Group #:
G06

Final Project

ENSC 350 1227

Project Submitted in Partial Fulfillment of the
Requirements for Ensc 350
Towards a Bachelor Degree in Engineering Science.

Last Name:	Last Name:	Last Name:
Schellenberg	**Lin**	**Li**
SID: 3 0 1 3 8	**SID:** 3 0 1 3 5	**SID:** 3 0 1 3 4
1 3 5 0	9 5 9 8	2 9 7 4

Table of Contents

Design Entities

ExecUnit (ExecUnit.vhd)

The **ExecUnit** is the top entity of the project that contains three smaller entities: **ArithUnit, LogicUnit** and **ShiftUnit**. These three smaller entities are all instantiated inside **ExecUnit**'s architecture **execBehaviour**. **ExecUnit** takes A and B, both 64-bit input operands, to produce Y, a 64-bit output. The operation performed by the **ExecUnit** is determined by the control signals **AddnSub, ShiftFN & ExtWord,** and **LogicFN**, which are also input signals to the entities **ArithUnit**, **ShiftUnit** and **LogicUnit** respectively. **Zero** is a status signal that if the **ArithUnit** operation produces an output of '0', would be set to '1'. **AltB** and **AltBu** are both status signals that outputs '1' if A is less than B (for the cases of signed and unsigned operations, respectively). The output of the **ExecUnit** is ultimately decided by a 4-input MUX that takes **FuncClass** as the control signals.

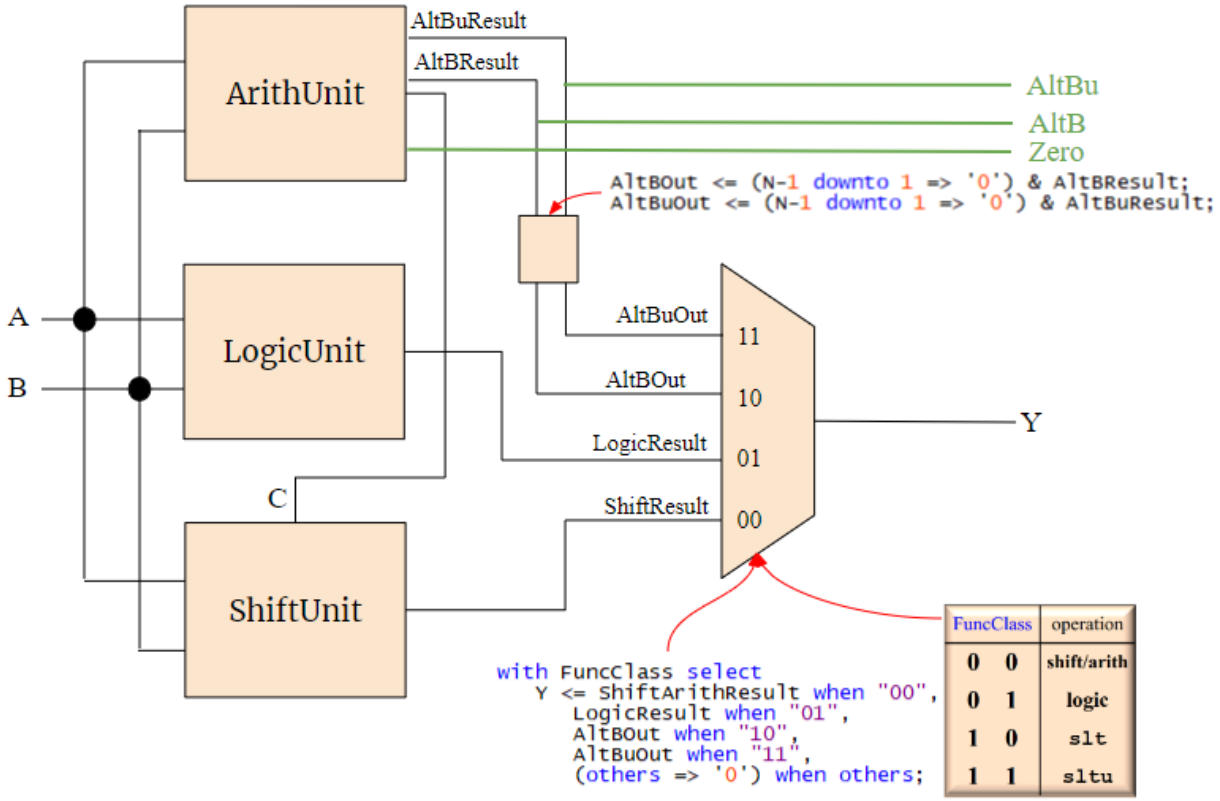


Figure 1: Logic within ExecUnit:

- Diagram shows the internal circuitry overview of ExecUnit
- Table on bottom right shows how FuncClass controls the output of the MUX
- Code Snippet on top right shows VHDL implementation of inserting 63 bits of '0's to the AltBResult and AltBuResult
- Code Snippet on bottom shows VHDL implementation of the MUX logic with 'with-select' syntax

ArithUnit (ArithUnit.vhd)

The **ArithUnit** can perform addition or subtraction of two 64-bit input operands A and B. Before entering the 64-bit adder entity, **ArithUnit** first decides whether the operation is an addition or a subtraction by realizing the select signal **AddnSub** (if **AddnSub** is '1', input B will become the 2's complement of itself). **Zero, AltB** and **AltBu** are calculated and then used as output for the ExecUnit.

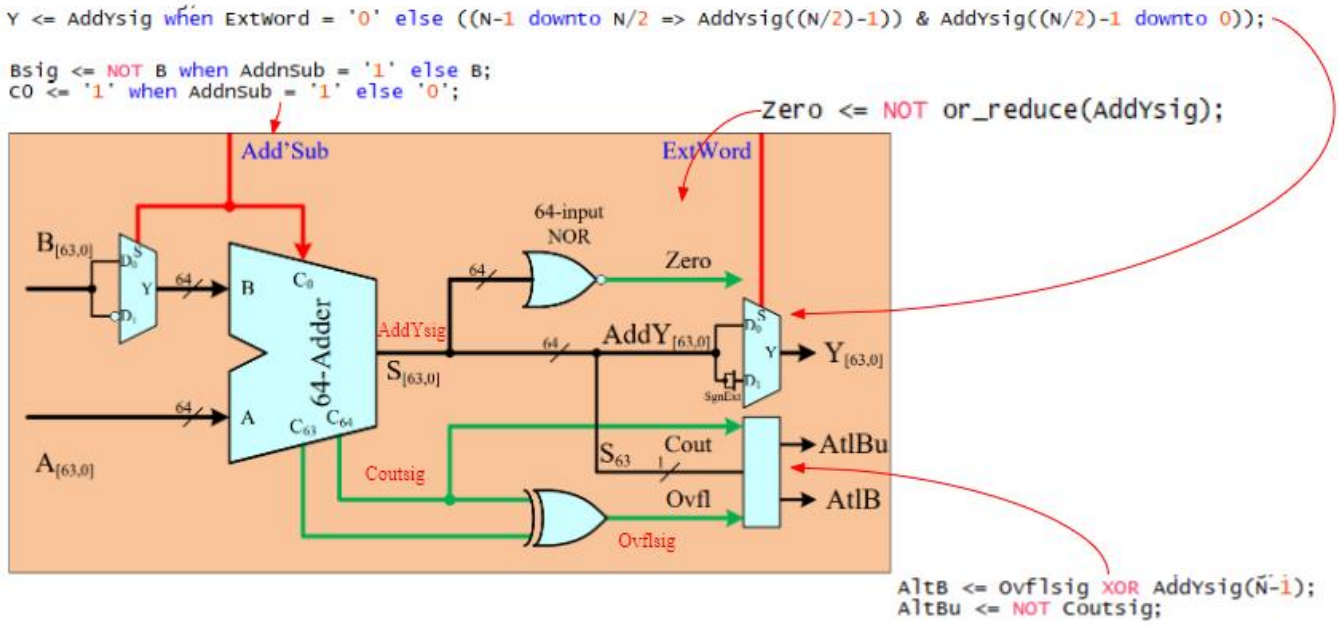


Figure 2: Logic within ArithUnit:

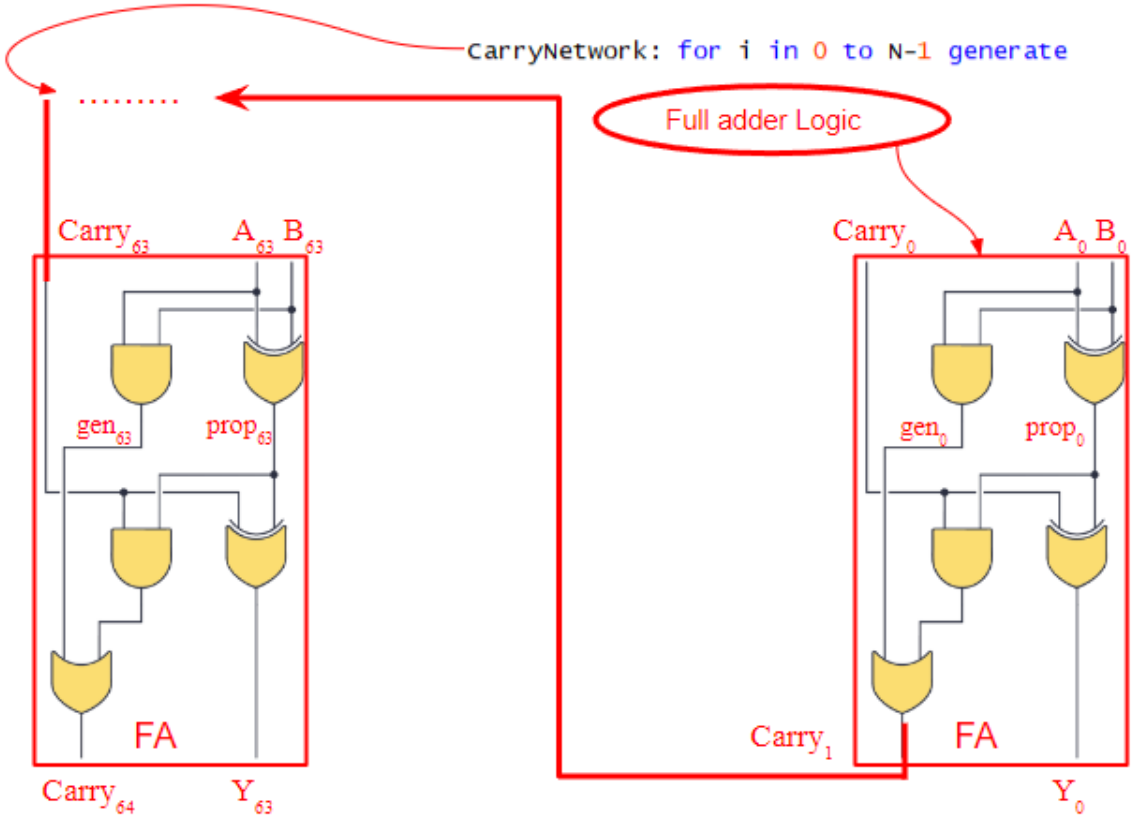
- Diagram shows the internal circuitry of the ArithUnit
- Code Snippet on top left shows the how AddnSub is used as a control signal for the MUX for input B and as carry in bit for the 64-bit adder
- Code Snippet with NOT or_reduce shows the VHDL implementation of a 64-input NOR gate
- Code Snippet on bottom right shows the logics of AltBu and AltB
- Code Snippet on very top shows the VHDL implementation of the Sign Extension MUX

Adder (Adder.vhd)

For the 64-bit adder, we have tried it with two different methods. Method 1 was to use a 64-bit ripple adder, which uses 64 Full Adders, where the carryout bits from the first 63 of them becomes to carry in bits for the next full adder in line, and the very last carryout bit becomes the **Cout.** However, we have realized that using a ripple adder produces way too much unnecessary propagation delay, that we have decided to use Method 2: a carry-select network.

Details of this optimization will be explained further down below.

