# Design of Reversible Quantum Logic Structures in CMOS Technology

Bahar Canga
bxc7483@rit.edu

# DESIGN OF REVERSIBLE QUANTUM LOGIC STRUCTURES IN CMOS TECHNOLOGY

BAHAR CANGA

# Design of Reversible Quantum Logic Structures in CMOS Technology

by

Bahar Canga

Graduate Thesis

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

August, 2021

# DESIGN OF REVERSIBLE QUANTUM LOGIC STRUCTURES IN CMOS TECHNOLOGY

BAHAR CANGA

**Committee Approval:**

We, the undersigned committee members, certify that Bahar Canga has completed the requirements for the Master of Science degree in Electrical Engineering.

Mr. Mark A. Indovina, *Graduate Research Advisor* Date
Senior Lecturer, Department of Electrical and Microelectronic Engineering

Dr. Dan Phillips Date
Associate Professor, Department of Electrical and Microelectronic Engineering

Mr. Carlos Barrios Date
Lecturer, Department of Electrical and Microelectronic Engineering

Dr. Ferat Sahin, *Department Head* Date
Professor, Department of Electrical and Microelectronic Engineering

## Dedication

I would like to dedicate this work first to my supporting and loving family, my mother Nursel Canga, my father Hakan Canga, my brother Alphan Canga, my significant other Caleb Klaver and my ferrets Greg, Peach, Shadow, Pinky and Nugget, and last but not least to Professor Mark Indovina who made this work possible by guiding, supporting and helping me through.

# Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Bahar Canga

August, 2021

## Acknowledgements

I would like to thank Professor Mark Indovina for guiding, helping and supporting me through this thesis work and being a great role model as an engineer and as an entrepreneur. I would also like to thank to Dr. Dorin Patru, Dr. Ferat Sahin, Dr. Dan Philips and Professor Carlos Barrios for their support, time and feedback. Finally, I would like to thank to my loving and supporting family and friends.

# Abstract

Reversible logic gates have an equal number of inputs and outputs, which also makes it possible to reverse calculate and reconstruct the inputs from the outputs. Quantum logic elements are inherently reversible and requires very little energy to operate. Some of the most common uses of Quantum Computers are in the design of Convolutional Neural Networks (CNN), Deep Neural Networks (DNN) and for machine learning (ML) purposes. In this research, the reversible logic gates were designed with 45$\eta$m CMOS technology modeled after reversible quantum logic gates. As a proof of concept, hardware that provided Sigmoid Neuron Functionality was carried out by processing the MNIST Dataset, a handwritten digit database for number recognition.

# Contents

# List of Figures

# List of Tables

# Listings

# Glossary

**Acronyms**

| | |
|---|---|
| CNN | Convolutional Neural Network |
| DFF | D-Flip Flop |
| DRC | Design Rule Check |
| LVS | Layout Versus Schematic |
| ML | Machine Learning |
| MNIST | Modified National Institute of Standards and Technology |
| MSB | Most Significant Bit |
| QPU | Quantum Processing Unit |
| Qubit | Quantum Bit, the basic unit of quantum information and the quantum version of the classic binary bit physically realized with a two-state device |
| ReLu | Rectified Linear Unit activation function |

# Chapter 1

# Introduction

Throughout history, advances in computers have led to many smart technological gadgets that have revolutionized modern life. Computers have evolved from room sized machines to compact devices thanks to the invention of transistors. Current technology allows the use of smart devices with very high-speed processing and computations. Classical computers manipulate information represented as a sequence of bits. These bits have values represented as either "1" or "0". Computing can also be accomplished by exploiting quantum-mechanical phenomena, where the use of superconducting qubits can have a state of either "1", "0" or both "1 and 0" at a given instance. This is possible due to the superposition and entanglement properties of subatomic particles. The computers that use the quantum-mechanical phenomena for the computation of certain algorithms are called "Quantum Computers". Quantum computers are able to solve certain problems substantially faster than classical computers especially when the problems are complex and well defined. Quantum computers have quantum gates just like the logic gates in classical computers, yet the quantum gates are all reversible. Reversible gates do not lose information, and the only reversible logic gate is a "Not" gate for classical computers. The information loss occurs when there are more inputs than outputs on logic gates, which means the input information is lost forever. However, reversible gates have the ability to reconstruct the inputs from the output information.

In this dissertation, reversible quantum gates will be explored as well as the basis of theory behind Quantum Computing. Many companies, such as Google, IBM and D-Wave were able to demonstrate Quantum computing phenomena, yet without the proper tools and manufacturing, to produce quantum components, it is almost impossible to test and verify the operation of actual Quantum processor. This is primarily due to the probabilistic nature of the Quantum phenomena. The state of a qubit is defined by its probability in superposition state, and once measured, the probability collapses into a known state.

One of the most common uses of Quantum processing methods is the implementation of machine learning (ML) based on Convolutional Neural Networks (CNN) or Deep Neural Networks (DNN). Quantum Processors can significantly reduce the time it takes to implement these algorithms. Normally Quantum Processors use magnetic field to spin the qubit up or down, which is used to calculate the probability of a solution. It is important to note that an actual Quantum Processor was not created. Instead, a cell library with circuits that emulate the functionality of reversible Quantum gates was implemented in CMOS technology. The cell library is documented in detail in Chapter 5, with schematic, symbol and the layout of each cell. The test methodology, test schematics and test results of these components were demonstrated in Chapter 6. Once the reversible single and multi-cell components were designed and verified, a Multiply Accumulate block was designed. The Multiply Accumulate block is commonly used in the implementation of Convolutional Neural Network structures. In order to test and verify the reversible cell based Multiply Accumulate block, Modified National Institute of Standards and Technology (MNIST) dataset was evaluated on the hardware. The MNIST dataset is a database for handwritten digits for digit recognition in CNN.

.

## 1.1 Research Goals

The aim of this work is to research and develop reversible Quantum gates, and then use those gates to create a Multiply Accumulate block for CNN or DNN as follows:

1. To develop Quantum circuits in CMOS technology for emulation and evaluation
2. To validate the operation of the emulated Quantum circuits versus their known Quantum behavior as logic elements
3. Combining emulated Quantum circuits into large building blocks that can be used with Convolutional Neural Networks

## 1.2 Thesis Contributions

The thesis contributions to research and development in the field of digital systems design and verification are as follows:

1. Creation of a CMOS based Reversible Logic and Quantum Gates.
2. Development of a suitable test environment to verify the operation of each Reversible Logic and Quantum Gate
3. Creation of a Reversible 32-bit Carry Look Ahead Adder
4. Development of a suitable test environment to verify the operation of the Reversible 32-bit Carry Look Ahead Adder
5. Creation of a Reversible 32-bit Register
6. Development of a suitable test environment to verify the operation of the Reversible 32-bit Register

7. Creation of a Reversible 16-bit Multiplier
8. Development of a suitable test environment to verify the operation of the Reversible 32-bit Register
9. Creation of Reversible Multiply Accumulate Block
10. Development of a suitable test environment to verify the operation of the Reversible Multiply Accumulate Block
11. Development of a suitable test environment to verify the operation of the Multiply Accumulate Block with MNIST Dataset

## 1.3 Organization

The structure of the thesis is as follows:

- Chapter 1: This chapter introduces the Thesis topic, as well as the goal and the contributions to the engineering filed.
- Chapter 2: This background information is detailed including the motivation for this work and the initial research done prior to beginning this work.
- Chapter 3: The theory of Quantum Mechanics and the difference between Binary and Quantum computations.
- Chapter 4: The Quantum notations and some of the well known Quantum Gates were explained in this chapter.
- Chapter 5: The cell library designed with 45$\eta$m CMOS technology is demonstrated in this chapter. The schematic, symbol and layout of each component are described.

- Chapter 6: The testing and verification of each component in reversible logic CMOS library is discussed in this chapter, including the test methodology, test schematics and the results that were acquired.
- Chapter 7: Through the use of designed and verified reversible transistor library, a Multiply Accumulate Block was designed for CNN. This chapter details the creation and validation of Multiply Accumulate Block.
- Chapter 8: The results and discussion including the measurements of each layout and the transition delay of each CMOS reversible structure are present in this chapter.
- Chapter 9: This chapter discusses the future work that can be done and the conclusion of this Thesis work.
- Appendix I: The testbench code for large components and simulation results for those large components are given in Appendix I.

# Chapter 2

# Background Research

Quantum Processing is considered to be the future of computation. Companies such as Google, IBM and D-Waves, are currently working on their own Quantum Processors. According to an article named "Quantum Supremacy using a Programmable Superconducting Processor" [3], Google and NASA collaboratively worked on creating a processor with programmable superconducting qubits and adjustable couplers. According to the authors of that article [3], their processor takes about 200 seconds to complete a task that would take a supercomputer around 10,000 years.

Even though Quantum Computers are known to compute some algorithms exponentially faster than that of Classical Computers, based on the paper called "D-Wave's Quantum Processing Unit" which was written by Bahar Canga [4], D-Wave's Quantum Processing Unit (QPU) demonstrates a significant performance increase over Classical Computers when the algorithm that runs is well defined and complex. Some of the research that has been done on D-Wave's QPU involved Convolutional Neural Networks (CNN). Both the authors of [5] and [6] conducted research on D-Wave's QPU by using a database of handwritten digits called MNIST Dataset [7]. Both research was conducted by running the MNIST dataset on D-Wave's QPU with autoencoders in an unsupervised way.

Based on [4], one of the most common uses of Quantum Processors are for machine learning purposes. As CNNs can also be implemented purely in hardware, having custom hardware can increase the performance of the simulations. This research paper focused on creating a medium for running CNN algorithms in a custom Sigmoid Neuron Function Hardware that is made out of reversible gates. For this purpose, there were three crucial elements needed to achieve a Reversible Multiply Accumulate Block, which are a fast multiplier, a fast adder and a register.

The register that was created in this thesis work is made out of D-Latches. The design for the D-Latch was referenced from an article titled "Design and analysis of Flip-Flops using reversible logic" [8] as well as "Design of Reversible Logic based Basic Convolutional Circuits" [9]. Initially the purpose was to construct a D-Flip Flop (DFF) based on [8] and [9], yet after implementing and testing the positive-edge triggered DFF design from [9], the result illustrated a D-Latch behavior. Even though the DFF design shown in [9] demonstrated a D-Latch behavior, the 32-bit Register was built out of that architecture, which also allowed time borrowing. Time borrowing allows the circuit to borrow time from a separate path within a latch.

The adder that is built for this project needed to be fast and reversible. The Carry Look Ahead Adder is well known to be one of the fastest adder. In this research, the Carry Look Ahead Adder was implemented with reversible Classical and Quantum gates. According to an article named "A Logarithmic-depth Quantum Carry Look Ahead Adder" [10], the information in scratch space needs to be erased, and the operations should not destroy any information. Information in scratch space is an extra signal that is only there to allow computation of another signal. This signal is just like the Quantum Full Adder, where the signal Z, the Zero signal is there to compute the carry out. In [10], the Carry Look Ahead was designed purely out of Quantum gates, which are Toffoli and CNOT. As reversible NAND gate was created in this research, the Carry Look Ahead Adder was designed based on Classical circuit architecture. The circuitry given in [10] can be used in future work, with the use of only CNOT and Toffoli gate.

For the multiplier to be fast, initially a multiplier with Booth encoder and Wallace Tree Adder design was implemented based on the architecture given in a book titled “CMOS VSLI Design: A Circuits and Systems Perspective” [11], yet the design did not correctly compute the partial products. The next design was implemented by simply using 256 reversible NAND Gates to calculate partial products. For the multiplier to calculate the result, the partial products either needed to be added by rows or by column. In one implementation, the partial products were added by column based on the Wallace Tree Adder architecture given in [11]. The Wallace tree works by adding every 3 input and adding the respective carry and sum as the next input in the form of a tree. By adding the partial products by column with Wallace Tree Adder architecture, the results were not accurate. This was because for each next column, all the previous carry bits needed to be added along with every partial product. This would exponentially increase the circuit size to correct the addition. Thus, instead of that, the 32-bit Carry Look Ahead Adder was used 8 times with Wallace Tree structure. Even though the wiring was done correctly, the results acquired were not accurate. As the previous multiplier implementations were not successful, another efficient multiplication design was created based on Vedic Mathematics. According to [12–15], the Vedic algorithm conducts the Multiplication operation both vertically, crosswise and in parallel. This means that the Vedic algorithm requires a 2 by 2 multiplier to multiply 2-bit numbers, and by concatenating 4 of those 2 by 2 multipliers, a 4-bit multiplier could be designed. The same thing applies for 4 of 4-bit multipliers concatenating to create and 8-bit multiplier and finally 4 of 8-bit multipliers to concatenate to create a 16-bit multiplier. This design successfully passed the pre-layout simulation and eventually was used in the Multiply Accumulate Block created for this project.

# Chapter 3

# Theory

Quantum computers operate with quantum principals. In quantum mechanics and particle physics, spin is referred to as the angular momentum in intrinsic form of subatomic particles. A quantum particle, which can be a single photon, a nucleus of an atom, or an electron, has a magnetic field around it. When an external magnetic field is applied to a quantum particle, the particle aligns with that field. In that state, the particle has the lowest energy level, which is the spin down or the "0" state. In order to spin the particle up, it requires an external force to amplify the energy level. These two states are just like the bits in classical computers, yet the qubits can be both at a given instance. This is mainly caused by the superposition principle, which states that any linear system can be in one of many possible configurations and the most general state is the combination of all the possible states. However, when the qubits are measured, the result collapses into a known classical state.

## 3.1 Schrödinger's Equation

In 1935, Erwin Schrödinger came up with a hypothetical experiment while having a course of discussion with Albert Einstein. This thought experiment is well known as "Schrödinger's Cat" and the purpose was to point out the paradox of probable events and the uncertainty of the results until observation. In this hypothetical experiment, Schrödinger inserts a cat in a box and seals it with a flask of poison as well as a radioactive source. In this thought experiment, there is a 50% chance that the radioactive material would decay, thus the flask would shatter, which would end up killing the hypothetical cat. As the box is sealed in this experiment, there is no way to know exactly what state the cat is. The cat is both dead and alive until somebody opens the box and observes the exact state of the cat for sure. The same principle applies to quantum mechanics. The quantum particles are in superposition state with certain probabilities of them being $|0>$ or $|1>$, such as 35% and 65% respectively. At that instance, the qubits are said to be both $|0>$ and $|1>$. Once the particle's state is measured, the probabilities collapse, which results in a definite qubit state of either $|0>$ or $|1>$. With everything in mind, Schrödinger came up with an equation which can be seen below. This equation describes the probability of finding a particle at a certain position. The equation 3.1 states that Hamiltonian operator, $\hat{H}$,and wavefunction of an electron, |ψ>, is equal to the square root of minus one, i, multiplied with the Planc's Constant, $\hbar$, multiplied with the rate of change of wavefunction with respect to time.

$$\hat{H}|\psi(t)>= i\hbar\frac{\partial}{\partial t}|\psi(0)>|\psi(t)> \tag{3.1}$$

## 3.2 Bloch Sphere

In Quantum Mechanics, the Bloch Sphere is a geometrical representation of the state of a single qubit which is on the surface of a unit sphere. The Bloch Sphere representation can be seen in

Figure 3.1. The Bloch Vector is shown as |ψ>, has a state of |Ψ> = αβ. The probability of the state of a qubit is defined around the Bloch Sphere.

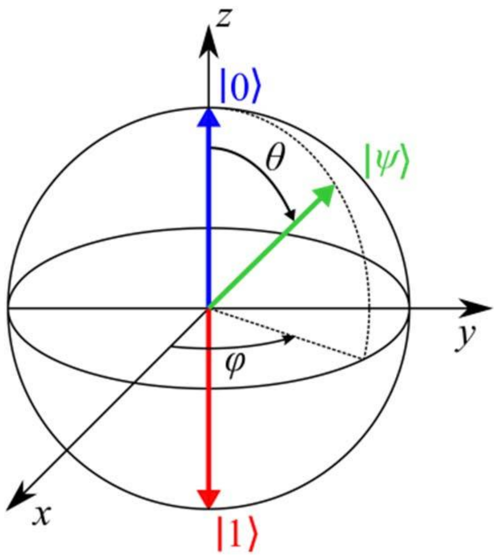


Figure 3.1: Bloch Sphere

## 3.3 Energy Conservation

According to the law of conservation of energy, energy can neither be created nor be destroyed, yet it can be transformed into another form. The majority of electronics are made out of logic gates, that usually have more number of inputs than outputs. This suggests that the energy entering that system through the input is larger than the energy coming out. What happens to that loss of energy? Most probably, the loss of energy turns into harmful radiation, such as radio frequency, microwave and photons, as well as excessive heat that both drains power unnecessarily and lowers the lifespan of electronics. On the other hand, through the use of reversible components, the loss of energy can be reduced significantly.

# Chapter 4

# Quantum Notations and Quantum Gates

In this chapter, a brief introduction to quantum notations and quantum gates will be explained. As mentioned in Chapter 1, all quantum gates are inherently reversible. Quantum logic can be constructed by using these reversible single-qubit, two-qubit and three-qubit gates. The quantum gates can be represented with matrix notations, truth tables and Bloch Spheres. These quantum gates rotate the qubits in certain ways either around x, y or z axis, or around a diagonal axis in the x-z plane.

## 4.1 Hilbert Spaces and Dirac Notation

In quantum mechanics, the state of qubits are represented in a Hilbert space which is a vector space with an inner product as well as a norm described by that inner product. In order to describe vectors in quantum mechanical systems, Dirac notation or Bra-Ket is commonly used. In the following subsections, the Dirac notations will be explained.

#### 4.1.0.1 Ket

Ket is a column vector, and it is used to indicate the state of a qubit. The wave function is represented with Ket notation. The Ket notation can be seen as below as |v>,

$$|v>= \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = v$$

#### 4.1.0.2 Bra

Bra is the dual vector of |v> or Ket, and it is the transposed complex conjugate square of v. The notation of Bra can be seen below as <v|.

$$< v|= \begin{bmatrix} \overline{v_0} & \overline{v_1} & \overline{v_2} & \cdots & \overline{v_n} \end{bmatrix} = \overline{v^T}$$

## 4.2 Quantum Gates

#### 4.2.0.1 Identity (I) Gate

Identity Gate is a single qubit gate, with a single input and a single output. Identity gate has no impact on the rotation of the qubits. It can also be represented as a wire. The symbol of 4.2.0.1 is shown in Figure Identity (I) Gate. The matrix representation of this gate can be seen in Algorithm 4.1 below. The circuit representation of the Identity gate is illustrated in Table 4.1.

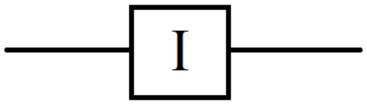


Figure 4.1: Identity Gate Symbol

**Algorithm 4.1** Identity Matrix Notation

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

| Input | Output |
|---|---|
| $\|0>$ | $\|0>$ |
| $\|1>$ | $\|1>$ |

Table 4.1: Truth Table of Identity Gate

#### 4.2.0.2 Pauli-X (X) Gate

Pauli-X Gate is a single qubit gate that rotates the qubit state by $180^\circ$ ($\pi$ radians) around x-axis. Pauli-X Gate is the quantum equivalent of NOT gate in classical computers. This implies that Pauli X converts |0> to |1>, and |1> to |0>. The symbol of Pauli-X (X) Gate is shown in Figure

4.2. The matrix representation of this gate can be seen in Algorithm 4.2 below. The circuit representation of Pauli-X gate is illustrated in Table 4.2.

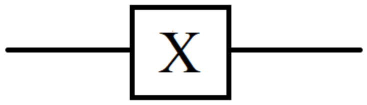


Figure 4.2: Pauli-X Gate Symbol

**Algorithm 4.2** Pauli-X Matrix Notation

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

| Input | Output |
|---|---|
| $\|0>$ | $\|1>$ |
| $\|1>$ | $\|0>$ |

Table 4.2: Truth Table of Pauli-X Gate

#### 4.2.0.3 Pauli-Y (Y) Gate

Pauli-Y Gate is a single qubit gate and rotates the qubit state by 180° (π radians) around y-axis. The symbol of Pauli-Y (Y) Gate is shown in Figure 4.3. The matrix representation of this gate can be seen in Algorithm 4.3 below. The circuit representation of the Pauli-Y gate can be seen in Table 4.3.

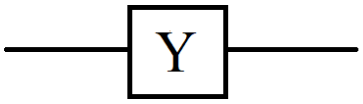


Figure 4.3: Pauli-Y Gate Symbol

**Algorithm 4.3** Pauli-Y Matrix Notation

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

| Input | Output |
|---|---|
| $\|0>$ | $i\|1>$ |
| $\|1>$ | $-i\|0>$ |

Table 4.3: Truth Table of Pauli-Y Gate

#### 4.2.0.4 Pauli-Z (Z) Gate

Pauli-Z Gate is a single qubit and rotates the qubit state by 180° (π radians) around z-axis. The symbol of Pauli-Z (Z) Gate is shown in Figure 4.4. The matrix representation of this gate can be seen in Algorithm 4.4 below. The circuit representation of the Pauli-Z gate is illustrated in Table 4.4.

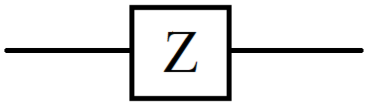


Figure 4.4: Pauli-Z Gate Symbol

**Algorithm 4.4** Pauli-Z Matrix Notation

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

| Input | Output |
|---|---|
| $\|0>$ | $\|0>$ |
| $\|1>$ | $-\|1>$ |

Table 4.4: Truth Table of Pauli-Z Gate

#### 4.2.0.5 Phase (S, P) Gate

Phase Gate or S Gate is a single qubit gate and rotates the qubit state by 90˚ (π/2 radians) around z-axis. The symbol of Phase (S, P) Gate is shown in Figure 4.5. The matrix representation of this gate can be seen in Algorithm 4.5 below. The circuit representation of the Phase gate is illustrated in Table 4.5.

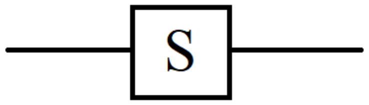


Figure 4.5: Phase Gate Symbol

**Algorithm 4.5** Phase Gate Matrix Notation

$$S = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix}$$

| Input | Output |
|---|---|
| $\|0>$ | $\|0>$ |
| $\|1>$ | $e^{i\frac{\pi}{2}}\|1>$ |

Table 4.5: Truth Table of Phase Gate

#### 4.2.0.6 Hadamard (H) Gate

Hadamard Gate is a single qubit gate that rotates the qubit state by 180° (π radians) around y-axis. Hadamard Gate is one of the most commonly used quantum gates as it inserts the qubits in superposition state, where the probability of the result being |0> and |1> are equally likely. The symbol of Hadamard (H) Gate is shown in Figure 4.6. The matrix representation of this gate can be seen in Algorithm 4.2 below. The circuit representation of the Hadamard gate is illustrated in Table 4.2.

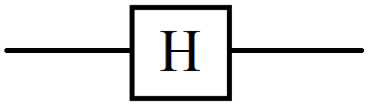


Figure 4.6: Hadamard Gate Symbol

**Algorithm 4.6** Hadamard Matrix Notation

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

| Input | Output |
|---|---|
| $\|0>$ | $\frac{\|0>+\|1>}{\sqrt{2}}$ |
| $\|1>$ | $\frac{\|0>-\|1>}{\sqrt{2}}$ |

Table 4.6: Truth Table of Hadamard Gate

#### 4.2.0.7 CNOT Gate

CNOT gate is a quantum gate, which is also known as Controlled Not Gate. The CNOT gate has a control signal, which the value of the control signal does not get changed. If the value of the control signal is $|1>$, the output of the second input gets inverted. The CNOT gate behaves like an XOR gate in classical computation. The quantum representation of the CNOT gate can be seen in Figure 4.7. The truth table of the CNOT gate is illustrated in Table 4.7 below.

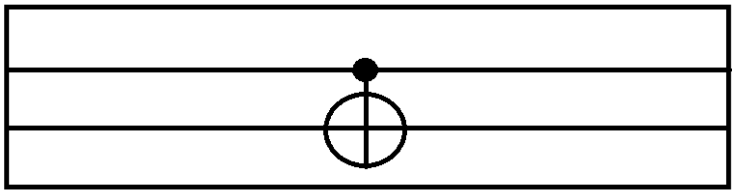


Figure 4.7: CNOT Gate Symbol

| Input 1 | Input 2 | Output 1 | Output 2 |
|---|---|---|---|
| $\|0>$ | $\|0>$ | $\|0>$ | $\|0>$ |
| $\|0>$ | $\|1>$ | $\|0>$ | $\|1>$ |
| $\|1>$ | $\|0>$ | $\|1>$ | $\|1>$ |
| $\|1>$ | $\|1>$ | $\|1>$ | $\|0>$ |

Table 4.7: Truth Table of CNOT Gate

#### 4.2.0.8 Toffoli Gate

Toffoli gate is a quantum gate, which is also known as Controlled Controlled Not Gate. The Toffoli gate has two control signals. The value of the control signals do not get changed and if the value of both of the control signals are $|1>$, the output of the third input gets inverted. The quantum representation of the Toffoli gate can be seen in Figure 4.8. The truth table of the Toffoli gate is illustrated in Table 4.8 below.

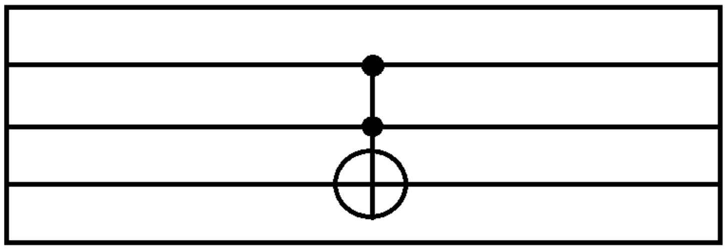


Figure 4.8: Toffoli Gate Symbol

| Input 1 | Input 2 | Input 3 | Output 1 | Output 2 | Output 3 |
|---|---|---|---|---|---|
| $\|0>$ | $\|0>$ | $\|0>$ | $\|0>$ | $\|0>$ | $\|0>$ |
| $\|0>$ | $\|0>$ | $\|1>$ | $\|0>$ | $\|0>$ | $\|1>$ |
| $\|0>$ | $\|1>$ | $\|0>$ | $\|0>$ | $\|1>$ | $\|0>$ |
| $\|0>$ | $\|1>$ | $\|1>$ | $\|0>$ | $\|1>$ | $\|1>$ |
| $\|1>$ | $\|0>$ | $\|0>$ | $\|1>$ | $\|0>$ | $\|0>$ |
| $\|1>$ | $\|0>$ | $\|1>$ | $\|1>$ | $\|0>$ | $\|1>$ |
| $\|1>$ | $\|1>$ | $\|0>$ | $\|1>$ | $\|1>$ | $\|1>$ |
| $\|1>$ | $\|1>$ | $\|1>$ | $\|1>$ | $\|1>$ | $\|0>$ |

Table 4.8: Truth Table of Toffoli Gate

#### 4.2.0.9 SWAP Gate

SWAP gate is a quantum gate, which swaps both inputs with the outputs. The quantum representation of the SWAP gate can be seen in Figure 4.9. The truth table of the SWAP gate is illustrated in Table 4.9 below.

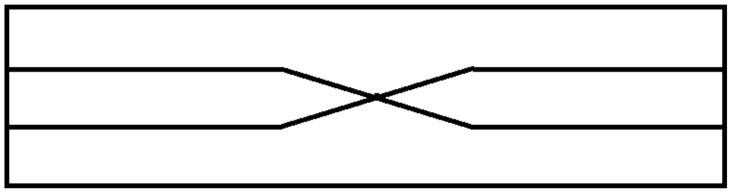


Figure 4.9: SWAP Gate Symbol

| Input 1 | Input 2 | Output 1 | Output 2 |
|---|---|---|---|
| $\|0>$ | $\|0>$ | $\|0>$ | $\|0>$ |
| $\|0>$ | $\|1>$ | $\|1>$ | $\|0>$ |
| $\|1>$ | $\|0>$ | $\|0>$ | $\|1>$ |
| $\|1>$ | $\|1>$ | $\|1>$ | $\|1>$ |

Table 4.9: Truth Table of SWAP Gate

#### 4.2.0.10 Fredkin Gate

Fredkin gate is a quantum gate, which is also known as Controlled SWAP Gate. The Fredkin gate has a control signal which stays at the same state throughout. The Fredkin gate swaps the second and third input if and only if the first input, the control signal, has a value of 1. The quantum representation of the Fredkin gate can be seen in Figure 4.10. The truth table of the Fredkin gate is illustrated in Table 4.10 below.

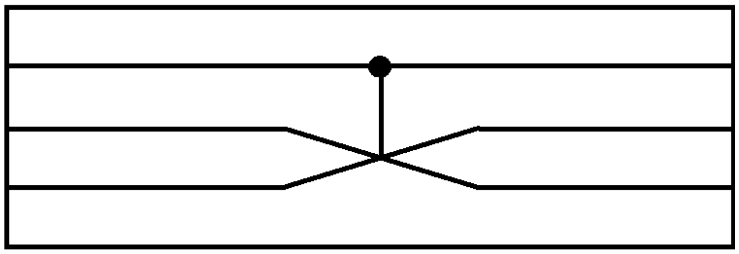


Figure 4.10: Fredkin Gate Symbol

| Input 1 | Input 2 | Input 3 | Output 1 | Output 2 | Output 3 |
|---|---|---|---|---|---|
| $\|0>$ | $\|0>$ | $\|0>$ | $\|0>$ | $\|0>$ | $\|0>$ |
| $\|0>$ | $\|0>$ | $\|1>$ | $\|0>$ | $\|0>$ | $\|1>$ |
| $\|0>$ | $\|1>$ | $\|0>$ | $\|0>$ | $\|1>$ | $\|0>$ |
| $\|0>$ | $\|1>$ | $\|1>$ | $\|0>$ | $\|1>$ | $\|1>$ |
| $\|1>$ | $\|0>$ | $\|0>$ | $\|1>$ | $\|0>$ | $\|0>$ |
| $\|1>$ | $\|0>$ | $\|1>$ | $\|1>$ | $\|1>$ | $\|0>$ |
| $\|1>$ | $\|1>$ | $\|0>$ | $\|1>$ | $\|0>$ | $\|1>$ |
| $\|1>$ | $\|1>$ | $\|1>$ | $\|1>$ | $\|1>$ | $\|1>$ |

Table 4.10: Truth Table of Fredkin Gate

# Chapter 5

# Cell Library

The cell library for the reversible quantum gates were designed through Cadence Custom IC Design tool flow, and the technology used was 45 $\eta$m using a generic library. In this chapter, the reversible gates that were designed can be seen in both schematic, symbol, layout as well as simulation view. As mentioned in earlier chapters, the reversible gates have equal number input and output. This means that the number of input are equal to the output of each cell hierarchy. The symbol of each single cell component has an accurate quantum representation.

### 5.0.1 Inverter

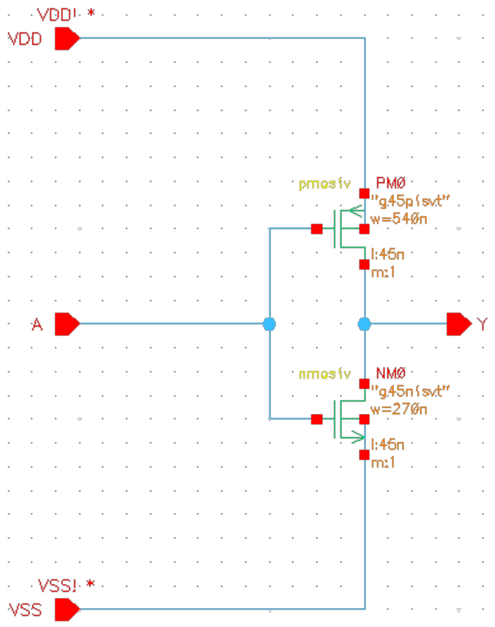


Figure 5.1: Inverter Schematic

The inverter is inherently reversible given that there is one input and an opposite output. In Figure 5.1, the schematic of the inverter can be seen. The width of the pMOS device is 540 ηm, thus based on 2:1 ratio, the nMOS device is 270 ηm. The 2:1 ratio is the ratio of width over length of pMOS compared to the width over length ratio of nMOS. This ratio is to keep the resistance of two transistors same, so that the rise and rall time of each transistor is similar. In Figure 5.2, the symbol of the inverter is present.

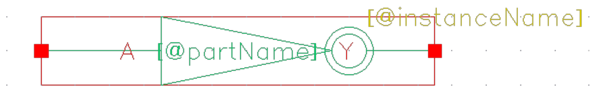


Figure 5.2: Inverter Symbol

The layout of the inverter is shown in Figure 5.3.

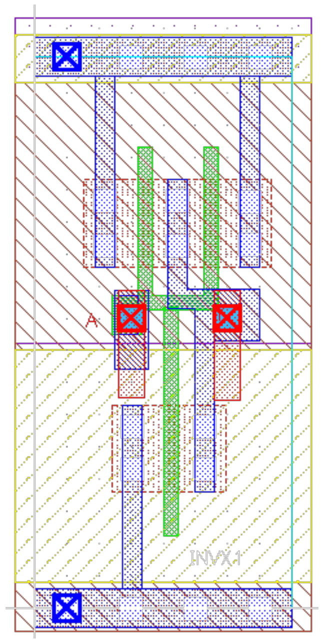


Figure 5.3: Inverter Layout

### 5.0.2 Reversible NAND

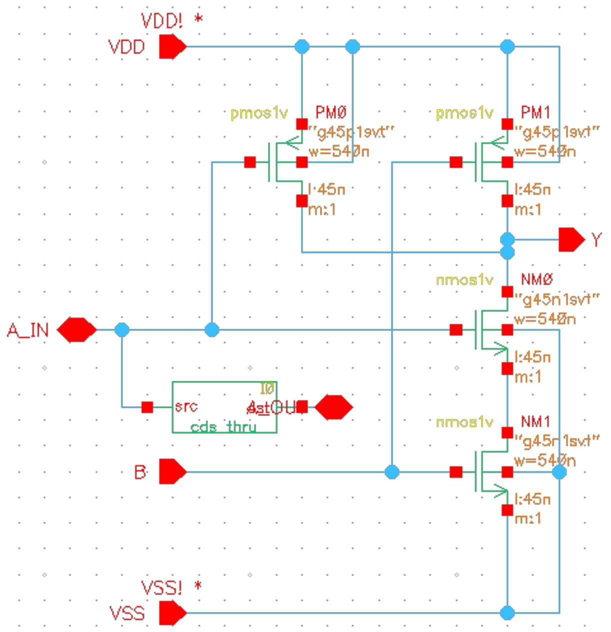


Figure 5.4: Reversible NAND Schematic

A NAND gate is not inherently reversible gate unlike an inverter. Thus, in order to equate the number of input and output pins, the input pin to be replicated was turned into an input-output pin. This allows for a pin to act both as an input as well as an output pin. Due to conflict issues, the pins were named distinctly of one another. The block named cds_tru was placed between the two input-output pins to connect two nets together and to avoid any netlist errors. Adding an extra input-output pin adds a marginal amount of capacitance to the input pin load, however, the fanout of the input-output pin would need to be taken into account when sizing the driving cell. In Figure 5.4, the schematic of the Reversible NAND gate can be seen. The width of the pMOS devices are 540 $\eta$m, thus based on 2:1 ratio rule of pMOS and nMOS, each of the series nMOS devices are 540 $\eta$m. The 2:1 ratio is the ratio of width over length of pMOS compared to the width over length ratio of nMOS. This ratio is to keep the resistance of two transistors same, so that the rise and fall time of each transistor is similar. In Figure 5.5, the symbol of the Reversible NAND is present.

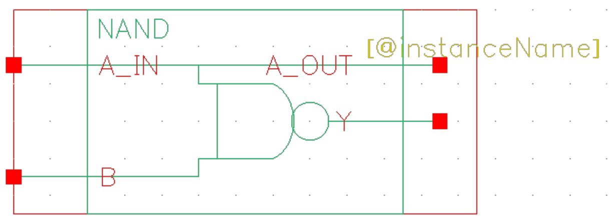


Figure 5.5: Reversible NAND Symbol

The layout of the NAND is shown in Figure 5.6.

