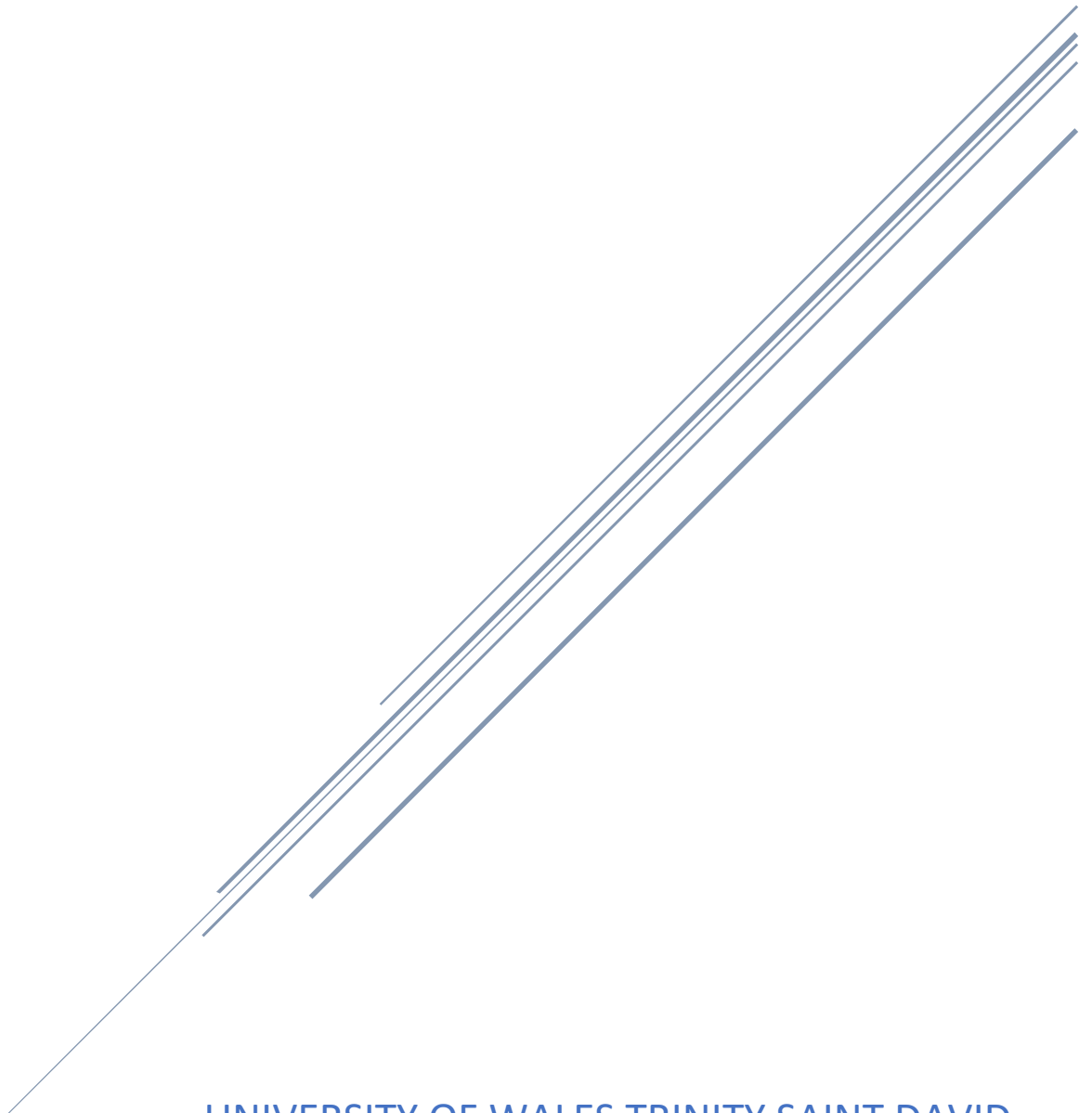


CRITICAL EVALUATION OF BRAUN'S MULTIPLIER AND WALLACE TREE MULTIPLIER USING VHDL

ASSIGNMENT 1

BY MICHAEL HOWELLS



UNIVERSITY OF WALES TRINITY SAINT DAVID
ELECTRONICS 2 – ACDA6001

CONTENTS

1	ABSTRACT.....	4
2	INTRODUCTION.....	5
3	BACKGROUND RESEARCH	6
3.1	DIGITAL SYSTEMS.....	6
3.2	SIGNED AND UNSIGNED NUMBERS	6
3.3	LOGIC GATES	6
3.3.1	AND GATE.....	7
3.3.2	OR GATE	7
3.3.3	XOR GATE	8
3.4	ADDERS	9
3.4.1	HALF ADDER.....	9
3.4.2	FULL ADDER.....	10
3.5	BINARY MULTIPLICATION.....	11
3.5.1	EXAMPLE MULTIPLICATON	11
3.5.2	BINARY MULTIPLICATION FUNCTION	11
3.6	BRAUN MULTIPLIER	12
3.7	WALLACE TREE MULTIPLIER.....	13
4	OTHER MULTIPLIER SCHEMES	13
4.1	VHDL.....	14
4.2	VIVADO.....	14
5	DESIGN BRIEF	15
6	BRAUN MULTIPLIER DESIGN	16
6.1	CIRCUIT DIAGRAM	17
6.2	VHDL CODE.....	18
6.3	ELABORATED DESIGN SCHEMATIC.....	18
6.4	SIMULATION RESULTS.....	19
6.5	SYNTHESIZED DESIGN SCHEMATIC	20
6.6	FPGA LAYOUT.....	21
7	WALLACE TREE MULTIPLIER DESIGN	22
7.1	DOT DIAGRAM	23
7.2	CIRCUIT DIAGRAM	24
7.3	VHDL CODE.....	24

7.4	ELABORATED DESIGN SCHEMATIC.....	25
7.5	SIMULATION RESULTS.....	26
7.6	SYNTHESIZED DESIGN SCHEMATIC	27
7.7	FPGA LAYOUT	28
8	COMPARISON.....	29
8.1	MAXIMUM AND MINIMUM DELAY.....	29
8.1.1	BRAUN MULTIPLIER LONGEST DELAY	30
8.1.2	WALLACE TREE MULTIPLIER LONGEST DELAY	31
8.2	TIMING CONSTRAINTS	32
8.3	BRAUN COMBINATIONAL DELAY	33
8.4	WALLACE TREE COMBINATIONAL DELAY	34
8.5	TIMING COMPARISON	35
8.6	POWER CONSUMPTION COMPARISON	37
9	CONCLUSION.....	40
10	RECCOMENDATIONS.....	41
11	APPENDICES	42
11.1	FULL DOCUMENT DOWNLOAD LINKS	42
11.2	HALF ADDER COMPONENT CODE	43
11.3	FULL ADDER COMPONENT CODE.....	44
11.4	BRAUN MULTIPLIER VHDL CODE.....	45
11.5	BRAUN MULTIPLIER TEST BENCH VHDL CODE.....	52
11.6	WALLACE MULTIPLIER VHDL CODE.....	54
11.7	WALLACE MULTIPLIER TEST BENCH VHDL CODE	62
11.8	IMPLEMENTATION POWER SETTINGS	64
11.9	BRAUN MULTIPLIER POWER SUMMARY.....	65
11.10	WALLACE TREE MULTIPLIER POWER SUMMARY	65
11.11	BRAUN MULTIPLIER UTILISATION SUMMARY	66
11.12	WALLACE TREE MULTIPLIER UTILISATION SUMMARY	66
11.13	BRAUN MULTIPLIER FULL COMBINATIONAL DELAY	67
11.14	WALLACE TREE FULL COMBINATIONAL DELAY	72
12	REFERENCES	77

1 ABSTRACT

This document details the process that was taken in designing 2 binary multipliers with different architecture (Braun and Wallace Tree). Each design was implemented in Xilinx Vivado using VHDL so that it could be simulated and then programmed into an FPGA.

Both types of multipliers were implemented successfully, and their characteristics were compared. The most notable difference between the two was the increased power consumption of the Wallace tree multiplier due to the increased number of LUTs required to implement its circuit.

Background research was undertaken to explain the 2 multiplying schemes.

2 INTRODUCTION

Binary multiplication is a fundamental function of computing and is used in graphics and signal processing devices. Fast multipliers are critical in systems where calculations must be computed at high speeds such as microprocessors and digital signal processors [1].

This document will explain the process of designing, simulating, and implementing two types of multipliers (Braun and Wallace Tree) using Vivado 2022.2. Each design will then be implemented onto a virtual FPGA so that it could be programmed onto a real device. Each multiplier circuit was defined and tested using VHDL.

The results of each simulation will then be compared against one another to determine the strengths and weaknesses of each design.

3 BACKGROUND RESEARCH

In this section, some fundamental concepts that are required to understand binary multipliers will be explained.

3.1 DIGITAL SYSTEMS

In digital systems, signals can be represented in 2 states. They are usually a known voltage level that is high or low and their actual value will depend on the logic system being implemented (3.3V (high) or 0V (low) for example) [2].

Any number can be represented using digits that only have two states. This can be described as having a base of 2 or a radix of 2. In this numbering system, each digit represents a different power of 2 with the first lowest bit as 2^0 . Table 1 shows how a binary bit related to a power and its weight.

Binary	Power of 2	Decimal
0001	2^0	1
0010	2^1	2
0100	2^2	4
1000	2^3	8

Table 1 – Binary Powers

To convert a binary number to decimal (base 10) the sum of each digits weight is calculated and then added together [3].

For example:

$$1011_2 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

$$1011_2 = 11_{10}$$

3.2 SIGNED AND UNSIGNED NUMBERS

There are schemes to represent both positive and negative numbers in binary. An unsigned number is one which can only represent a positive quantity. Negative numbers can only be represented using signed schemes such as sign/magnitude and two's complement [4].