Ripple Adder (UNUSED)

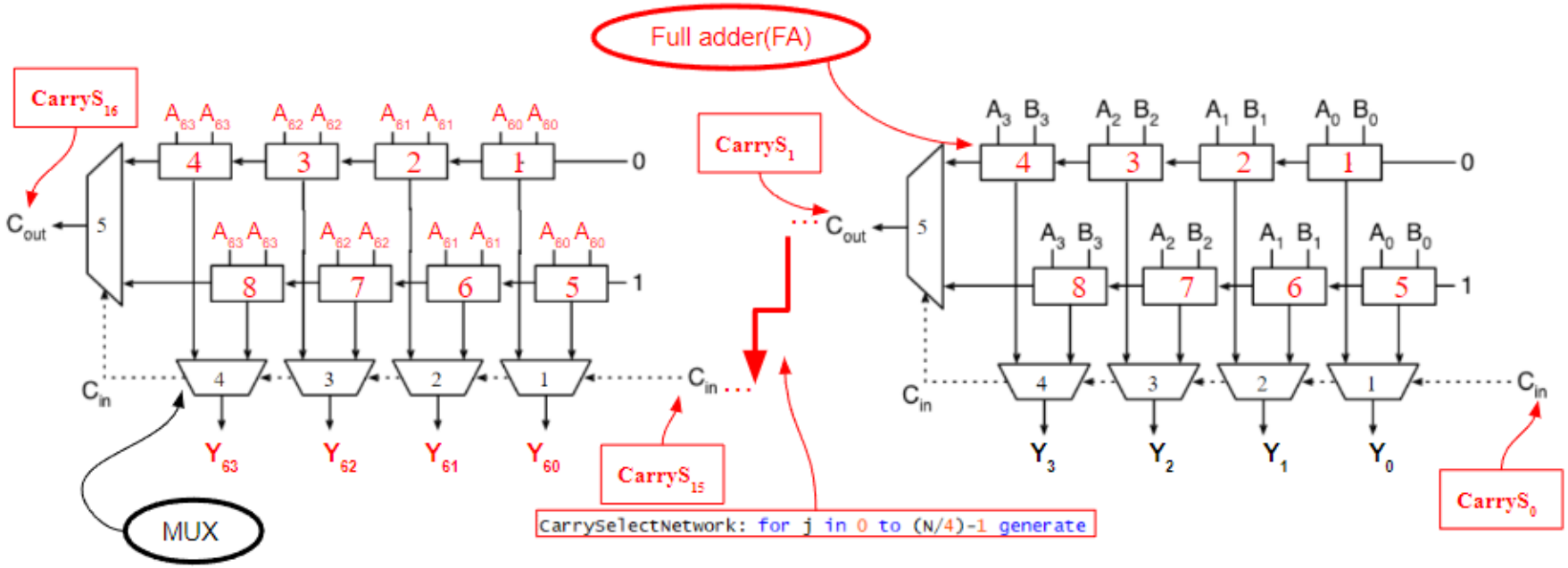



```
ovfl <= carry(N) XOR carry(N-1);
```

Figure 3: Logic within ArithUnit:
- Diagram shows the circuitry overview of the Ripple, which consists of 64 1-bit Full Adders
- Code Snippet on top right shows the VHDL implementation of generating 64 of the same Full Adder logic
- Code Snippet on bottom shows the VHDL implementation of calculating Ovfl

Full Adder Logic Diagram Source: https://www.build-electronic-circuits.com/full-adder/

Carry-Select Adder (Optimized & USED)




```
OvMux: myMux Port Map(c0(N-2),c1(N-2),carryS((N/4)-1), carryNMinus1);
Ovfl <= carryS(N/4) XOR carryNMinus1;
```

Figure 4: Design of CarrySelectNetwork:

- Diagram shows the circuitry overview of the CarrySelectNetwork, which consists of 16 (64/4) 4-bit carry-select adders
- The Full Adders and MUXes are all instantiated as components
- Code Snippet at centre bottom shows the VHDL implementation of generating 16 of same port mapping patterns
- Code Snippet at the very bottom shows the VHDL implementation of fetching the second last CarryS by using an extra MUX OvMux in order to calculate Ovfl

4-bit Carry-Select Adder Diagram Source: https://vhdlguru.blogspot.com/2015/04/vhdl-code-for-carry-select-adder.html

LogicUnit (LogicUnit.vhd)

The **LogicUnit** is only used when **ExecUnit** select signal **FuncClass** is '01' ('Logic'), meaning a logic operation is to be performed. The **LogicUnit** consists of a 4-input MUX, where input 0 passes B straight through and input 1 to 3 performed XOR, OR and AND operation of A and B respectively.

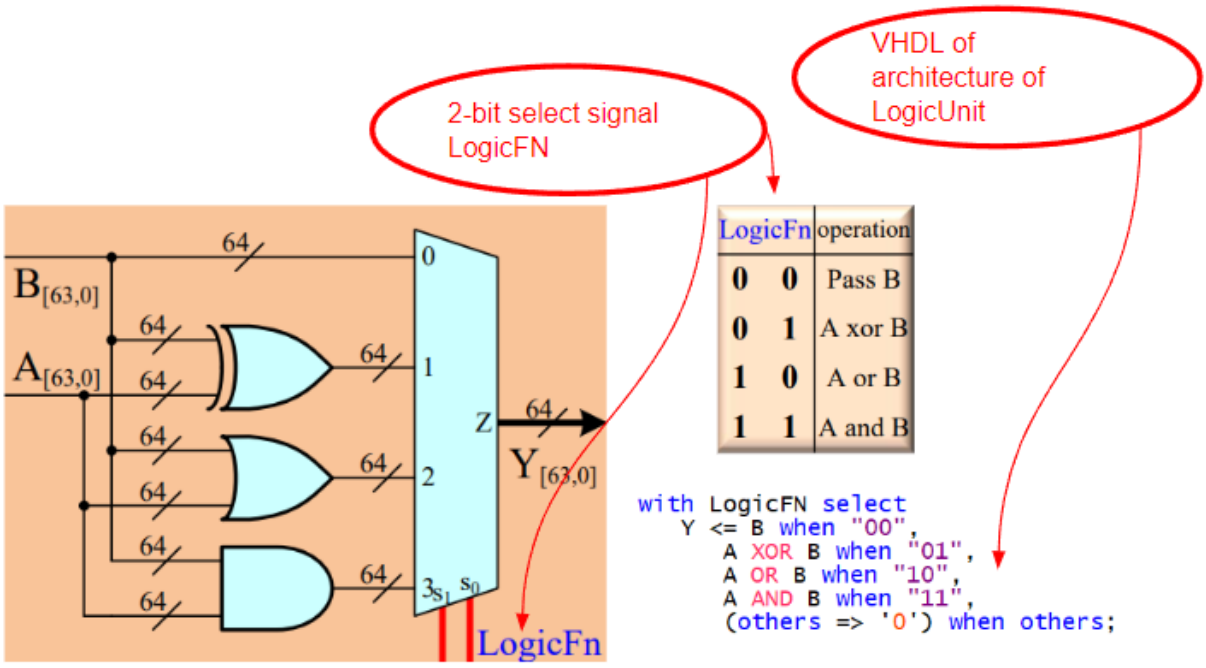


Figure 5: Logic within LogicUnit:

- Diagram on the left shows the internal circuitry of LogicUnit
- Table on top right shows operation of the MUX with its corresponding select signals
- Code snippet on bottom right shows how the logic of LogicUnit is implemented in VHDL

ShiftUnit (ShiftUnit.vhd)

The **ShiftUnit** has 64-bit input operands A, B and C (C is the output of the **ArithUnit**, meaning if **ShiftFN** is '00', no shifts are performed). The **ShiftUnit** uses input operand B for **ShiftCount**, which determines the number of shifts of input operand A. **ShiftFN** and **Extword** then are used as the select signal for a series of MUXes to determine which of the three shift results are needed and if Sign Extension is needed (for 32-bit operations).

