CMPE 260 Laboratory Exercise 1
Introduction to Vivado & Simple ALU

Glenn Vodra
Performed: January 26, 2023
Submitted: February 2, 2022

Lab Section: 3
Instructor Sutradhar
TA: Cole Johnson
Robbie Riviere
Jonathan Russo
Colin Vo

Lecture Section: 1
Professor Cliver

[illegible]

Abstract

The purpose of this exercise was to become familiar with Xilinx Vivado Design Suite and design several essential aspects of an ALU (Arithmetic Logic Unit). There were two parts to this exercise. Part one included the design and testing of a four-bit ALU that only supported two operations, “not” which is a bit-wise inverter, and a logical shift left, which shifts a specified number of bits to the left, filling in with zeros. A test bench was provided to test this ALU. The test bench was run in the following ways through Vivado's various simulations: Behavioral, Post-synthesis timing, and Post-implementation timing. Each of these methods tested the design in different ways, with the final simulating how the design would work with the propagation delay present on an actual FPGA board. Utilization information and a synthesis schematic were also generated at this stage. For the final bit of part one, code for generic, 4-bit logical right shift was written. An RTL schematic was also generated for the right shift.

The second part of this exercise involved completing the ALU to support six different arithmetic operations. These six were, OR, AND, XOR, Logical Left Shift, Logical Right Shift, and Arithmetic Right Shift. The NOT operation was removed from this version of the design. A test bench was also written to test this new ALU. It included a test record of forty-three test cases, some provided as “edge cases”, and others created as additional cases. The test bench also included “assert” statements that report errors in the case a result that is not expected is returned. This exercise was successful as all tests from both part one and part two, passed all three stages of simulation.

Design Methodology

The following figure, figure one, shows a block diagram representation of a full 32-bit ALU. In this figure, each of the blocks found in the middle represents a logical operation. There are six, operations, OR, AND, XOR, Logical Left Shift, Logical Right Shift, and Arithmetic Right Shift. The results of these computations are then sent a MUX on the right side of figure one. There, the signal OP (short for Operation Code) selects which result is needed.

