Britt Ahlgrim

Sophia Shanahan

Chris Procak

# **Project 2: Creating Single Processor Components**

# **Introduction**

The objective of project 2 was to design and test the following datapath components: register file, arithmetic logic unit with control, 32-bit adder, memory unit, and single cycle control unit. The Vivado integrated development environment was used to encode these components in VHDL (very high-speed integrated circuit hardware description language) and run simulations on each component individually. This project serves as preparation for project 3, where the hardware for an instruction set architecture (a modified subset of the MIPS 32-bit ISA) will be implemented.

After performing calculations on the timing most critical path, we were able to find the worst negative slack and the total negative slack for each of the components. The component that will have the biggest impact on timing delay is the memory unit. This time delay will be considered as we use these components in project 3, and we will revisit them to see if anything can be done to speed up these components.

After performing thorough testing via the creation of testbench files in the Vivado IDE, we utilized the simulation tools to analyze the output of the components, and verify that all five of them work as intended. The test cases written provide coverage on areas that we were aware of, and confidence that our components work the way they should. More specifics on testing are included in the verification section of the report.

We did not try to complete any of the Bonus Eligibility challenges that were provided. Our components work for the base cases that were provided.

# **Discussion of Design Decisions**

## Register File

The register file was the first component created. This meant learning how VHDL worked and operated for creating a hardware project. As such, the file for the register file component is called “tutorial” because it was adapted from an example given from class. The code was created from researching how VHDL worked properly and what the syntax was to generate a component. The first step was understanding how the language worked. Doing so was accomplished through trial and error and looking for different avenues of success in creating a working project. The next step was to create the test bench to ensure correct functionality. This portion went with understanding how VHDL as a language is organized. The language used for testing is the same for writing the working code. Therefore, this provided more resources with understanding how to create the component properly.

## Arithmetic Logic Unit with Control

The ALU with control was the final piece to be constructed because of its heavy reliance on the other pieces of this project. The ALU is composed of the Arithmetic Unit, the Logical Unit, and the Control. The ALU has ports setup for both the Arithmetic Unit and the Logical Unit, and once the outputs are calculated from those components, the control determines which output to use for the output values on the ALU. The control also determines whether a value will be inverted or not when it goes into the arithmetic unit for determination of subtraction or other inversion use cases. The Shifter component was designed as a single entity, as opposed to a left shift and a right shift, and takes in a bit parameter to distinguish which of the bits in the value provided will be shifted. The maximum number of places a value can be shifted in one operation is 3 because we wanted to keep the number of input bits to a minimum so as to not waste resources. The main design decisions that went into building the ALU were more experimental, and finding practical solutions to the issues we ran into.

## Adder

The 32-bit adder was created using repetition of simple components to ensure stability and consistency across the operations. The 32-bit adder is constructed of 31 standard 1-bit adders, and one 1-bit output adder. This design was implemented so that the adder could output the final carryout and result values on the last, most-significant bit operation, without affecting the process of the other 31 adders. The 32-bit adder takes in a carry-in bit which is set to the previous operation’s carry-out bit, so that the addition can be computed as intended; one thing to note is that the carry-in bit should be 0 when adding, and 1 when subtracting, with all of the second input values inverted. This design simplifies the arithmetic unit, and allows for less repetition within the code. The final adder has flags for CarryOut to determine overflow, and Set to determine if the first input is less than the second. The adder was for the most part straightforward, and the decisions that were made were what would be generally expected for this project.

## Memory Unit

The memory unit was created from a close look at the schematic. Once it was understood that the static random memory access required 256 addresses to be stored, it was simple to create. The first step involved creating an array of 32 bit addresses which could be used to store the data. The next step was to write the behavioral logic needed to operate the component and to ensure correct functionality. The final step was creating the test bench and testing the design. There were some strange outputs that came from it, including the data and registers being read and written on a falling clock edge when it was suppose to have those actions occur on rising clock edges. This was corrected and the component soon began operating as expected.