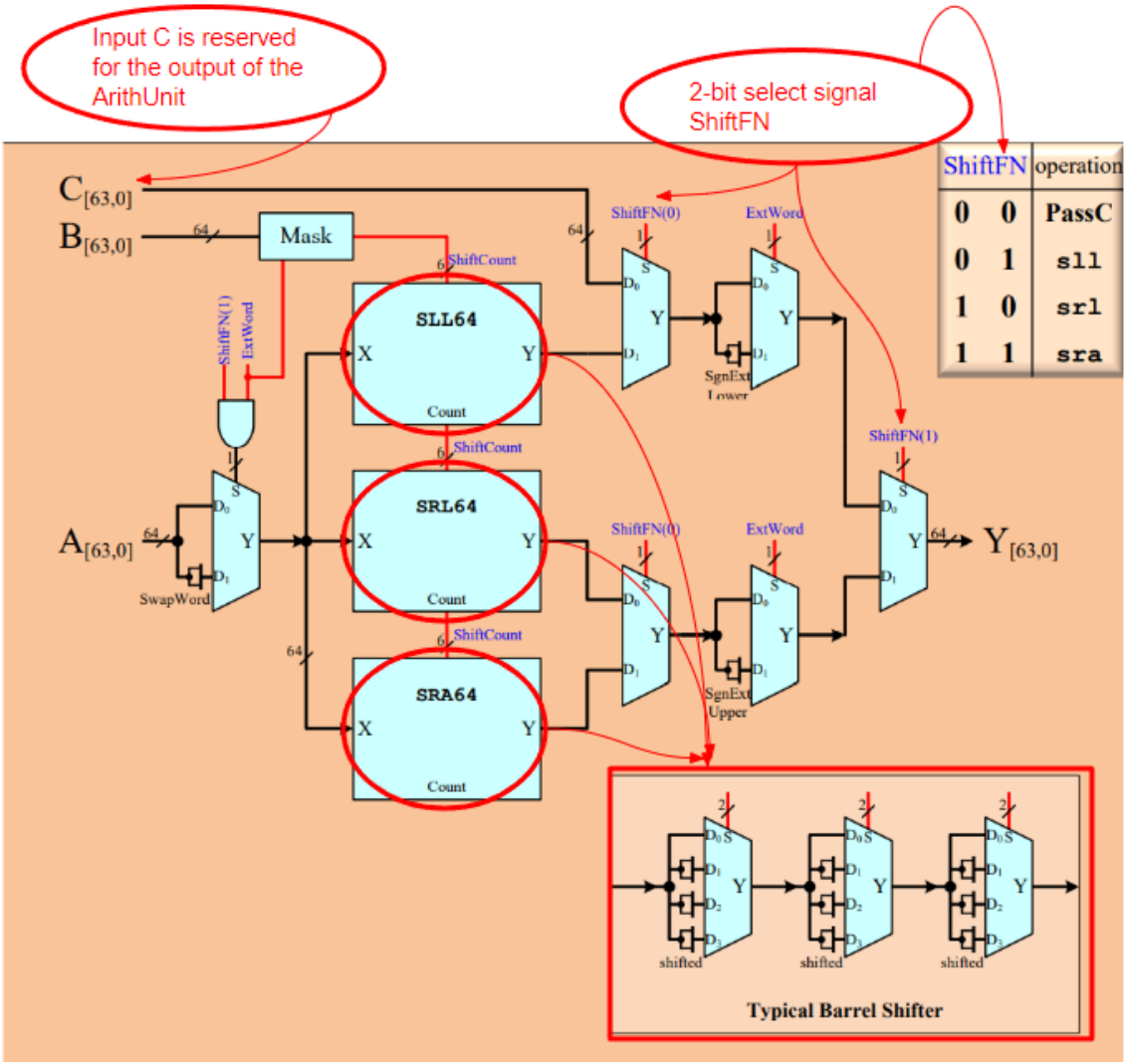


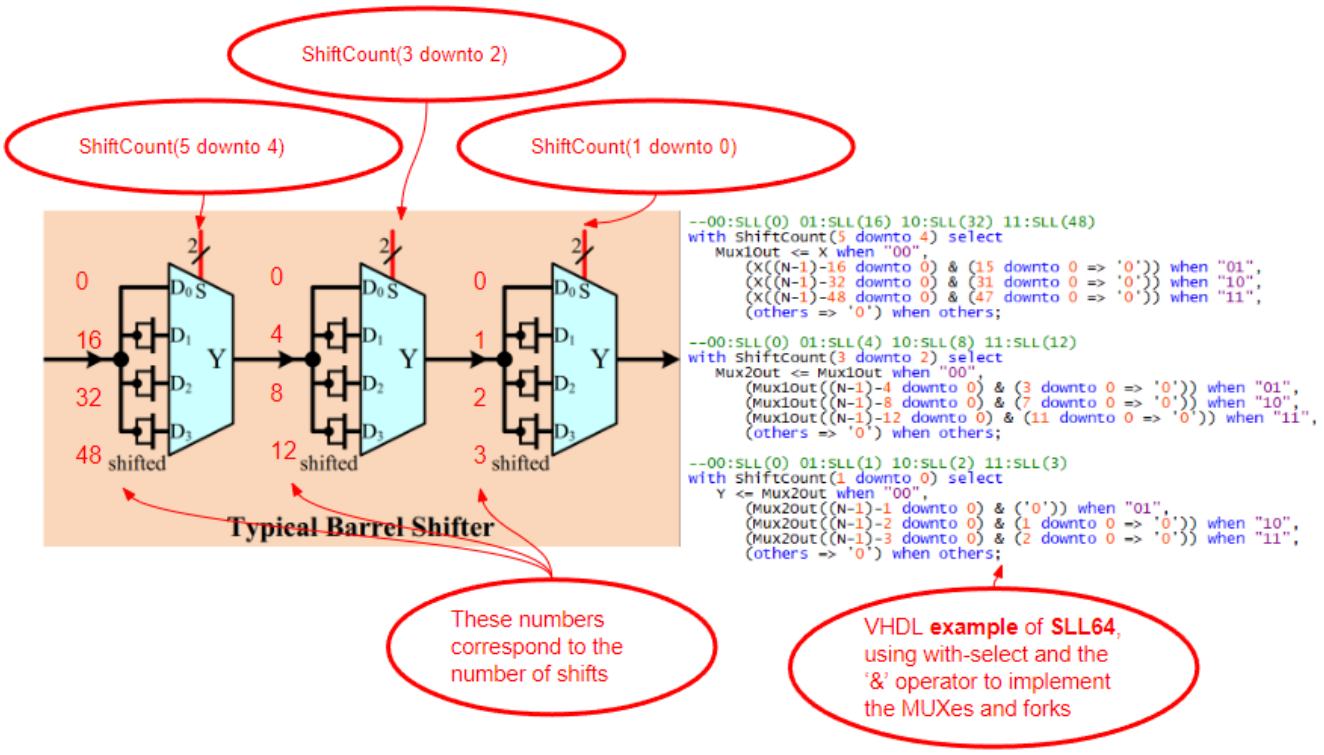
Figure 6: Logic within ShiftUnit:

- Diagram on top left shows the internal circuitry of ShiftUnit
- Table on top right shows operation of the MUXes with its corresponding select signals
- Diagram on bottom right shows how SLL64, SRL64 and SRA64 all consist of typical barrel shifters.

Barrel Shifter (SLL64.vhd, SRL64.vhd, SRA64.vhd)

SLL64, **SRL64** and **SRA64** all consist of 3-stage barrel shifters. The 6-bit signal **ShiftCount** is divided into three 2-bit signals, which are select signals for the three MUXes as shown in the figure below. The number of shifts is determined by the sum of the number of shifts from each MUX.

For example, if **ShiftCount** is '111111', meaning a shift number of 63. 63 would then come from summing up the shift number of input '11' on all three MUXes: 48+12+3 = 64.




```
--00:SLL(0) 01:SLL(16) 10:SLL(32) 11:SLL(48)
with ShiftCount(5 downto 4) select
   Mux1Out <= X when "00",
       (X((N-1)-16 downto 0) & (15 downto 0 => '0')) when "01",
       (X((N-1)-32 downto 0) & (31 downto 0 => '0')) when "10",
       (X((N-1)-48 downto 0) & (47 downto 0 => '0')) when "11",
       (others => '0') when others;

--00:SLL(0) 01:SLL(4) 10:SLL(8) 11:SLL(12)
with ShiftCount(3 downto 2) select
   Mux2Out <= Mux1Out when "00",
       (Mux1Out((N-1)-4 downto 0) & (3 downto 0 => '0')) when "01",
       (Mux1Out((N-1)-8 downto 0) & (7 downto 0 => '0')) when "10",
       (Mux1Out((N-1)-12 downto 0) & (11 downto 0 => '0')) when "11",
       (others => '0') when others;

--00:SLL(0) 01:SLL(1) 10:SLL(2) 11:SLL(3)
with ShiftCount(1 downto 0) select
   Y <= Mux2Out when "00",
       (Mux2Out((N-1)-1 downto 0) & ('0')) when "01",
       (Mux2Out((N-1)-2 downto 0) & (1 downto 0 => '0')) when "10",
       (Mux2Out((N-1)-3 downto 0) & (2 downto 0 => '0')) when "11",
       (others => '0') when others;
```

Figure 7: Logic of the barrel shifters:
- Diagram on the left shows the internal circuitry of ShiftUnit
- Code Snippet on the right shows the VHDL implementation example of SLL64. Same code layout is used for SRL64 and SRA64.

Netlist Views

ExecUnit

The RTL and Post-Fit circuit look identical to the diagram drawn in the design entity section, confirming that our design is correctly done. In this Cyclone IV Device, 1592 logic elements (1%) and 203 pins (38%) are used for this circuit. No memory elements are observed, further proving that this is a purely combinational circuit.

RTL

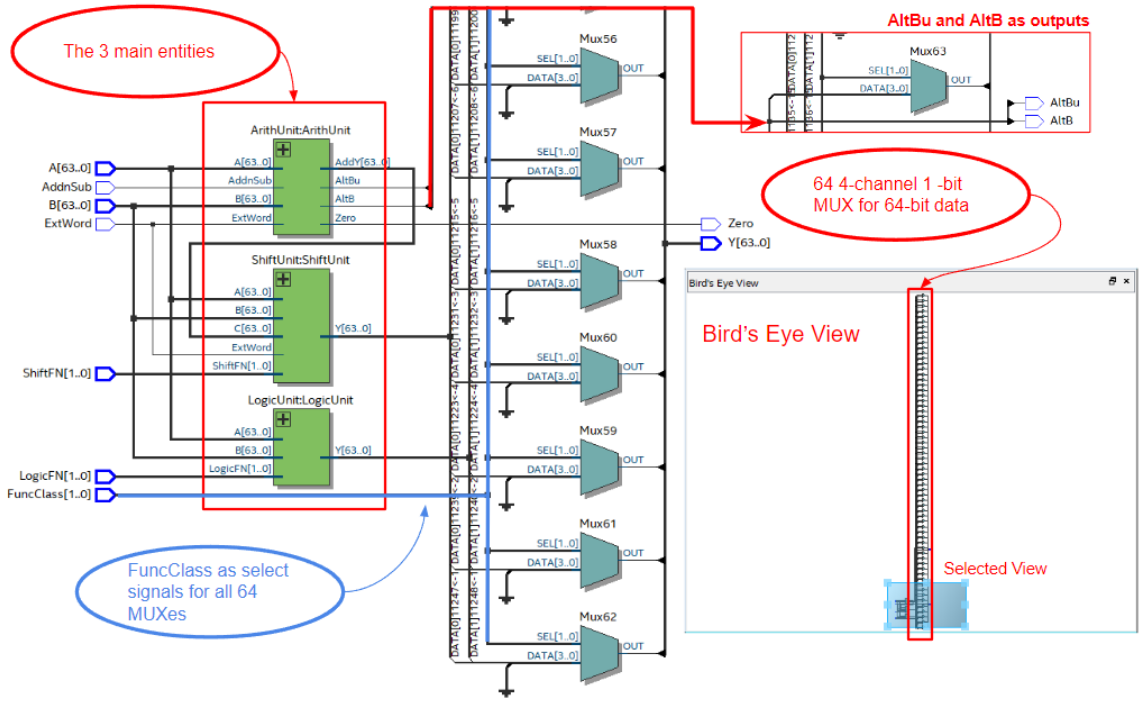


Figure 8: RTL View of ExecUnit:

- Bird's Eye View shows the internal circuitry overview of ExecUnit
- Selected View zooms in into the placement of the three main entities ArithUnit, ShiftUnit and LogicUnit, as described in the design entity section
- Separated view window on the top right shows where AltBu and AltB goes

Post-Fitting

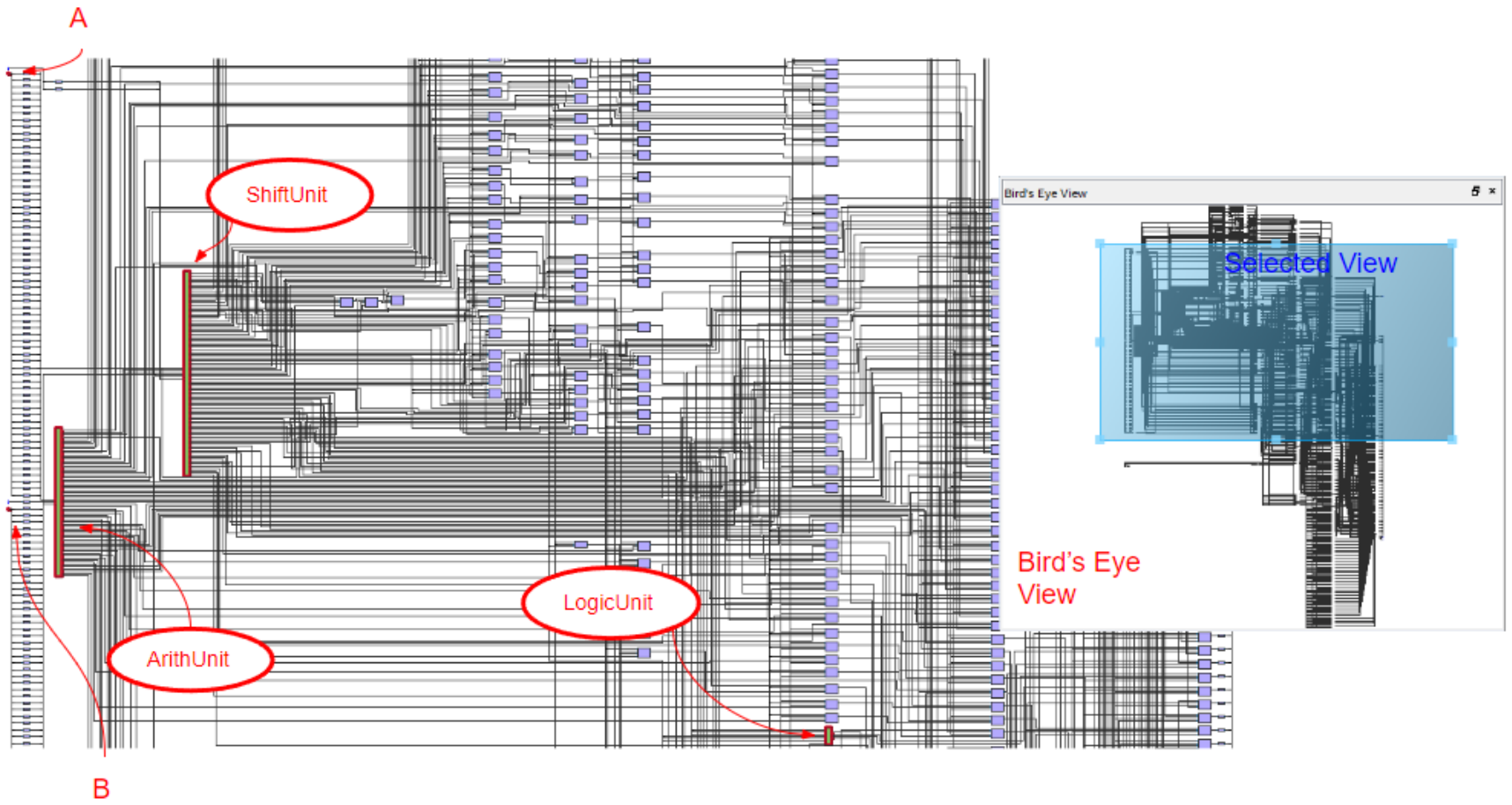


Figure 9: Post-Fit Circuit of ExecUnit:

- Bird's Eye View shows the internal Post-Fit circuitry overview of ExecUnit
- Selected View zooms in into the placement of the three main entities ArithUnit, ShiftUnit and LogicUnit, as described in the design entity section
- Input Operands A and B can be observed above on the left with the arrow indicators

ArithUnit

385 out of 1592 Logic Elements are used in **ArithUnit**, and as observed in the Post-Fit Circuit in the following pages, most of the logic elements are used in implementing the 64-input NOR gate as well as the AddInst. However, many of the logic elements in AddInst are in parallel therefore significantly decreased the propagation delay.