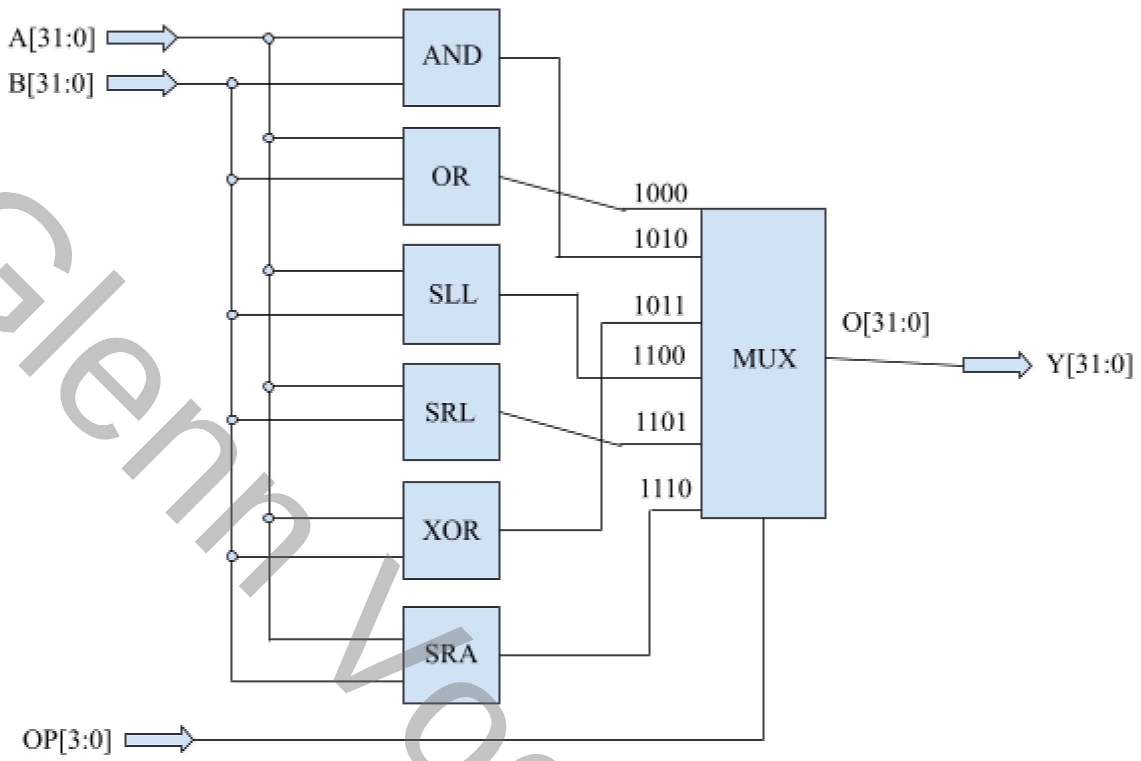


Figure 1: *Block Diagram of Full 32-bit ALU*

In this figure, it is also shown that the inputs to this design A and B are both 32 bits (31 down to 0), and the "select" for the MUX, OP, is four bits (3 down to 0). Figure 1 also shows the OP Codes needed for each logical operation. These are as follows, 1000 (OR), 1010 (AND), 1011 (XOR), 1100 (Shift Left Logical), 1101 (Shift Right Logical), and 1110 (Shift Right Arithmetic).

To highlight two aspects of this design, the right shifts, this is how they were designed. After reviewing the provided VHDL code for the Logical Shift Left, the following table was used to adapt the code to perform a bit shift in the opposite direction.

Table 1: *Logical Shift Right Logic*

S		Y(3) Y(2) Y(1) Y(0)		Assignment
"11"		'0' '0' '0' A(3)'		Y(3 downto 1) <= '0' and Y(0) <= Y(3)
"10"		'0' '0' A(3) A(2)		Y(3 downto 2) <= '0' and Y(1 downto 0) <= Y(3 downto 2)
"01"		'0' A(3) A(2) A(1)		Y(3) <= '0' and Y(2 downto 0) <= Y(3 downto 1)
"00"		A(3) A(2) A(1) A(0)		Y <= A

This table shows how different values of S (shift) affect the assignment of the outputs. The outputs follow the logic above and were assigned using a "for generate" loop. This allows the design above, which is meant for four bits, to be scaled up to 32 for the full ALU. This scaling is done with a feature of VHDL called generics. These allow a separate file to define values, such as bit width, or shift amount in this case. In the event this file is missing, the VHDL design will "fall back" to a default value defined in the design code. This feature allows a design to be tested at a smaller scale, and resized later, with little effort.

Table one only shows a logical shift operation. To change this design and build an arithmetic shift, only one change was needed. When zero values are assigned, the most significant bit is used instead. This allows a value to maintain its sign in two's compliment when being shifted.

Results and Analysis

As described above, the two ALU designs were tested using two separate test benches. Each test bench thoroughly tested the functionality of the design. The following figure, figure 2, shows the behavioral simulation results for the first ALU.

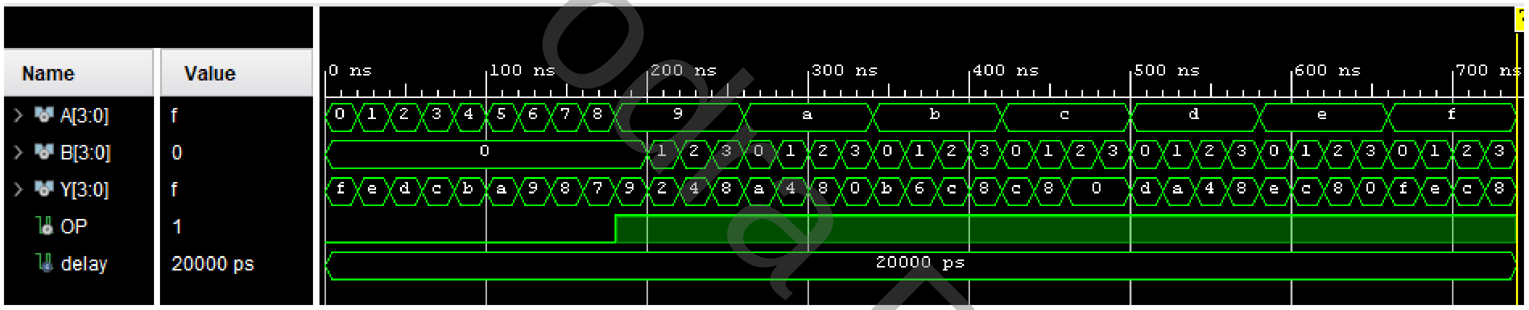


Figure 2: *ALU One Behavioral Simulation*

For this design, it was not mentioned above, but the first ALU utilized a different OP code. '0' is a NOT operation, and '1' is a logical shift. The first test case shows, the value "0" being inverted to "F", all 1s. The next shows the bits, "0001" being inverted to "1110" or "e" as shown in the waveform. Looking at the shift operations, the first test is not very interesting as no shift occurs. The next test shifts the value "9", "1001" left one bit, to "0010", or "2" as shown on the waveform. Another example is at 400ns, " B", "1011", is shifted left three times to get "1000" or "8". These are the expected results, and this ensures that the full 32-bit ALU will work as well, for its logical left shift operations.

