CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 1 Report

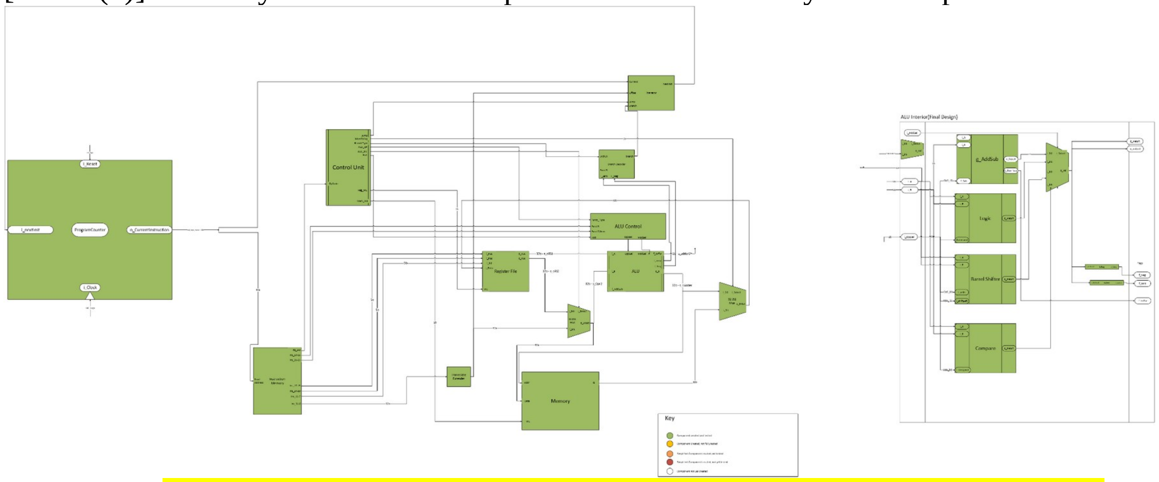
Team Members: James Gaul

Andy Eslick

Project Teams Group #: D

***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.

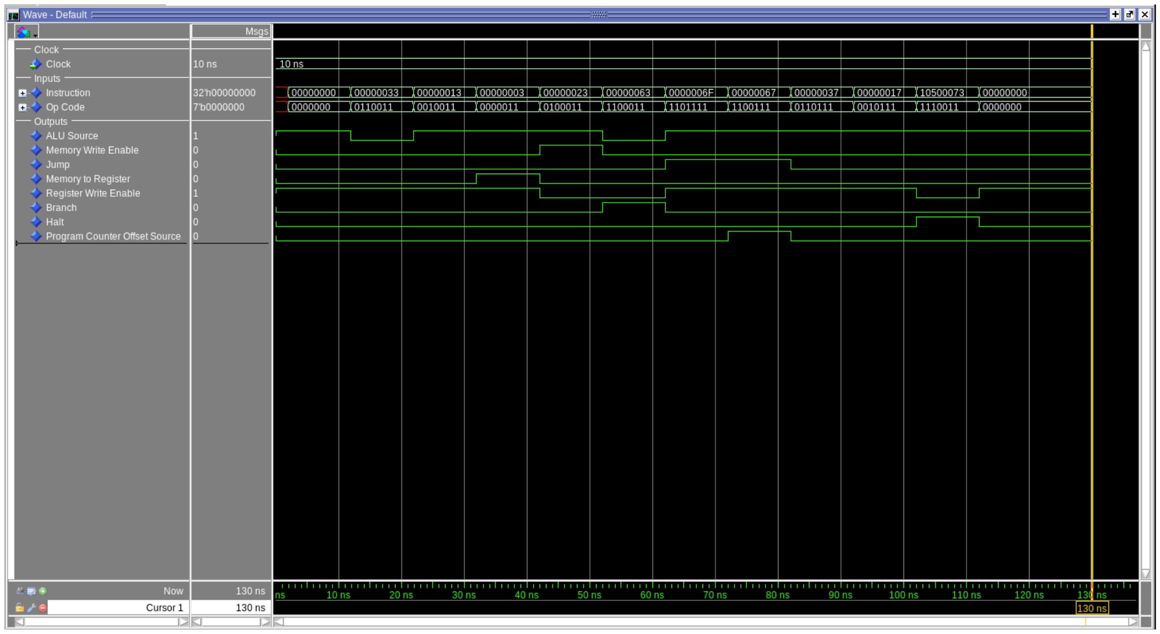


[Part 3.1.a.] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a separate column for each binary bit. Inside this spreadsheet, create a new column for the *N* control signals needed by your 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.