RTL

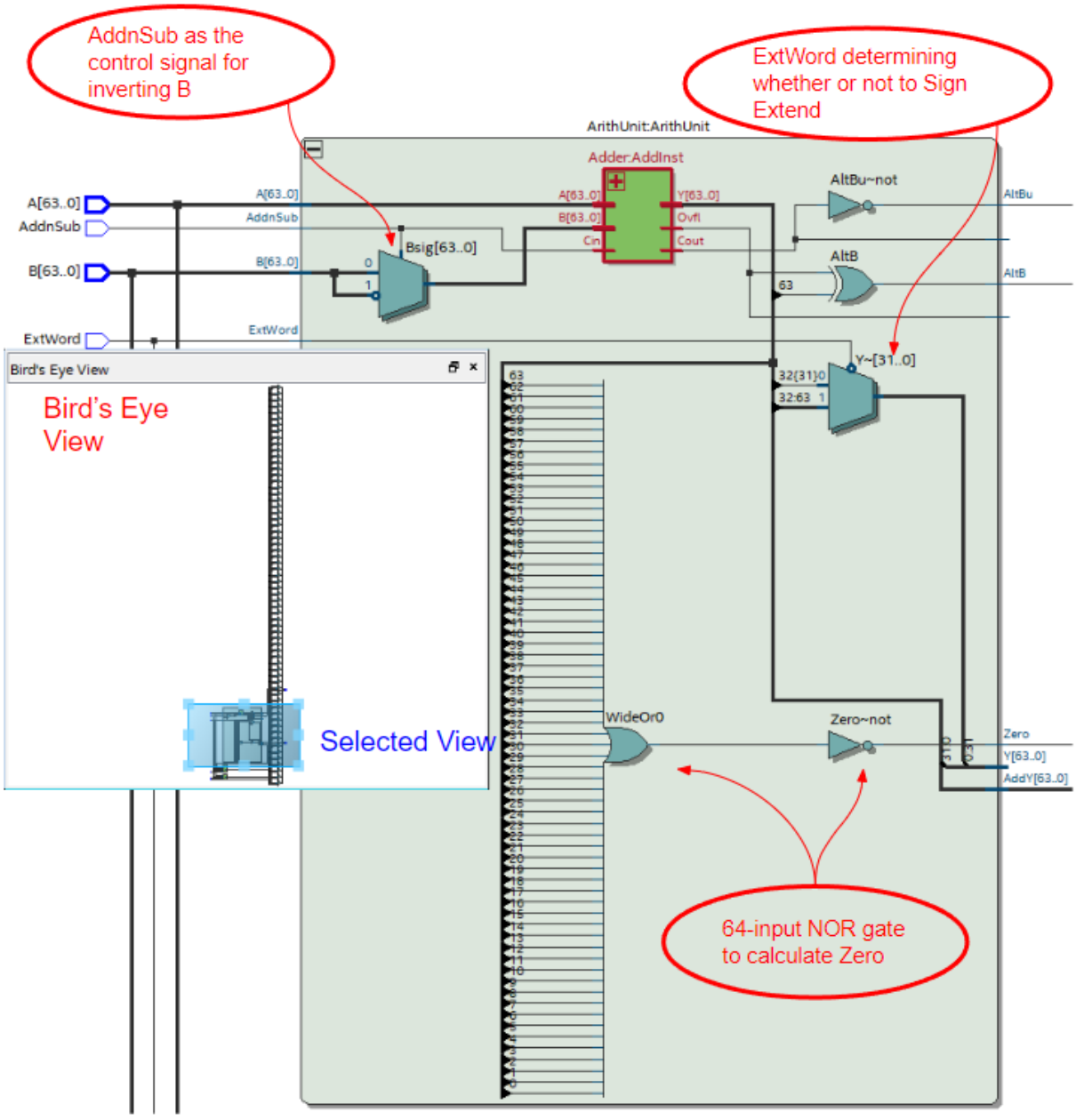


Figure 10: RTL View of ArithUnit:
- Bird's Eye View shows the relative placement and size of the ArithUnit within the ExecUnit
- Selected View shows the internal circuitry of the ArithUnit, which matches identically to the diagram drawn in the design entity section, proving our design to be correct

RTL Cont. (AddInst)

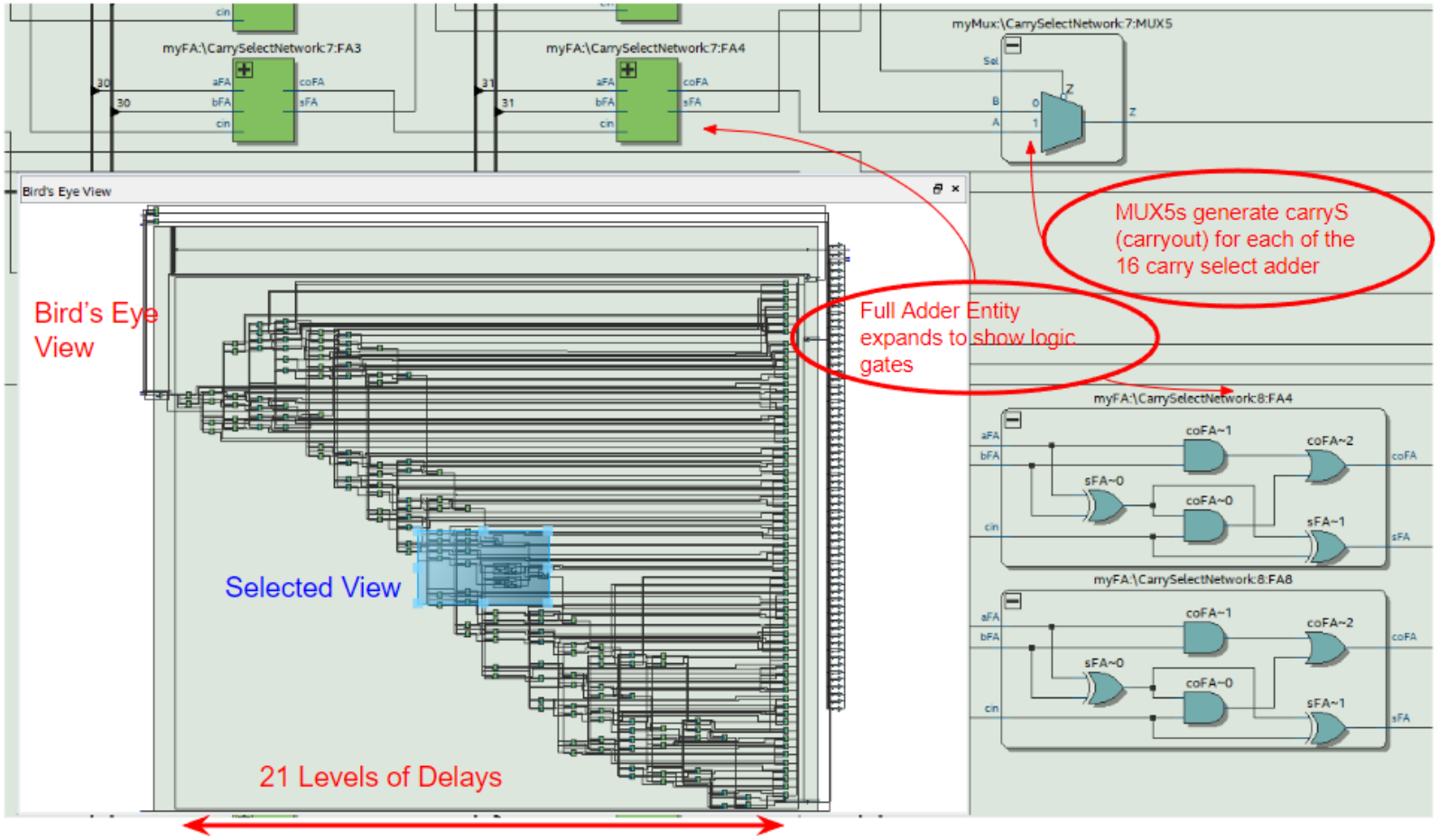


Figure 11: RTL View of AddInst within ArithUnit:

- Bird's Eye View shows the overall structure of the carry-select network
- Selected View zooms in into the subentities Full Adders and MUX, where they expand to show the corresponding logic
- 21 levels of delay can be observed by counting the columns within the AddInst

Post-Fitting (Carry-Select Network, Optimized and USED)

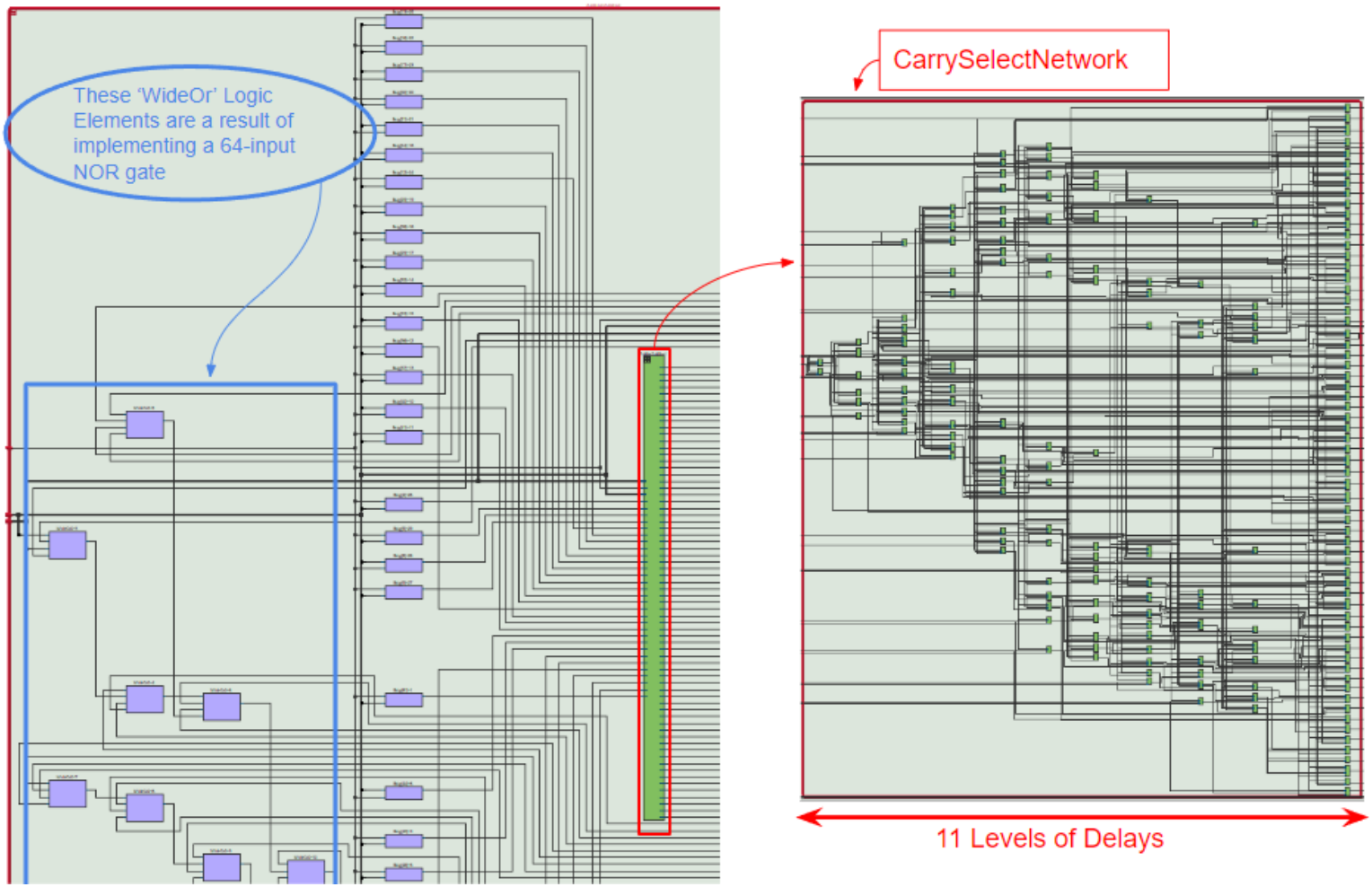


Figure 12: Post-Fit View of ArithUnit and CarrySelectNetwork within ArithUnit:
- Diagram on the left shows how the 64-input NOR gate has used a significant number of logic elements
- Diagram on the right shows the internal circuitry of AddInst, which is the CarrySelectNetwork
- 11 levels of delay can be observed by counting the columns within CarrySelectNetwork

Post-Fitting Cont. (Ripple Carry Network, Unused because of propagation delay)

