CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: _____Luke Auderer_____

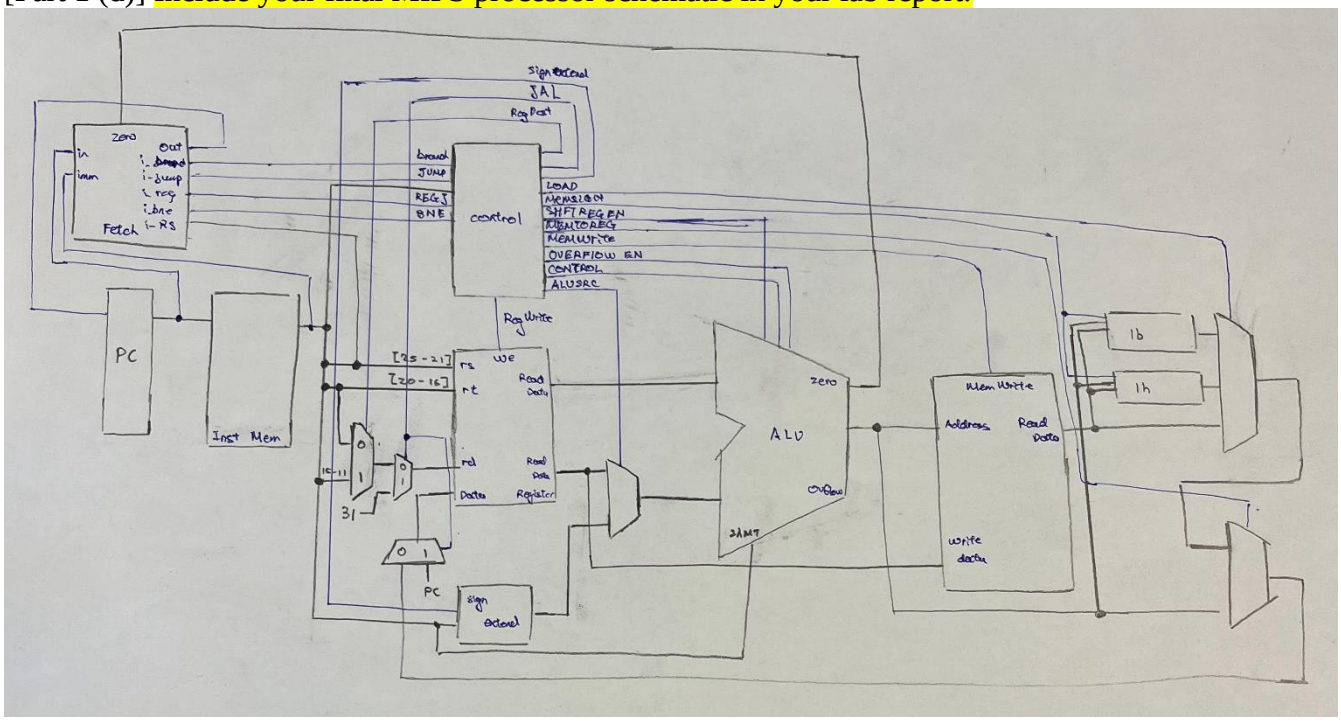
_____Yin Choong_____

_____

Project Teams Group #:___A_02_____

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.

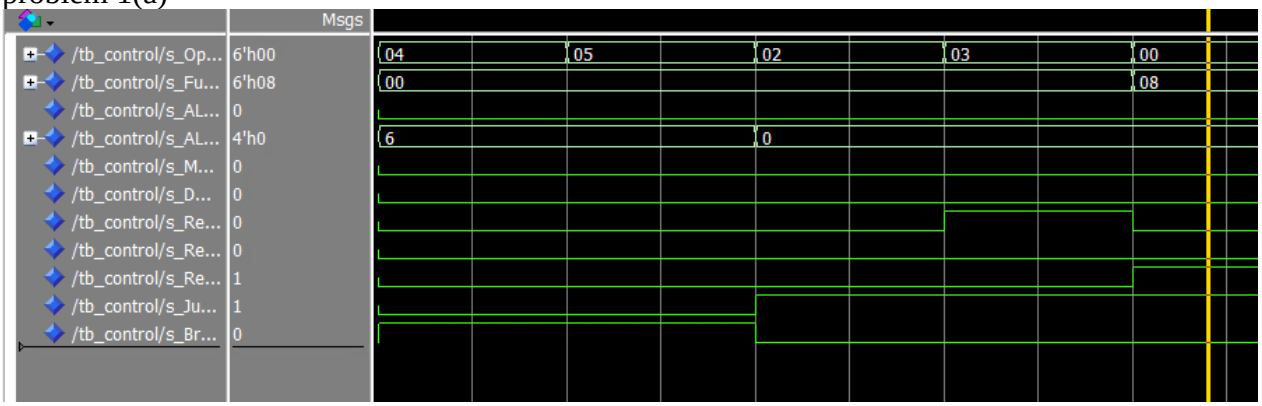


[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your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

Done.

	A	B	C	D	E	F	G	H	I	J	K	L
1								Control Signals				
2	Instruction	Opcode (Binary)	Funct (Binary)	ALUSrc	ALUControl	MemtoReg	s_DMemWr [MemWrite from text]	s_RegWr [RegWrite from text]	RegDst	Regjump	Jump	Branch
3	addi	"001000"	"——"	1 [use the appropriately extended immediate as source B]	0010 [the alu will perform an add of A and B]	0 [addi does NOT read from memory]	0 [addi does NOT write to memory]	1 [addi writes back to a register]	0 [addi uses rt as destination register rather than rd]			
4	add	"000000"	"010100"	0	"0010"	0	0	1	1			
5	addiu	"001001"	"——"	1	"0010"	0	0	1	0			
6	addu	"000000"	"010101"	0	"0010"	0	0	1	1			
7	and	"000000"	"011000"	0	"0000"	0	0	1	1			
8	andi	"001100"	"——"	1	"0000"	0	0	1	0			
9	lui	"001111"	"——"	1	"1110"	0	0	1	0			
10	lw	"100011"	"——"	1	"0010"	1	0	1	0			
11	nor	"000000"	100111	0	1100	0	0	1	1			
12	xor	"000000"	"100110"	0	1111	0	0	1	1			
13	xori	"001110"	"——"	1	1111	0	0	1	0			
14	or	"000000"	"100101"	0	"0001"	0	0	1	1			
15	ori	"001101"	"——"	1	"0001	0	0	1	0			
16	slt	"000000"	"101010"	0	0111	0	0	1	1			
17	slti	"001010"	"——"	1	"0111"	0	0	1	0			
18	sll	"000000"	"000000"	0	"1000	0	0	1	1			
19	srl	"000000"	"000010"	0	1001	0	0	1	1			
20	sra	"000000"	"000011"	0	1010	0	0	1	1			
21	sw	"101011"	"000000"	1	"0010	0	1	0	0			
22	sub	"000000"	"100010"	0	0110	0	0	1	1			
23	subu	"000000"	"100011"	0	"0110	0	0	1	1			
24	beq	"000100"	"——"	0	"0110	0	0	0	0			1
25	bne	"000101"	"——"	0	"0110	0	0	0	0			1
26	j	"000010"	"——"	0	"0000"	0	0	0	0		1	
27	jal	"000011"	"——"	0	0	0	0	1	0		1	
28	jr	"000000"	"001000"	0	0	0	0	0	0	1	1	
29	lb	"100000"	"——"	1	"0010	1	0	1	0			
30	lh	"100001"	"——"	1	10	1	0	1	0			
31	lbu	"100100"	"——"	1	"0010	1	0	1	0			
32	lhu	"100101"	"——"	1	"0010	1	0	1	0			
33	sllv	"000000"	"000100"	0	1000	0	0	1	1			
34	srlv	"000000"	"000110"	0	"1001"	0	0	1	1			
35	srav	"000000"	"000111"	0	"1010"	0	0	1	1			
36												