3.3 LOGIC GATES

A logic gate is a circuit with one or more input signals and one output signal, their signals can only be binary logic states. The gates used in the multiplier circuits will be explained in this section. Each logic gate shown will have a Boolean expression which explains its function.

3.3.1 AND GATE

A basic and gate has 2 inputs. The output is only high when both inputs are high. Shown in Figure 1. And gates can have more than 2 inputs.

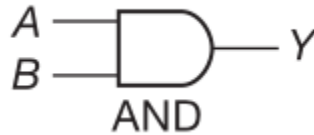


Figure 1 - AND gate symbol [5]

$$Y = AB$$

Equation 1 - AND gate Boolean expression [6]

A	B	OUTPUTS
0	0	0
0	1	0
1	0	0
1	1	1

Table 2 – AND gate truth table

3.3.2 OR GATE

A basic OR gate has 2 inputs, and the output is only high when 1 or both of the inputs are high. Shown in Figure 2.

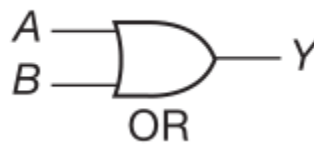


Figure 2 - OR gate symbol [5]

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Table 3 - OR gate truth table

$$Y = A + B$$

Equation 2 - OR gate Boolean expression [6]

3.3.3 XOR GATE

A basic XOR gate has 2 inputs. The output of the XOR gate is only high when only one of the inputs is high.



Figure 3 - XOR gate symbol [5]

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 4 - XOR gate truth table

$$Y = \bar{A}B + A\bar{B}$$

Or

$$Y = A \oplus B$$

Equation 3 - XOR gate Boolean expressions [6]

3.4 ADDERS

An adder is a digital circuit that is capable of adding two binary number together.

3.4.1 HALF ADDER

A half adder is capable of adding two bits. It has 2 inputs (A and B) and two outputs (Carry and Sum). It is made up of a single and gate and an XOR gate. Shown in Figure 4.

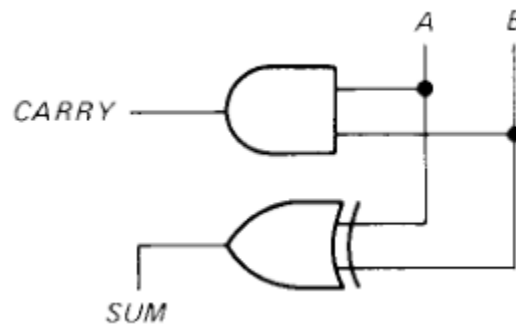


Figure 4 – Half adder circuit diagram [7]

A	B	CARRY	SUM
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 5 - Half adder truth table [7]

$$SUM = A \oplus B$$

$$CARRY = AB$$

Equation 4 - Half adder Boolean expressions [7]

3.4.2 FULL ADDER

A full adder is similar to the half adder in its function; however, it can add 3 inputs. The circuit is comprised of 3 AND gates, 1 OR gate and an XOR, shown in Figure 5.

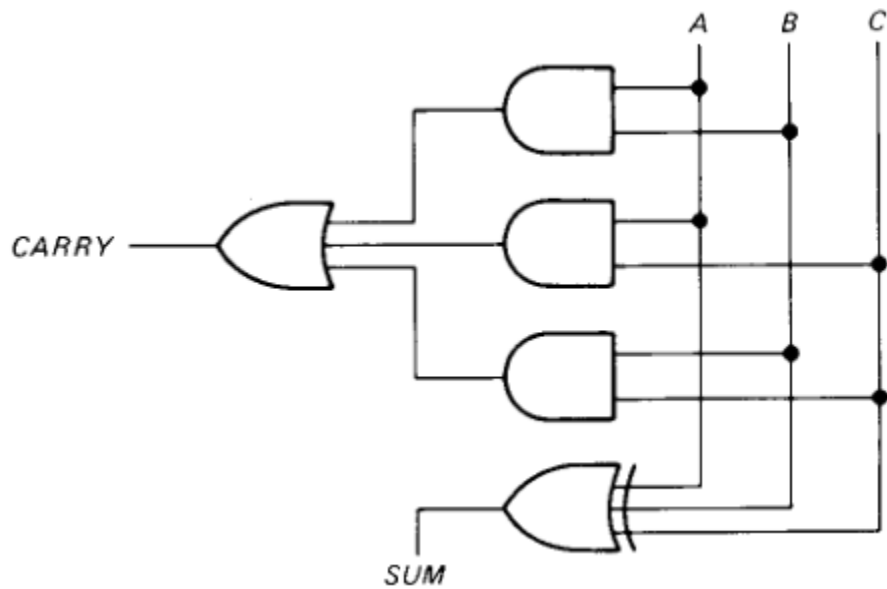


Figure 5 – Full adder circuit diagram [7]

A	B	C (Carry in)	CARRY (out)	SUM
0	0	0	0	1
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

Table 6 - Full adder truth table [7]

$$SUM = A \oplus B \oplus C$$

$$CARRY = AB + AC + BC$$

Equation 5 - Full adder Boolean expression [7]

Figure 7 - Binary multiplication function [8]

3.6 BRAUN MULTIPLIER

A Braun multiplier (also referred to as an array multiplier) is one implementation of a binary multiplier circuit. The design of a Braun multiplier follows the stages shown in the binary multiplication function shown in Figure 7.

The partial product bit will only be 1 if the multiplier and multiplicand bit is 1 therefore an AND gate can be used. The adding of partial products is done using half and full adders depending on the number of inputs at each addition stage. The carry bits of the addition are input to adders in the column to the left [8].

An example of a Braun multiplier can be seen in Figure 8, where it shows the array of adders required to add 2 4bit binary numbers.

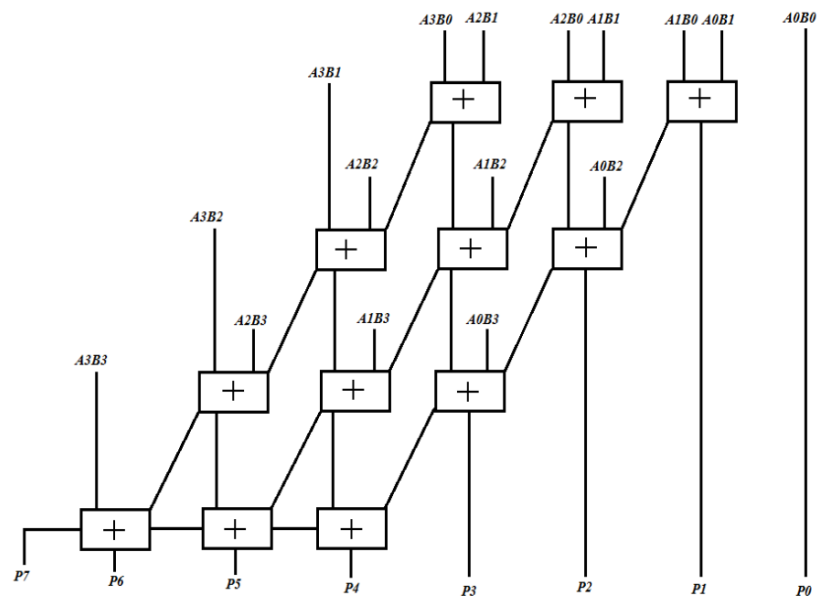


Figure 8 - 4bit x 4bit Braun multiplier [9]

3.7 WALLACE TREE MULTIPLIER

A Wallace tree multiplier is another binary multiplier architecture. The Wallace tree multiplier reduces the number of layers in the circuit by grouping partial products in 2s using half adders and 3s using full adders at each stage. This means that these calculations are done in parallel.

The reductions can be shown using a partial product dot matrix and can be seen in Figure 9.

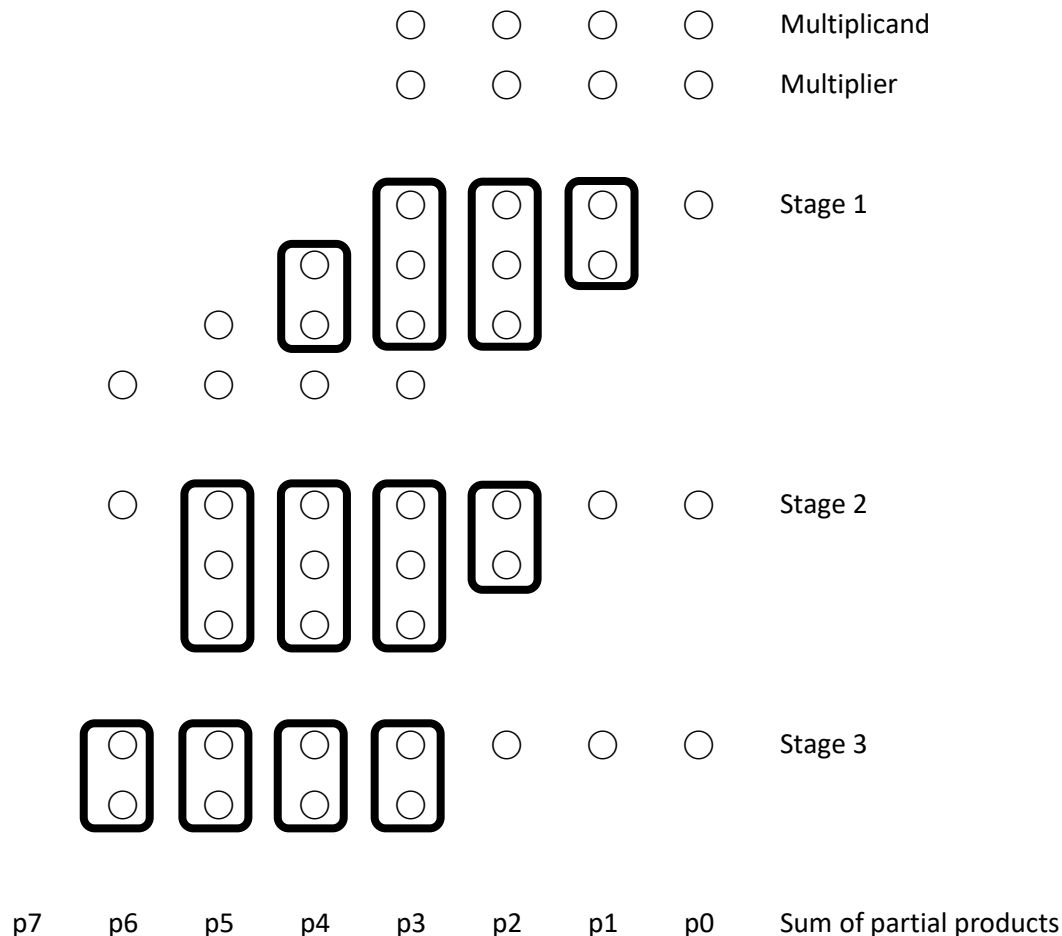


Figure 9 - Wallace tree dot matrix

The main benefit of the Wallace tree is its high speed due to the number of parallel calculation and the reduction in steps [10].

