CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members: Joseph Barnes III

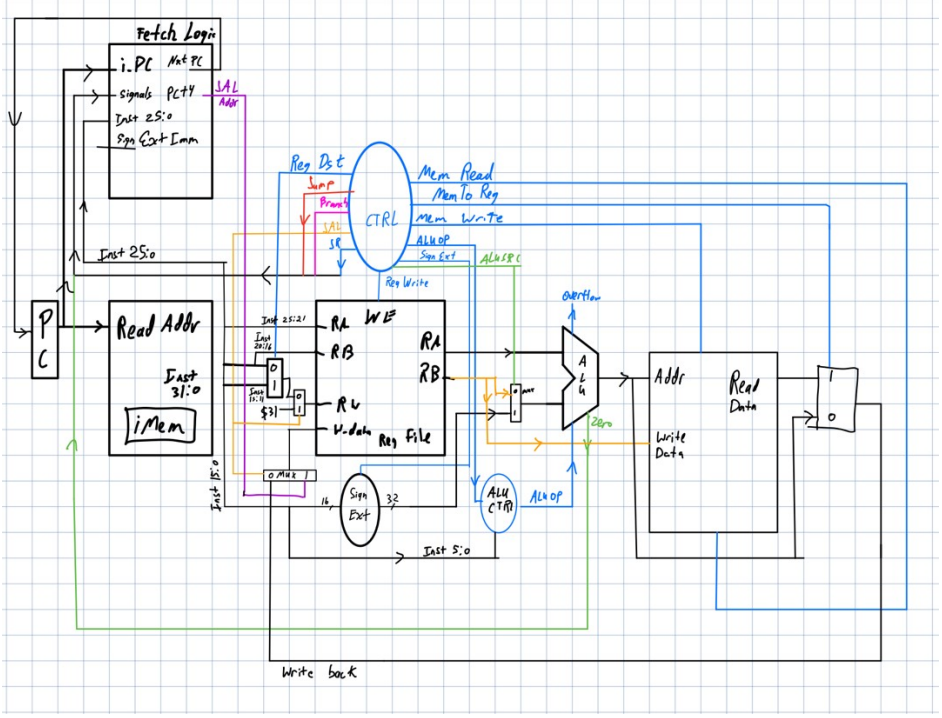
Josh Arceo

Evan Shiber

Project Teams Group #: Sec C 03

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

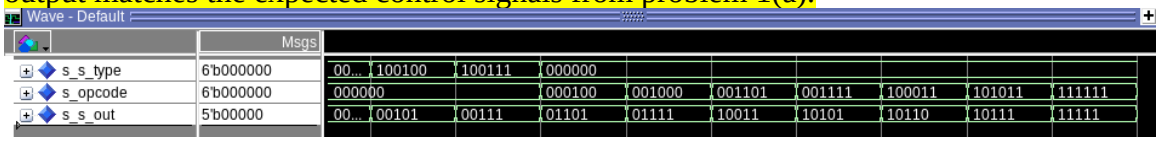
[Part 1 (d)] Include your final MIPS processor schematic in your lab report.



[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your datapath implementation. The end result should be an *N*M* table where each row corresponds to the output of the control logic module for a given instruction.