https://docs.google.com/spreadsheets/d/10bik38IxV7eW_53F9f2dHTgDHpZimR-x2GsRa_nxa-s/edit?usp=sharing

Since the branch condition is only dependent on Funct3 and it does not rely on Funct7 or rely on other signals, the branch condition is decoded inside the ALU's internal Compare block, and not in the Control Unit block or the ALU Control block.

[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).



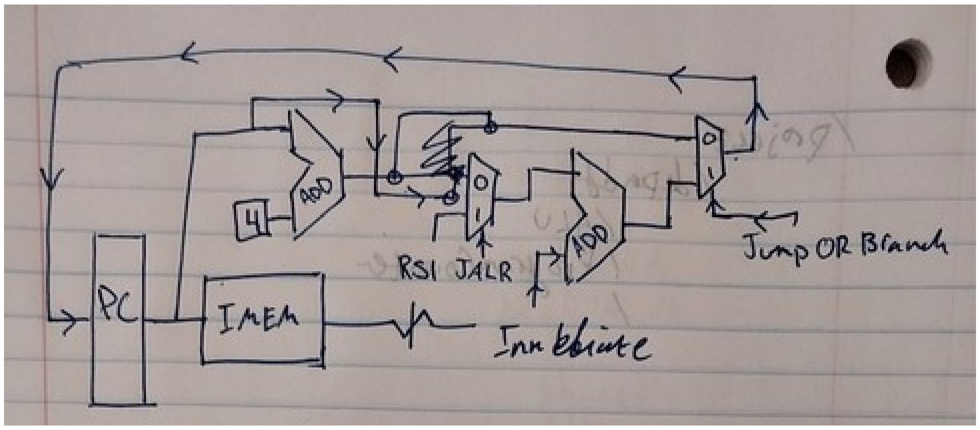
```
Wave - Default
                                   Msgs
Clock
  Clock                            10 ns            10 ns
Inputs
  Instruction                      32'h00000000     00000000  00000033  00000013  00000003  00000023  00000063  0000006F  00000067  00000037  00000017  10500073  00000000
  Op Code                          7'b0000000       0000000   0110011   0010011   0000011   0100011   1100011   1101111   1100111   0110111   0010111   1110011   0000000
Outputs
  ALU Source                       1
  Memory Write Enable              0
  Jump                             0
  Memory to Register               0
  Register Write Enable            1
  Branch                           0
  Halt                             0
  Program Counter Offset Source    0

Now       130 ns    ns  10 ns  20 ns  30 ns  40 ns  50 ns  60 ns  70 ns  80 ns  90 ns  100 ns  110 ns  120 ns  130 ns
Cursor 1  130 ns    130 ns
```

[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.

We will only set the program counter in 3 ways - in the standard case, the immediate case, or the JALR case. In the standard case, we just add 4 to the program counter. In the immediate case, used by branches and JAL, we add an immediate value to the program counter. In the JALR case, we do not offset the program counter, but set it to another value unrelated to the current counter value.

We used two adders and two multiplexers in our implementation. The first adder's inputs were the current program counter value and the number 4. The second adder's input

[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?



We needed flags for Jumping, Branching, and JALR.

[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.