[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).
Done, as can be seen below, the output matches the expected control signals from problem 1(a)



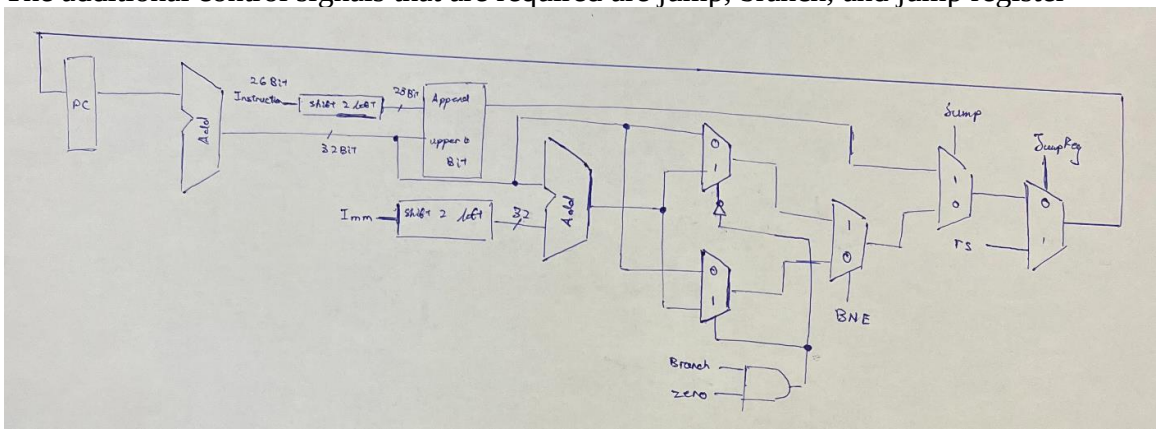
[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.
The main output of the fetch unit is setting the PC. There are three different outputs: PC+4, branch, jump (technically jump registers as well but we will group it with jump)

- PC + 4 – This is the default case, where the next instruction is simply the next sequential instruction in memory. It occurs when there is no branch or jump instruction.
- Branch Target Address – This occurs when a branch instruction (e.g., beq, bne) is taken. The new PC is calculated as PC + 4 + (Sign-Extended Immediate << 2). If the branch condition is not met, the PC remains PC + 4.
- Jump Target Address – This occurs for jump instructions (e.g., j, jal). The new PC is set to (PC + 4) upper 4 bits concatenated with (Jump Immediate << 2).
    - Jump Register Target Address – This occurs for jr (jump register) instructions, where the PC is set to the value stored in a register (e.g., PC =

Register[rs]). This is different from a normal jump because the target address is determined dynamically rather than being encoded in the instruction.

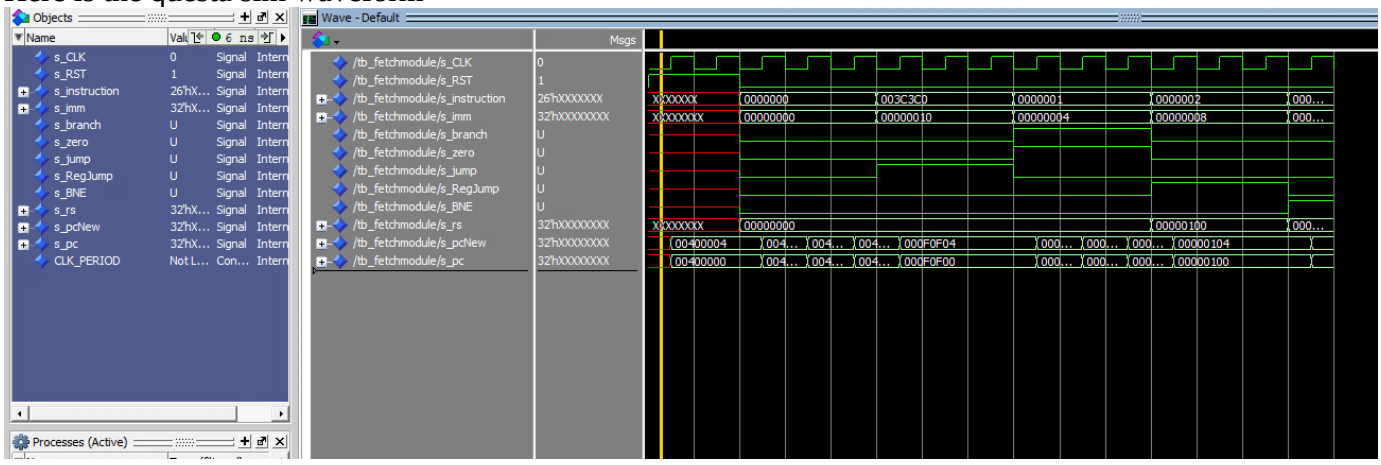
[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

The additional control signals that are required are jump, branch, and jump register



[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.

Here is the questa sim waveform



The instruction fetch unit is responsible for updating the program counter (PC) based on the type of instruction executed. The possible updates are:

1. PC + 4 (Sequential Execution): Default case where the program counter increments by 4.
2. Branch Target Address: If a branch instruction (e.g., beq, bne) is taken, PC is updated using PC + 4 + (Sign-Extended Immediate << 2).

3. Jump Target Address: For j and jal instructions, the new PC is (PC + 4) upper 4 bits concatenated with (Jump Immediate << 2).
4. Jump Register (jr): The PC is set to the value stored in register rs

These possibilities are seen in the waveform as
1. Sequential Execution (PC + 4)
Expected Behavior: The PC increments by 4 at each clock cycle.