The second test performed with this design was the post-implementation timing simulation where the delay of real gates is added and can be seen in the waveform. While nothing was done for the first ALU, the second design needed additional time for the signals to settle, before they could be read and evaluated. This test is shown in the figure below.

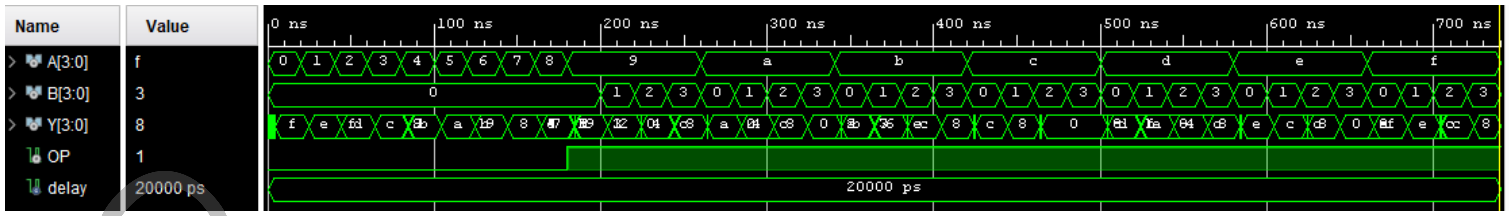


Figure 3: *Post-Implementation Timing Waveforms*

Two notable changes are seen in this waveform. These are the setup, at the start of the Y, output, signal, and the delay that can be seen throughout the output signal. There is no delay for the input signals, because they are coming from a test bench, and not another design. When additional components are added, that may fetch or decode the input signals, they will also have added delay.

The same two types of tests were run with ALU two, the 32-bit design. There is a lot of information in this waveform as it contains the result of all forty-three test bench cases. To explain the organization of this a little bit, the top half of the figure is the first half of the tests, which start with the nineteen, provided "edge cases". There are also four additional test cases for each operation.

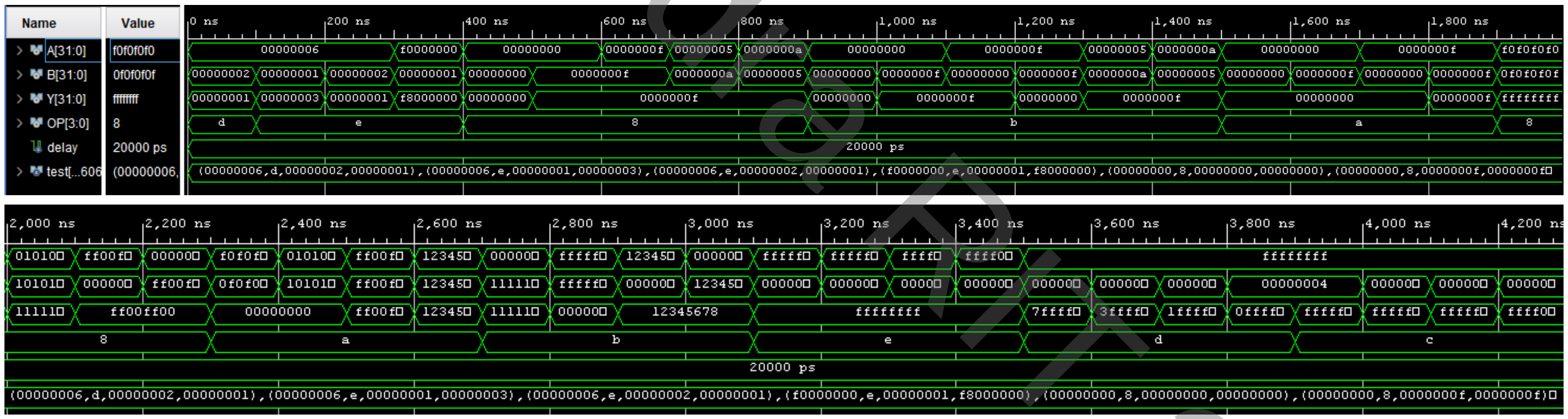
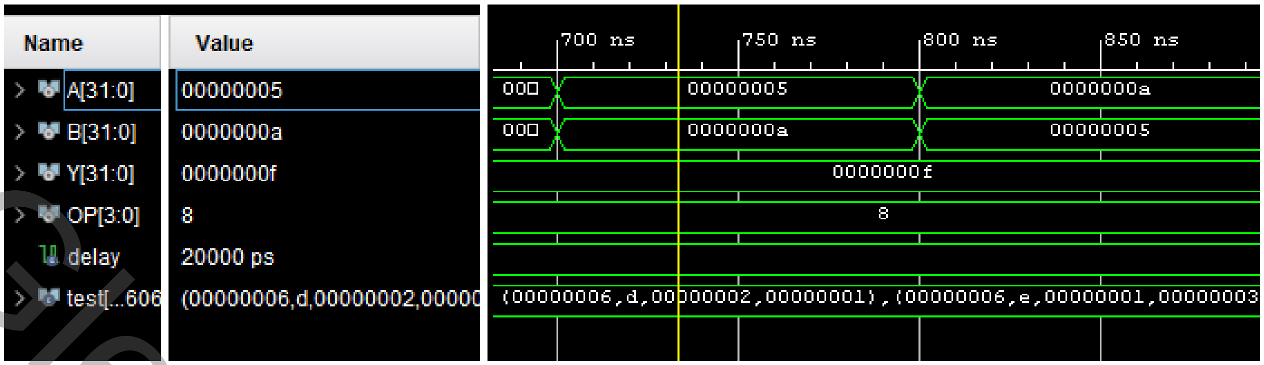