Instruction	Opcode (Binary)	Funct (Binary)	Control Signals													
			ALUSrc	ALUControl	MemtoReg	s_DMemWr (MemWrite from text)	s_RegWr (RegWrite from text)	RegDst	JAL	JR	Jump	Branch	BNE	Halt	SignExt	Shift
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)	X	X	X	X	X	0	1	0
add	"000000"	"100000"*	0	0010 (the alu will perform an add of A and B)	0	0	1	1	0	0	0	0	0	0	0	0
addiu	"001001"	"------"	1	0100	0	0	1	0	0	0	0	0	0	0	1	0
and	"000000"	"100100"	0	0000	0	0	1	1	0	0	0	0	0	0	0	0
andi	"001100"	"------"	1	0000	0	0	1	0	0	0	0	0	0	0	1	0
lui	"001111"	"------"	1	0100	0	0	1	0	0	0	0	0	0	0	0	0
lw	"000000"	"100011"	0	0010	1	0	1	0	0	0	0	0	0	0	0	0
nor	"000000"	"100111"	0	1101	0	0	1	0	0	0	0	0	0	0	0	0
xor	"000000"	"100110"	0	0101	0	0	1	0	0	0	0	0	0	0	0	0
xori	"001110"	"------"	1	0101	0	0	1	0	0	0	0	0	0	0	0	0
or	"000000"	"100101"	0	0001	0	0	1	1	0	0	0	0	0	0	0	0
ori	"001101"		1	0001	0	0	1	0	0	0	0	0	0	0	0	0
slt	"000000"	"101010"	0	0111	0	0	1	1	0	0	0	0	0	0	0	0
slti	"001010"	"------"	1	0111	0	0	1	0	0	0	0	0	0	0	0	0
sll	"000000"	"000000"	1	1001	0	0	1	1	0	0	0	0	0	0	0	1
srl	"000000"	"000010"	1	1010	0	0	1	1	0	0	0	0	0	0	0	1
sra	"000000"	"000011"	1	1011	0	0	1	1	0	0	0	0	0	0	0	1
sw	"000000"	"101011"	1	0010	0	1	0	0	0	0	0	0	0	0	0	0
sub	"000000"	"100010"	0	0110	0	0	1	1	0	0	0	0	0	0	0	0
subu	"000000"	"100011"	0	1000	0	0	1	1	0	0	0	0	0	0	0	0
beq	"000100"	"------"	0	0110	0	0	0	0	0	0	0	1	0	0	0	0
bne	"000101"	"------"	0	0110	0	0	0	0	0	0	0	0	1	0	0	0
j	"000010"	"------"	1	XXXX	0	0	0	0	0	0	1	1	0	0	0	0
jal	"000011"	"------"	1	XXXX	0	0	1	0	1	0	1	0	0	0	0	0
jr	"000000"	"001000"	0	XXXX	0	0	0	0	0	1	0	0	0	0	0	0

[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually, and show that your output matches the expected control signals from problem 1(a).



[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 fetch logic needs to be able to manage three different control types:

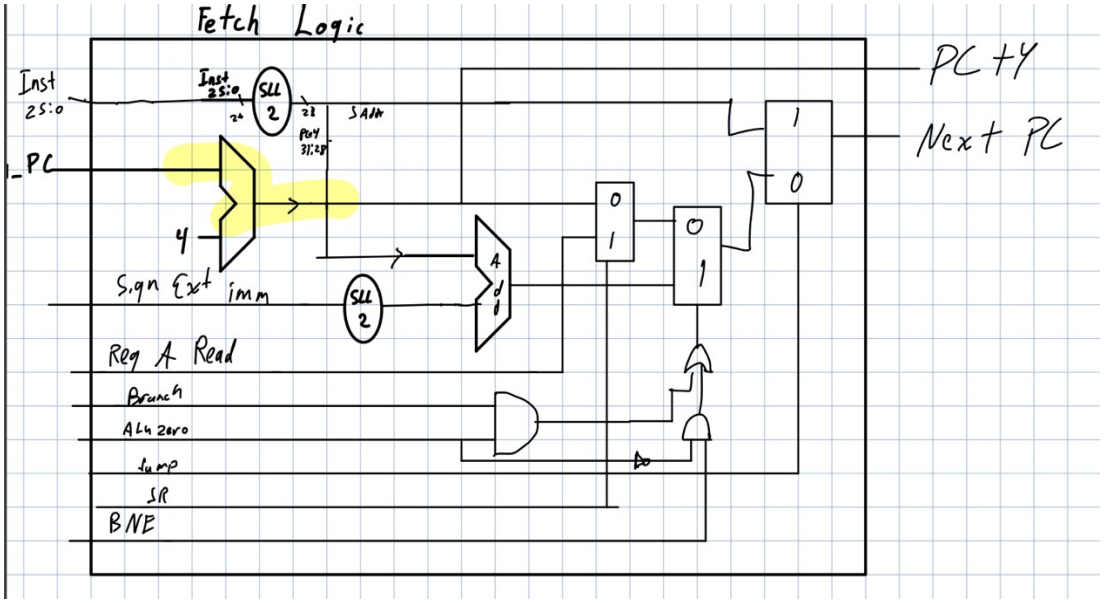
- branch
- jump
- regular increment (PC+4 for most instructions).

It is necessary that it supports both the branch and jump commands.