Waveform Observation: o_pcNew changes from 0x00000000 → 0x00000004 → 0x00000008 ...

2. Jump Instruction (J)
Expected Behavior: When i_jump is high, the PC jumps to (PC + 4) upper 4 bits concatenated with (Jump Immediate << 2).
Waveform Observation: A sudden jump in o_pcNew, confirming that the jump instruction was correctly executed.

3. Branch Taken (BEQ)
Expected Behavior: When i_branch is high and i_zero = 1, the PC updates to PC + 4 + (Sign-Extended Immediate << 2).
Waveform Observation: o_pcNew skips the usual PC + 4 and takes a new target address when the condition is met.

4. Jump Register (JR)
Expected Behavior: When i_RegJump = 1, the PC updates to i_rs (the value of the register).
Waveform Observation: o_pcNew takes the value of i_rs, confirming a correct register jump.

5. Branch Not Equal (BNE)
Expected Behavior: When i_BNE is high and i_zero = 0, the PC updates to PC + 4 + (Sign-Extended Immediate << 2).
Waveform Observation: Similar to BEQ, but the condition is zero = 0. If the condition is met, a jump occurs; otherwise, execution continues sequentially.

The QuestaSim waveforms confirm that the fetch logic is correctly implemented.

[Part 2 (c.i.1)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?
The difference between logical shift right (srl) and arithmetic shift right (sra) lies in how they handle the most significant bit (MSB), which represents the sign in a signed integer.

1. Logical Shift Right (srl)
    - Zeros are shifted into the MSB regardless of the original value.
    - Used for unsigned numbers because it does not preserve the sign bit.
    - Example: srl $t0, $t1, 2 shifts the bits of $t1 two places to the right, inserting zeros at the left.

2. Arithmetic Shift Right (sra)
    - The MSB is preserved, meaning the sign bit is extended when shifting.
    - Used for signed numbers because it maintains the sign of the original value.
    - Example: If $t1 holds 0b10000000 (-128 in 8-bit signed representation), shifting it right using sra keeps the MSB as 1 to maintain the negative value.

Why MIPS Does Not Have sla
MIPS does not include a sla instruction because a left shift in both signed and unsigned numbers behaves the same—zeros are always shifted into the least significant bit (LSB). This operation is equivalent to sll, which is why MIPS does not need a separate sla instruction. Additionally, left shifts in two's complement arithmetic can cause overflow, so compilers and programmers handle multiplication by powers of two carefully to avoid unintended results.

[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.
In my VHDL implementation, both logical shift and arithmetic shift operations are handled by:

Logical Left Shift (SLL):
When i_shiftright = '0', the code performs a left shift using shift_left(unsigned(i_data), to_integer(unsigned(i_shiftamount))).
This shifts i_data left by i_shiftamount bits, filling the least significant bits (LSBs) with zeros.

Logical Right Shift (SRL):
When i_shiftright = '1' and i_arithmetic = '0', the code executes a logical right shift using shift_right(unsigned(i_data), to_integer(unsigned(i_shiftamount))).
Since i_data is treated as an unsigned value, zeroes are inserted in the most significant bits (MSBs).

Arithmetic Right Shift (SRA):
When i_shiftright = '1' and i_arithmetic = '1', the code executes an arithmetic right shift using shift_right(signed(i_data), to_integer(unsigned(i_shiftamount))).
This treats i_data as a signed value, ensuring sign extension (i.e., the MSB is propagated to preserve the sign for negative numbers).

[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 enhance the right barrel shifter to also support left shifting operations, I would modify the multiplexer connections and introduce an additional control signal to determine the shift direction. Here's how I would approach it:

Dual Multiplexer Paths for Left and Right Shifting:
Instead of hardwiring the multiplexers only for right shifting, I would modify them to select between left and right shift operations.
This can be achieved by adding a direction control signal (DIR) that determines whether the shift occurs left (DIR = 1) or right (DIR = 0).

Rewiring Multiplexer Inputs:
In the existing design, each multiplexer in stage 1 shifts the input one bit to the right, stage 2 shifts by two bits, and stage 3 shifts by four bits.
To support left shifts, I would add additional inputs to each multiplexer so that, when DIR = 1, they take inputs from the next-higher bit instead of the next-lower bit.

This means that each stage would be like this:
Stage 1: Shifts left by 1 when SHL_1 is active, right by 1 otherwise.
Stage 2: Shifts left by 2 when SHL_2 is active, right by 2 otherwise.
Stage 3: Shifts left by 4 when SHL_4 is active, right by 4 otherwise.

Zero Padding for Left and Right Shifts:
For right shifts, the least significant bits (LSBs) are filled with zeroes, as in the current design.
For left shifts, the most significant bits (MSBs) need to be zeroed out instead. This ensures that data integrity is maintained in both directions.

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

The execution of the different shifting operations in the testbench corresponds to the changes observed in the QuestaSim waveforms as follows: for the logical shift right (Test case 2), the bits are shifted to the right with zero-padding as expected. In the arithmetic shift right (Test case 3), the sign extension is applied during the shift, preserving the sign bit. The waveforms show the correct shift amounts and results based on the shift operation (logical or arithmetic) and the input values provided in the test cases.

[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.

These are the components I used in my ALU

Shifter:

Design approach: For the Shifter design, I focused on implementing both logical and arithmetic shifts, depending on the control signals i_shiftright and i_arithmetic. I used VHDL's shift_left and shift_right operators, taking care to handle both unsigned and signed data based on the arithmetic shift mode. The shift amount is extracted from the 32-bit input, but only the lower 5 bits are used for the actual shift operation, allowing for efficient handling of shift ranges.

One design decision: A key design decision was choosing how to handle the arithmetic right shift. I decided to distinguish between logical and arithmetic shifts by checking the i_arithmetic signal and using signed versus unsigned types for the right shifts. This ensures that the shifting behavior properly accounts for sign extension when performing arithmetic shifts.

Adder/Subtractor:

Design approach: The Adder/Subtractor is implemented using a full adder structure with a carry-in signal that handles both addition and subtraction. When performing subtraction, the second input is complemented using a ones' complement module, and a carry-in of '1' is applied. This ensures that the subtraction operation is properly handled by adding the two's complement of the second operand.

One design decision: A key design decision was choosing to implement subtraction using the two's complement method by inverting the bits and adding one. This approach simplifies the logic by reusing the adder for both addition and subtraction, reducing complexity in the design.

AND Gate (32-bit):

Design approach: The AND gate for the 32-bit operation is implemented using a simple bitwise AND operation, where each bit of the two 32-bit inputs is ANDed individually.

