

# Contents

[1. Contents 1](#_Toc38051066)

[2. Introduction 2](#_Toc38051067)

[3. LogicUnit 2](#_Toc38051068)

[2.1 Overview & Implementation 2](#_Toc38051069)

[2.2 LogicGates 4](#_Toc38051070)

[2.3 Functional Behaviour 5](#_Toc38051071)

[2.4 Circuit Synthesis 5](#_Toc38051072)

[2.5 Timing Simulations 6](#_Toc38051073)

[4. ArithUnit 7](#_Toc38051074)

[3.1 Overview 7](#_Toc38051075)

[4.2 Implementation 7](#_Toc38051076)

[4.3 Adder 8](#_Toc38051077)

[4.3.2 Implementation 8](#_Toc38051078)

[4.4 Functional Simulation 8](#_Toc38051079)

[4.5 Circuit Synthesis 9](#_Toc38051080)

[4.6 Timing Simulations 9](#_Toc38051081)

[4.6.1 Procedure 9](#_Toc38051082)

[4.6.2 Observations 9](#_Toc38051083)

[5. Designing Barrel Shifters 9](#_Toc38051084)

[5.1 4.1 Overview 9](#_Toc38051085)

[5.2 Implementation 10](#_Toc38051086)

[6. Shift Unit 10](#_Toc38051087)

[5.2 Overview 10](#_Toc38051088)

# Introduction

The objective of this project is to design, synthesise and test a 64-bit Execution Unit. This is in preparation in assembling a RISC-V processor. This Execution Unit completes both 64-bit and 32-bit operations.

One of the two main components is the Arithmetic Logic Unit (ALU). The ALU is responsible for performing arithmetic and bitwise operations on integer binary numbers and is an essential building block of many other types of computing circuits. This ALU design is separated into two parts: the Logic Unit and the Arithmetic Unit. The Logic Unit appropriately performs logical bitwise operations on two 64-bit input signals while the Arithmetic Unit is responsible for producing the arithmetic result of two 64-bit input signals, depending on certain context variables.

The other main component is the Shift Unit is responsible for performing shifting operations on numbers. It uses three different barrel shifters, Shifting Left Logical, Shifting Right Logical and Shifting Right Arithmetic.

With these two main components, the Execution Unit is responsible for selecting the desired function to execute on the input numbers. This is done with key multiplexers placed in-between the inputs, components and outputs. Fundamentally, the Execution Unit is the unit that processes data and instructions in a processor.

After designing the respective units, the circuits are synthesized using Quartus Prime’s RTL netlist viewer. The execution unit and its sub-components will be synthesised for a Cyclone IV FPGA, specifically EP4CE115F29C7. We will then be using ModelSim to perform functional and timing simulations, which will be compared to the provided testbench values to verify that the units are working as intended.

The main objective is to complete a RISC-V Execution Unit which is compatible with RV64I architecture. This procedure has been further divided into 5 sub-categories as listed below:

1. A simple Logic Unit is also implemented which carries out bitwise operations (OR, AND, XOR) on two 64-bit signals
2. Implementing Arithmetic Unit which includes an output from the adder
3. To design Barrel Shifters
4. To design a shift unit that performs both 64-bit & 32-bit operations (using barrel shifters)
5. To design the execution unit that is compatible with RV64I operations using all the above circuits

# LogicUnit

## Overview & Implementation

The LogicUnit is responsible for selecting and operating Logic Bitwise operations of two 64-bit input signals, A and B. This design incorporates the following operations:

* Pass the signal of B
* The result of A XOR B
* The result of A OR B
* The result of A AND B

These initial logical operations are computed immediately, with the results passed along by a multiplexer as signal Y, depending on the signal LogicFn. The block diagram of the LogicUnit is represented in Figure 1 and the truth table of the LogicUnit is indicated in Table 1. The VHDL representation is given in Figure 2.

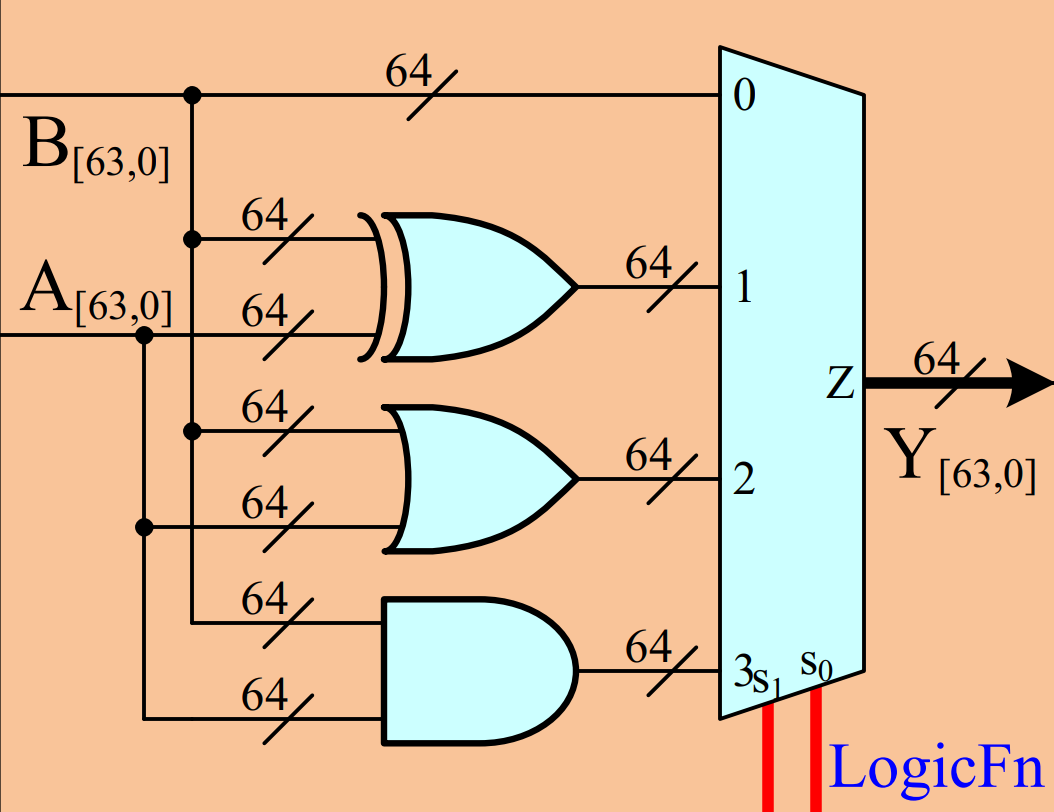


Figure 1:Block Diagram of LogicUnit Circuit

|  |  |
| --- | --- |
| LogicFn Signal | Operation (Signal Y) |
| 0 0 | B |
| 0 1 | A XOR B |
| 1 0 | A OR B |
| 1 1 | A AND B |

Table 1: Truth Table of LogicUnit

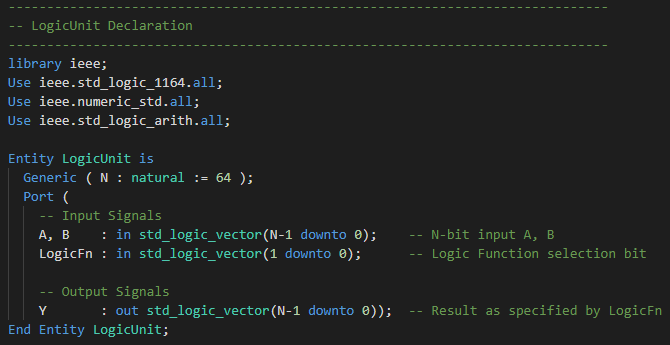


Figure 2: VHDL Interface of LogicUnit

## LogicGates

To aid the implementation of LogicUnit in VHDL, we implement the following four entities:

* XorGate – Y(i) <= A(i) XOR B(i)
* OrGate – Y(i) <= A(i) OR B(i)
* AndGate - Y(i) <= A(i) AND B(i)
* MUX4 – Selects one channel out of 4 channels using a 2-bit selectBit

These four entites are implemented in LogicGates.vhd and its VHDL interfaces are represented in Figure 3, Figure 4, Figure 5 and Figure 6 respectively. These 4 entities are used to abstract complexities away and to assist readers when reading our source code and debugging our applications.

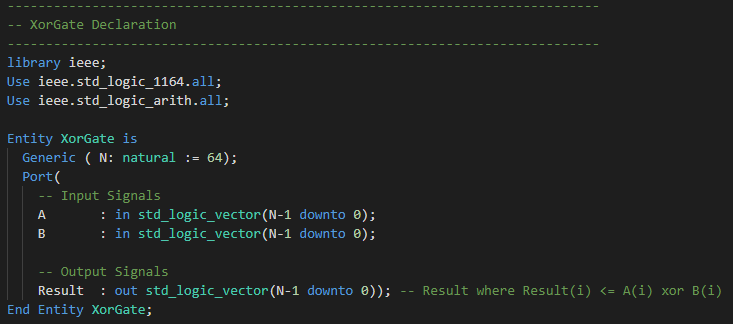


Figure 3: VHDL Interface of XorGate

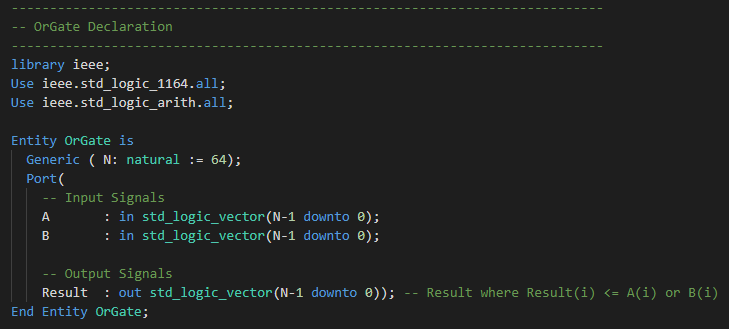


Figure 4: VHDL Interface of OrGate

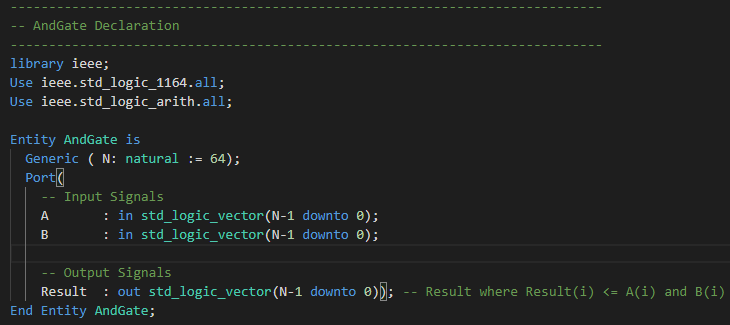


Figure 5: VHDL Interface of AndGate

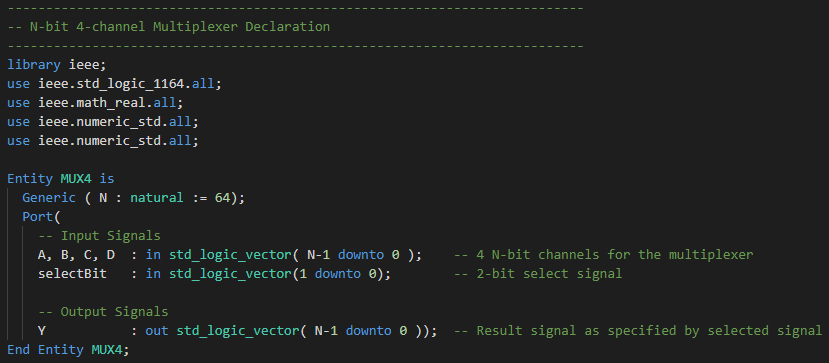


Figure 6: VHDL Interface of MUX4

## Functional Behaviour

The functional behaviour of LogicUnit can be demonstrated in . In , the four different LogicFn signals are demonstrated in the first 10 measurements. Because this figure is demonstrating functional behaviour, timing is not considered; this simulation is done as a proof of concept that our fundamental logic is correct. We performed 772 measurements, predefined in testbenches, with each measurement taking about 9 ns each and calculation one of the above listed bitwise operations.

## Circuit Synthesis

In Figure 6, we determine there are two levels of propagation delay. The first is due to the initial logic operations. The three logical diagrams are XorGate, AndGate, and OrGate. Each of those logical diagrams have a propagation delay of 1 gate as shown in Figure 6.

The second is due to the propagation delay of passing the results from the logic gates (or B) through the multiplexer. This second delay is expected to be much shorter than the initial logical operations. The propagation delay is one multiplexer as shown in Figure 7.

Figure 7: Synthesised Circuit of Multiplexer

Altogether, there would be a propagation delay of, at worst, one gate and one multiplexer as shown in Figure 8. At best, there would only be one multiplexer delay as shown in Figure 9; when the selectBit is “00”. This implementation aims to reduce the delay through the LogicUnit as much as possible. We will explore this in further detail when we run our timing simulations.

Figure 8: Example of slowest propagation through LogicGate

Figure 9: Fastest propagation through LogicGate when selectBit is “00”

Figure 7: Synthesised Circuit of LogicUnit

The timing of the LogicUnit is verified with this simulation. From Figures 7, 8, and 9, we can see that the results are obtained before the time period is up, with measurement #1 taking 13.9 ns, and measurement #772 taking 14.0 ns. For simplicity sake, we will take the propagation delay to be 14 ns.