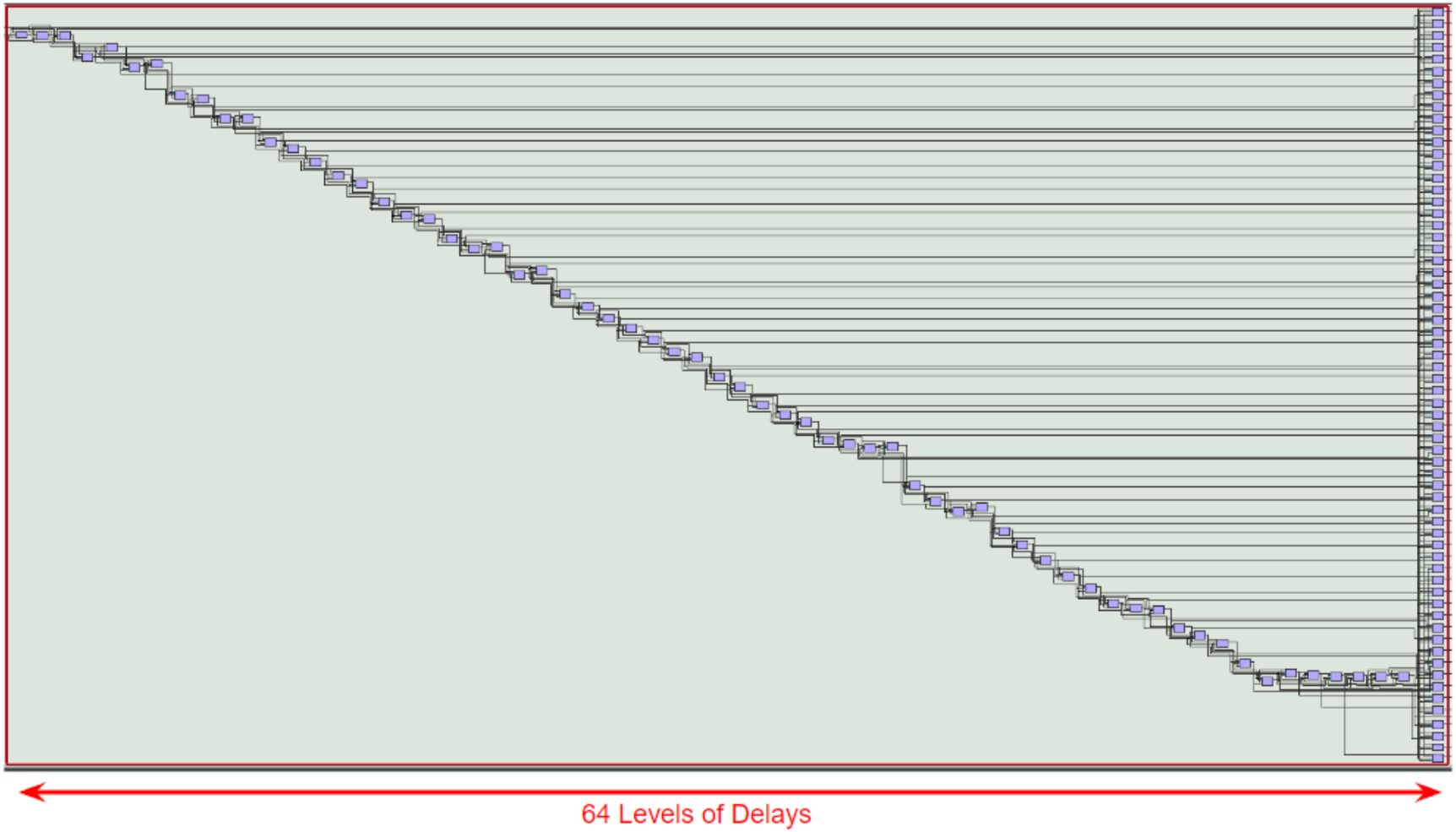


Figure 13: Post-Fit View of Ripple within ArithUnit (UNUSED):
- This figure shows the internal circuitry if the ripple carry network were to be used instead of the carry-select network
- 64 levels of delay can be observed by counting the columns within Ripple
- Although the number of logic elements is less, the propagation delay is significantly more

LogicUnit

63 of 1592 logic elements are used in **LogicUnit**, as this unit contains much simpler logics. And as we observe in the post-fit view in the following page, only four LUTs are used and only one level of delay is observed.

RTL

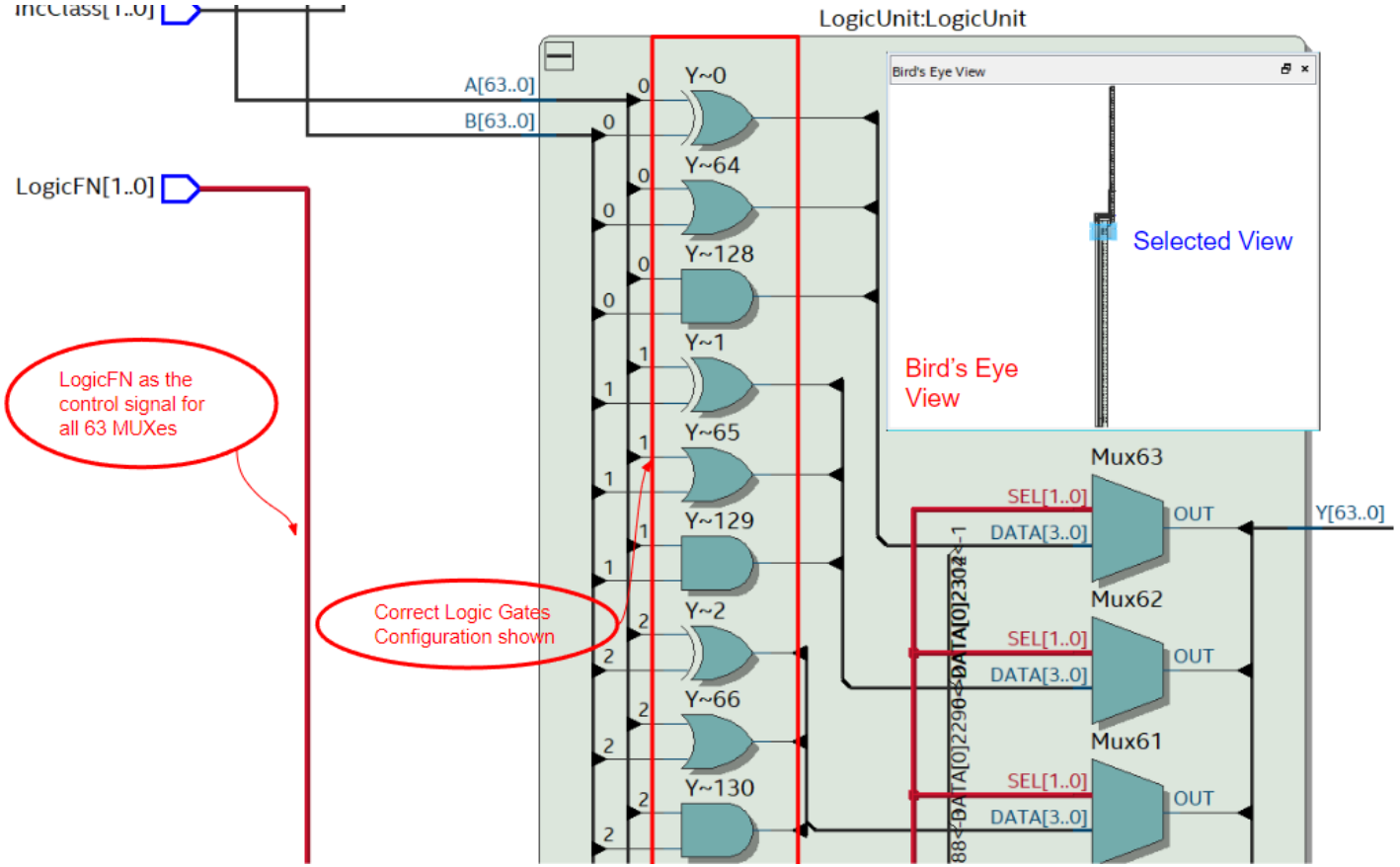


Figure 14: RTL View of LogicUnit:

- Selected View zooms into the first couple of logic gates and MUXes to show the correct logics being implemented and where the control signal (LogicFN) goes to

Post-Fitting

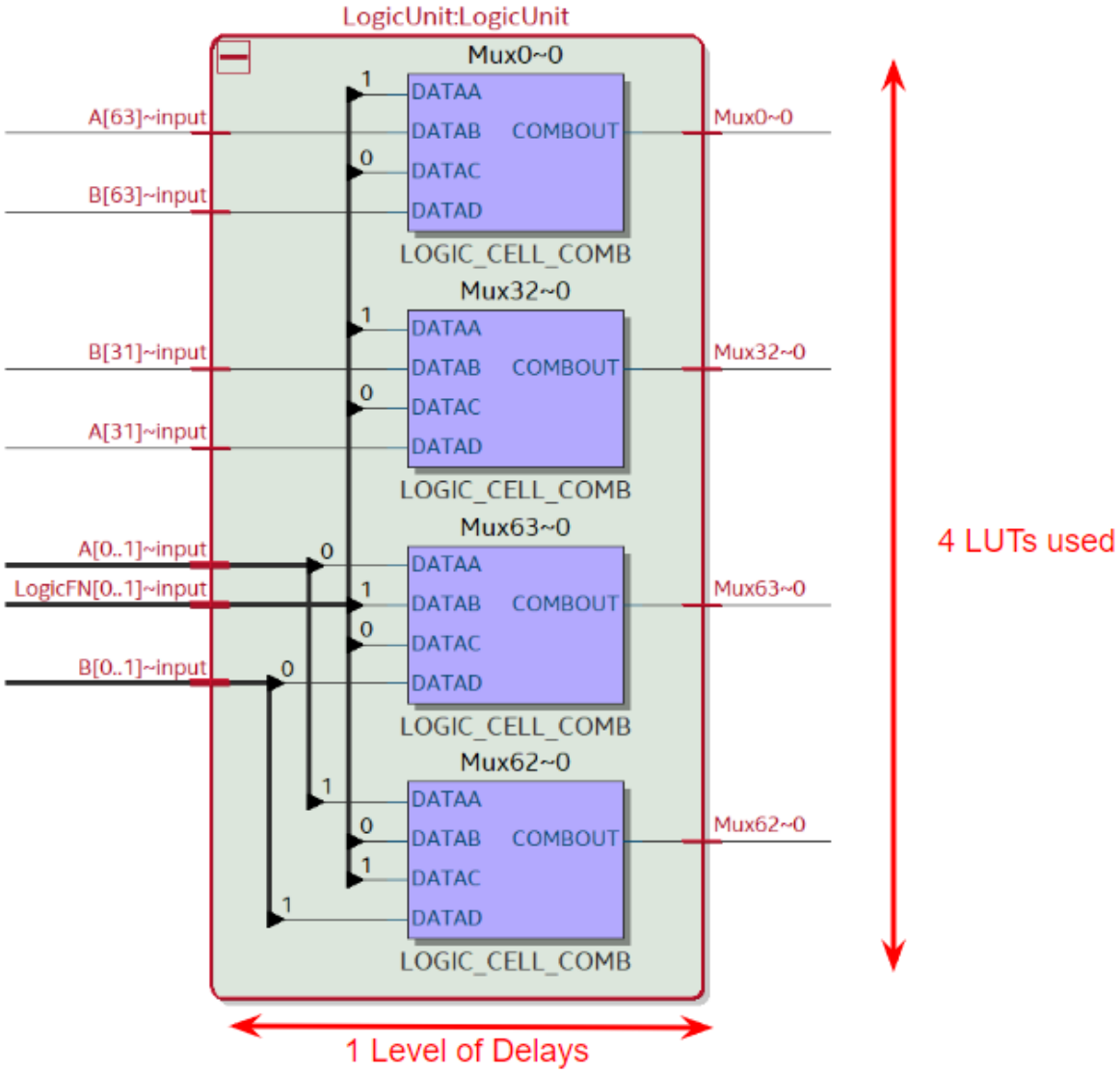


Figure 15: Post-Fit View of LogicUnit:
- Diagram shows the 4 LUTs used and the 1 level of delay observed

ShiftUnit

1142 of 1592 logic elements are used in **ShiftUnit**, making it the unit with the most logic elements used. This is because the three sub-entities, **SLL64**, **SRL64** and **SRA64** all have three stages of MUXes, and generating all these logics require a significantly large number of logic elements. However, these logic elements are mostly in parallel, making little effect to the propagation delay.

