Design Project 2

Part A

Nina Lutz

Kat Meza

Marina Monacelli

Zach Thompson

COEN 4720

18 April 2021

I. 32-bit Adder Component

A. Summary

The 32-bit adder component was successfully tested using Vivado simulation tools. The simulations produced the expected outputs for various test cases that were implemented within the testbench file. The resulting simulation waveforms are further discussed in Part I, Section C: Testing, Verification, Results. The most time critical path was determined to be from input b[1] to output z[29] with a total delay of 22.72ns.

B. Description

To begin the design process for the 32-bit adder component, it was first necessary to understand the function of its base components. The base components of the 32-bit adder consist of the full adder component and the four bit adder component. The files for the four bit adder were provided by the instructor as a resource and as such were used as a baseline for the design of the 32-bit adder. In the four bit adder files, four full adders are used to construct one four bit adder. The modifications needed to transform the four bit adder into a 32-bit adder were relatively simple. First, all logic vectors had to be adjusted to support 32 bits rather than just 4. Similarly, the addition process needed to be modified to support the larger number of bits. In the four bit adder file, the addition operation is able to be carried out using a few concise lines of code. However, when dealing with 32 bits it is ideal to use some type of looping structure to achieve a similar outcome. These modifications were made in order to produce a functional 32-bit adder.

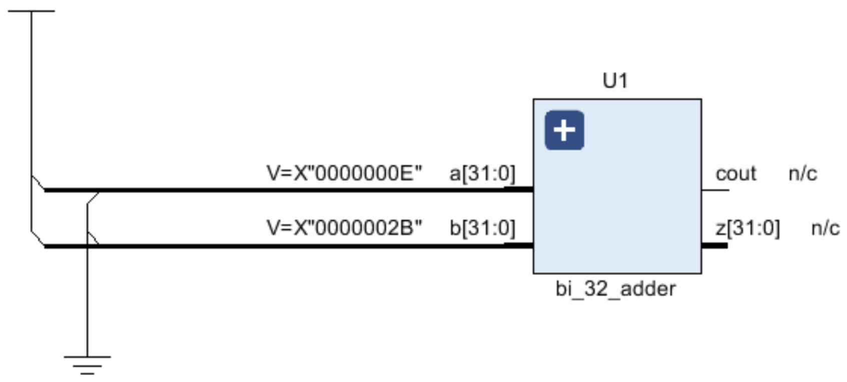
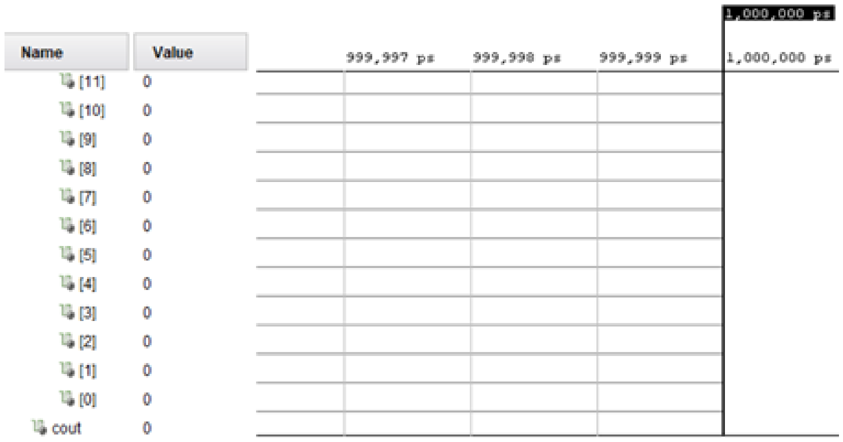


Fig1: Vivado collapsed schematic generated for the 32-bit adder component.

Figure 1 shows the schematic generated using the Vivado software tools. The version shown in Fig.1. is the collapsed view as the expanded view includes all of the full adder components and is too large to concisely display with any significant detail. Fig.1. shows the two 32-bit inputs, a and b, as well as the carry-out bit and the 32-bit output, z. Simply put, the 32-bit adder will intake two 32-bit inputs and complete an addition operation to produce a single 32-bit output, also producing a carry-out bit of 0 or 1. The big picture role of the 32-bit adder is to provide incrementation for the program counter.