4 OTHER MULTIPLIER SCHEMES

There are more than two binary multiplication schemes. Some are able to multiply signed numbers such as the Booth multipliers and the Baugh Wooley multipliers.

There are also other tree multipliers such as the Dadda Tree multiplier which is potentially faster than the Wallace tree [11].

4.1 VHDL

VHDL (Very High-Speed Integrated Circuit Hardware Description Language) is a HDL (Hardware Description Language) that can be used to describe circuits. It is often used when designing circuits to be used in FPGA (Field Programmable Gate Array), which is an IC that can be programmed with custom logic [12]. The logic within an FPGA is configured using look up tables (LUT). A LUT is a device that contains both multiplexers and registers. They are used in FPGAs to implement any Boolean function [13].

The circuits in this report will be programmed using VHDL.

4.2 VIVADO

Vivado is a software package produced by Xilinx which uses HDLs to create and simulate circuits [14].

5 DESIGN BRIEF

The first brief is to:

“Design a $m \times n$ Parallel Array Braun’s Multiplier. The following instructions to be considered in designing the multiplier.

- The multiplier can be designed by modifying any two unsigned number.*
- The multiplier should consider, m and n - bits to be an 8-bit unsigned numbers for each input and provide a 16-bit unsigned partial product as the output.*
- Provide the complete logic circuit design.*
- Implement the components in VHDL and then integrate the components using port maps to build the full multiplier.*
- Simulate its timing and extract its schematics and results.*
- Synthesize the multiplier onto FPGA.”*

The second brief is to:

“A 8 x 8 bit unsigned number is to be designed using parallel multiplication for the diagram given below. Implement this using Wallace Tree Multiplier.

- Give a dot diagram of these multipliers, clearly indicating the number of Full Adders and Half Adders.*
- Provide the complete logic circuit design.*
- Implement the components in VHDL and then integrate the components using port maps to build the full multiplier.*
- Simulate its timing and extract its schematics and results.*
- Synthesize the multiplier onto FPGA.”*

6 BRAUN MULTIPLIER DESIGN

All schematics shown in this section have links for full PDFs in the appendices.

The first thing that was done in the design phase was to determine the partial products of multiplying two 8bit numbers together. The partial products can be seen in Figure 10.

								m7	m6	m5	m4	m3	m2	m1	m0
								n7	n6	n5	n4	n3	n2	n1	n0
								n0m7	n0m6	n0m5	n0m4	m0m3	n0m2	n0m1	n0m0
								n1m7	n1m6	n1m5	n1m4	m1m3	n1m2	n1m1	n1m0
								n2m7	n2m6	n2m5	n2m4	m2m3	n2m2	n2m1	n2m0
								n3m7	n3m6	n3m5	n3m4	m3m3	n3m2	n3m1	n3m0
								n4m7	n4m6	n4m5	n4m4	m4m3	n4m2	n4m1	n4m0
								n5m7	n5m6	n5m5	n5m4	m5m3	n5m2	n5m1	n5m0
								n6m7	n6m6	n6m5	n6m4	m6m3	n6m2	n6m1	n6m0
								n7m7	n7m6	n7m5	n7m4	m7m3	n7m2	n7m1	n7m0
p15	p14	p13	p12	p11	p10	p9	p8	p7	p6	p5	p4	p3	p2	p1	p0

Figure 10 - Partial products in 8bit Braun multiplier

6.1 CIRCUIT DIAGRAM

The next step was to design the circuit using the partial products shown in Figure 10. This was initially done on Proteus, a circuit design software package. The circuit was designed and then simulated to ensure it was functioning correctly. The circuit diagram can be seen in Figure 11.

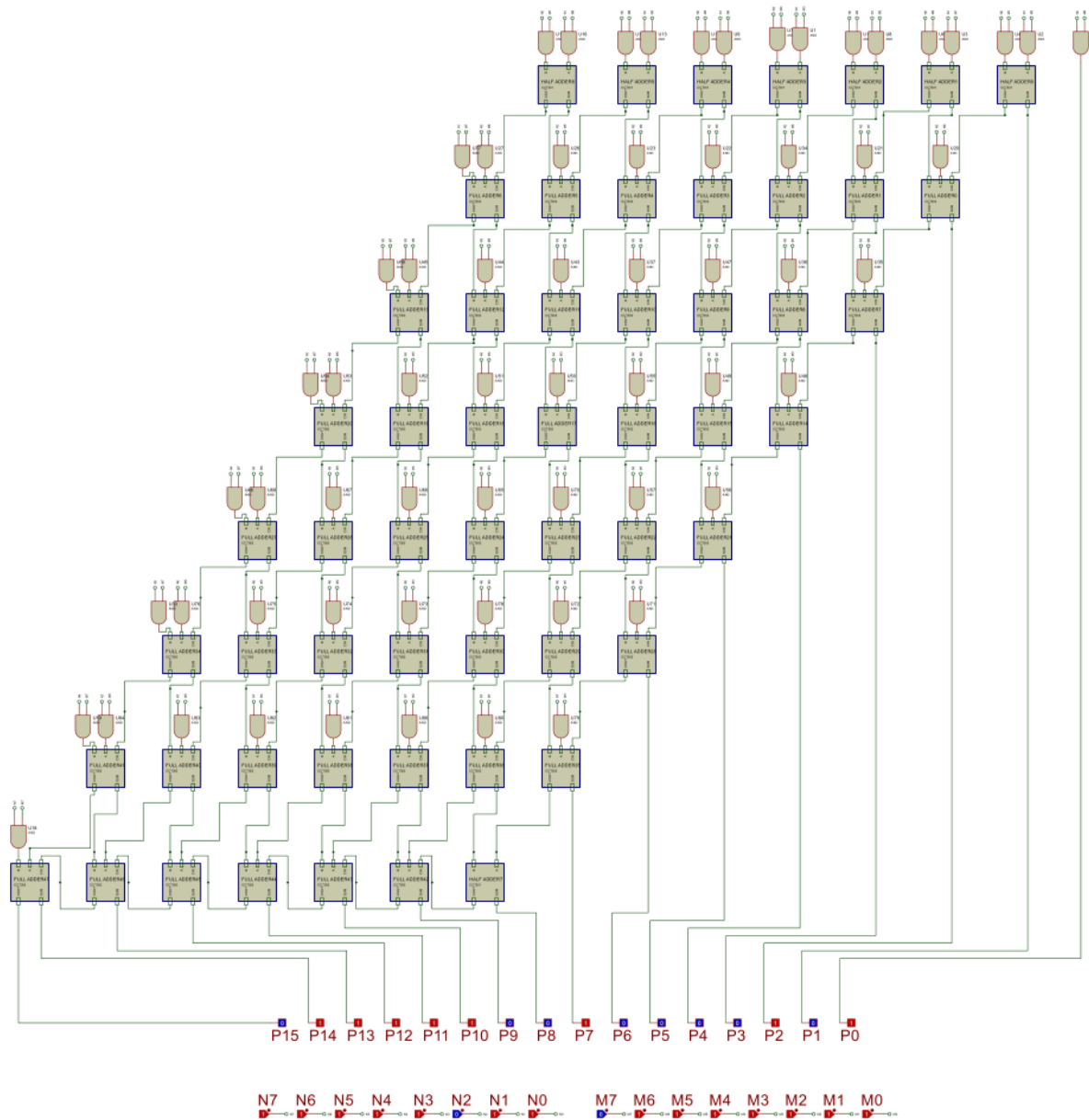


Figure 11 - Braun Circuit schematic in Proteus

To reduce the overall wires crossing over and improve readability of the circuit, net names were used to connect the input bits into the circuit. The circuit simulation was successful after numerous calculations were checked.

The calculation:

$$1111\ 1011_2 \times 0111\ 1111_2 = 0111\ 1100\ 1000\ 0101_2$$

$$251_{10} \times 127_{10} = 31877_{10}$$

can be seen in Figure 11.

6.2 VHDL CODE

The full VHDL code and project link can be found in the appendices.

The VHDL code was written using the structural modelling. Full and half adder components were created in VHDL and referenced in the multiplier VHDL document. The wiring was configured using port mapping.

6.3 ELABORATED DESIGN SCHEMATIC

The circuit described by the VHDL code can be seen in Figure 12. It shows the elaborated schematic generated using Vivado.