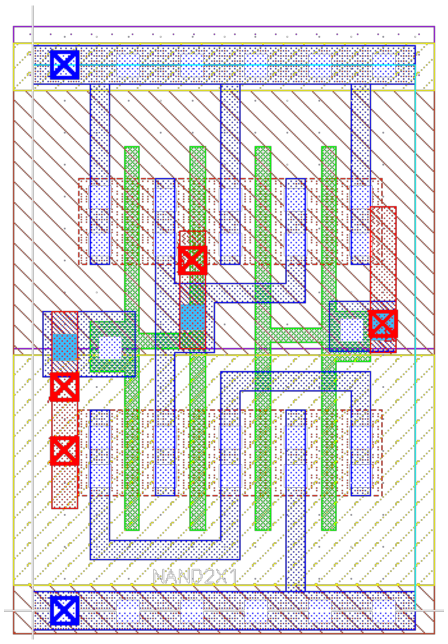


Figure 5.6: Reversible NAND Layout

### 5.0.3 Reversible NOR

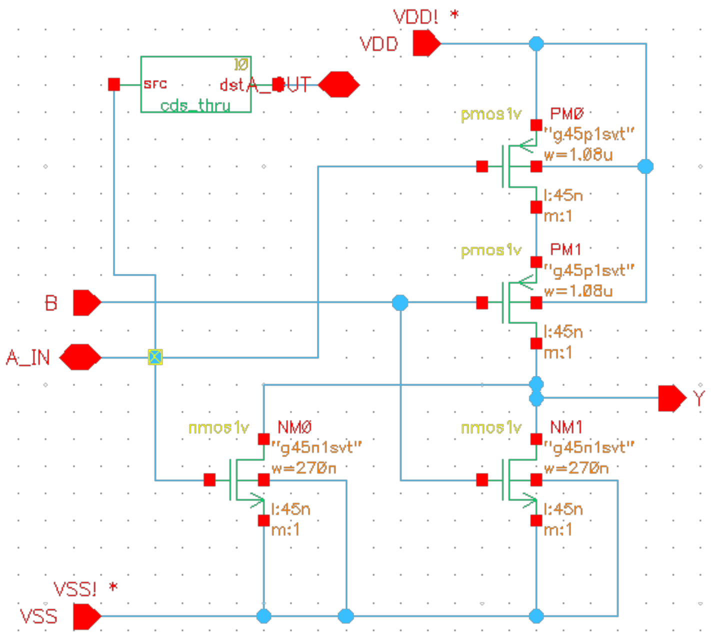


Figure 5.7: Reversible NOR Schematic

Just like the NAND gate, a NOR gate is also not inherently reversible. Thus, in order to equate the number of fan-in and fan-out, the input pin to be replicated was also turned into an input-output pin. As mentioned above, converting the pin to be replicated into an input-output pin allows for a pin to act both as an input as well as an output pin, creating a two way flow. Due to conflict issues, the pins were named distinctly of one another. The block named cds_tru was placed between the two input-output pins to connect two nets together and to avoid any netlist errors. Adding an extra input-output pin adds a marginal amount of capacitance to the input pin load, however, the fanout of the input-output pin would need to be taken into account when sizing the driving cell. This applies to the rest of the cells as well. In Figure 5.7, the schematic of the NOR gate can be seen. The width of the pMOS devices are 1080 $\eta$m, thus based on 2:1 ratio rule of pMOS and nMOS, each of the parallel nMOS devices are 270 $\eta$m. In Figure 5.8, the symbol of the Reversible NOR is present.

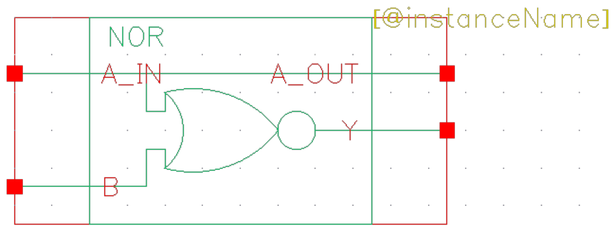


Figure 5.8: Reversible NOR Symbol

The layout of the reversible NOR gate is shown in Figure 5.9.

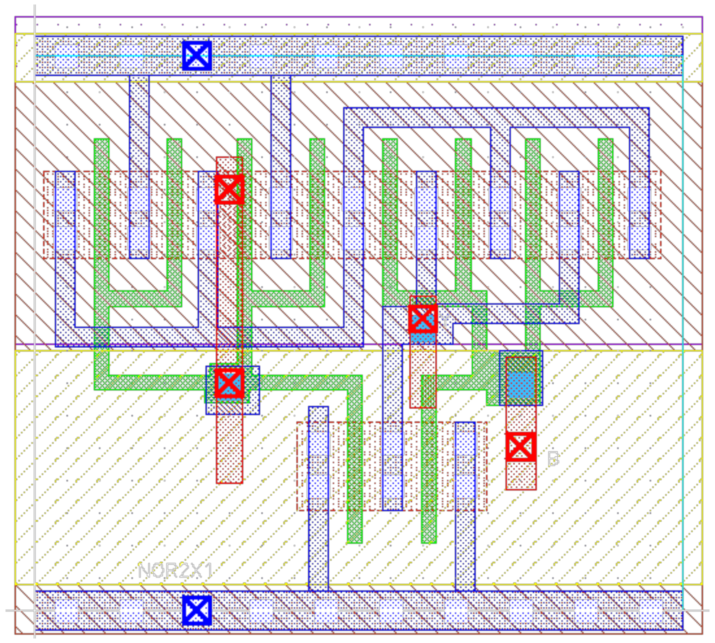


Figure 5.9: Reversible NOR Layout

### 5.0.4 CNOT

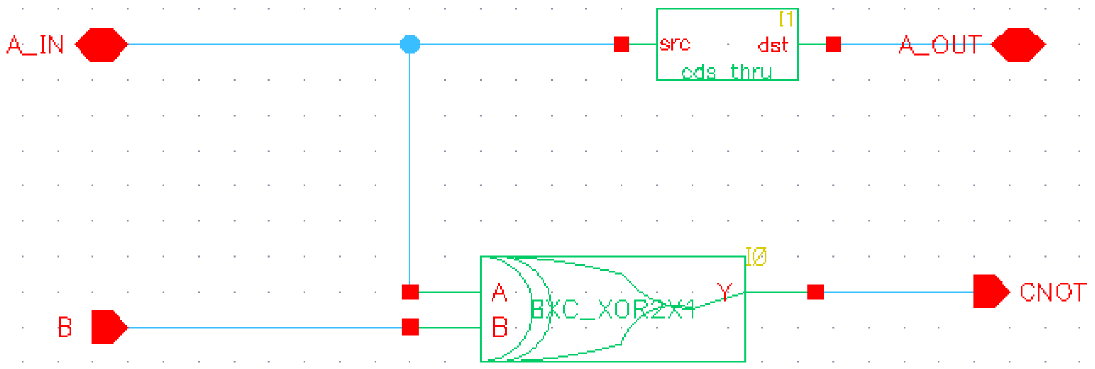


Figure 5.10: CNOT Schematic

The CNOT gate is a quantum gate which is also known as Controlled Not Gate. Based on the controlled reversible input, the function XORs the two incoming input. In Figure 5.10, the schematic of the CNOT gate can be seen. In 5.11, the symbol of the CNOT gate is present.

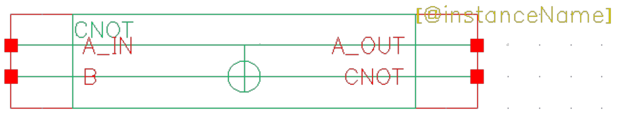


Figure 5.11: CNOT Symbol

The layout of the CNOT is shown in 5.12.

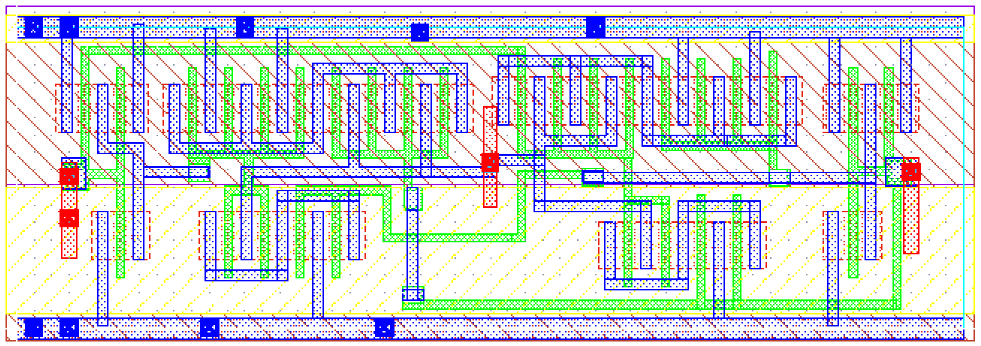


Figure 5.12: CNOT Layout

### 5.0.5 SWAP

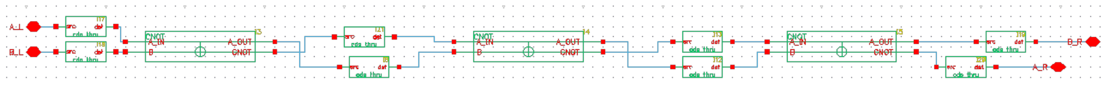


Figure 5.13: SWAP Schematic

The SWAP gate is another quantum gate that switches the two incoming inputs to be each other's outputs. This gate is made out of three XOR gates connected back to back. The schematic of the SWAP gate can be seen in Figure 5.13. In 5.14, the symbol of the SWAP gate is present.

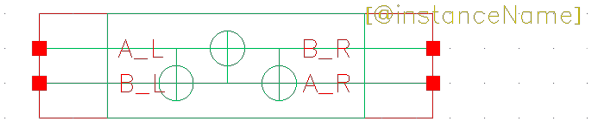


Figure 5.14: SWAP Symbol

The layout of the SWAP is shown in 5.15.

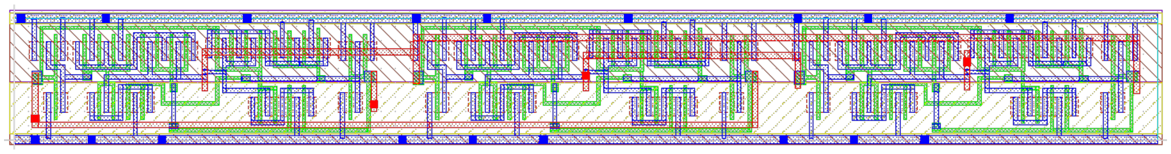


Figure 5.15: SWAP Layout

### 5.0.6 Toffoli Gate

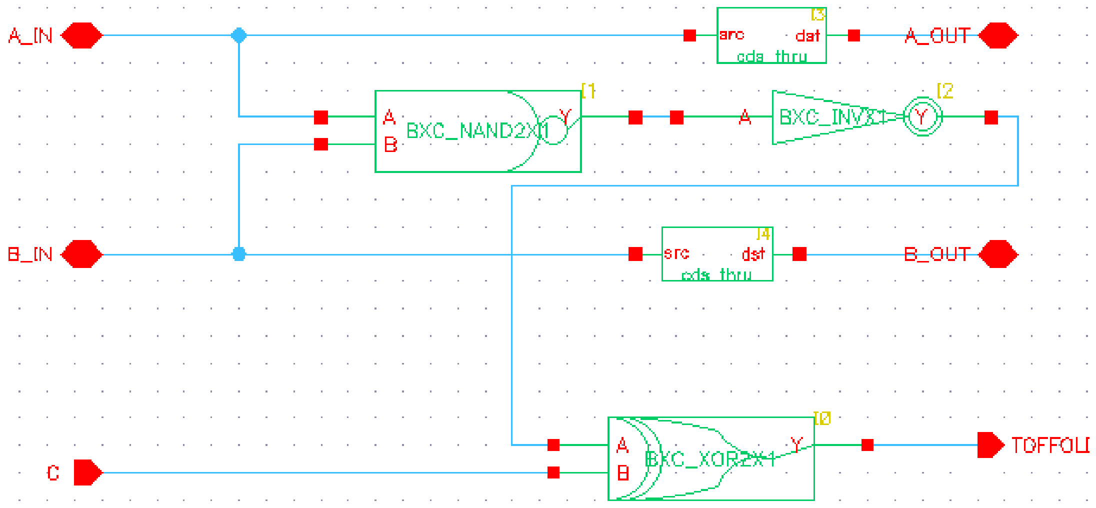


Figure 5.16: Toffoli Schematic

The Toffoli gate is another quantum gate which is also known as Controlled Controlled NOT Gate (CCNOT). The two control qubit, A and B, are ANDed together and then XORed with the third input C. Unless the two control signals have a value of 1, the Toffoli output is not get inverted. The two control inputs A and B are input-output pin due to replicating the pins to allow same input and output count. In Figure 5.16, the schematic of the Toffoli gate can be seen. In Figure 5.17, the symbol of the Toffoli gate is present.

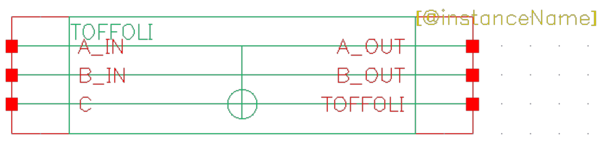


Figure 5.17: Toffoli Symbol

The layout of the Toffoli gate is shown in Figure 5.18.

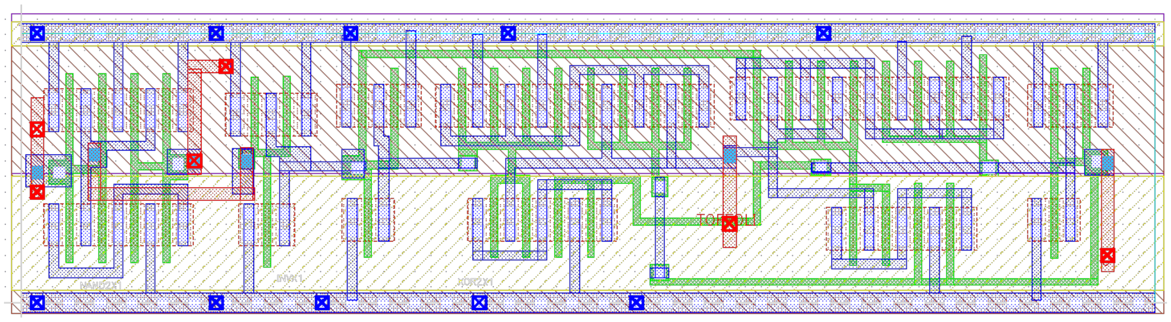


Figure 5.18: Toffoli Layout

### 5.0.7 Reversible Full Adder

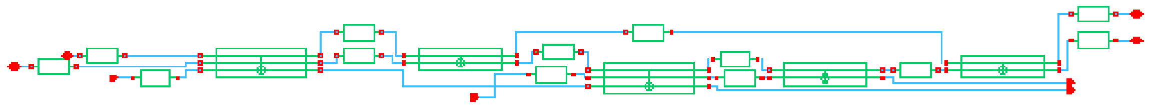


Figure 5.19: Reversible Full Adder Schematic

The Toffoli and CNOT gates can be combined to create a reversible Full Adder. In Figure 5.19, the schematic of the Reversible Full Adder can be seen. In Figure 5.20, the symbol of the Reversible Full Adder is present.

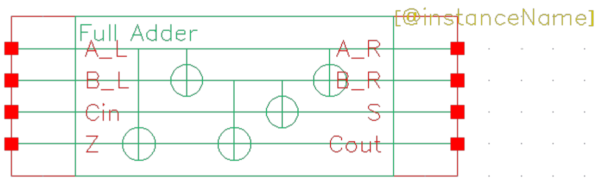


Figure 5.20: Reversible Full Adder Symbol

The layout of the Reversible Full Adder is shown in Figure 5.21.

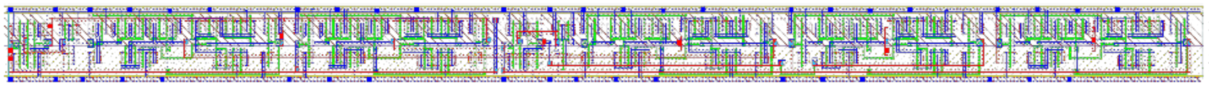


Figure 5.21: Reversible Full Adder Layout

### 5.0.8 Fredkin Gate

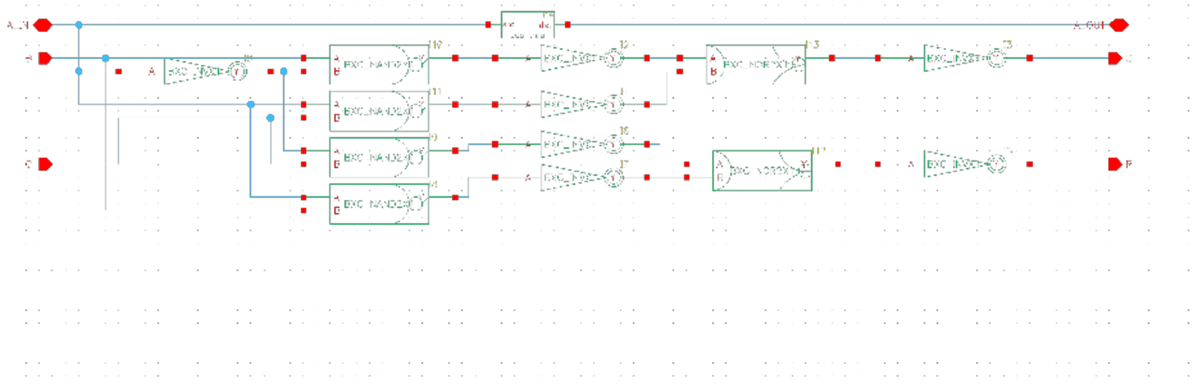


Figure 5.22: Fredkin Schematic

The Fredkin gate is also known as a Controlled Swap Gate. Through a controlled signal, the Fredkin Gate determines when to swap the inputs. This hierarchy is a quantum gate. In Figure 5.22, the schematic of the Fredkin gate can be seen. In Figure 5.23, the symbol of the Fredkin Gate is present.

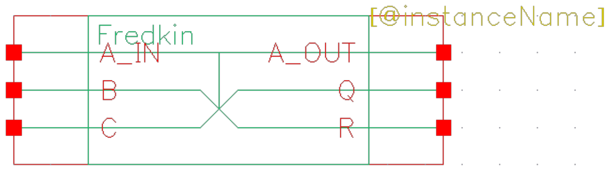


Figure 5.23: Toffoli Symbol

The layout of the Fredkin gate is shown in Figure 5.24.

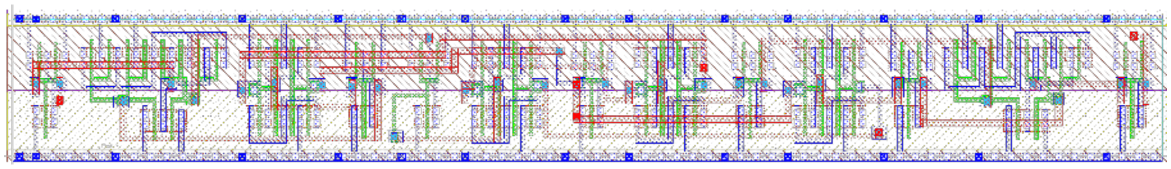


Figure 5.24: Fredkin Layout

### 5.0.9 Reversible D-Latch

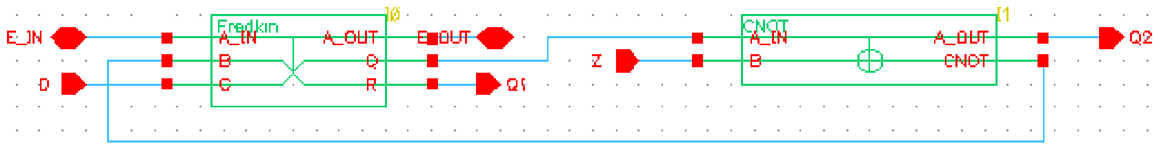


Figure 5.25: Reversible D-Latch Schematic

The combination of Fredkin gate and a CNOT gate allows a Reversible D-Latch to be created. The Reversible D-Latch follows the exact pattern of an input D as long as the clock is high or the signal is enabled. The output stays at the edge where the last position of the D input was once the negative edge is present. In Figure 5.25, the schematic of the D-Latch can be seen. In Figure 5.26, the symbol of the Reversible D-Latch is present. According to [8] and [9], combining a Fredkin gate and a CNOT gate should have created a D-Flip Flop. However after testing the implementation, the design demonstrated a behavior similar to a D-Latch rather than a D-Flip Flop. The simulation result of this block is illustrated in Chapter 6 Section 6.0.9.

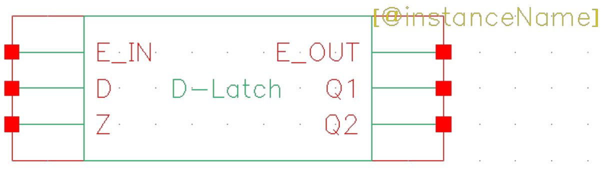


Figure 5.26: Reversible D-Latch Symbol

The layout of the Reversible D-Latch is shown in Figure 5.27.

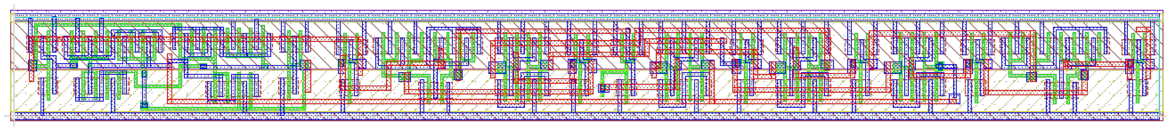


Figure 5.27: Reversible D-Latch Layout

### 5.0.10 Reversible 32-bit Register

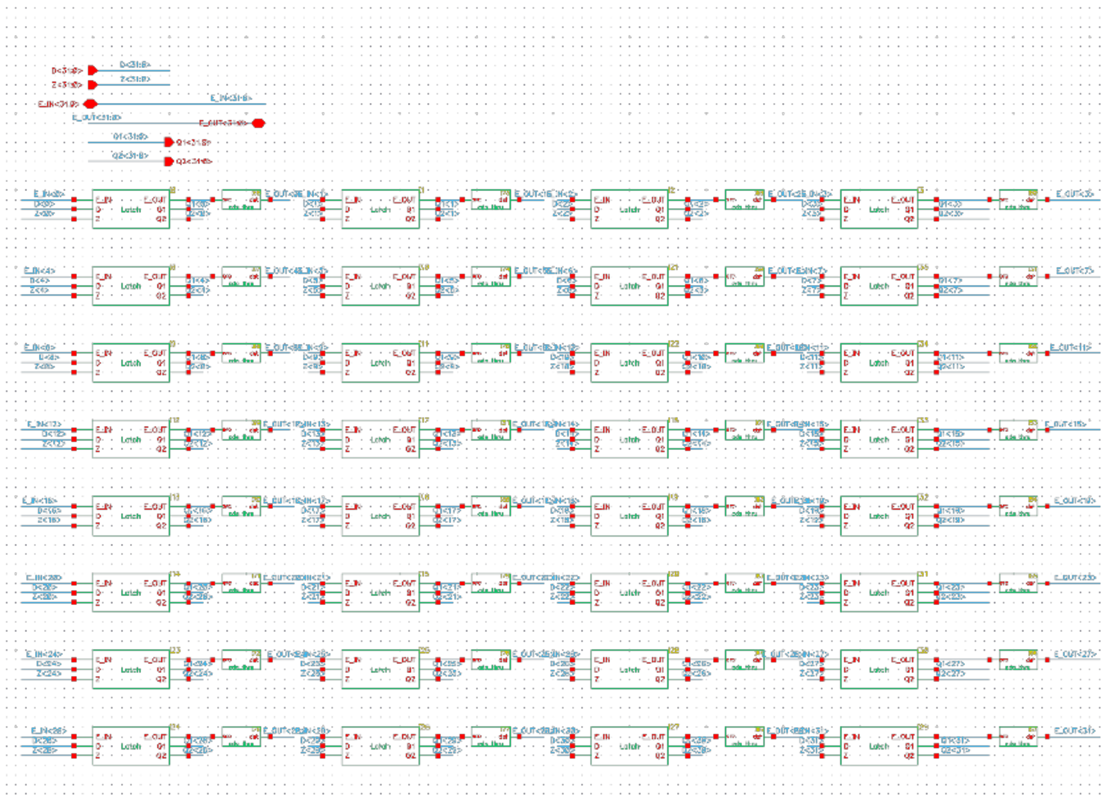


Figure 5.28: Reversible 32-bit Register Schematic

The Reversible D-Latch was used to create a Reversible 32-bit Register. In Figure 5.28, the schematic of the Reversible 32-bit Register can be seen. In Figure 5.29, the symbol of the Reversible 32-bit Register is present.

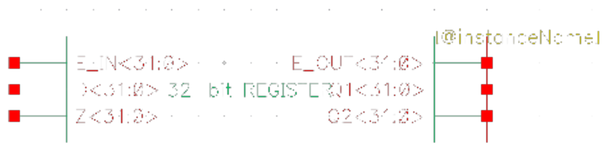


Figure 5.29: Reversible 32-bit Register Symbol

The layout of the Reversible 32-bit Register is shown in Figure 5.30.

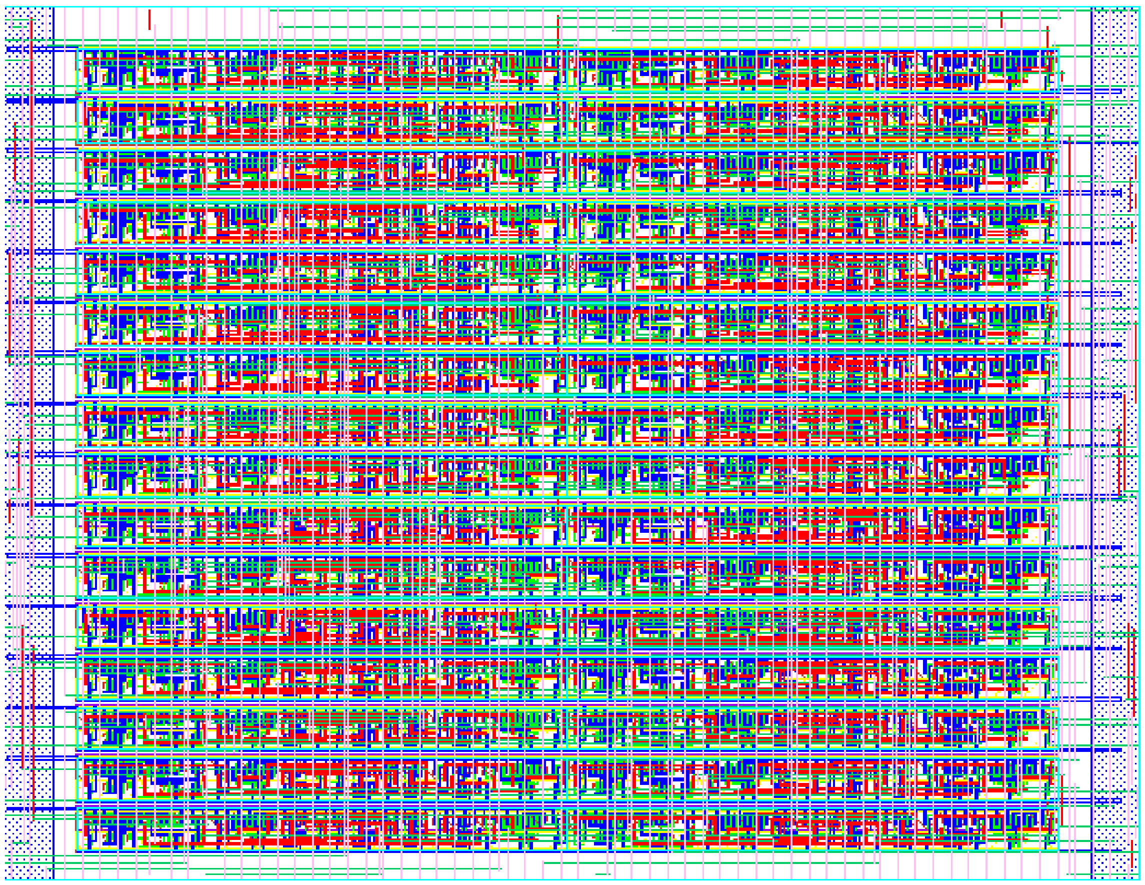


Figure 5.30: Reversible 32-bit Register Layout

### 5.0.11 Reversible 32-bit Carry Look Ahead Adder

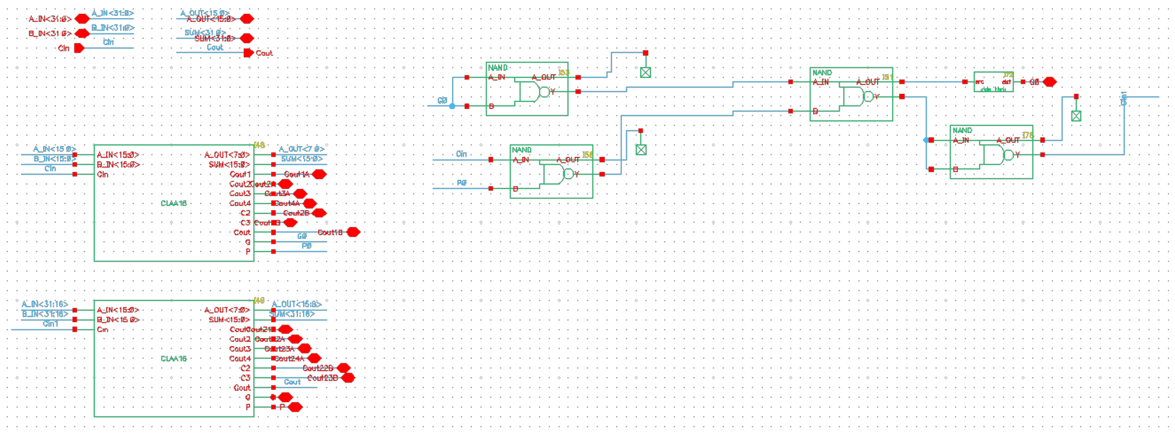


Figure 5.31: Reversible 32-bit Carry Look Ahead Adder Schematic

The Carry Look Ahead Adder performs addition while separately calculating the carry for the next adder. This speeds up the addition and the delay caused by the carry out signal is diminished. In Figure 5.31, the schematic of the Reversible 32-bit Carry Look Ahead Adder can be seen. As shown, the Reversible 32-bit Carry Look Ahead Adder is composed of other cells, which are the Reversible 16-bit Carry Look Ahead Adder and the Reversible 4 bit Carry Look Ahead cells. The Reversible 16-bit Carry Look Ahead Adder is also made out of another cell named Reversible 4-bit Carry Look Ahead Adder. These are illustrated respectively in Figure 5.36, Figure 5.34 and Figure 5.34. The reason for designing the Carry Look Ahead Adder in hierarchy base is to reduce complexity of the design. The smaller adders were also used in the multiplier. The symbol of 16-bit Carry Look Ahead Adder is given in Figure 5.37, the 4-bit Carry Look Ahead Unit Symbol is shown as Figure 5.35 and the 4-bit 32-bit Carry Look Ahead Adder Symbol is given as Figure 5.33. In Figure 5.32, the symbol of the 32-bit Carry Look Ahead Adder is present. From these schematics, it can be observed that in order to equate the input and output pin numbers, some signals were randomly added in some hierarchy, and removed by not connecting in others.

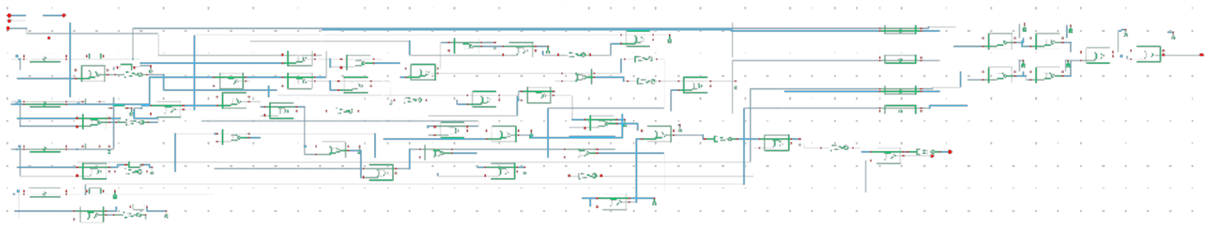


Figure 5.32: Reversible 4-bit Carry Look Ahead Adder Schematic

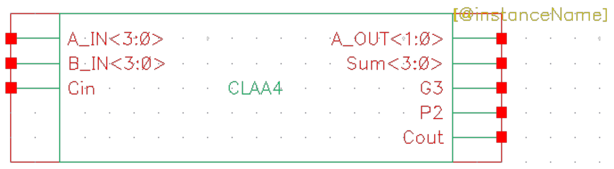


Figure 5.33: Reversible 4-bit Carry Look Ahead Adder Symbol

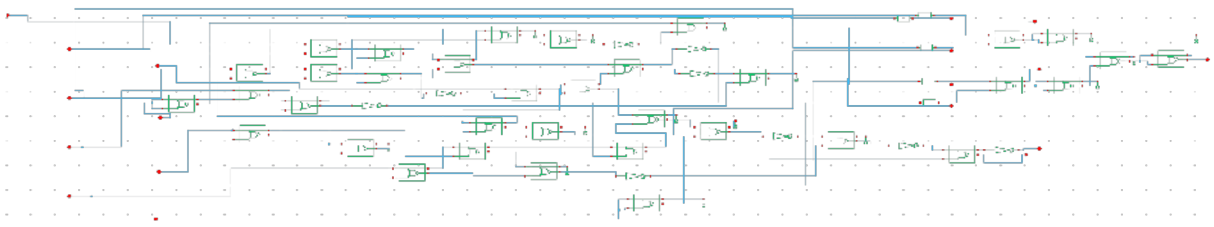


Figure 5.34: Reversible 4-bit Carry Look Partial Product Unit Schematic

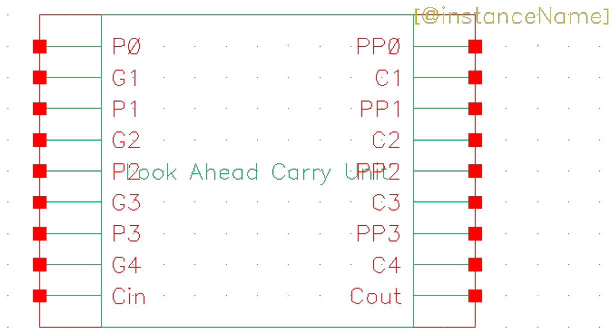


Figure 5.35: Reversible 4-bit Carry Look Ahead Partial Product Unit Symbol

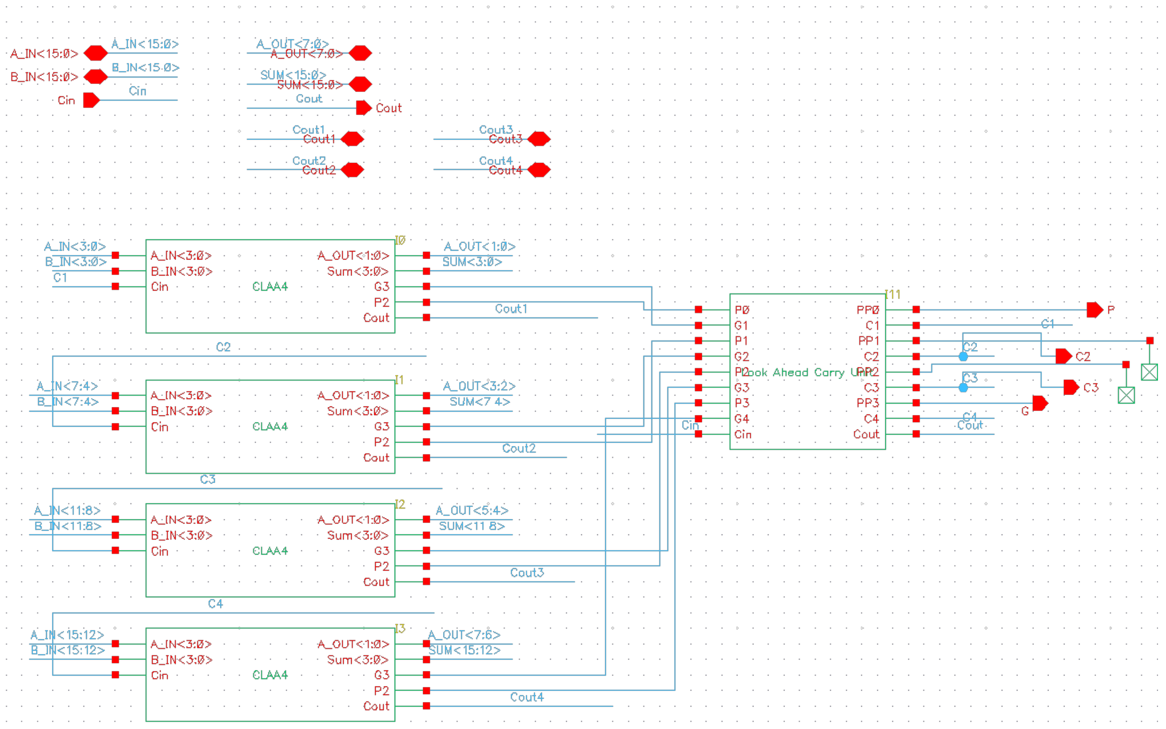


Figure 5.36: Reversible 16-bit Carry Look Ahead Adder Schematic

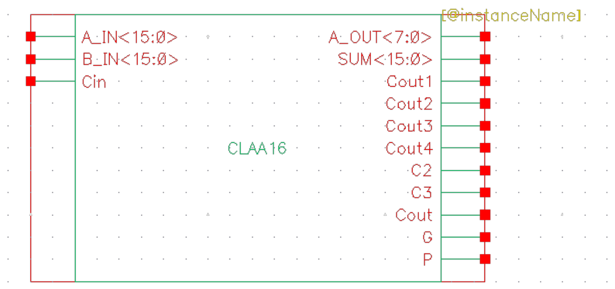


Figure 5.37: Reversible 16-bit Carry Look Ahead Adder Symbol

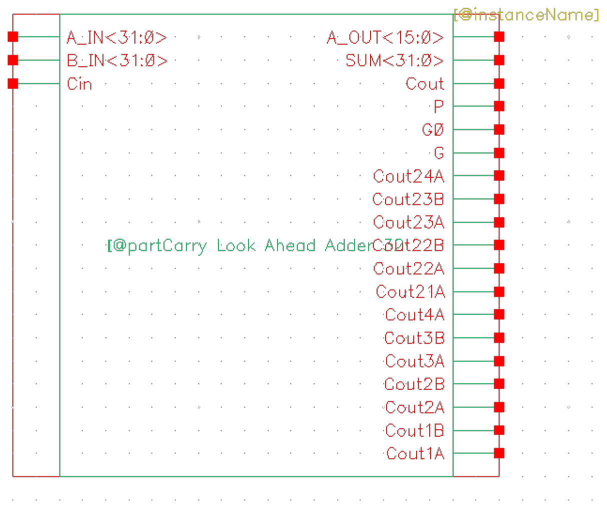


Figure 5.38: Reversible 32-bit Carry Look Ahead Adder Symbol

The layout of the Reversible 32-bit Carry Look Ahead Adder is shown in Figure 5.39. As the tools and machines that were available for this work at Rochester Institute of Technology did not have enough memory to auto place this large adder, the tools gave up during auto placement and left a large amount of space. Thus, all of the components in the Reversible 32-bit Carry Look Ahead Adder were placed by hand.