Figure 7: Timing Simulation for LogicUnit of Measurement #1

Figure 8: Timing Simulation for LogicUnit from measurement #400 to measurement #416

## Timing Simulations

A closer examination of measurement #400 to #404 shows that the propagation delay is the same for all four operations.This means that LogicUnit will have a “constant” propagation delay for all operations. This is contrary to theoretical expectations. It is hypothesised that either ModelSim is not accounting for propagation delay inside for the logic gates or Quartus optimised the logic blocks in the simulated FPGA very efficiently.

Figure 9: Timing Simulation for LogicUnit of Measurement #400 to 404

Note that there are times in each of the measurements that the output stabilises at the correct result briefly before fluctuating again. This can be seen in Figure 10 and will be closely examined in Figure 11. In Figure 11, we can see that measurement #772 stabilizes at the correct result after around 1.2 ns, but it starts to fluctuate again due to a single bit having an undefined behaviour. This behaviour was observed on almost all the test cases. The computation is still happening during the 'fluctuation' phase despite the correct answer 'calculated' early. This is why we wait for the results to settle before reading the answer; not taking this into account may lead to garbage results due to timing.

Figure 10: Timing Simulation for LogicUnit of Measurement #772

Figure 11: Closer Look at Measurement #772

# ArithUnit

The ArithUnit is responsible for producing the appropriate arithmetic result depending on the context. Taking into consideration that adder output should be included in the execution of Arithmetic Unit, a new signal ‘adderOutput’ has been initiated which includes the output from Adder and stores it in ‘AddY’ signal within Arithmetic Unit implementation. In depth implementation of the Arithmetic Unit is described below along with the Functional and Timing simulations.

## Implementation

The implementation of the Arithmetic Unit is based on the block diagram in Figure 10. The VHDL interface of the ArithUnit is given in Figure 11.

The input context variables declared:

* A, B : 64-bit input signals to be used in the adder
* AddnSub : determines whether the operation carried out is an add or a subtract
* NotA : to be used later for retrieval of instructions
* ExtWord : determines whether or not to sign extend the value

The output variables declared:

* AddY : Raw output of Y (Stores output from adder)
* Y : result of the arithmetic operation
* Cout : outgoing carry of the result
* Ovfl : signifies an overflow in the result
* Zero : signifies if A is equal to B
* AltB, AltBu : signed and unsigned flags that indicate whether A is less than B

The ArithUnit was designed to handle both 64 bit and 32 bit numbers through the ‘ExtWord’ flag. It currently does not support any other operations other than addition and subtraction. If the operation performed is a subtraction, it simply negates the value in B and feeds it into the adder. The output signals are designed with future function implementations in mind. For example, AltB and AltBu may seem insignificant, but will eventually be used to implement branching instructions in the future.

Figure 10: Block diagram of the ArithUnit

Figure 11: VHDL interface of the ArithUnit

## Adder

#### Overview

The adder performs the addition bit-by-bit and propagates any carry that exists. Other than the result of the operation, it also returns the outgoing carry value, Cout and the Overflow flag, Ovfl. Cout is simply the final carry value of the carry array, and Overflow is computed as the XOR of the last and second-to-last carry values of the carry array. These values are used in the ArithUnit to compute AltB and AltBu. The timing considerations of an adder is important to the ALU (and thus, the execution unit) as most operations will involve addition and subtraction which heavily relies on the adder.

### Implementation

The entity Adder consists of 3 input signals and 3 output signals

Input Signals:

* A : value that needed to be added
* B : value that needed to be added
* Cin : Carry in value for that execution (previous carry or given carry)

Output Signals:

* Cout: Carry generated after the performed operation
* Ovrl : Overflow from the operation
* Y : result of the Computation

We chose to implement a ripple adder due to its simplicity and our priority focus is on the creation of an execution unit. Better adders can easily be substituted into Adder.vhd in the future. The block diagram of the Ripple Adder is represented in Figure 12. The VHDL interface of the Adder is given in Figure 13.

Figure 12: Block diagram of the Adder

Figure 13: VHDL interface of the Adder

## Functional Simulation

Figure 14: Functional Simulation of the ArithUnit from t = 0 to 76 ns

We can confirm that the ArithUnit is functioning as intended by observing that the Unit’s Y values are the same as the testbench – TbY. This figure also encapsulates all the different combinations of the context variables and its resulting output. For example, as annotated on the figure, we have a test case where the arithmetic operation is an ADD with no sign extension, that produces a Cout and no Overflow. We also have another test case that showcases SUBTRACT with sign extension, that similarly produces a Cout with no Overflow, but this time the AltB flag is also set.

Figures 15 and 16 further goes to show that our Arithmetic Unit is functioning properly, with both Y and TbY matching each other. However, it is important to also consider the timing of the unit, as a circuit that requires too long for computation may not be feasible. We will be performing timing simulations in the next section.

## Circuit Synthesis

Figures 18 and 19 represent the RTL synthesized circuit of the ArithUnit. The circuit is organized so that every input bit can be observed from the figure. The functional logic portions of the Unit are mostly performed in parallel, and only take up two layers of delay before producing the final result. The ripple adder used in this circuit is represented in green, and the synthesized circuit diagram of it is displayed in Figure 17. Since a basic ripple adder was used, the resulting propagation delay is quite significant. We could reduce this delay by implementing a carry-skip or Brent Kung adder instead. The output of the Adder is then used in the ArithUnit in the process of obtaining the final output value Y.

Figure 18: RTL synthesized circuit diagram of ArithUnit

Figure 19: RTL synthesized circuit diagram of Adder

## Timing Simulations

### Procedure

Timing simulations can be run by typing “do TimingArithUndi.do” in ModelSim Command window which compiles “ArithUnit.vho”, “TbArithUnit.vhd” & “ConfigExU.vhd”,load simulator using the configuration, TimeAUSim, sets up wave window using “waveArithUnit.do”, and runs the simulation for 5740 ns. The observations are as listed below

### Observations

# Designing Barrel Shifters

Barrel shifter is a digital circuit that can shift a data word by a specified number of bits without any use of sequential logic. Barrel shifters are used because it can accomplish in one cycle what it would take several cycles for a system without a barrel shifter. The barrel shifters operate on 64-bit data and use a series of three 4-channel multiplexers (MUXs) to shift the bits. Each MUX passes its result into the next, shifting the bits accordingly. Three barrel shifters were implemented in order to achieve different results:

SLL64 (Shift Left Logical 64-bits)

Shifts the value left by a particular amount of bits (up to 64). The shifted values are replaced by ‘0’s.

SRL64 (Shift Right Logical 64-bits)

Shifts the value right by a particular amount of bits (up to 64). The shifted values are replaced by ‘0’s.

SRA64 (Shift Right Arithmetic 64-bits)

Shifts the value right by a particular amount of bits (up to 64). The shifted values are replaced by the sign bit.

Each of the above operations have two input signals. One that contains data to be shifted & the other contains the amount it should be shifted - ‘ShiftCount’. And an output signal which stores the shifted data.

## Implementation

The main idea for designing the barrel shifters was to send in shifted signals(different for each MUX) along with the original signal. Nine new signals have been initiated which includes the shifted value of the signal by 1, 2, 3, 4, 8, 12, 16, 32, 48 respectively. For all the MUX’s defined below, appropriate shift count for separate bits is taken from the input signal ‘ShiftCount’ and passed as a fifth input for each MUX. For example, in case of First MUX, the last 2 bits of ‘ShiftCount’ is taken as a fourth input for an already defined ‘MUX4’ entity. Similarly for the second MUX, bits 3 & 2 is chosen as shifting amount and bits 1 & 0 for the third MUX.

For the first MUX, the original signal ‘Xsignal’ is passed into the MUX along with three shifted signals by 16, 32 & 48. The output of the First MUX, ‘firstMuxOutput’ along with another three shifted signals by 4, 8 & 12 is passed as an input for the second MUX. The output of the second MUX, ‘secondMuxOutput’ along with another three shifted signals by 1, 2 & 3 is passed as an input for the third MUX. The final output from the Third MUX results in the shifted value of the signal by amount of ‘ShiftCount’.

# Shift Unit

The ShiftUnit is responsible for shifting input value A left or right by a certain amount, determined by the last six bits of input signal B. It selects from three barrel shifters to use, which is then shifted again if needed for compatibility with 32-bit values.

## 6.1 Implementation

The ShiftUnit has 5 inputs and 1 output:

* A, B, and C: input values to be used in shifting or arithmetic
* ShiftFN: control signal that determines the function to be done
  + “00”: Pass C as the result Y
  + “01”: Shift Left Logical
  + “10”: Shift Right Logical
  + “11”: Shift Right Arithmetic
  + For shifting operations, the six least significant bits in B determines the number of bits to be shifted
* ExtWord: control signal that determines whether or not it’s dealing with 32-bit values, in which case a sign extension would be required
* Y: the result of the shift, or the input value C

The implementation of Shift Unit is performed based on the figure ##### which shows the circuit diagram and VHDL interface (if including the entity declaration for shiftUnit).

The ShiftUnit is designed to be able to operate on 32-bit values. A 32-bit value is still stored inside 64 bits so we can still use our 64-bit barrel shifters. However, if a 32-bit value was to be shifted right, the two halves of the 64-bit value are swapped, and then sign extended as needed. If a 32-bit value is shifted left by more than 32 bits, the halves are also swapped before sign extension. This is to ensure that we maintain useful information, and we do not end up passing a zero value when there are still shifted bits to be considered.

The circuit diagram and VHDL interface are given in figure ####

## Functional Simulation

A testbench for evaluating the ShiftUnit’s functional requirements was provided, as well as six test vectors. The test vectors each have a different combination of control signals (ShiftFN and ExtWord) in order to check different functions: SLL64, SRL64, SRA64, SLL32, SRL32, and SRA32. Each test vector has 512 different test cases for input values A, B, and C, and provides the correct result TbY to be compared with the output of the ShiftUnit, Y. The following diagrams only display two intermediate and the final three test cases for each test vector.

<< waveFSU\_SLL64\_1\_annotated.png >>

<< waveFSU\_SLL64\_2\_annotated.png >>

The first test vector tests for a logical shift left on 64-bit values. This can be seen through the control signals of ShiftFN and ExtWord: ShiftFN being “01” signifies a logical shift left operation, and ExtWord being “0” signifies that the values being operated on are 64-bit. We can verify that the ShiftUnit is working by observing that the output value Y is the same as the testbench expected output TbY. We can also see that only the last six bits of the input value B is populated, as expected from a shift operation. For each test case, the value in A is kept constant while B is incremented by 1 up to 63, before the constant in A is changed. This ensures that we are testing as many possible cases as possible. Converting the hexadecimal to binary can clearly show that our circuit is working, but we are unable to display this in the diagram due to how lengthy the value is.

<< waveFSU\_SRL64\_1\_annotated.png >>

<< waveFSU\_SRL64\_2\_annotated.png >>

The second set of test vectors tests for a logical shift right on 64-bit values. This can once again be seen through the control signals of ShiftFN and ExtWord: ShiftFN being “10” signifies a logical shift right operation, and ExtWord being “0” signifies that the values being operated on are 64-bit. TbY and Y matches, so ShiftUnit is outputting the expected values. B is once again incrementing by 1 for each constant of A, up to 63. We can see that this time, the output value Y is lesser than A, which is to be expected from a shift right operation. The final test case exemplifies why we only allow a shift up to 63 bits: a shift of 64 bits is a waste of processing time, as the result will always be 0.

<< waveFSU\_SRA\_1\_annotated.png >>

<< waveFSU\_SRA\_2\_annotated.png >>

The third set of test vectors tests for an arithmetic shift right on 64-bit values. ShiftFN being “11” signifies an arithmetic shift right operation, and ExtWord being “0” signifies a 64-bit input. It also further tests that our shift value extraction from B is working. The RISC-V ISA specifies that only the least significant six digits of B should be extracted as the shift value. For test case 249 in the figure, we can see that the value in B is 120ten, but as mentioned previously, it is a waste of processing power to shift more than 64 bits. Thus, only the six bits are considered, with the final result only shifting by 56ten. Since this is an arithmetic shift right operation, the shifted bits are replaced with the sign bit. This is done to ensure that the value maintains the correct result when interpreting it as a signed number.