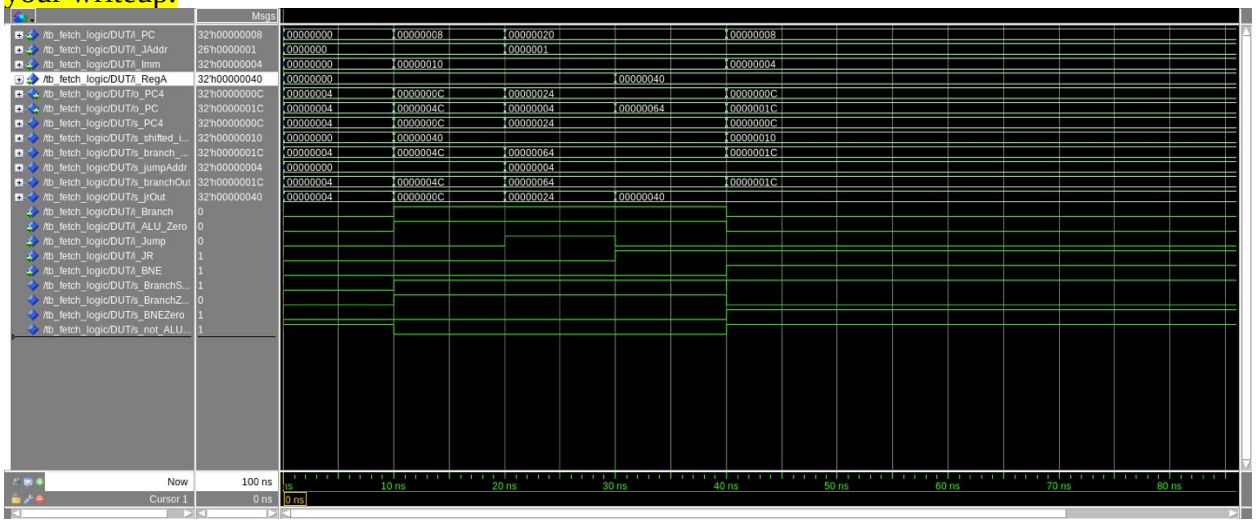
- The regular increment is utilized for all other command sets.

Additionally, a *halt* instruction (which assists in halting program execution process when required) should have been included in the fetch 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?



[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use Modelsim to test your design thoroughly to make sure it is working as expected. Describe how the execution of the control flow possibilities corresponds to the Modelsim waveforms in your writeup.



[Part 2 (c.i.1)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does MIPS not have a `sla` instruction?

SRL shifts right and replaces the shifted bits with 0's while SRA will replace the shifted bits with the most significant bit of the input, to maintain it's sign. If input was a negative value it will maintain it's negative sign by filling in with 1's whereas a shift logical would turn a negative number positive by filling with 0's.

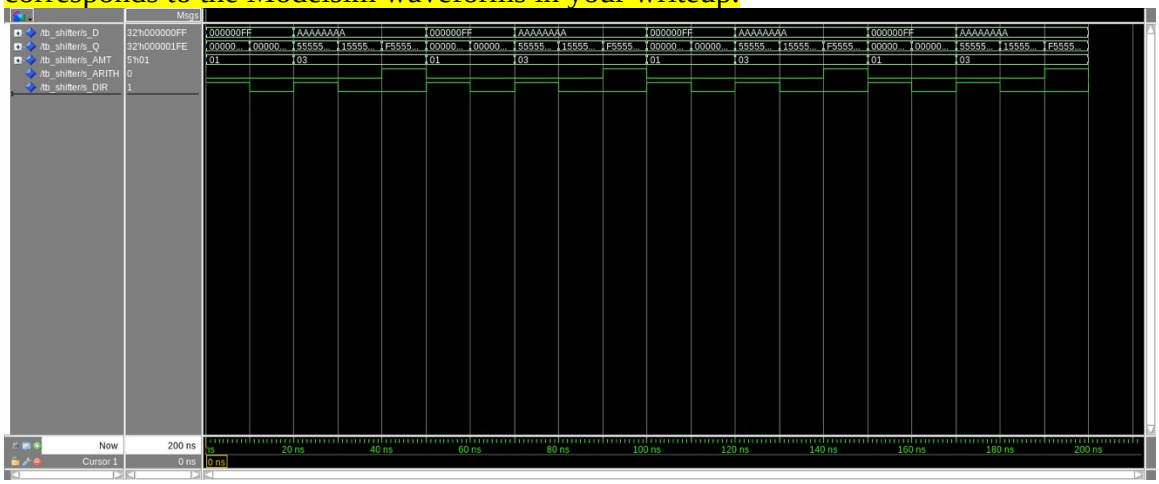
[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

Our shifter uses a select bit call i_ARITH that uses Arithmetic shifting when set to 1 and logical shifting when set to 0.
The logical shifting simply shifts in the desired order and filling the shifted spaces with 0's.
The arithmetic shift will replace the shifted bits with the most significant bit of the value to maintain it's sign.
We implement this with a mux that outputs shift_bit and has D0 as '0' and D1 as the most significant bit of the input. The mux uses i_ARITH as it's select bit.

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

The right shifter can support left operations by adding a select bit to select which way the shift will go (left or right) if the shift goes left, we will reverse the order of the bits on the since the barrel shifter will always shift right we can mimic a left shift by flipping the order of the bits and shifting them right. Then once all the bits are shifted we can flip the bits once more and we will find that our bits have been shifted left by the desired amount.