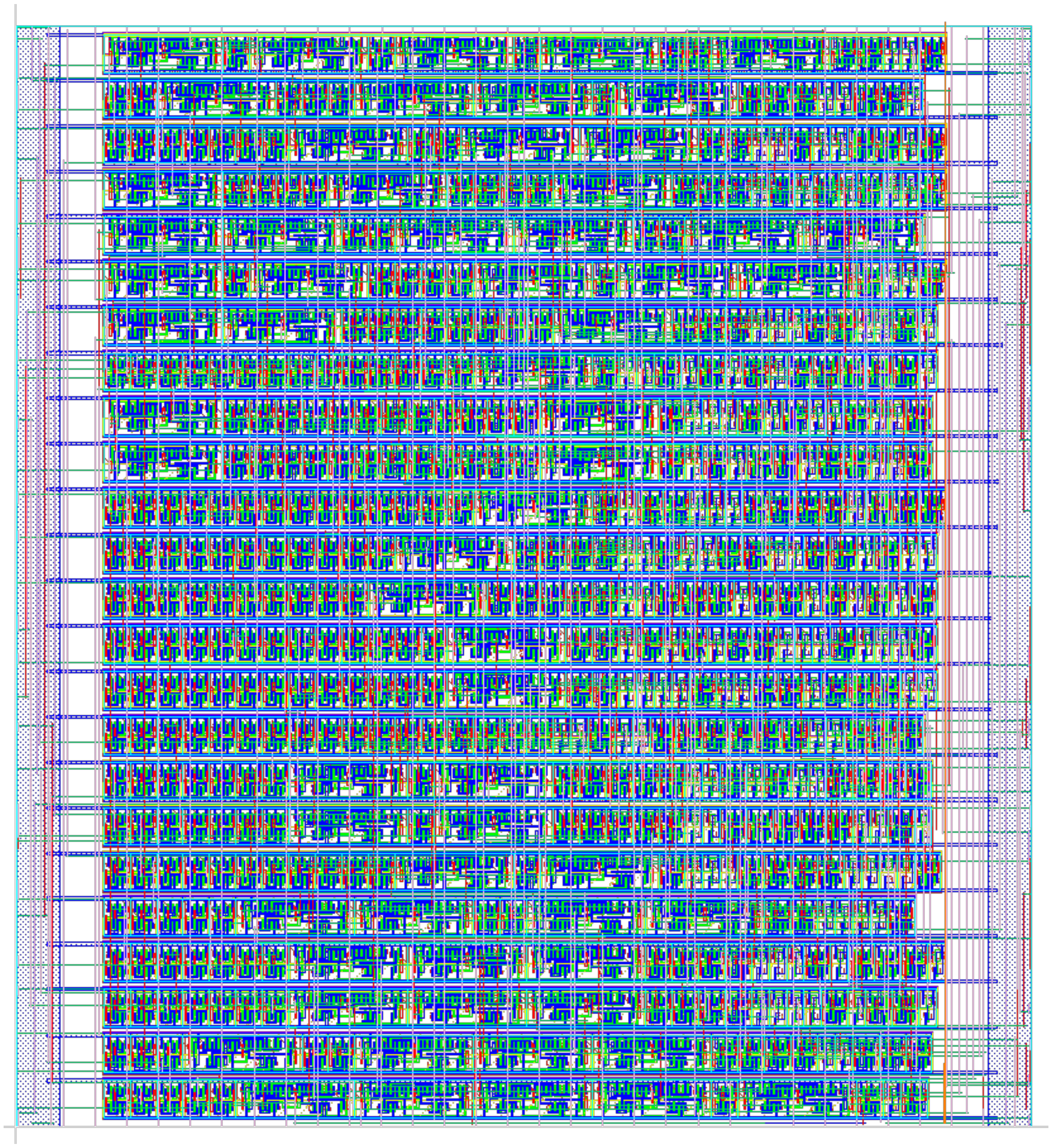


Figure 5.39: Reversible 32-bit Carry Look Ahead Adder Layout

### 5.0.12 Reversible 16-bit Multiplier

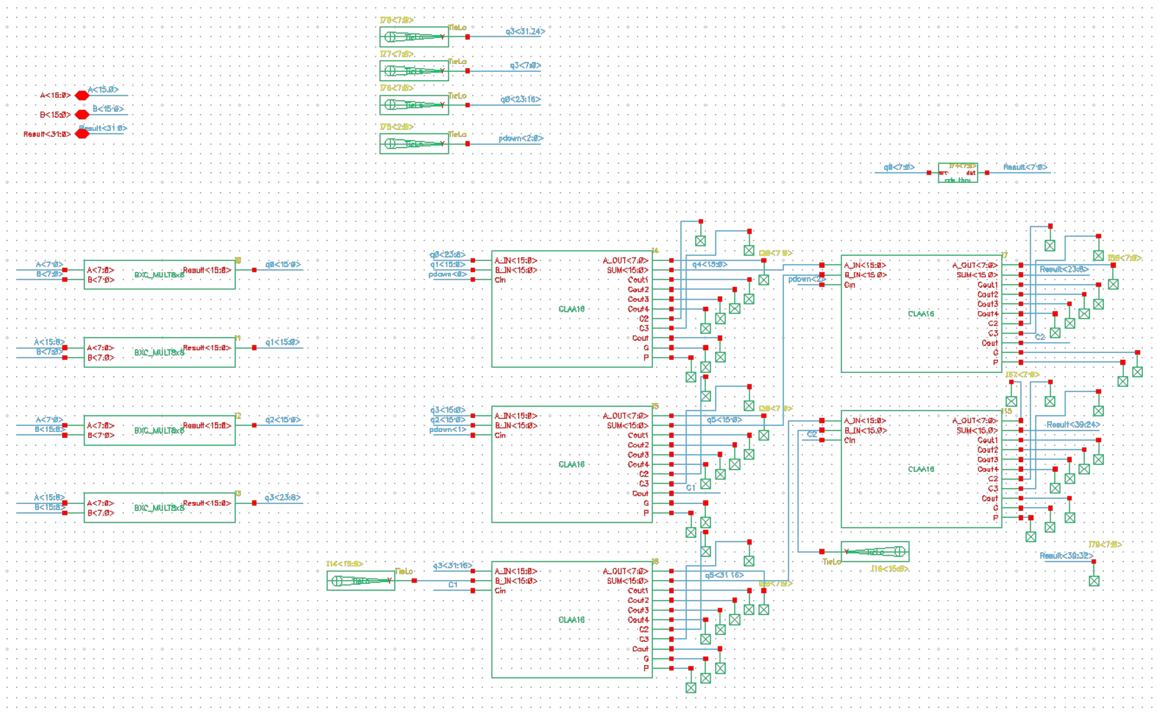


Figure 5.40: Reversible16-bit Multiplier Schematic

The Reversible 16-bit multiplier was designed using the Vedic technique by concatenating smaller multipliers together. First a Reversible 2-bit Multiplier was designed, which was used to create a Reversible 4-bit Multiplier, which was used to design an Reversible 8-bit Multiplier, and the Reversible 8-bit Multiplier was used while creating the final Reversible 16-bit Multiplier. The schematic of the Reversible 16-bit Multiplier can be seen in Figure 5.40, whereas the Reversible 8-bit Multiplier is present in Figure 5.41, Reversible 4-bit Multiplier in Figure 5.43 and Reversible 2-bit Multiplier in Figure 5.43. The symbol of Reversible 16-bit Multiplier is indicated by Figure 5.47. The Reversible 8-bit,4-bit and 2-bit Multiplier Symbols are present in Figure 5.42, Figure 5.44, and Figure 5.46 respectively.

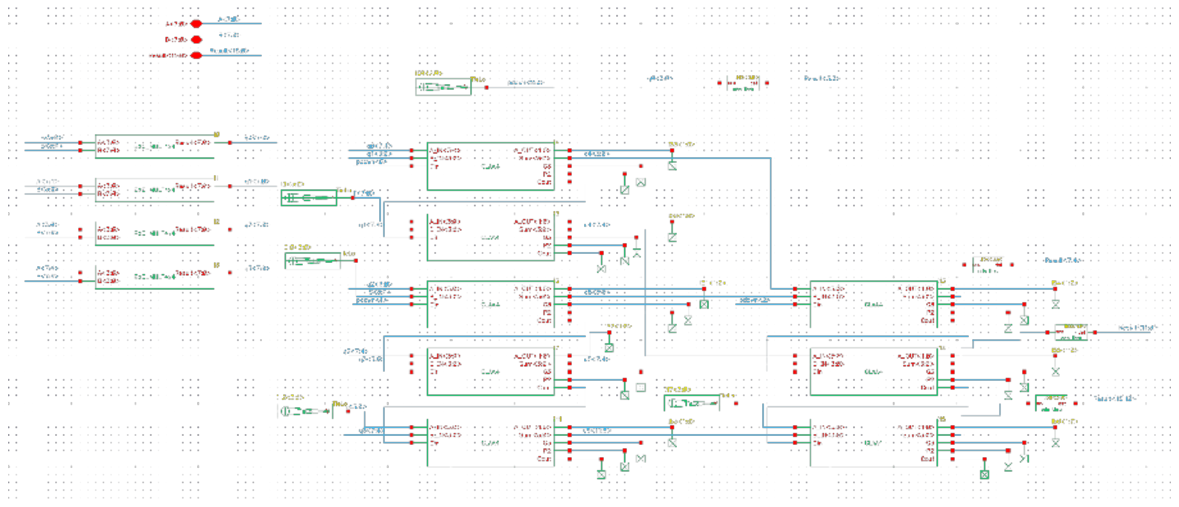


Figure 5.41: Reversible 8-bit Multiplier Schematic

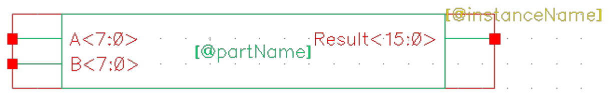


Figure 5.42: Reversible 8-bit Multiplier Symbol

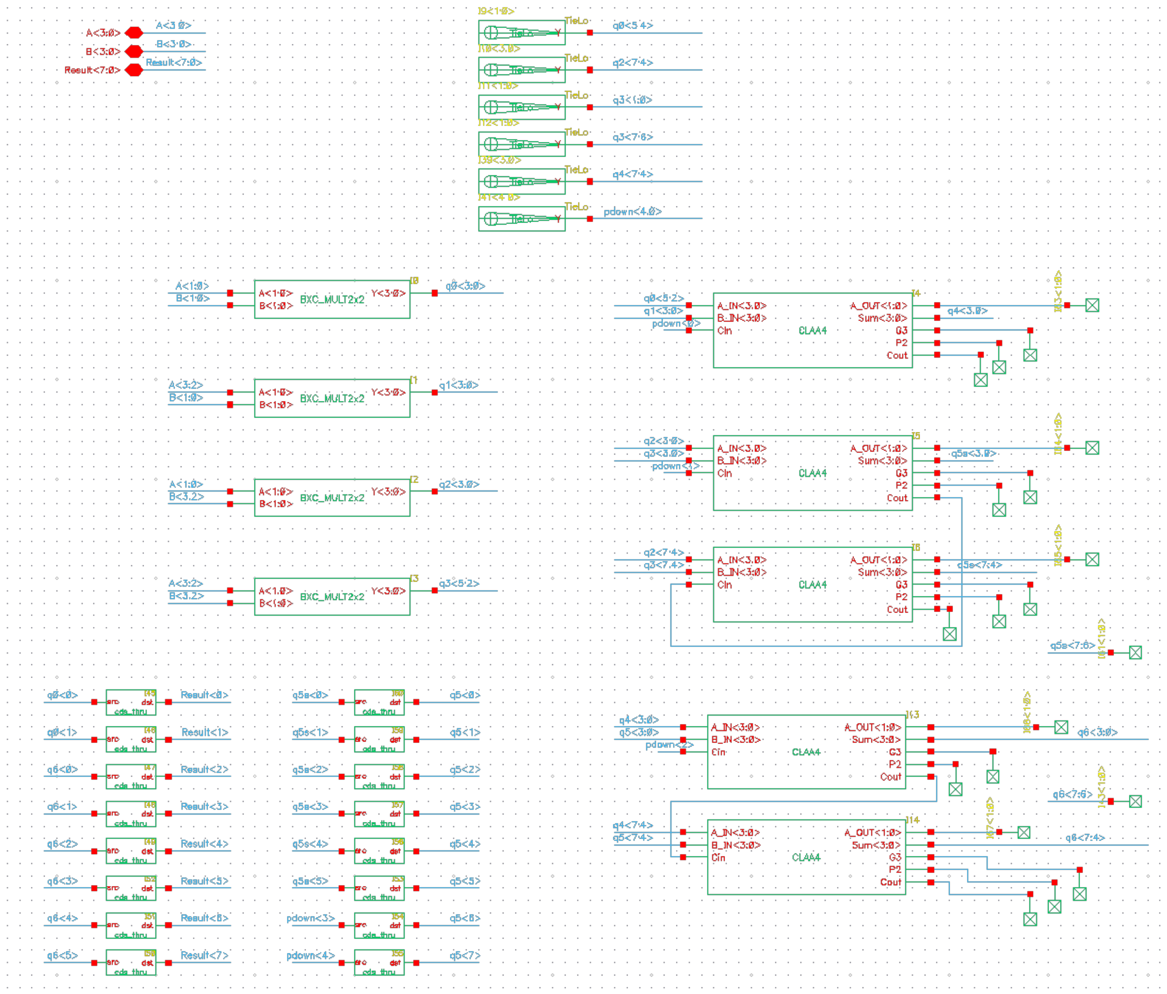


Figure 5.43: Reversible 4-bit Multiplier Schematic

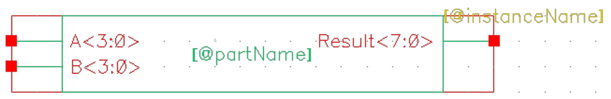


Figure 5.44: Reversible 4-bit Multiplier Symbol

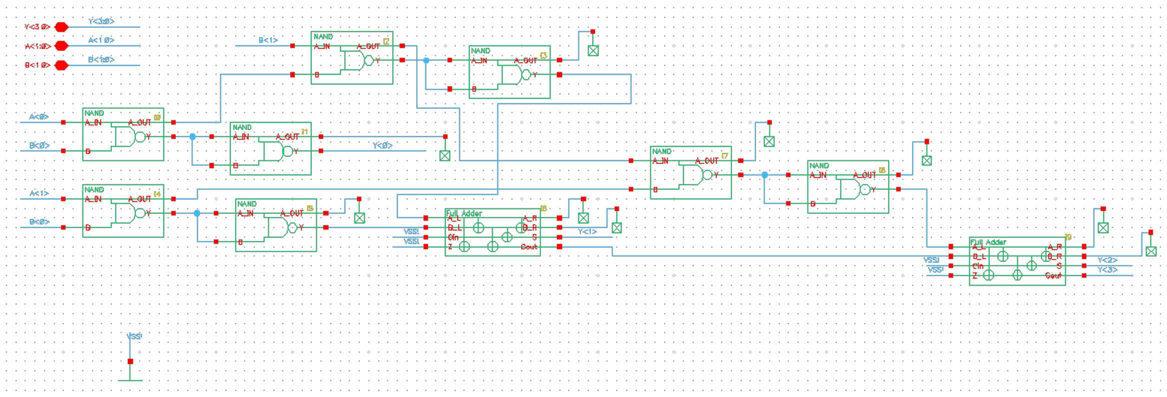


Figure 5.45: Reversible 2-bit Multiplier Schematic

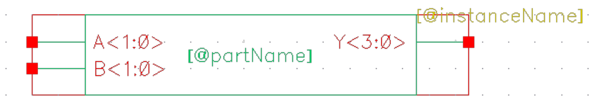


Figure 5.46: Reversible 2-bit Multiplier Symbol

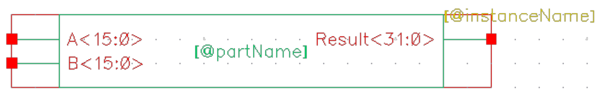


Figure 5.47: Reversible 16-bit Multiplier Symbol

After verifying the function of these designs, the layout of the Reversible 16-bit Multiplier was made. The layout of the Reversible 16-bit Multiplier can be seen in Figure 5.48. Even though the components in the Reversible 32-bit Carry Look Ahead Adder were placed by hand, given the size of the multiplier, the components in the multiplier were not placed by hand. Instead, the design generated through autoplacement with large amount of spacing was used as it would have taken couple of weeks to place each component by hand for the multiplier.

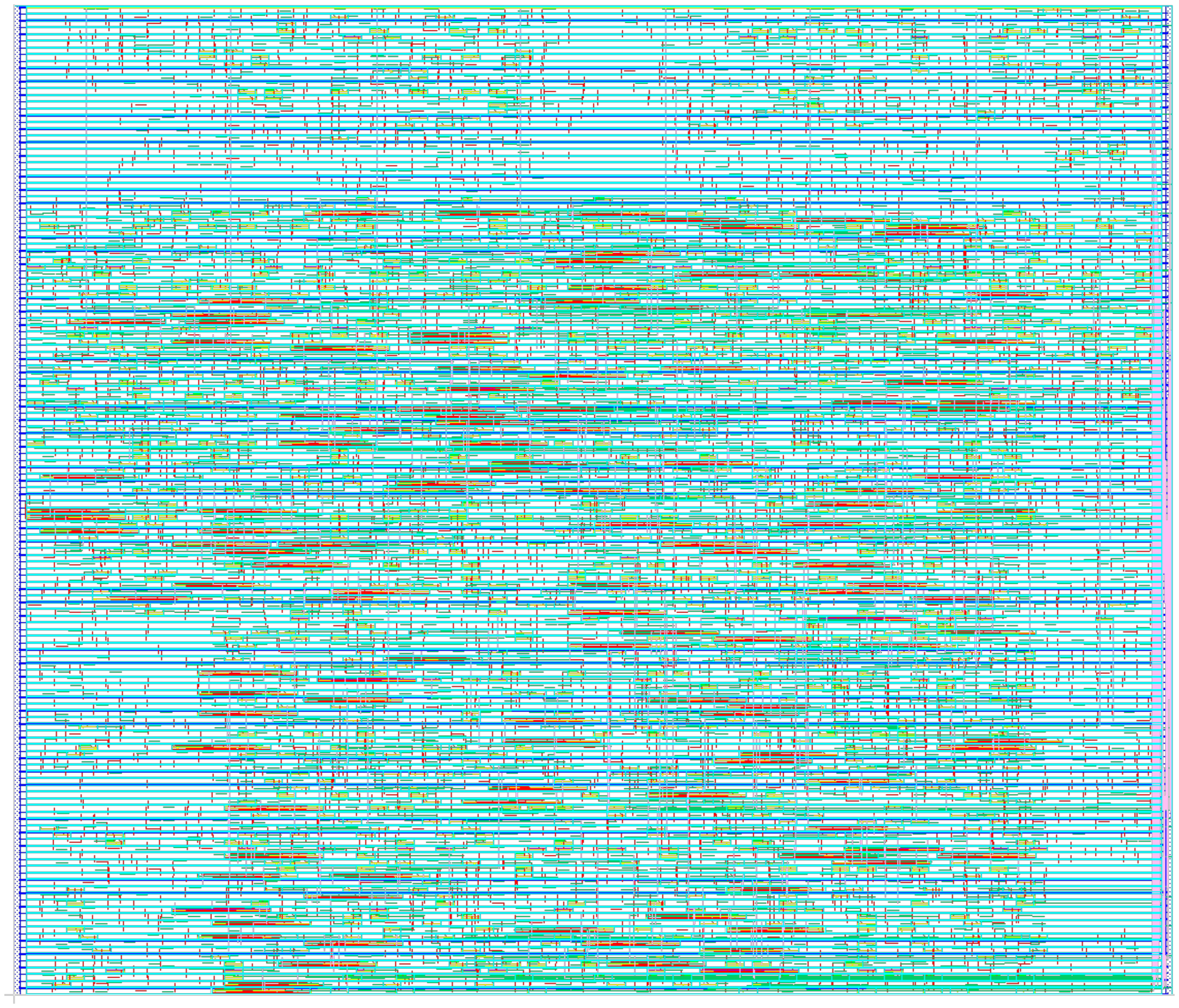


Figure 5.48: Reversible 16-bit Multiplier Layout

# Chapter 6

# Testing Components

Each designed cell and design were verified functionally and behaviorally before creating the layouts.The pre-layout testing allowed for the verification of the schematic. Single cell components were tested by adding a 100 fF capacitor on the output pin to be tested. Multi-cell designs were tested through coding individual testbenches. The Verilog testbench code can be further seen in the Appendix A.

### 6.0.1 Inverter Test

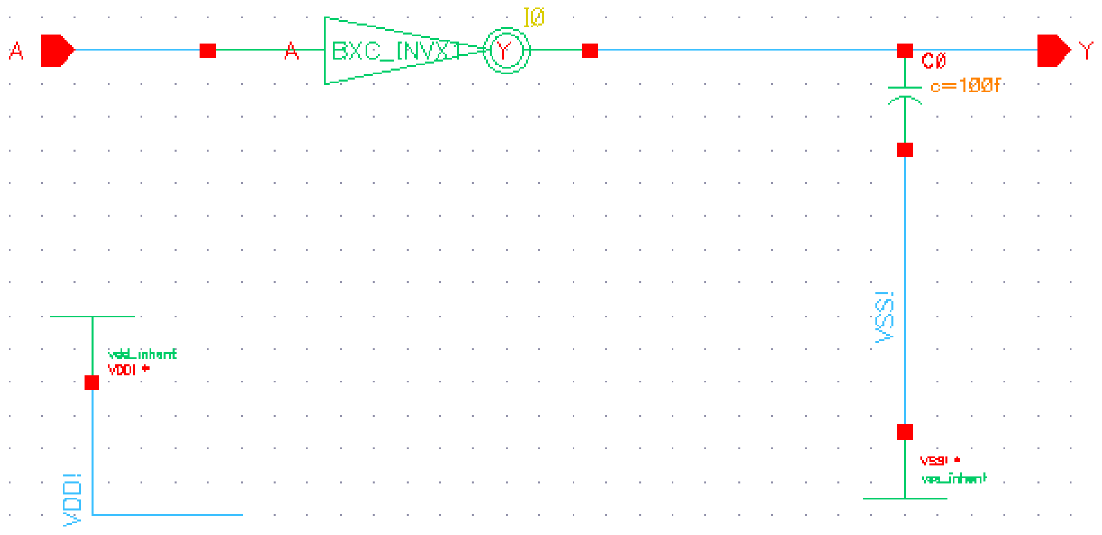


Figure 6.1: Inverter Test Schematic

The inverter test schematic setup can be seen in Figure 6.1. By adding appropriate input and output pins to the symbol created through schematic, and adding a load capacitor along with power and ground, the test schematic was created. The results acquired by running this schematic can be seen in Figure 6.2. It can be seen that the inverter inverts the input signal A to be the output signal Y.

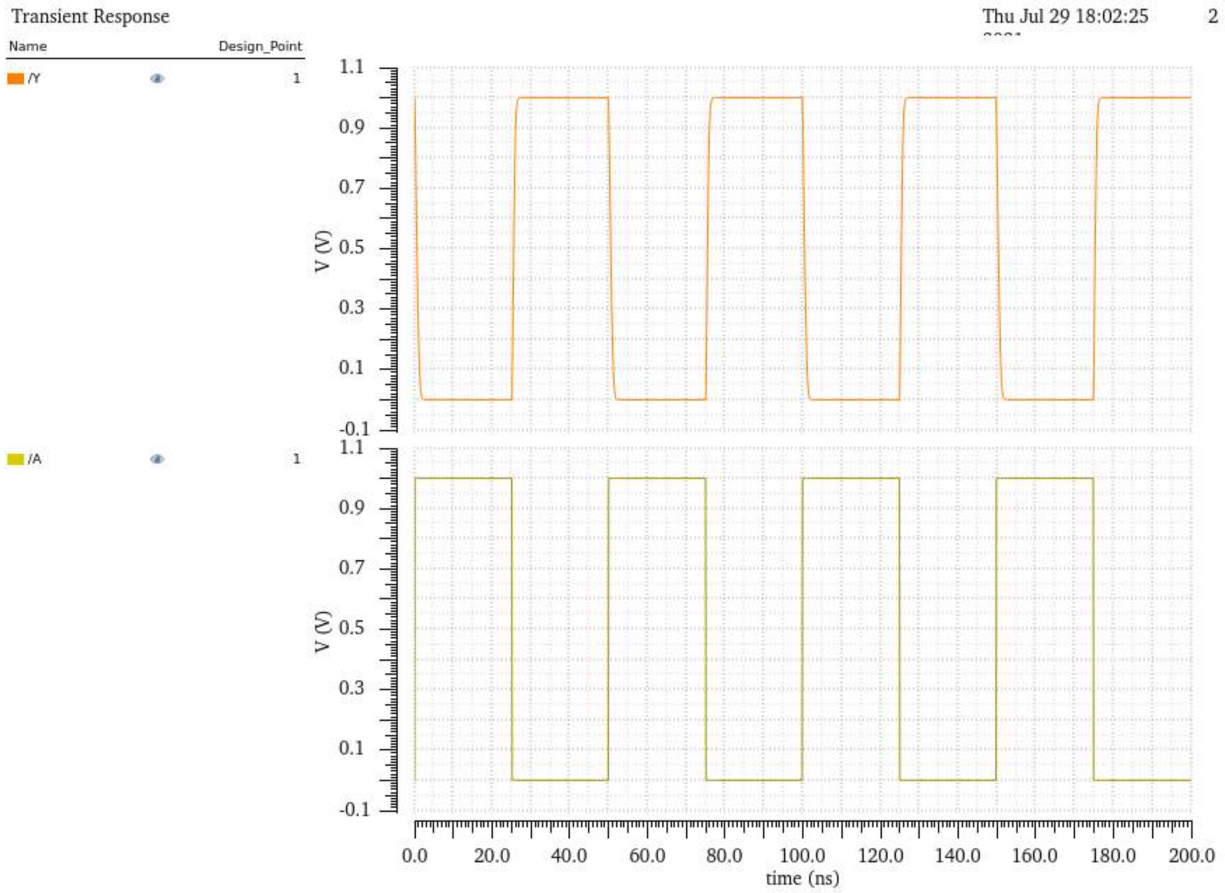


Figure 6.2: Inverter Simulation

### 6.0.2 Reversible NAND Test

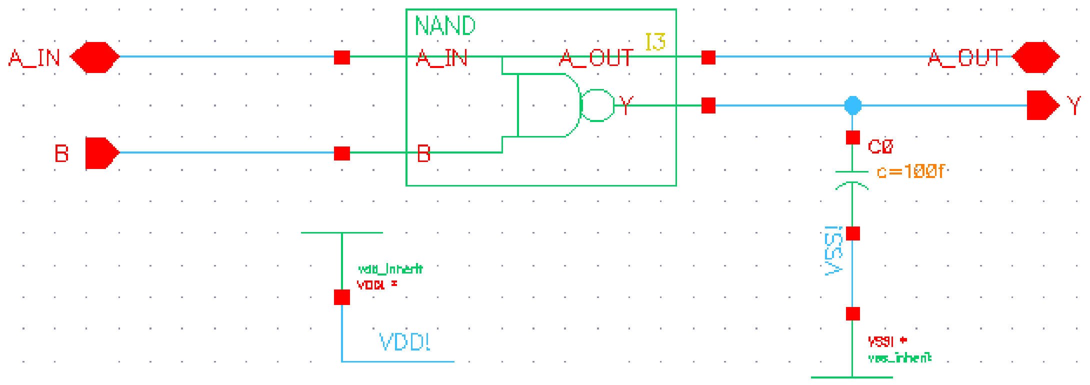


Figure 6.3: Reversible NAND Test Schematic

The Reversible NAND test schematic setup can be seen in Figure 6.3. By adding appropriate input, output and input-output pins to the symbol created through schematic, and including a load capacitor along with power and ground nets, the test schematic was created. The results acquired by running test schematic for reversible Reversible NAND can be seen in Figure 6.4. As shown, the output of the Reversible NAND gate is low only when both of the inputs are high and the A_IN and A_OUT demonstrate an equivalent signal property. This applies to everys

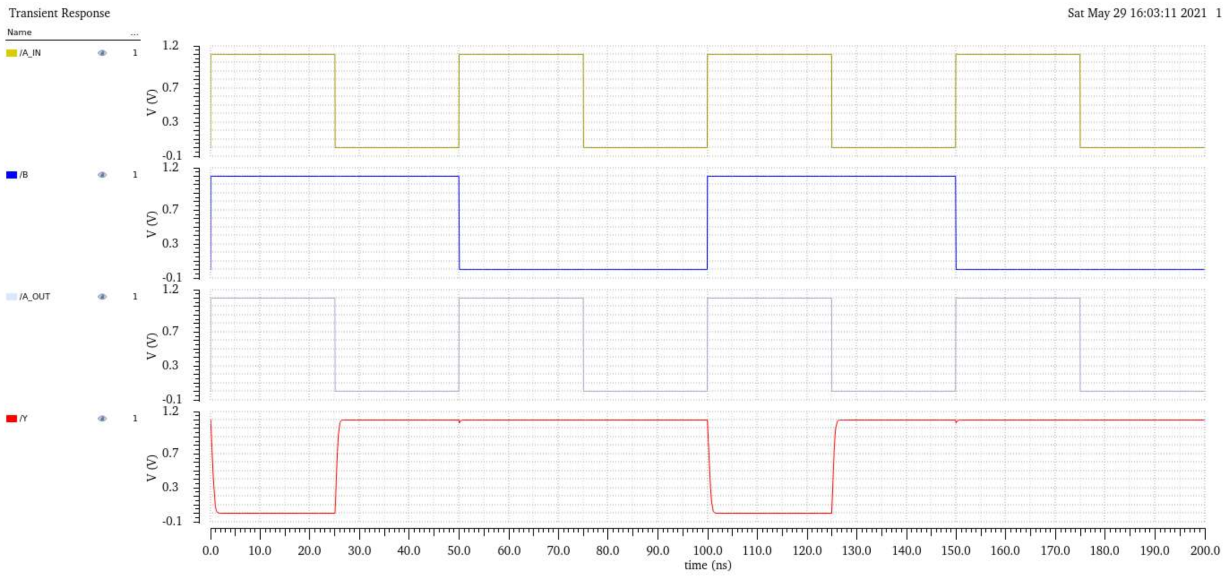


Figure 6.4: Reversible NAND Simulation

### 6.0.3 Reversible NOR Test

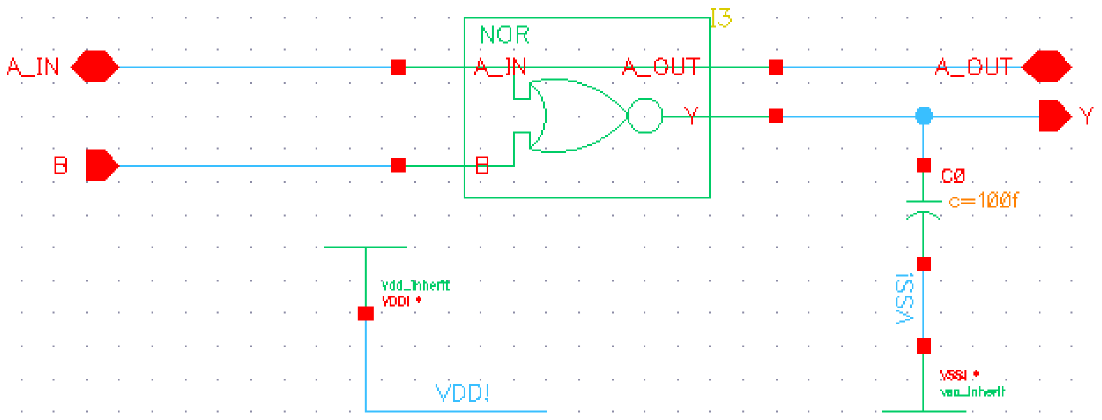


Figure 6.5: Reversible NOR Test Schematic

The Reversible NOR test schematic setup can be seen in Figure 6.5. Load capacitor was added to this test schematic just like the previous test setups. The results acquired by running test schematic for Reversible NOR can be seen in Figure 6.6. The NOR signal turns high when both of the input signals are low.