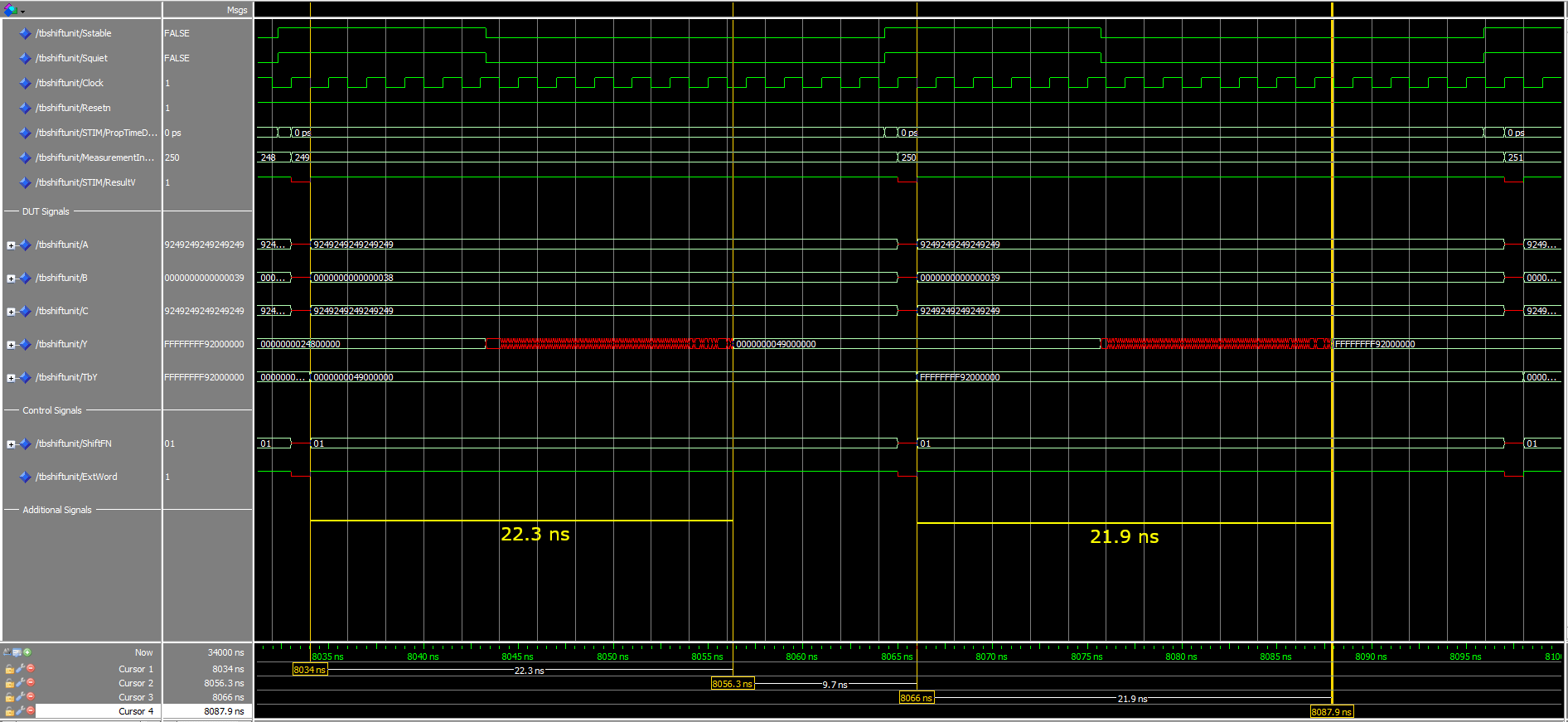


Figure 7: Functional Simulation of SLL32\_1

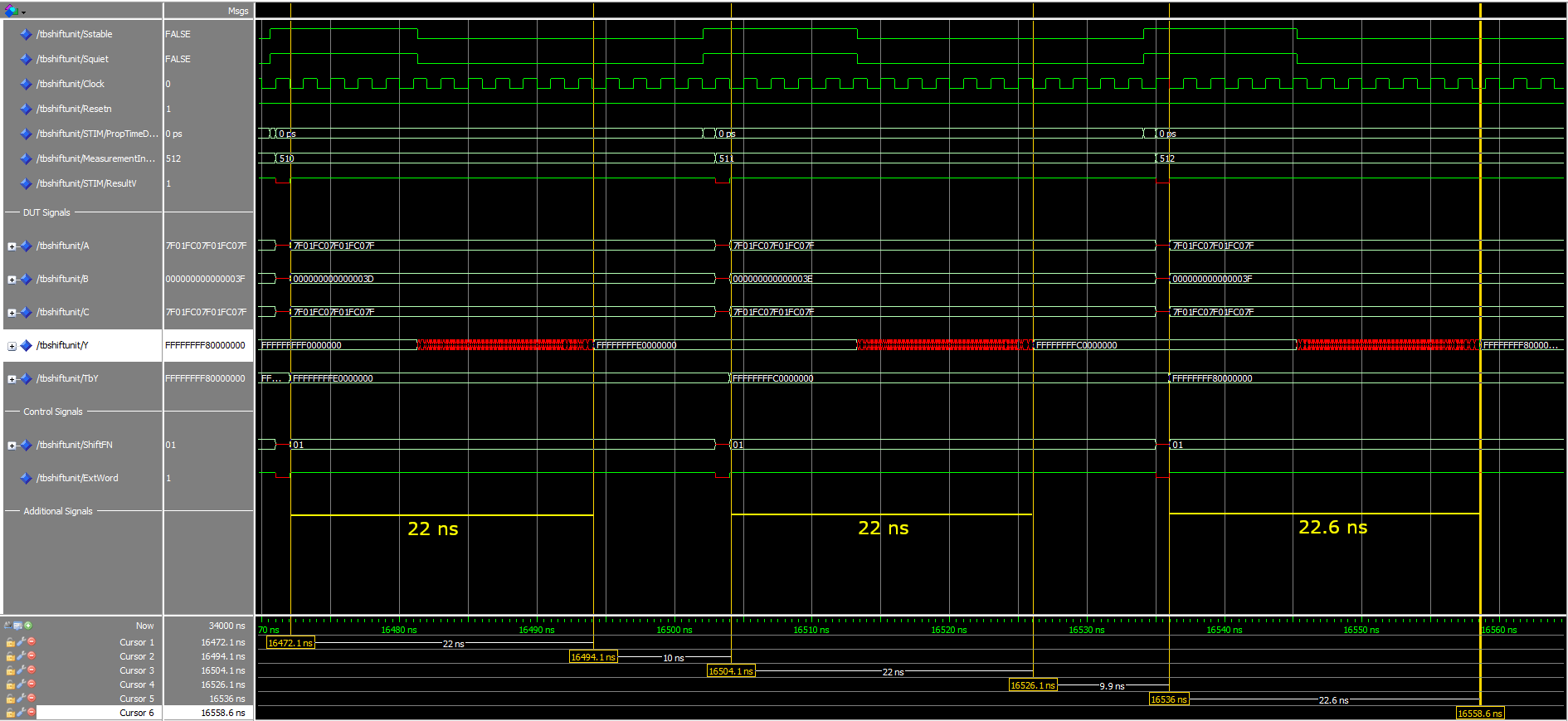


Figure 8: Functional Simulation of SLL32\_2

The next three set of test vectors will be testing shift operations on 32-bit values, as signified by ExtWord being “1”. For this set, ShiftFN being “01” indicates a logical shift left operation, so we are testing for a 32-bit logical shift left. Because the values are 32-bit, the first half of the 64-bit output should all have the same bit value from word extension. In test case 249, the number of bits shifted is 56, which is larger than 32 bits. Thus, our logic for swapping the halves described in Section 3.2 comes into play. In test case 512, the same situation occurs. If we were to simply shift left by 63 bits, and then perform extension, we will be left with a zero value result. However, because we performed the swap, our final result still retains useful information.



Figure 9: Functional Simulation of SRL32\_1

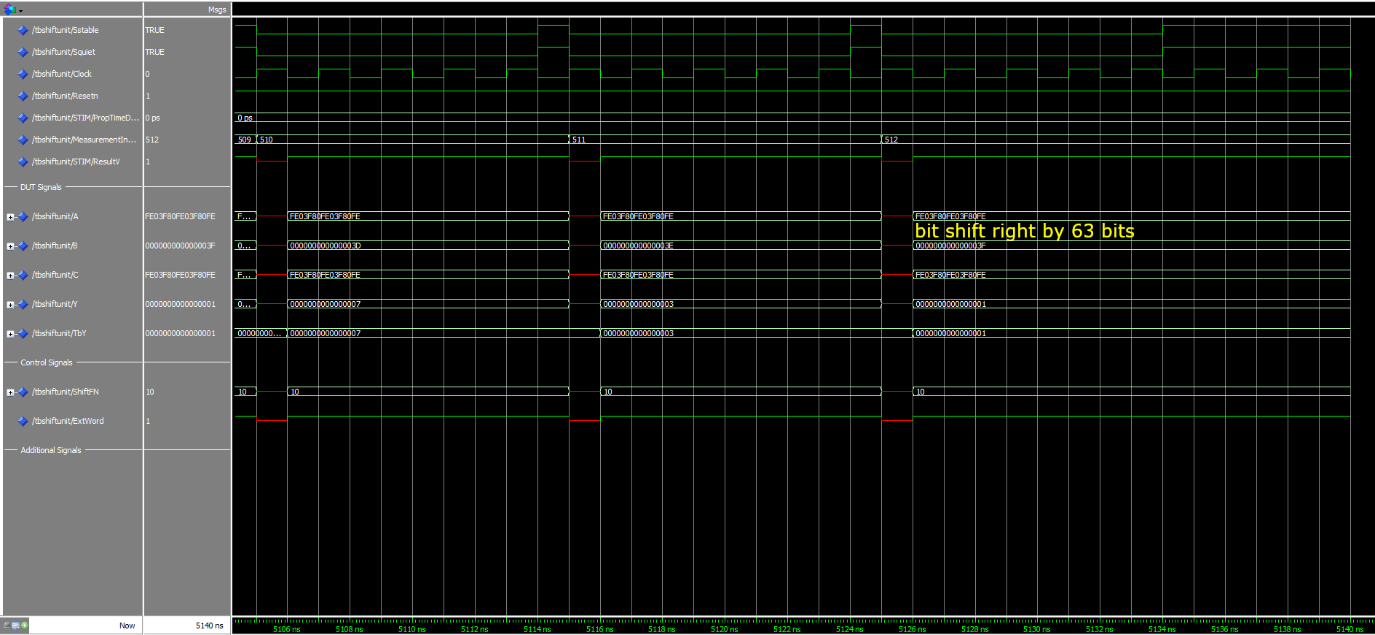


Figure 10: Functional Simulation of SRL32\_2

This set of test vectors tests for a logical shift right on 32-bit values, signified by “ShiftFN” being “10” (SRL) and “ExtWord” being “1” (32-bit). In the case of right-shifting 32-bit values, we always swap the two halves, as described in Section 3.2. This is so that we hold on to the shifted bits, just in case another swap is required. This time, however, we only swap the halves again if the number of shifted bits is less than 32 bits. Our ultimate goal here is to keep the useful information in the less significant half, and then perform sign extension.