Single Cycle Control Unit

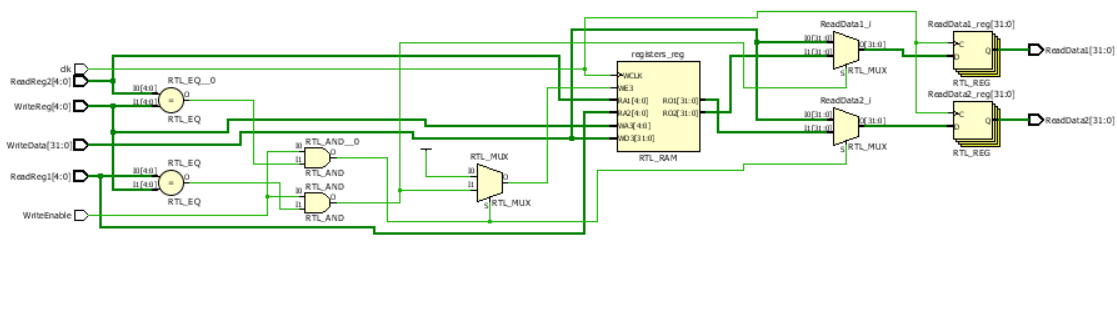
I had a little bit of prior experience with VHDL, and the single cycle control unit reminded me of a multiplexor I implemented in a previous project, except that instead of having multiple inputs and one output, the control unit has one input and multiple outputs. Hence, what I needed to create was more like a demultiplexor. The design process went as follows: I first thought about generating case statements that accounted for every opcode in the instruction set our datapath needed to accommodate. I instead encoded the circuit in figure D.2.5 of our class textbook (*Computer Organization and Design* 5th edition, by Patterson Hennessy), with the thought that this would be a more modular way to construct the program. However, after running the simulation and manually working through the circuit by hand to double-check that the results were incorrect, I realized that this circuit design did not work for I-type instructions. After a brief discussion with our professor, Cris Ababei, I agreed that the easiest way to implement the single cycle control unit was to go back to my original idea of “case/when” statements that recognized the instruction opcode (bits 31-26) and then set the control lines accordingly. There also was a “when others” default case that fits with the Vivado syntax of case statements that sets all control lines to zero when the opcode is not one we were asked to implement. The instruction set accounted for in this design is as follows: add, sub, and, or, sll, srl, slt, jr, addi, ori, lui, lw, sw, beq, jal. The corresponding opcodes for each of those instructions generated a 1-bit value (either 1 or 0) for each of the following control lines: register destination, jump, branch, memory read, memory to register, memory write, ALU source, register write, and a 2-bit value for ALUop.

# **Full Design Specifications**

## Register File

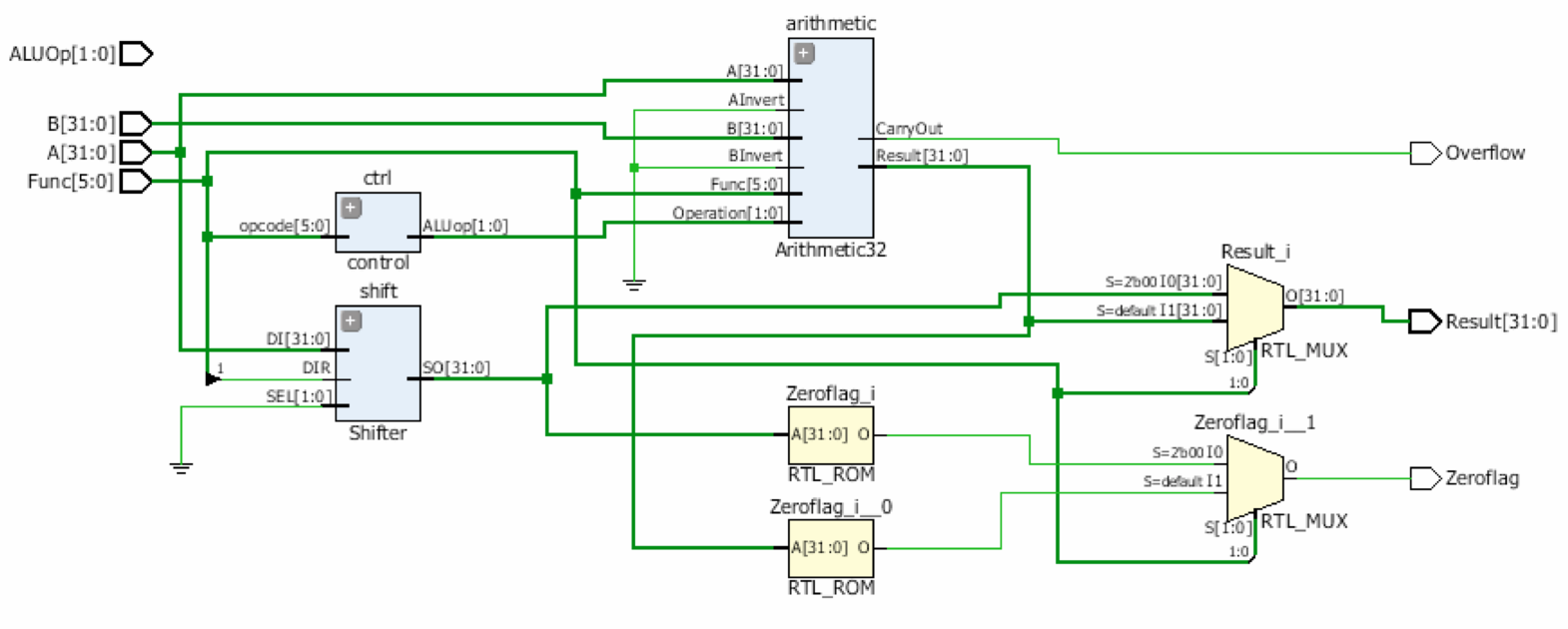
Below is the Vivado schematic for how the Register File is designed. The inputs are ReadReg1, ReadReg2, WriteReg, WriteData, Clk, and WriteEnable. The outputs are ReadData1 and ReadData2.