Figure 4: *ALU Two Behavioral Simulation*

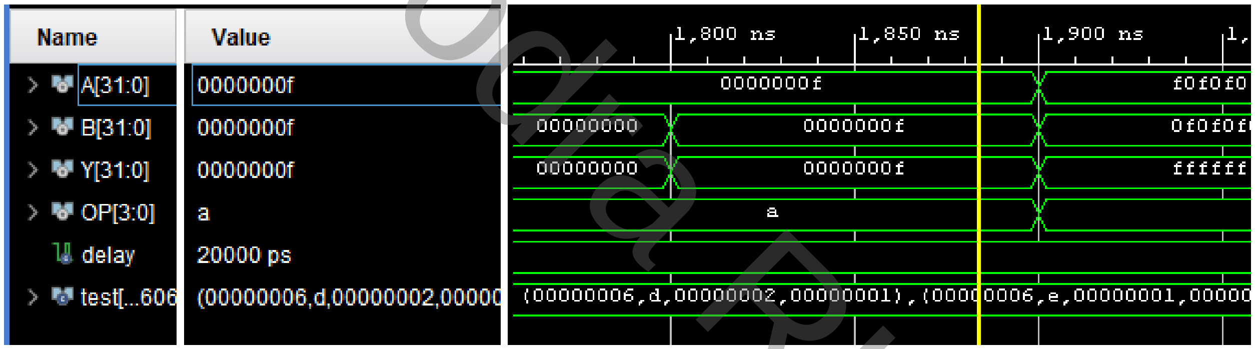
Since all of the information above is a lot, here are a few of the cases and how the results are represented in the waveform. The first of these, in figure five, is an OR operation.



Name	Value
A[31:0]	00000005
B[31:0]	0000000a
Y[31:0]	0000000f
OP[3:0]	8
delay	20000 ps
test[...606	(00000006,d,00000002,00000

Figure 5: *OR Operation*

At the maker (in yellow), the values read, A is “0x5” and B is “0xA”. The OR operation is the OP code “1000” shown as decimal “8”. Looking at the bits in “5”, “0101” and “A”, “1010” it is apparent that the result should be all 1s or “0xF” in hex. This test was performed correctly as that was the output result. Another operation that was tested was the AND operation. This operation has the OP code “1010” or “A” in hex. The result of an AND operation is shown in figure six.



Name	Value
A[31:0]	0000000f
B[31:0]	0000000f
Y[31:0]	0000000f
OP[3:0]	a
delay	20000 ps
test[...606	(00000006,d,00000002,00000

Figure 6: *AND Operation*

This operation takes the input “0xF”, “1111”, and ANDs it with “0xF”, also “1111”. Since all of the values are 1s (in this nibble) then the operation returns “1111” or “0xF”. As is with the above example, the result is extended to 32 bits. This brings up an important question. Do operations work in the other 28 bits? To answer this question the additional tests were designed to test all 32 bits. One of these, a logical left shift, is shown in figure seven.