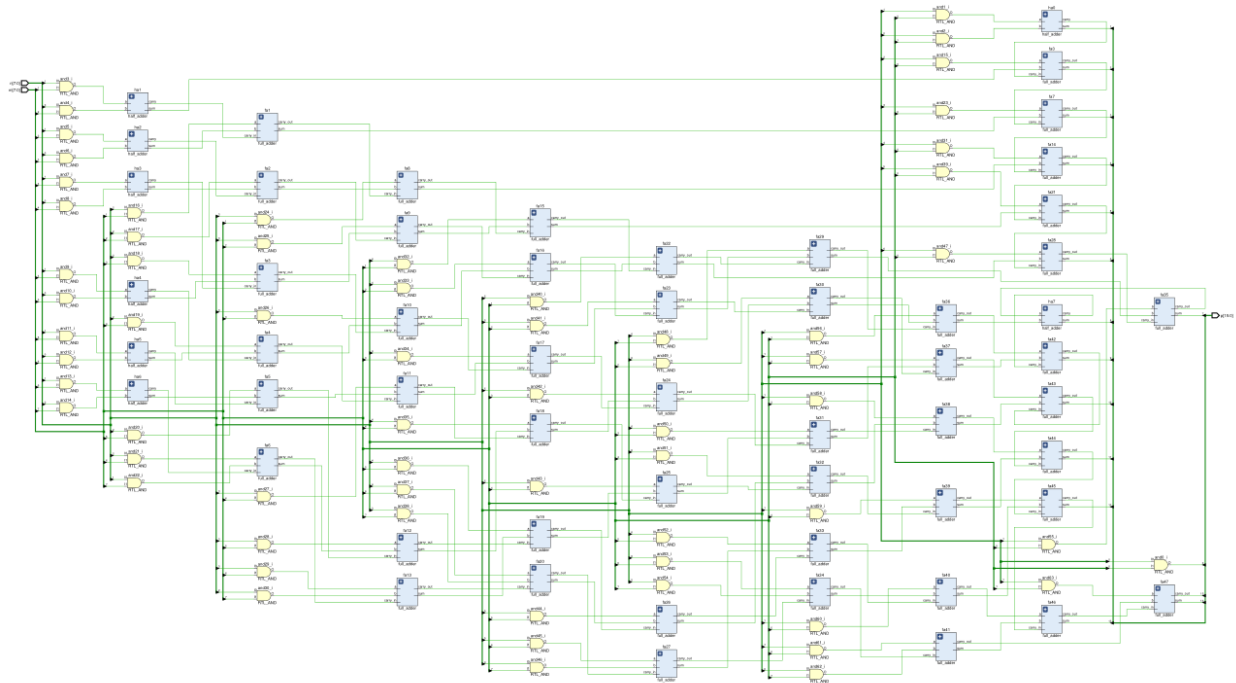


Figure 12 - Braun Circuit schematic in Vivado

6.4 SIMULATION RESULTS

The VHDL testbench code can be found in the appendices.

To validate the accuracy of the multiplier circuit, all combinations of input were simulated into the system. This was done using a nested for loop in the test bench VHDL. The outer loop increments a number from 0 – 254. Every instance of this count has a second loop that counts from 0-254. Each instance of each count is used as multiplier and multiplicand inputs into the system.

The output of the system was printed to the console and each calculation was checked using an excel spread sheet (link to excel shown in appendices). The calculations were duplicated using excel formulas and then the difference between the excel calculation and the simulation calculation was detected. If there was a difference, then it indicated an error in the multiplication. 0 errors were detected.

The simulation for the Braun multiplier took 1.31ms to multiply each combination of input.

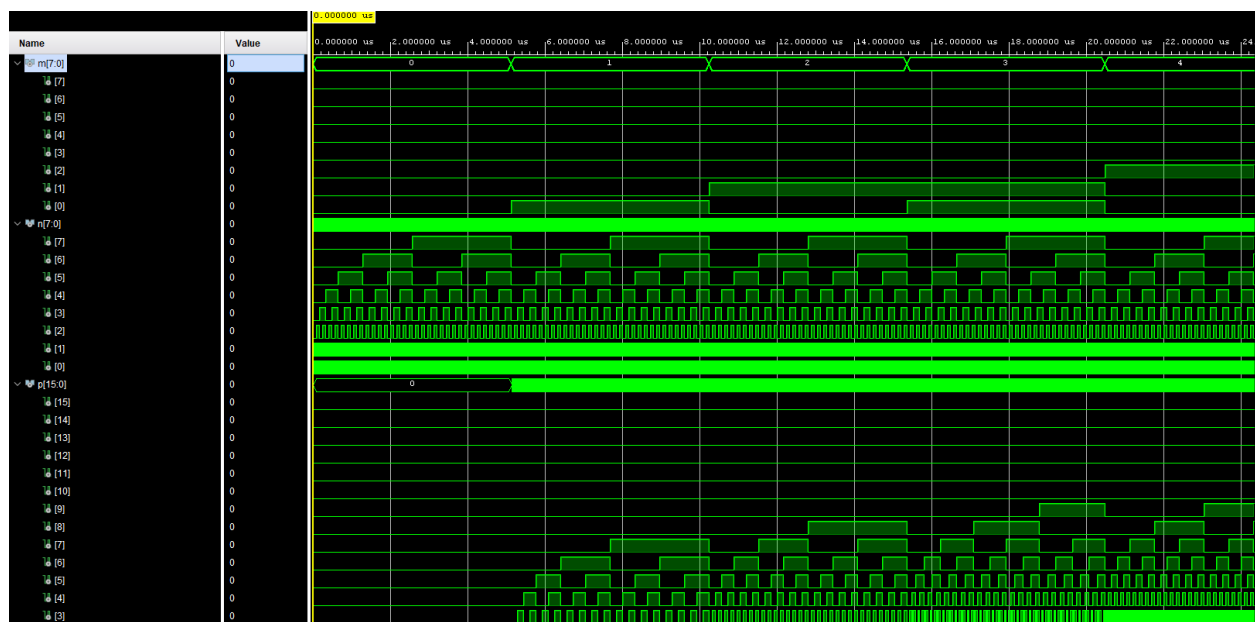


Figure 13 - Braun multiplier simulation sample

6.5 SYNTHESIZED DESIGN SCHEMATIC

The full synthesized schematic can be seen in Figure 14.

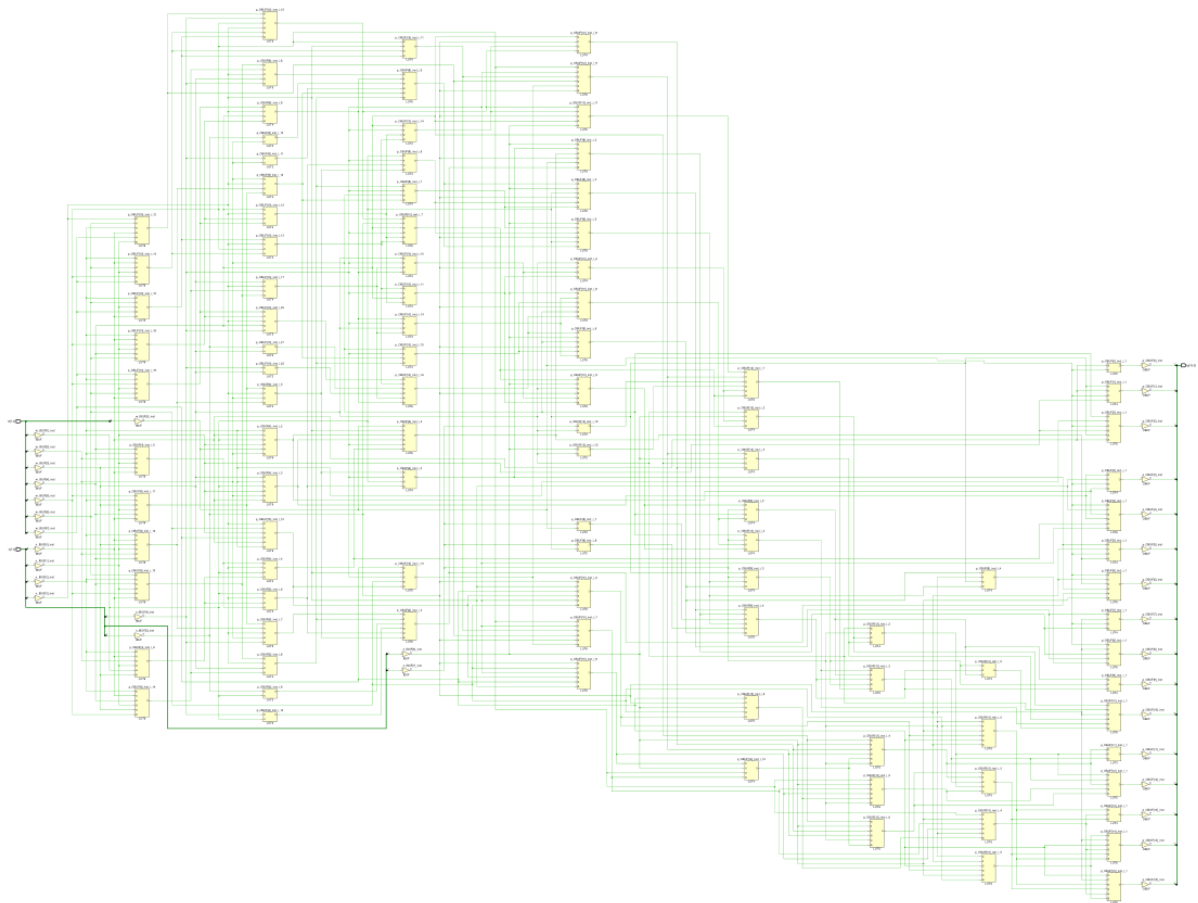


Figure 14 - Braun multiplier synthesized design schematic

6.6 FPGA LAYOUT

The FPGA layout of the Braun multiplier can be seen in Figure 15.

