

Contents

[1. Introduction 2](#_Toc38050781)

[2. LogicUnit 2](#_Toc38050782)

[2.1 Overview & Implementation 2](#_Toc38050783)

[2.2 LogicGates 4](#_Toc38050784)

[2.3 Functional Behaviour 5](#_Toc38050785)

[2.4 Circuit Synthesis 5](#_Toc38050786)

[2.5 Timing Simulations 6](#_Toc38050787)

[3. ArithUnit 7](#_Toc38050788)

[3.1 Overview 7](#_Toc38050789)

[3.2 Implementation 7](#_Toc38050790)

[3.3 Adder 8](#_Toc38050791)

[3.3.2 Implementation 8](#_Toc38050792)

[3.4 Functional Simulation 8](#_Toc38050793)

[3.5 Circuit Synthesis 9](#_Toc38050794)

[3.6 Timing Simulations 9](#_Toc38050795)

[3.6.1 Procedure 9](#_Toc38050796)

[3.6.2 Observations 9](#_Toc38050797)

[4. Designing Barrel Shifters 9](#_Toc38050798)

# 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.

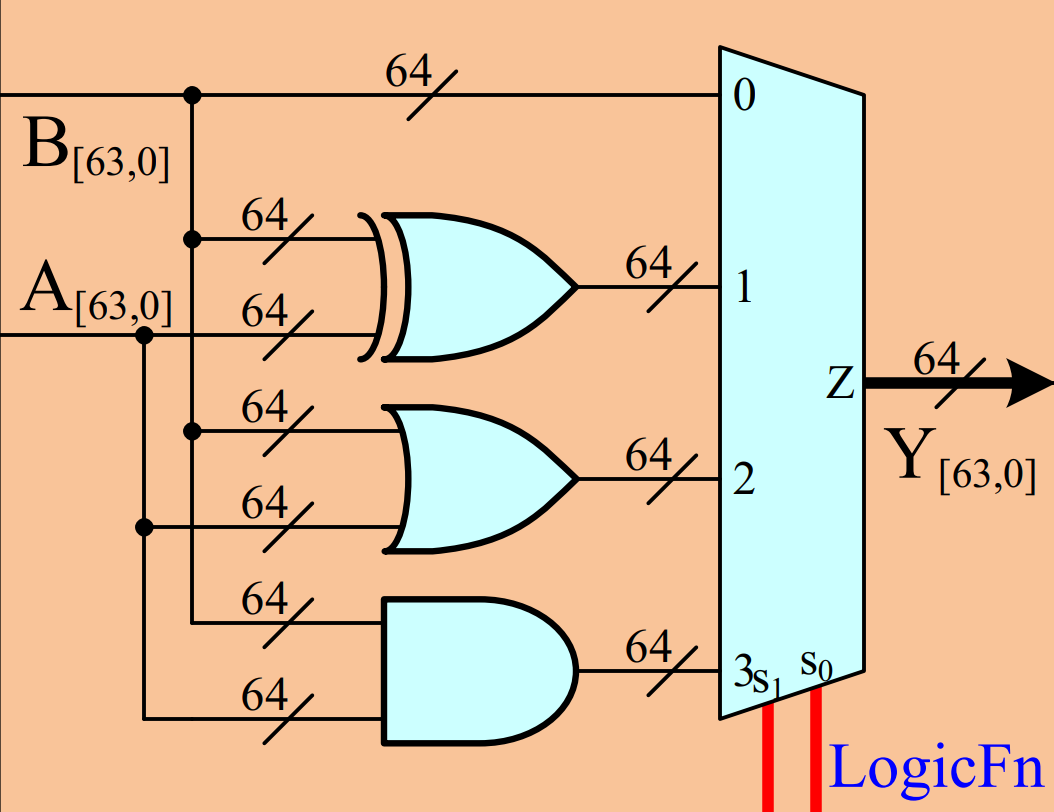