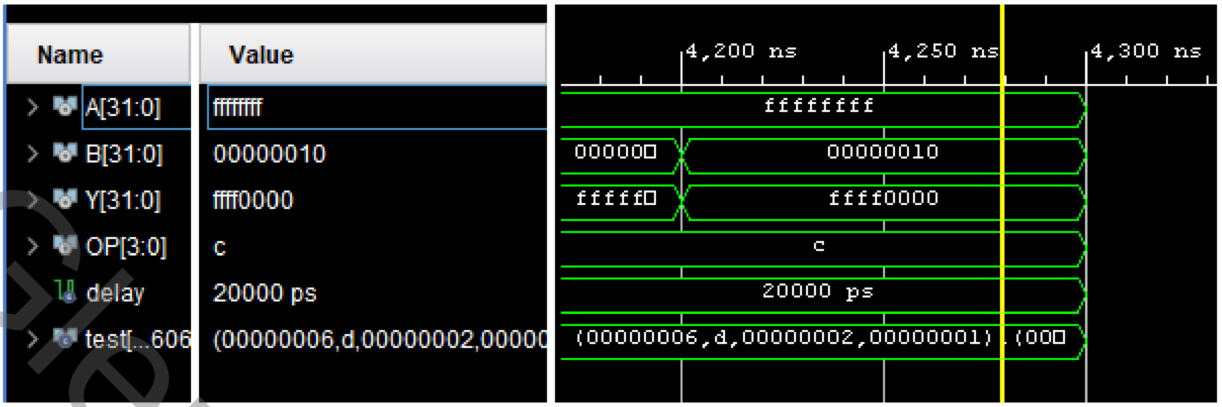

```
Name            Value
A[31:0]         ffffffff
B[31:0]         00000010
Y[31:0]         ffff0000
OP[3:0]         c
delay           20000 ps
test[...606     (00000006,d,00000002,00000
```

Figure 7*: Logical Shift Left Operation*

This operation uses OP code “0xC” or “1100”. The input A is the value to be shifted, “0xFFFFFFFF” or all 1s. Input B determines the shift amount, in this case, it is “0x10” or the decimal value “16”. This operation shifts the bits in A 16 bits to the left, filling in the remaining values with 0s. In the waveform, it is clear this is working because the output value is “0xFFFF0000”, the original value, shifted left, with 0 binary 0s. The next operation XOR has OP code of “1011” or “0xB”. This is shown in the next figure.

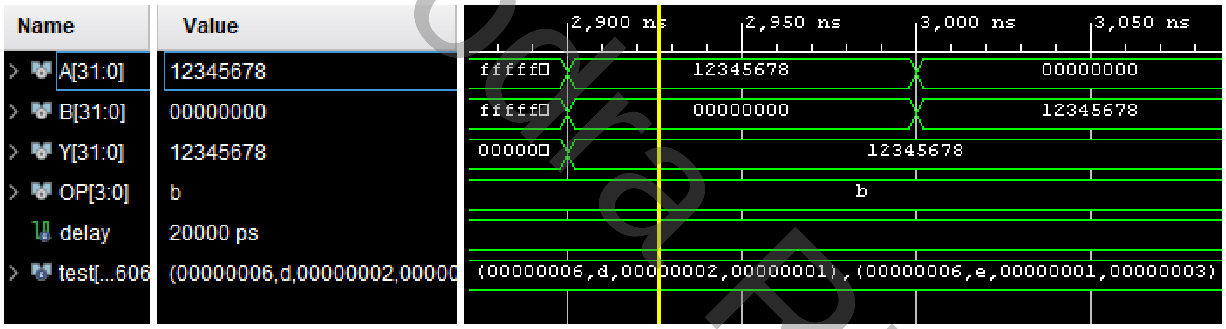

```
Name            Value
A[31:0]         12345678
B[31:0]         00000000
Y[31:0]         12345678
OP[3:0]         b
delay           20000 ps
test[...606     (00000006,d,00000002,00000
```

Figure 8: *XOR Operation*

XOR is true or ‘1’, if and only if one input is ‘1’. For this operation, the value “0x12345678” is XOR’ed with “0x00000000”. Because “0x00000000” doesn’t contain any 1s, the value in A is unaffected. In every place where there is a “1” a “1” is outputted, so the result is “0x12345678”. This is the value in the waveform above, in figure eight.