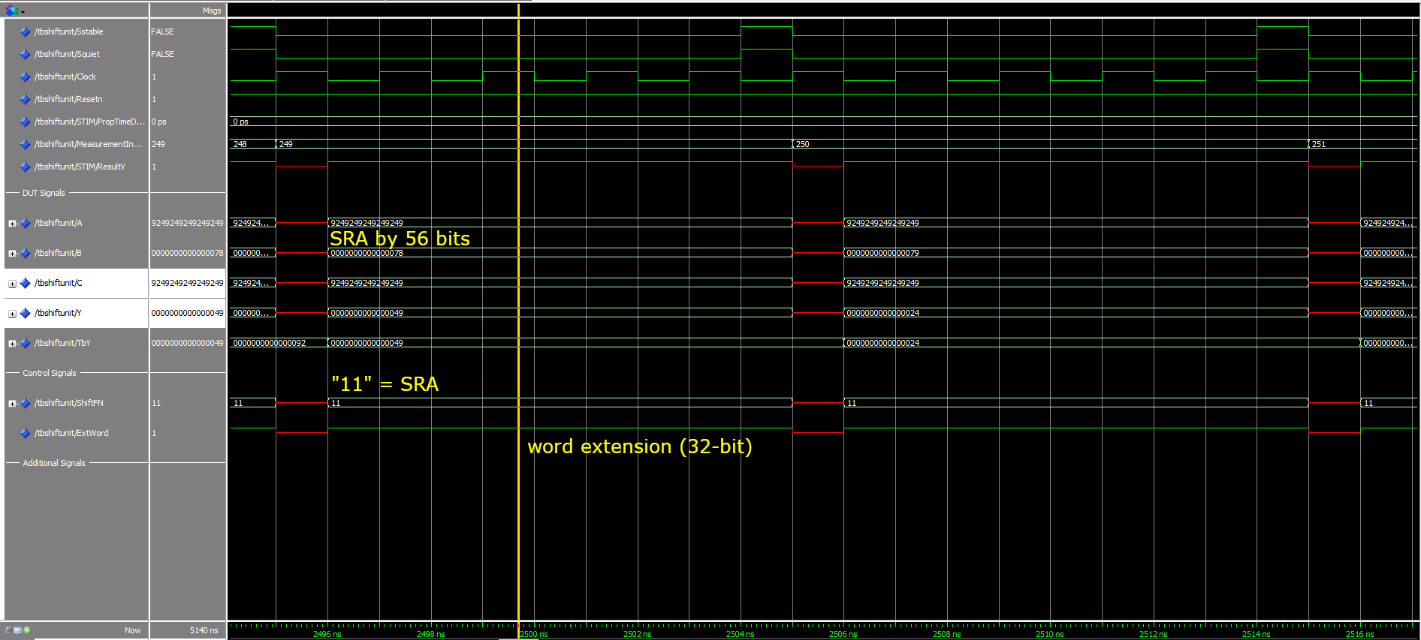


Figure 11: Functional Simulation of SRA32\_1

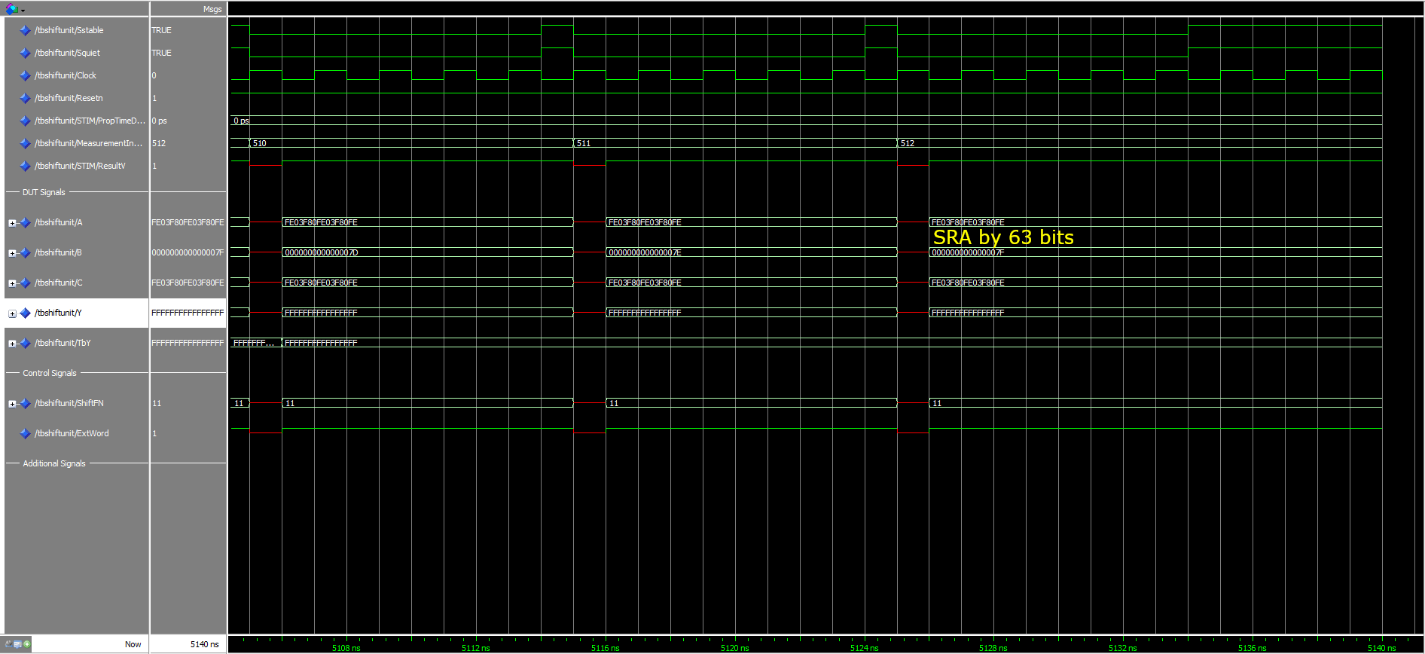


Figure 12: Functional Simulation of SRA32\_2

This final set of test vectors tests for an arithmetic shift right on 32-bit values, indicated by “ShiftFN” being “11” (SRA) and “ExtWord” being “1” (32-bit). The figures once again demonstrate that the bit extraction from B is working as intended. The same logic from the previous test vector applies here, in that the swap always occurs due to it being a 32-bit right shift operation. However, since this is an arithmetic shift, the shifted bits are replaced with the sign bit instead.

## Synthesis

The ShiftUnit circuit is synthesized in Quartus Prime, and the netlist viewers were used for visualization. The resulting figures were obtained, zoomed in on various parts of the Shift Unit circuit:

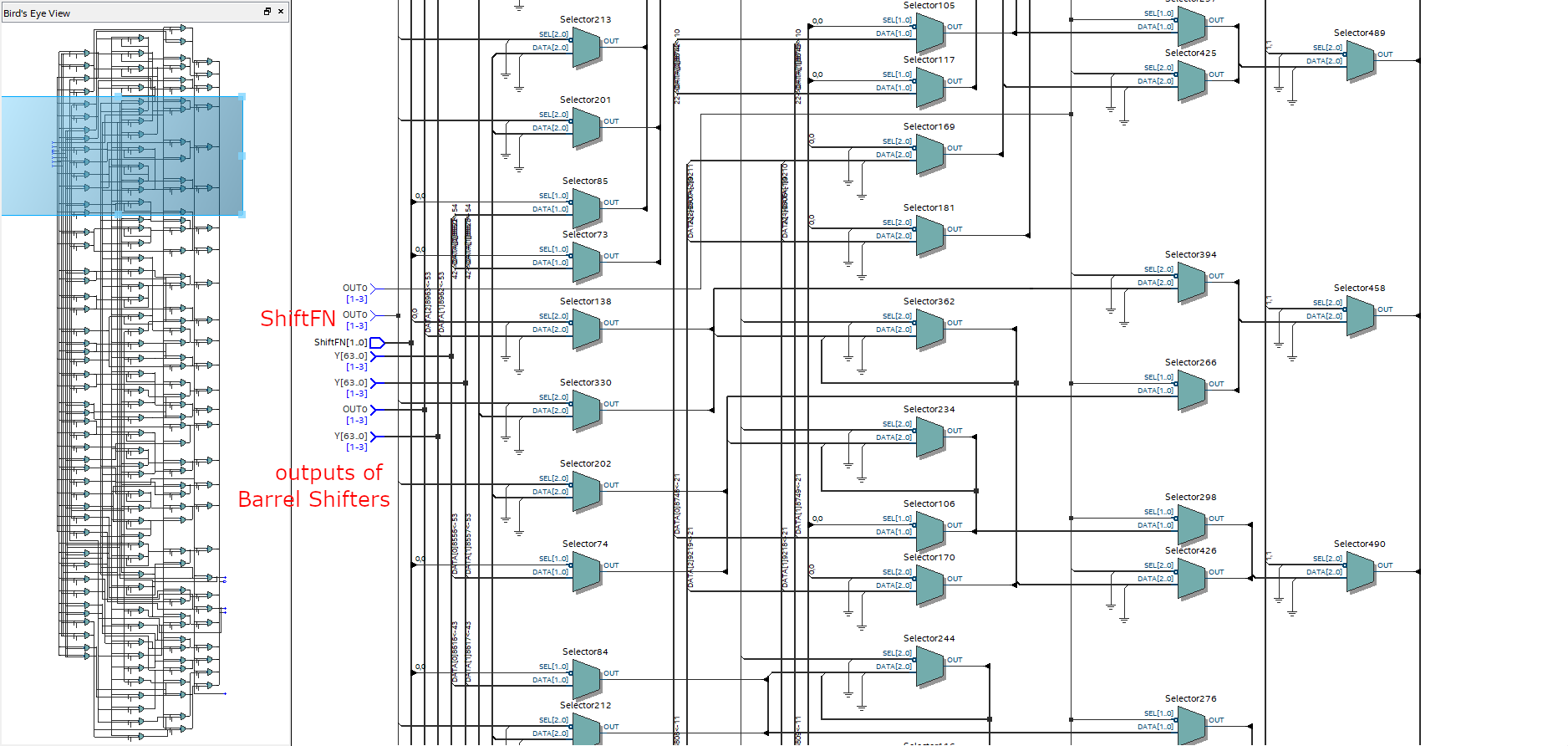


Figure 13: ShiftUnit RTL 1\_1

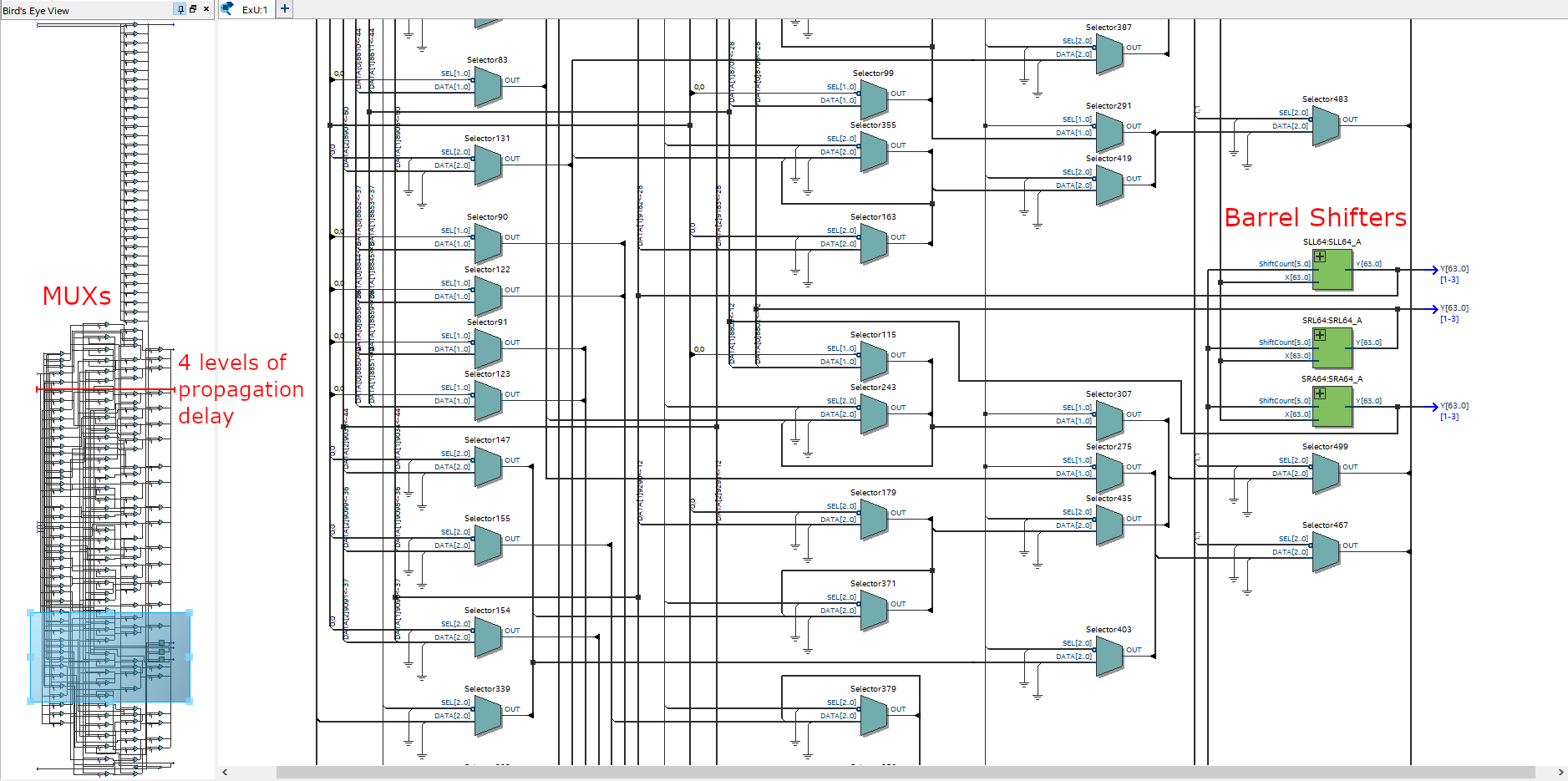


Figure 14: ShiftUnit RTL 2\_2

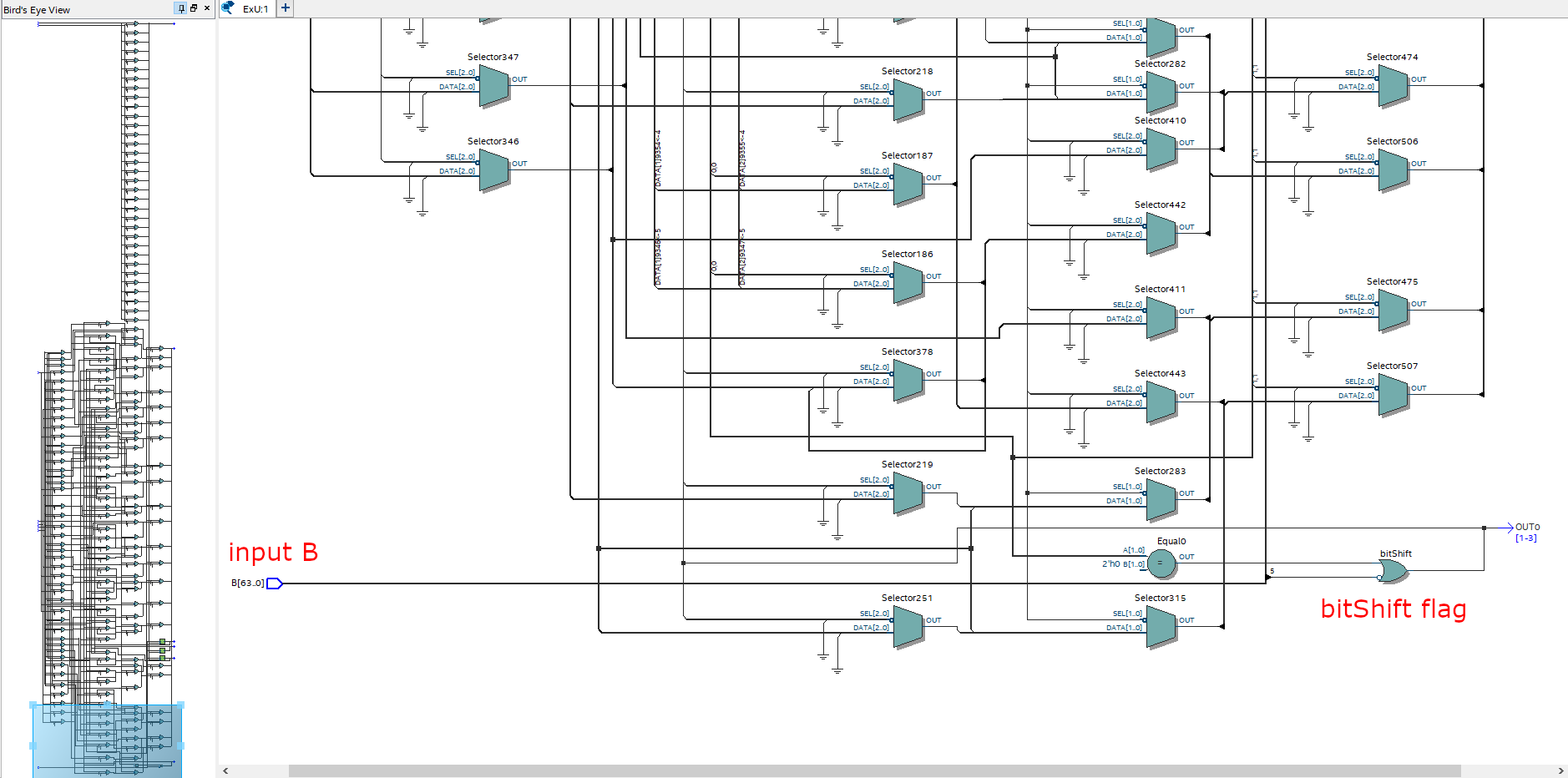


Figure 15: ShiftUnit RTL 2\_3

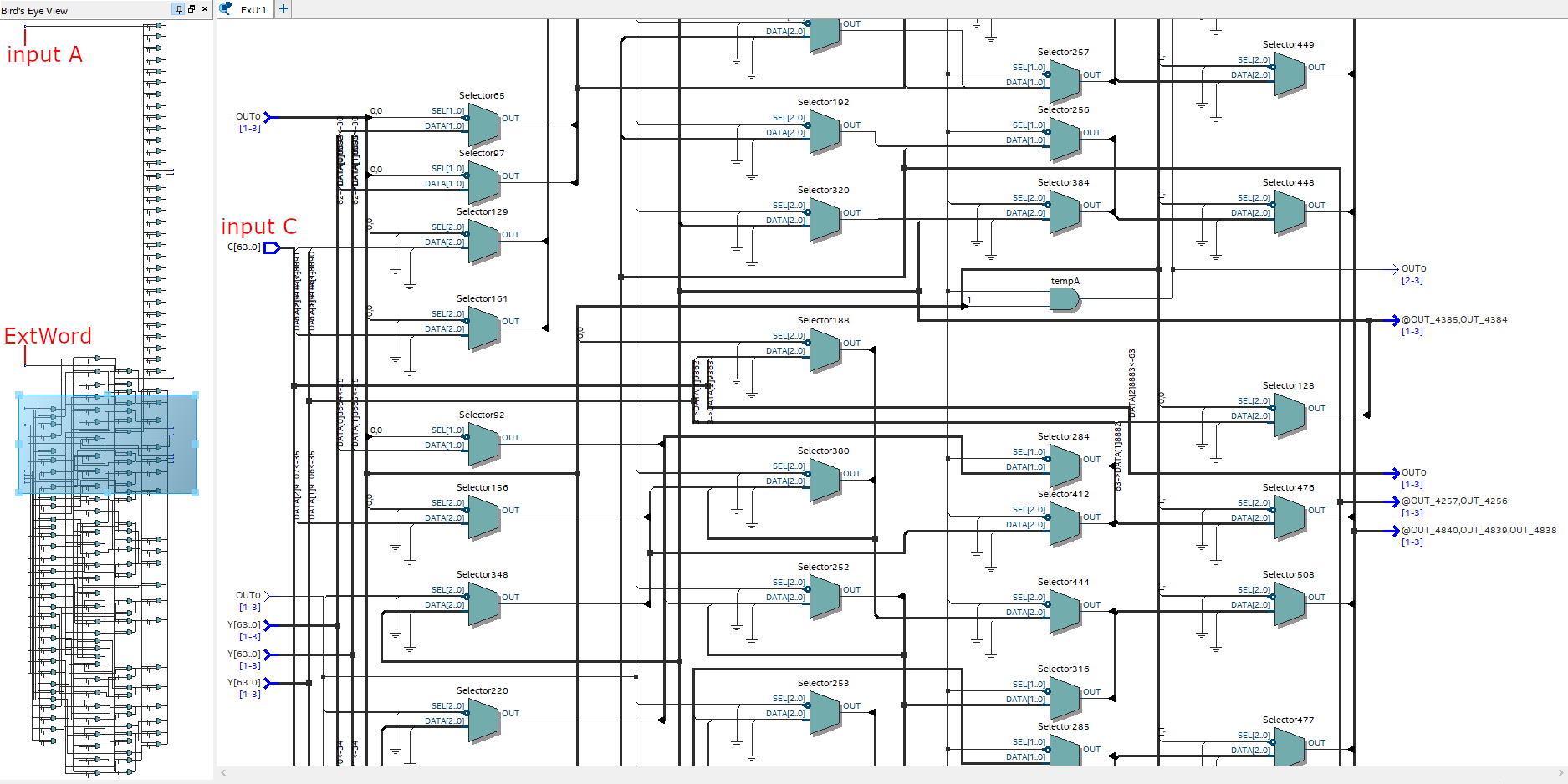


Figure 16: ShiftUnit RTL 3\_1

The figures above show the close up implementation of Shift Left Logical for 64 bit signals in RTL View.The values are first fed through each barrel shifter (represented in green) to obtain the respective shifted results. Each MUX then selects the appropriate result to be fed into the next MUX. These selections are determined by the control signals ShiftFN and ExtWord, as well as whether or not the words have to be swapped. The resulting circuit has 4 levels of propagation delay, as represented in figure ####. We can also see all the input signals A, B, C, ShiftFN, and ExtWord that feeds into their respective MUXes. The bitshift intermediate flag is also signified in FIgure ###, which signifies that a word swap is required. This logic was described in section 6.1.

## Timing Simulation

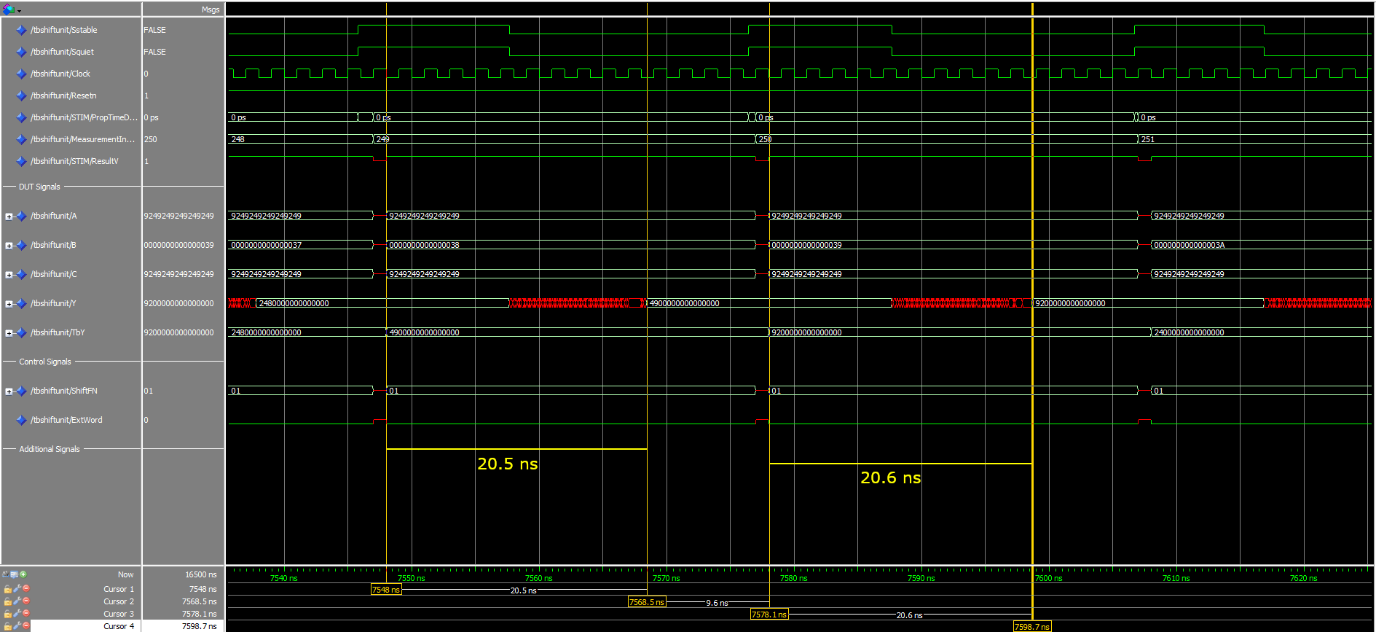
A testbench for evaluating the ShiftUnit’s timing requirements was also provided, making use of the same six test vectors as the functional simulation in Section 3.3. The following diagrams display two intermediate and the final three test cases for each test vector.

Figure 17: Timing Simulation of SLL64\_1

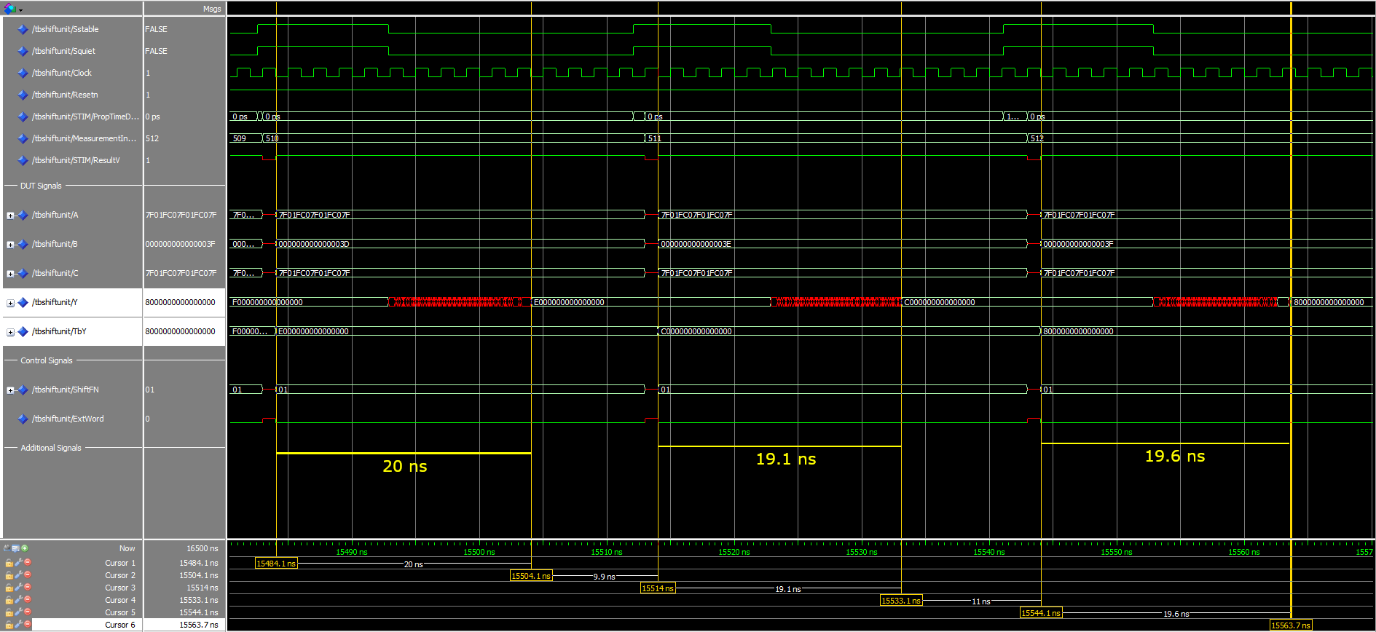


Figure 18: Timing Simulation of SLL64\_2

The first test vector is a 64-bit logical shift left. There is a consistent propagation delay of between 19.0 ns to 21.0 ns. This consistency can be attributed to the barrel shifters, discussed in section 2. If we were to re-implement this without using barrel shifters, there will be varying amounts of delay for different shifts, as a larger shift would require more cycles.