The design reads data asynchronously while writing data occurs on a rising clock edge with the write enable bit set to 1. If the clock is on a rising edge but the write enable is 0 then the input data does not write to the write register.

An internal signal called “registers” is used. “Registers” is an array of 32 registers of length 32 bits. This is part of the design specification outlined in the assignment. The type defined is called registerFile.

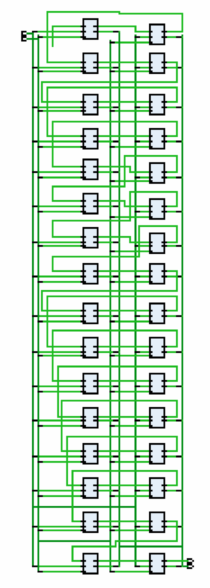
## ALU

The ALU is comprised of the Arithmetic Unit, the Shifter, and the Control Unit. Using the schematic creation tools in Vivado, below is a schematic of the ALU with each of the components and ports as designed in our project. The ALU takes in the two values to operate with, the ALUOp, and the Function code. The ALU then routes the function code to the single control unit to determine the ALU operation control that is used in the arithmetic unit. The A and B inputs are routed to both the arithmetic unit, and the shift unit so that the operations can be performed with both components. Then the results of those components are routed to the ALU Result port, depending on what the opcode is. The zero-flag is set to true if the result evaluates to zero, and the overflow is set to true if the carry out from the arithmetic unit is true. Using the three components, we can get a modular ALU that can perform and produce the desired results.



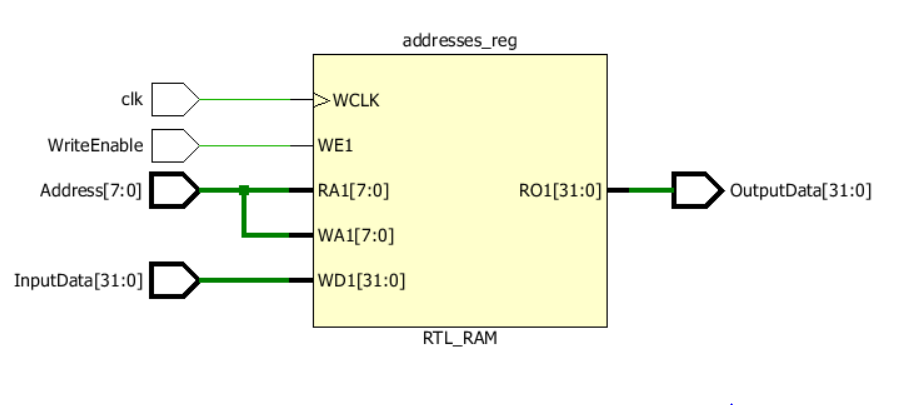
## Adder

The 32-bit adder is comprised of 32 1-bit full-adders. This decision was made to use thirty-two one-bit adders for consistency and simplicity in design. This approach was followed using the examples provided in the project sample files, and proved to be a quick way to implement the necessary addition functionality. The intended purpose of this adder is for incrementing the Program Counter, which will be required in the next portion of the project.



## SRAM

Below is the schematic for the Static Random Access Memory. The inputs are the clk, WriteEnable, Address, and InputData. The output is OutputData. The schematic is simple because we are storing values into an array within the component so that we can provide speed improvements when looking up data and instructions for the processor to use.



The SRAM can be preloaded with data which it can do by writing data to the registers and then reading from those specified registers on reads. The data is read asynchronously while writes occur on a rising clock edge and when WriteEnable is set to 1. This is ensured since the Read will occur first when the component is used while the write is controlled through if statements that check that the clock is on a rising edge and the WriteEnable bit is enabled.

Inside, the data that is inputted gets stored inside an array type called SRAMAddresses. This will contain 256 standard logic vectors of 32 bits. This matches the design specification of the component having dimensions 256x32. That is 256 entries for register datas of 32 bits each.

## Single Cycle Control Unit

Below is the schematic for the single cycle control unit. This component was relatively simple to implement. The input to the component is a 6-bit opcode, encoded in VHDL as the standard logic vector “opcode.” The 9 outputs are encoded as the 2-bit ALUop, and 1-bit ALUsrc, B, J, MR, MW, MtoR, RW, and Rd. Based on the value of the opcode, the control lines were generated accordingly.