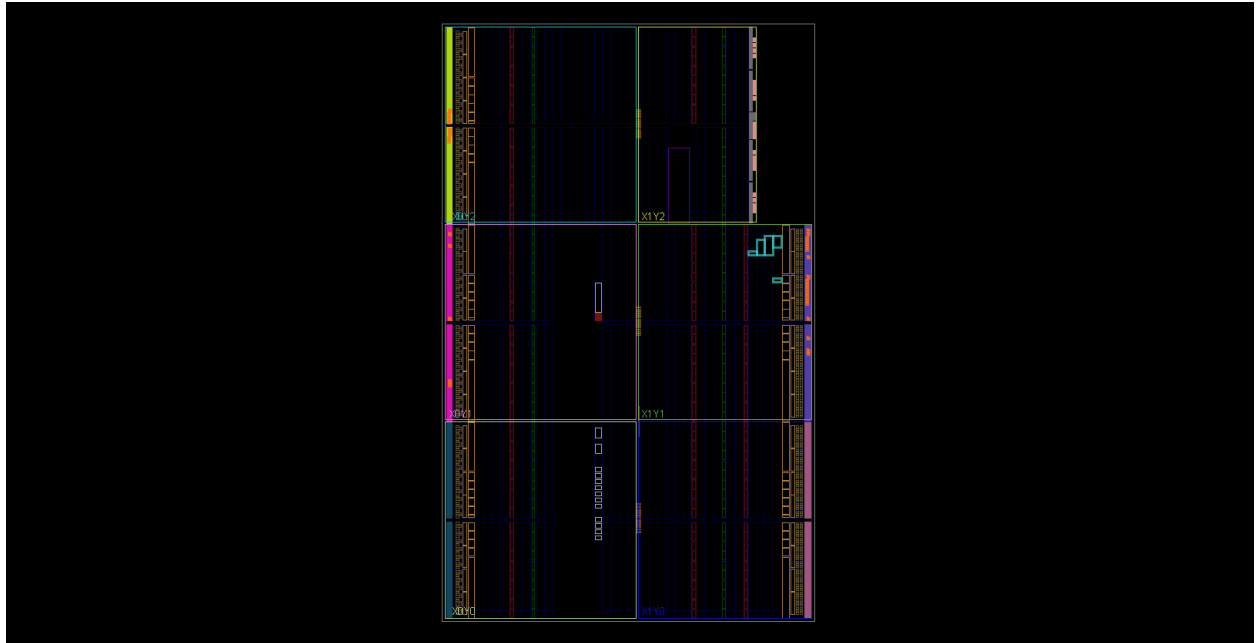


Figure 15 - Braun multiplier synthesized FPGA layout

7 WALLACE TREE MULTIPLIER DESIGN

All schematics shown in this section have links for full PDFs in appendices.

The first step in the Wallace Tree design is to identify the partial products. This is identical to what is shown in Figure 10.

7.1 DOT DIAGRAM

The next step is to work out the simplification of the partial products using a dot matrix. This can be seen in Figure 16.

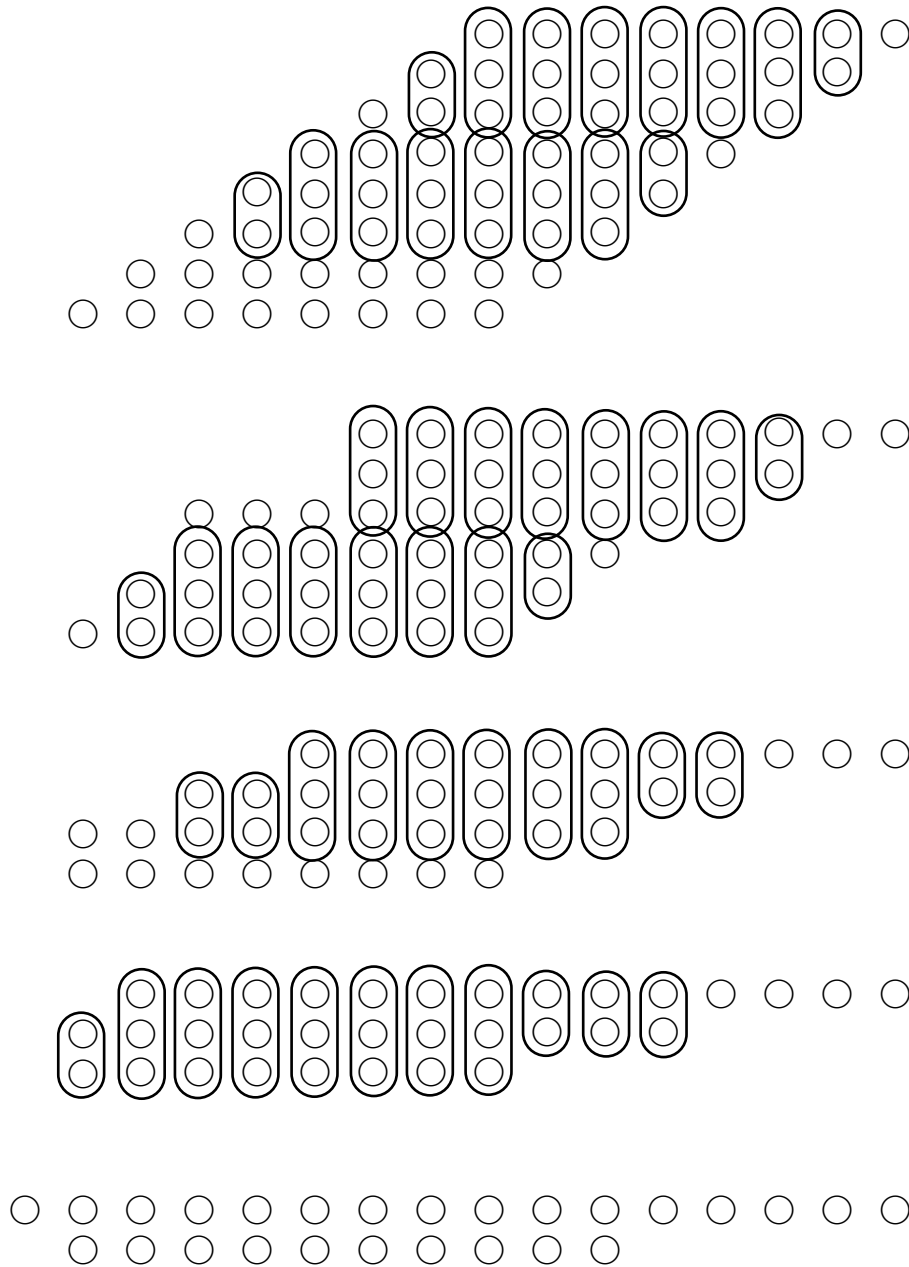


Figure 16 - 8bit Wallace Tree dot matrix

7.2 CIRCUIT DIAGRAM

Figure 17 shows the schematic for the Wallace tree multiplier that was done in Proteus. The circuit was tested before any VHDL was written to confirm that it was functioning correctly.

A full PDF of the schematic is available in the appendices.

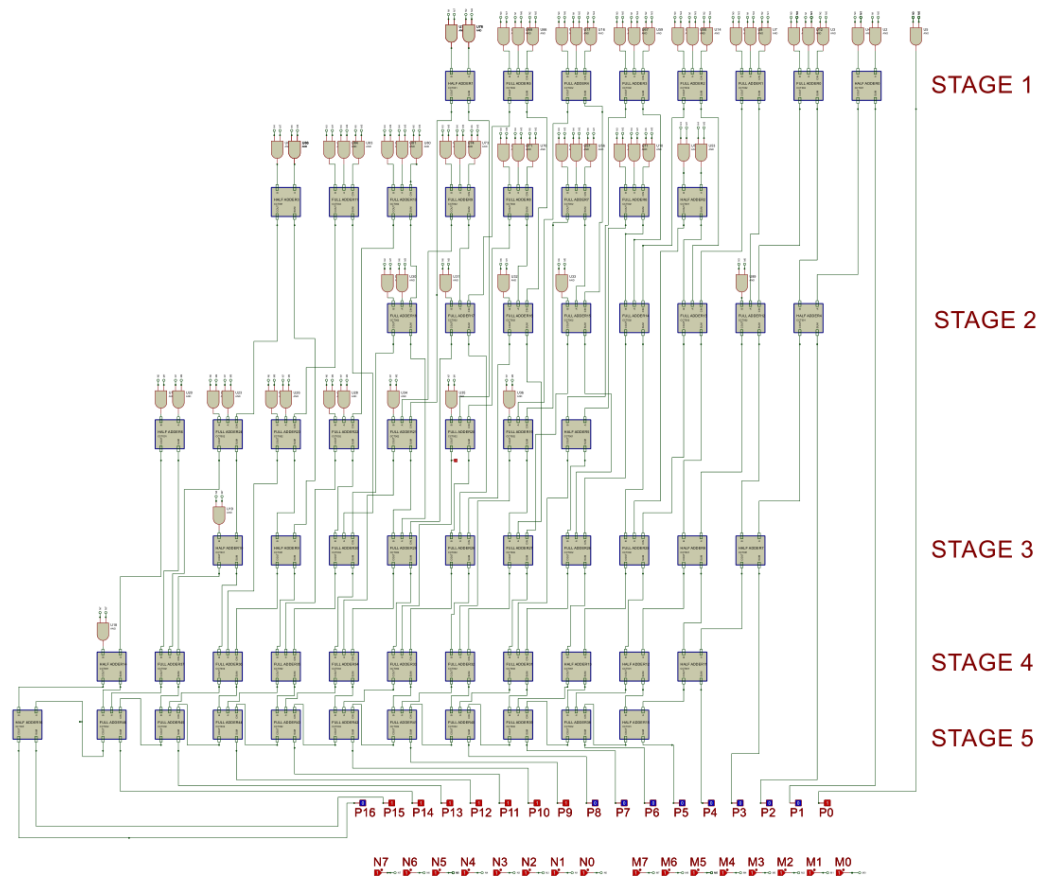


Figure 17 - Wallace tree multiplier schematic in Proteus

7.3 VHDL CODE

The full VHDL code and project link can be found in the appendices.