RTL

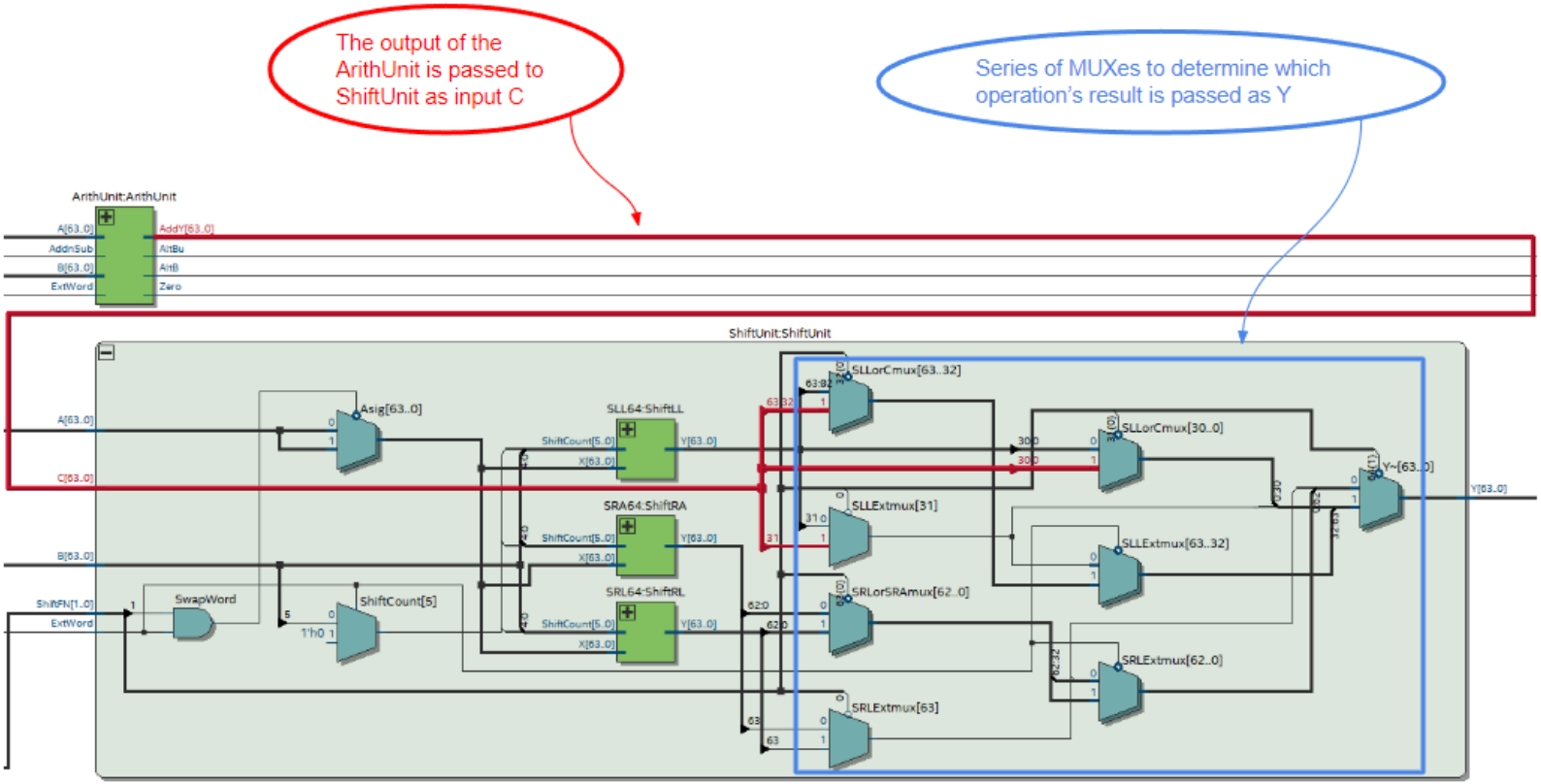


Figure 16: RTL View of ShiftUnit:

- Diagram shows the internal circuitry of ShiftUnit, which match identically with the diagram drawn in the design entity section, proving our implementation to be correct.
- Red line shows that the output of ArithUnit is passed to the ShiftUnit as Input Operand C, same as shown in the design entity section

RTL Cont. (SLL64 Example)

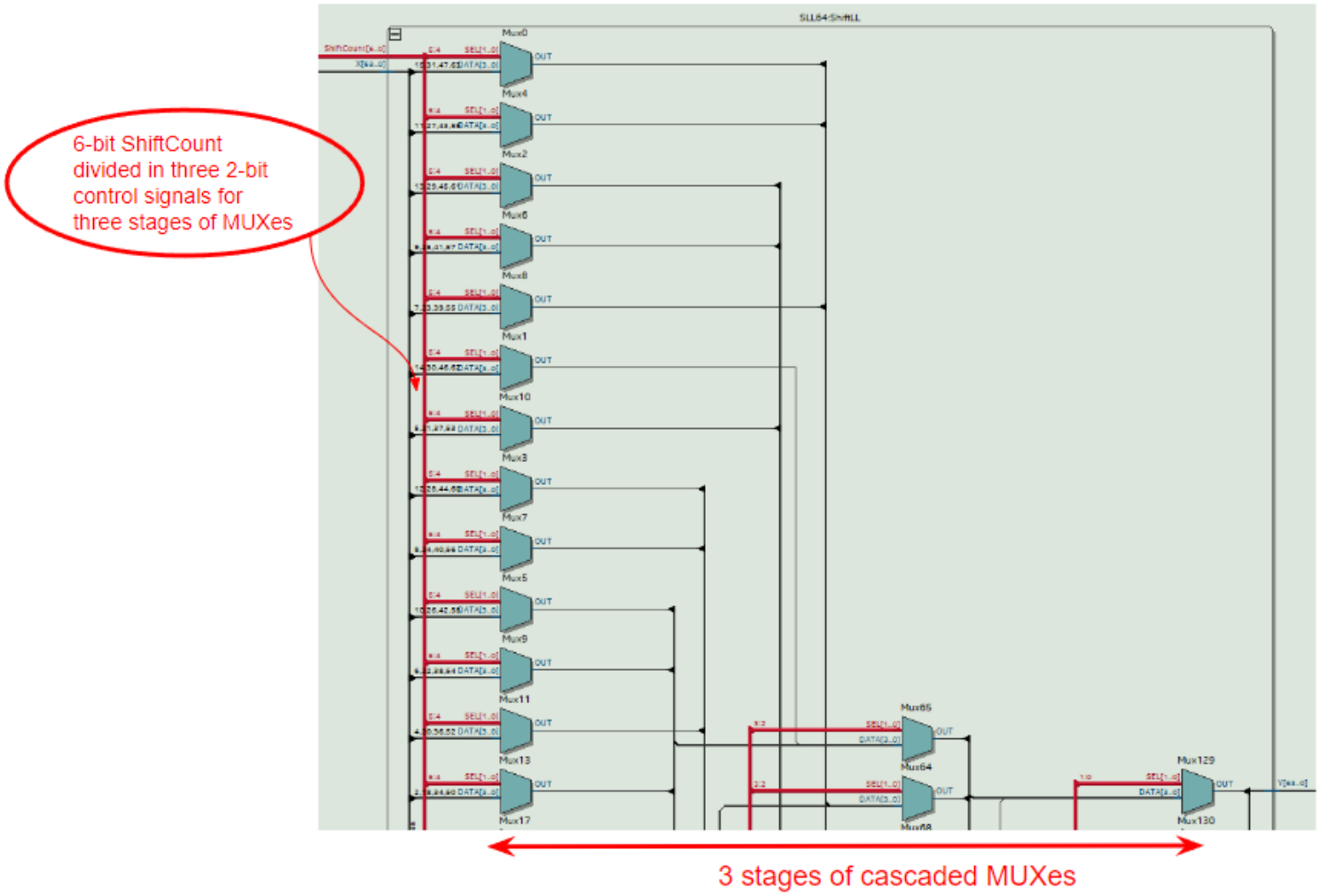


Figure 17: RTL View of SLL64 (Example of Barrel Shifters):

- Diagram shows the three stages of MUXes, same as shown in the design entity section, proving our implementation to be correct
- Red Lines show how the ShiftCount acts as a control signal for the barrel shifter
- Same Circuit Patterns can be found SRL64 and SRA64 as well

Post-Fitting

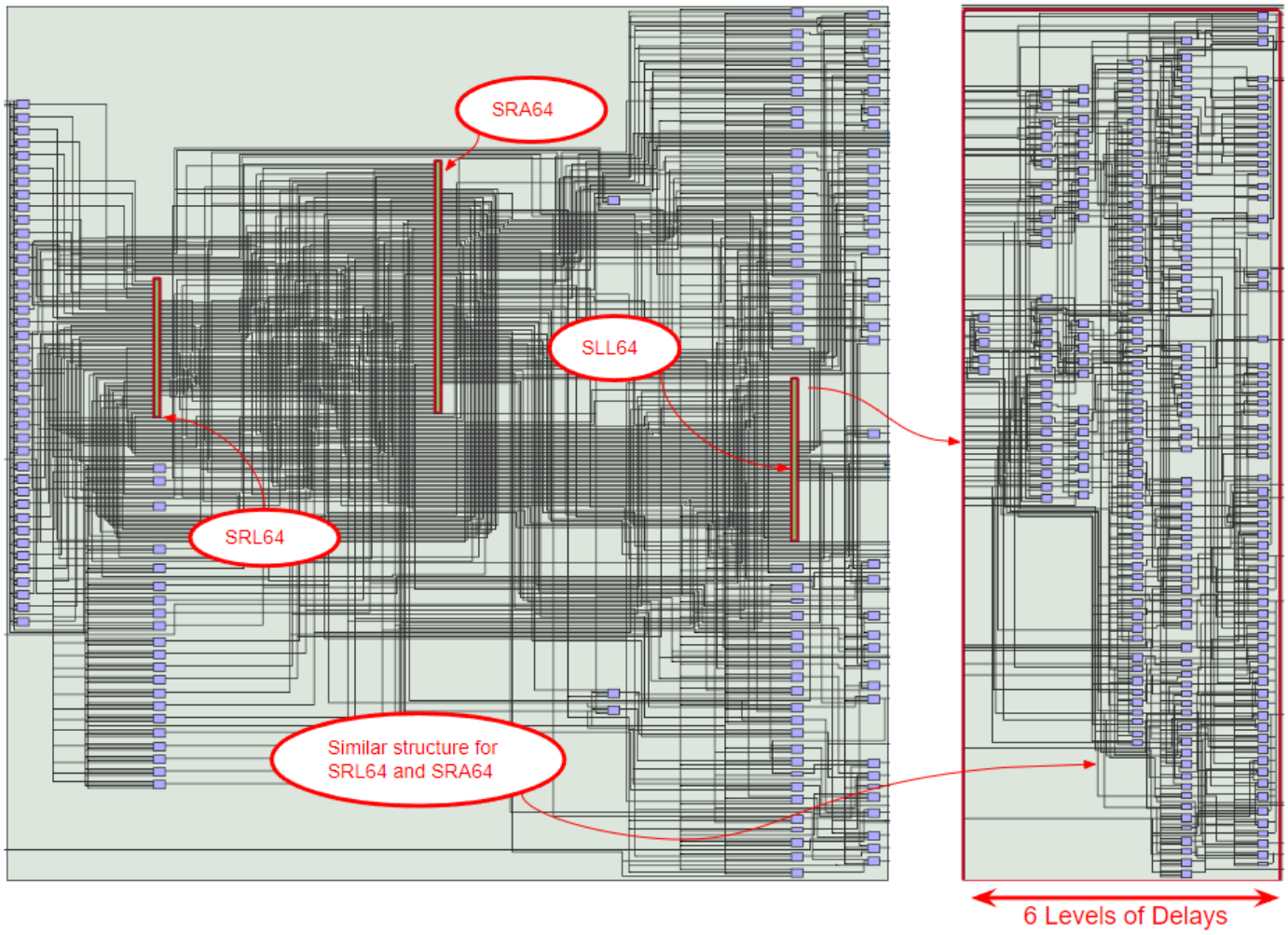


Figure 18: Post-Fit View of ShiftUnit and SLL64:

- Diagram on the left shows the post-fit circuit of the ShiftUnit with the three subunits highlighted and marked
- Diagram on the right shows the internal post-fit circuitry of the SLL64 barrel shifter with 6 levels of delay observed
- The significant amount of logic elements can be easily observed as SRL64 and SRA64 contain similar structure as SLL64