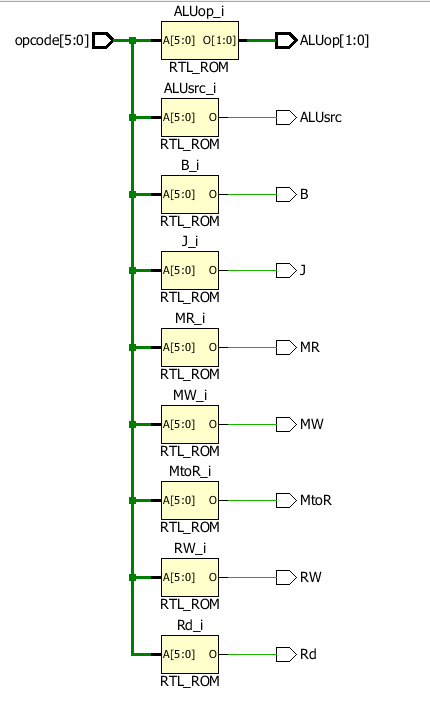
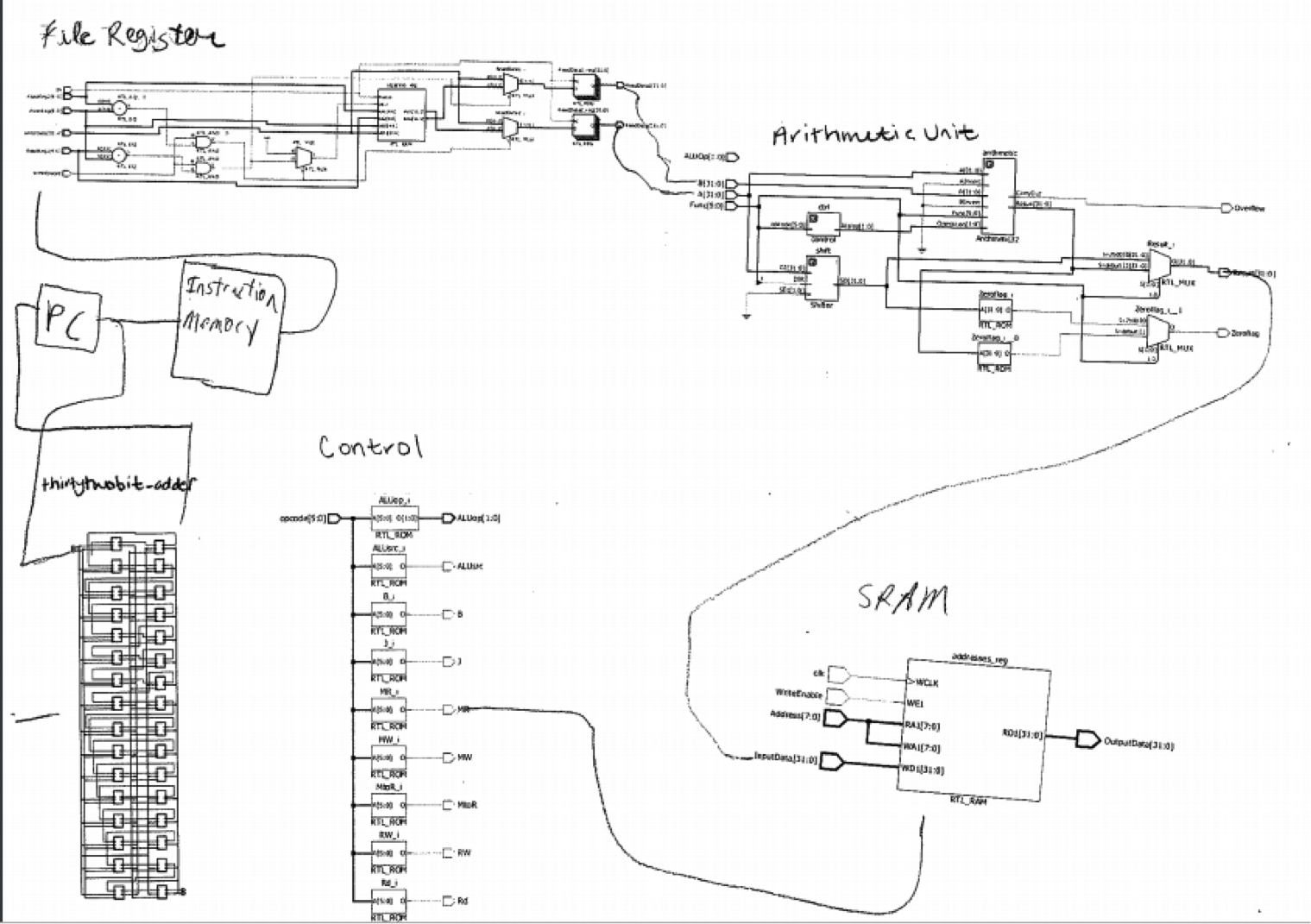


Figure : Datapath of the five components created for project 2. The PC and Instruction Memory have been included in the diagram for readability.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction  Type | Rd | ALUsrc | MtoR | RW | MR | MW | B | J | ALUop1 | ALUop2 |
| R-Type | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| addi | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ori | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| lui | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | X | X |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| jal | 1 | X | 1 | 1 | 0 | 0 | X | 1 | X | X |