C. Testing, Verification, Results

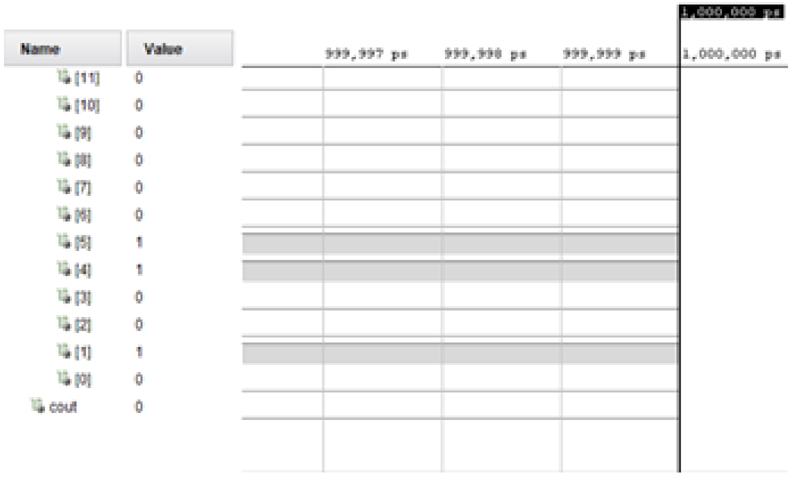
______The testing methods for the 32-bit adder were implemented in the Vivado testbench file. The tests consisted of assigning 32-bit input values to the inputs a and b and observing the z output and carry-out bit via simulation. For each test case, the programmer is able to select the z output and view the output waveform. Figure 2 shows the results for test case 1. Test case 1 analyzes a simple case where all 32 bits of both a and b are 0 and so the output z and the carry-out bit are both expected to be 0 as well.



Name	Value
[11]	0
[10]	0
[9]	0
[8]	0
[7]	0
[6]	0
[5]	0
[4]	0
[3]	0
[2]	0
[1]	0
[0]	0
cout	0

Fig2: Results of Test case 1 where inputs a and b are both 32 bit zero inputs. Shows the final 12 bits of the 32-bit output z.

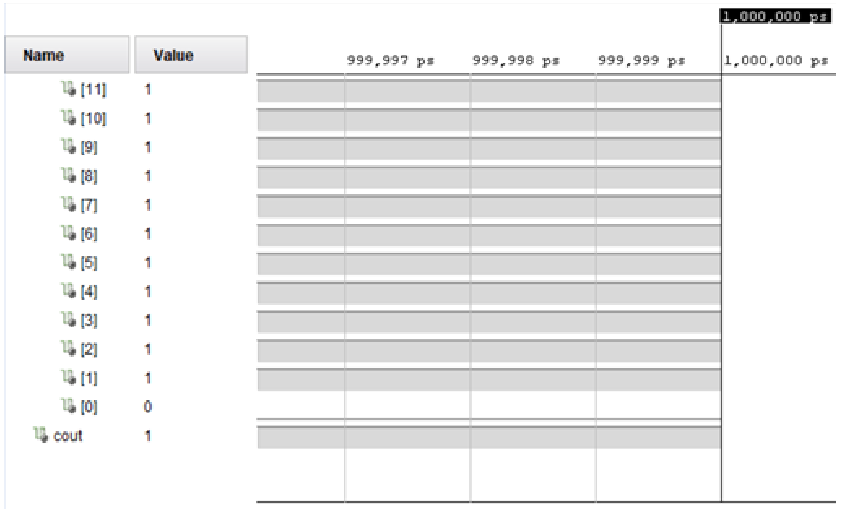
The results displayed in Fig.2. are in alignment with what is expected from the inputs of test case 1. All bits of the output z display a low signal, 0 and the carry-out is 0 as expected. Test case 2 displayed in Fig.3. shows the z output of the addition of decimal 20 and 30 in 32-bit binary.



Name	Value
[11]	0
[10]	0
[9]	0
[8]	0
[7]	0
[6]	0
[5]	1
[4]	1
[3]	0
[2]	0
[1]	1
[0]	0
cout	0

Fig3: Results of test case 2 where the z output is shown and inputs a and b are binary 00010100 and 00011110 , respectively (represented in 32-bit binary in the program).

