3, 8, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: ");
    for (int i = 0; i < n; i++)
```

Merge sort code reference in C language

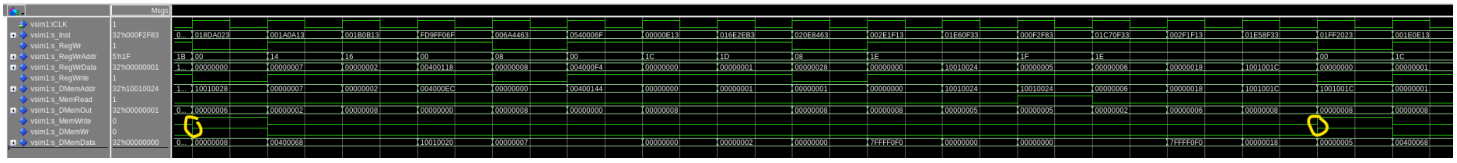
```
bash-5.1$ ./3810_tf.sh test proj/riscv/Proj1_mergesort.s
Using VDI Python Environment
Testing
WARNING: Software tree location not set or invalid, using
All VHDL src files compiled successfully
Failed to remove output/Proj1_mergesort.s. Is questasim o
Saving instead to "output/Proj1_mergesort.s_1"
Testing file: proj/riscv/Proj1_mergesort.s
Rars simulation: pass
Modelsim simulation: pass
Test Result: pass
Rars Instructions: 817
Processor Cycles: 817
CPI:    1.0
Results in: output/Proj1_mergesort.s_1
```

Proj1_mergesort.s (proj/riscv) and tested successfully with our own RISCVProcessor

The example array the application tested: .word 7, 2, 9, 1, 6, 3, 8, 5. N=8
The waveform is very long, we would pick out a few point to show that our test is working.

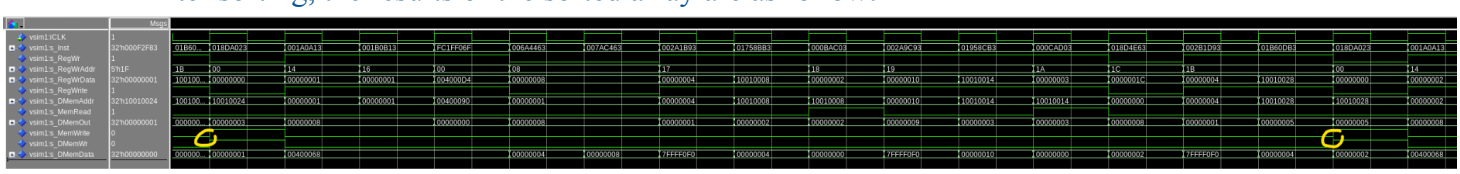


At the yellow marker, the s_DMemWr signal is asserted high for the first time in the waveform, indicating that a store operation is taking place. The s_DMemData value at this moment shows 7 and 2, which correspond to the first two elements from the unsorted array being written. This confirms that the processor is currently storing the original, unsorted data into memory at the start of the merge process.

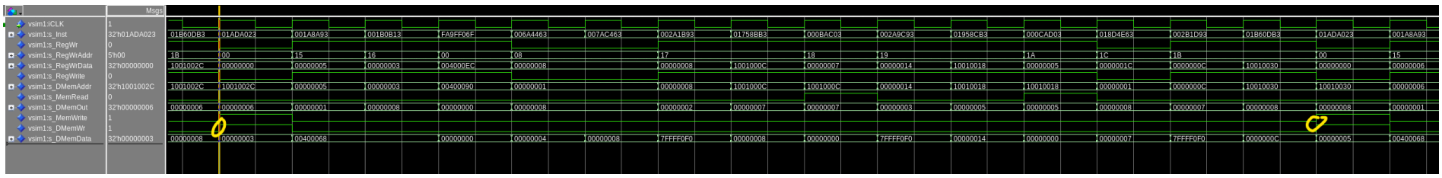


As shown by the yellow marker, at this point, the last two elements of the unsorted array, 8 and 5, has been written.