ExecUnit Simulations

Testbench

Block Diagram

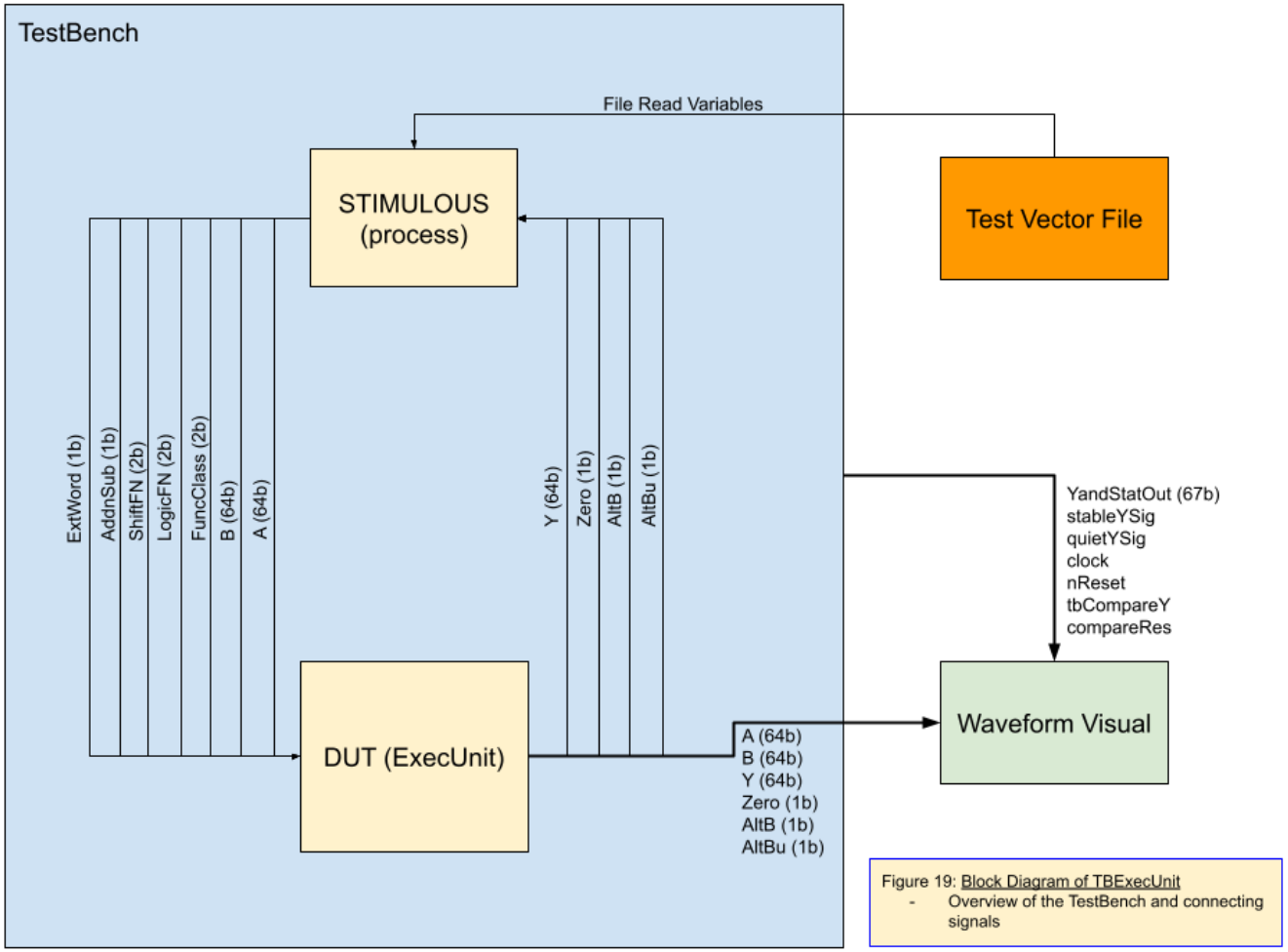


Figure 19: Block Diagram of TBExecUnit
- Overview of the TestBench and connecting signals

Flow Chart

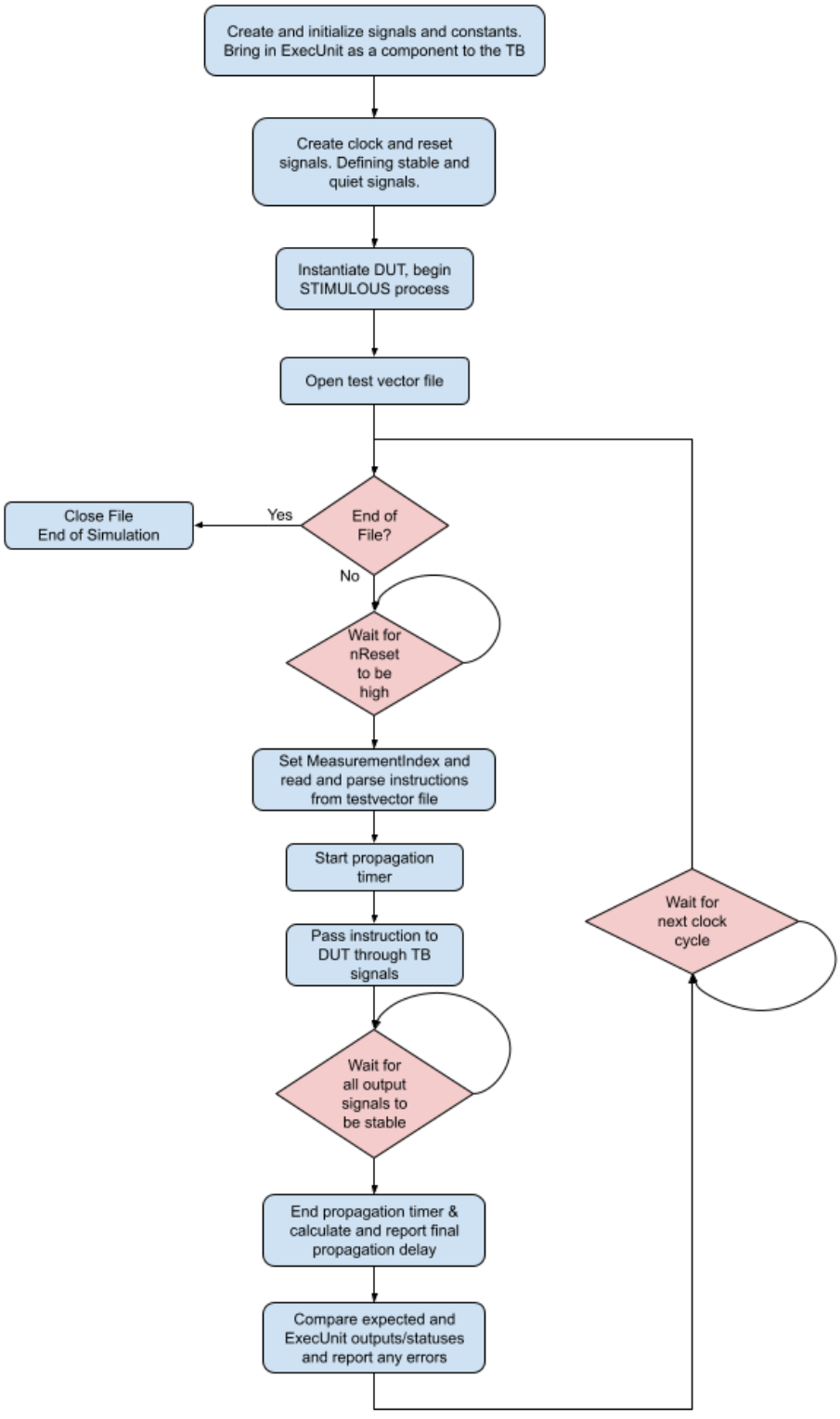


Figure 20: Flow Chart of TBExecUnit
- Overview of the testbench flow

Functional Simulation

Note that all simulation waveforms are done with the original 00 testvector fileset. The 01 fileset was used after release and found to function 100% as well

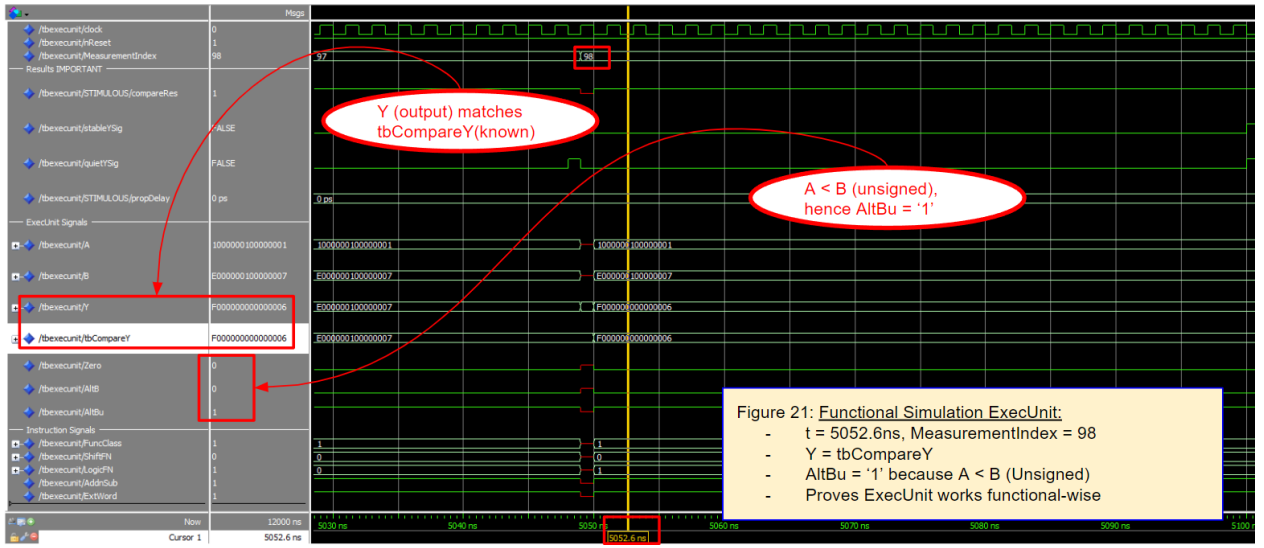


Figure 21: Functional Simulation ExecUnit:
- t = 5052.6ns, MeasurementIndex = 98
- Y = tbCompareY
- AltBu = '1' because A < B (Unsigned)
- Proves ExecUnit works functional-wise