The result is a decimal 50 which is 00110010 in binary representation. This output can be read from Fig.3. when looking at z[7]-z[0]. The rest of the z output z[31]-z[8] are zeros, making it a 32-bit representation. Finally, test case 3 shown in Fig.4. displays the resulting output when inputs a and b have all 32 digits set as 1.



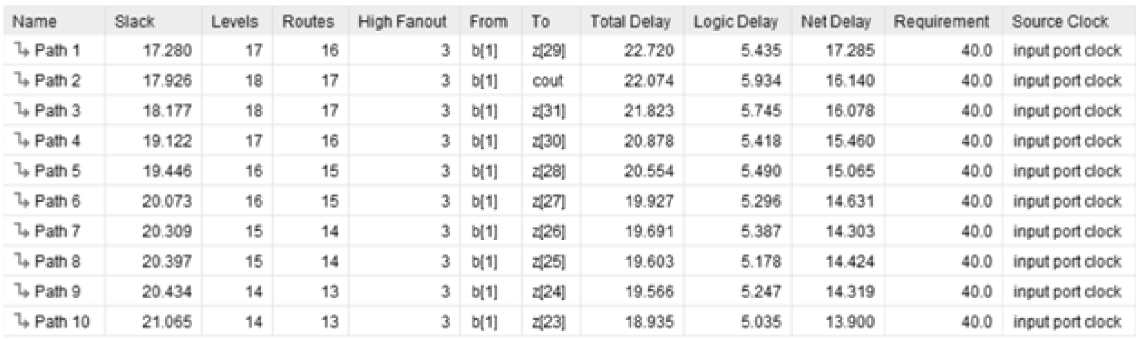
Name	Value
[11]	1
[10]	1
[9]	1
[8]	1
[7]	1
[6]	1
[5]	1
[4]	1
[3]	1
[2]	1
[1]	1
[0]	0
cout	1

Fig4: Results of test case 3 where z output is shown and inputs a and b have all 32 bits set high as 1.

The expected output from inputs a and b is an output z that is also entirely 1's except for the MSB and LSB; 0111 1111 1111 1111 1111 1111 1111 1110. The carry-out bit in this case is expected to be 1. As shown in Fig.4. the actual output matches what is expected. Figures 2, 3, and 4 confirm full operation of the 32-bit adder component.

D. Performance Analysis

Timing analysis tools in the Vivado simulator were used to determine the most time critical path of the 32-bit adder component. The most time critical path is the path that takes the longest to execute in the entire component. Figure 5 shows that the most time critical path is from input b[1] to output z[29] with a total delay of 22.72ns.



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	17.280	17	16	3	b[1]	z[29]	22.720	5.435	17.285	40.0	input port clock
Path 2	17.926	18	17	3	b[1]	cout	22.074	5.934	16.140	40.0	input port clock
Path 3	18.177	18	17	3	b[1]	z[31]	21.823	5.745	16.078	40.0	input port clock
Path 4	19.122	17	16	3	b[1]	z[30]	20.878	5.418	15.460	40.0	input port clock
Path 5	19.446	16	15	3	b[1]	z[28]	20.554	5.490	15.065	40.0	input port clock
Path 6	20.073	16	15	3	b[1]	z[27]	19.927	5.296	14.631	40.0	input port clock
Path 7	20.309	15	14	3	b[1]	z[26]	19.691	5.387	14.303	40.0	input port clock
Path 8	20.397	15	14	3	b[1]	z[25]	19.603	5.178	14.424	40.0	input port clock
Path 9	20.434	14	13	3	b[1]	z[24]	19.566	5.247	14.319	40.0	input port clock
Path 10	21.065	14	13	3	b[1]	z[23]	18.935	5.035	13.900	40.0	input port clock

Fig5: Report Timing Summary of the 32-bit adder component.

E. Conclusions

Through creation of the 32-bit adder component, the group learned how to create a schematic for a component in Vivado. The group also learned how to use the timing analysis tools which were found to be useful and relevant to material covered in lectures about the pipelining process. The biggest obstacle faced in creating the 32-bit adder component was addressing errors in syntax when actually creating the component and test cases and also addressing errors in setting up the timing analysis and constraints.