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



When s_DIR is 1, i_ARITH is 0, the shifter will shift i_AMT times to the left using a SLL. In terms of the design: the order of the bits will be flipped in the input and stored to a signal. This signal will be right shifter i_AMT and then this value will be flipped once more before outputted into o_Q.
With Arith 1 and Dir 0, the shifter will preserve the most significant bit of D which can be seen at 90ns. i_D is a negative value and o_Q is negative because the shift bit is '1', the most significant bit of i_D.

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

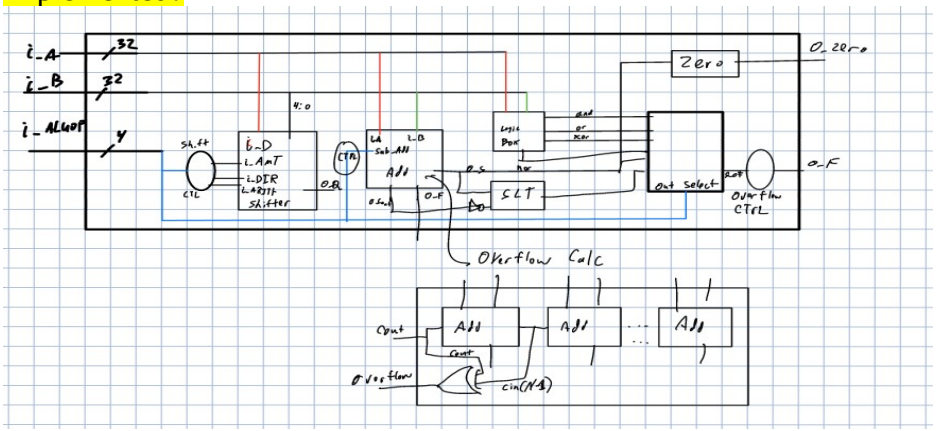
Our ALU design approach was to start with the basic features and slowly build up. We started with simple add and subtracting then implementing overflow detection, and logical functions like and, or, xor. Eventually leading us to an ALU with support for REPL.qb, shifting arithmetics and lui functionality. We had to make the design decision

of putting the input controls on the outside of the ALU so as to minimize the amount of input ports on the ALU, meaning each input A and B are controlled by their respective MUXes to allow for a wide variety of inputs into the ALU.

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

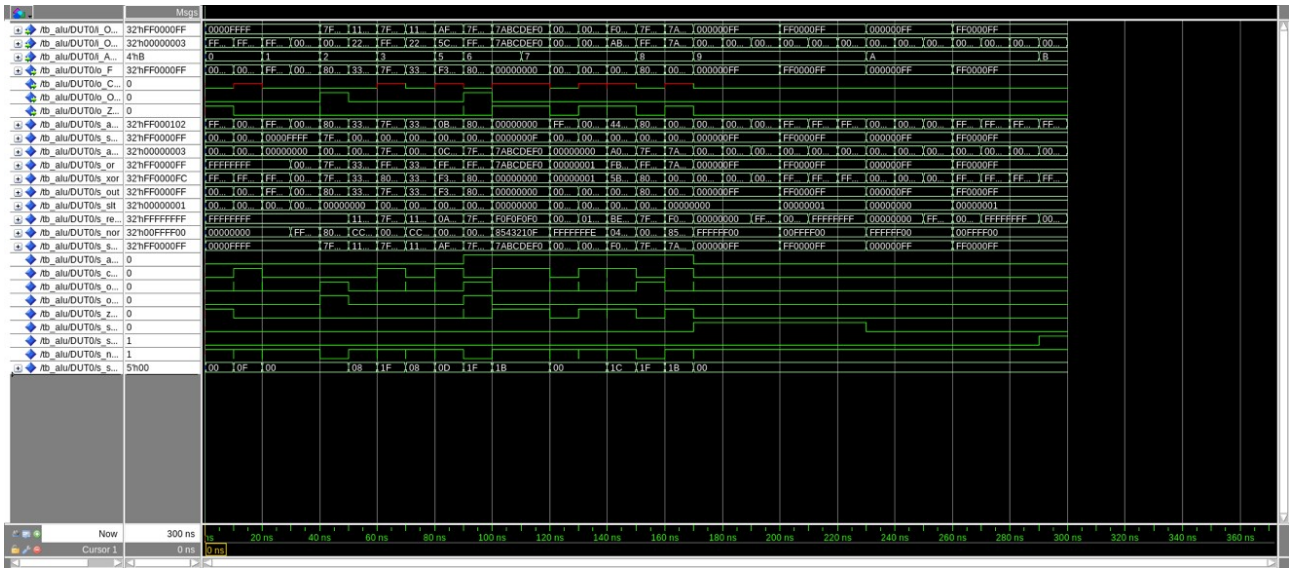
In the replicator a chose byte from a 32 bit value is repeated over the entire data stream. For example when byte 01 is chosen for input ABCD1234 the byte “12” is repeated and the output then becomes “121212121212”. In the ALU this operation had it’s own ALUOP of “1100” and would set the output of the ALU to the replicated field.