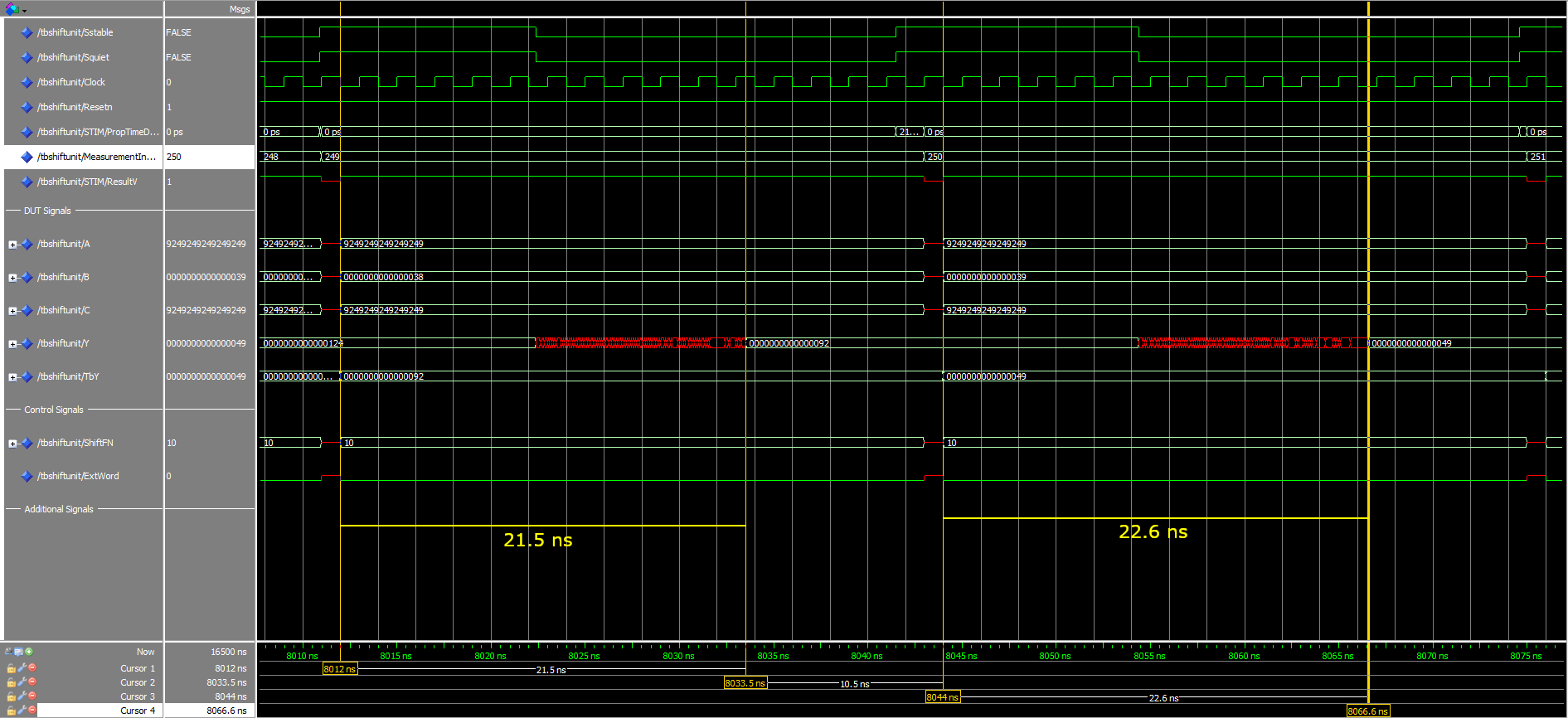


Figure 19: Timing Simulation of SRL64\_1

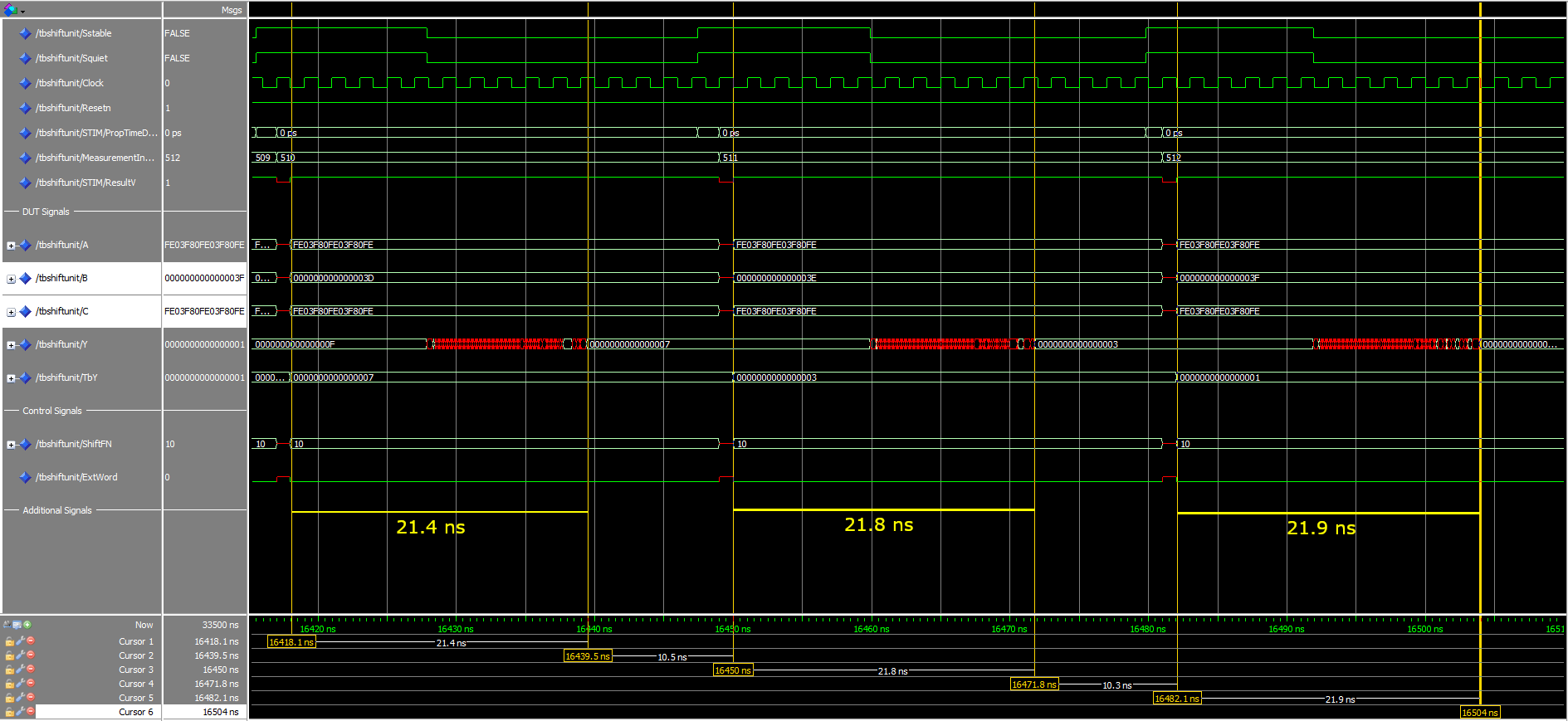


Figure 20: Timing Simulation of SRL64\_2

The second test vector is for 64-bit logical shift right. Once again, we see a consistent propagation delay, but slightly higher than SLL64 at approximately 22 ns. However, this increase in delay is still miniscule and should not significantly affect the speed of our processor.

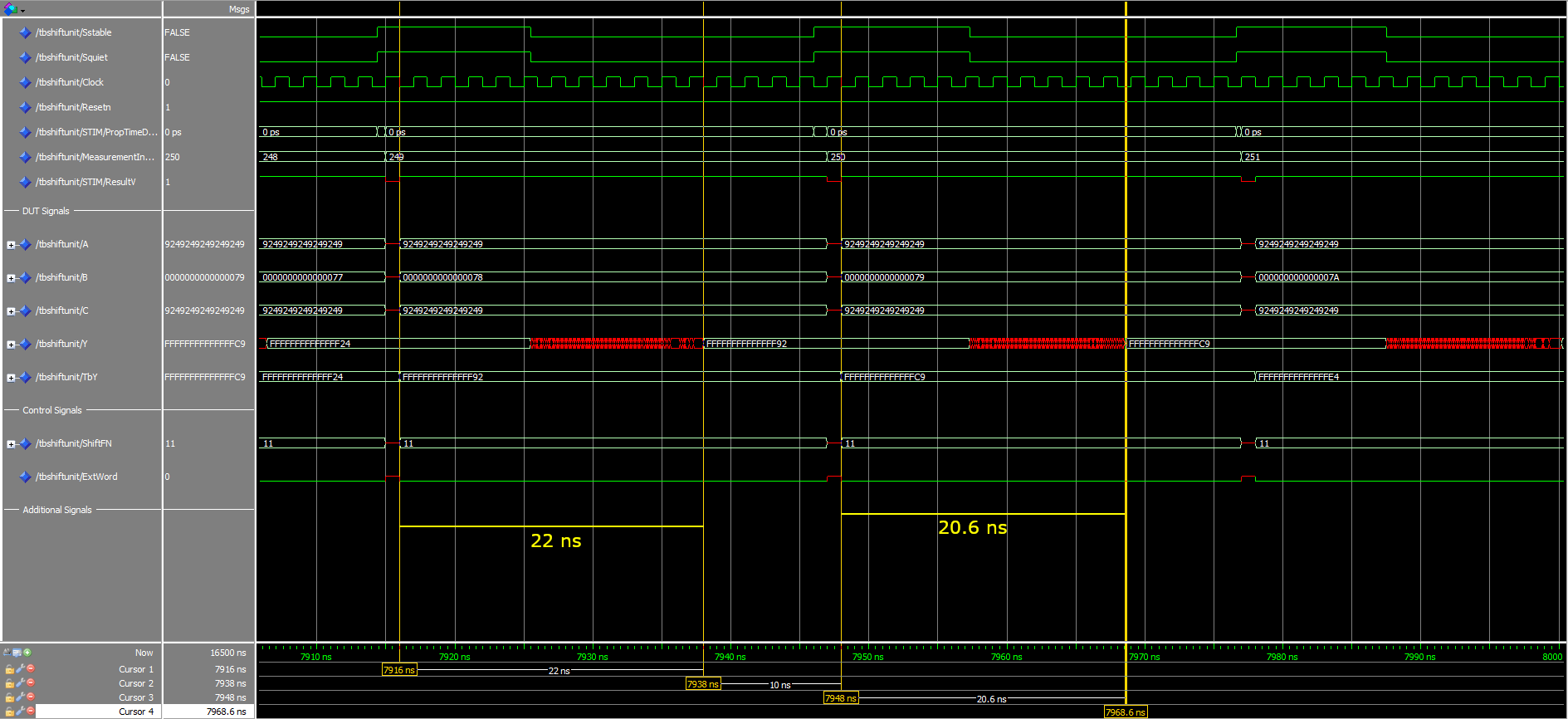


Figure 21: Timing Simulation of SRA64\_1

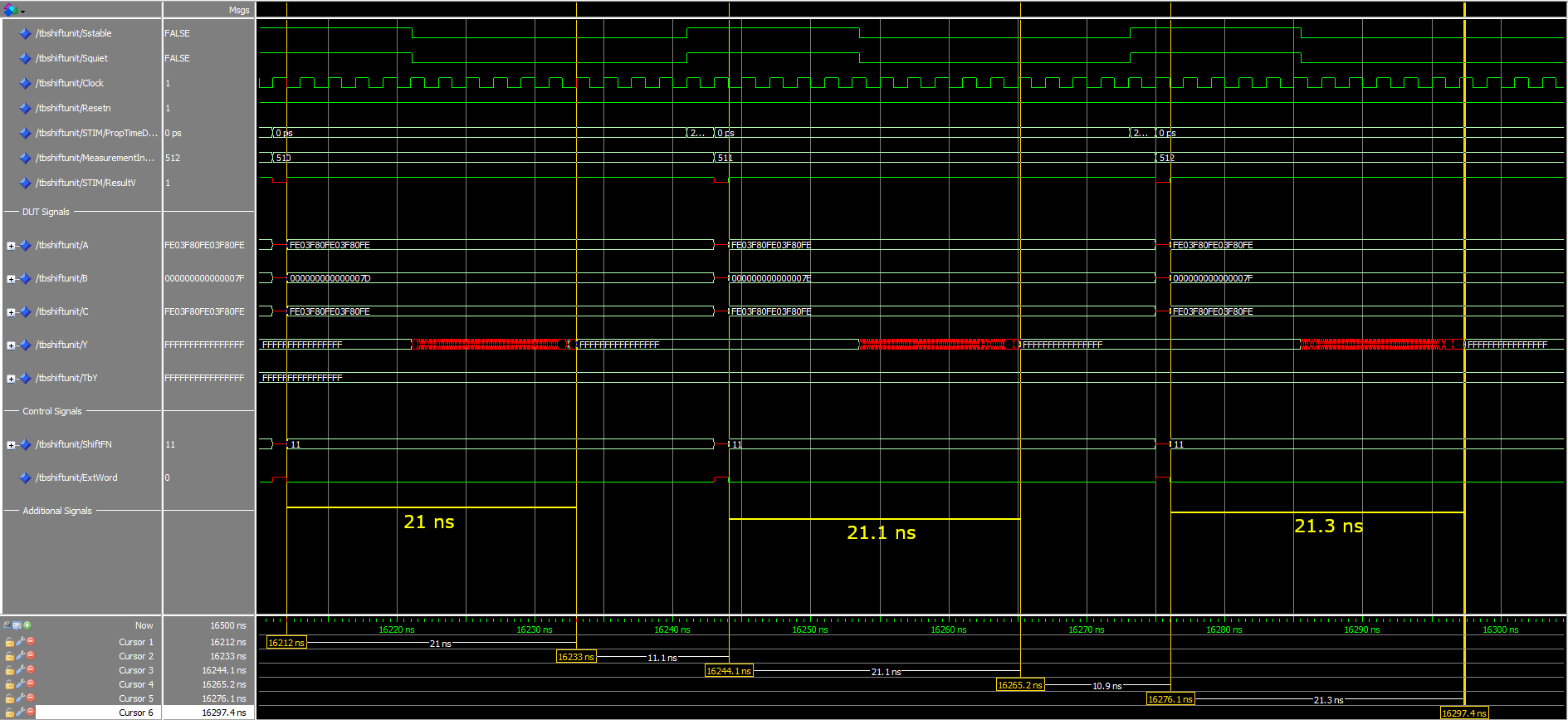


Figure 22: Timing Simulation of SRA64\_2

The third test vector is for 64-bit arithmetic shift right. The timings of the shift are all within the allocated time frame, verifying the timing requirements of the entity. We can see that even though SRA requires that we extract the sign bit and duplicate it, the amount of time required is still around the same.