Table 1: Control truth table for the instructions that will be implemented in the full datapath

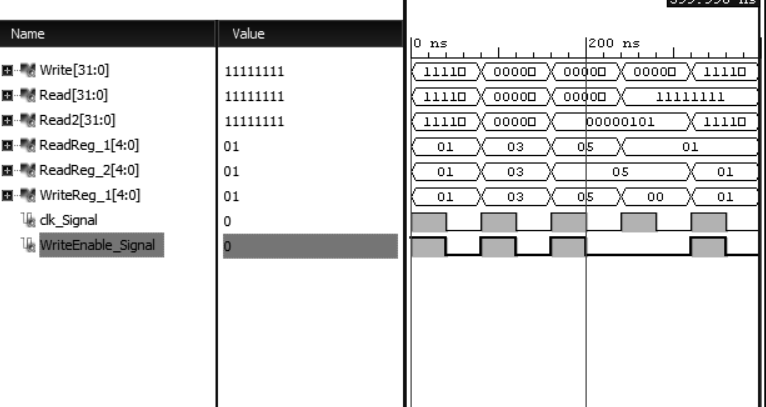
# **Verification**

## Register File

The test cases on the register file included writing a value to a specified register on a rising clock edge and when the write signal was 1. In this waveform, the read registers and the write register have values of 1 initially and the data being written is 1. On a rising clock edge and a write enable, the 1 data will be written to the two read registers. This is shown to happen below when the two read registers contain the value of 1 after execution. The data of 1 is then changed to 3 and then again to 5 to show that the registers can be written to with different values.

The next test case is that values can be written to two different registers. This is shown in the waveform at approximately 250 ns. The data in Read Reg. 1 is 1 while the data in Read Reg 2. Is 5.

The next test case is that a write does not occur when the write is not enabled. This is seen at time 240 ns where the write is still 0. The contents in the two read registers do not change even though the write and read data values change.



## Arithmetic Logic Unit with Control

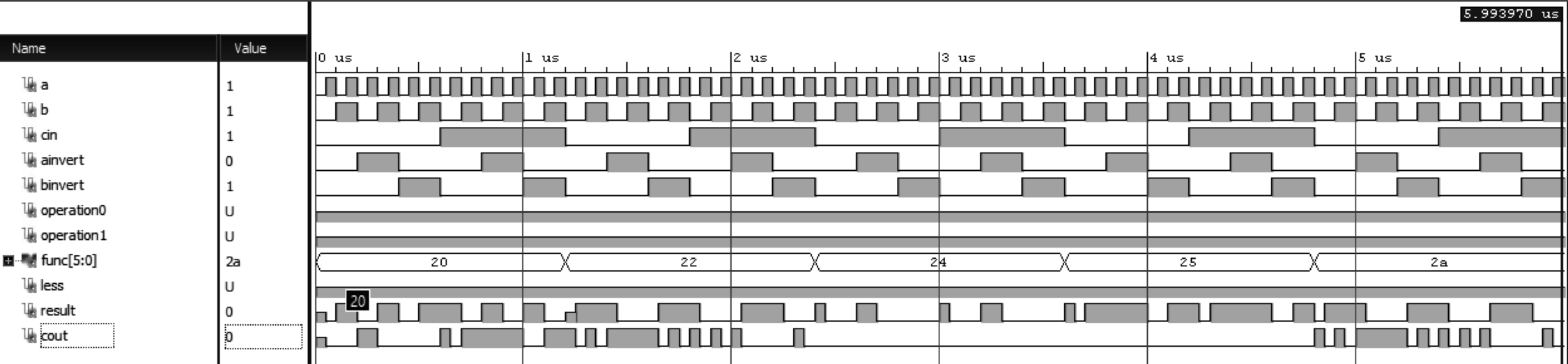
The ALU with control is comprised of an Arithmetic Unit, Shifter, and Single Cycle Control. The testing for the Single Cycle control can be found in the below section describing results of the independent component. The results from the arithmetic unit and shifter are shown and described below.

### Arithmetic Unit

The 32-bit Arithmetic Unit was built to handle ADD, SUB, AND, OR, and SLT functions. To accomplish this, 31 copies of the Bit0To30 component were added, and one copy of the Bit31 component was added, which allows us to perform operations on 32-bit inputs.