[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?



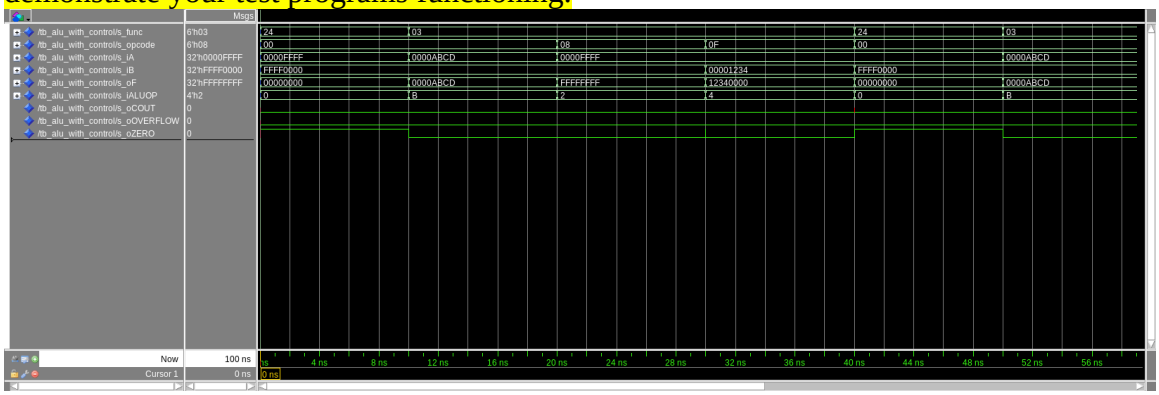
Overflow is calculated by an XOR of the adder Carry out and the last carry in of the adder.
Zero is set to 1 when all bits of the adder output are 0.
SLT is implemented with and and of the most significant bit of the adder output and the negation of carry out. It sets the 0^{th} bit of the output to either 1 or 0 and the remaining bits are filled with 0.

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



Different operations are selected through the ALUOP bits.In this case we only need 4 bits since our ALU doesn't need to support more than 16 different operations (13). Each component is actually used in the ALU and the output is selected using select with logic. However the components aren't used the same way each time. Depending on the ALUOP bits the adder could add or subtracting, we could throw away and ignore overflow flags, a shifter can shift either left or right and right logically or arithmetically. All of these functions are implemented all though hard to see in the wave form.