When the control signal indicates a Jalr or Jal function, the ALU is set to override the A input with PC value, and the B input with an offset of four

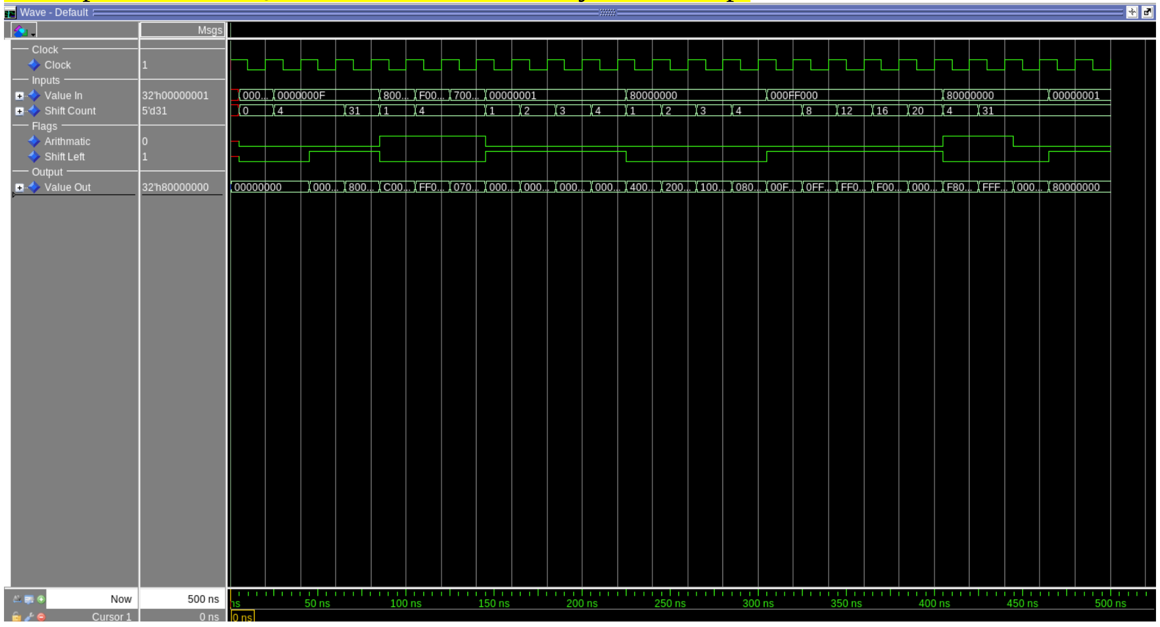
[Part 3.3.1.(a)] Describe the difference between logical (`srl`) and arithmetic (`sra`) shifts. Why does RISC-V not have a `sla` instruction?
The srl instruction stands for Shift-Right-Logical, while the sra instruction stands for Shift-Right-Arithmetic. In RISC-V, a logical shift sets the incoming value with each shift as 0, while an arithmetic shift sets the incoming value to the value of the sign bit. This is used to preserve the sign of shifted values. As any left shift inherently overwrites the sign bit with other bits of the input value, an arithmetic shift would be invalid.

[Part 3.3.1.(b)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.
The shifter implements five rows of 2t1 mux units, switching between either the corresponding (n) bit, or offsetting by 2^i bits, where i is the current row number. Thus, each bit of i_sCnt shifts i_valIn by its binary value. To store intermediate values, the shifter uses a 6x32 bit array.

For bit shifts where the value would overflow the boundary of the function, the module sets the values based on the i_arrshift signal. When i = 0 (logical shift), the bits are always set to zero. When i=1(arithmetic shift), the bits are set to the value of the rightmost bit of i_in.

[Part 3.3.1.(c)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.
By reversing the bits, shifting them right, then reversing them again, we achieve left shifts. As left shifting does not have an arithmetic shift, we automatically tie the arithmetic flag low during this process.

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



Wave - Default
Msgs
Clock
Clock 1
Inputs
Value In 32'h00000001
Shift Count 5'd31
Flags
Arithmatic 0
Shift Left 1
Output
Value Out 32'h80000000
000.. 0000000F 800.. F00.. 700.. 00000001 80000000 000FF000 80000000 00000001
0 4 31 1 4 1 2 3 4 1 2 3 4 8 12 16 20 4 31
00000000 000.. 800.. C00.. FF0.. 070.. 000.. 000.. 000.. 000.. 400.. 200.. 100.. 080.. 00F.. 0FF.. FF0.. F00.. 000.. F80.. FFF.. 000.. 80000000
Now 500 ns
Cursor 1 0 ns
ns 50 ns 100 ns 150 ns 200 ns 250 ns 300 ns 350 ns 400 ns 450 ns 500 ns
0 ns