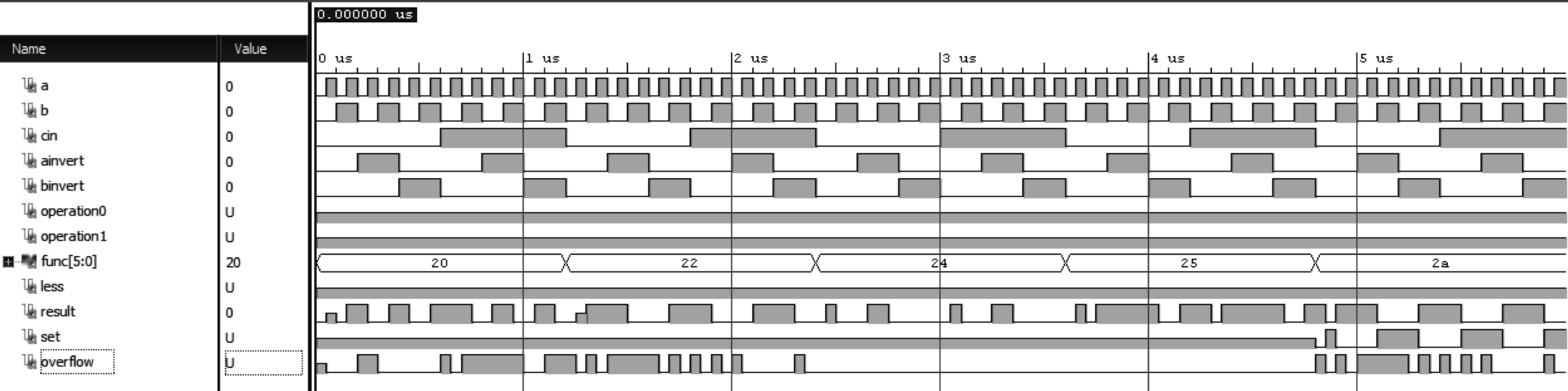
#### Bit0To30

The results from testing the Bit0To30 component are displayed below. The tests covered cases for each of the combinations of inputs. The different inputs that affect the output of the 1-bit arithmetic unit are a, b, cin, ainvert, binvert, operation0, operation1, function, and less. Changing these values, as is demonstrated below, affects both the output result and carryout values.



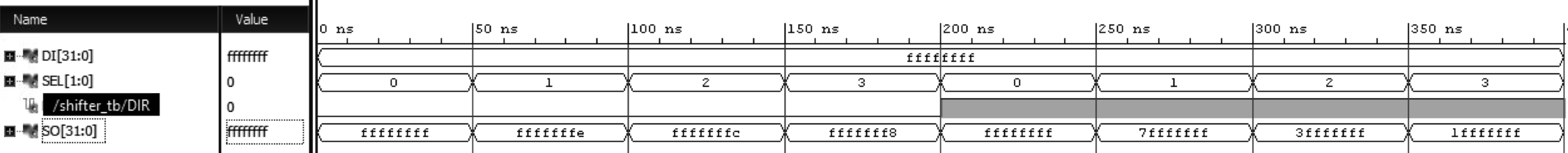
#### Bit31

Similar to the Bit0To30 component, there was also extensive testing performed on the Bit31 component. Below is a demonstration of the test cases that were run against the bit31 component; the notable difference between this component and the previous one is the addition of the set output. The set output parameter is only set on the SLT operations to be 1 if a < b and 0 otherwise.



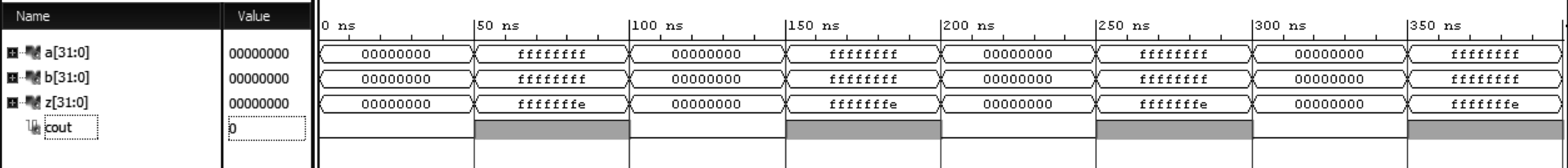
### Shifter

The shifter was tested across all the possible inputs that are expected to shift the values by. The shifter only allows shifting up to 3 bits max; below in the results from the simulation, the left half of the graph displays the 4 possible left shifts, and the right half displays the 4 possible right shifts. The output values on the bottom line are representative of what is expected for each of these shifts, in hexadecimal notation.



## Adder

The 32-bit adder is comprised of 32 1-bit full adders. This was done so that there would be consistency across each of the bits in the adder, and to eliminate the number of components that needed to be created. The testing results, below, display the results when adding the smallest and largest possible value, which are arguably the most important values, and we can conclude that every value in between will behave as expected.