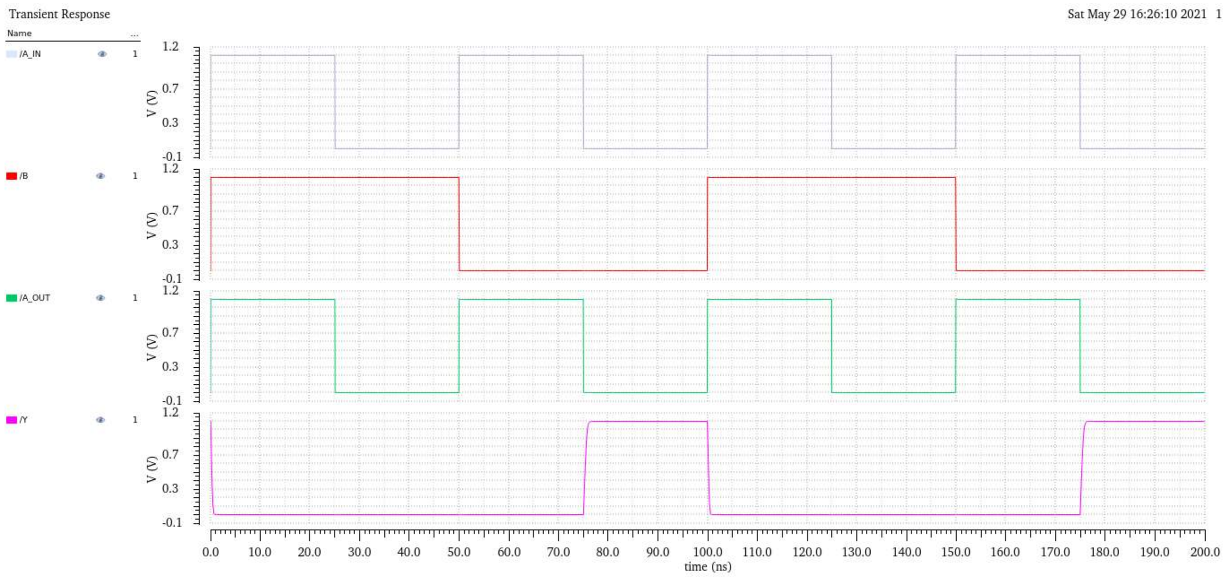


Figure 6.6: Reversible NOR Simulation

### 6.0.4 CNOT Test

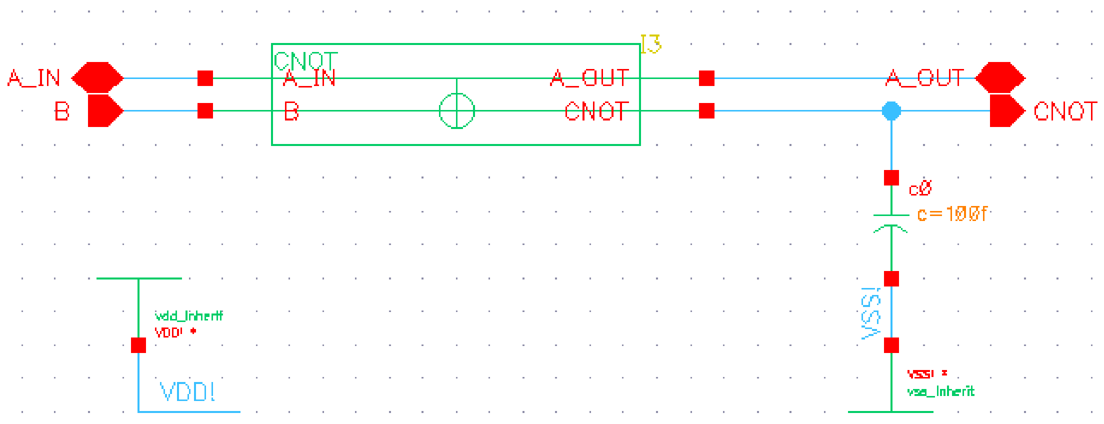


Figure 6.7: CNOT Test Schematic

The CNOT test schematic setup can be seen in Figure 6.7. Appropriate pins were placed along with a load capacitor. The results acquired by running test schematic for reversible CNOT can be seen in Figure 6.8. It can be seen in Figure 6.8 that due to high output impedance, the CNOT gate did not have enough drive to switch the output, which resulted in CNOT signal to be in superposition state, neither 1 nor 0. When the control signal, A, has a value of 1, the other input signal, B, gets inverted.

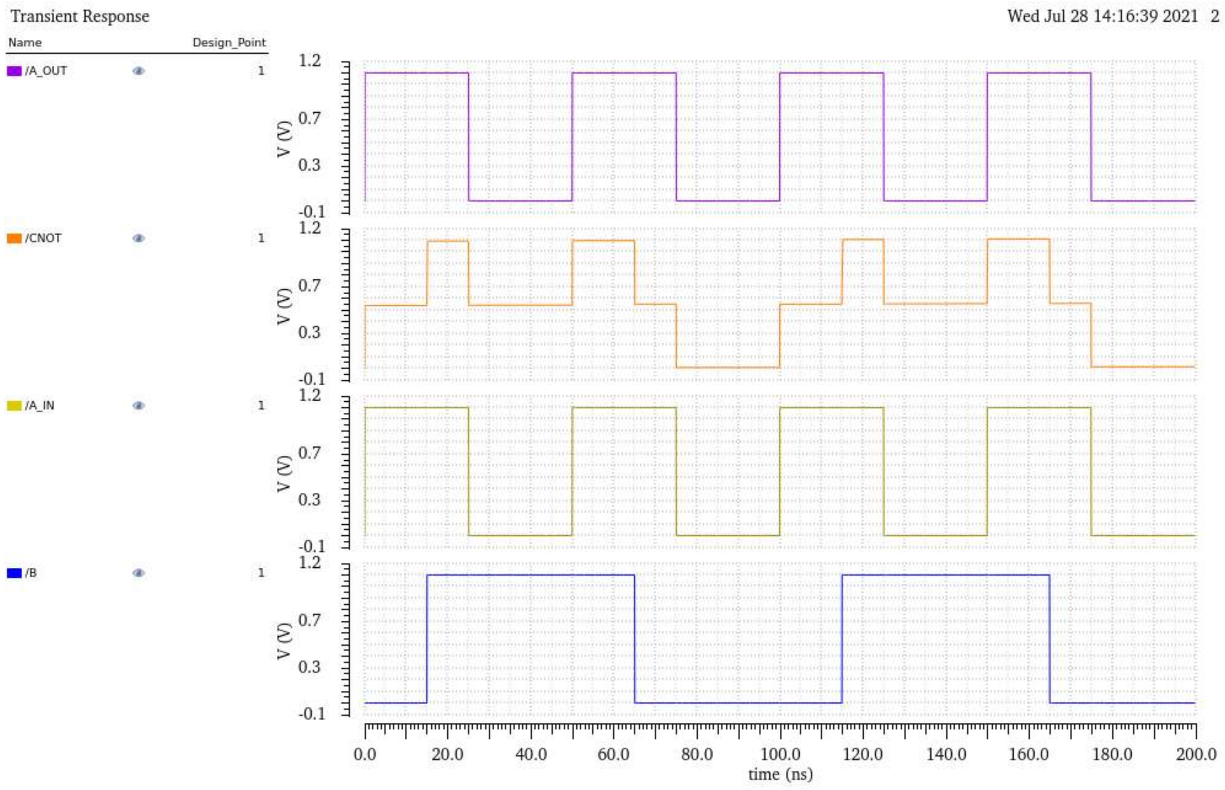


Figure 6.8: CNOT Simulation

### 6.0.5 SWAP Test

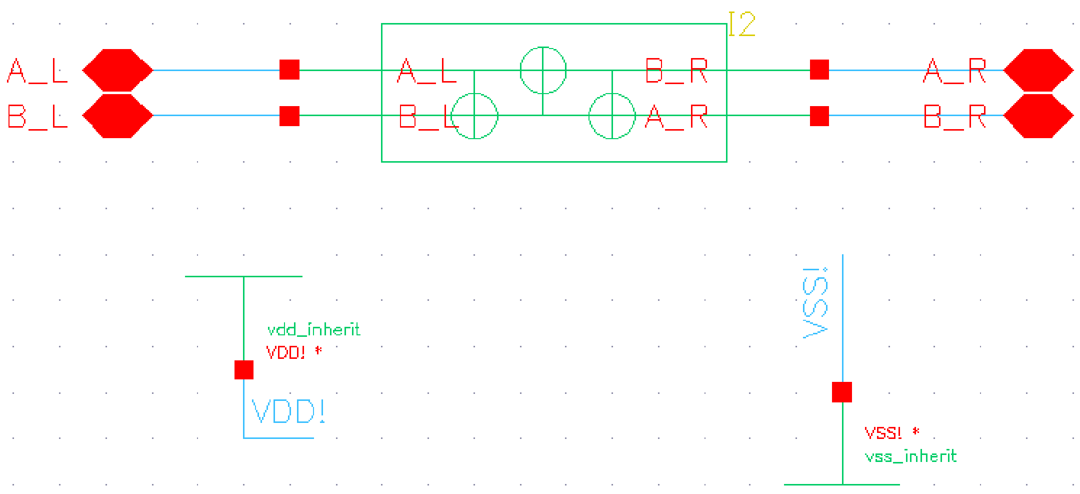


Figure 6.9: SWAP Test Schematic

The SWAP gate test schematic setup can be seen in Figure 6.9. The results acquired by running test schematic for reversible SWAP can be seen in Figure 6.10. It can be seen that the SWAP gate swaps both of the input pins. The glitches are caused by the transition arc. Transition arcs were caused as the signal turns low momentarily before settling down, and transition arcs were perfectly expected to be seen.

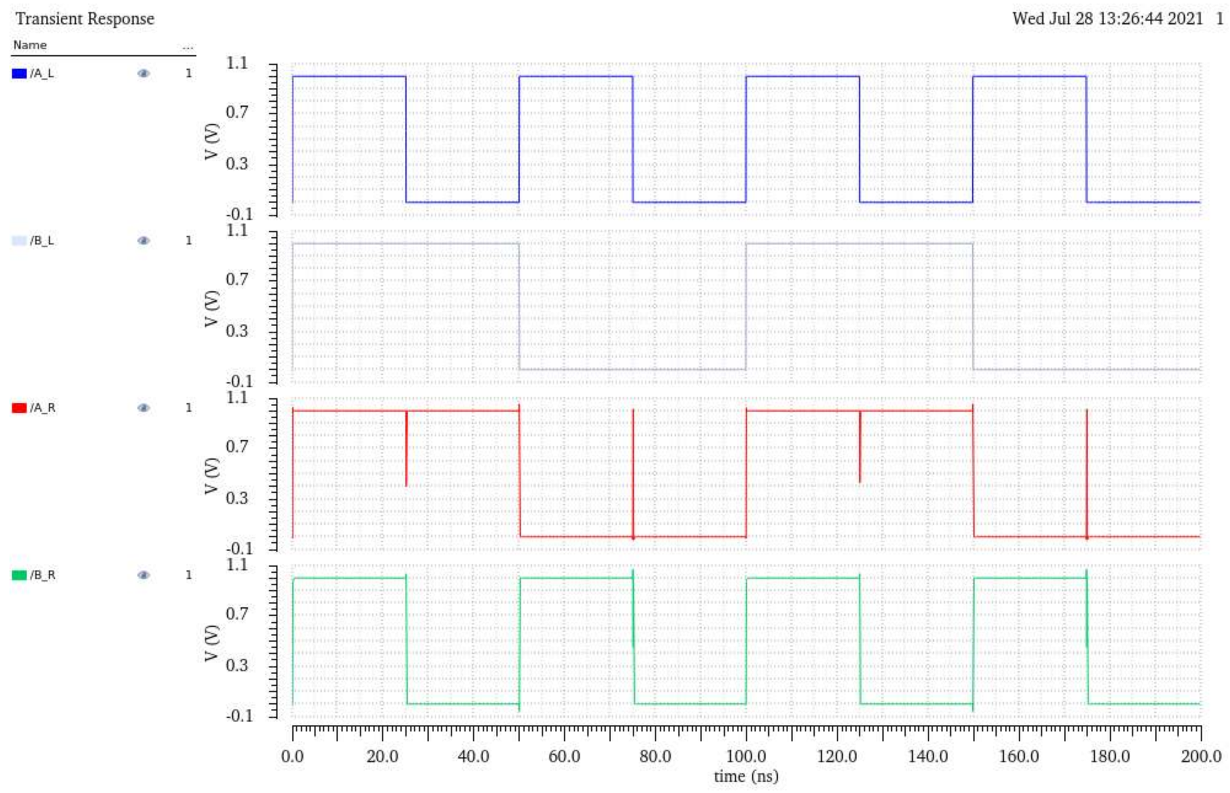


Figure 6.10: SWAP Simulation

### 6.0.6 Toffoli Test

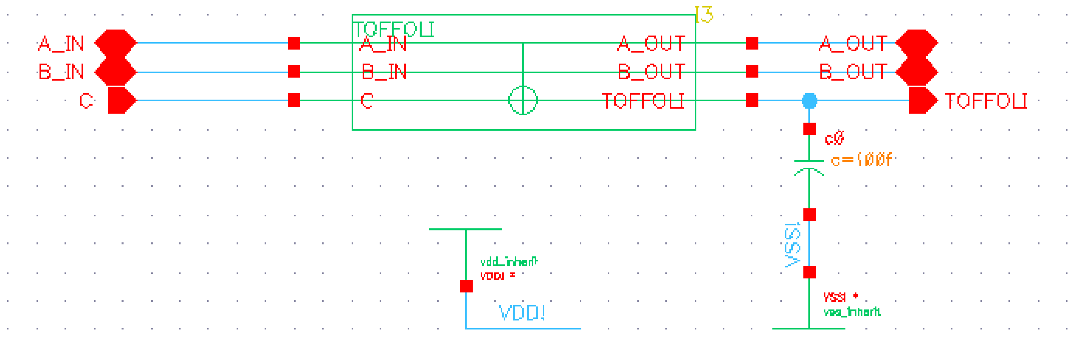


Figure 6.11: Toffoli Test Schematic

The Toffoli test schematic setup can be seen in Figure 6.11. The results acquired by running test schematic for reversible NAND can be seen in Figure 6.12. It can be seen that when both the control signals, A and B, were high, the third input, C, gets inverted.

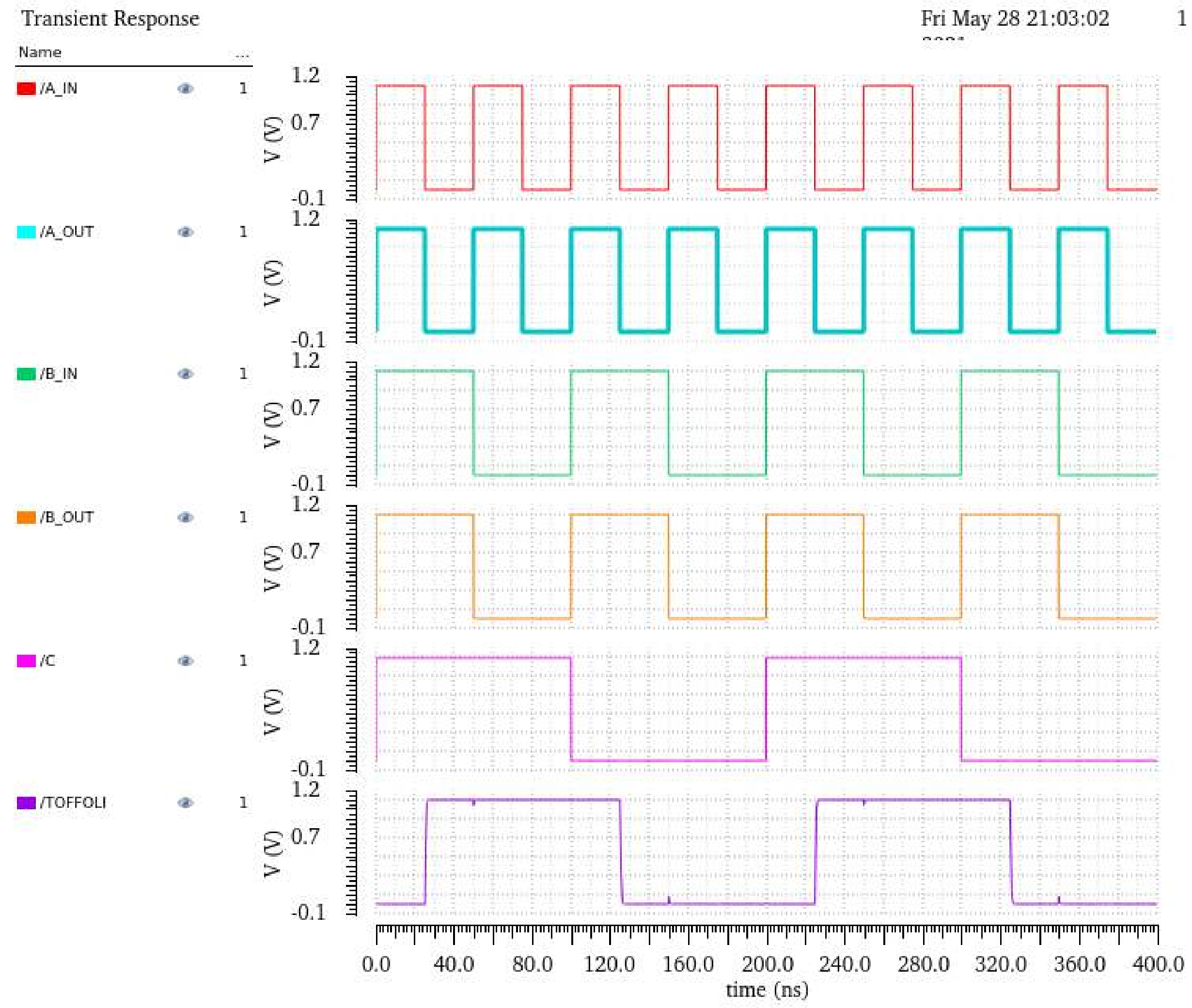


Figure 6.12: Toffoli Simulation

### 6.0.7 Reversible Full Adder Test

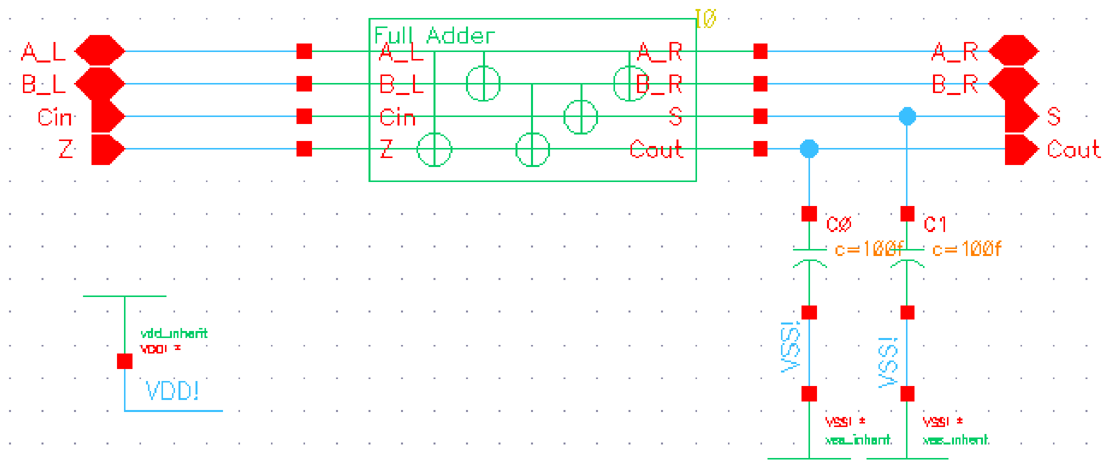


Figure 6.13: Reversible Full Adder Test Schematic

The Reversible Full Adder test schematic setup can be seen in Figure 6.13. The results acquired by running test schematic for the Reversible Full Adder can be seen in Figure 6.14. Once again, the glitches are caused by the transition arc. Transition arcs were perfectly expected to be seen and were caused do to momentary output transition when the signal passes through the transistors.

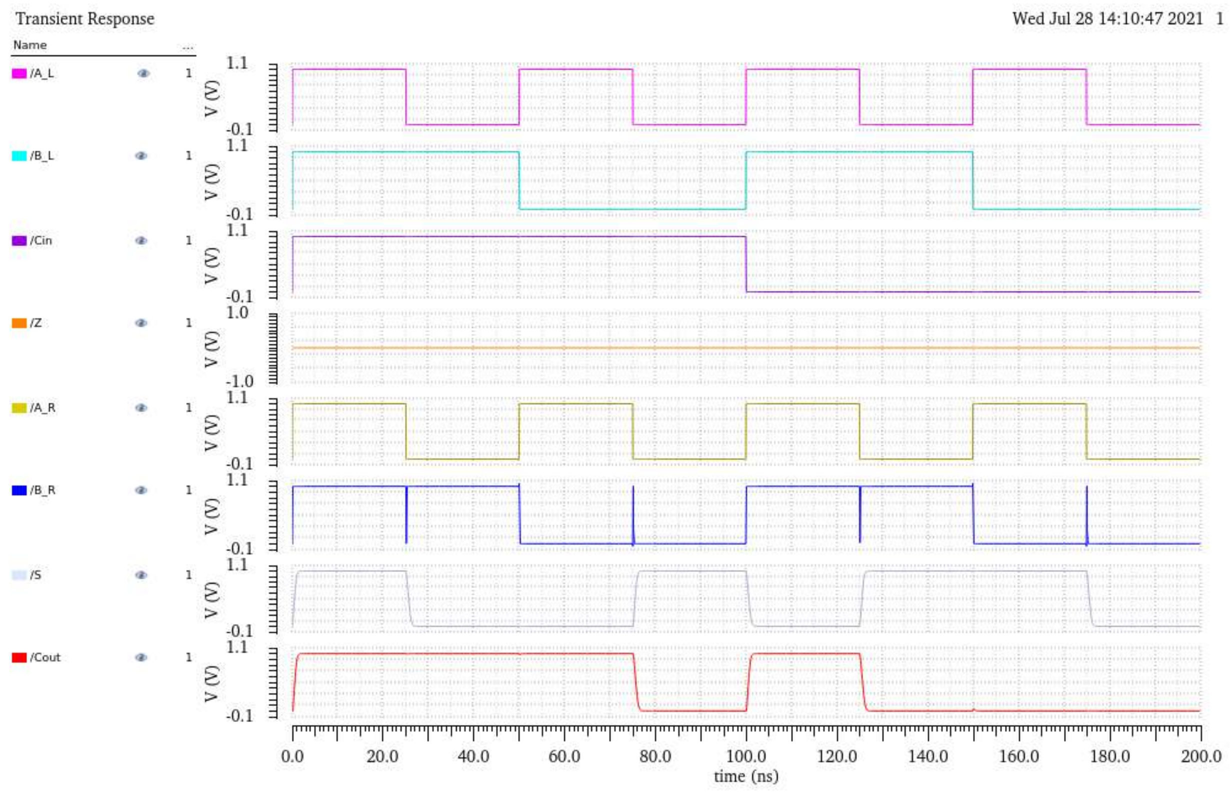


Figure 6.14: Reversible Full Adder Simulation

### 6.0.8 Fredkin Test

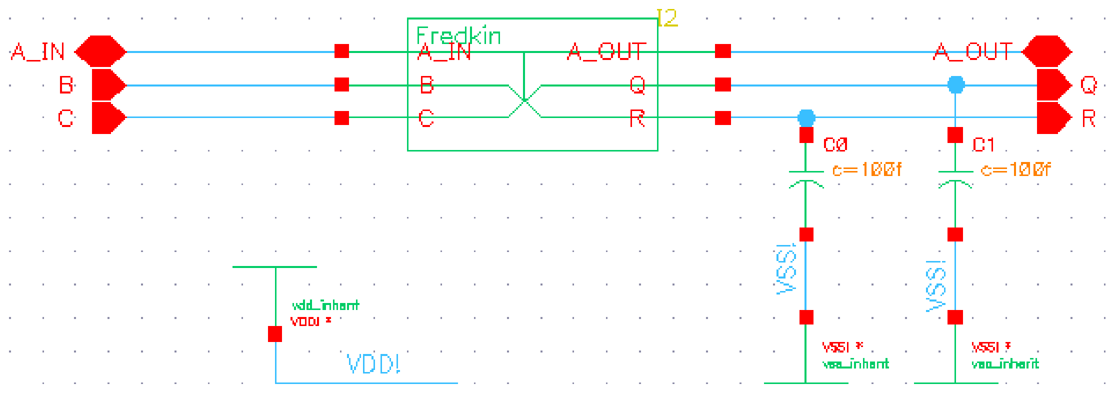


Figure 6.15: Fredkin Test Schematic

The Fredkin test schematic setup can be seen in Figure 6.15. The results acquired by running test schematic for Fredkin can be seen in Figure 6.16. It can be seen that, the Fredkin gate swaps the second and input signals when the control signal, A, is high.

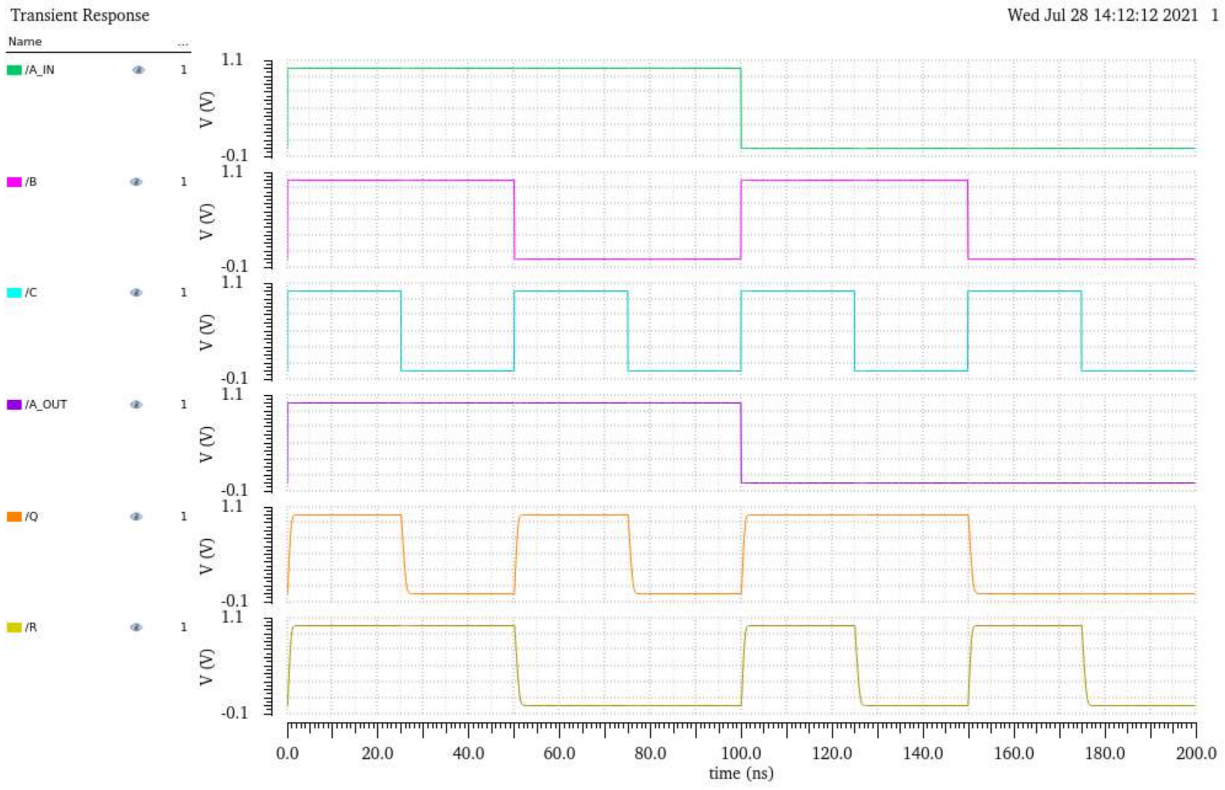


Figure 6.16: Fredkin Simulation

### 6.0.9 D-Latch Test

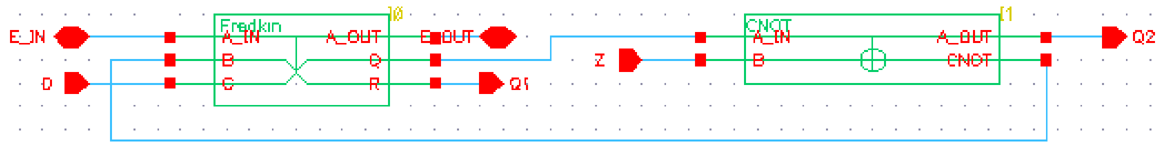


Figure 6.17: D-Latch Schematic

The combination of Fredkin gate and a CNOT gate allows a D-Latch to be created. D-Latch follows the exact pattern of an input D as long as the clock is high or the signal is enabled. The output stays at the edge where the last position of the D input was once the negative edge is present. In Figure 6.17, the schematic of the D-Latch can be seen. In Figure 6.18, the symbol of the D-Latch is present. It can be seen that the Zero input has a zero value throughout, and both of the results follow the D input through the high clock edge, and follow the same value through the negative edge.

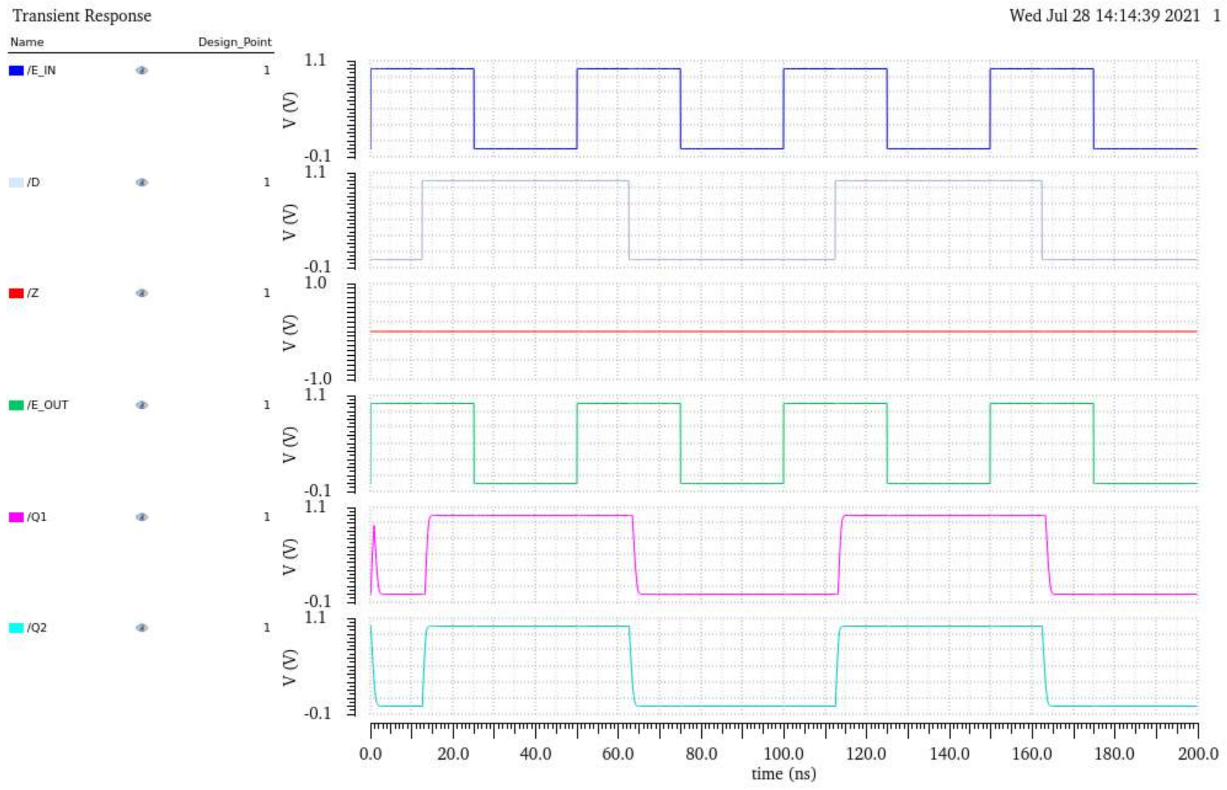


Figure 6.18: D-Latch Simulation

### 6.0.10 Reversible 32-bit Register Test

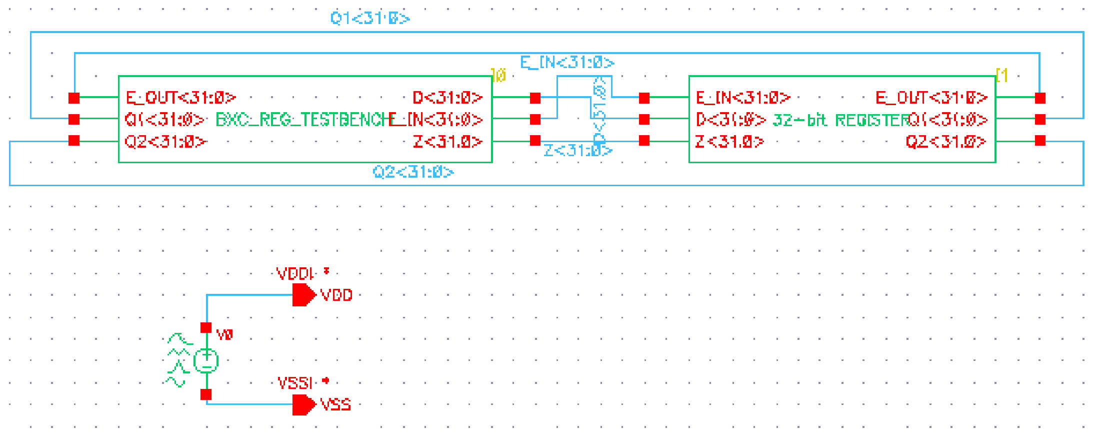
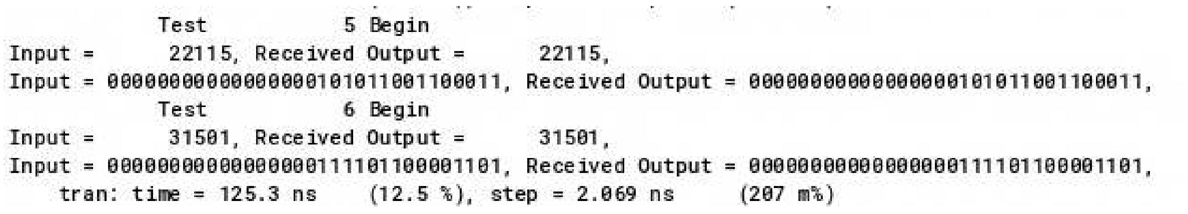


Figure 6.19: Reversible 32-bit Register Test Schematic

The test setup for the Reversible 32-bit Register is shown in Figure 6.19, the schematic of the Reversible 32-bit Register can be seen. In Figure 6.20, the pre-layout simulation results of one test is illustrated. It can be seen that the given input is received from the memory. Both binary and decimal values can be seen for inputs and outputs. Further results can be seen in AppendixI.



```
              Test              5 Begin
Input =      22115, Received Output =      22115,
Input = 00000000000000000101011001100011, Received Output = 00000000000000000101011001100011,
              Test              6 Begin
Input =      31501, Received Output =      31501,
Input = 00000000000000000111101100001101, Received Output = 00000000000000000111101100001101,
     tran: time = 125.3 ns     (12.5 %), step = 2.069 ns     (207 m%)
```

Figure 6.20: Reversible 32-bit Register Simulation

### 6.0.11 Reversible 32-bit Carry Look Ahead Adder Test

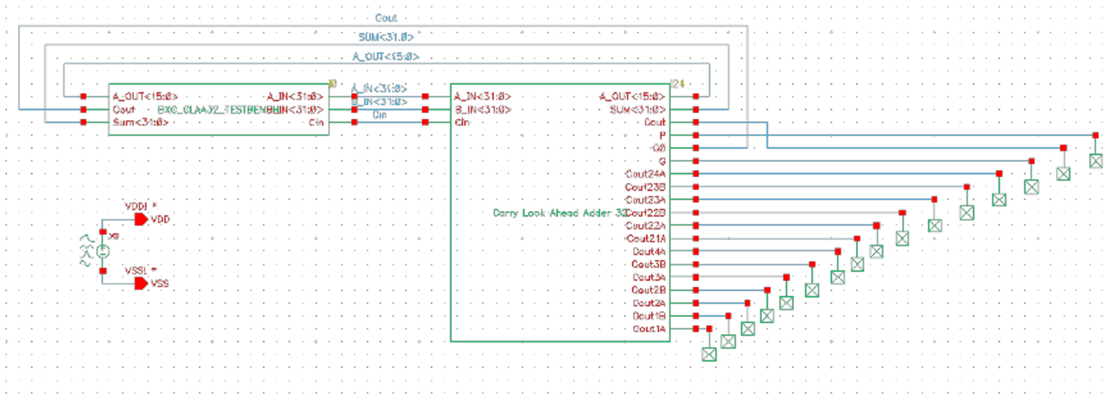
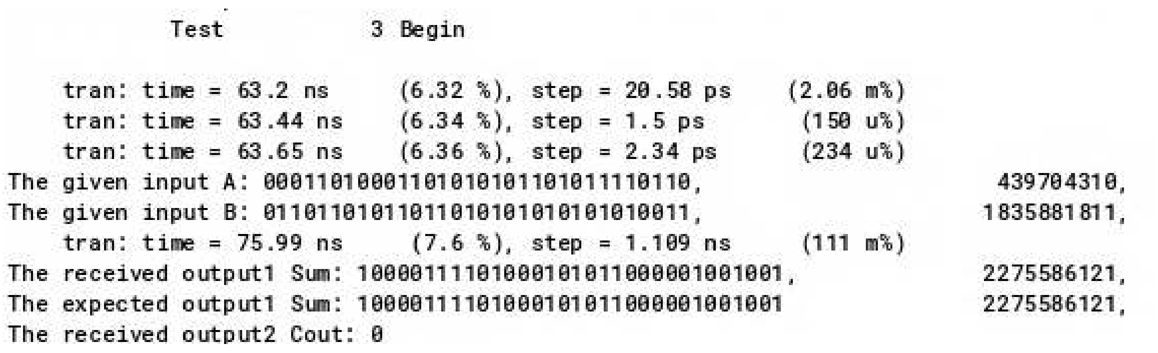