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

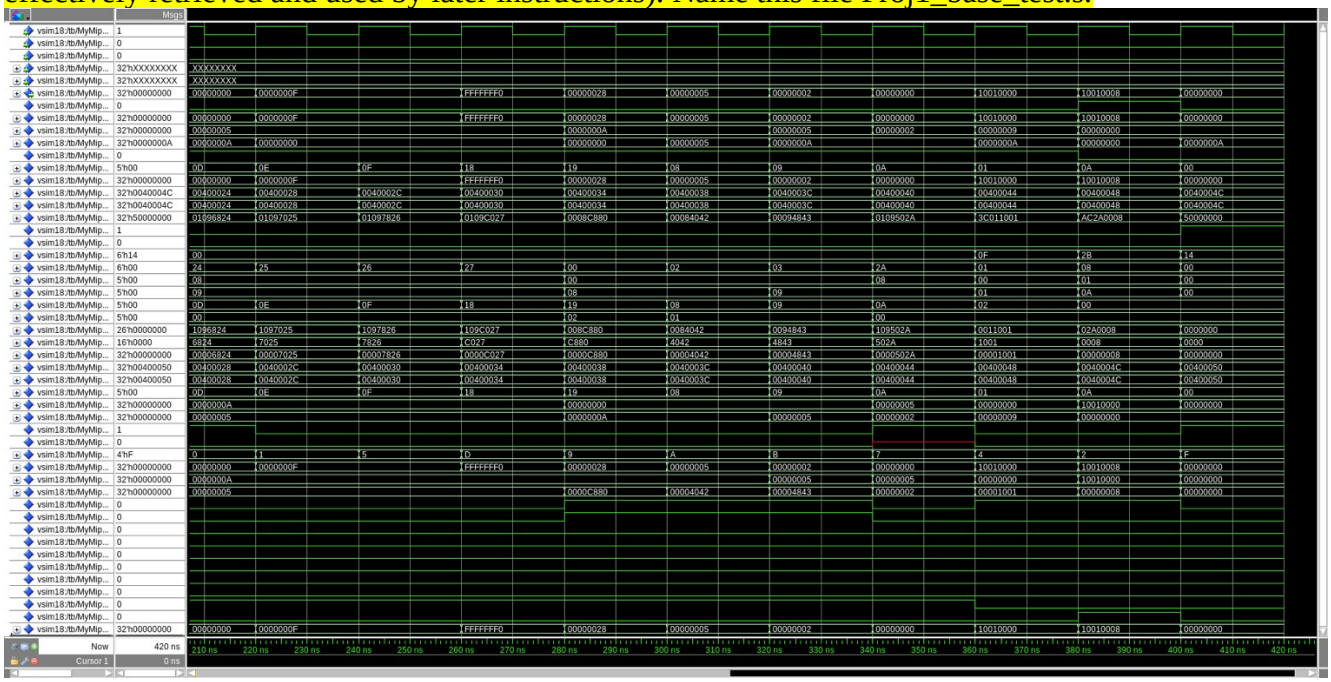


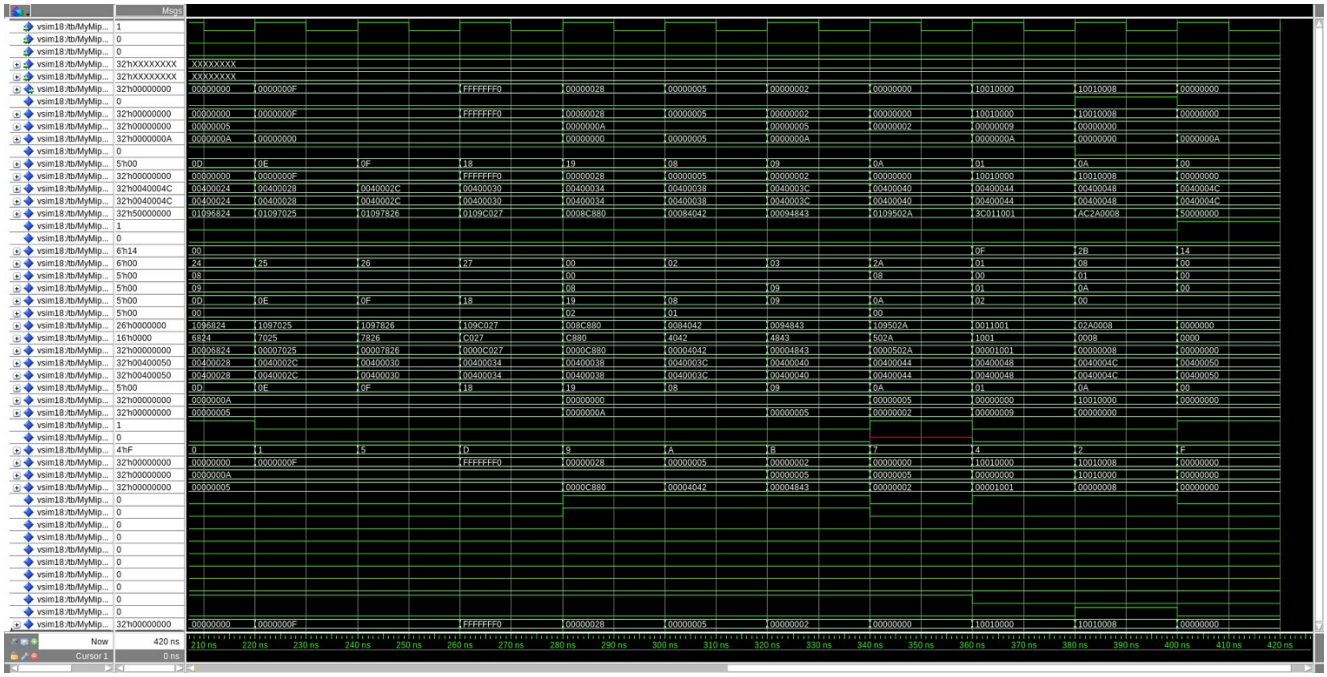
We have a test bench with the ALU and ALU control tied together allowing us to send in certain opcodes and function codes. This allows for a comprehensive test of our ALU and control signals while being able to skip over registers and memory. Allowing us to test

for extreme edge cases easily and not having to waste cycles and have cleaner wavelengths to see our outputs.