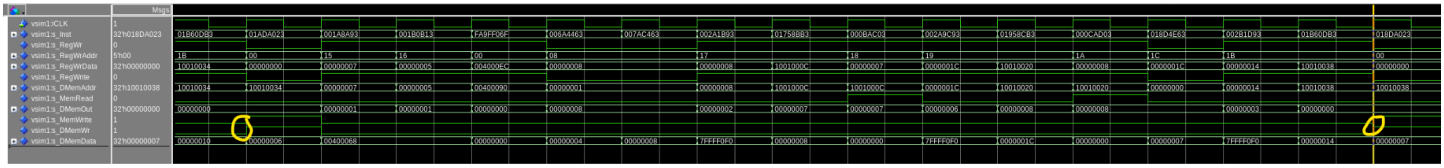
After sorting, the results of the sorted array are as follow:



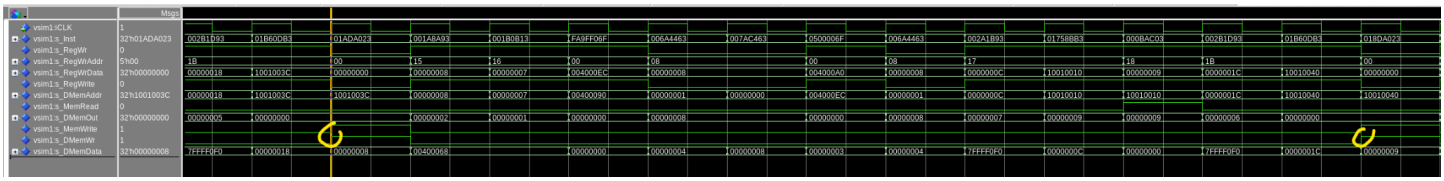
As shown by the yellow marker, at this point, the first two elements of the sorted array, 1 and 2, has been written. This is correct as 1 and 2 are the smallest elements in the array.



As shown by the yellow marker, at this point, 3 and 5 has been written. This is correct order.



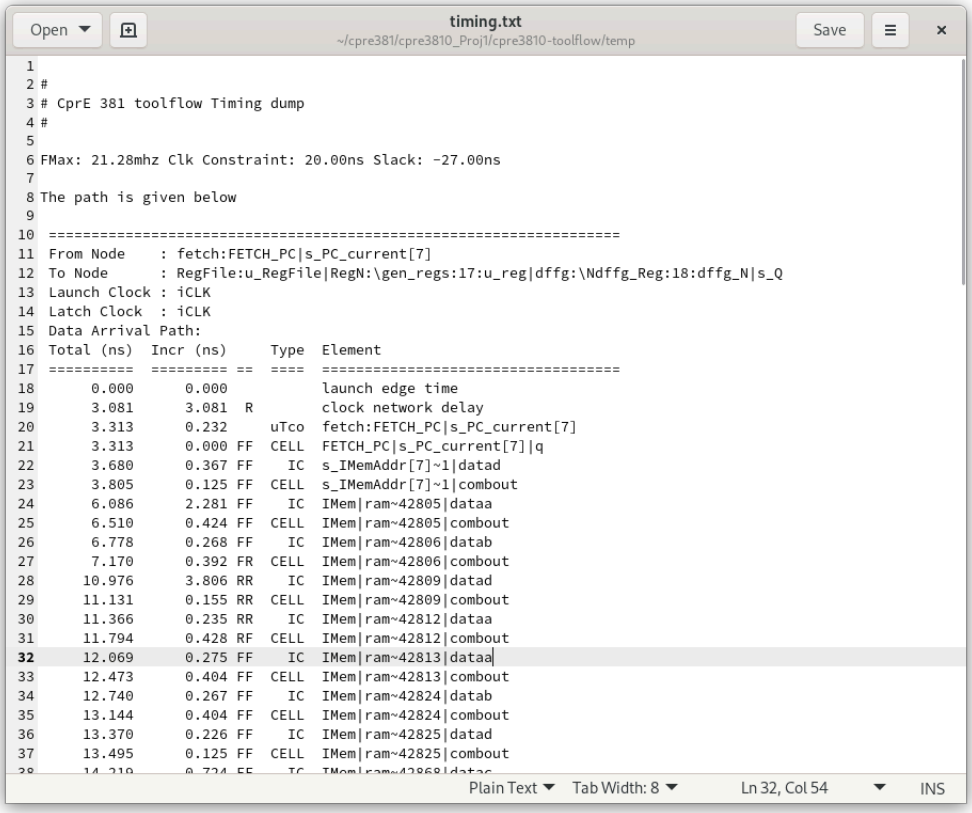
As shown by the yellow marker, at this point, 6 and 7 has been written. This is correct order.