Figure 6.21: Reversible 32-bit Carry Look Ahead Adder Test Schematic

The Reversible 32-bit Carry Look Ahead Adder test schematic setup can be seen in Figure 6.21. The test setup was done by combining the symbol created through the testbench and the symbol created through the design schematic together. Some of the unwanted pins with the purpose to generate reversible gates, were ignored by adding no connect pins. The same methodology was done for Reversible 16-bit Carry Look Ahead Adder, shown as Figure 6.23 and for the Reversible 4-bit Carry Look Ahead Adder, shown as Figure 6.25. The simulation results for Reversible32 bit Carry Look Ahead Adder are shown in Figure 6.22, along with Reversible 16-bit in Figure 6.24, and Reversible 4-bit in Figure 6.26. It can be seen in each simulation result that the additions were computed correctly and the expected results were received. Both binary and decimal values can be seen for inputs and outputs. Further results can be seen in AppendixI.



```
            Test           3  Begin

    tran: time = 63.2 ns     (6.32 %), step = 20.58 ps    (2.06 m%)
    tran: time = 63.44 ns    (6.34 %), step = 1.5 ps       (150 u%)
    tran: time = 63.65 ns    (6.36 %), step = 2.34 ps      (234 u%)
The given input A: 00011010001101010101101011110110,                      439704310,
The given input B: 01101101011011010101010101010011,                     1835881811,
    tran: time = 75.99 ns     (7.6 %), step = 1.109 ns     (111 m%)
The received output1 Sum: 10000111101000101011000001001001,              2275586121,
The expected output1 Sum: 10000111101000101011000001001001               2275586121,
The received output2 Cout: 0
```

Figure 6.22: Reversible 32-bit Carry Look Ahead Adder Test Simulation

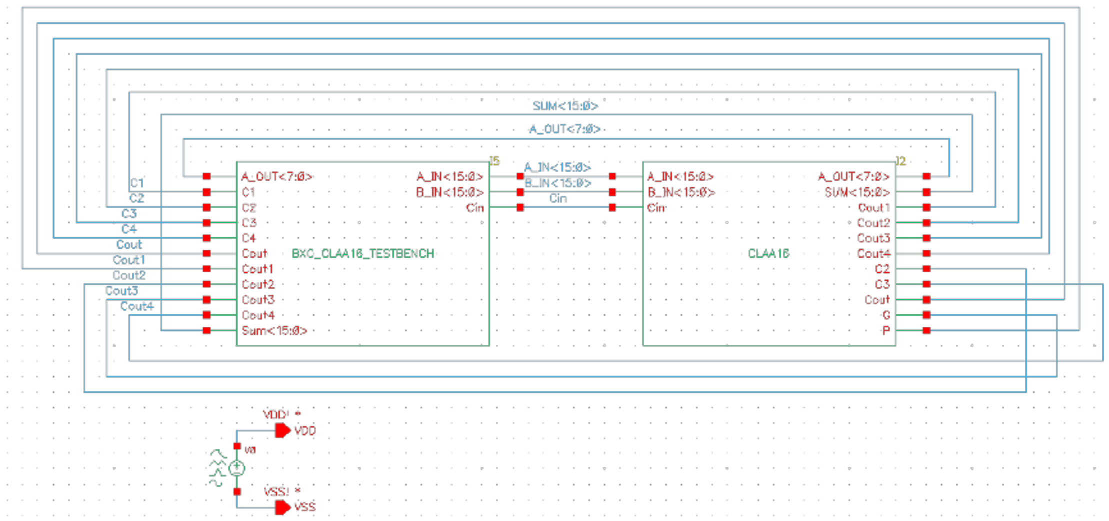
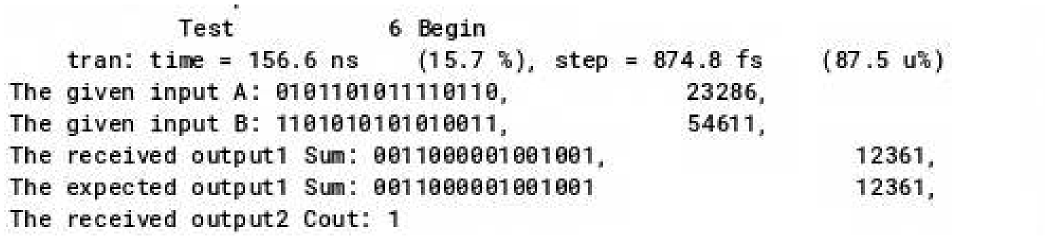


Figure 6.23: Reversible 16-bit Carry Look Ahead Adder Test Schematic



```
            Test           6  Begin
    tran: time = 156.6 ns    (15.7 %), step = 874.8 fs    (87.5 u%)
The given input A: 0101101011110110,           23286,
The given input B: 1101010101010011,           54611,
The received output1 Sum: 0011000001001001,             12361,
The expected output1 Sum: 0011000001001001              12361,
The received output2 Cout: 1
```

Figure 6.24: Reversible 16-bit Carry Look Ahead Adder Test Simulation

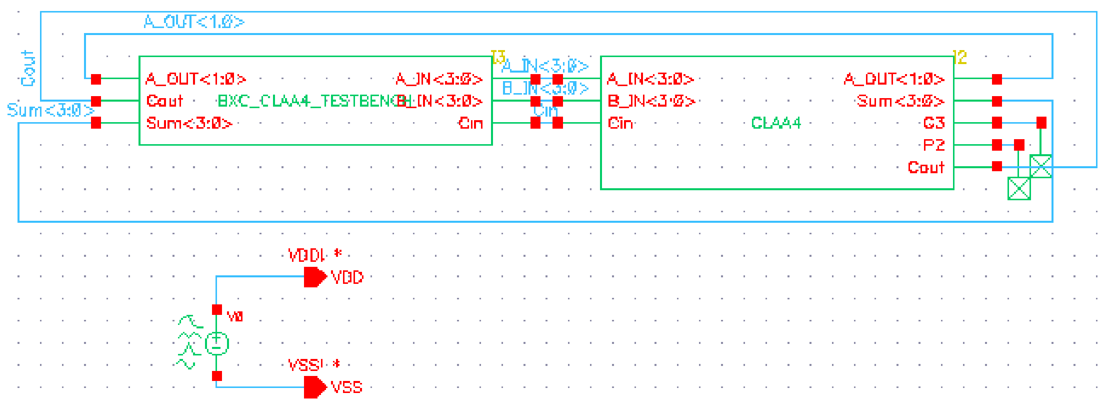
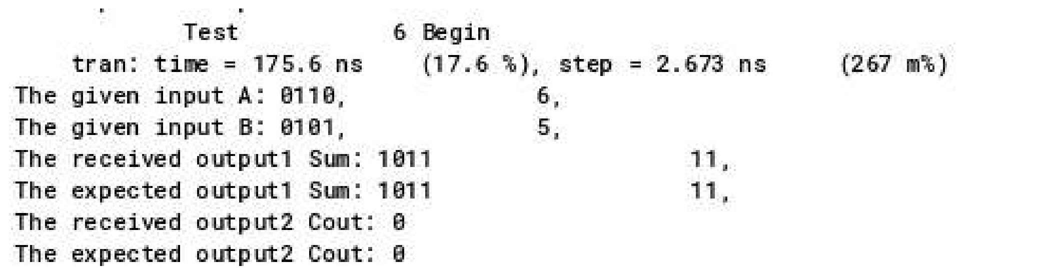


Figure 6.25: Reversible 4-bit Carry Look Ahead Adder Test Schematic



```
              Test              6 Begin
         tran: time = 175.6 ns    (17.6 %), step = 2.673 ns      (267 m%)
    The given input A: 0110,            6,
    The given input B: 0101,            5,
    The received output1 Sum: 1011              11,
    The expected output1 Sum: 1011              11,
    The received output2 Cout: 0
    The expected output2 Cout: 0
```

Figure 6.26: Reversible 4-bit Carry Look Ahead Adder Test Simulation

### 6.0.12 Reversible 16-bit Multiplier Test

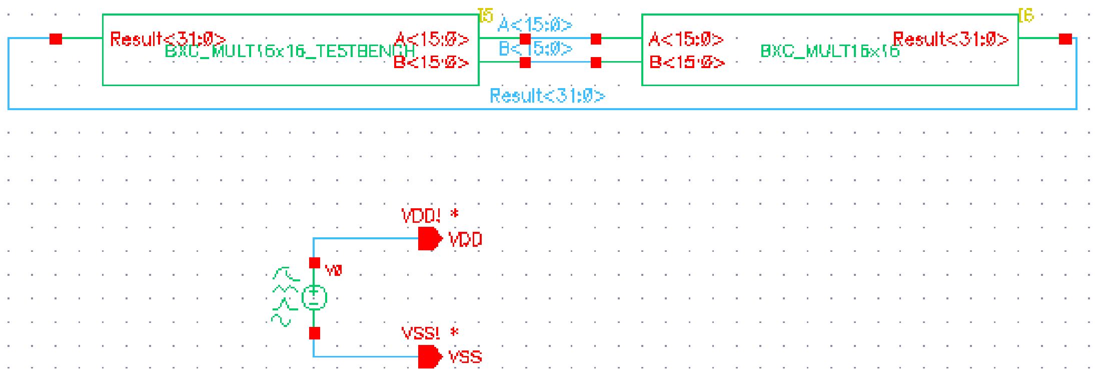
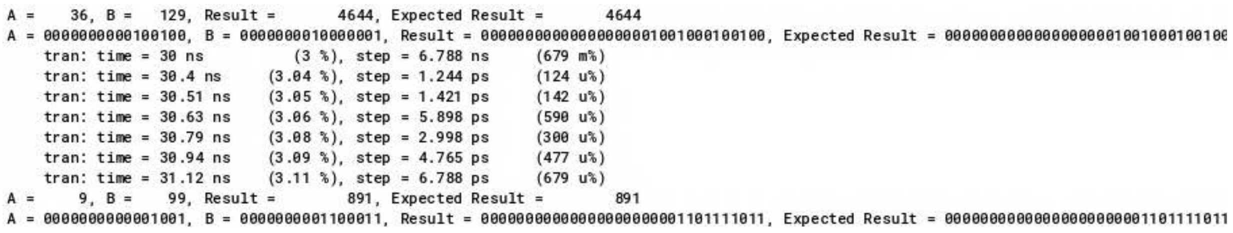


Figure 6.27: Reversible 16-bit Multiplier Test Schematic

The Reversible 16-bit Multiplier test schematic setup can be seen in Figure 6.27. The test setup was done by combining the symbol created through the testbench and the symbol created through the design schematic together. The same methodology was done for Reversible 8-bit Multiplier, shown as Figure 6.29, Reversible 4-bit Multiplier shown as Figure 6.31 as well as Reversible 2-bit Multiplier shown in Figure 6.33. The simulation results for Reversible 16-bit Multiplier is demonstrated in Figure 6.28, along with Reversible 8-bit in Figure 6.30, Reversible 4-bit in Figure 6.32 and Reversible 2-bit in Figure 6.34. Through the test results, it can be seen that the multiplication operation worked correctly and the expected results were acquired. Both binary and decimal values can be seen for inputs and outputs. Further results can be seen in Appendix I.



```
A =    36, B =   129, Result =       4644, Expected Result =       4644
A = 0000000000100100, B = 0000000010000001, Result = 00000000000000000001001000100100, Expected Result = 00000000000000000001001000100100
    tran: time = 30 ns         (3 %), step = 6.788 ns     (679 m%)
    tran: time = 30.4 ns      (3.04 %), step = 1.244 ps     (124 u%)
    tran: time = 30.51 ns     (3.05 %), step = 1.421 ps     (142 u%)
    tran: time = 30.63 ns     (3.06 %), step = 5.898 ps     (590 u%)
    tran: time = 30.79 ns     (3.08 %), step = 2.998 ps     (300 u%)
    tran: time = 30.94 ns     (3.09 %), step = 4.765 ps     (477 u%)
    tran: time = 31.12 ns     (3.11 %), step = 6.788 ps     (679 u%)
A =     9, B =    99, Result =        891, Expected Result =        891
A = 0000000000001001, B = 0000000001100011, Result = 00000000000000000000001101111011, Expected Result = 00000000000000000000001101111011
```

Figure 6.28: Reversible 16-bit Multiplier Simulation Result

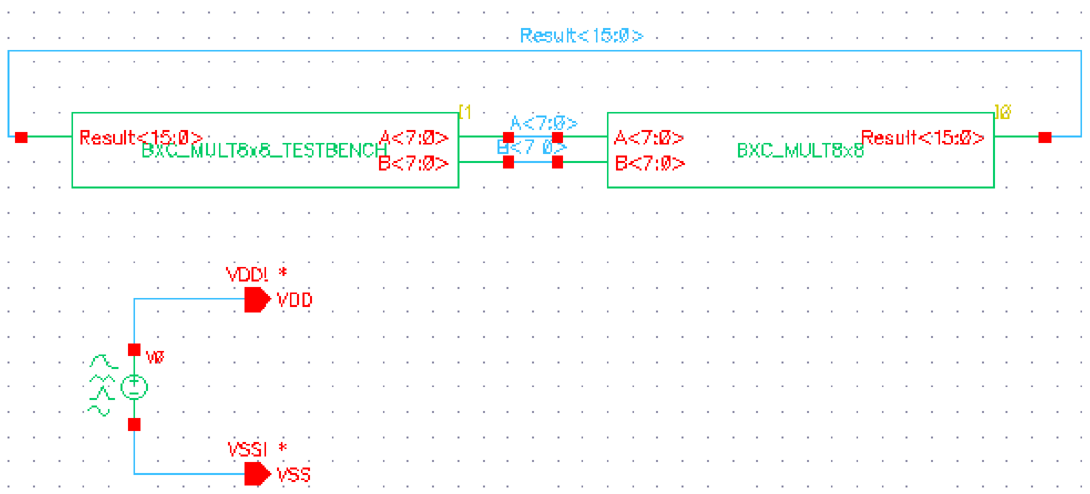
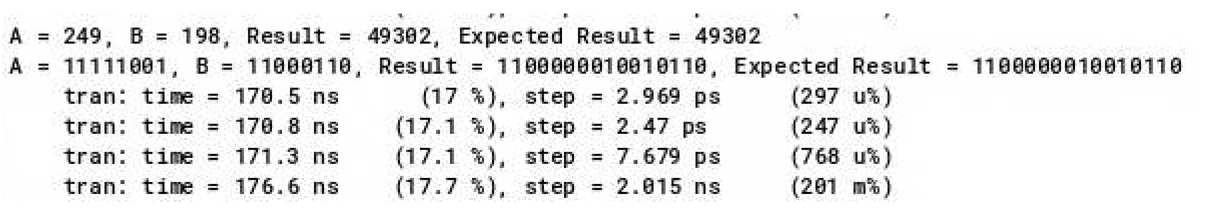


Figure 6.29: Reversible 8-bit Multiplier Test Schematic



```
A = 249, B = 198, Result = 49302, Expected Result = 49302
A = 11111001, B = 11000110, Result = 1100000010010110, Expected Result = 1100000010010110
    tran: time = 170.5 ns      (17 %), step = 2.969 ps     (297 u%)
    tran: time = 170.8 ns     (17.1 %), step = 2.47 ps     (247 u%)
    tran: time = 171.3 ns     (17.1 %), step = 7.679 ps    (768 u%)
    tran: time = 176.6 ns     (17.7 %), step = 2.015 ns    (201 m%)
```

Figure 6.30: Reversible 8-bit Multiplier Simulation Result

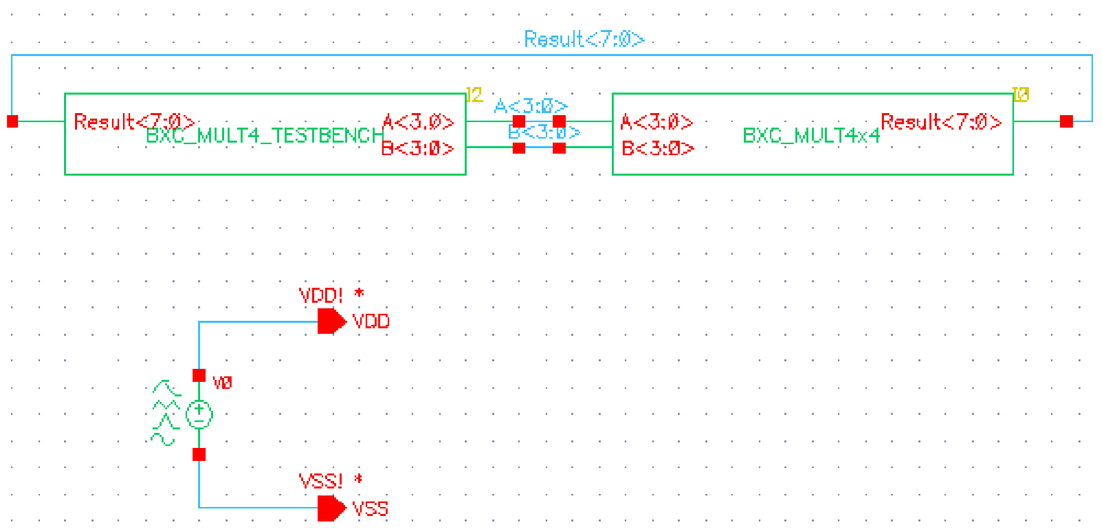
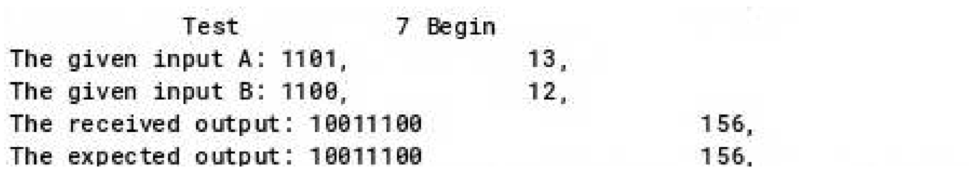


Figure 6.31: Reversible 4-bit Multiplier Test Schematic



```
             Test              7  Begin
The given input A: 1101,                 13,
The given input B: 1100,                 12,
The received output: 10011100                         156,
The expected output: 10011100                         156.
```

Figure 6.32: Reversible 4-bit Multiplier Simulation Result

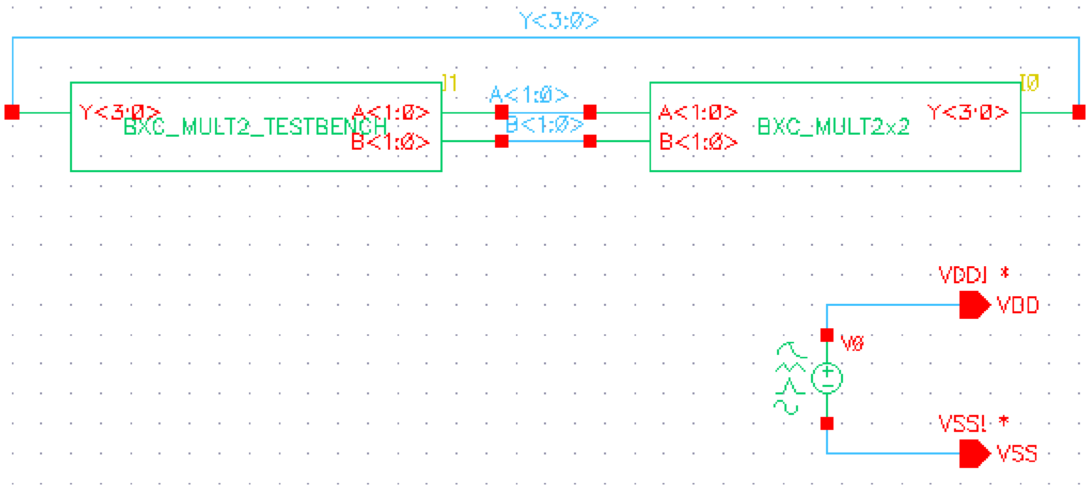
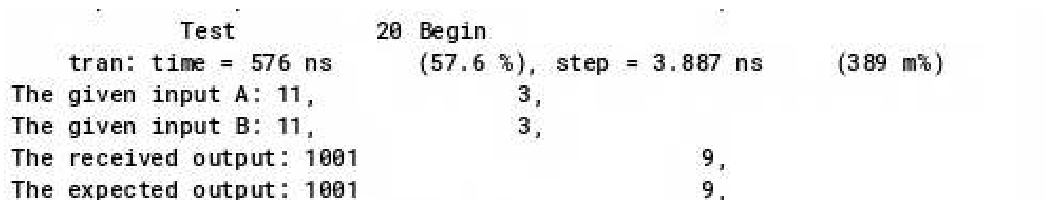


Figure 6.33: Reversible 2-bit Multiplier Test Schematic



```
            Test          20  Begin
        tran: time = 576 ns      (57.6 %), step = 3.887 ns     (389 m%)
    The given input A: 11,             3,
    The given input B: 11,             3,
    The received output: 1001                      9,
    The expected output: 1001                      9.
```

Figure 6.34: Reversible 2-bit Multiplier Simulation Result

# Chapter 7

# Reversible Multiply Accumulate Block

As mentioned in the previous chapters, the most common uses of quantum processors are for machine learning with Convolutional Neural Networks (CNN) or Deep Neural Networks (DNN). This is because the Quantum processors cut back on time it takes to compute results through probabilistic nature. This not only gives the best result, but also other possible results. Due to limitations, this project implemented the computational elements through CMOS technology. The Reversible Multiply Accumulate block was created by combining the Reversible 32-bit Carry Look Ahead Adder, Reversible 16-bit Multiplier and Reversible 32-bit Register together. In order to keep the simulation time manageable due to simulating at the transistor level, a single neuron of the CNN was implemented and then verified by using a portion of the MNIST Dataset. The MNIST Dataset is a collection of data files that contain a set of hand written digits (0 through 9) to be identified. The data was fed into the Reversible Multiply Accumulate Block and the results were generated. As mentioned before, running the entire dataset with multiple layers would have taken months to simulate at the transistor level, thus only a portion of the dataset was run in order to verify the functionality.

## 7.1 Algorithm and the Implementation

A classical way of computing a CNN layer is built using Neurons that require three inputs. Early research forcused on Single-layer Perceptron Networks; current work focuses on Multi-layer Neural Networks. For recognition of hand written digits, a multi-layer CNN which is composed of 3 layers can be used for processing the MNIST dataset. The structure of the CNN can be seen in Figure 7.1. There are 30 input nodes with 30 neurons on the left hand side. Based on classification of the first layer, the outputs from the 30 neurons enter to the second layer with 10 neurons. The outputs of the 10 neurons in the second layer enter the third layer with one neuron.

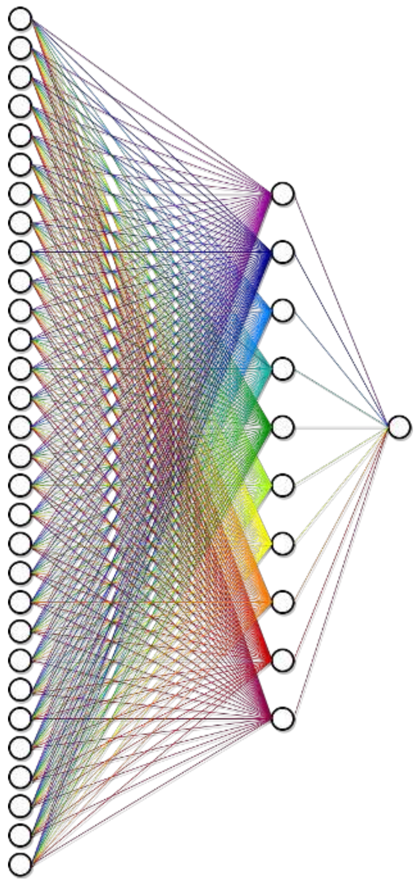


Figure 7.1: Handwriting Digit Inference CNN Structure Showing Layers

Each neuron could be built as shown in Figure 7.2, and this model was used as the algorithm for this work. As shown, each neuron contains two parts, (i) Summation and Bias, and (ii) Activation Function, which in this case is the Sigmoid Function (there are many other possible activation functions, such as the ReLu Function). The exact calculations for each part can be found in Equations 7.1 and 7.2 below. The three inputs to the network are: (i) “W” indicates the Weights, (ii) “X” indicates the input, and (iii) “B” indicates the Biases. Finally, through the Activation Function, identification of digits is possible.

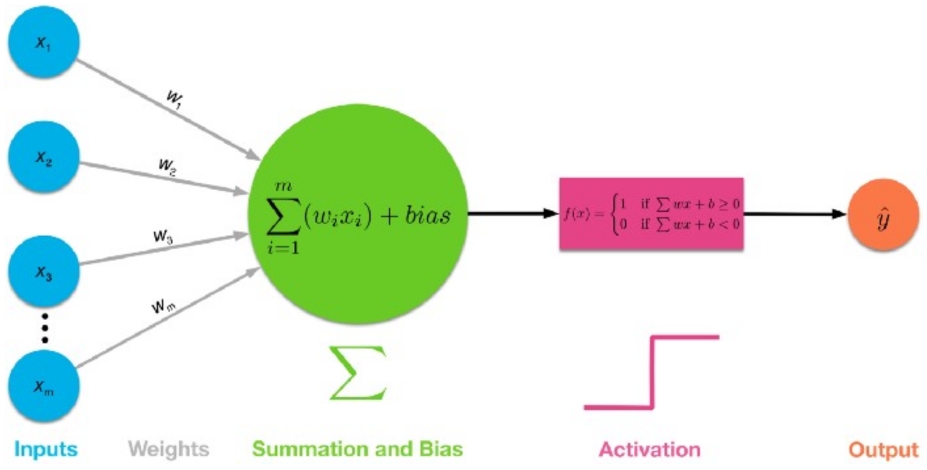

$x_1$
$x_2$
$x_3$
$x_m$
$w_1$
$w_2$
$w_3$
$w_m$
$\sum_{i=1}^{m}(w_i x_i) + bias$
$f(x) = \begin{cases} 1 & \text{if } \sum wx + b \geq 0 \\ 0 & \text{if } \sum wx + b < 0 \end{cases}$
$\hat{y}$
$\sum$
Inputs
Weights
Summation and Bias
Activation
Output

Figure 7.2: Single Neuron [1, 2]

$$Z = \sum [W_j * X_j] + B \tag{7.1}$$

This work focused on the Summation and Bias portion of the Neuron. Equation 7.1 indicates that each of the Weights needs to be multiplied with the input data, and the summation result needs to be added to a bias. In order to reduce several day’s worth of simulation time, only a single input node was used for verification at the transistor level. While the Activation Function was not implemented in this work, the Sigmoid Function is being used this CNN because the result varies between 0 and 1 simplifying the classification. The Sigmoid Function has an “S” shaped curve seen in Figure 7.3, with a threshold at one extreme and saturates at the other extreme. Thus a small change in the input does not significantly impact the result at either extreme.

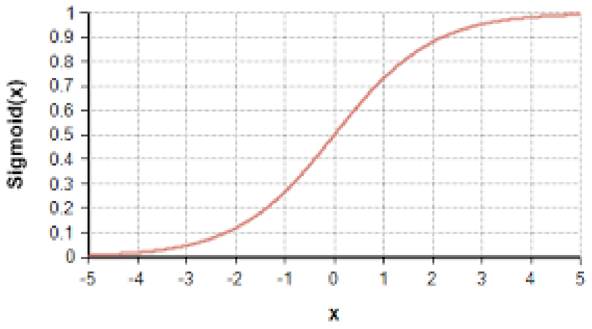


Figure 7.3: Sigmoid Function Graph

$$\sigma(Z) = \frac{1}{1+e^{-Z}} \tag{7.2}$$

The Summation and Bias portion shown in Equation 7.1 is implemented in hardware through the use of Reversible 16-bit Multiplier, Reversible 32-bit Carry Look Ahead Adder, and Reversible 32-bit Register. The block diagram of this function can be seen in Figure 7.4. Figure 7.5 illustrates the schematic of the Reversible Multiply Accumulate design, with the inputs to the multiplier fed with 16-bit Weights, and 16-bit input X. Then the resultant 32-bit product was added with the Biases since a single input node was simulated. Note the biases in MNIST dataset are 8-bits. Thus 24 bits of logic 0 were added to the most significant bit (MSB) to pad the bias to 32-bits. The result from the adder was then stored in the 32-bit Register. In order to give a 0 for Carry in and Zero signal, a Tielo block was inserted. The Tielo pulls the network down and creates a low input signal. In order to keep the input and output pin numbers equal, some unnecessary pins were connected to noConnect, which means those nets were not connected to anything and still pass the netlist checks without error.

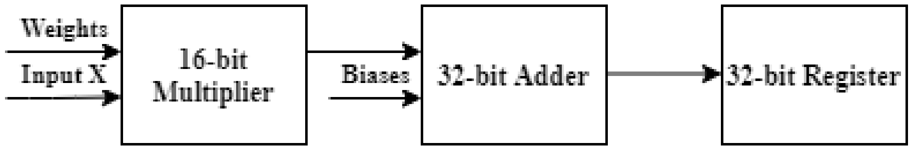


Figure 7.4: Multiply Accumulate Block Diagram

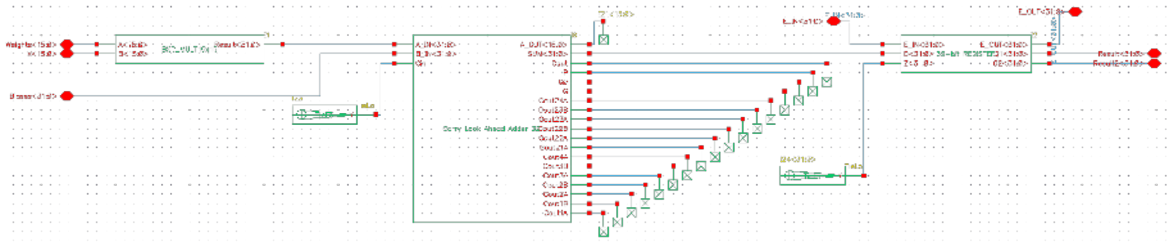


Figure 7.5: Multiply Accumulate Schematic

The symbol of the Multiply Accumulate can be seen in Figure 7.6 illustrates the schematic of the Multiply Accumulate design.

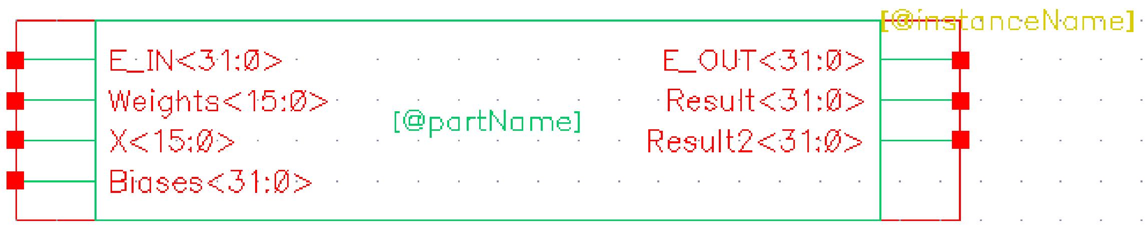


Figure 7.6: Reversible Multiply Accumulate Symbol

The symbol was then used to create a test instance with a testbench that would feed the Weights and Biases from the .mif files from the MNIST Dataset. The test schematic can be seen in Figure 7.7. After verifying the pre-layout functionality of the block, the layout of the Reversible Multiply Accumulate was created by routing the Reversible 16-bit Multiplier, Reversible 32-bit Carry Look Ahead Adder, and Reversible 32-bit Register together, which is shown in Figure 7.8. The layout of the Multiply Accumulate block passed both Design Rule Check (DRC) and Layout Versus Schematic (LVS) checks, which verified the correctness of the layout design.

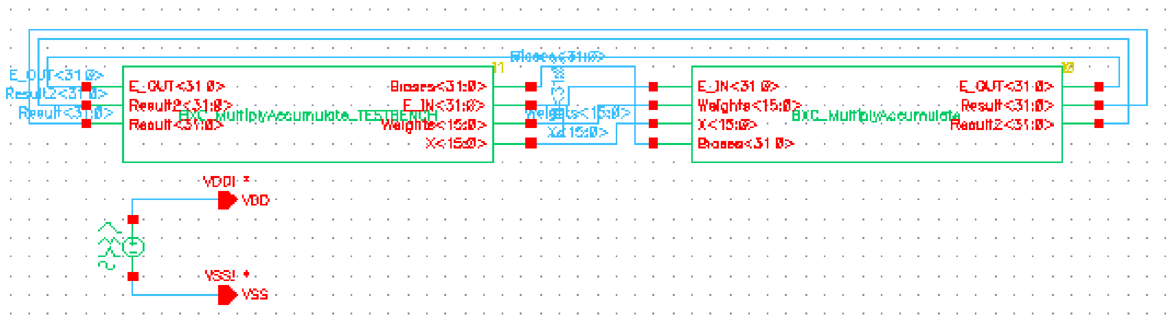


Figure 7.7: Reversible Multiply Accumulate Test Schematic

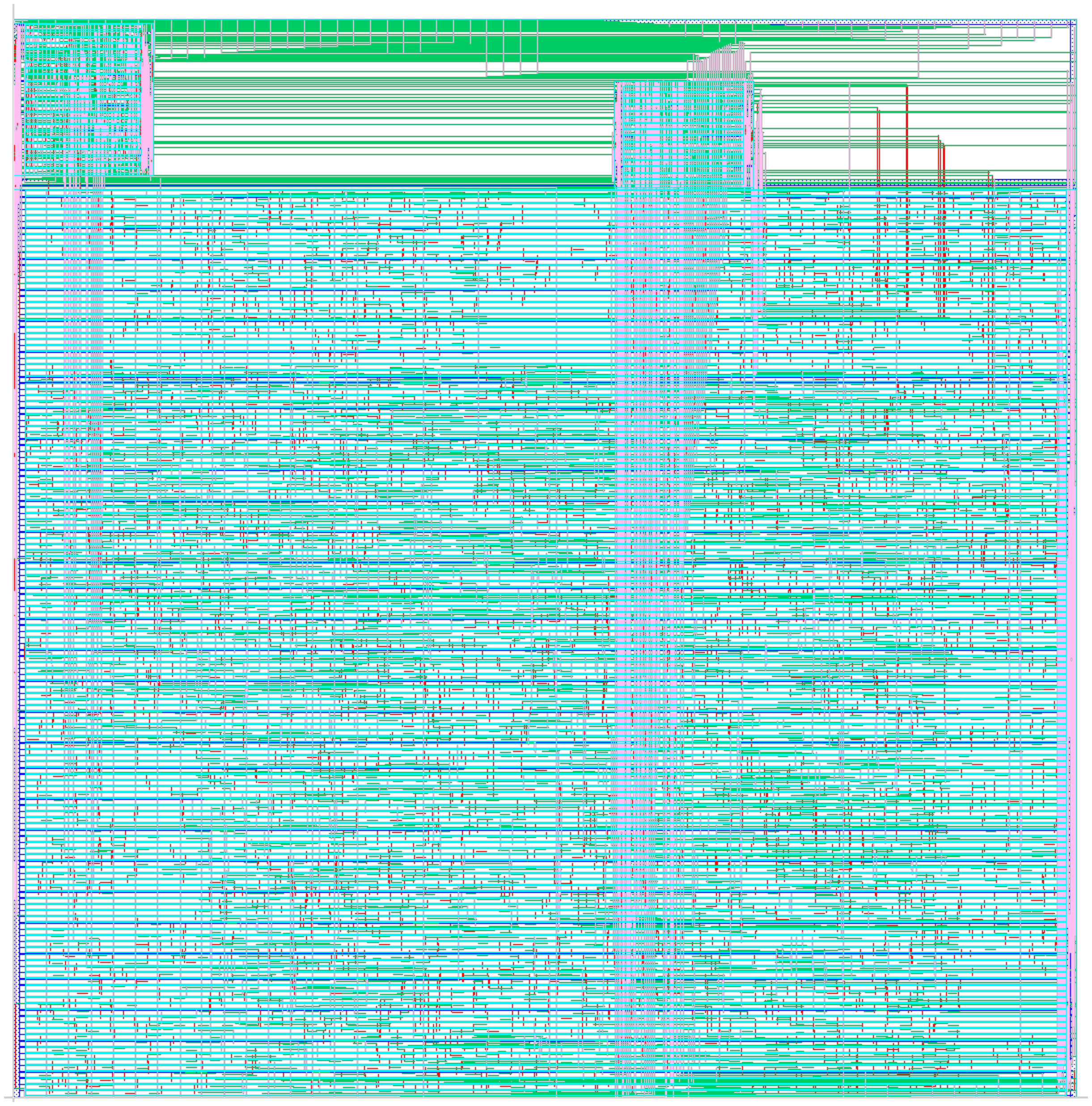
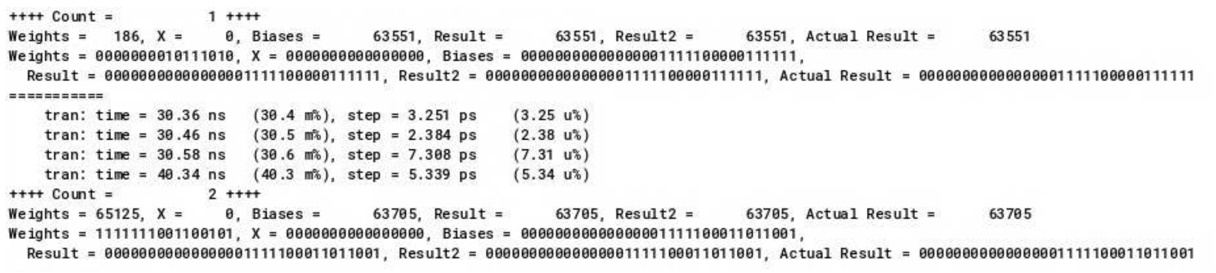


Figure 7.8: Reversible Multiply Accumulate Layout

The results obtained by the pre-layout simulation are illustrated in Figure 7.9. It can be seen from the pre-layout simulation that the Reversible Multiply Accumulate block successfully multiplies weights and the input X together and adds the biases correctly. Both binary and decimal values can be seen for inputs and outputs. Further results can be seen in Appendix I.