One design decision: The decision to implement the AND operation as a simple bitwise logic operation was straightforward because it provides the most efficient implementation

OR Gate (32-bit):

Design approach: Similar to the AND gate, the OR gate is implemented using a bitwise OR operation. Each bit of the two 32-bit inputs is ORed individually, producing a 32-bit output

One design decision: The OR operation is also implemented with bitwise logic for simplicity and efficiency.

XOR Gate (32-bit):

Design approach: The XOR operation is performed using a bitwise XOR function, where each bit from the two 32-bit inputs is XORed individually. This produces a 32-bit output, with each bit being the logical XOR of the corresponding bits from the inputs.

One design decision: The bitwise XOR operation was chosen because it directly supports the functionality required by the ALU, and it can be efficiently implemented in hardware using standard logic gates.

One's Complement:
Design approach: The one's complement operation is implemented using an inverter, which simply flips all the bits of the input. This operation is used in the ALU to compute the two's complement of a number for subtraction.
One design decision: Implementing the one's complement as an inversion of the bits was chosen to simplify the design and ensure efficient performance. This choice also allowed for easy integration with the Adder/Subtractor for the subtraction operation.

Zero Flag:
Design approach: The Zero Flag is calculated by checking if the output of the ALU is zero. If all bits in the 32-bit output are '0', the Zero Flag is set to '1', indicating that the result of the operation was zero.
One design decision: The decision to use a separate component to check the ALU's output for zero was made to isolate the flag logic and make the design more modular. This also helps maintain clarity in the ALU's operation.

Overflow Flag:
Design approach: The Overflow Flag is calculated based on the result of the addition/subtraction operation. If there is a carry into the most significant bit that is not properly propagated, an overflow occurs. The flag is enabled using an AND gate, combining the overflow signal from the adder with the external enable signal.
One design decision: A decision had to be made regarding when to enable the overflow flag. I decided to use a dedicated overflow enable signal to control whether the overflow flag is active, which allows for more flexible operation depending on the specific needs of the ALU.

Multiplexer (2-to-1 and 16-to-1):
Design approach: The 2-to-1 multiplexer selects between two inputs based on a control signal, used for operations like choosing between shifted data and the original input. The 16-to-1 multiplexer selects among 16 possible inputs based on the control signals to choose the correct ALU operation output.
One design decision: The choice to use multiplexers for selecting between different operations (AND, OR, XOR, Adder/Subtractor, etc.) was made to efficiently manage the different operations and control the output based on the control signals, minimizing the need for more complex logic. The multiplexer size was chosen based on the number of operations in the ALU.

<u>These are the major components I used</u>
Fetch Module:
Design approach: The Fetch Module design follows a structural approach using multiple interconnected components to manage program counter (PC) updates efficiently. The design leverages modularity with arithmetic units, multiplexers, and logical gates to

handle different control signals like branching, jumping, and resets. Resources such as the MIPS instruction set architecture and digital design principles guided the implementation.

One design decision: A key design decision was how to handle branch instructions efficiently. Instead of a single branch calculation, the design includes separate logic for both standard branching (i_branch) and branch-not-equal (i_BNE). This required adding an inverter and an additional multiplexer to ensure proper selection of the next PC value when a branch is taken.

Control:
Design approach: The control unit was designed using a dataflow architecture to map opcode and function code inputs to control signals efficiently. The design follows a straightforward approach of using conditional signal assignments to determine the appropriate control outputs. To ensure correctness, I referenced MIPS instruction set documentation and logic design principles for control signal generation. The use of concurrent signal assignment simplifies the design by avoiding complex process blocks.

One design decision: A key design decision was how to implement ALU control logic efficiently. Instead of using a process block with case statements, I opted for direct conditional assignments to determine o_ALUControl. This approach makes the design more readable and allows for easier expansion in the future when adding new instructions. However, it also required careful structuring to ensure that priority conditions were correctly assigned.

ALU:
Design approach: For the ALU design, I adopted a modular and hierarchical approach, using a combination of custom components like multiplexers, adders, and shift registers. I relied on VHDL for structural modeling, ensuring each ALU operation (addition, bitwise operations, shifting, etc.) was implemented as a separate component for flexibility and scalability. Resources like the IEEE std_logic libraries and VHDL documentation were utilized to structure the ALU with proper bit-widths and operations.

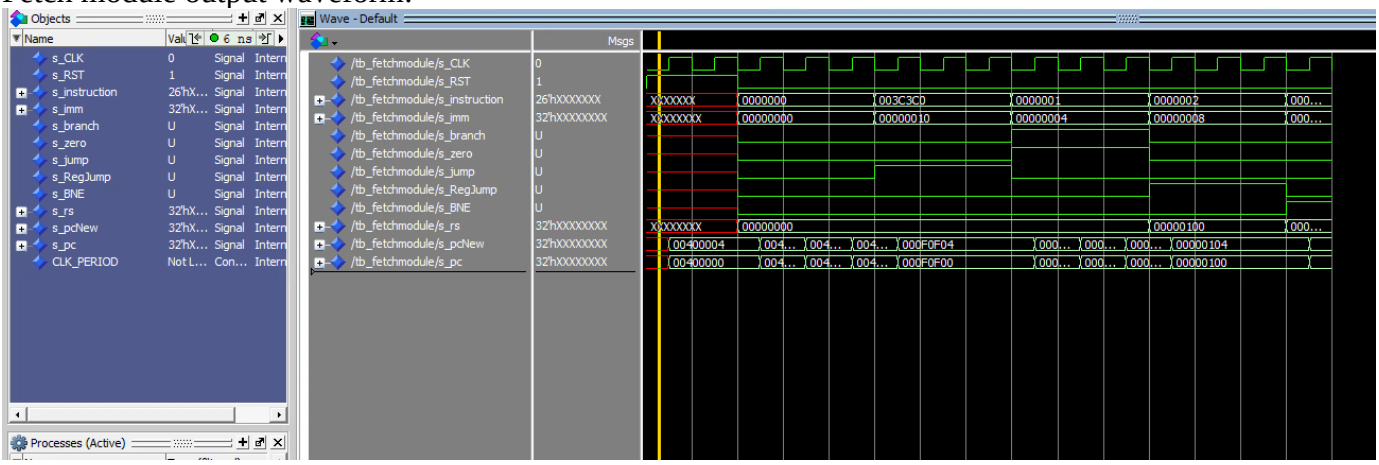
One design decision: A key design decision was selecting between using a 2-to-1 multiplexer (mux2t1_N) or a more complex multiplexer for the shift amount selection. I opted for the 2-to-1 multiplexer to control whether the shift amount is from the decoder or directly from the input based on the i_shiftregEN signal

Load variations (lb, lh)
Design approach: The load variations (lb and lh) modules are designed to handle byte and half-word loads with sign extension or zero extension based on the control signals. The lb_module extracts a byte from a 32-bit input and extends it to 32 bits based on the i_sign signal, while the lh_module handles half-word extraction and extension. For both, I used VHDL's resize function to perform the extension, ensuring correct handling of both signed and unsigned data.