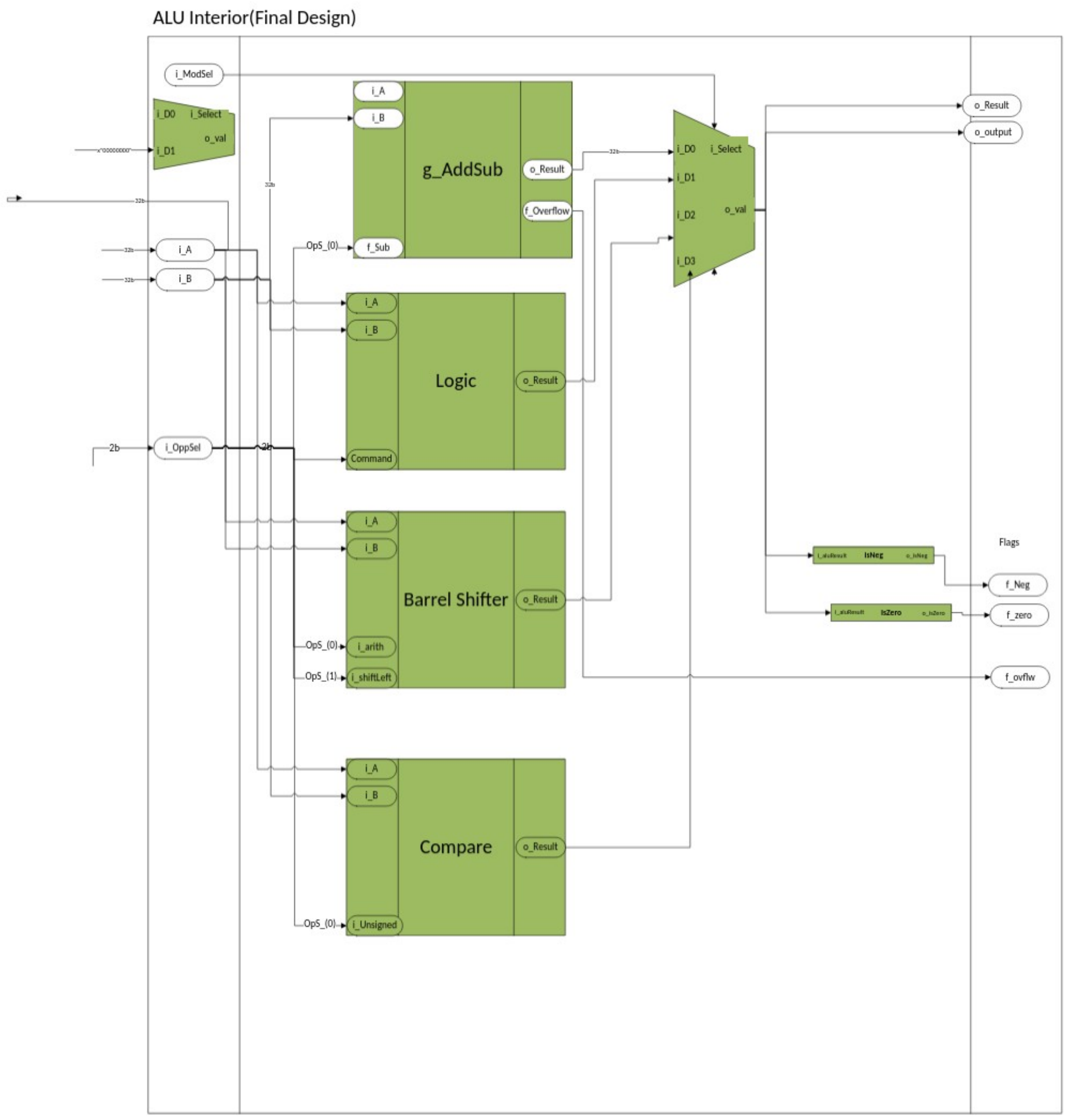
The module shifts the input value up to 31 bits, either left or right based on the “Shift Left” value. When the Arithmetic value is high and the shift left bit is low, the sign value is preserved during the shift. Otherwise, the value is zero.