II. Register File Component

A. Summary

The Register File component was successfully simulated as shown in Section II, C: Testing, Verification, Results. Simulations produced expected outputs for this component. The most time critical path could not be determined for this component because an error occurred when running the implementation. This error will be addressed and the most timing critical path will be acquired before the next project part is due.

B. Description

The design process for the register file was aided heavily by the provided example of the data memory component. The function of the component as a whole is to act as memory storage for the CPU. It lets the CPU know where program data is located and what data is needed to perform certain functions. The write_register provides the index telling the storage array the location for saving the information received from the write_data input. The read_registers act to direct the component where to draw information from, which is then output by the read_data lines. The register file also has a clock input because it only writes on a rising clock edge, and the write_enable also needs to be set in order to add the write_data information. The reset acts to clear the register file of all information.

The register file component begins by defining all in the input and output ports. Once the portmap is established a 32x32 bit array is created to be able to receive information from the write_data port. The location of where each 32 bits of information in the array is provided by the write_register port, which has to be converted into an integer value from decimal in order to be easily used as an index. The system then checks the reset input and clears the data array if the reset has been set. Next, the clock and write_enable value need to be checked because the register file will only write to the data array on the clock's rising edge, and when write_enable is set. The location of outgoing information is provided by the read_registers and sent through the read_data ports.

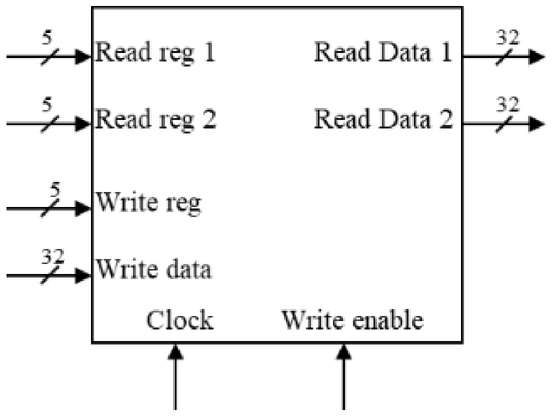


Fig6: Sample schematic design of the Register File component with inputs and outputs

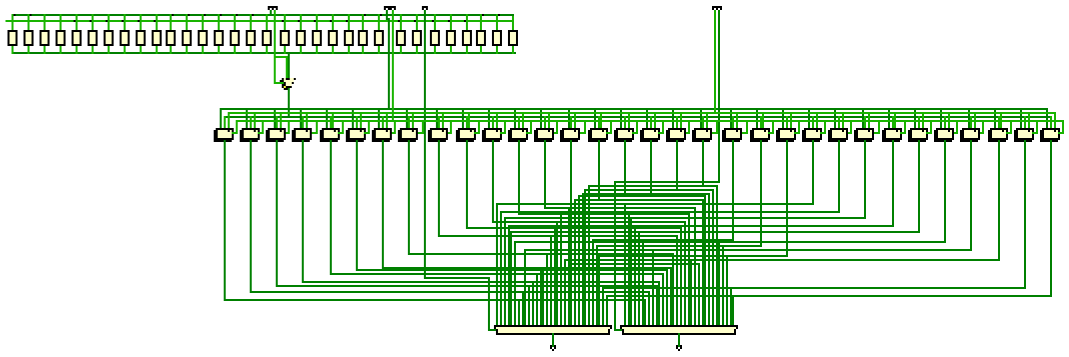
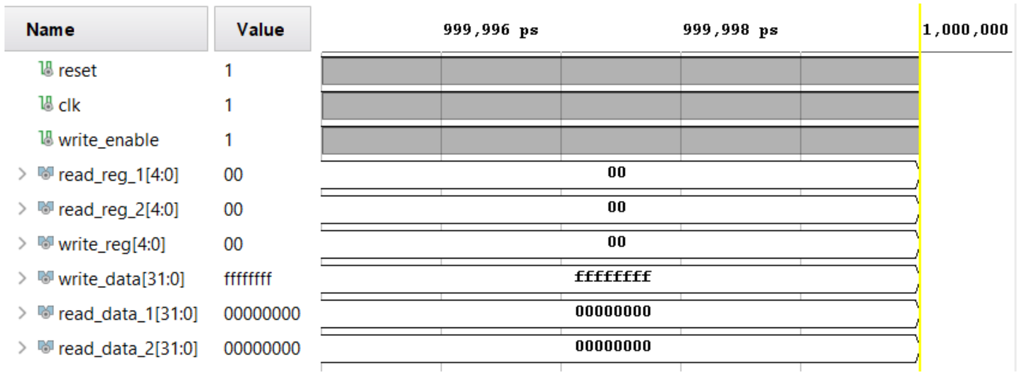