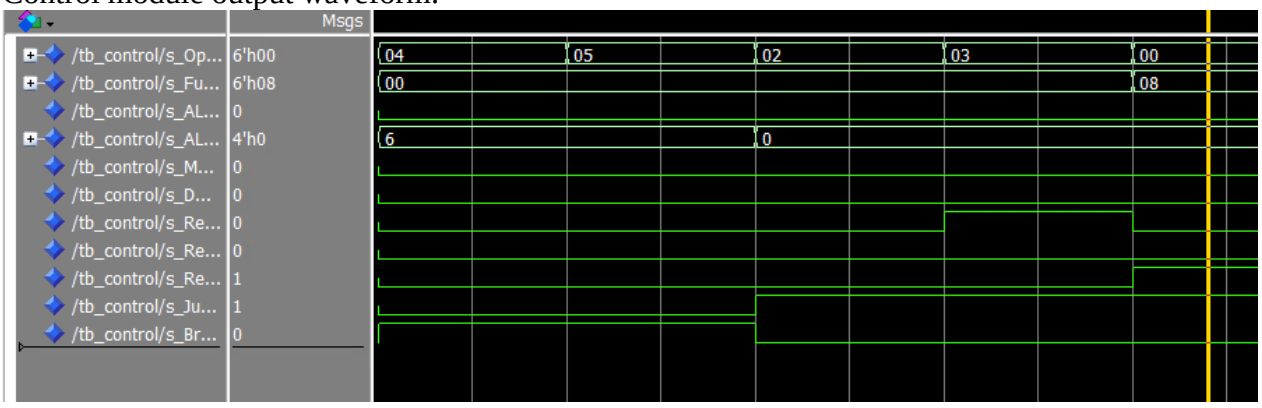
One design decision: A key design decision was choosing how to handle sign extension. For both lb and lh modules, I decided to use the resize function to either sign-extend or zero-extend the extracted byte or half-word, depending on the value of the i_sign control signal. This ensures that the extension behavior is consistent with the desired load type (signed or unsigned).

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.
Note: I talked with the Cory (TA) and he said I just need the waveforms of the major components for this question.
These are the output waveforms for each of the major components I described above. The output waveforms show the beavior I described above and all are working correctly.

Fetch module output waveform:

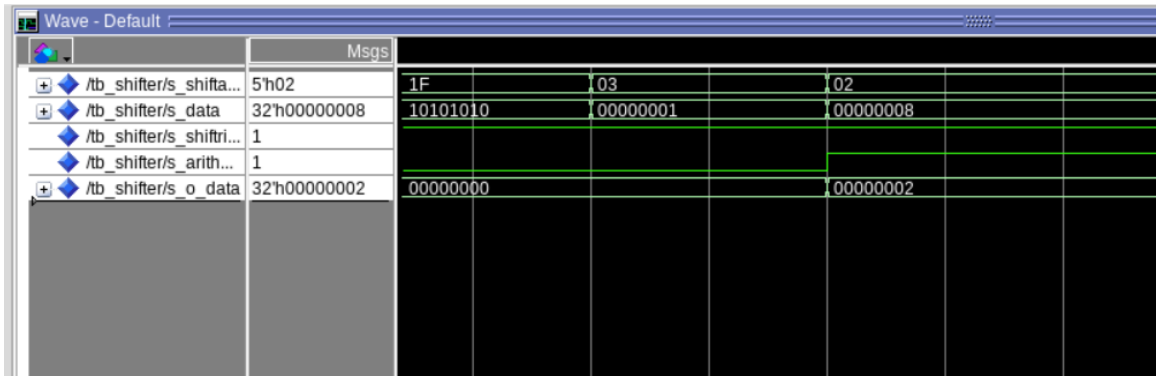


Control module output waveform:

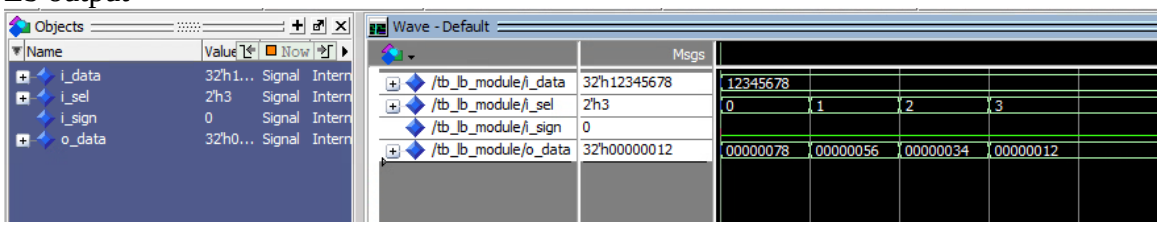


ALU output waveform

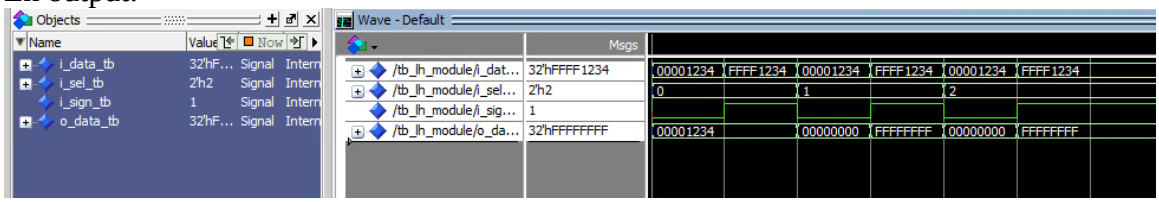
Shifter output waveform



Lb output

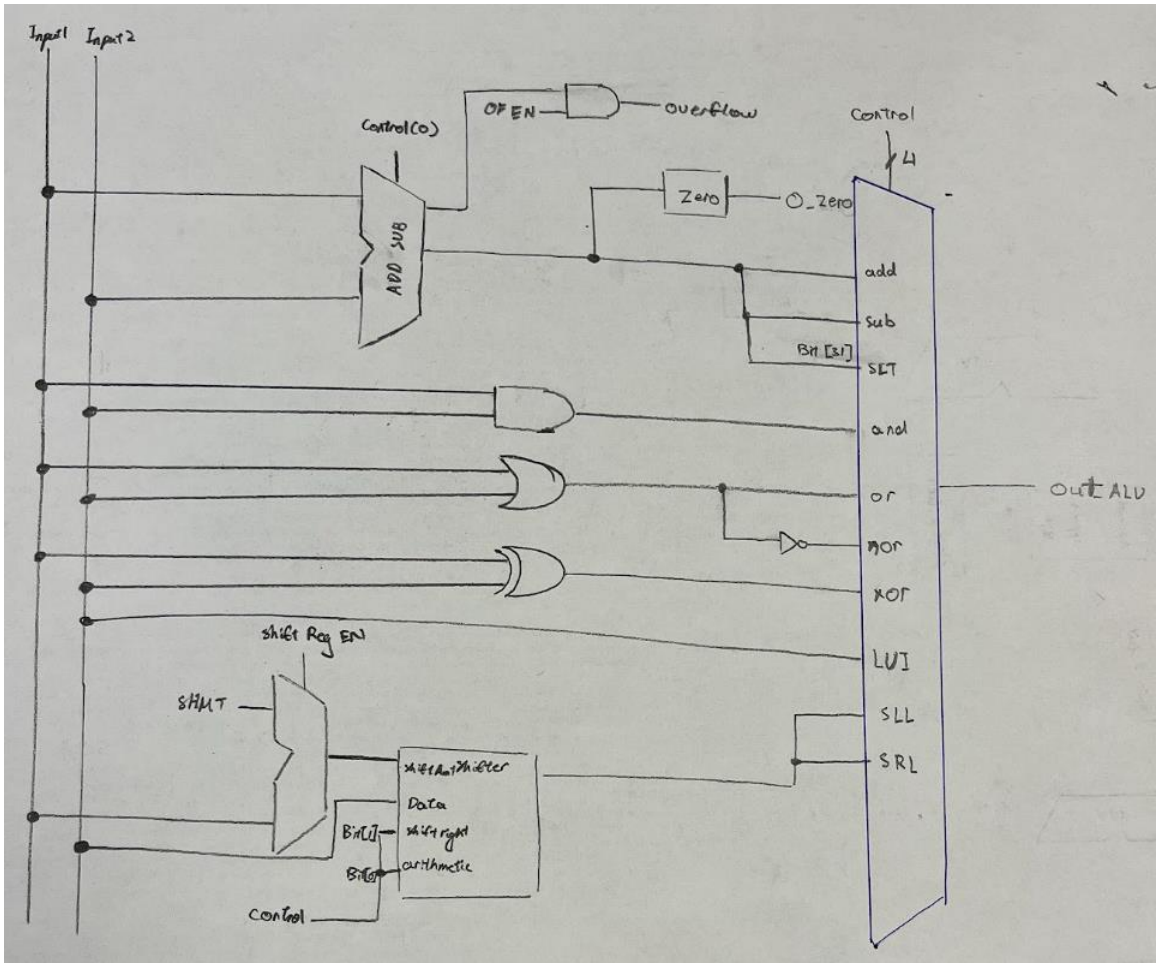


Lh output:



[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?

Schematic



Overflow Calculation: Overflow occurs when the result of an arithmetic operation exceeds the representable range of the ALU. In this ALU, overflow is primarily associated with addition and subtraction operations. The nAdder_Sub component performs addition or subtraction of i_input1 and i_input2, and the i_C_out signal from this adder indicates if an overflow has occurred. This output is then ANDed with the i_overflowEN signal in the andg2 component to determine if overflow should be enabled. The result is then output through o_overflow.

Zero Calculation: The Zero flag is set if the output of the ALU is zero, which would indicate that the result of the operation was zero. In this implementation, the ZeroFlag component is responsible for checking the s_output (the ALU result) and setting the o_zero flag to 1 if the result is zero. This is done through simple comparison in the ZeroFlag component, which compares the 32-bit result and generates the zero_flag output accordingly.

Set Less Than (slt) Implementation: The Set Less Than (slt) operation compares the two inputs and sets the result to 1 if the first input is less than the second, and 0 otherwise. This comparison is performed using the signed function to treat the inputs as signed numbers. The comparison is made in the s_lessthan signal, which is assigned a value of

x"00000001" if i_input1 is less than i_input2 (after treating them as signed values), otherwise it is set to x"00000000". This result is then passed through the ALU's mux logic (s_mux_input(7)) to be included in the final output, depending on the control signals. If the i_control signals select the s_lessthan result, it will reflect whether i_input1 is less than i_input2.

[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.
The execution of different ALU operations in the testbench directly corresponds to the waveforms generated in QuestaSim by showing the changing values of inputs and outputs over time. For example, when performing the addition operation, the waveform will display the progression of i_input1 and i_input2 values, followed by the resulting sum in o_output. Similarly, for the shift operations, the waveform will show the bit shifting process, where i_input1 shifts to the left or right based on the control signal. The overflow and zero detection are also reflected in the waveforms, with changes in o_overflow and o_zero when conditions for overflow or zero results are met.

[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.
In this testbench, I have designed multiple test cases to thoroughly verify the functionality of the ALU's arithmetic and logic operations, covering both normal and edge cases. Here's a breakdown of the approach:

Basic Operations: I tested the basic arithmetic operations (addition, subtraction) and logic operations (OR, SLT). These operations cover the fundamental use cases for the ALU. For example:

Overflow Handling: A specific test case (Overflow 8 with 1) is included to check the overflow detection functionality. This ensures the ALU's overflow detection logic works properly, specifically when the sum exceeds the maximum value a 32-bit signed integer can hold.

Shift Operations: Multiple shift operations are tested:
SLL (Shift Left Logical) tests that the ALU can correctly shift the bits to the left and fill the new positions with 0.
SRA (Shift Right Arithmetic) checks the arithmetic shift right, which should extend the sign bit when shifting right.

Comparison: The SLT (Set Less Than) operation tests the comparison functionality of the ALU, ensuring it can correctly determine whether one value is less than another and return the appropriate result.