[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.

When we got to implementing the Control Unit, we struggled deciding how much responsibility it has over the ALU. For example, how should we encode a subtract operation versus a shift operation versus a comparison operation. We decided that instead of having an intermediary control unit (ALU Control in the provided high level diagram) decode in-between instructions from the control unit, we would have a dedicated ALU Control block that took in the OpCode, Funct3, and Funct7 as inputs, and set the ALU operation signals according to them. Eventually, we ended up tying the appropriate bits to the ALU Control unit's inputs in the RISCV_Processor.vhd file.
This decision was risky. The ALU Control block doesn't necessarily decode Funct7, it decodes bits 31 downto 25. If by chance, bits 31 downto 25 create the Funct7 code for SUB or SRA, those operations would go through. To help minimize the times a collision like this would happen, we referenced the full Funct7 code in the ALU Control block, and default the Operation Select and Module Select signals to 'X', so unexpected errors rise quicker.

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

[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?



We implemented slt by casting the std_logic_vector to signed, then using VHDL's less-than operator, we set the bottom bit of the output to 1 or 0 depending on which one was greater. For sltu, we casted to unsigned and did the same thing. Zero is calculated by getting a std_logic_vector that's equal to zero, then using VHDL's equality operator, we set the zero flag.

[Part 3.3.5] Describe how the execution of the different operations corresponds to the QuestaSim waveforms in your writeup.

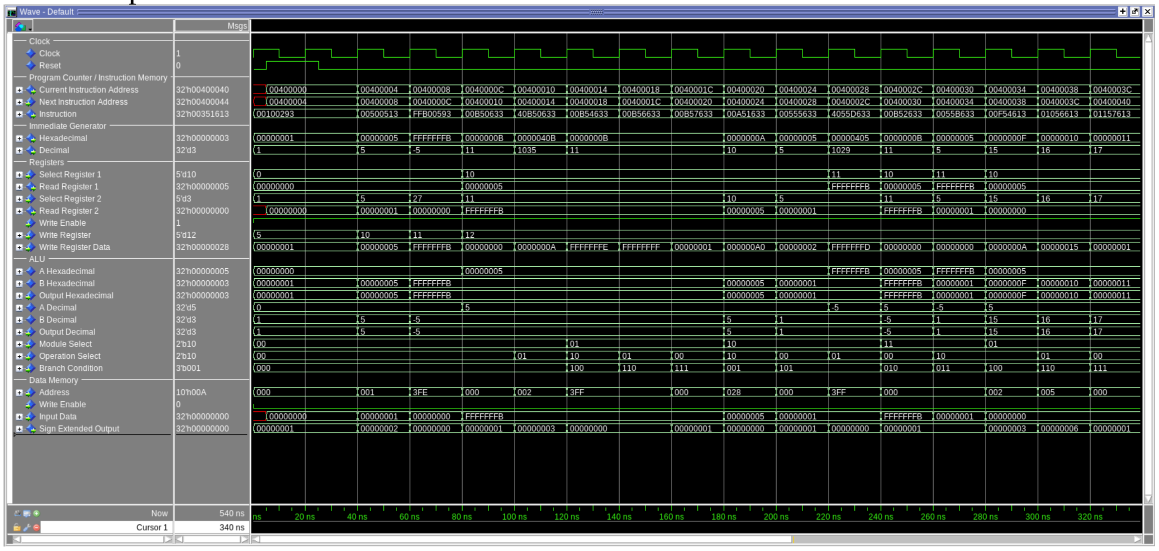
[Part 3.3.8] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

These tests resemble Unit Tests. Unit Tests catch coding and implementation errors early, helping to minimize expensive refactoring later in the design process. Unit tests assume that as long as a block's dependencies have no errors, if errors arise in the development process of this block, the errors are in this block.

[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.