The Wallace tree multiplier VHDL code was also written using the structural modelling. Full and half adder components were created in VHDL and referenced in the multiplier VHDL document. The wiring was configured using port mapping.

7.4 ELABORATED DESIGN SCHEMATIC

The circuit described by the VHDL code can be seen in Figure 18. It shows the elaborated schematic generated using Vivado.

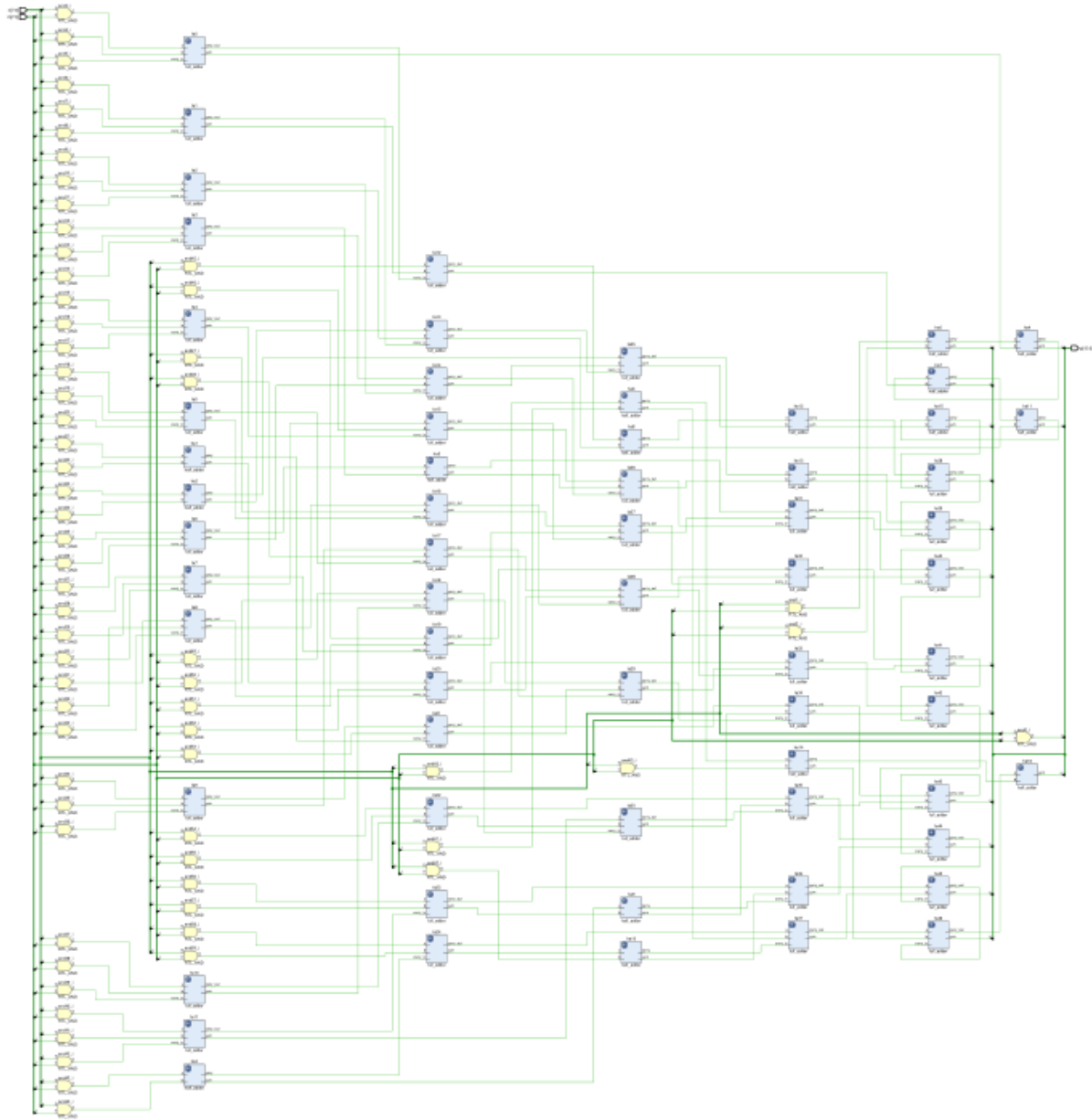


Figure 18 – Wallace Tree Circuit schematic in Vivado

7.5 SIMULATION RESULTS

The same method to validate the results of the Braun multiplier was used on the Wallace Tree multiplier. 0 errors were detected in the calculations.

The simulation for the Wallace tree multiplier also took 1.31ms to multiply each combination of input.



Figure 19 - Wallace tree multiplier simulation sample

7.6 SYNTHESIZED DESIGN SCHEMATIC

The full synthesized schematic can be seen in Figure 20.

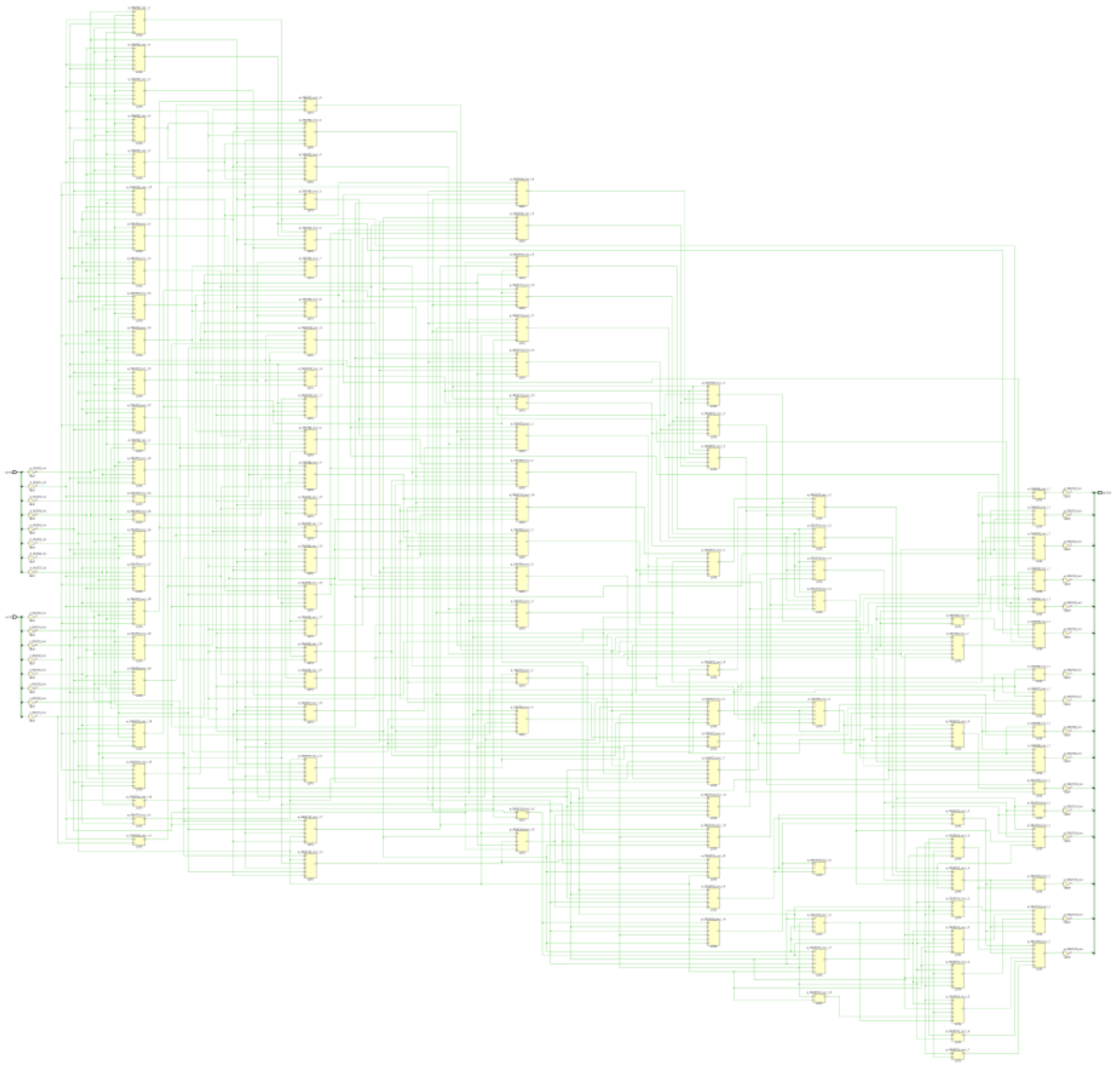


Figure 20 - Wallace Tree multiplier synthesized design schematic

7.7 FPGA LAYOUT

The FPGA layout of the Braun multiplier can be seen in Figure 21.