Additionally, the same checking was done for the logical shift right and arithmetic shifts. These were both explained extensively above in the previous section. The test benches used for these tests outputted “assert” statements if the output was correct. After running the simulation a few times, and correcting a few errors in the expected results, there were no messages printed indicating an issue.

As with the first ALU, a Post-Implementation Timing Simulation was run for the second, 32-bit, ALU design. The following figure shows this simulation. Due to the delay actual hardware would experience, an additional 90ns is added to each test bench test, before checking for errors.

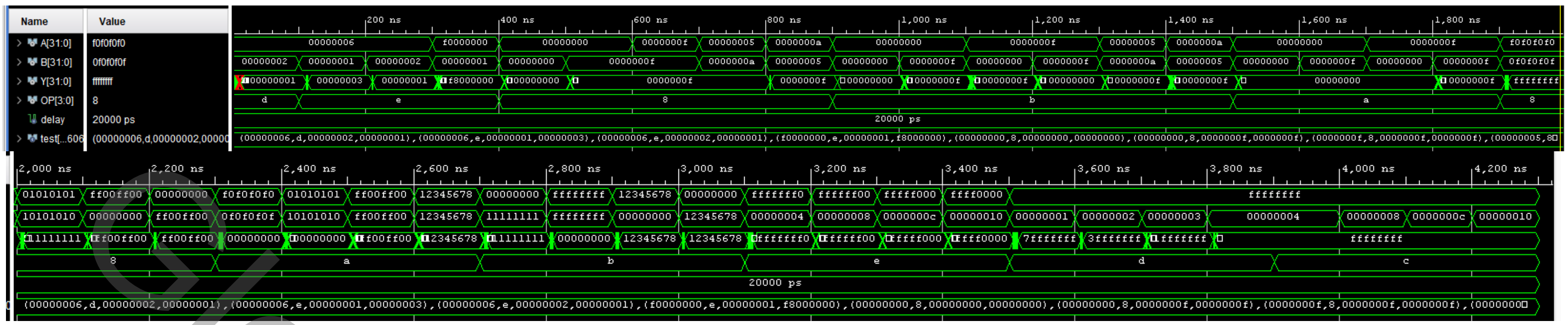


Figure 9: *ALU Two Post-Implementation Timing Simulation*

A note about this figure. There is a visual error, where there appears to be green before the setup has finished. A closer look can be seen in figure ten. Additionally, the light green regions are where the single is changing rapidly as a result is being selected from the MUX. This is normal and is the delay one might find with hardware.

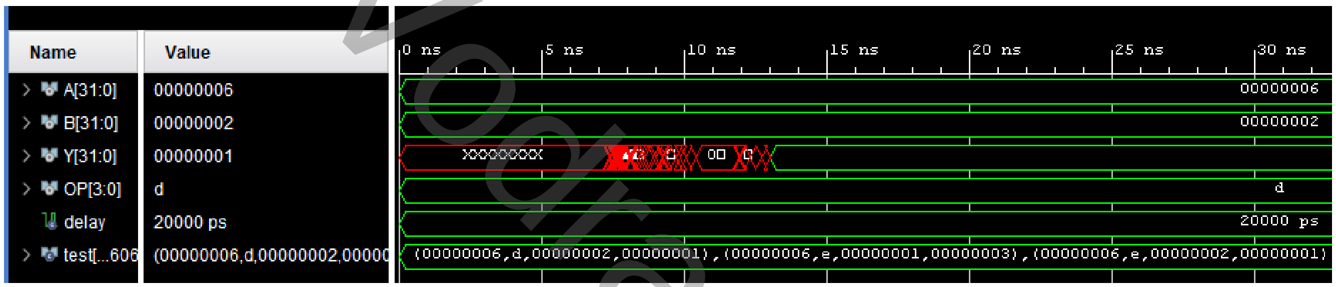


Figure 10: *ALU Two setup time*

As shown in figure ten, the design is actually performing normally.

Conclusion

This exercise was successful because a 32-bit ALU design was created and simulated in Xilinx Vivado Design Suite. While there was no hardware for this exercise, hardware conditions were simulated for both designs. The first design worked as intended performing NOT and Left Shift operations and the second design also performed as intended, building on the first design with, OR, AND, XOR, and both versions of the right shift. The ALU is an important piece of hardware for any machine that does computations. These simple operations are essential to have as they are used not only when doing “math”, but also implementing more complex logic and programs. This ALU will be used for future exercises, so it is very important to test thoroughly with accurate organized tests.