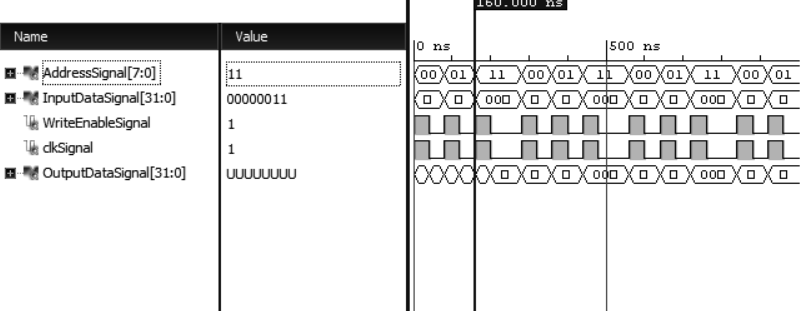


## Memory Unit

The first test case is to write data to the registers. Below we write data to register 0. Initially we write the value of 7 to the register which it does on a rising clock edge and a write enabled. At a time of about 50 ns this is what happens when the input signal at 7 is written to the output signal and it becomes 7.

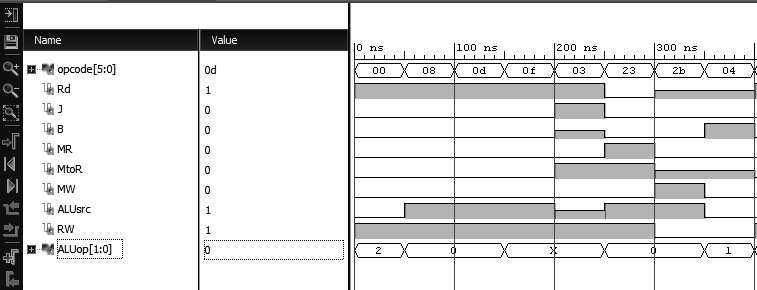
The next test case is that the register can be written to again with different values. This change of values occurs at 300 ns when the input data is 0. The new output signal then becomes 0.

The last test case is to make sure that writes are only occurring when the clock is on a rising edge and the write signal is enabled. From the waveform this can be seen with the output data signal changing on a write enable signal and a clk signal are both 1.



## Single Cycle Control Unit:

In order to test the single cycle control unit, the 6-bit standard logic vector named “opcode” in this component’s VHDL code was provided eight inputs at 50 ns intervals, with each input being an opcode for an instruction that the component must implement. Opcodes for the following instructions were provided as inputs: R-type (add, sub, and, or, sll, srl, slt, and jr), addi, ori, lui, lw, sw, beq, and jal. These eight opcodes sufficiently test all instructions this subset of the MIPS architecture must implement. The inputs, as well as the control line outputs can be seen in the figure below. Testing of this component passed.

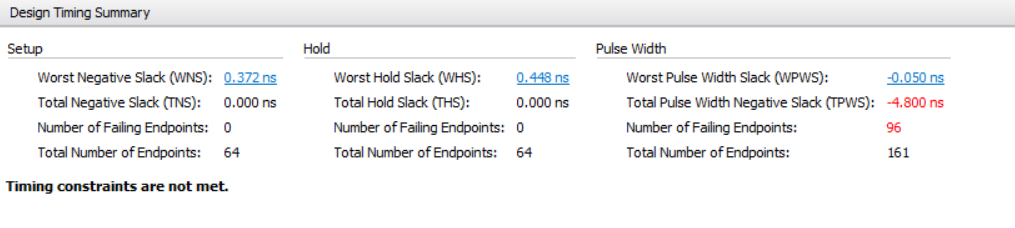


# **Performance Analysis**

The performance of all components was analyzed as follows: a synthesis and implementation were both implemented via Vivado. A “Report Timing Summary” was generated, and the “Worst Negative Slack” time value was noted as the timing most critical path. A clock source was necessary in each of the component VHDL projects for this to work.

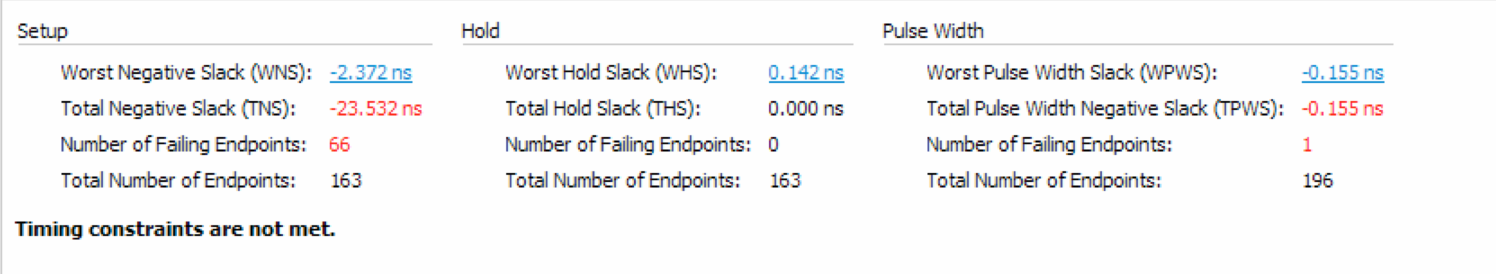
## Register File

The critical path for this component is 0.372 ns and the total negative slack is 0.0 ns. Note that the clock constraint used had a period of 2.0 ns and a waveform from 0-1. This portion influenced how the clock would operate since these are parameters that change with the clock. The timing analysis gives 10 different paths taken. From here the first one is the one with the longest time. Double clicking on it gives additional information. The source appears to be from registers\_reg\_r1 and the destination is to ReadData2\_reg. Additional information is the clock skew is -0.024ns and the clock uncertainty is 0.035ns. The skew is the difference between when a clock signal arrives between two different components. The clock uncertainty is the potential error due to problems run during the implementation.