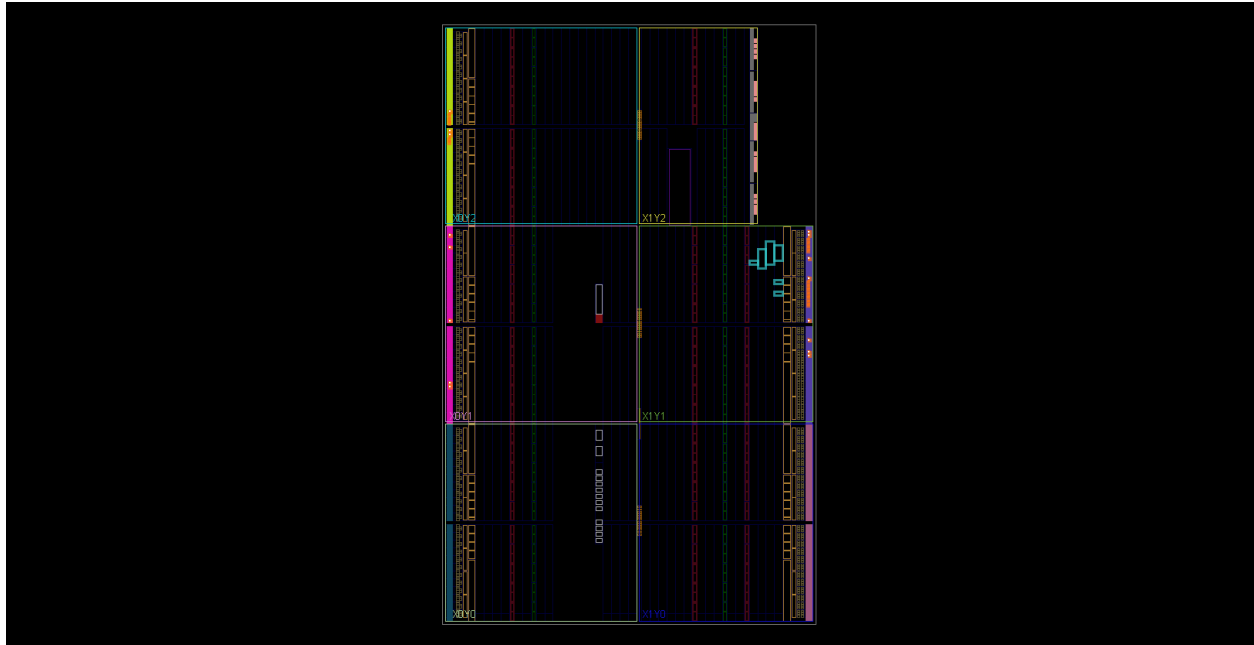


Figure 21 - Wallace Tree multiplier synthesized design

8 COMPARISON

Screenshots of reports generated after each design can be found in the appendices.

8.1 MAXIMUM AND MINIMUM DELAY

For both designs, a maximum and minimum delay was calculated by analyzing the schematics and counting the number of components in the longest path through each multiplier. The calculated delays will be used as timing constraints in the Vivado project for each design. Each component has a known delay specified in the assignment document. The component delays are shown in Table 7.

COMPONENT	DELAY
<i>AND GATE</i>	1ns
<i>HALF ADDER</i>	2ns
<i>FULL ADDER</i>	2ns

Table 7 – Component Delays

8.1.1 BRAUN MULTIPLIER LONGEST DELAY

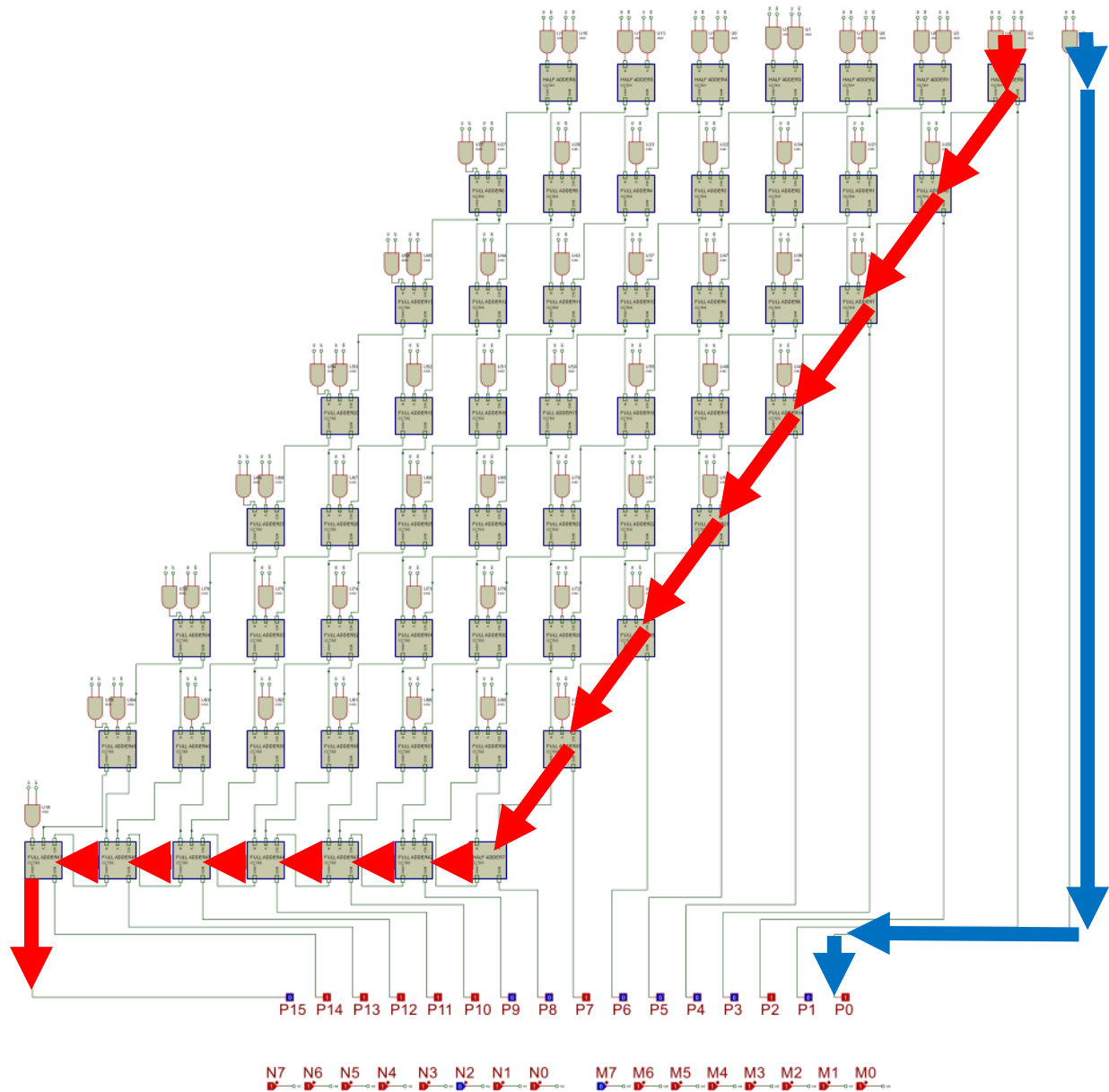


Figure 22 – 8x8 Braun multiplier maximum delay path (red) and minimum delay path (blue)

Longest delay contains 1 and gate and 14 adders.

$$\text{Max Delay} = 1(1ns) + 14(2ns)$$

$$\text{Max Delay} = 29ns$$

$$\text{Min Delay} = 1(1ns)$$

8.1.2 WALLACE TREE MULTIPLIER LONGEST DELAY

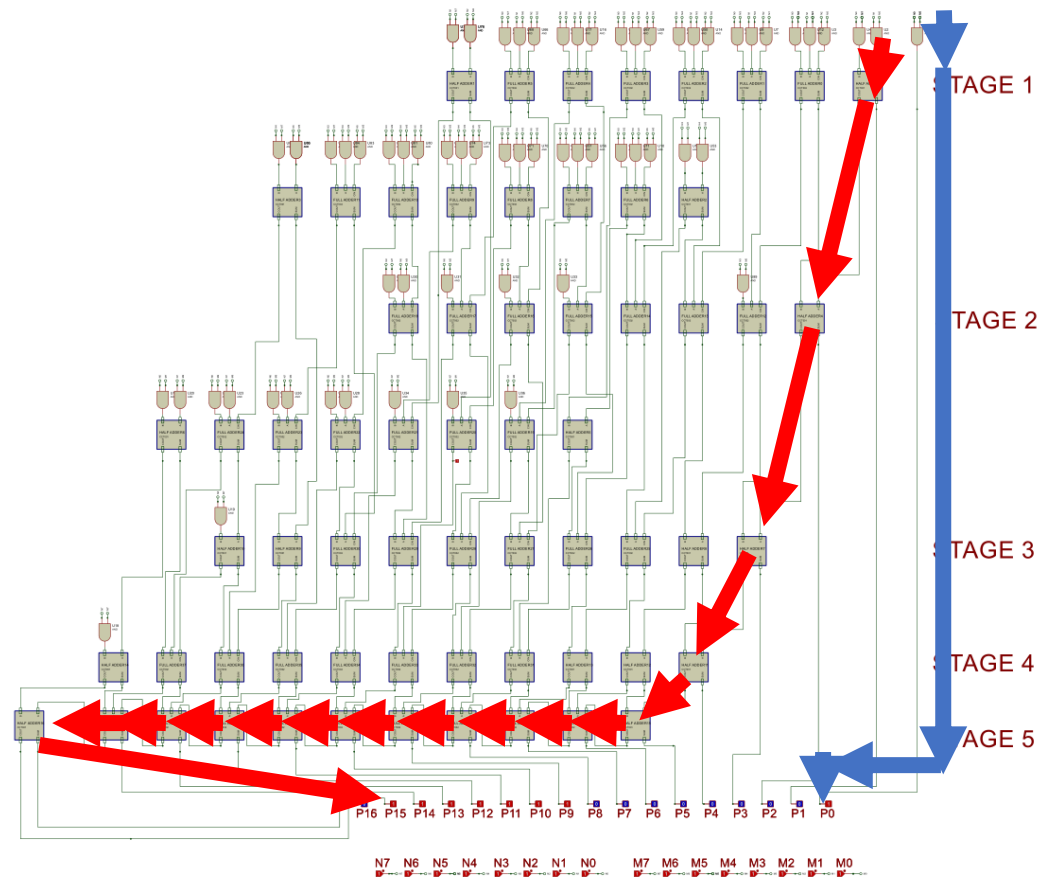


Figure 23 – 8x8 Wallace Tree multiplier maximum delay path (red) and minimum delay path (blue)

The longest delay contains 1 and gate and 15 adders.

$$\text{Max Delay} = 1(1ns) + 15(2ns)$$

$$\text{Max Delay} = 31ns$$

$$\text{Min Delay} = 1(1ns)$$

8.2 TIMING CONSTRAINTS

Table 8 shows the timing constraints that were set during the synthesis of the designs. They will constrain the design to work within the timings specified.