Figure 1:Block Diagram of LogicUnit Circuit

|  |  |
| --- | --- |
| 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

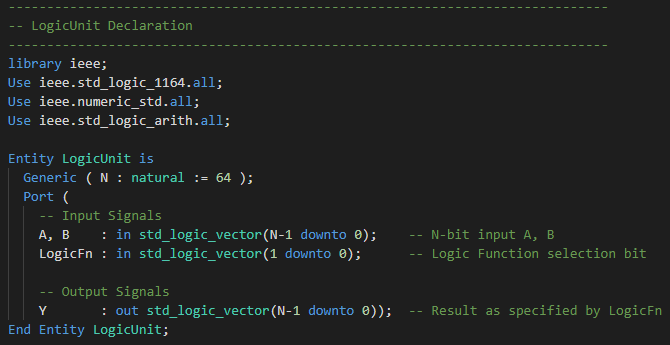


Figure 2: VHDL Interface of LogicUnit

## 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.

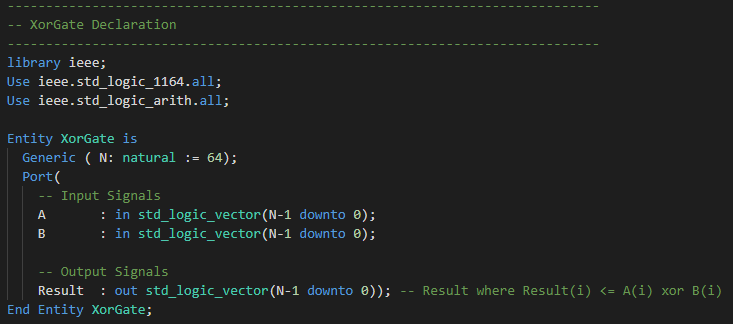


Figure 3: VHDL Interface of XorGate

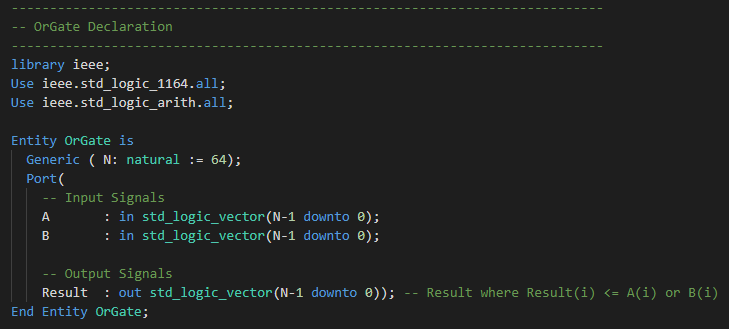


Figure 4: VHDL Interface of OrGate

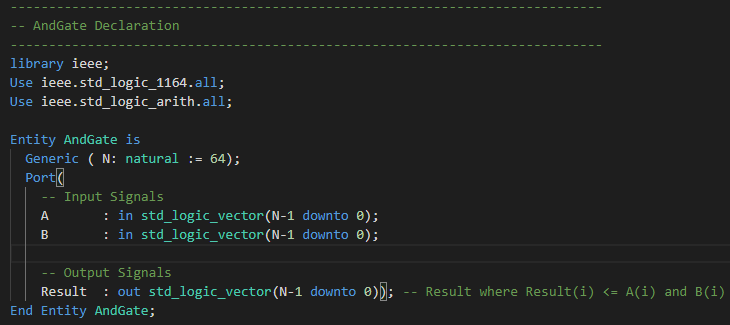


Figure 5: VHDL Interface of AndGate

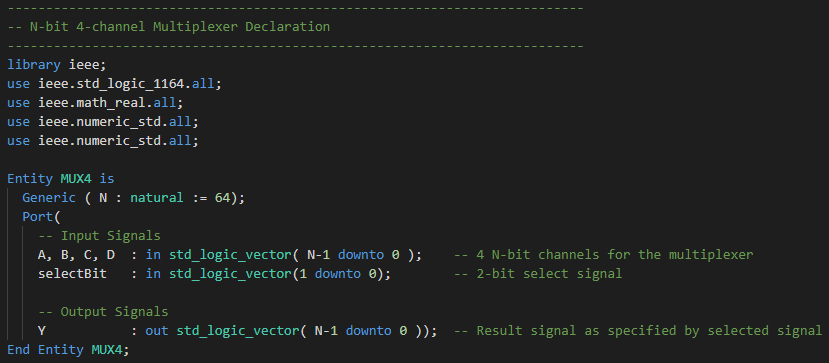


Figure 6: VHDL Interface of MUX4

## 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

## 3.1 Overview

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

## 4.1 Overview

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.