Fig7: Vivado generated schematic of the Register File component

For undetermined reasons, the group was unable to create a condensed schematic of the register file. It is believed that issues arising during synthesis and implementation were also caused by the same errors that prevented a proper schematic to be produced.

C. Testing, Verification, Results

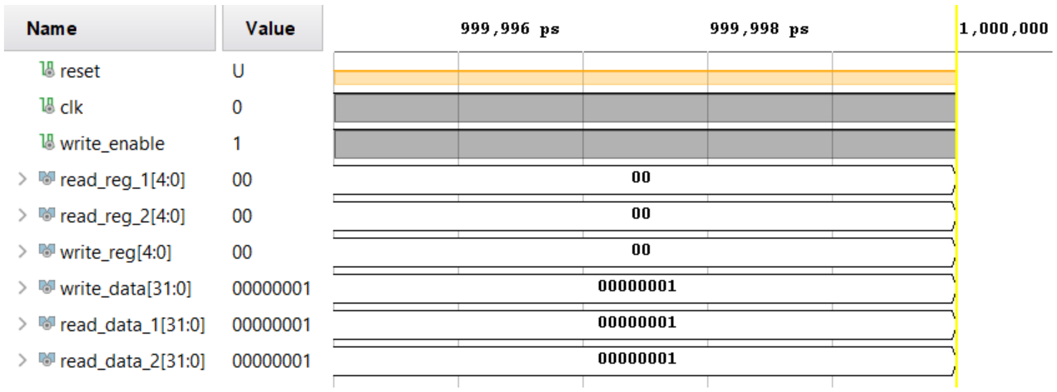
Testing was carried out in Vivado via the use of a separate testbench file. For each individual test case, users are able to adjust the input values of the reset, the write_enable, both read_registers, the write_register, and the write_data value. The expected output can also be declared and compared to the actual result. If the two do not match, an error message is displayed. The group tested the functionality of the reset, the effect of changing the write_enable values, and using different read_registers and write_data values.



Name	Value
reset	1
clk	1
write_enable	1
read_reg_1[4:0]	00
read_reg_2[4:0]	00
write_reg[4:0]	00
write_data[31:0]	ffffffff
read_data_1[31:0]	00000000
read_data_2[31:0]	00000000

Fig8: Results of test case 1 with defined write_data value, but the reset input set ('1')

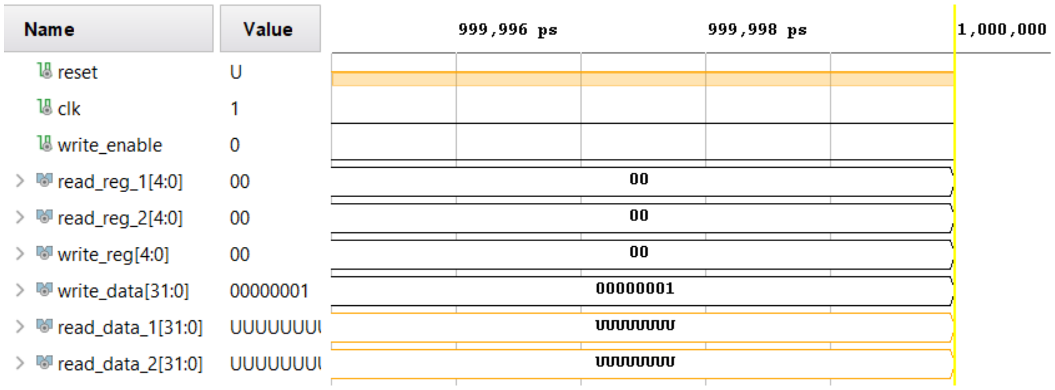
Fig8 shows the result of the first test case where a defined value for write_data was stored in the first index of the data array. However, the output from both read_data ports was zero because the reset value was set.