As shown by the yellow marker, at this point, the last two elements of the sorted array, 8 and 9, has been written. This is correct as 8 and 9 are the largest elements in the array.

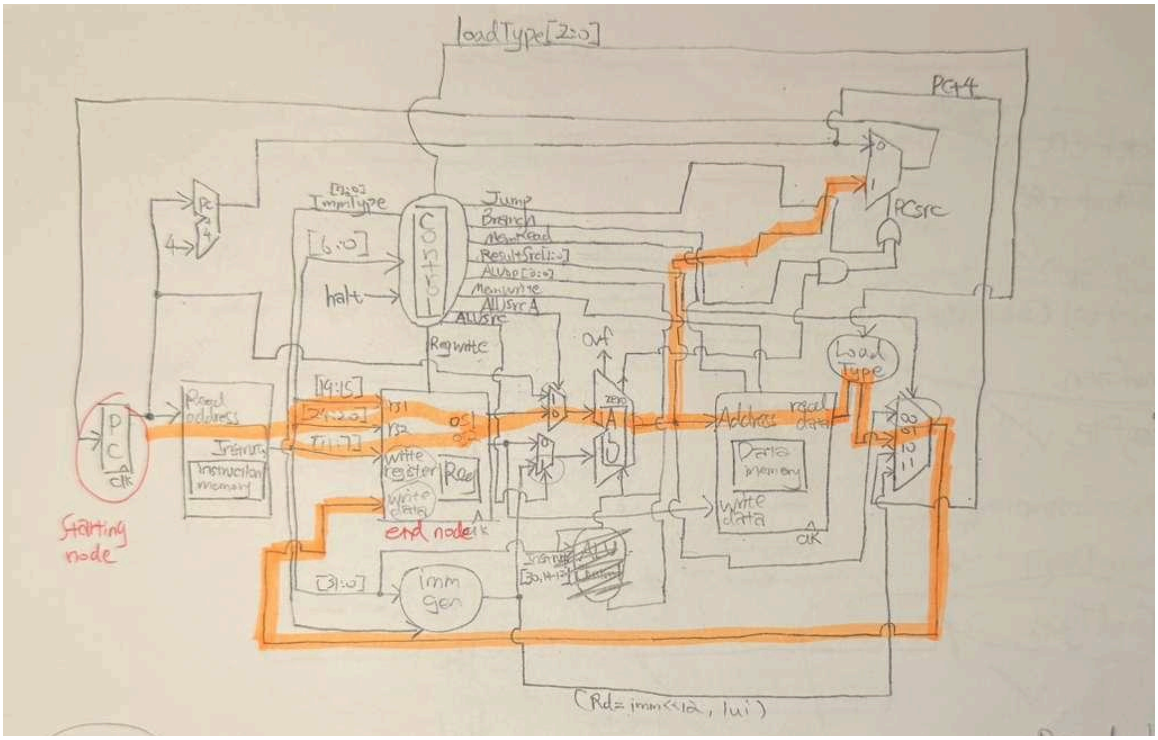
This confirms that our mergeSort program has successfully sorted the array, demonstrating that our processor is functioning correctly and performing all operations as intended.