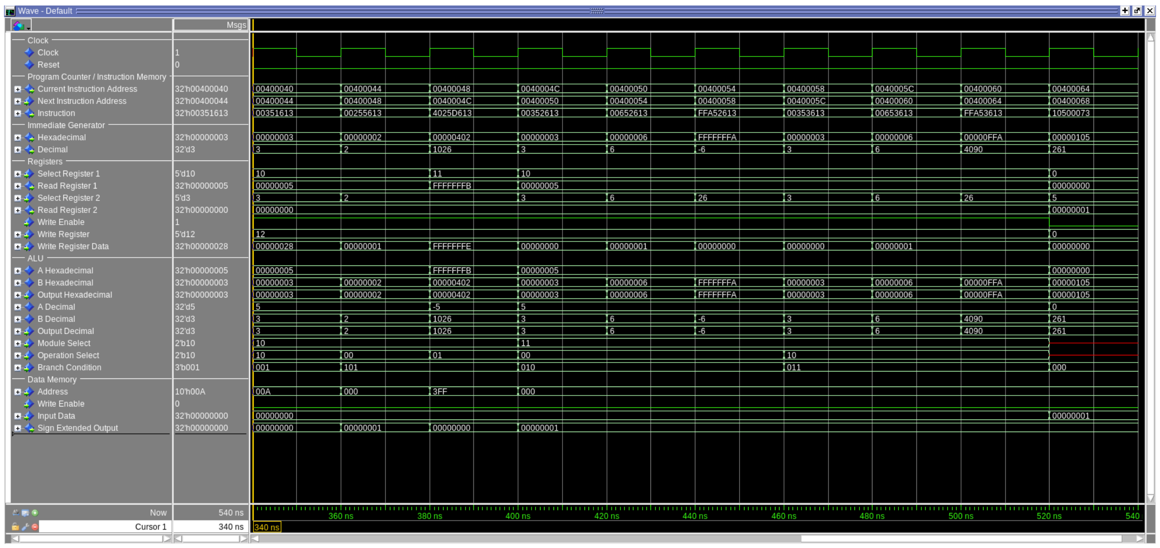
[Part 4.a] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.

Part 1 - Up to ANDI



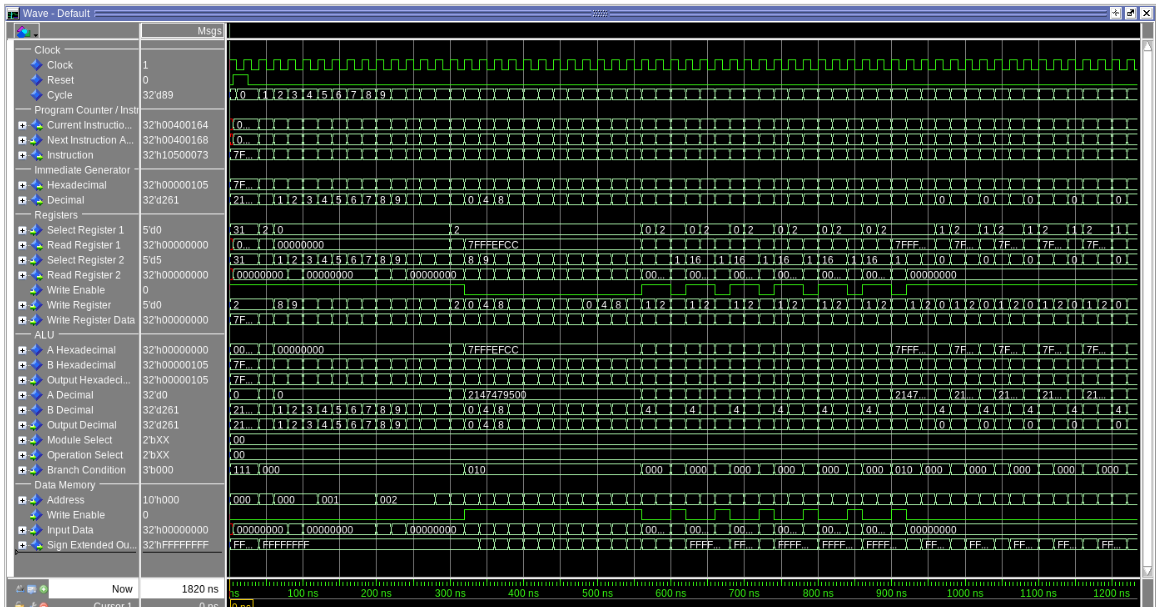
Here, the instructions ADDI, ADD, SUB, XOR, AND, SLL, SRL, SRA, SLT, SLTU, and XORI are tested.