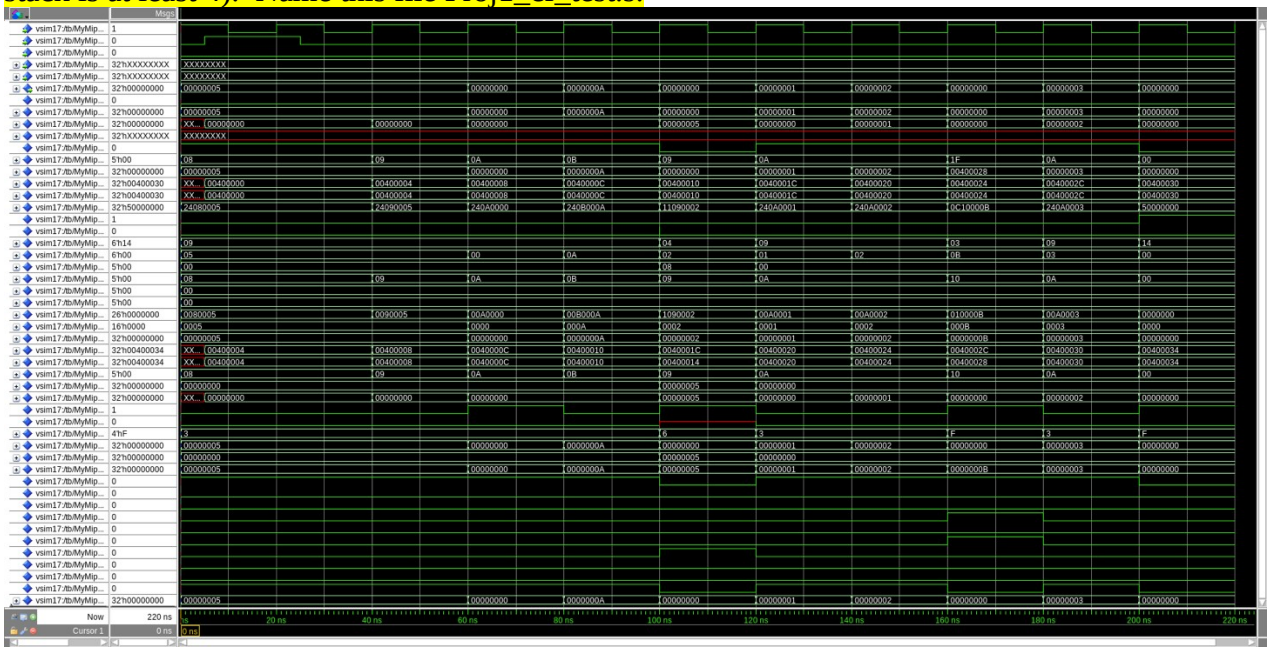
[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.