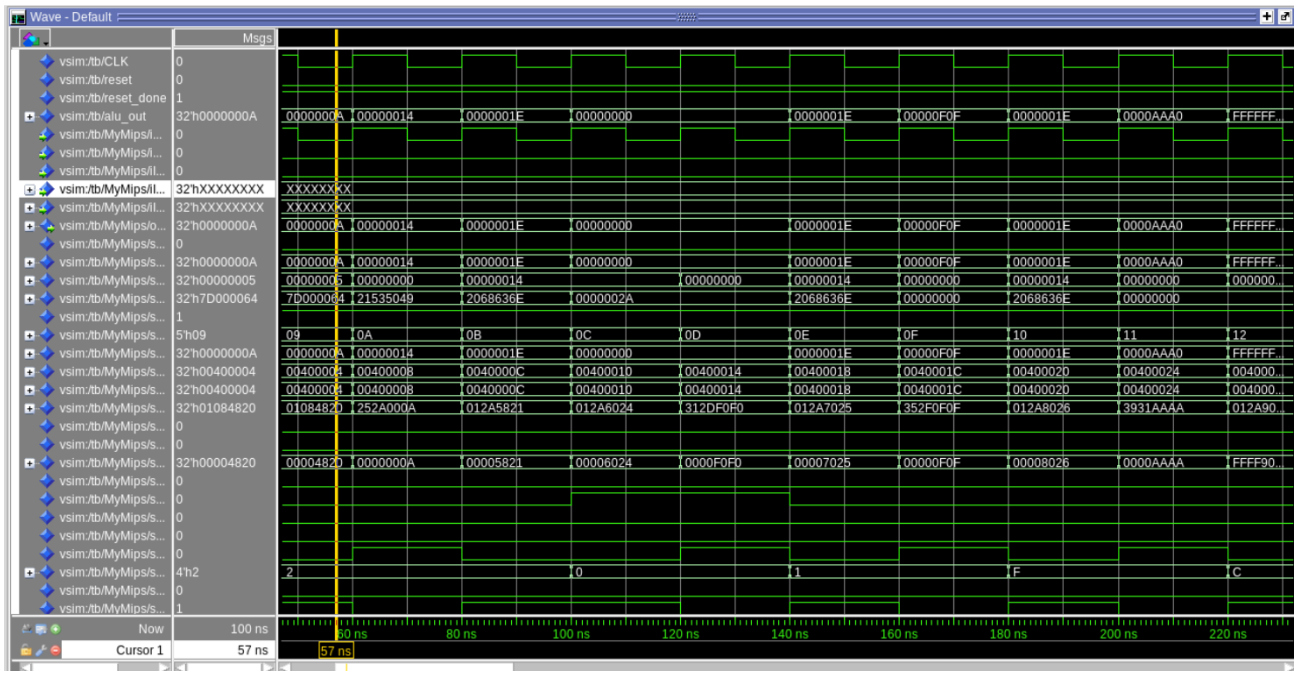
Edge Cases: Each test case incorporates edge values such as 0x7FFFFFFF for overflow testing, ensuring that the ALU behaves as expected under boundary conditions. These

cases test the full range of inputs, ensuring robustness against edge cases like overflow or sign extension.

This test plan is comprehensive because it covers not only basic functionality but also edge cases

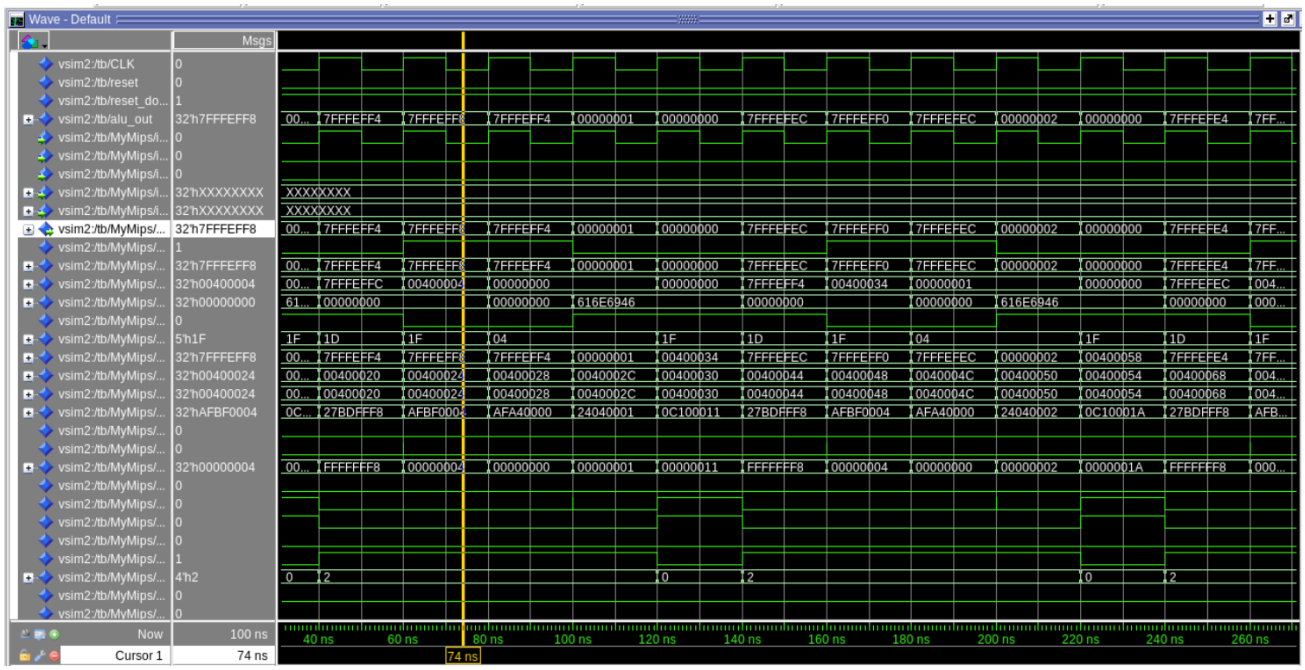
This is the waveform

[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.
Done.



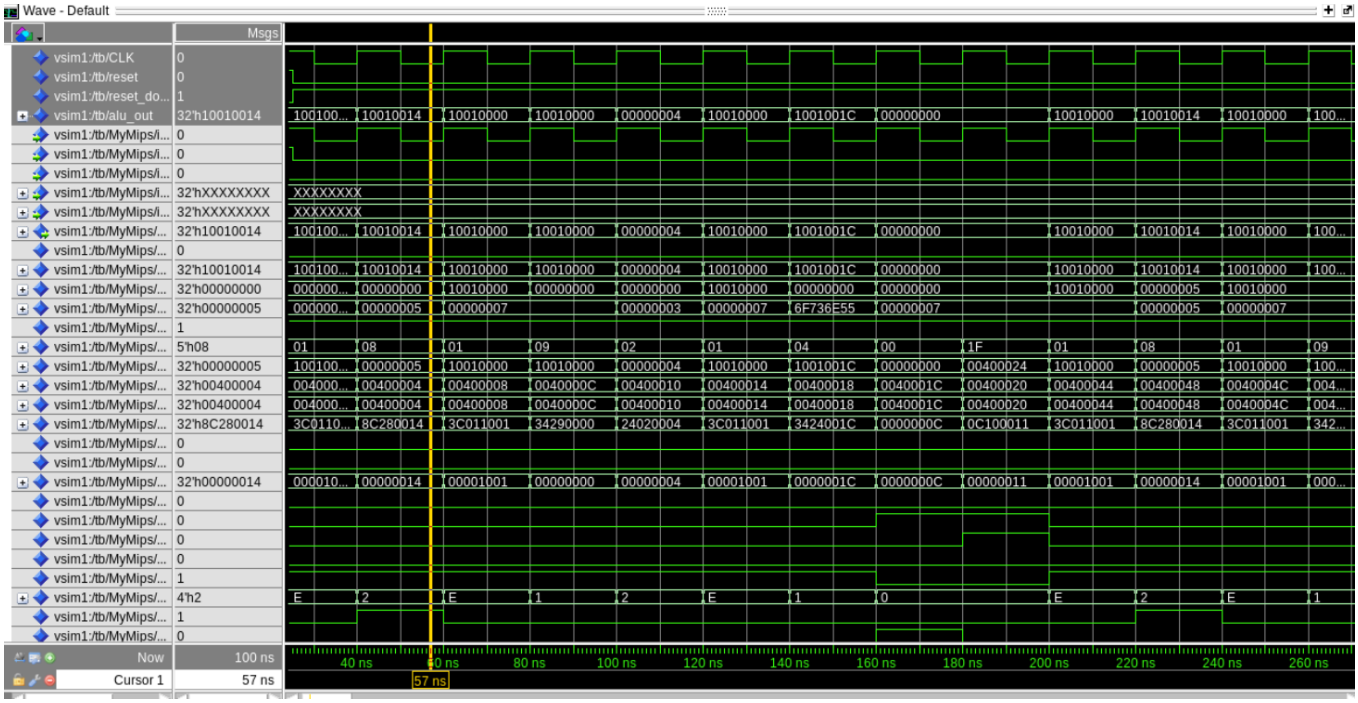
[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.