Thus, the project was successfully completed with the given results in Chapter 8.



```
++++ Count =           1 ++++
Weights =   186, X =     0, Biases =      63551, Result =      63551, Result2 =      63551, Actual Result =      63551
Weights = 0000000010111010, X = 0000000000000000, Biases = 00000000000000001111100000111111,
  Result = 00000000000000001111100000111111, Result2 = 00000000000000001111100000111111, Actual Result = 00000000000000001111100000111111
===========
    tran: time = 30.36 ns   (30.4 m%), step = 3.251 ps    (3.25 u%)
    tran: time = 30.46 ns   (30.5 m%), step = 2.384 ps    (2.38 u%)
    tran: time = 30.58 ns   (30.6 m%), step = 7.308 ps    (7.31 u%)
    tran: time = 40.34 ns   (40.3 m%), step = 5.339 ps    (5.34 u%)
++++ Count =           2 ++++
Weights = 65125, X =     0, Biases =      63705, Result =      63705, Result2 =      63705, Actual Result =      63705
Weights = 1111111001100101, X = 0000000000000000, Biases = 00000000000000001111100011011001,
  Result = 00000000000000001111100011011001, Result2 = 00000000000000001111100011011001, Actual Result = 00000000000000001111100011011001
```

Figure 7.9: Reversible Multiply Accumulate Pre-Layout Simulation

# Chapter 8

# Results and Discussion

## 8.1 Results

In this chapter, the size of each created component as well as the rise and fall time associated with each component is documented. As each component was successfully ran pre-layout simulation and the Layout Versus Schematic (LVS) and Design Rule Check (DRC) were passed for each layout, the design and verification of each component were made.

In Table 8.1, the width, height and area of each component is present.

Table 8.1: The Width, Height and Area of each Cell

| Cell Name | Width $(um)$ | Height $(um)$ | Area $(um^2)$ |
|---|---|---|---|
| Inverter | 0.8 | 1.71 | 1.368 |
| Reversible NAND Gate | 1.2 | 1.71 | 2.052 |
| Reversible NOR Gate | 2 | 1.71 | 3.42 |
| CNOT Gate | 5.4 | 1.71 | 9.234 |
| SWAP Gate | 16.2 | 1.71 | 27.702 |
| Toffoli Gate | 7.2 | 1.71 | 12.312 |
| Full Adder | 30 | 1.71 | 51.3 |
| Fredkin Gate | 14.46 | 1.71 | 24.7266 |
| D-Latch | 19.86 | 1.71 | 33.9606 |
| 32-bit Register | 46.04 | 35.29 | 1,624.7516 |
| 32-bit Carry Look Ahead Adder | 46.14 | 50.005 | 2,307.2307 |
| 16-bit Multiplier | 351.23 | 300 | 105,369 |
| Multiply Accumulate | 351.23 | 356.105 | 125,074.759 |

The delay of each component were measured and can be found in Table 8.1. It can be observed that the transition delay of each component is below nanoseconds.

Table 8.2: The Delay of each Cell

| Cell Name | Delay ($s$) |
|---|---|
| Inverter | 576.0E-12 |
| Reversible NAND Gate | 529.8E-12 |
| Reversible NOR Gate | 246.2E-12 |
| CNOT Gate | 431.7E-12 |
| SWAP Gate | 89.97E-12 |
| Toffoli Gate | 353.5E-12 |
| Full Adder | 592.8E-12 |
| Fredkin Gate | 458.7E-12 |
| D-Latch | 461.0E-12 |
| 32-bit Register | 598.6E-12 |
| 32-bit Carry Look Ahead Adder | 535.6E-11 |
| 16-bit Multiplier | 387.2E-11 |

## 8.2 Discussion

It can be seen in Table 8.1 that the 16-bit multiplier and the Reversible Multiply Accumulate unit are the largest. This is because during auto placement, the tools leave unnecessary spacing between each component and gave up auto placement as the machines that were available for this work at Rochester Institute of Technology did not have enough memory to support auto placement. As the 32-bit Carry Look Ahead Adder and the 32-bit Register had less components, the cells were gathered closer by hand. However due to the large number of cells that the 16-bit Multiplier had, it would have taken several weeks to move and place each component by hand. Thus, the auto-placed layout design was used and the area of the 16-bit Multiplier and the Multiply Accumulate unit ended up being unnecessarily larger than needed.

Each design successfully ran pre-layout simulation and then layouts were completed. The verification of layout was done first by comparing the layout and schematic through LVS, and then by checking the design constraints through DRC. Even though the LVS were passed for each layout, for multiplier and adder, the tools complained about shorts in input-output nets. As these were just warnings that were expected, they could safely be ignored.

From a timing perspective, as can be seen in Table 8.2, the basic gates logic gates were all reasonably fast, with timing in the picoseconds. The hierarchical cells, 16-bit Multiplier and 32-bit Carry Look Ahead Adder were also quite fast as expected since the architecture of these blocks are known to provide fast designs.

# Chapter 9

# Conclusion

Through this work, reversible Quantum logic structures in CMOS technology were researched and successfully created, designed and verified. Moreover, some classical gates were converted into reversible gates and tested as well. Through the use of fully reversible gates, a Reversible Multiply Accumulate Block used as part of a Perceptron was successfully designed in hardware for Convolutional Neural Network use. The functionality of the hardware was then tested and verified by using the the MNIST Dataset, the database of handwritten digits. The Reversible Multiply Accumulate Block throughout this paper, computed each calculation accurately. For each designed component, a layout was also created. The physical size was measured along with the delay time, and these were detailed in the Results Section in Chapter 8. Even though each component designed was reversible, the reconstruction of inputs from outputs have not been worked on in this research. Achieving forward and backward computation could cut back on time to train the CNNs, which is detailed in Future Work below, in Section 9.1.

## 9.1 Future Work

For future work, the entire MNIST dataset for all nodes can be ran on the Reversible Multiply Accumulate hardware instead of just running a single node. The Sigmoid Function could be added to the Neuron to complete the classification. Even though the Reversible Multiply Accumulate Block is composed of reversible gates, as the Reversible Multiplier block is made out of hierarchical blocks, the inputs may not be fully reconfigurable. When CNN undergoes calculation and machine learning, the input nodes adjust the weights and biases, which means if there is a miscategorization, the reversible gates should have the capability to go back a layer instead of rerunning the entire algorithm. This would potentially save a tremendous amount of time and resources as every miscalculation requires the layers to be categorized all over again. Reconstruction of inputs from the outputs can be tested as a future work as well.

# References

[1] Nahua Kang. Multi-Layer Neural Networks with Sigmoid Function - Deep Learning for Rookies (2). *Medium*, 2017.

[2] Shiwei Xing and Chenjian Wu. Implementation of A Neuron Using Sigmoid Activation Function with CMOS. In *2020 IEEE 5th International Conference on Integrated Circuits and Microsystems (ICICM)*, pages 201–204, 2020. doi:10.1109/ICICM50929.2020.9292239.

[3] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando Brandao, David Buell, Brian Burkett, Yu Chen, Jimmy Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Michael Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew Harrigan, Michael Hartmann, Alan Ho, Markus Rudolf Hoffmann, Trent Huang, Travis Humble, Sergei Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, Dave Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore MandrÃ , Jarrod Ryan McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John Platt, Chris Quintana,

Eleanor G. Rieffel, Pedram Roushan, Nicholas Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin Jeffery Sung, Matt Trevithick, Amit Vainsencher, Benjamin Villalonga, Ted White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John Martinis. Quantum Supremacy using a Programmable Superconducting Processor. *Nature*, 574:505–510, 2019. doi:10.1038/s41586-019-1666-5.

[4] Bahar Canga. D-Wave's Quantum Processing Unit. April 2021.

[5] Amir Khoshaman, Walter Vinci, Brandon Denis, Evgeny Andriyash, Hossein Sadeghi, and Mohammad H. Amin. Quantum variational autoencoder. *Quantum Science and Technology*, 4(1), Sep 2018. doi:10.1088/2058-9565/aada1f.

[6] Jason Tyler Rolfe. Discrete Variational Autoencoders. 2017. arXiv:1609.02200.

[7] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[8] R. Jayashree and M. Kiran Kumar. DESIGN AND ANALYSIS OF FLIP-FLOPS USING REVERSIBLE LOGIC. *International Journal of Advanced Information Science and Technology (IJAIST)*, 23(23):210–217, 2014.

[9] H. Rohini and S. Rajashekar. Design of Reversible Logic based Basic Combinational Circuits. *Communications on Applied Electronics*, 5(9):38–43, Sep 2016. doi:10.5120/cae2016652372.

[10] Thomas G. Draper, Samuel A. Kutin, Eric M. Rains, and Krysta M. Svore. A logarithmic-depth quantum carry-lookahead adder. *Quant. Inf. Comp.*, 6(4-5):351–369, 2006.

[11] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[12] D.V.Manjunatha Savita Patil, D.V.Manjunatha. Design of speed and power efficient multipliers using vedic mathematics with vlsi implementation. *International Conference on Advances in Electronics, Computers and Communications (ICAECC)*, page 6, 2014.

[13] Amit Kumar and Hitesh Pahuja. Design and Analysis of Faster Multiplier using Vedic Mathematics Technique. *IJCA Proceedings on International Conference on Advancements in Engineering and Technology*, ICAET 2016(9):28–31, September 2016. Full text available.

[14] Deepak Kumar Shiksha Pandey. A Fast 16x16 Vedic Multiplier Using Carry Select Adder on FPGA. *International Journal of Advanced Research in Computer and Communication Engineering*, 5:989–994, April 2016. doi:10.17148/IJARCCE.2016.54243.

[15] J M Rudagi, Vishwanath Ambli, Vishwanath Munavalli, Ravindra Patil, and Vinaykumar Sajjan. Design and implementation of efficient multiplier using vedic mathematics. In *3rd International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom 2011)*, pages 162–166, 2011. doi:10.1049/ic.2011.0071.

# Appendix I

# Testbench Code and Simulation Results

## I.1 Reversible Carry Look Ahead Adder 4-bit Testbench

```
//Verilog HDL for "bxc7483_bxc_lib", "BXC_CLAA4_TESTBENCH" "
    functional"


module BXC_CLAA4_TESTBENCH ( A_IN, B_IN, Cin,Cout,Sum ,A_OUT);

  output [3:0] A_IN;
  output [3:0] B_IN;
  output Cin;
  input [3:0] Sum;
  input [1:0] A_OUT;
  input  Cout;

  reg [3:0] A_IN_T;
  reg [3:0] B_IN_T;
  reg[3:0] Result;
  reg Cin_T;
  reg ECout;
  integer loop1 ;

assign Cin = Cin_T;
assign A_IN = A_IN_T;
assign B_IN = B_IN_T ;

initial begin

  A_IN_T = 4'b0000;
  B_IN_T = 4'b0000;
  Cin_T = 0;
  Result = 0;
  ECout = 0;

  loop1 = 0;
end
always begin
  #3; //Propagation
  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;
```

```
    A_IN_T = $random;
    B_IN_T = $random;
    Cin_T = $random;
    #10;
    {ECout,Result} = A_IN_T + B_IN_T + Cin_T;
    $display ("The given input A: %b,   %d,", A_IN, A_IN);
    $display ("The given input B: %b,   %d,", B_IN, B_IN);
    #10;
    $display ("The received output1 Sum: %b   %d,", Sum, Sum);
    $display ("The expected output1 Sum: %b   %d,", Result,
        Result);
    $display ("The received output2 Cout: %b", Cout);
    $display ("The expected output2 Cout: %b", ECout);
    #10;
    if (loop1 == 20)
      $finish;
end
endmodule
```

Listing I.1: Carry Look Ahead Adder 4-bits Testbench

## I.2 Reversible Carry Look Ahead Adder 16-bit Testbench

```
//Verilog HDL for "bxc7483_bxc_lib", "BXC_CLAA16_TESTBENCH" "
    functional"


module BXC_CLAA16_TESTBENCH (  A_IN, B_IN, Cin, Cout, Cout1,
    Cout2, Cout3, Cout4,Sum ,A_OUT, C1, C2, C3, C4);

  output [15:0] A_IN;
  output [15:0] B_IN;
  output Cin;
  input [15:0] Sum;
  input [7:0] A_OUT;
  input  Cout;
  input  Cout1;
  input  Cout2;
  input  Cout3;
  input  Cout4;
  input C1, C2, C3, C4;

  reg [15:0] A_IN_T;
  reg [15:0] B_IN_T;
  reg [15:0] Result;

  reg Cin_T;
  reg Cout_T;
  integer loop1 ;
assign Cin = Cin_T;
assign A_IN = A_IN_T;
assign B_IN = B_IN_T ;

initial begin

  A_IN_T = 16'b0000;
  B_IN_T = 16'b0000;
  Result = 16'b0000;
  Cin_T = 0;

  loop1 = 0;
end
always begin
  #3; //Propagation
  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;
  Cin_T = 0;
  Result = 16'b0000;
  A_IN_T= 16'b01010;
  B_IN_T= 16'b0111001;
  Result = A_IN_T + B_IN_T;
```

```
#10;
$display ("The given input A: %b,   %d,", A_IN, A_IN);
$display ("The given input B: %b,   %d,", B_IN, B_IN);
#10;
$display ("The received output1 Sum: %b   %d,", Sum, Sum);
$display ("The expected output1 Sum: %b   %d,", Result,
    Result);
$display ("The received output2 Cout: %b", Cout);
$display ("The received output2 Cout1: %b   vs   %b", Cout1
    , C1);
$display ("The received output2 Cout1: %b   vs   %b", Cout2
    , C2);
$display ("The received output2 Cout1: %b   vs   %b", Cout3
    , C3);
$display ("The received output2 Cout1: %b   vs   %b", Cout4
    , C4);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 16'b0000;
A_IN_T= 16'b111010;
B_IN_T= 16'b01101;
Result = A_IN_T + B_IN_T;
#10;
```

```verilog
$display ("The given input A: %b,   %d,", A_IN, A_IN);
$display ("The given input B: %b,   %d,", B_IN, B_IN);
#10;
$display ("The received output1 Sum: %b,    %d,", Sum, Sum);
$display ("The expected output1 Sum: %b   %d,", Result,
    Result);
$display ("The received output2 Cout: %b", Cout);
$display ("The received output2 Cout1: %b   vs    %b", Cout1
    , C1);
$display ("The received output2 Cout1: %b   vs    %b", Cout2
    , C2);
$display ("The received output2 Cout1: %b   vs    %b", Cout3
    , C3);
$display ("The received output2 Cout1: %b   vs    %b", Cout4
    , C4);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 16'b00000;
A_IN_T= 16'b0101101011110110;
B_IN_T= 16'b1101010101010011;
Result = A_IN_T + B_IN_T;
#10;
$display ("The given input A: %b,   %d,", A_IN, A_IN);
```

```
  $display ("The given input B: %b,   %d,", B_IN, B_IN);
  #10;
  $display ("The received output1 Sum: %b,    %d,", Sum, Sum);
  $display ("The expected output1 Sum: %b   %d,", Result,
      Result);
  $display ("The received output2 Cout: %b", Cout);
  $display ("The received output2 Cout1: %b   vs    %b", Cout1
      , C1);
  $display ("The received output2 Cout1: %b   vs    %b", Cout2
      , C2);
  $display ("The received output2 Cout1: %b   vs    %b", Cout3
      , C3);
  $display ("The received output2 Cout1: %b   vs    %b", Cout4
      , C4);
  #10;


end
endmodule
```

Listing I.2: Carry Look Ahead Adder 16-bits Testbench

## I.3 Reversible Carry Look Ahead Adder 32-bit Testbench

```
//Verilog HDL for "bxc7483_bxc_lib", "BXC_CLAA32_TESTBENCH" "
    functional"


module BXC_CLAA32_TESTBENCH (  A_IN, B_IN, Cin, Cout,Sum ,A_OUT
    );

  output [31:0] A_IN;
  output [31:0] B_IN;
  output Cin;
  input [31:0] Sum;
  input [15:0] A_OUT;
  input  Cout;

  reg [31:0] A_IN_T;
  reg [31:0] B_IN_T;
  reg [31:0] Result;

  reg Cin_T;
  reg Cout_T;
  integer loop1 ;
assign Cin = Cin_T;
assign A_IN = A_IN_T;
assign B_IN = B_IN_T ;

initial begin

  A_IN_T = 32'b0000;
  B_IN_T = 32'b0000;
  Result = 32'b0000;
  Cin_T = 0;

  loop1 = 0;
end
always begin
  #3; //Propagation
  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;
  A_IN_T= 32'b01010;
  B_IN_T= 32'b0111001;
  Result = A_IN_T + B_IN_T;
  #10;
  $display ("The given input A: %b,  %d,", A_IN, A_IN);
  $display ("The given input B: %b,  %d,", B_IN, B_IN);
  #10;
  $display ("The received output1 Sum: %b  %d,", Sum, Sum);
  $display ("The expected output1 Sum: %b  %d,", Result,
     Result);
```

```
$display ("The received output2 Cout: %b", Cout);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 32'b0000;
A_IN_T= 32'b111010;
B_IN_T= 32'b01101;
Result = A_IN_T + B_IN_T;
#10;
$display ("The given input A: %b,   %d,", A_IN, A_IN);
$display ("The given input B: %b,   %d,", B_IN, B_IN);
#10;
$display ("The received output1 Sum: %b,    %d,", Sum, Sum);
$display ("The expected output1 Sum: %b   %d,", Result,
    Result);
$display ("The received output2 Cout: %b", Cout);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 32'b00000;
A_IN_T= 32'b00110100011010101011010111110110;
B_IN_T= 32'b01101101011011010101010101010011;
Result = A_IN_T + B_IN_T;
```

```
    #10;
    $display ("The given input A: %b,   %d,", A_IN, A_IN);
    $display ("The given input B: %b,   %d,", B_IN, B_IN);
    #10;
    $display ("The received output1 Sum: %b,    %d,", Sum, Sum);
    $display ("The expected output1 Sum: %b   %d,", Result,
        Result);
    $display ("The received output2 Cout: %b", Cout);
    #10;


end
endmodule
```

Listing I.3: Carry Look Ahead Adder 32-bits Testbench

## I.4 Reversible Multiplier 2-bit Testbench

```
//Verilog HDL for "bxc7483_bxc_lib", "BXC_CLAA4_TESTBENCH" "
    functional"


module BXC_MULT2_TESTBENCH (A, B , Y);

  output [1:0] A;
  output [1:0] B;
  input [3:0] Y;


  reg [1:0] A_T;
  reg [1:0] B_T;
  reg [3:0] Result;

  integer loop1 ;
assign A = A_T;
assign B = B_T ;

initial begin

  A_T = 2'b00;
  B_T = 2'b00;
  loop1 = 0;
end
always begin
  #3; //Propagation
  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;
  Result = 4'b0000;
  A_T  = 2'b00;
  B_T  = 2'b00;
  #0 Result = A * B;
  #10;
  $display ("The given input A: %b,   %d,", A_T, A_T);
  $display ("The given input B: %b,   %d,", B_T, B_T);
  #10;
  $display ("The received output: %b    %d,", Y, Y);
  $display ("The expected output: %b    %d,", Result, Result);
  #10;

  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;
  Result = 4'b0000;
  A_T  = 2'b01;
  B_T  = 2'b00;
  #0 Result = A * B;
  #10;
```

```
$display ("The given input A: %b,   %d,", A_T, A_T);
$display ("The given input B: %b,   %d,", B_T, B_T);
#10;
$display ("The received output: %b    %d,", Y, Y);
$display ("The expected output: %b    %d,", Result, Result);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 4'b0000;
A_T  = 2'b11;
B_T  = 2'b00;
#0 Result = A * B;
#10;
$display ("The given input A: %b,   %d,", A_T, A_T);
$display ("The given input B: %b,   %d,", B_T, B_T);
#10;
$display ("The received output: %b    %d,", Y, Y);
$display ("The expected output: %b    %d,", Result, Result);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 4'b0000;
A_T  = 2'b10;
```

```
B_T  = 2'b00;
#0 Result = A * B;
#10;
$display ("The given input A: %b,  %d,", A_T, A_T);
$display ("The given input B: %b,  %d,", B_T, B_T);
#10;
$display ("The received output: %b   %d,",  Y, Y);
$display ("The expected output: %b   %d,", Result, Result);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 4'b0000;
A_T  = 2'b11;
B_T  = 2'b10;
#0 Result = A * B;
#10;
$display ("The given input A: %b,  %d,", A_T, A_T);
$display ("The given input B: %b,  %d,", B_T, B_T);
#10;
$display ("The received output: %b   %d,",  Y, Y);
$display ("The expected output: %b   %d,", Result, Result);
#10;

loop1 = loop1 +1;
```

```
 $display (" Test %d Begin", loop1) ;
 Result = 4'b0000;
A_T  = 2'b11;
B_T  = 2'b00;
#0 Result = A * B;
#10;
$display ("The given input A: %b,  %d,", A_T, A_T);
$display ("The given input B: %b,  %d,", B_T, B_T);
#10;
$display ("The received output: %b   %d,",  Y, Y);
$display ("The expected output: %b   %d,", Result, Result);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
 Result = 4'b0000;
A_T  = 2'b01;
B_T  = 2'b11;
#0 Result = A * B;
#10;
$display ("The given input A: %b,  %d,", A_T, A_T);
$display ("The given input B: %b,  %d,", B_T, B_T);
#10;
$display ("The received output: %b   %d,",  Y, Y);
$display ("The expected output: %b   %d,", Result, Result);
```

```
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 4'b0000;
A_T  = 2'b11;
B_T  = 2'b10;
#0 Result = A * B;
#10;
$display ("The given input A: %b,   %d,", A_T, A_T);
$display ("The given input B: %b,   %d,", B_T, B_T);
#10;
$display ("The received output: %b    %d,",  Y, Y);
$display ("The expected output: %b    %d,", Result, Result);
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
Result = 4'b0000;
A_T  = 2'b01;
B_T  = 2'b10;
#0 Result = A * B;
#10;
$display ("The given input A: %b,   %d,", A_T, A_T);
$display ("The given input B: %b,   %d,", B_T, B_T);
```

```
  #10;
  $display ("The received output: %b   %d,",  Y, Y);
  $display ("The expected output: %b   %d,", Result, Result);
  #10;

  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;
  Result = 4'b0000;
 A_T  = 2'b11;
 B_T  = 2'b11;
 #0 Result = A * B;
  #10;
  $display ("The given input A: %b,  %d,", A_T, A_T);
  $display ("The given input B: %b,  %d,", B_T, B_T);
  #10;
  $display ("The received output: %b   %d,",  Y, Y);
  $display ("The expected output: %b   %d,", Result, Result);
  #10;



end
endmodule
```

Listing I.4: Multiplier 2-bits Testbench

## I.5 Reversible Multiplier 4-bit Testbench

```
//Verilog HDL for "bxc7483_bxc_lib", "BXC_CLAA4_TESTBENCH" "
    functional"


module BXC_MULT4_TESTBENCH (A, B , Result);

  output [3:0] A;
  output [3:0] B;
  input [7:0] Result;


  reg [3:0] A_T;
  reg [3:0] B_T;
  reg [7:0] ResultT;

  integer loop1 ;
assign A = A_T;
assign B = B_T ;

initial begin

  A_T = 4'b00;
  B_T = 4'b00;
  loop1 = 0;
end
always begin
  #3; //Propagation
  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;
  A_T  = $random;
  B_T  = $random;
  #10;
  ResultT = A * B;
  $display ("The given input A: %b,   %d,", A_T, A_T);
  $display ("The given input B: %b,   %d,", B_T, B_T);
  #15;
  $display ("The received output: %b    %d,", Result, Result);
  $display ("The expected output: %b    %d,", ResultT, ResultT)
     ;
  #5;
  if (loop1 == 30)
    $finish;

end
endmodule
```

Listing I.5: Multiplier 4-bits Testbench

## I.6 Reversible Multiplier 8-bit Testbench

```
// Verilog HDL for "bxc7483_bxc_lib", "BXC_MULT8x8_TESTBENCH" "
    functional"


module BXC_MULT8x8_TESTBENCH ( A, B, Result );

  output [7:0] A;
  output [7:0] B;
  input  [15:0] Result;

  wire [7:0] A;
  wire [7:0] B;
  reg    [7:0] a, b;

  reg clk;

integer cnt;

wire [15:0] r = a * b ;

assign A = a;
assign B = b;

```

```
initial
begin
  a = 0;
  b = 0;
  clk = 0;
  cnt = 0;

end

always @(posedge clk)
begin
  $display("A = %d, B = %d, Result = %d, Expected Result = %d",
      a, b, Result, r);
  $display("A = %b, B = %b, Result = %b, Expected Result = %b",
      a, b, Result, r);
  if (Result != r)
    $display("ERROR: Result(%d) != r(%d)\n", Result, r);

  a <= ($random & 8'hff);
  b <= ($random & 8'hff);
  cnt <= cnt + 1;
 @(negedge clk);
  if (cnt > 30)
    $finish;
end
```

```

always
  #10 clk = ~clk;

endmodule
```

Listing I.6: Multiplier 8-bits Testbench

## I.7 Reversible Multiplier 16-bit Testbench

```
//Verilog HDL for "bxc7483_bxc_lib", "BXC_Mult16" "functional"


module BXC_MULT16x16_TESTBENCH ( A, B, Result);

  output [15:0] A;
  output [15:0] B;
  input  [31:0] Result;

  wire [15:0] A;
  wire [15:0] B;

  reg    [15:0] a, b;
  reg clk;

integer cnt;

wire [31:0] r = a * b ;

assign A = a;
assign B = b;

initial
```

```
begin
  a = 0;
  b = 0;
  clk = 0;
  cnt = 0;
end

always @(posedge clk)
begin
  $display("A = %d, B = %d, Result = %d, Expected Result = %d",
      a, b, Result, r);
  $display("A = %b, B = %b, Result = %b, Expected Result = %b",
      a, b, Result, r);
  if (Result != r)
    $display("ERROR: Result(%d) != r(%d)\n", Result, r);

  a <= ($random & 16'hff);
  b <= ($random & 16'hff);
  cnt <= cnt + 1;
 @(negedge clk);
  if (cnt > 30)
    $finish;
end

always
```

```
  #10 clk = ~clk;

endmodule
```

Listing I.7: Multiplier 16-bits Testbench

## I.8 Reversible Register 32-bit Testbench

```
//Verilog HDL for "bxc7483_bxc_lib", "BXC_REG_TESTBENCH" "
    functional"


module BXC_REG_TESTBENCH ( D, Z, E_IN, E_OUT, Q1, Q2);
  output [31:0] D;
  output [31:0] Z;
  output [31:0] E_IN;
  input [31:0] Q1;
  input [31:0] Q2;
  input [31:0] E_OUT;

  reg [31:0] D_T;
  reg [31:0] Z_T;
  reg [31:0] E_IN_T;
  reg [31:0] Q1_T;
  reg [31:0] Q2_T;
  reg [31:0] E_OUT_T;
  integer flag ;
  integer loop1 ;

assign D=D_T;
assign Z= Z_T ;
assign E_OUT = E_OUT_T;
assign E_IN = E_IN_T;
initial begin

   flag=0;
   Z_T = 32'b00000000000000000000000000000000;
   D_T = 32'b00000000000000000000000000000000;

   E_IN_T = 0;

   loop1 = 0;
end
always begin
   #3; //Propagation
   loop1 = loop1 +1;
   $display (" Test %d Begin", loop1) ;
   E_IN_T <= 0;
   #10;
   E_IN_T <= 1;
   #10;
   E_IN_T <= 0;
   #10;
   E_IN_T <= 1;
   #10;
   D_T= 32'b 00101011101010111010100101100100;
```

```
D_T= 32'b 00101011101010111010100101100100;
$display ("The given input: %b,", D);
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
$display ("The received output1: %b", Q1);
$display ("The received output2: %b", Q2);
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
D_T= 32'b 11111111101010111010100101100100;
$display ("The given input: %b,",D);
```

```
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
$display ("The received output: %b", Q1);
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
D_T= 32'b 00101011101010111010100101111111;
$display ("The given input: %b,", D);
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
```

```
  E_IN_T <= 0;
  #10;
  E_IN_T <= 1;
  #10;
  $display ("The received output: %b", Q1);
//  $display ("The expected output: %b", Q1_T);
  $display ("The expected output: %b", D);
  E_IN_T <= 0;
  #10;
  E_IN_T <= 1;
  #10;
  E_IN_T <= 0;
  #10;
  E_IN_T <= 1;
  #10;

  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;
  D_T= 32'b 00101011101010111010100101100101;
  $display ("The given input: %b,", D);
  E_IN_T <= 0;
  #10;
  E_IN_T <= 1;
  #10;
  E_IN_T <= 0;
```

```
#10;
E_IN_T <= 1;
#10;
$display ("The received output: %b", Q1);
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;

loop1 = loop1 +1;
$display (" Test %d Begin", loop1) ;
D_T= 32'b 00101011111110111010100101100100;
$display ("The given input: %b,", D);
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
E_IN_T <= 0;
#10;
E_IN_T <= 1;
#10;
```

```
  $display ("The received output: %b", Q1);

  E_IN_T <= 0;
  #10;
  E_IN_T <= 1;
  #10;
  E_IN_T <= 0;
  #10;
  E_IN_T <= 1;
  #10;



end
endmodule
```

Listing I.8: Register 32-bits Testbench

## I.9 Reversible Register 32-bit Testbench 2

```
//Verilog HDL for "bxc7483_bxc_lib", "BXC_REG_TESTBENCH" "functional"


module BXC_REG_TESTBENCH ( D, Z, E_IN, E_OUT, Q1, Q2);
  output [31:0] D;
  output [31:0] Z;
  output [31:0] E_IN;
  input [31:0] Q1;
  input [31:0] Q2;
  input [31:0] E_OUT;

  reg [31:0] D_T;
  reg [31:0] Z_T;
  reg [31:0] E_IN_T;
  reg [31:0] Q1_T;
  reg [31:0] Q2_T;
  reg [31:0] E_OUT_T;
  reg     [31:0] clk;
  integer flag ;
  integer loop1 ;
  integer cnt;
//  assign Q2 = Q2_T;
// assign Q1 = Q1_T;
assign D=D_T;
assign Z= Z_T ;
assign E_OUT = E_OUT_T;
assign E_IN = clk;
initial begin

  flag=0;
  Z_T = 32'b00000000000000000000000000000000;
  D_T = 32'b00000000000000000000000000000000;

  clk = 0;
  cnt = 0;
  loop1 = 0;
end
always @(posedge clk)

begin
  #3; //Propagation
  loop1 = loop1 +1;
  $display (" Test %d Begin", loop1) ;

  $display("Input = %d, Received Output = %d,", D, Q1);
  $display("Input = %b, Received Output = %b,", D, Q1);
  if (D != Q1)
```

```
      $display("ERROR: Result(%d) != Actual_Result(%d)\n", Q1, D)
          ;

   D_T <= ($random & 32'hffff);

   cnt <= cnt + 1;
   @(negedge clk);
   if (cnt > 30)
      $finish;



end
always
   #10 clk = ~clk;

endmodule
```

Listing I.9: Register 32-bits Testbench 2

## I.10 Reversible Multiply Accumulate Testbench

```
//Verilog HDL for "bxc7483_bxc_lib", "
    BXC_MultiplyAccumulate_TESTBENCH" "functional"


module BXC_MultiplyAccumulate_TESTBENCH (E_IN, Weights, X,
    Biases, E_OUT, Result, Result2 );
  output [31:0] E_IN;
  output [15:0] Weights;
  output [15:0] X;
  output [31:0] Biases;
  input  [31:0] E_OUT;
  input  [31:0] Result;
  input  [31:0] Result2;


  wire   [15:0] Weights;
  wire   [15:0] X;
  wire   [31:0] Biases;

  reg    [15:0] weights, x;
  reg    [31:0] biases;
  reg    [31:0] clk;

```

```

integer cnt;
wire [31:0] r = weights * x ;
wire [31:0] Actual_Result = r + biases;
assign Weights = weights;
assign X = x;
assign Biases = biases;
assign E_IN = clk;

initial
begin
  weights = 0;
  x = 0;
  biases = 0;
  clk = 0;
  cnt = 0;
end

always @(posedge clk)
begin
  $display("Weights = %d, X = %d, Biases = %d, Result = %d,
     Result2 = %d, Actual Result = %d", Weights, X, Biases,
     Result, Result2, Actual_Result);
  $display("Weights = %b, X = %b, Biases = %b, Result = %b,
     Result2 = %b, Actual Result = %b", Weights, X, Biases,
     Result, Result2, Actual_Result);
  if (Result != Actual_Result)
    $display("ERROR: Result(%d) != Actual_Result(%d)\n", Result
        , Actual_Result);

  weights <= ($random & 16'hff);
  x <= ($random & 16'hff);
  biases <= ($random & 32'hff);
  cnt <= cnt + 1;
 @(negedge clk);
  if (cnt > 30)
    $finish;
end

always
  #10 clk = ~clk;


endmodule
```

Listing I.10: Multiply Accumulate Testbench

## I.11 Reversible Multiply Accumulate Code

```
//systemVerilog HDL for "bxc7483_bxc_lib", "
    BXC_MultiplyAccumulate_Verilog"
//"systemVerilog"

import "DPI-C" function string getenv(input string env_name);

module BXC_MultiplyAccumulate_TESTBENCH ( E_IN, Weights, X,
    Biases, E_OUT, Result,
Result2 );

  output [31:0] E_IN;
  output [15:0] Weights;
  output [15:0] X;
  output [31:0] Biases;
  input  [31:0] E_OUT;
  input  [31:0] Result;
  input  [31:0] Result2;


  reg    [15:0] Weights;
  reg    [15:0] X;
  reg    [31:0] Biases;

  reg      [31:0] clk;

parameter samples = 16384;
parameter wb = 1024;

// simulation passes
parameter maxcnt = 128;

reg [15:0] weights_m [0:wb-1];
reg [15:0] x_m [0:samples-1];
reg [31:0] biases_m [0:wb-1];

integer cnt, i, errcnt;

wire [31:0] r = Weights * X ;
wire [31:0] Actual_Result = r + Biases;

assign E_IN = clk;

initial
begin
  $write("pwd = %s\n", getenv("PWD"));
  for(i = 0 ; i < samples ; i = i + 1)
  begin
    weights_m[i] = 0;
```

```
      x_m[i] = 0;
      biases_m[i] = 0;
    end
    Weights = 0;
    X = 0;
    Biases = 0;
    clk = 0;
    cnt = 0;
    errcnt = 0;
    $readmemh("../../../../../mnist/weights_m.t", weights_m);
    $readmemh("../../../../../mnist/x_m.t", x_m);
    $readmemh("../../../../../mnist/biases_m.t", biases_m);
    for(i = 0 ; i < 16 ; i = i + 1)
    begin
      $display("weights_m[%d] = %h", i, weights_m[i]);
      $display("x_m[%d] = %h", i, x_m[i]);
      $display("biases_m[%d] = %h", i, biases_m[i]);
    end
  end

  always @(posedge clk)
  begin
    $display("++++ Count = %d ++++", cnt);
    $display("Weights = %d, X = %d, Biases = %d, Result = %d,
        Result2 = %d, Actual Result = %d", Weights, X, Biases,
    Result, Result2, Actual_Result);
  $display("Weights = %b, X = %b, Biases = %b,\n Result = %b,
    Result2 = %b, Actual Result = %b", Weights, X, Biases,
    Result, Result2, Actual_Result);
  if (Result !== Actual_Result)
  begin
    $display("ERROR: Result(%d) != Actual_Result(%d)\n", Result
      , Actual_Result);
    errcnt = errcnt + 1;
  end
  $display("===========");

  Weights <= weights_m[cnt];
  X <= x_m[cnt];
  Biases <= biases_m[cnt];
  cnt <= cnt + 1;

  @(negedge clk);
  if (cnt > maxcnt)
  begin
    if (errcnt == 0)
    begin
      $display(">>>> TEST PASSED <<<");
    end
    else
```