[Part 5] 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?




```
#
# CprE 381 toolflow Timing dump
#

FMax: 21.28mhz Clk Constraint: 20.00ns Slack: -27.00ns

The path is given below

================================================================
From Node    : fetch:FETCH_PC|s_PC_current[7]
To Node      : RegFile:u_RegFile|RegN:\gen_regs:17:u_reg|dffg:\Ndffg_Reg:18:dffg_N|s_Q
Launch Clock : iCLK
Latch Clock  : iCLK
Data Arrival Path:
Total (ns)  Incr (ns)     Type  Element
==========  ========= ==  ====  ===================================
     0.000     0.000           launch edge time
     3.081     3.081  R        clock network delay
     3.313     0.232    uTco  fetch:FETCH_PC|s_PC_current[7]
     3.313     0.000 FF  CELL  FETCH_PC|s_PC_current[7]|q
     3.680     0.367 FF    IC  s_IMemAddr[7]~1|datad
     3.805     0.125 FF  CELL  s_IMemAddr[7]~1|combout
     6.086     2.281 FF    IC  IMem|ram~42805|dataa
     6.510     0.424 FF  CELL  IMem|ram~42805|combout
     6.778     0.268 FF    IC  IMem|ram~42806|datab
     7.170     0.392 FR  CELL  IMem|ram~42806|combout
    10.976     3.806 RR    IC  IMem|ram~42809|datad
    11.131     0.155 RR  CELL  IMem|ram~42809|combout
    11.366     0.235 RR    IC  IMem|ram~42812|dataa
    11.794     0.428 RF  CELL  IMem|ram~42812|combout
    12.069     0.275 FF    IC  IMem|ram~42813|dataa
    12.473     0.404 FF  CELL  IMem|ram~42813|combout
    12.740     0.267 FF    IC  IMem|ram~42824|datab
    13.144     0.404 FF  CELL  IMem|ram~42824|combout
    13.370     0.226 FF    IC  IMem|ram~42825|datad
    13.495     0.125 FF  CELL  IMem|ram~42825|combout
```

After checking the timing.txt result from the synth, our Processor maximum frequency is **21.28mhz**. Our critical path start from the node fetch:FETCH_PC|s_PC_current[7] and goes all the way to the node RegFile:u_RegFile|RegN:\gen_regs:17:u_reg|dffg:\Ndffg_Reg:18:dffg_N|s_Q.
The critical path highlight on our TopLevel schematic after observing the data arrival path:



It also went through the barrelshifter in the ALU which is not labeled by the highlight path.
The signals u_ALU|u_sh|s2RA[3]~58, s4RA[3]~23, etc., suggest a shift operation is occurring in the ALU.
Overview of the datapath:
Start: From FETCH_PC and s_PC_current[7] at 0.000 ns.
Instruction Fetch: Delays from instruction memory access up to 6.086 ns.
ALU Operations: Between 17 ns and 28 ns.
Data Memory Access: 36ns and 42ns
After that going into LoadType (41ns to 42ns) and then resultSrc (mux4t1 47ns to 48ns) before going to Register file as the end node (arrival at 50.4ns).
PC -> IMem -> ALU (barrel shifter) -> DMem -> LoadType -> resultSrc mux -> RegFile

To raise Fmax, we target the longest stretches in the path report: (1) the IMem fetch chain from FETCH_PC through s_IMemAddr and the IMem taps (between 3.7ns to 15.2ns) (2) the ALU operand routing and shifter from s_ALU_A into u_ALU (around 16.9ns to 28.1ns) (3) the DMem/load path (from 36.2ns to 41.9ns) (4) the write-back fan-in where Mux14~0 (most likely resultSrc) shows a 5.525ns incr. Therefore, the PC counter, instruction memory, ALU, data memory, and result source are the components that cause the longest delays and lag; thus, these are the components we would focus on to improve the frequency.