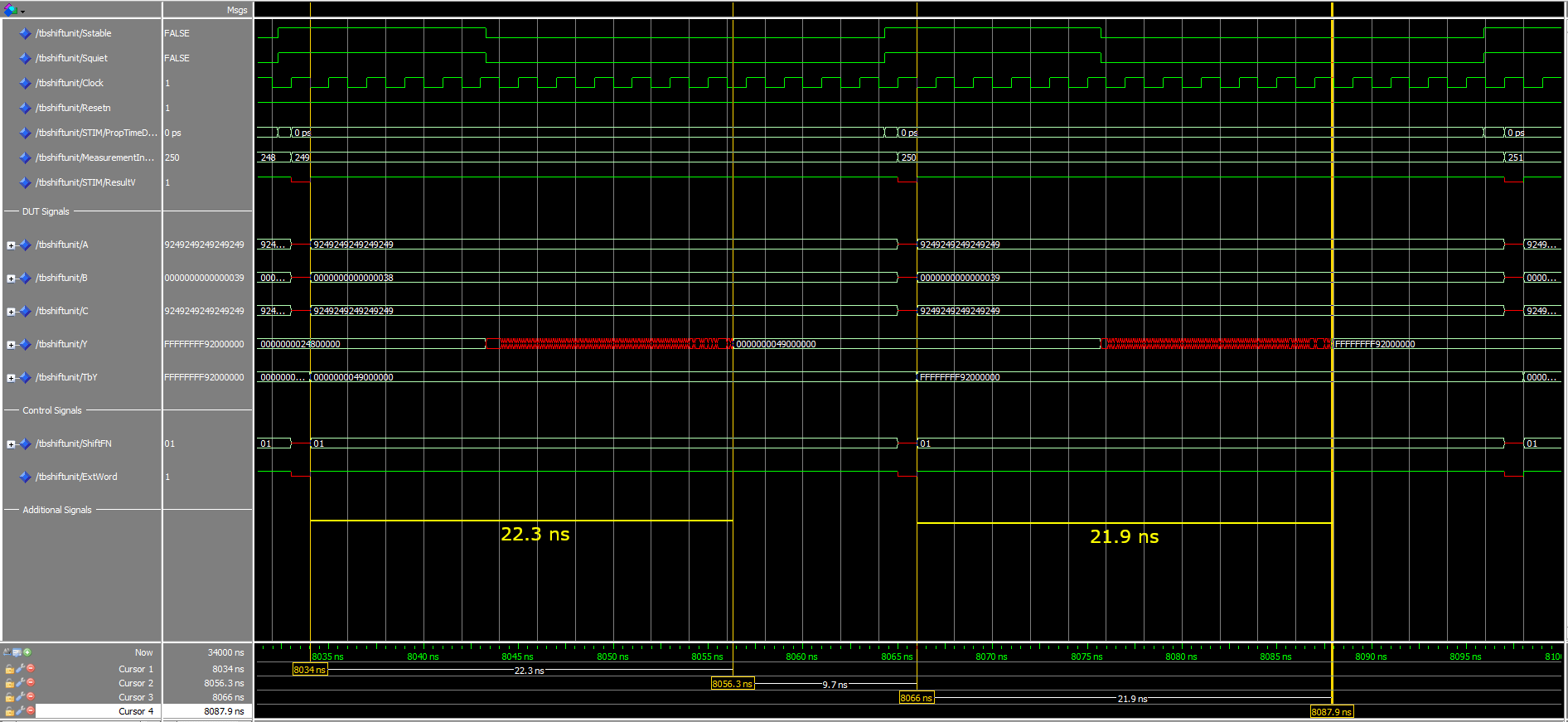


Figure 23: Timing Simulation of SLL32\_1

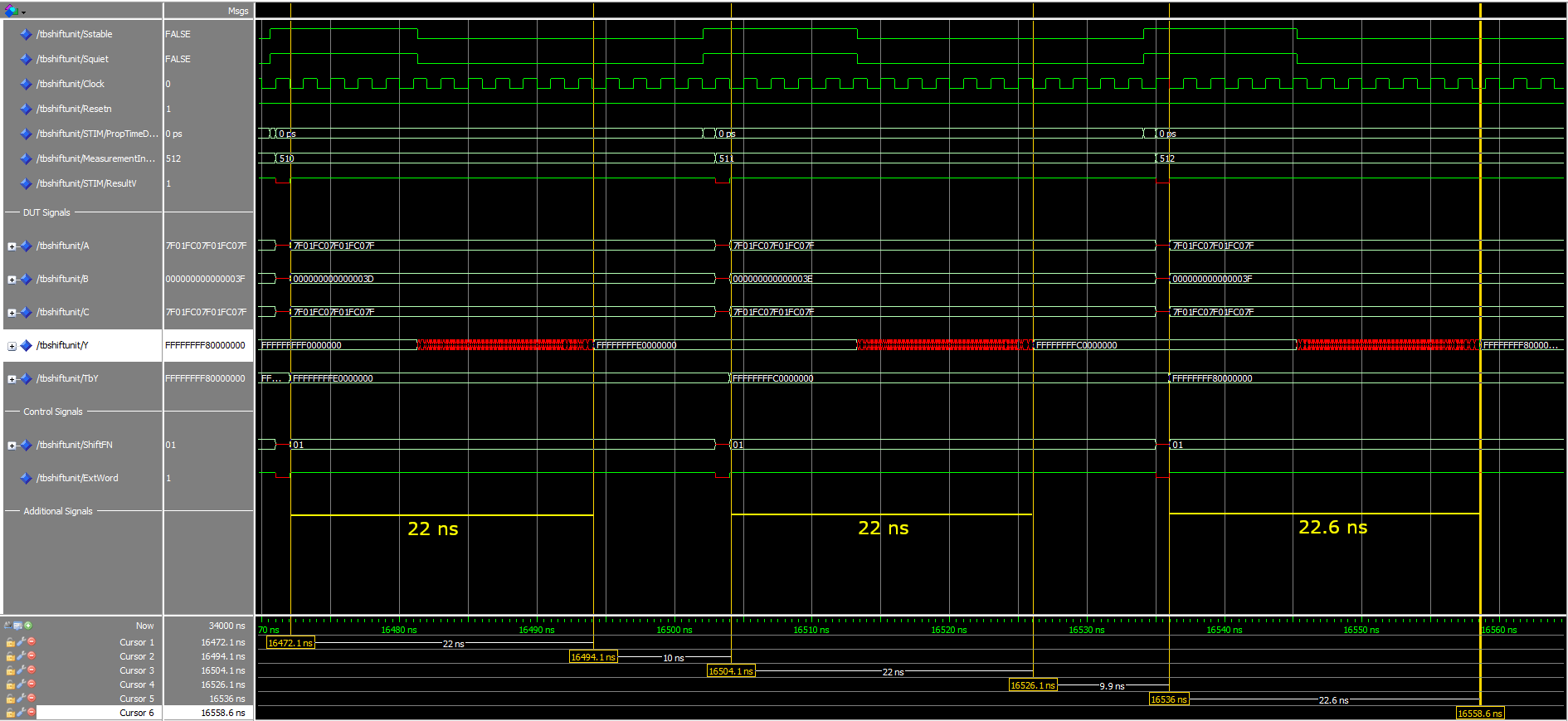


Figure 24: Timing Simulation of SLL32\_2

As expected, the operations on 32-bit values will have a slightly longer average processing time than that of the 64-bit operations. This is because our system is optimized for 64-bit. Even though it is compatible with 32-bit, a few more operations are required to output the correct value.