Name	Value
reset	U
clk	0
write_enable	1
read_reg_1[4:0]	00
read_reg_2[4:0]	00
write_reg[4:0]	00
write_data[31:0]	00000001
read_data_1[31:0]	00000001
read_data_2[31:0]	00000001

Fig9: Results of test case 2 without reset, and defined write_data value

Fig9 shows the results from the second test case where a defined value for write_data of 1 was stored in the first index of the data array. The output from both read_data ports was 1 because the reset value was not set.



Name	Value
reset	U
clk	1
write_enable	0
read_reg_1[4:0]	00
read_reg_2[4:0]	00
write_reg[4:0]	00
write_data[31:0]	00000001
read_data_1[31:0]	UUUUUUUU
read_data_2[31:0]	UUUUUUUU

Fig10: Results of test case 3 without reset, a defined write_data value, but with write_enable not set

Fig10 shows the results from the third test case where a defined value for write_data of 1 was stored in the first index of the data array. The output from both read_data ports was undefined because write_enable was not set so no data could be stored.

D. Performance Analysis

The most time critical path for this component was not obtained because of errors occurring with synthesis and implementation. The synthesis was able to be completed successfully, but the implementation failed as a result of an error that conveyed that there were not enough inputs on the part to support running the implementation for the Register File part. This issue will be further addressed going forward.

E. Conclusions

The main lesson learned from creation of the register file component was gaining a better understanding of how the component fits into the bigger picture of the processor datapath. The largest obstacle for this component was and still is the failed synthesis and implementation, but the group is optimistic that the error causing it is small and can be easily fixed with a little guidance from the handsome instructor. Overall this component of the project is a success and the group feels confident moving forward designing the ALU.

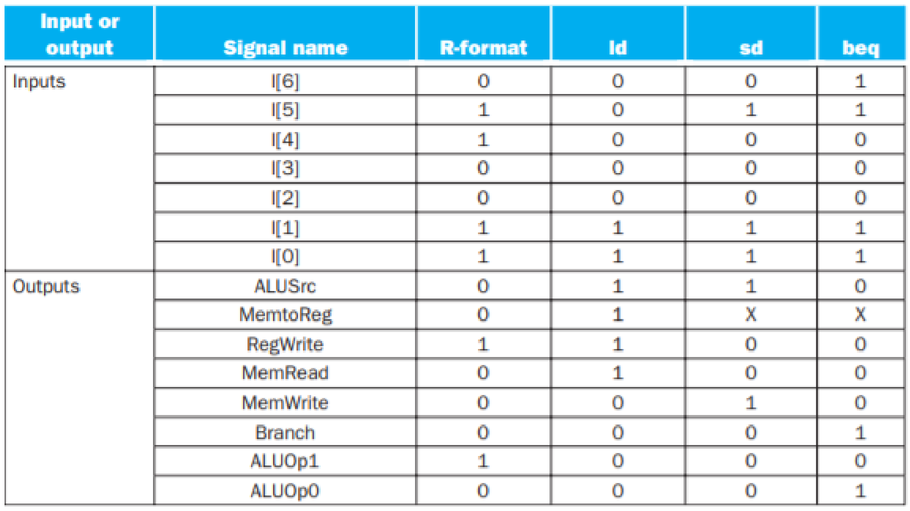
III. Main Control Component

A. Summary

The Main Control Component was tested using Vivado simulation tools. Various test cases were implemented within the testbench file and when tested, the simulation produced the expected outputs . The resulting simulation waveforms are further discussed in Part III, Section C: Testing, Verification, Results.

B. Description

For the main control unit, there were a variety of ways to complete the design using Vivado. As the combinations for the inputs for the main control unit were limited to the four found on the truth table in fig. 11, the decision was made to manually hardcode the unit's input and output. This allowed a quicker design as the team renewed their knowledge of Vivado and VHDL, it also allowed for additional flexibility and simplicity when it came to building test cases and troubleshooting. The main control is part of the overall control unit which also includes the ALU control shown in Part IV, its purpose includes fetching and executing instructions from the memory of the computer. The unit receives the input instruction/information from the user and converts it into control signals, these are then sent to the CPU for further examination and execution. The control units are also responsible for providing time and control signals and directing the execution of programs.



Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Fig11: Main Control Unit Truth Table given

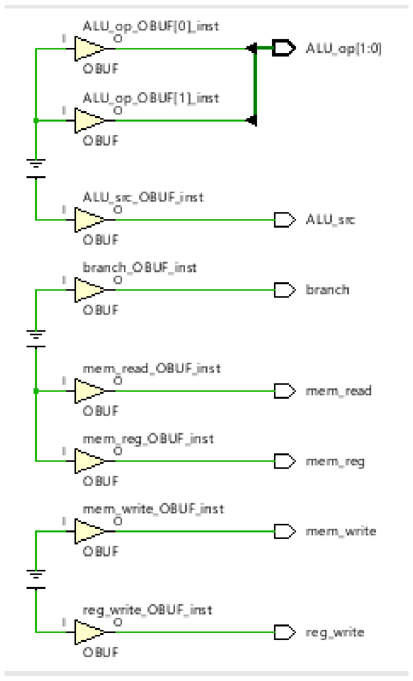


Fig12: Vivado Implementation schematic generated for the Main Control Unit.

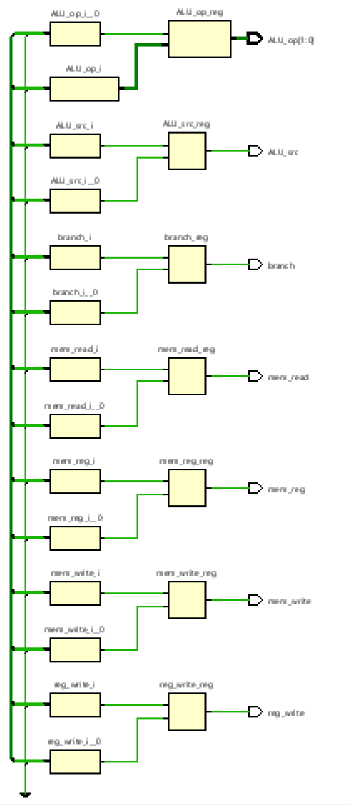


Fig13: Vivado RTL Analysis schematic generated for the Main Control Unit.

Figures 12 and 13 shown above depict two schematics given from Vivado of the main control unit. As can be seen, the main issue that arose was the lack of recognition given to the input of the opcode, creating a connection from the outputs straight to ground thus negating any mention of an input. This problem arose possibly due to the conversion of the array "opcode" to an integer variable "opcode1" using the conv_integer function in Vivado however, despite the troubleshooting attempted this was issue was unable to be resolved and will be worked out prior to the submission of the next part of this project.

C. Testing, Verification, Results

The testing methods used to verify the functionality of the main control unit is shown in the testbench file, 'mainControl_tb'. Four test cases were created to test the main control design, the simulation would not run due to the errors described above but the following test cases highlight the expected function of this unit.

Test Case 1

Case 1 tests when the input Opcode is "0110011". Here all inputs have a valuable input and the expected output is "00100010".

Test Case 2

Case 2 tests when the input Opcode is '0000011'. Here all inputs have a valuable input and the expected output is "1111000".

Test Case 3

Case 3 tests when the input Opcode is '0100011'. Here the mem_reg instruction is made up of don't care values in the truth table and so should not be taken into account. The expected output is "1x01000".

Test Case 4

Case 4 tests when the input Opcode is '1100011'. Here the mem_reg instruction is made up of don't care values in the truth table and so should not be taken into account. The expected output is "0x00101".

D. Performance Analysis

The performance analysis of the Main Control component could not be determined using Vivado simulation tools as a result of errors during the simulations as described in Part III Section B: Description. This will be addressed before subsequent projects parts are due and resolved to ensure successful implementation of the Main Control Component.

E. Conclusions

Through the creation and implementation of the Main Control Unit, the group was able to more closely associate themselves with the Vivado and VHDL systems to create schematics as well as learn valuable troubleshooting skills. Despite the roadblock found in the recognition of the inputs, students were able to create two different schematics and closely study the function of the Main Control. The largest roadblock found was the disconnect with the input and the problems that subsequently arose with the lack of input however this seems to be a relatively straightforward problem which will be resolved prior to the next portion of this project.