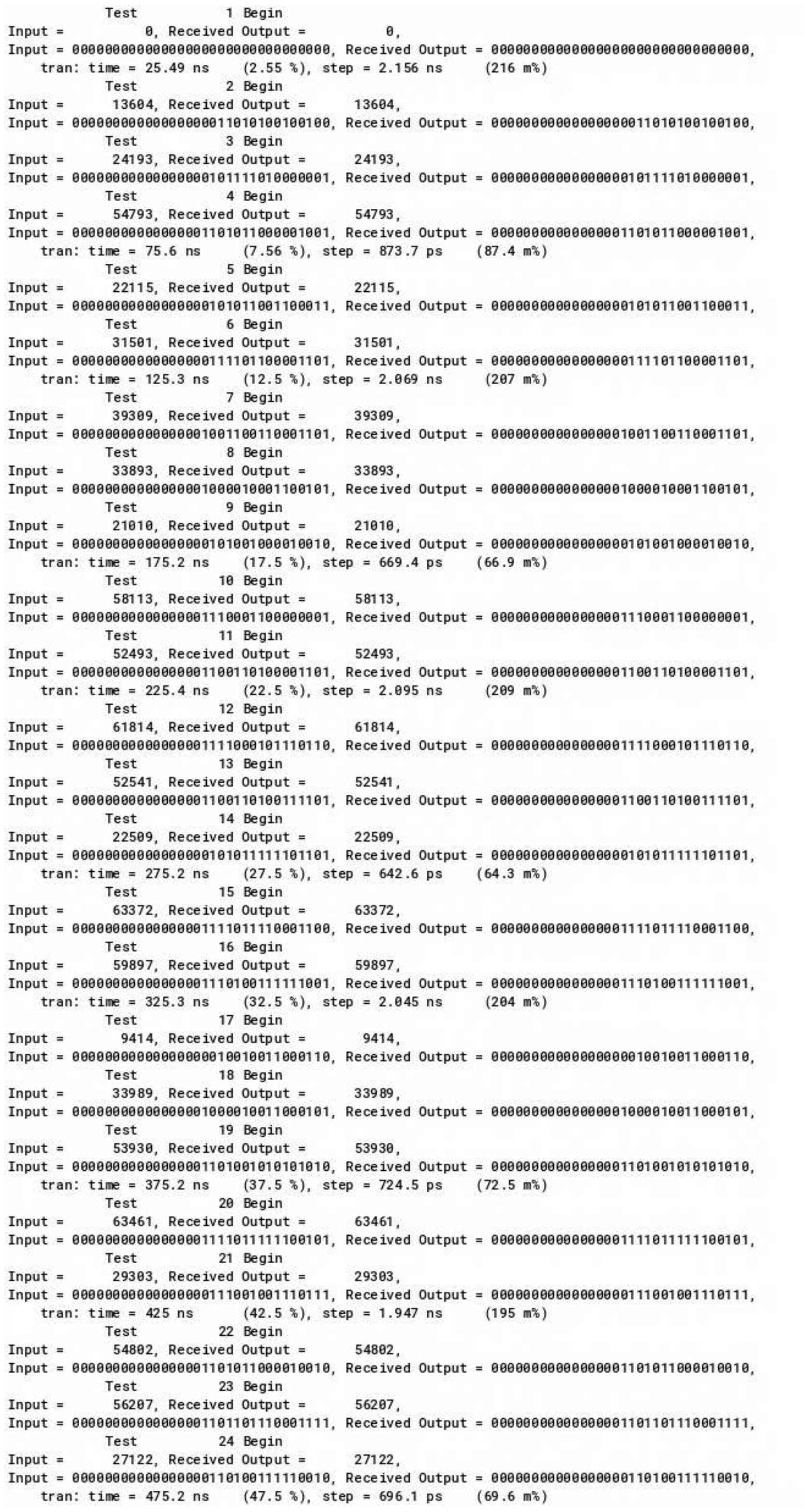
```
      begin
         $display(">>>> TEST FAILED <<<");
         $display(">>>> Error Count = %d <<<<", errcnt);
      end
      $finish;
   end
end

always
   #10 clk = ~clk;

endmodule // BXC_MultiplyAccumulate_TESTBENCH
```

Listing I.11: Multiply Accumulate Code

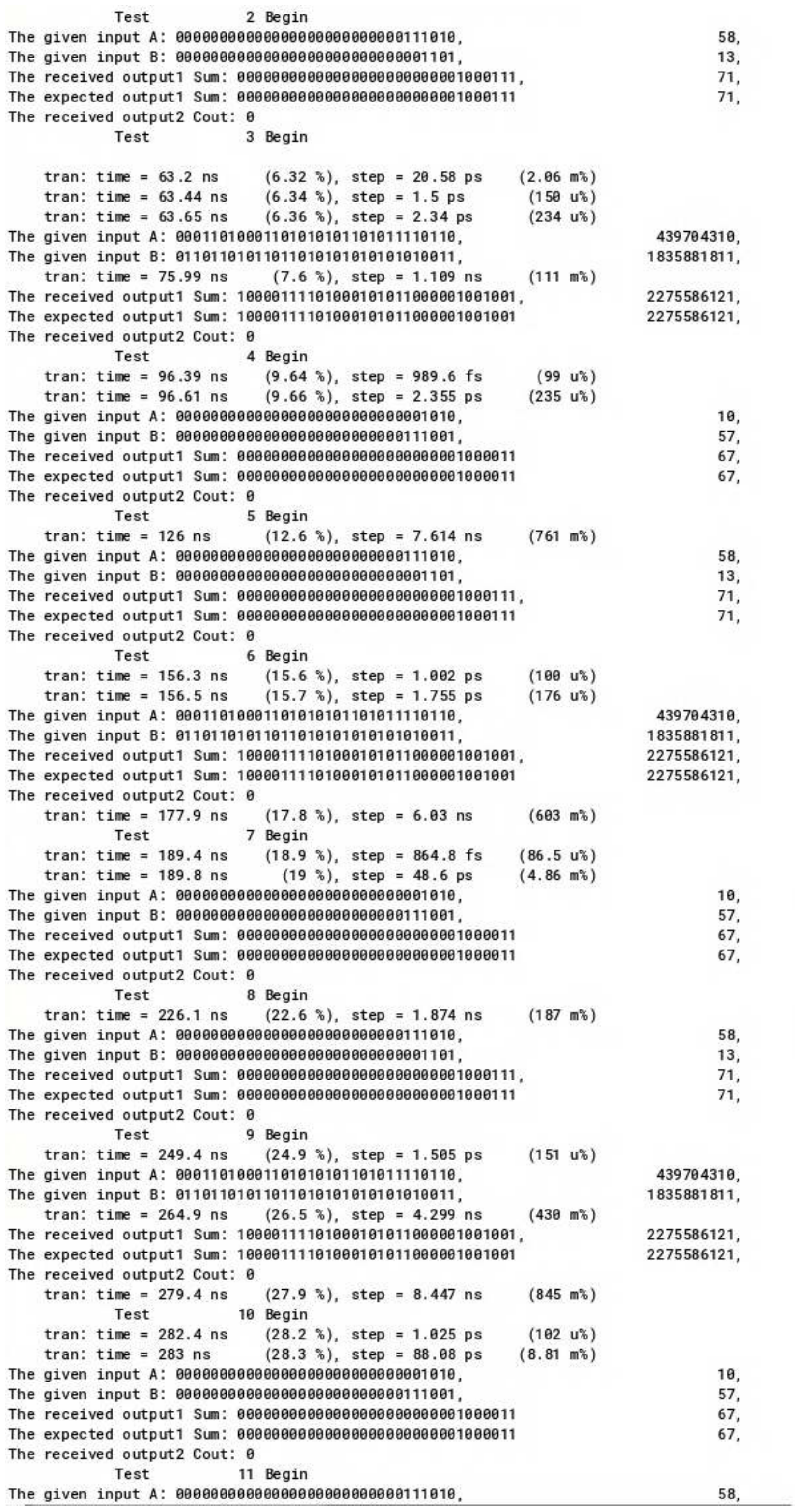
## I.12 Reversible Register 32-bit Results



```
            Test           1 Begin
Input =          0, Received Output =          0,
Input = 00000000000000000000000000000000, Received Output = 00000000000000000000000000000000,
    tran: time = 25.49 ns    (2.55 %), step = 2.156 ns     (216 m%)
            Test           2 Begin
Input =      13604, Received Output =      13604,
Input = 00000000000000000011010100100100, Received Output = 00000000000000000011010100100100,
            Test           3 Begin
Input =      24193, Received Output =      24193,
Input = 00000000000000000101111010000001, Received Output = 00000000000000000101111010000001,
            Test           4 Begin
Input =      54793, Received Output =      54793,
Input = 00000000000000001101011000001001, Received Output = 00000000000000001101011000001001,
    tran: time = 75.6 ns     (7.56 %), step = 873.7 ps    (87.4 m%)
            Test           5 Begin
Input =      22115, Received Output =      22115,
Input = 00000000000000000101011001100011, Received Output = 00000000000000000101011001100011,
            Test           6 Begin
Input =      31501, Received Output =      31501,
Input = 00000000000000000111101100001101, Received Output = 00000000000000000111101100001101,
    tran: time = 125.3 ns    (12.5 %), step = 2.069 ns     (207 m%)
            Test           7 Begin
Input =      39309, Received Output =      39309,
Input = 00000000000000001001100110001101, Received Output = 00000000000000001001100110001101,
            Test           8 Begin
Input =      33893, Received Output =      33893,
Input = 00000000000000001000010001100101, Received Output = 00000000000000001000010001100101,
            Test           9 Begin
Input =      21010, Received Output =      21010,
Input = 00000000000000000101001000010010, Received Output = 00000000000000000101001000010010,
    tran: time = 175.2 ns    (17.5 %), step = 669.4 ps    (66.9 m%)
            Test          10 Begin
Input =      58113, Received Output =      58113,
Input = 00000000000000001110001100000001, Received Output = 00000000000000001110001100000001,
            Test          11 Begin
Input =      52493, Received Output =      52493,
Input = 00000000000000001100110100001101, Received Output = 00000000000000001100110100001101,
    tran: time = 225.4 ns    (22.5 %), step = 2.095 ns     (209 m%)
            Test          12 Begin
Input =      61814, Received Output =      61814,
Input = 00000000000000001111000101110110, Received Output = 00000000000000001111000101110110,
            Test          13 Begin
Input =      52541, Received Output =      52541,
Input = 00000000000000001100110100111101, Received Output = 00000000000000001100110100111101,
            Test          14 Begin
Input =      22509, Received Output =      22509,
Input = 00000000000000000101011111101101, Received Output = 00000000000000000101011111101101,
    tran: time = 275.2 ns    (27.5 %), step = 642.6 ps    (64.3 m%)
            Test          15 Begin
Input =      63372, Received Output =      63372,
Input = 00000000000000001111011110001100, Received Output = 00000000000000001111011110001100,
            Test          16 Begin
Input =      59897, Received Output =      59897,
Input = 00000000000000001110100111111001, Received Output = 00000000000000001110100111111001,
    tran: time = 325.3 ns    (32.5 %), step = 2.045 ns     (204 m%)
            Test          17 Begin
Input =       9414, Received Output =       9414,
Input = 00000000000000000010010011000110, Received Output = 00000000000000000010010011000110,
            Test          18 Begin
Input =      33989, Received Output =      33989,
Input = 00000000000000001000010011000101, Received Output = 00000000000000001000010011000101,
            Test          19 Begin
Input =      53930, Received Output =      53930,
Input = 00000000000000001101001010101010, Received Output = 00000000000000001101001010101010,
    tran: time = 375.2 ns    (37.5 %), step = 724.5 ps    (72.5 m%)
            Test          20 Begin
Input =      63461, Received Output =      63461,
Input = 00000000000000001111011111100101, Received Output = 00000000000000001111011111100101,
            Test          21 Begin
Input =      29303, Received Output =      29303,
Input = 00000000000000000111001001110111, Received Output = 00000000000000000111001001110111,
    tran: time = 425 ns      (42.5 %), step = 1.947 ns     (195 m%)
            Test          22 Begin
Input =      54802, Received Output =      54802,
Input = 00000000000000001101011000010010, Received Output = 00000000000000001101011000010010,
            Test          23 Begin
Input =      56207, Received Output =      56207,
Input = 00000000000000001101101110001111, Received Output = 00000000000000001101101110001111,
            Test          24 Begin
Input =      27122, Received Output =      27122,
Input = 00000000000000000110100111110010, Received Output = 00000000000000000110100111110010,
    tran: time = 475.2 ns    (47.5 %), step = 696.1 ps    (69.6 m%)
```

Figure I.1: Reversible 32-bit Register Simulation Results

## I.13 Reversible Carry Look Ahead Adder 32-bit Results

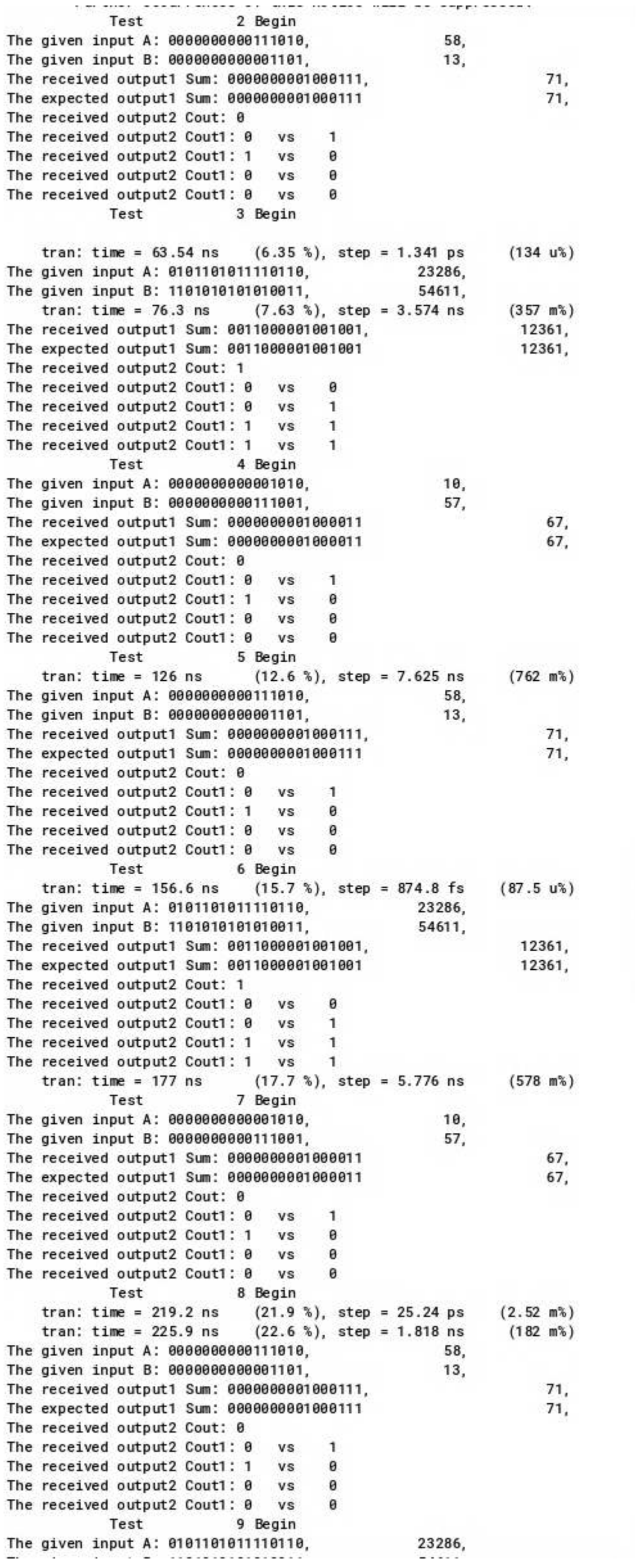


```
            Test           2 Begin
The given input A: 00000000000000000000000000111010,                       58,
The given input B: 00000000000000000000000000001101,                       13,
The received output1 Sum: 00000000000000000000000001000111,                71,
The expected output1 Sum: 00000000000000000000000001000111                 71,
The received output2 Cout: 0
            Test           3 Begin

    tran: time = 63.2 ns     (6.32 %), step = 20.58 ps    (2.06 m%)
    tran: time = 63.44 ns    (6.34 %), step = 1.5 ps       (150 u%)
    tran: time = 63.65 ns    (6.36 %), step = 2.34 ps      (234 u%)
The given input A: 00011010001101010101101011110110,                439704310,
The given input B: 01101101011011010101010101010011,               1835881811,
    tran: time = 75.99 ns     (7.6 %), step = 1.109 ns     (111 m%)
The received output1 Sum: 10000111101000101011000001001001,        2275586121,
The expected output1 Sum: 10000111101000101011000001001001         2275586121,
The received output2 Cout: 0
            Test           4 Begin
    tran: time = 96.39 ns    (9.64 %), step = 989.6 fs      (99 u%)
    tran: time = 96.61 ns    (9.66 %), step = 2.355 ps     (235 u%)
The given input A: 00000000000000000000000000001010,                       10,
The given input B: 00000000000000000000000000111001,                       57,
The received output1 Sum: 00000000000000000000000001000011                 67,
The expected output1 Sum: 00000000000000000000000001000011                 67,
The received output2 Cout: 0
            Test           5 Begin
    tran: time = 126 ns      (12.6 %), step = 7.614 ns     (761 m%)
The given input A: 00000000000000000000000000111010,                       58,
The given input B: 00000000000000000000000000001101,                       13,
The received output1 Sum: 00000000000000000000000001000111,                71,
The expected output1 Sum: 00000000000000000000000001000111                 71,
The received output2 Cout: 0
            Test           6 Begin
    tran: time = 156.3 ns    (15.6 %), step = 1.002 ps     (100 u%)
    tran: time = 156.5 ns    (15.7 %), step = 1.755 ps     (176 u%)
The given input A: 00011010001101010101101011110110,                439704310,
The given input B: 01101101011011010101010101010011,               1835881811,
The received output1 Sum: 10000111101000101011000001001001,        2275586121,
The expected output1 Sum: 10000111101000101011000001001001         2275586121,
The received output2 Cout: 0
    tran: time = 177.9 ns    (17.8 %), step = 6.03 ns      (603 m%)
            Test           7 Begin
    tran: time = 189.4 ns    (18.9 %), step = 864.8 fs    (86.5 u%)
    tran: time = 189.8 ns      (19 %), step = 48.6 ps     (4.86 m%)
The given input A: 00000000000000000000000000001010,                       10,
The given input B: 00000000000000000000000000111001,                       57,
The received output1 Sum: 00000000000000000000000001000011                 67,
The expected output1 Sum: 00000000000000000000000001000011                 67,
The received output2 Cout: 0
            Test           8 Begin
    tran: time = 226.1 ns    (22.6 %), step = 1.874 ns     (187 m%)
The given input A: 00000000000000000000000000111010,                       58,
The given input B: 00000000000000000000000000001101,                       13,
The received output1 Sum: 00000000000000000000000001000111,                71,
The expected output1 Sum: 00000000000000000000000001000111                 71,
The received output2 Cout: 0
            Test           9 Begin
    tran: time = 249.4 ns    (24.9 %), step = 1.505 ps     (151 u%)
The given input A: 00011010001101010101101011110110,                439704310,
The given input B: 01101101011011010101010101010011,               1835881811,
    tran: time = 264.9 ns    (26.5 %), step = 4.299 ns     (430 m%)
The received output1 Sum: 10000111101000101011000001001001,        2275586121,
The expected output1 Sum: 10000111101000101011000001001001         2275586121,
The received output2 Cout: 0
    tran: time = 279.4 ns    (27.9 %), step = 8.447 ns     (845 m%)
            Test          10 Begin
    tran: time = 282.4 ns    (28.2 %), step = 1.025 ps     (102 u%)
    tran: time = 283 ns      (28.3 %), step = 88.08 ps    (8.81 m%)
The given input A: 00000000000000000000000000001010,                       10,
The given input B: 00000000000000000000000000111001,                       57,
The received output1 Sum: 00000000000000000000000001000011                 67,
The expected output1 Sum: 00000000000000000000000001000011                 67,
The received output2 Cout: 0
            Test          11 Begin
The given input A: 00000000000000000000000000111010,                       58,
```

Figure I.2: Reversible 32-bit Carry Look Ahead Adder Simulation Results

## I.14 Reversible Carry Look Ahead Adder 16-bit Results

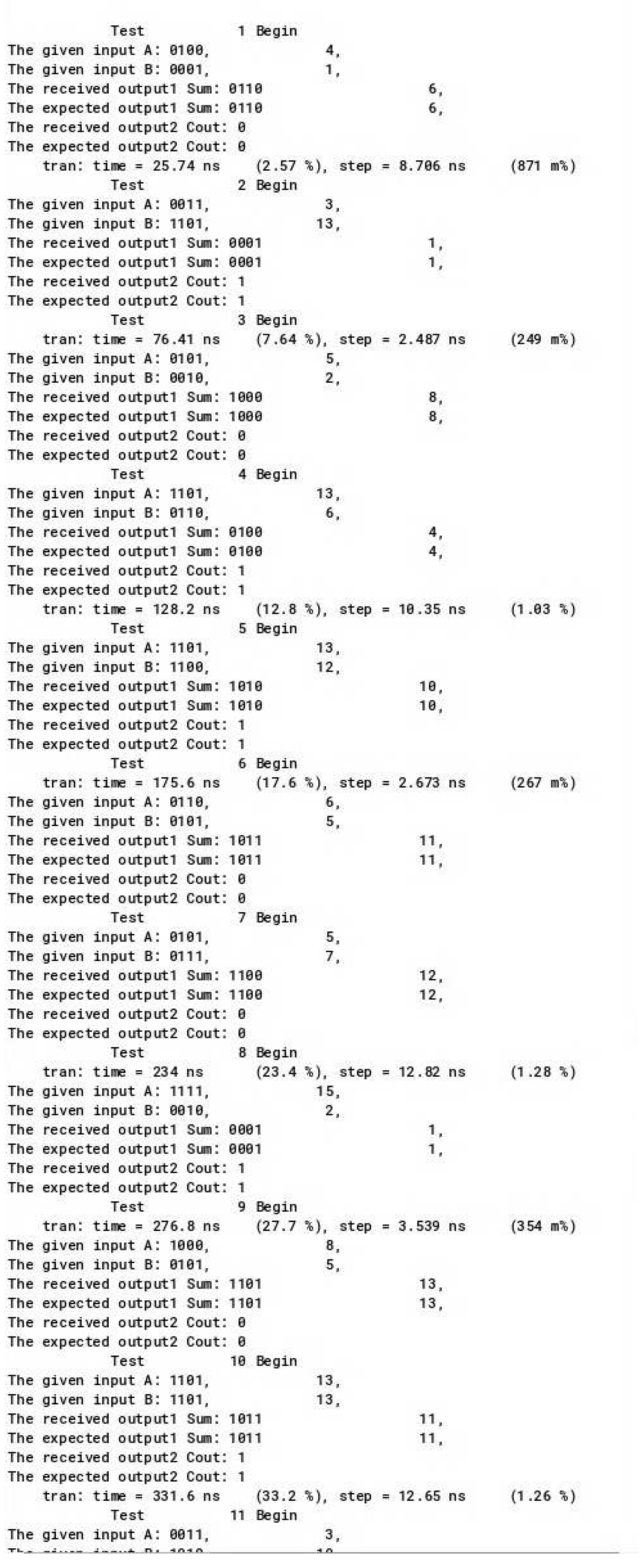


```
            Test          2 Begin
The given input A: 0000000000111010,              58,
The given input B: 0000000000001101,              13,
The received output1 Sum: 0000000001000111,                   71,
The expected output1 Sum: 0000000001000111                    71,
The received output2 Cout: 0
The received output2 Cout1: 0   vs    1
The received output2 Cout1: 1   vs    0
The received output2 Cout1: 0   vs    0
The received output2 Cout1: 0   vs    0
            Test          3 Begin

    tran: time = 63.54 ns    (6.35 %), step = 1.341 ps     (134 u%)
The given input A: 0101101011110110,           23286,
The given input B: 1101010101010011,           54611,
    tran: time = 76.3 ns     (7.63 %), step = 3.574 ns     (357 m%)
The received output1 Sum: 0011000001001001,                 12361,
The expected output1 Sum: 0011000001001001                  12361,
The received output2 Cout: 1
The received output2 Cout1: 0   vs    0
The received output2 Cout1: 0   vs    1
The received output2 Cout1: 1   vs    1
The received output2 Cout1: 1   vs    1
            Test          4 Begin
The given input A: 0000000000001010,              10,
The given input B: 0000000000111001,              57,
The received output1 Sum: 0000000001000011                    67,
The expected output1 Sum: 0000000001000011                    67,
The received output2 Cout: 0
The received output2 Cout1: 0   vs    1
The received output2 Cout1: 1   vs    0
The received output2 Cout1: 0   vs    0
The received output2 Cout1: 0   vs    0
            Test          5 Begin
    tran: time = 126 ns      (12.6 %), step = 7.625 ns     (762 m%)
The given input A: 0000000000111010,              58,
The given input B: 0000000000001101,              13,
The received output1 Sum: 0000000001000111,                   71,
The expected output1 Sum: 0000000001000111                    71,
The received output2 Cout: 0
The received output2 Cout1: 0   vs    1
The received output2 Cout1: 1   vs    0
The received output2 Cout1: 0   vs    0
The received output2 Cout1: 0   vs    0
            Test          6 Begin
    tran: time = 156.6 ns    (15.7 %), step = 874.8 fs    (87.5 u%)
The given input A: 0101101011110110,           23286,
The given input B: 1101010101010011,           54611,
The received output1 Sum: 0011000001001001,                 12361,
The expected output1 Sum: 0011000001001001                  12361,
The received output2 Cout: 1
The received output2 Cout1: 0   vs    0
The received output2 Cout1: 0   vs    1
The received output2 Cout1: 1   vs    1
The received output2 Cout1: 1   vs    1
    tran: time = 177 ns      (17.7 %), step = 5.776 ns     (578 m%)
            Test          7 Begin
The given input A: 0000000000001010,              10,
The given input B: 0000000000111001,              57,
The received output1 Sum: 0000000001000011                    67,
The expected output1 Sum: 0000000001000011                    67,
The received output2 Cout: 0
The received output2 Cout1: 0   vs    1
The received output2 Cout1: 1   vs    0
The received output2 Cout1: 0   vs    0
The received output2 Cout1: 0   vs    0
            Test          8 Begin
    tran: time = 219.2 ns    (21.9 %), step = 25.24 ps    (2.52 m%)
    tran: time = 225.9 ns    (22.6 %), step = 1.818 ns     (182 m%)
The given input A: 0000000000111010,              58,
The given input B: 0000000000001101,              13,
The received output1 Sum: 0000000001000111,                   71,
The expected output1 Sum: 0000000001000111                    71,
The received output2 Cout: 0
The received output2 Cout1: 0   vs    1
The received output2 Cout1: 1   vs    0
The received output2 Cout1: 0   vs    0
The received output2 Cout1: 0   vs    0
            Test          9 Begin
The given input A: 0101101011110110,           23286,
```

Figure I.3: Reversible 16-bit Carry Look Ahead Adder Simulation Results

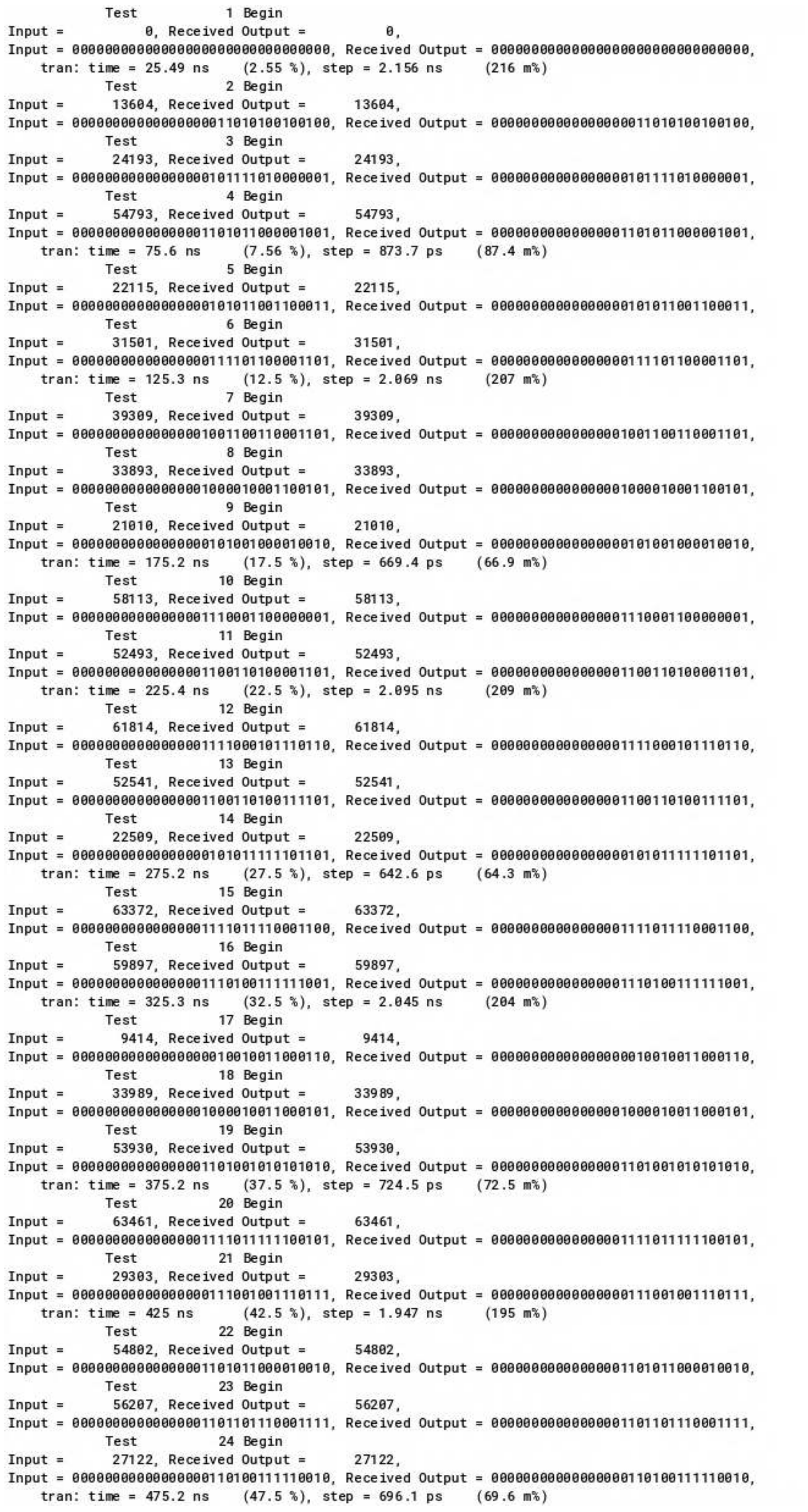
## I.15 Reversible Carry Look Ahead Adder 4-bit Results



```
          Test           1 Begin
The given input A: 0100,            4,
The given input B: 0001,            1,
The received output1 Sum: 0110                 6,
The expected output1 Sum: 0110                 6,
The received output2 Cout: 0
The expected output2 Cout: 0
    tran: time = 25.74 ns    (2.57 %), step = 8.706 ns     (871 m%)
          Test           2 Begin
The given input A: 0011,            3,
The given input B: 1101,           13,
The received output1 Sum: 0001                 1,
The expected output1 Sum: 0001                 1,
The received output2 Cout: 1
The expected output2 Cout: 1
          Test           3 Begin
    tran: time = 76.41 ns    (7.64 %), step = 2.487 ns     (249 m%)
The given input A: 0101,            5,
The given input B: 0010,            2,
The received output1 Sum: 1000                 8,
The expected output1 Sum: 1000                 8,
The received output2 Cout: 0
The expected output2 Cout: 0
          Test           4 Begin
The given input A: 1101,           13,
The given input B: 0110,            6,
The received output1 Sum: 0100                 4,
The expected output1 Sum: 0100                 4,
The received output2 Cout: 1
The expected output2 Cout: 1
    tran: time = 128.2 ns    (12.8 %), step = 10.35 ns     (1.03 %)
          Test           5 Begin
The given input A: 1101,           13,
The given input B: 1100,           12,
The received output1 Sum: 1010                10,
The expected output1 Sum: 1010                10,
The received output2 Cout: 1
The expected output2 Cout: 1
          Test           6 Begin
    tran: time = 175.6 ns    (17.6 %), step = 2.673 ns     (267 m%)
The given input A: 0110,            6,
The given input B: 0101,            5,
The received output1 Sum: 1011                11,
The expected output1 Sum: 1011                11,
The received output2 Cout: 0
The expected output2 Cout: 0
          Test           7 Begin
The given input A: 0101,            5,
The given input B: 0111,            7,
The received output1 Sum: 1100                12,
The expected output1 Sum: 1100                12,
The received output2 Cout: 0
The expected output2 Cout: 0
          Test           8 Begin
    tran: time = 234 ns      (23.4 %), step = 12.82 ns     (1.28 %)
The given input A: 1111,           15,
The given input B: 0010,            2,
The received output1 Sum: 0001                 1,
The expected output1 Sum: 0001                 1,
The received output2 Cout: 1
The expected output2 Cout: 1
          Test           9 Begin
    tran: time = 276.8 ns    (27.7 %), step = 3.539 ns     (354 m%)
The given input A: 1000,            8,
The given input B: 0101,            5,
The received output1 Sum: 1101                13,
The expected output1 Sum: 1101                13,
The received output2 Cout: 0
The expected output2 Cout: 0
          Test          10 Begin
The given input A: 1101,           13,
The given input B: 1101,           13,
The received output1 Sum: 1011                11,
The expected output1 Sum: 1011                11,
The received output2 Cout: 1
The expected output2 Cout: 1
    tran: time = 331.6 ns    (33.2 %), step = 12.65 ns     (1.26 %)
          Test          11 Begin
The given input A: 0011,            3,
```