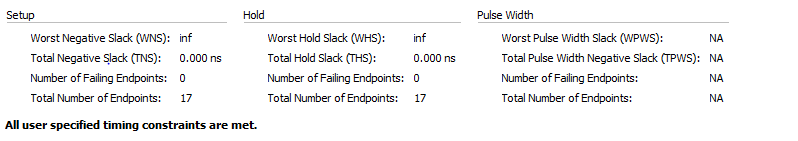
## Arithmetic Logic Unit with Control

The measurement of the critical path of the ALU with Control was done with a clock period of 2.0 ns and a waveform of [0-1]. The critical path under these constraints for the ALU with Control is 2.372 ns and the total negative slack is 23.532 ns. The timing analysis gives 10 different paths taken with the results from the worst as indicated below.



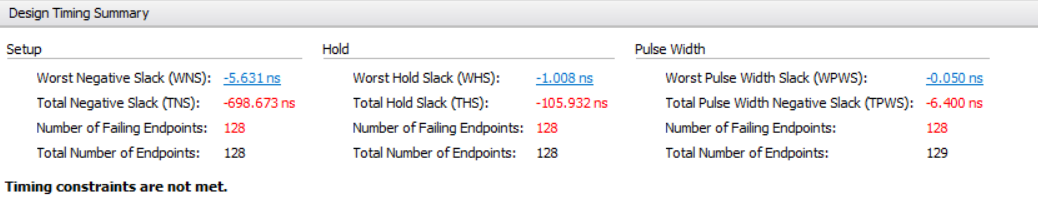
## Adder

The results from the time analysis for the adder are below. One thing to note is that the value for the Negative Slack is infinity. This indicates that depending on the path taken, there is no limit to how much negative slack the component will take. This is not ideal, as it means that we don’t have a good reading on how long the component will actually take to complete its processes. This will be taken into account as we go into project 3, and we will revisit the adder to determine if there’s anything we can do to adjust these numbers to be more in-line with what we desire.

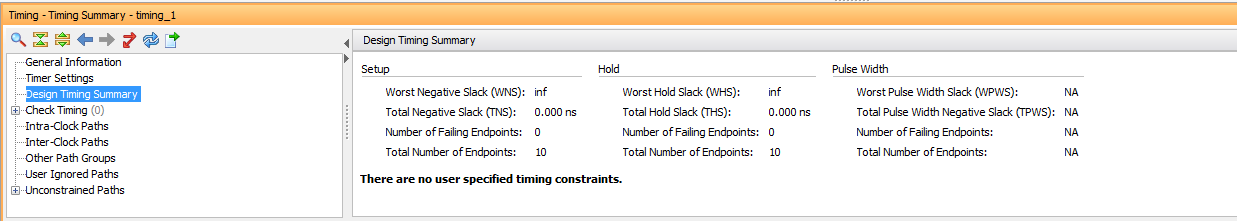


## Memory Unit

The critical path for the Memory Unit is -5.631 ns. This value should be positive but it is something that we will need to figure out in the next project. The possible reason for this is that we are missing necessary constraints in the constraint file. After researching online about vivado and xilinix, this is the direction for the answer. We did improve it from -10 ns down to -5 ns.



## Single Cycle Control Unit

The process for calculating the timing delay for the single cycle control unit was followed the same as the other components, but the worst negative slack value was equating to infinity. A clock was added to the source file for the control unit, and the test bench, but were later taken out as they did not cause a numerical worst negative slack value to be generated. Were a value in that location in the chart, then it would be considered the timing most critical path, and the value we are looking at when considering time delay.

# **Conclusion**

This project allowed us to explore and learn more about VHDL and working in the Vivado environment. The components have each been tested individually, and successfully, but the full connections between the components have not been made yet. In our design decisions, we tried to be as resourceful as possible, while keeping our components modular so that when we put them together, the operations return what is expected.

Testing proves that our code works. Had we had more time and planned better, perhaps we would have been able to incorporate some of the bonus features and suggestions which would have helped to speed up our components. This will be something considered in project 3 as we start to combine all of the components to create a fully working processor.

The results from this project show that the components function properly. We have had enough exposure to VHDL and the Vivado ISE so that in project 3 there will be less initial learning overhead and a smaller learning curve with these technologies.

**Works Cited**

Patterson, David and Hennessy, John. *Computer Organization and Design.* 5th edition. 2014.