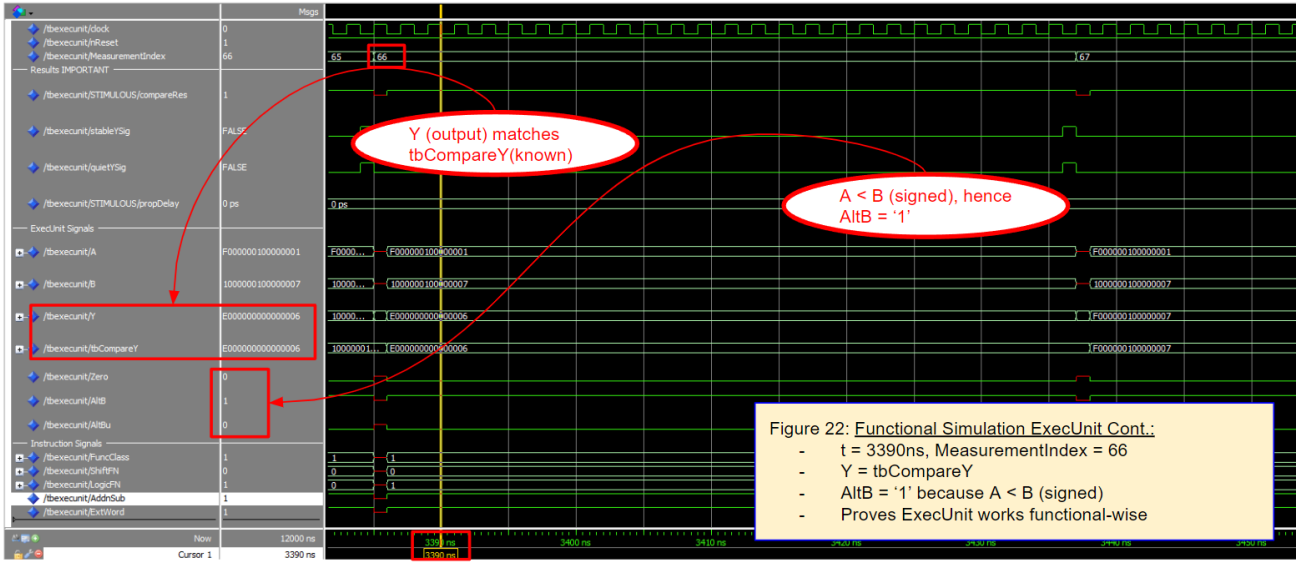
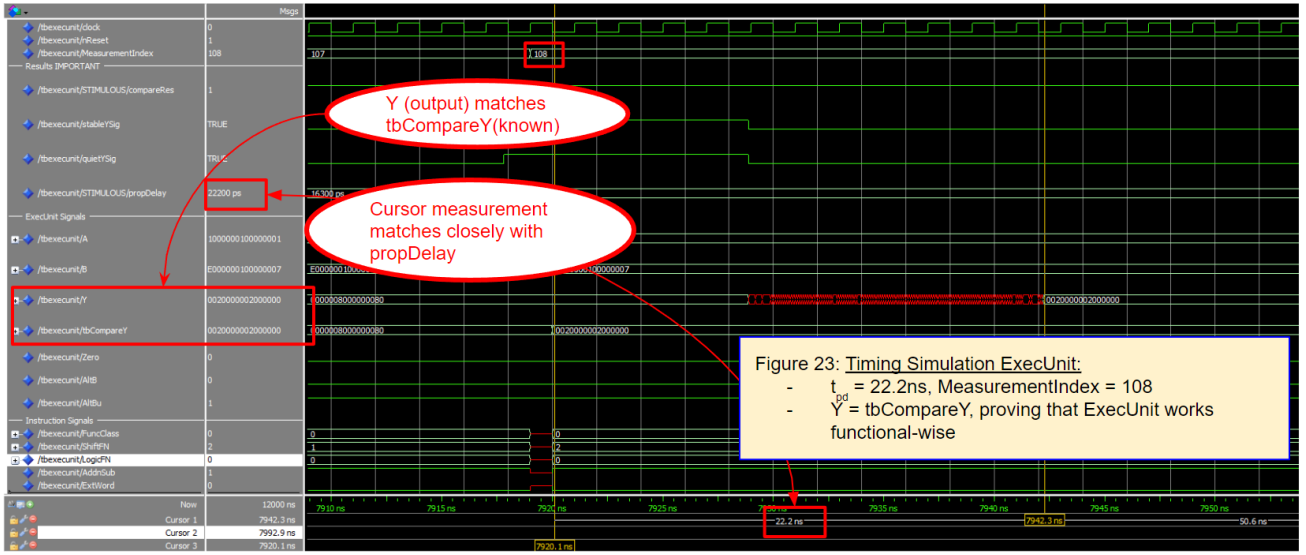
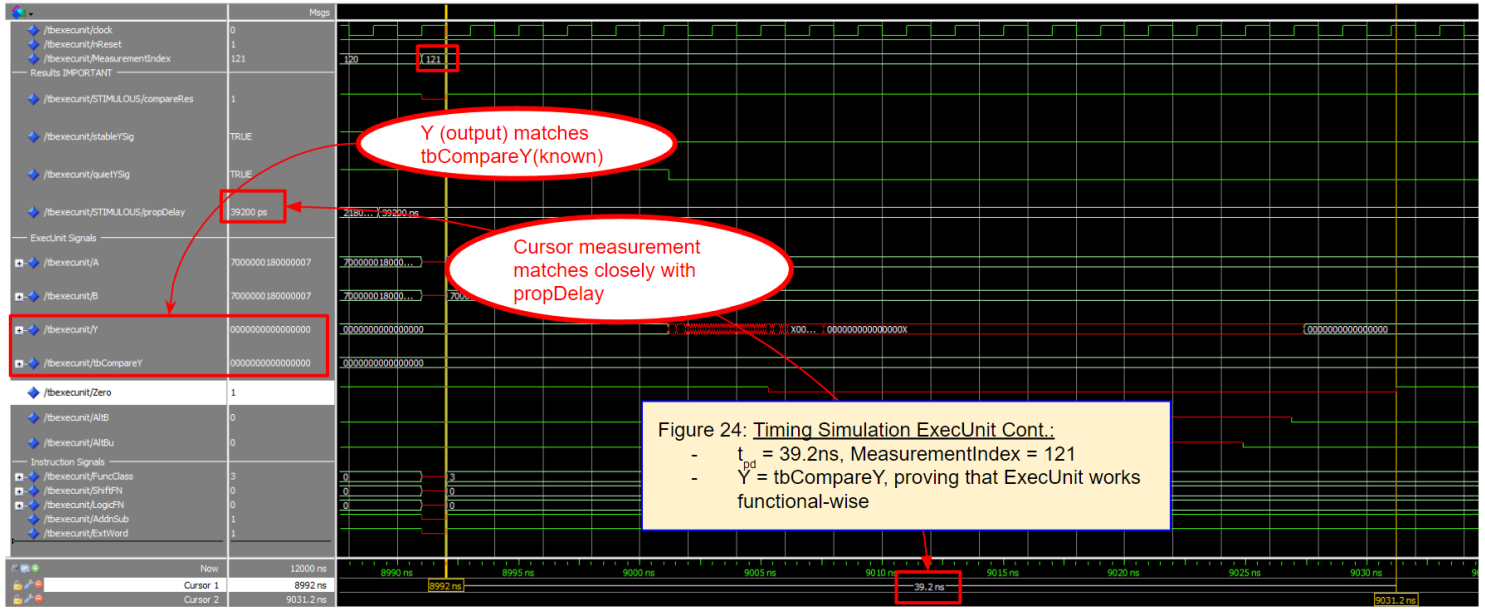


Figure 22: Functional Simulation ExecUnit Cont.:
- t = 3390ns, MeasurementIndex = 66
- Y = tbCompareY
- AltB = '1' because A < B (signed)
- Proves ExecUnit works functional-wise

Timing Simulation



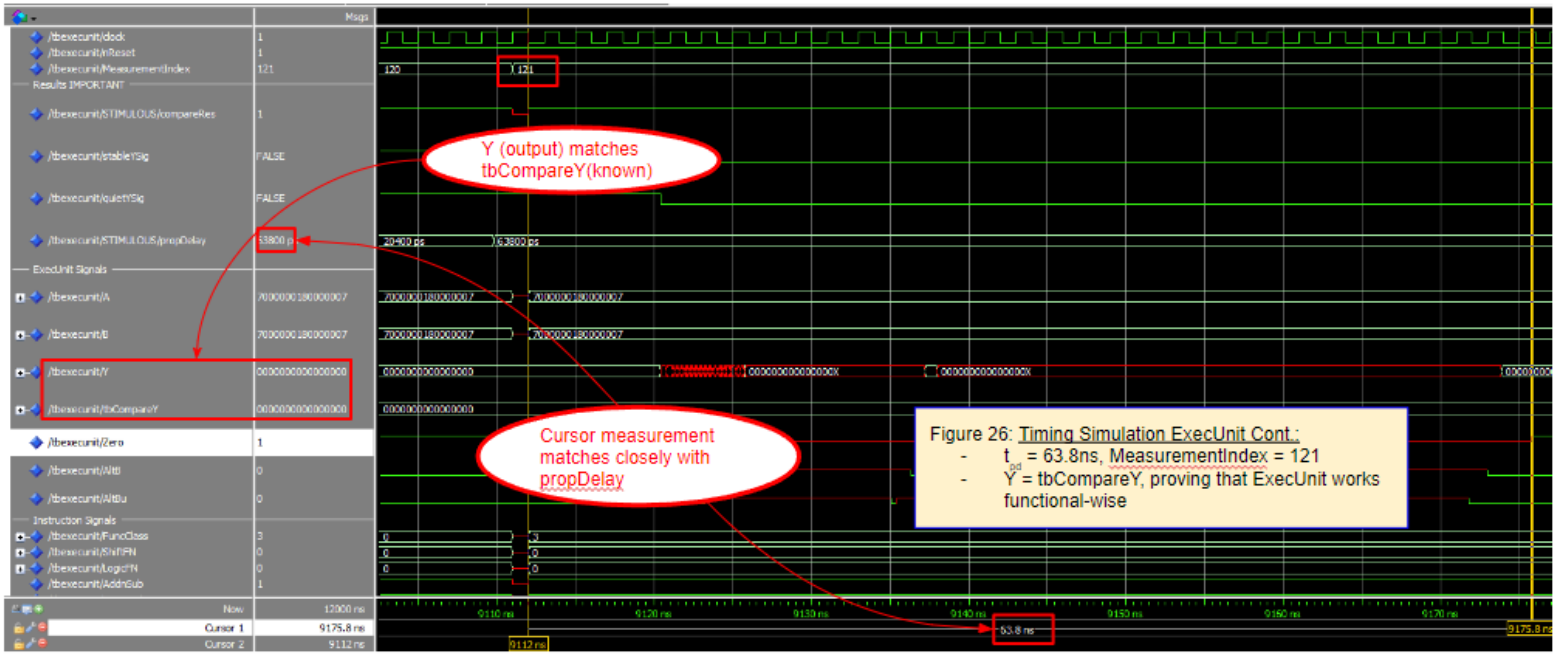


```
# ** Note:    Measurement Index: 108  --  Propagation Delay = 22200 ps
#    Time: 7992200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 109  --  Propagation Delay = 23100 ps
#    Time: 8067100 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 110  --  Propagation Delay = 23100 ps
#    Time: 8143100 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 111  --  Propagation Delay = 21600 ps
#    Time: 8217600 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 112  --  Propagation Delay = 21400 ps
#    Time: 8291400 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 113  --  Propagation Delay = 39200 ps
#    Time: 8383200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 114  --  Propagation Delay = 39200 ps
#    Time: 8475200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 115  --  Propagation Delay = 39200 ps
#    Time: 8567200 ps  Iteration: 0  Instance: /tbexecunit

# ** Note:    Measurement Index: 121  --  Propagation Delay = 39200 ps
#    Time: 9081200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 122  --  Propagation Delay = 39200 ps
#    Time: 9173200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 123  --  Propagation Delay = 39200 ps
#    Time: 9265200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 124  --  Propagation Delay = 39200 ps
#    Time: 9357200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 125  --  Propagation Delay = 39200 ps
#    Time: 9449200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 126  --  Propagation Delay = 39200 ps
#    Time: 9541200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 127  --  Propagation Delay = 39200 ps
#    Time: 9633200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 128  --  Propagation Delay = 39200 ps
#    Time: 9725200 ps  Iteration: 0  Instance: /tbexecunit
# ** Note: TestVector Finished, Closing File and Ending Simulation
#    Time: 9727 ns  Iteration: 0  Instance: /tbexecunit
# transcript off
```

Figure 25: Screenshots from the ModelSim Transcript Window showing that the propagation delay matches with PropDelay from the waveforms in the figures on the previous page

Timing Simulation Cont. (using Ripple Carry Adders instead of Carry-Select Adders)




```
# ** Note:    Measurement Index: 121  --  Propagation Delay = 63800 ps
#    Time: 9225800 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 122  --  Propagation Delay = 63800 ps
#    Time: 9341800 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 123  --  Propagation Delay = 63800 ps
#    Time: 9457800 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 124  --  Propagation Delay = 63800 ps
#    Time: 9573800 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 125  --  Propagation Delay = 63800 ps
#    Time: 9689800 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 126  --  Propagation Delay = 63800 ps
#    Time: 9805800 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 127  --  Propagation Delay = 63800 ps
#    Time: 9921800 ps  Iteration: 0  Instance: /tbexecunit
# ** Note:    Measurement Index: 128  --  Propagation Delay = 63800 ps
#    Time: 10037800 ps  Iteration: 0  Instance: /tbexecunit
# ** Note: TestVector Finished, Closing File and Ending Simulation
#    Time: 10039 ns  Iteration: 0  Instance: /tbexecunit
# transcript off
```

Figure 27: Screenshots from the ModelSim Transcript Window showing that the propagation delay matches with PropDelay from the waveforms in the figures on the previous page

Conclusion

By comparing the propagation delay of the instruction that takes the longest from the two different types of adders above, we observed that the ripple adder network produces a significantly larger propagation delay, having 63800 ps at MeasurementIndex = 121. While using the carry-select adder network produces much less, having 39200 ps at the same index. This optimization produces an 38.6% decrease in propagation delay and it is the ultimate reason that we have decided to use the carry-select adder network instead of the ripple carry network.

Appendix

Please refer to ‘FP-Appendix-G06-350-1227.pdf’.