Figure I.4: Reversible 4-bit Carry Look Ahead Adder Simulation Results

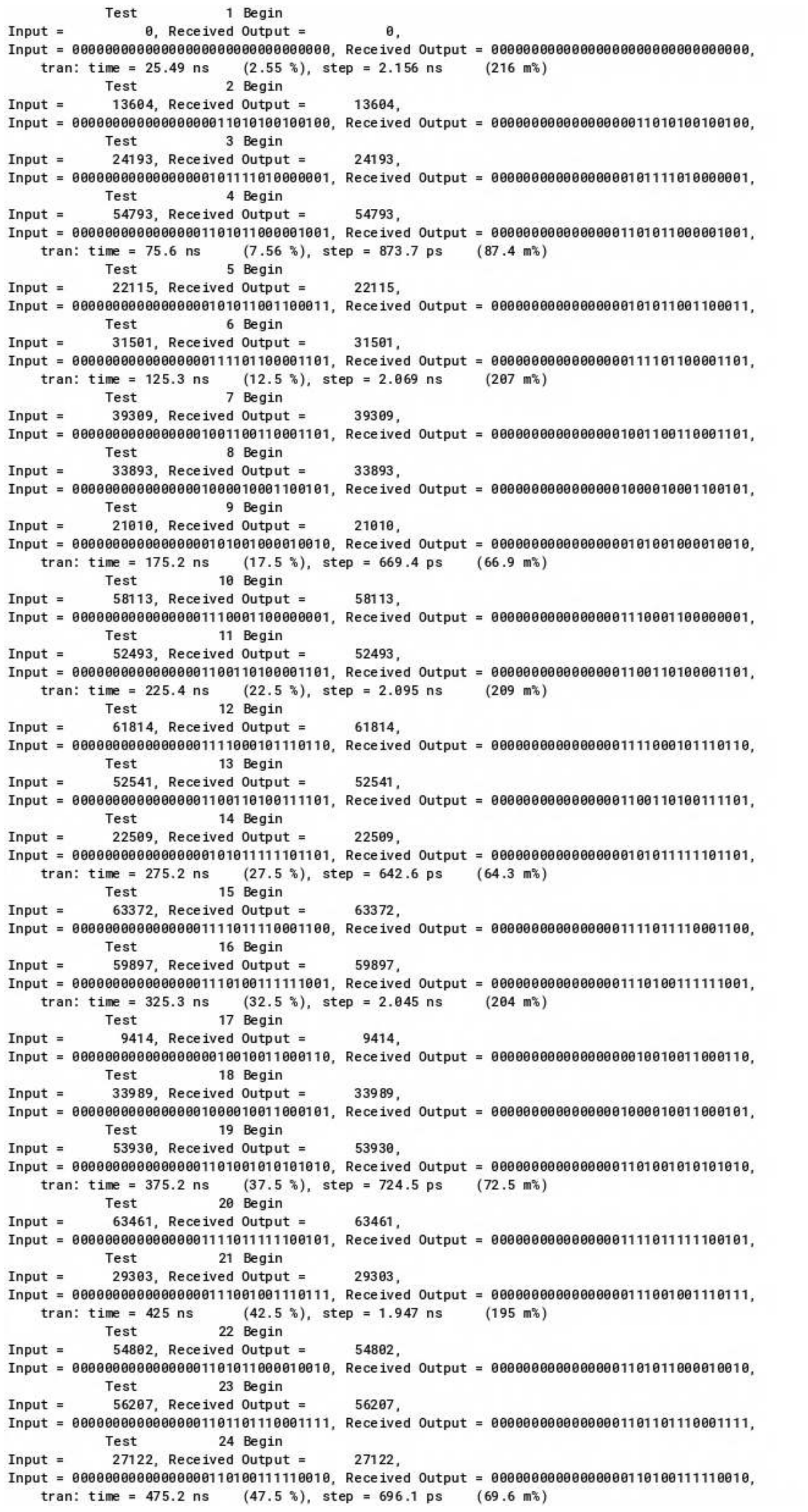
## I.16 Reversible 16-bit Multiplier Results



```
            Test           1 Begin
Input =          0, Received Output =          0,
Input = 00000000000000000000000000000000, Received Output = 00000000000000000000000000000000,
    tran: time = 25.49 ns    (2.55 %), step = 2.156 ns     (216 m%)
            Test           2 Begin
Input =      13604, Received Output =      13604,
Input = 00000000000000000011010100100100, Received Output = 00000000000000000011010100100100,
            Test           3 Begin
Input =      24193, Received Output =      24193,
Input = 00000000000000000101111010000001, Received Output = 00000000000000000101111010000001,
            Test           4 Begin
Input =      54793, Received Output =      54793,
Input = 00000000000000001101011000001001, Received Output = 00000000000000001101011000001001,
    tran: time = 75.6 ns     (7.56 %), step = 873.7 ps    (87.4 m%)
            Test           5 Begin
Input =      22115, Received Output =      22115,
Input = 00000000000000000101011001100011, Received Output = 00000000000000000101011001100011,
            Test           6 Begin
Input =      31501, Received Output =      31501,
Input = 00000000000000000111101100001101, Received Output = 00000000000000000111101100001101,
    tran: time = 125.3 ns    (12.5 %), step = 2.069 ns     (207 m%)
            Test           7 Begin
Input =      39309, Received Output =      39309,
Input = 00000000000000001001100110001101, Received Output = 00000000000000001001100110001101,
            Test           8 Begin
Input =      33893, Received Output =      33893,
Input = 00000000000000001000010001100101, Received Output = 00000000000000001000010001100101,
            Test           9 Begin
Input =      21010, Received Output =      21010,
Input = 00000000000000000101001000010010, Received Output = 00000000000000000101001000010010,
    tran: time = 175.2 ns    (17.5 %), step = 669.4 ps    (66.9 m%)
            Test          10 Begin
Input =      58113, Received Output =      58113,
Input = 00000000000000001110001100000001, Received Output = 00000000000000001110001100000001,
            Test          11 Begin
Input =      52493, Received Output =      52493,
Input = 00000000000000001100110100001101, Received Output = 00000000000000001100110100001101,
    tran: time = 225.4 ns    (22.5 %), step = 2.095 ns     (209 m%)
            Test          12 Begin
Input =      61814, Received Output =      61814,
Input = 00000000000000001111000101110110, Received Output = 00000000000000001111000101110110,
            Test          13 Begin
Input =      52541, Received Output =      52541,
Input = 00000000000000001100110100111101, Received Output = 00000000000000001100110100111101,
            Test          14 Begin
Input =      22509, Received Output =      22509,
Input = 00000000000000000101011111101101, Received Output = 00000000000000000101011111101101,
    tran: time = 275.2 ns    (27.5 %), step = 642.6 ps    (64.3 m%)
            Test          15 Begin
Input =      63372, Received Output =      63372,
Input = 00000000000000001111011110001100, Received Output = 00000000000000001111011110001100,
            Test          16 Begin
Input =      59897, Received Output =      59897,
Input = 00000000000000001110100111111001, Received Output = 00000000000000001110100111111001,
    tran: time = 325.3 ns    (32.5 %), step = 2.045 ns     (204 m%)
            Test          17 Begin
Input =       9414, Received Output =       9414,
Input = 00000000000000000010010011000110, Received Output = 00000000000000000010010011000110,
            Test          18 Begin
Input =      33989, Received Output =      33989,
Input = 00000000000000001000010011000101, Received Output = 00000000000000001000010011000101,
            Test          19 Begin
Input =      53930, Received Output =      53930,
Input = 00000000000000001101001010101010, Received Output = 00000000000000001101001010101010,
    tran: time = 375.2 ns    (37.5 %), step = 724.5 ps    (72.5 m%)
            Test          20 Begin
Input =      63461, Received Output =      63461,
Input = 00000000000000001111011111100101, Received Output = 00000000000000001111011111100101,
            Test          21 Begin
Input =      29303, Received Output =      29303,
Input = 00000000000000000111001001110111, Received Output = 00000000000000000111001001110111,
    tran: time = 425 ns      (42.5 %), step = 1.947 ns     (195 m%)
            Test          22 Begin
Input =      54802, Received Output =      54802,
Input = 00000000000000001101011000010010, Received Output = 00000000000000001101011000010010,
            Test          23 Begin
Input =      56207, Received Output =      56207,
Input = 00000000000000001101101110001111, Received Output = 00000000000000001101101110001111,
            Test          24 Begin
Input =      27122, Received Output =      27122,
Input = 00000000000000000110100111110010, Received Output = 00000000000000000110100111110010,
    tran: time = 475.2 ns    (47.5 %), step = 696.1 ps    (69.6 m%)
```

Figure I.5: Reversible 16-bit Multiplier Simulation Result

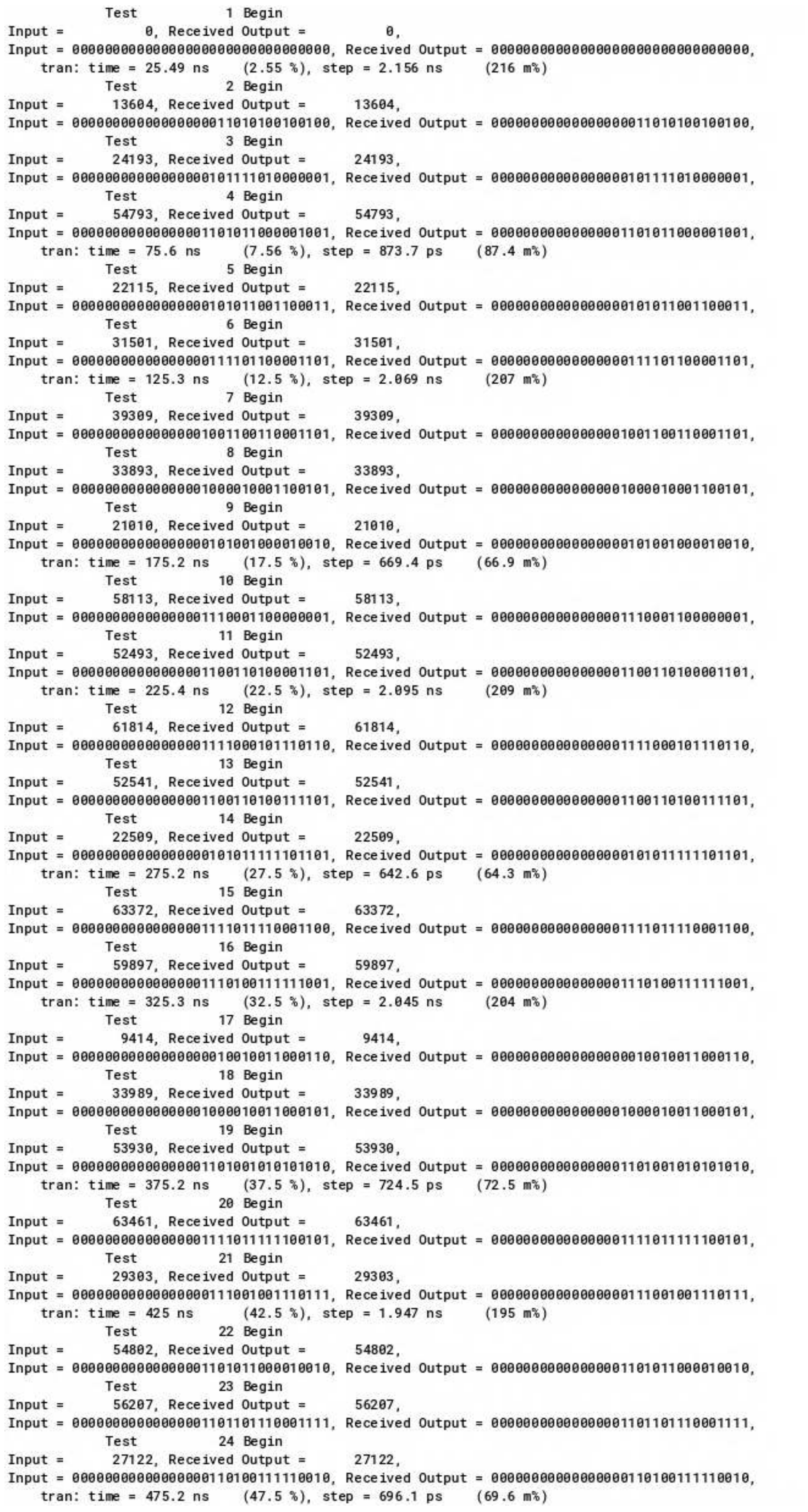
## I.17 Reversible 8-bit Multiplier Results



```
            Test           1 Begin
Input =          0, Received Output =          0,
Input = 00000000000000000000000000000000, Received Output = 00000000000000000000000000000000,
    tran: time = 25.49 ns    (2.55 %), step = 2.156 ns     (216 m%)
            Test           2 Begin
Input =      13604, Received Output =      13604,
Input = 00000000000000000011010100100100, Received Output = 00000000000000000011010100100100,
            Test           3 Begin
Input =      24193, Received Output =      24193,
Input = 00000000000000000101111010000001, Received Output = 00000000000000000101111010000001,
            Test           4 Begin
Input =      54793, Received Output =      54793,
Input = 00000000000000001101011000001001, Received Output = 00000000000000001101011000001001,
    tran: time = 75.6 ns     (7.56 %), step = 873.7 ps    (87.4 m%)
            Test           5 Begin
Input =      22115, Received Output =      22115,
Input = 00000000000000000101011001100011, Received Output = 00000000000000000101011001100011,
            Test           6 Begin
Input =      31501, Received Output =      31501,
Input = 00000000000000000111101100001101, Received Output = 00000000000000000111101100001101,
    tran: time = 125.3 ns    (12.5 %), step = 2.069 ns     (207 m%)
            Test           7 Begin
Input =      39309, Received Output =      39309,
Input = 00000000000000001001100110001101, Received Output = 00000000000000001001100110001101,
            Test           8 Begin
Input =      33893, Received Output =      33893,
Input = 00000000000000001000010001100101, Received Output = 00000000000000001000010001100101,
            Test           9 Begin
Input =      21010, Received Output =      21010,
Input = 00000000000000000101001000010010, Received Output = 00000000000000000101001000010010,
    tran: time = 175.2 ns    (17.5 %), step = 669.4 ps    (66.9 m%)
            Test          10 Begin
Input =      58113, Received Output =      58113,
Input = 00000000000000001110001100000001, Received Output = 00000000000000001110001100000001,
            Test          11 Begin
Input =      52493, Received Output =      52493,
Input = 00000000000000001100110100001101, Received Output = 00000000000000001100110100001101,
    tran: time = 225.4 ns    (22.5 %), step = 2.095 ns     (209 m%)
            Test          12 Begin
Input =      61814, Received Output =      61814,
Input = 00000000000000001111000101110110, Received Output = 00000000000000001111000101110110,
            Test          13 Begin
Input =      52541, Received Output =      52541,
Input = 00000000000000001100110100111101, Received Output = 00000000000000001100110100111101,
            Test          14 Begin
Input =      22509, Received Output =      22509,
Input = 00000000000000000101011111101101, Received Output = 00000000000000000101011111101101,
    tran: time = 275.2 ns    (27.5 %), step = 642.6 ps    (64.3 m%)
            Test          15 Begin
Input =      63372, Received Output =      63372,
Input = 00000000000000001111011110001100, Received Output = 00000000000000001111011110001100,
            Test          16 Begin
Input =      59897, Received Output =      59897,
Input = 00000000000000001110100111111001, Received Output = 00000000000000001110100111111001,
    tran: time = 325.3 ns    (32.5 %), step = 2.045 ns     (204 m%)
            Test          17 Begin
Input =       9414, Received Output =       9414,
Input = 00000000000000000010010011000110, Received Output = 00000000000000000010010011000110,
            Test          18 Begin
Input =      33989, Received Output =      33989,
Input = 00000000000000001000010011000101, Received Output = 00000000000000001000010011000101,
            Test          19 Begin
Input =      53930, Received Output =      53930,
Input = 00000000000000001101001010101010, Received Output = 00000000000000001101001010101010,
    tran: time = 375.2 ns    (37.5 %), step = 724.5 ps    (72.5 m%)
            Test          20 Begin
Input =      63461, Received Output =      63461,
Input = 00000000000000001111011111100101, Received Output = 00000000000000001111011111100101,
            Test          21 Begin
Input =      29303, Received Output =      29303,
Input = 00000000000000000111001001110111, Received Output = 00000000000000000111001001110111,
    tran: time = 425 ns      (42.5 %), step = 1.947 ns     (195 m%)
            Test          22 Begin
Input =      54802, Received Output =      54802,
Input = 00000000000000001101011000010010, Received Output = 00000000000000001101011000010010,
            Test          23 Begin
Input =      56207, Received Output =      56207,
Input = 00000000000000001101101110001111, Received Output = 00000000000000001101101110001111,
            Test          24 Begin
Input =      27122, Received Output =      27122,
Input = 00000000000000000110100111110010, Received Output = 00000000000000000110100111110010,
    tran: time = 475.2 ns    (47.5 %), step = 696.1 ps    (69.6 m%)
```

Figure I.6: Reversible 8-bit Multiplier Simulation Results

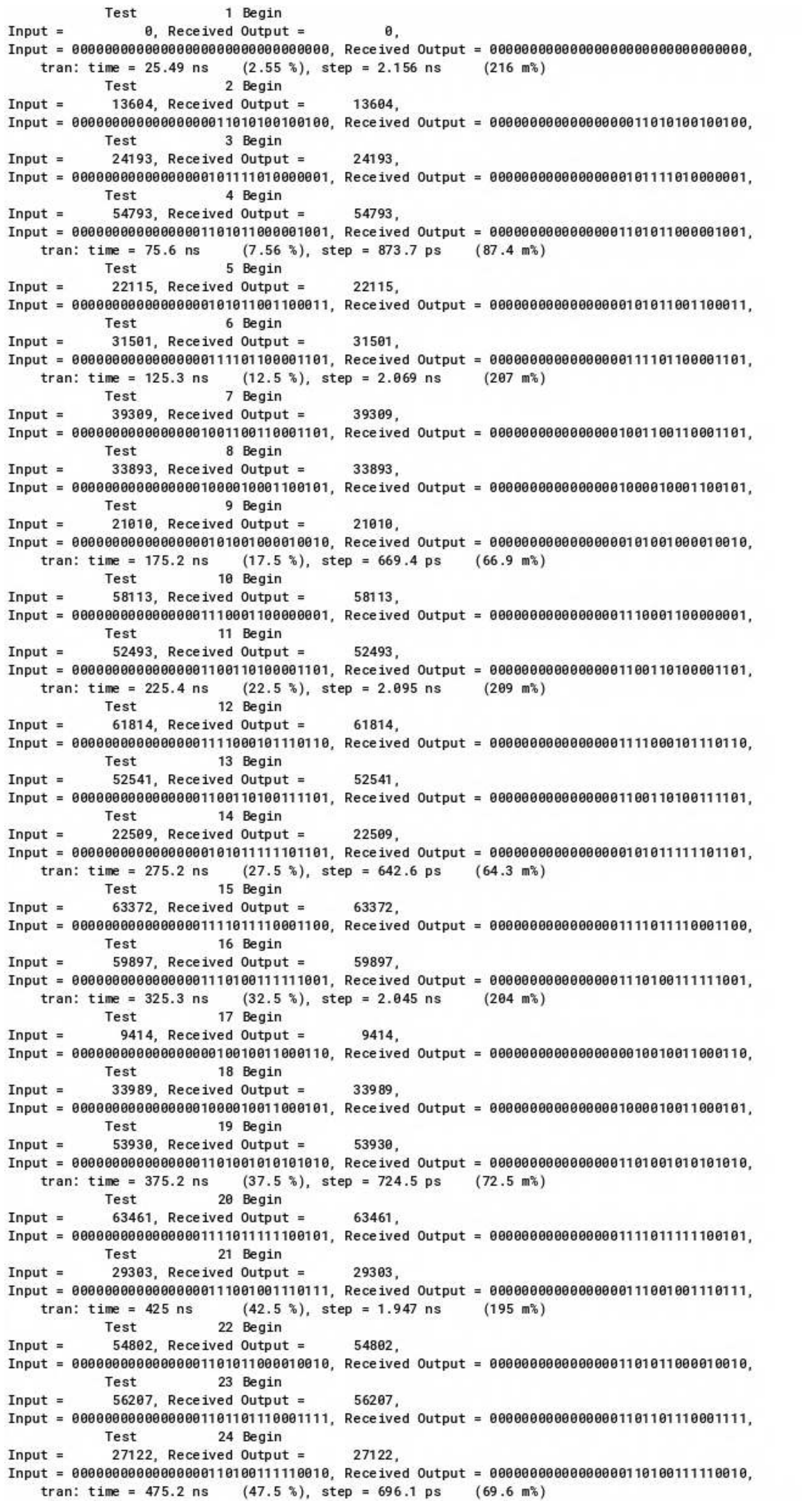
## I.18 Reversible 4-bit Multiplier Results



```
              Test           1 Begin
Input =          0, Received Output =          0,
Input = 00000000000000000000000000000000, Received Output = 00000000000000000000000000000000,
    tran: time = 25.49 ns    (2.55 %), step = 2.156 ns     (216 m%)
              Test           2 Begin
Input =      13604, Received Output =      13604,
Input = 00000000000000000011010100100100, Received Output = 00000000000000000011010100100100,
              Test           3 Begin
Input =      24193, Received Output =      24193,
Input = 00000000000000000101111010000001, Received Output = 00000000000000000101111010000001,
              Test           4 Begin
Input =      54793, Received Output =      54793,
Input = 00000000000000001101011000001001, Received Output = 00000000000000001101011000001001,
    tran: time = 75.6 ns     (7.56 %), step = 873.7 ps    (87.4 m%)
              Test           5 Begin
Input =      22115, Received Output =      22115,
Input = 00000000000000000101011001100011, Received Output = 00000000000000000101011001100011,
              Test           6 Begin
Input =      31501, Received Output =      31501,
Input = 00000000000000000111101100001101, Received Output = 00000000000000000111101100001101,
    tran: time = 125.3 ns    (12.5 %), step = 2.069 ns     (207 m%)
              Test           7 Begin
Input =      39309, Received Output =      39309,
Input = 00000000000000001001100110001101, Received Output = 00000000000000001001100110001101,
              Test           8 Begin
Input =      33893, Received Output =      33893,
Input = 00000000000000001000010001100101, Received Output = 00000000000000001000010001100101,
              Test           9 Begin
Input =      21010, Received Output =      21010,
Input = 00000000000000000101001000010010, Received Output = 00000000000000000101001000010010,
    tran: time = 175.2 ns    (17.5 %), step = 669.4 ps    (66.9 m%)
              Test          10 Begin
Input =      58113, Received Output =      58113,
Input = 00000000000000001110001100000001, Received Output = 00000000000000001110001100000001,
              Test          11 Begin
Input =      52493, Received Output =      52493,
Input = 00000000000000001100110100001101, Received Output = 00000000000000001100110100001101,
    tran: time = 225.4 ns    (22.5 %), step = 2.095 ns     (209 m%)
              Test          12 Begin
Input =      61814, Received Output =      61814,
Input = 00000000000000001111000101110110, Received Output = 00000000000000001111000101110110,
              Test          13 Begin
Input =      52541, Received Output =      52541,
Input = 00000000000000001100110100111101, Received Output = 00000000000000001100110100111101,
              Test          14 Begin
Input =      22509, Received Output =      22509,
Input = 00000000000000000101011111101101, Received Output = 00000000000000000101011111101101,
    tran: time = 275.2 ns    (27.5 %), step = 642.6 ps    (64.3 m%)
              Test          15 Begin
Input =      63372, Received Output =      63372,
Input = 00000000000000001111011110001100, Received Output = 00000000000000001111011110001100,
              Test          16 Begin
Input =      59897, Received Output =      59897,
Input = 00000000000000001110100111111001, Received Output = 00000000000000001110100111111001,
    tran: time = 325.3 ns    (32.5 %), step = 2.045 ns     (204 m%)
              Test          17 Begin
Input =       9414, Received Output =       9414,
Input = 00000000000000000010010011000110, Received Output = 00000000000000000010010011000110,
              Test          18 Begin
Input =      33989, Received Output =      33989,
Input = 00000000000000001000010011000101, Received Output = 00000000000000001000010011000101,
              Test          19 Begin
Input =      53930, Received Output =      53930,
Input = 00000000000000001101001010101010, Received Output = 00000000000000001101001010101010,
    tran: time = 375.2 ns    (37.5 %), step = 724.5 ps    (72.5 m%)
              Test          20 Begin
Input =      63461, Received Output =      63461,
Input = 00000000000000001111011111100101, Received Output = 00000000000000001111011111100101,
              Test          21 Begin
Input =      29303, Received Output =      29303,
Input = 00000000000000000111001001110111, Received Output = 00000000000000000111001001110111,
    tran: time = 425 ns      (42.5 %), step = 1.947 ns     (195 m%)
              Test          22 Begin
Input =      54802, Received Output =      54802,
Input = 00000000000000001101011000010010, Received Output = 00000000000000001101011000010010,
              Test          23 Begin
Input =      56207, Received Output =      56207,
Input = 00000000000000001101101110001111, Received Output = 00000000000000001101101110001111,
              Test          24 Begin
Input =      27122, Received Output =      27122,
Input = 00000000000000000110100111110010, Received Output = 00000000000000000110100111110010,
    tran: time = 475.2 ns    (47.5 %), step = 696.1 ps    (69.6 m%)
```

Figure I.7: Reversible 4-bit Multiplier Simulation Results

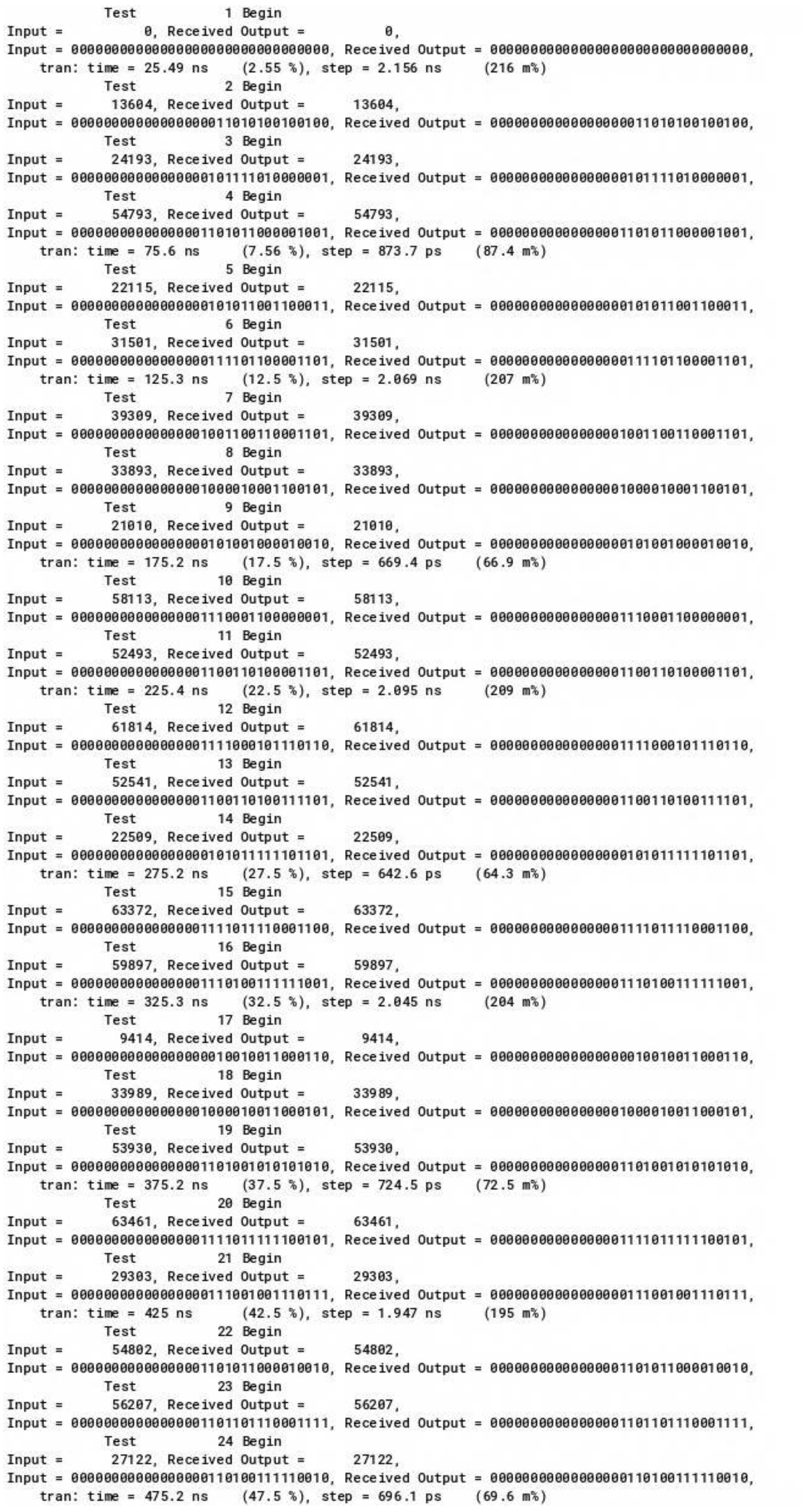
## I.19 Reversible 2-bit Multiplier Results



```
            Test           1 Begin
Input =          0, Received Output =          0,
Input = 00000000000000000000000000000000, Received Output = 00000000000000000000000000000000,
    tran: time = 25.49 ns    (2.55 %), step = 2.156 ns     (216 m%)
            Test           2 Begin
Input =      13604, Received Output =      13604,
Input = 00000000000000000011010100100100, Received Output = 00000000000000000011010100100100,
            Test           3 Begin
Input =      24193, Received Output =      24193,
Input = 00000000000000000101111010000001, Received Output = 00000000000000000101111010000001,
            Test           4 Begin
Input =      54793, Received Output =      54793,
Input = 00000000000000001101011000001001, Received Output = 00000000000000001101011000001001,
    tran: time = 75.6 ns     (7.56 %), step = 873.7 ps    (87.4 m%)
            Test           5 Begin
Input =      22115, Received Output =      22115,
Input = 00000000000000000101011001100011, Received Output = 00000000000000000101011001100011,
            Test           6 Begin
Input =      31501, Received Output =      31501,
Input = 00000000000000000111101100001101, Received Output = 00000000000000000111101100001101,
    tran: time = 125.3 ns    (12.5 %), step = 2.069 ns     (207 m%)
            Test           7 Begin
Input =      39309, Received Output =      39309,
Input = 00000000000000001001100110001101, Received Output = 00000000000000001001100110001101,
            Test           8 Begin
Input =      33893, Received Output =      33893,
Input = 00000000000000001000010001100101, Received Output = 00000000000000001000010001100101,
            Test           9 Begin
Input =      21010, Received Output =      21010,
Input = 00000000000000000101001000010010, Received Output = 00000000000000000101001000010010,
    tran: time = 175.2 ns    (17.5 %), step = 669.4 ps    (66.9 m%)
            Test          10 Begin
Input =      58113, Received Output =      58113,
Input = 00000000000000001110001100000001, Received Output = 00000000000000001110001100000001,
            Test          11 Begin
Input =      52493, Received Output =      52493,
Input = 00000000000000001100110100001101, Received Output = 00000000000000001100110100001101,
    tran: time = 225.4 ns    (22.5 %), step = 2.095 ns     (209 m%)
            Test          12 Begin
Input =      61814, Received Output =      61814,
Input = 00000000000000001111000101110110, Received Output = 00000000000000001111000101110110,
            Test          13 Begin
Input =      52541, Received Output =      52541,
Input = 00000000000000001100110100111101, Received Output = 00000000000000001100110100111101,
            Test          14 Begin
Input =      22509, Received Output =      22509,
Input = 00000000000000000101011111101101, Received Output = 00000000000000000101011111101101,
    tran: time = 275.2 ns    (27.5 %), step = 642.6 ps    (64.3 m%)
            Test          15 Begin
Input =      63372, Received Output =      63372,
Input = 00000000000000001111011110001100, Received Output = 00000000000000001111011110001100,
            Test          16 Begin
Input =      59897, Received Output =      59897,
Input = 00000000000000001110100111111001, Received Output = 00000000000000001110100111111001,
    tran: time = 325.3 ns    (32.5 %), step = 2.045 ns     (204 m%)
            Test          17 Begin
Input =       9414, Received Output =       9414,
Input = 00000000000000000010010011000110, Received Output = 00000000000000000010010011000110,
            Test          18 Begin
Input =      33989, Received Output =      33989,
Input = 00000000000000001000010011000101, Received Output = 00000000000000001000010011000101,
            Test          19 Begin
Input =      53930, Received Output =      53930,
Input = 00000000000000001101001010101010, Received Output = 00000000000000001101001010101010,
    tran: time = 375.2 ns    (37.5 %), step = 724.5 ps    (72.5 m%)
            Test          20 Begin
Input =      63461, Received Output =      63461,
Input = 00000000000000001111011111100101, Received Output = 00000000000000001111011111100101,
            Test          21 Begin
Input =      29303, Received Output =      29303,
Input = 00000000000000000111001001110111, Received Output = 00000000000000000111001001110111,
    tran: time = 425 ns      (42.5 %), step = 1.947 ns     (195 m%)
            Test          22 Begin
Input =      54802, Received Output =      54802,
Input = 00000000000000001101011000010010, Received Output = 00000000000000001101011000010010,
            Test          23 Begin
Input =      56207, Received Output =      56207,
Input = 00000000000000001101101110001111, Received Output = 00000000000000001101101110001111,
            Test          24 Begin
Input =      27122, Received Output =      27122,
Input = 00000000000000000110100111110010, Received Output = 00000000000000000110100111110010,
    tran: time = 475.2 ns    (47.5 %), step = 696.1 ps    (69.6 m%)
```

Figure I.8: Reversible 2-bit Multiplier Simulation Results

## I.20 Reversible Multiply Accumulate Results



```
            Test           1 Begin
Input =          0, Received Output =          0,
Input = 00000000000000000000000000000000, Received Output = 00000000000000000000000000000000,
    tran: time = 25.49 ns    (2.55 %), step = 2.156 ns     (216 m%)
            Test           2 Begin
Input =      13604, Received Output =      13604,
Input = 00000000000000000011010100100100, Received Output = 00000000000000000011010100100100,
            Test           3 Begin
Input =      24193, Received Output =      24193,
Input = 00000000000000000101111010000001, Received Output = 00000000000000000101111010000001,
            Test           4 Begin
Input =      54793, Received Output =      54793,
Input = 00000000000000001101011000001001, Received Output = 00000000000000001101011000001001,
    tran: time = 75.6 ns     (7.56 %), step = 873.7 ps    (87.4 m%)
            Test           5 Begin
Input =      22115, Received Output =      22115,
Input = 00000000000000000101011001100011, Received Output = 00000000000000000101011001100011,
            Test           6 Begin
Input =      31501, Received Output =      31501,
Input = 00000000000000000111101100001101, Received Output = 00000000000000000111101100001101,
    tran: time = 125.3 ns    (12.5 %), step = 2.069 ns     (207 m%)
            Test           7 Begin
Input =      39309, Received Output =      39309,
Input = 00000000000000001001100110001101, Received Output = 00000000000000001001100110001101,
            Test           8 Begin
Input =      33893, Received Output =      33893,
Input = 00000000000000001000010001100101, Received Output = 00000000000000001000010001100101,
            Test           9 Begin
Input =      21010, Received Output =      21010,
Input = 00000000000000000101001000010010, Received Output = 00000000000000000101001000010010,
    tran: time = 175.2 ns    (17.5 %), step = 669.4 ps    (66.9 m%)
            Test          10 Begin
Input =      58113, Received Output =      58113,
Input = 00000000000000001110001100000001, Received Output = 00000000000000001110001100000001,
            Test          11 Begin
Input =      52493, Received Output =      52493,
Input = 00000000000000001100110100001101, Received Output = 00000000000000001100110100001101,
    tran: time = 225.4 ns    (22.5 %), step = 2.095 ns     (209 m%)
            Test          12 Begin
Input =      61814, Received Output =      61814,
Input = 00000000000000001111000101110110, Received Output = 00000000000000001111000101110110,
            Test          13 Begin
Input =      52541, Received Output =      52541,
Input = 00000000000000001100110100111101, Received Output = 00000000000000001100110100111101,
            Test          14 Begin
Input =      22509, Received Output =      22509,
Input = 00000000000000000101011111101101, Received Output = 00000000000000000101011111101101,
    tran: time = 275.2 ns    (27.5 %), step = 642.6 ps    (64.3 m%)
            Test          15 Begin
Input =      63372, Received Output =      63372,
Input = 00000000000000001111011110001100, Received Output = 00000000000000001111011110001100,
            Test          16 Begin
Input =      59897, Received Output =      59897,
Input = 00000000000000001110100111111001, Received Output = 00000000000000001110100111111001,
    tran: time = 325.3 ns    (32.5 %), step = 2.045 ns     (204 m%)
            Test          17 Begin
Input =       9414, Received Output =       9414,
Input = 00000000000000000010010011000110, Received Output = 00000000000000000010010011000110,
            Test          18 Begin
Input =      33989, Received Output =      33989,
Input = 00000000000000001000010011000101, Received Output = 00000000000000001000010011000101,
            Test          19 Begin
Input =      53930, Received Output =      53930,
Input = 00000000000000001101001010101010, Received Output = 00000000000000001101001010101010,
    tran: time = 375.2 ns    (37.5 %), step = 724.5 ps    (72.5 m%)
            Test          20 Begin
Input =      63461, Received Output =      63461,
Input = 00000000000000001111011111100101, Received Output = 00000000000000001111011111100101,
            Test          21 Begin
Input =      29303, Received Output =      29303,
Input = 00000000000000000111001001110111, Received Output = 00000000000000000111001001110111,
    tran: time = 425 ns      (42.5 %), step = 1.947 ns     (195 m%)
            Test          22 Begin
Input =      54802, Received Output =      54802,
Input = 00000000000000001101011000010010, Received Output = 00000000000000001101011000010010,
            Test          23 Begin
Input =      56207, Received Output =      56207,
Input = 00000000000000001101101110001111, Received Output = 00000000000000001101101110001111,
            Test          24 Begin
Input =      27122, Received Output =      27122,
Input = 00000000000000000110100111110010, Received Output = 00000000000000000110100111110010,
    tran: time = 475.2 ns    (47.5 %), step = 696.1 ps    (69.6 m%)
```

Figure I.9: Reversible Multiply Accumulate Simulation Results