Done.



[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.

Done.



[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 is 21.69 mhz

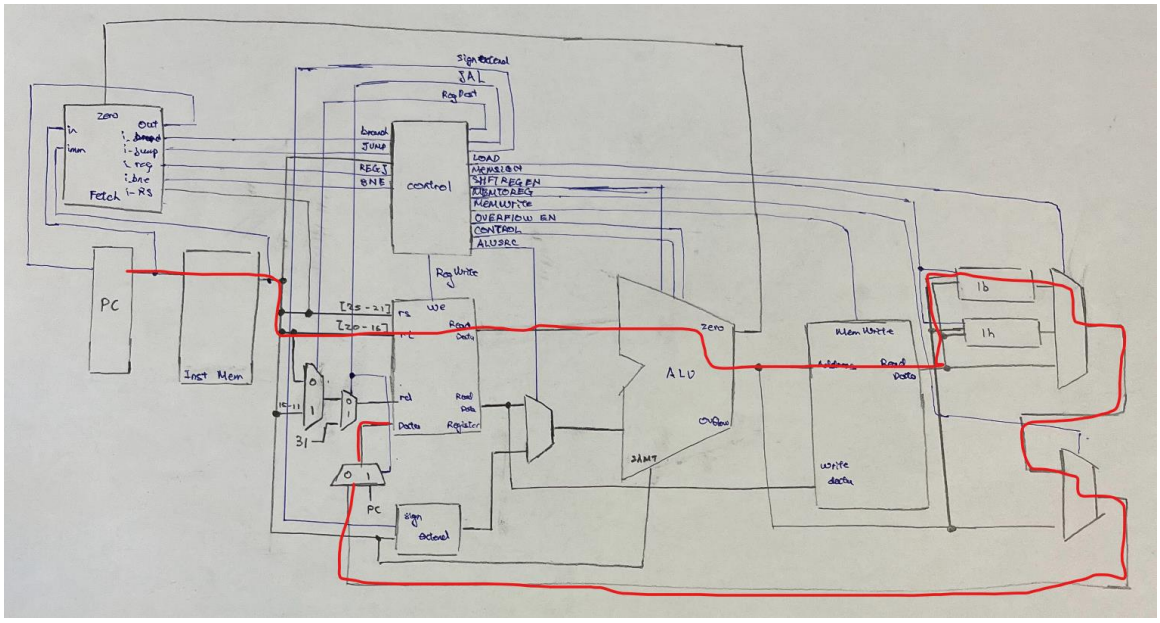
```
#
# CprE 381 toolflow Timing dump
#

FMax: 21.69mhz Clk Constraint: 20.00ns Slack: -26.11ns

The path is given below

 ================================================================
 From Node    : Fetchmodule:Fetch|N_Reg:g_pc|dffg:\NBit_DFF:2:dffi|s_Q
 To Node      : MIP_REG:MainRegister|N_Reg:\g_reg:31:g_regOther:g_regi|dffg:\NBit_DFF:18:dffi|s_Q
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
 ==========  ========= ==  ====  ==================================
       0.000     0.000           launch edge time
       3.095     3.095  R        clock network delay
       3.327     0.232     uTco  Fetchmodule:Fetch|N_Reg:g_pc|dffg:\NBit_DFF:2:dffi|s_Q
       3.327     0.000 FF  CELL  Fetch|g_pc|\NBit_DFF:2:dffi|s_Q|q
       3.674     0.347 FF    IC  s_IMemAddr[2]~6|datad
       3.799     0.125 FF  CELL  s_IMemAddr[2]~6|combout
       5.956     2.157 FF    IC  IMem|ram~42616|datab
       6.379     0.423 FR  CELL  IMem|ram~42616|combout
       6.582     0.203 RR    IC  IMem|ram~42617|datad
       6.737     0.155 RR  CELL  IMem|ram~42617|combout
       7.481     0.744 RR    IC  IMem|ram~42620|datac
       7.768     0.287 RR  CELL  IMem|ram~42620|combout
      10.246     2.478 RR    IC  IMem|ram~42623|datab
      10.624     0.378 RF  CELL  IMem|ram~42623|combout
      10.859     0.235 FF    IC  IMem|ram~42655|datac
      11.139     0.280 FF  CELL  IMem|ram~42655|combout
      11.372     0.233 FF    IC  IMem|ram~42656|datac
      11.653     0.281 FF  CELL  IMem|ram~42656|combout
      13.289     1.636 FF    IC  IMem|ram~42699|datac
      13.570     0.281 FF  CELL  IMem|ram~42699|combout
      13.795     0.225 FF    IC  IMem|ram~42870|datad
      13.920     0.125 FF  CELL  IMem|ram~42870|combout
      14.155     0.235 FF    IC  IMem|ram~43041|datac
      14.436     0.281 FF  CELL  IMem|ram~43041|combout
      15.239     0.803 FF    IC  MainRegister|g_mux1|Mux17~12|datad
      15.389     0.150 FR  CELL  MainRegister|g_mux1|Mux17~12|combout
      15.593     0.204 RR    IC  MainRegister|g_mux1|Mux17~13|datad
      15.748     0.155 RR  CELL  MainRegister|g_mux1|Mux17~13|combout
```

Critical Path:



We would focus on improving these things: our ALU is taking a lot of time and our JumpandLinkMUX is taking a lot of time. If we were to improve these and make them faster/more efficient to save time.