Part 2 - Past ANDI



[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.

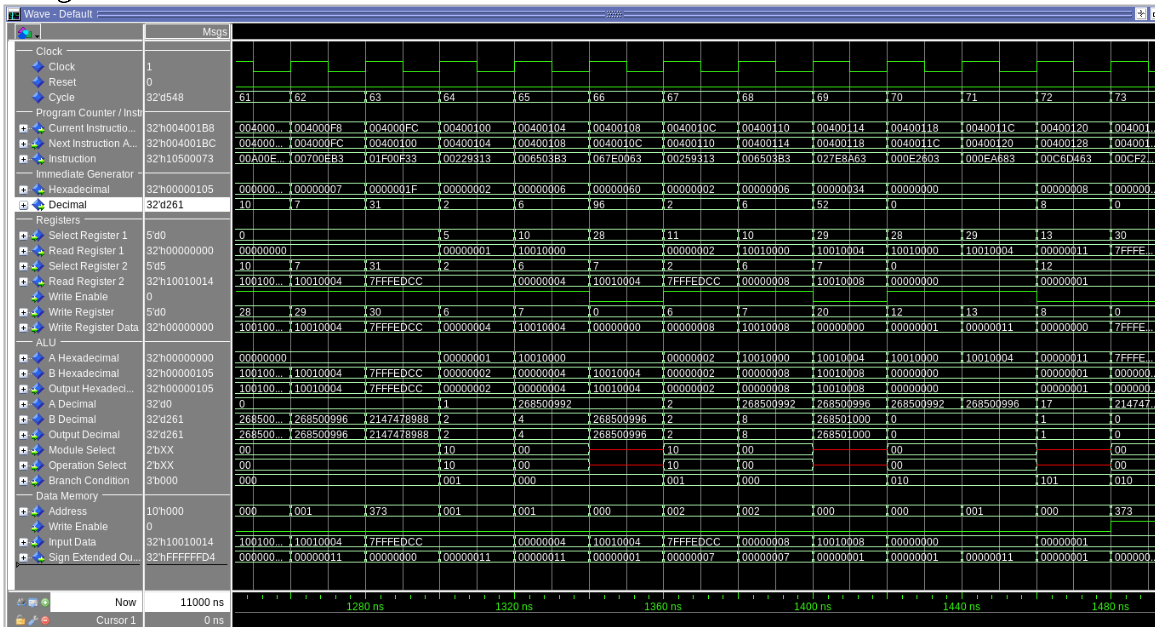
Control Flow Waveform: The system succesfully branches through the different portions of the program based on input.



EXEMPT

[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.

Merge Sort Waveform:



	Msgs
Clock	
Clock	1
Reset	0
Cycle	32'd548
Program Counter / Instr	
Current Instructio...	32'h004001B8
Next Instruction A...	32'h004001BC
Instruction	32'h10500073
Immediate Generator	
Hexadecimal	32'h00000105
Decimal	32'd261
Registers	
Select Register 1	5'd0
Read Register 1	32'h00000000
Select Register 2	5'd5
Read Register 2	32'h10010014
Write Enable	0
Write Register	5'd0
Write Register Data	32'h00000000
ALU	
A Hexadecimal	32'h00000000
B Hexadecimal	32'h00000105
Output Hexadeci...	32'h00000105
A Decimal	32'd0
B Decimal	32'd261
Output Decimal	32'd261
Module Select	2'bXX
Operation Select	2'bXX
Branch Condition	3'b000
Data Memory	
Address	10'h000
Write Enable	0
Input Data	32'h10010014
Sign Extended Ou...	32'hFFFFFFD4
Now	11000 ns
Cursor 1	0 ns

[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?

The maximum frequency is 27.47 mhz, largely due to the data arrival time of 39.685 nanoseconds. Pipelining should help reduce this.