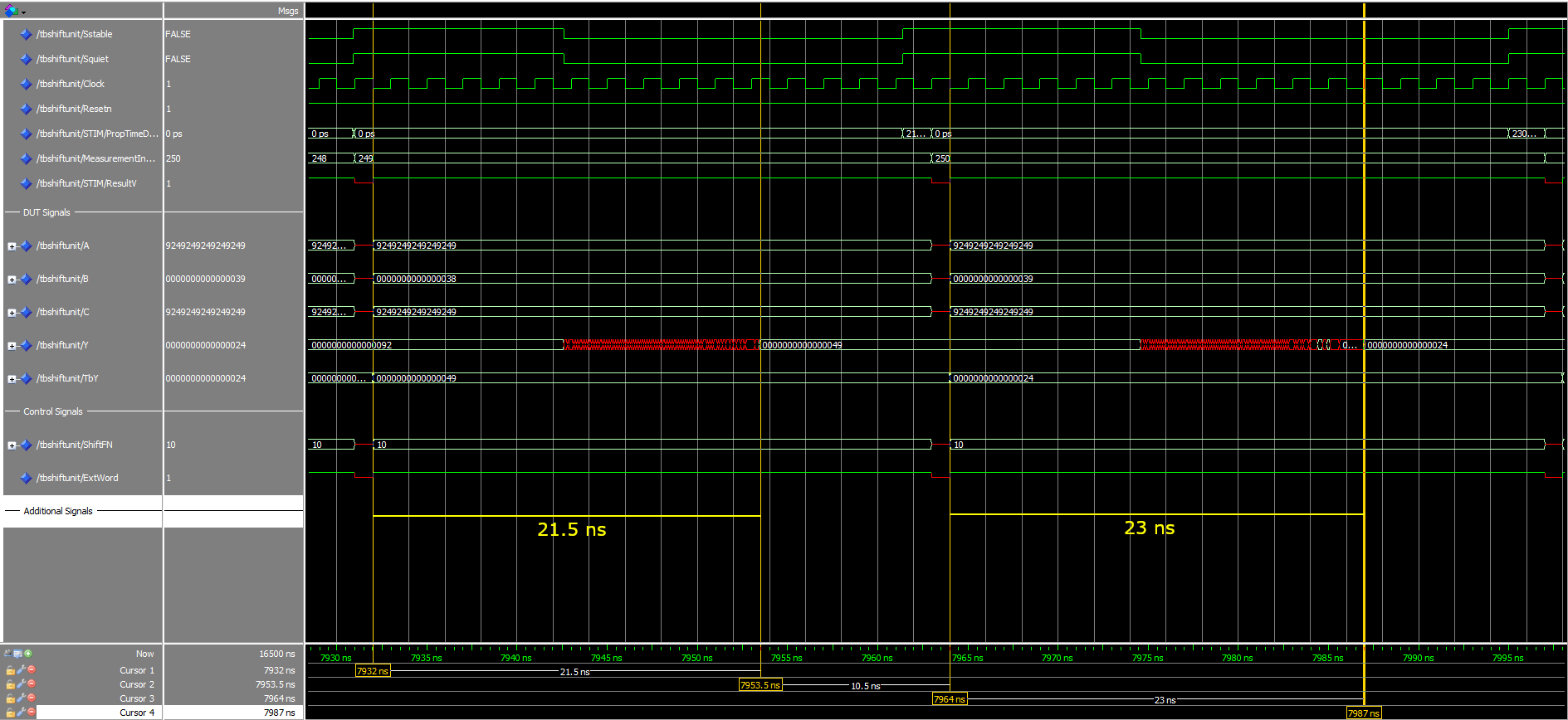


Figure 25: Timing Simulation of SRL32\_1

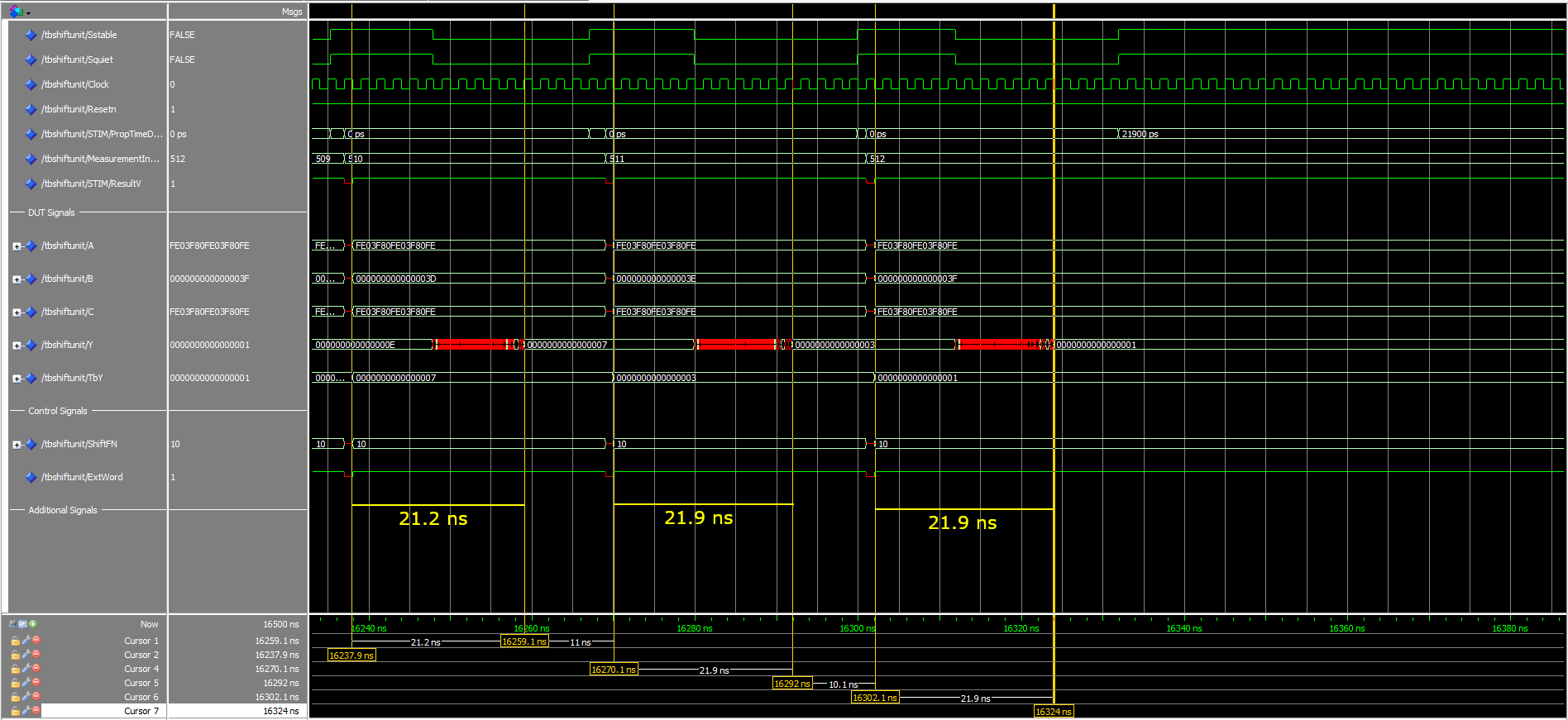


Figure 26: Timing Simulation of SRL32\_2

From Figure ###, we see that the propagation delay for test case 250 is actually significantly higher than that of our previous measurements. This is due to a combination of all the factors: the requirement to word swap due to it being a 32-bit right shift, the compatibility delay mentioned previously, as well as the fact that the shift is 0x39 = 0b111001, which means that all three barrel shifter MUX’s have an input that is not “00”, which requires slightly more time than if it was to be unchanged.

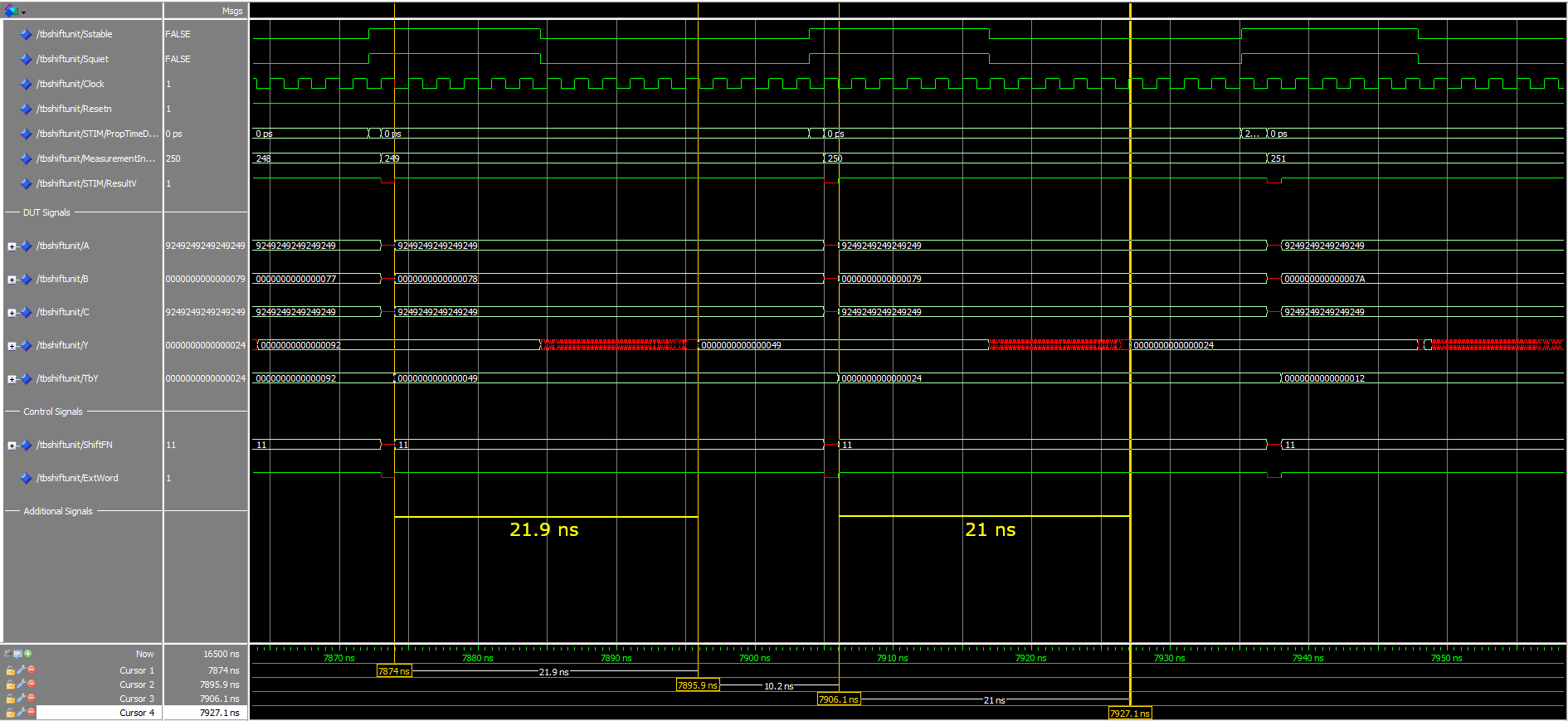


Figure 27: Timing Simulation of SRA32\_1

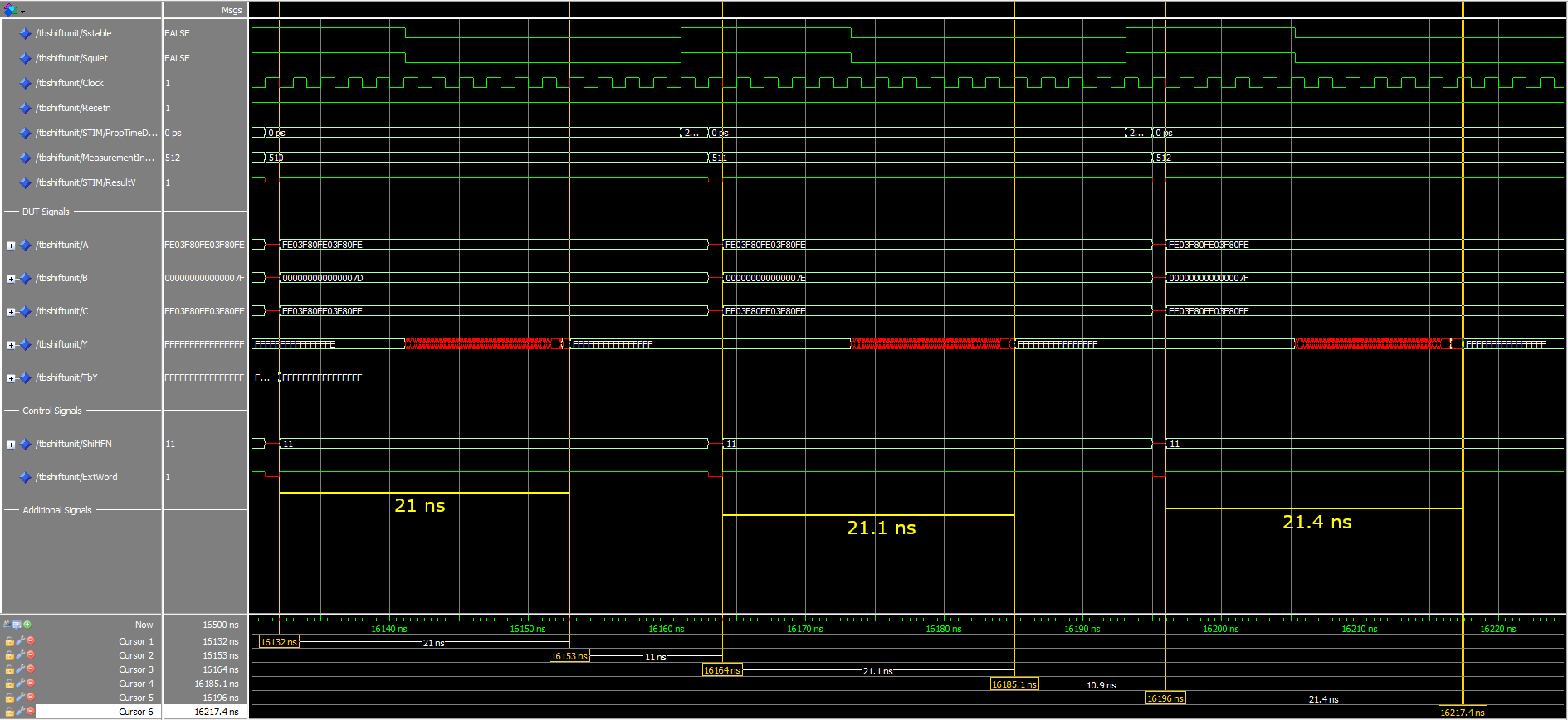


Figure 28: Timing Simulation of SRA32\_2

The final test vector is for 32-bit arithmetic shift right.Once again, the average propagation delay is around the same as the other 32-bit operations. We can observe that in test cases 251, 510, and 512, there is a short time frame where the result stabilizes. The computation of the result is still happening - the short portion of stability is likely a intermediate step before calculating the final result. It is important in timing simulations like these to ensure that the result remains stable for a set amount of time before concluding that the computation is complete. For these timing simulations, we waited for 8 ns of stable output before concluding the tests.

# Execution Unit

Execution Unit is a combination of Logic Unit, Arithmetic Unit and Shift Unit. All these circuits are linked together and based on the control signal ‘FuncClass’, appropriate results are taken as output. Execution unit circuit diagram is shown in figure #### is implemented.

## Implementation

Based on the control signals for each circuit, all the computations are calculated and then passed through a Final MUX with control signal ‘FuncClass’ to select the appropriate output. The VHDL interface for execution Unit is shown in figure ####.

## Functional Simulation

Functional simulation for an execution unit can be executed by running the scripts “FunctionalExecUnit.do” and “waveExecUnit.do”. The test vector that is used for this simulation is “ExecUnit00.tvs”

Image 1 <<waveFXU6 >>

The image in figure (above image) shows two intermediate measurements (72 & 73) of the wave after running the scripts. It also lists the propagation delay for each measurement which is approximately the same, in this case.

Image 2 << waveFXU3>> showing last 3 measurements

The image in figure (above image) shows the last 3 measurements (126, 127 & 128) of the wave after running the scripts. It also lists the propagation delay for each measurement which is approximately 32 ns, in this case.

## Synthesis

The ExecUnit circuit implemented in above sections is now synthesized in Quartus Prime and viewed in netlist viewers for visualization. As shown in the image below, the Execution unit is simply a combination of Logic Unit, Arithmetic Unit & Shift Unit together followed by a 4 input MUX to select the appropriate operation which is operated by a control signal. Following images show the zoomed view for the Execution Unit.

<< Add most of the images from directory >>

<< XU\_RTL >>

## Timing Simulation

Functional simulation for an execution unit can be executed by running the scripts “TimingExecUnit.do” and “waveExecUnit.do”. The test vector that is used for this simulation is “ExecUnit00.tvs”

For Timing Run and document ONE test, using “ExecUnit00.tvs”. - LUCKY

<< waveTXU6 >> To be added

The image in figure (above image) shows a timing simulation wave for two intermediate measurements (72 & 73) after running the scripts.

<< waveTXU3 Time wave for last 3 measurements>>

The image in figure (above image) shows a timing simulation wave for the last three measurements (126, 127 & 128), after running the scripts.

# Conclusion