IV. ALU Control Component

A. Summary

The goal for the ALU control component was to test and simulate using vivado simulation tools. For this component, the group was able to successfully code this component in VHDL based on the given truth table. When creating the testbench, an error occurred that will be addressed before the next project implementation is due. The testbench for the ALU control should be fully functional by the due date of the next part of the project. Because of the obstacles faced during the simulation phase, there are currently no performance numbers to report.

B. Description

The process for the ALU Control started with creating the .vhd file itself. The inputs were defined first. An input called "func_73" was created to combine the instruction opcode of function 7 and function 3. This design decision was made for ease of code going forward in the implementation. Next the "alu_opin" input was defined for a two bit input in alignment with the alu opcode specified in the truth table given in the textbook example and shown during lecture. Next, a combination of if and else statements were used to incorporate the truth table into the design and give the ALU Control component the desired functionality. The next piece of the design was creating the testbench file. In the testbench, the inputs and outputs were once again defined and test cases were created. Each test case was designed to analyze a specific ALU opcode input and simulate to show whether or not the ALU was functioning properly.

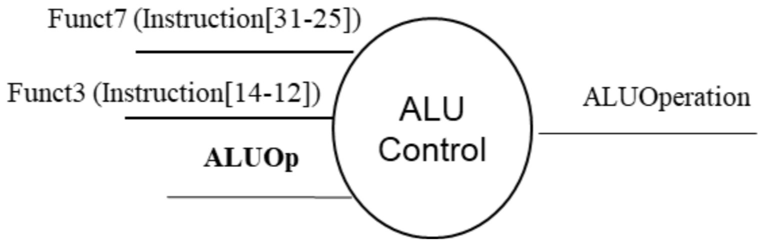


Fig14: General schematic of ALU Control component with input and output.

In the Vivado software, the schematic was not able to generate due to difficulties running the simulation. However, the expected schematic is shown in Fig.14. Figure 14 highlights the function 7 and function 3 inputs as well as the ALUOp input from the Main Control unit. The schematic also shows the ALUOp output which gets routed to the main Arithmetic Logic Unit in the pipelined processor. The big picture function of the ALU Control is to produce an input to the main Arithmetic Logic Unit in the processor. This input to the main ALU is necessary to determine operation is to be performed on the data.

C. Testing, Verification, Results

The testing methods used to verify the functionality of the ALU are described in the ALU
testbench file. Three test cases were created in order to successfully run the simulation. The simulation would not run due to unresolved errors, but the following test cases highlight what would be expected from a successful simulation.

Test Case 1

Case 1 tests when the input alu opcode is '00'. Here the func_73 instruction is made up of don't cares, so that input can be neglected. The expected ALUOpcode output is '0010'.

Test Case 2

Case 2 tests when the input alu opcode (0) is '1'. Here the func_73 instruction is made up of don't cares, so that input can be neglected. The expected ALUOpcode output is '0110'.

Test Case 3

Case 3 tests when the input alu opcode (1) is '0'. Here the func_73 instruction is no longer made up of don't care values in the truth table and so must be taken into account. The func_73 input in this case is '0000000000'. The expected ALUOpcode output is '0110'.

The ALU Control component program and testbench will be modified for subsequent parts of Project 2 to confirm functionality and allow for successful overall implementation.

D. Performance Analysis

The performance analysis of the ALU Control component could not be determined using Vivado simulation tools as a result of the errors which occurred during simulation. This will be addressed before subsequent projects parts are due and resolved for successful implementation of the ALU Control Component.

E. Conclusions

Through creation of the ALU Control component, the group has a better understanding of how the ALU Control component fits into the bigger picture of the processor as a whole. Additionally, through design and attempted implementation we were able to troubleshoot and determine possible sources of error that are going to be addressed as we move forward in the next step of the project. One of the biggest obstacles faced was addressing the errors given in the IDE. Our code was free of any obvious errors, so it was difficult to determine why the program would not produce a simulation. Overall, our ALU Control component is fairly complete and only needs to be analyzed to resolve minor errors. Once this is done, the ALU component should be fully functional for the next part of Project 2.