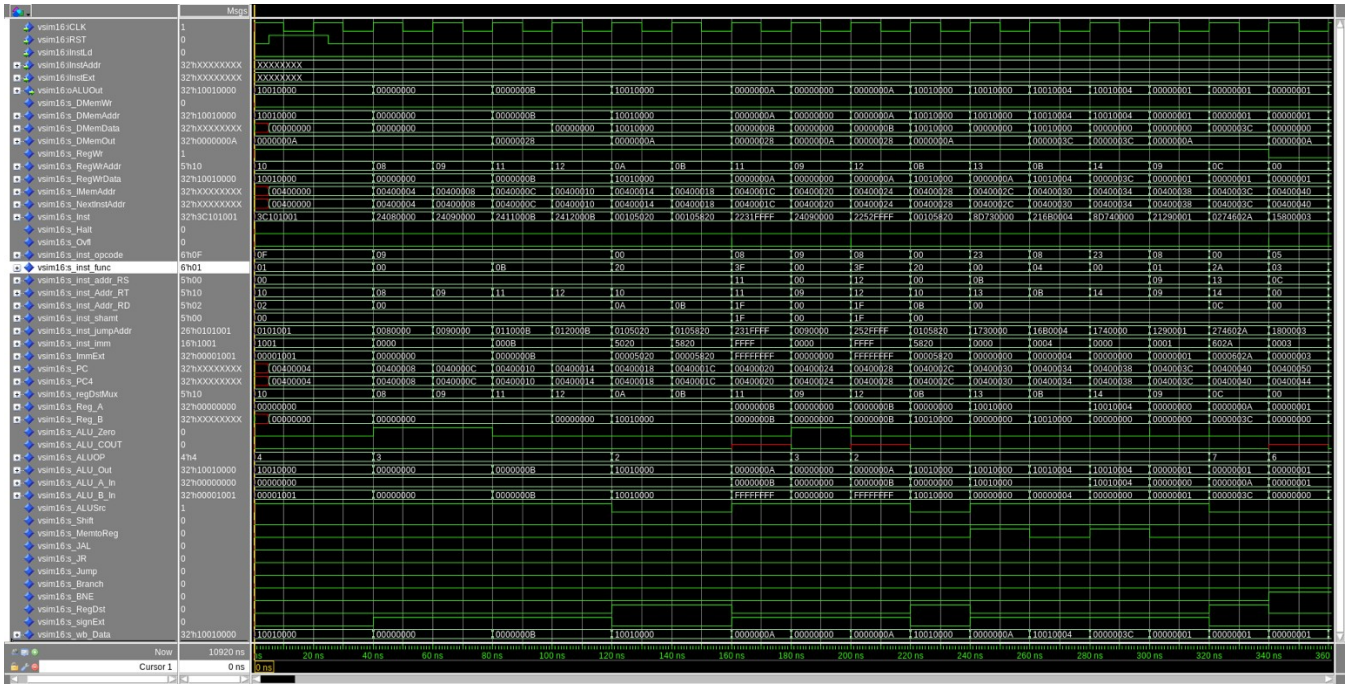


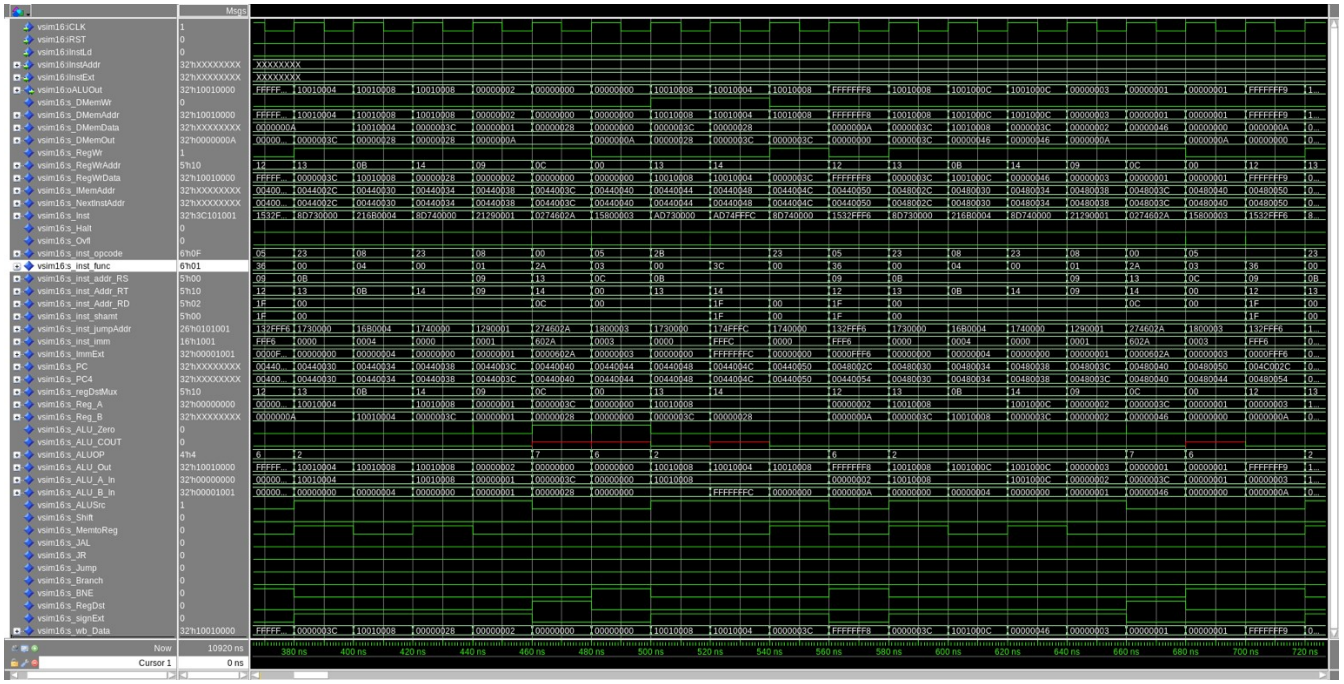


[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Proj1_cf_test.s.



[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.





[Part 4] report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematics. What components would you focus on to improve the frequency?

Our processor can run at a max of 22.41mhz according to our timing.txt after our synthesis.

One component we can focus on improve the frequency would be the ALU. We could remove some functions in our ALU. This could affect the frequency of the processor.