CONSTRAINT	BRAUN	WALLACE TREE
<i>CLOCK PERIOD</i>	10ns	10ns
<i>MAX DELAY</i>	29ns	31ns
<i>MIN DELAY</i>	1ns	1ns

Table 8 – Timing Constraints

8.3 BRAUN COMBINATIONAL DELAY

A full list of combinational delays can be found in the appendices. The delays were taken after timings constraints were added.

The max delay was 20.383ns between port m[5] and p[13]. The critical path of between these ports is shown in Figure 24.

The min delay was 2.91ns between port m[2] and p[2].

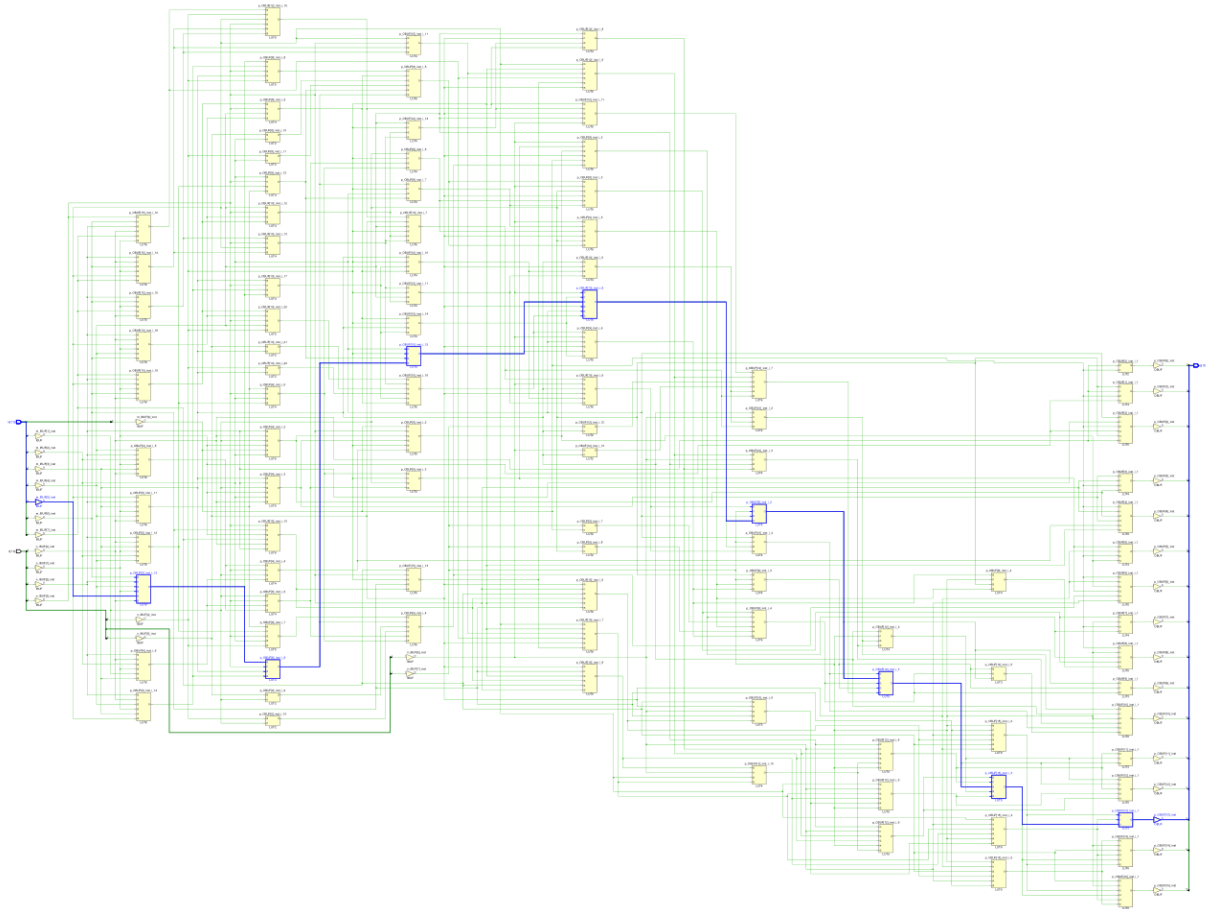


Figure 24 - Braun Multiplier – Critical Path

8.4 WALLACE TREE COMBINATIONAL DELAY

A full list of combinational delays can be found in the appendices. The delays were taken after timings constraints were added.

The max delay was 20.51ns between port m[5] and p[13]. The critical path of between these ports is shown in Figure 25.

The min delay was 3.00ns between port n[1] and p[2].

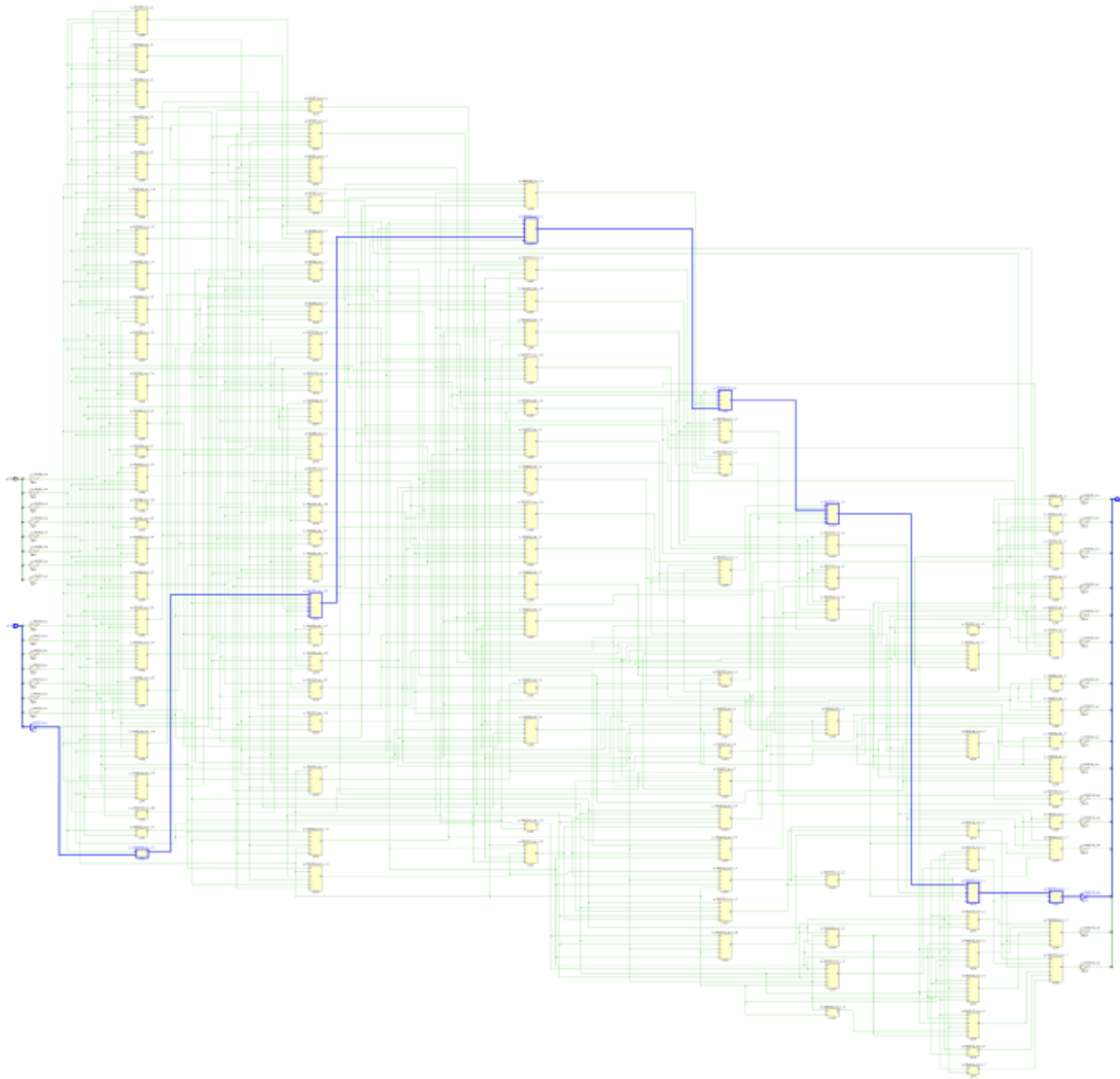


Figure 25- Wallace Tree Multiplier – Critical Path

8.5 TIMING COMPARISON

This section will compare the timing delays of both multiplier designs.

The power environment settings for each multiplier implementation were the same. This specified things such as ambient temperature and airflow. A screen shot of the power settings can be found in the appendices. The purpose of keeping these settings the same was so that the results are more comparable as there will be less variance between the implementations, other than the actual multiplier design architecture.

Figure 26 shows the maximum delays that were simulated during the synthesis of the designs. The Wallace Tree multiplier has the greater maximum when compared.

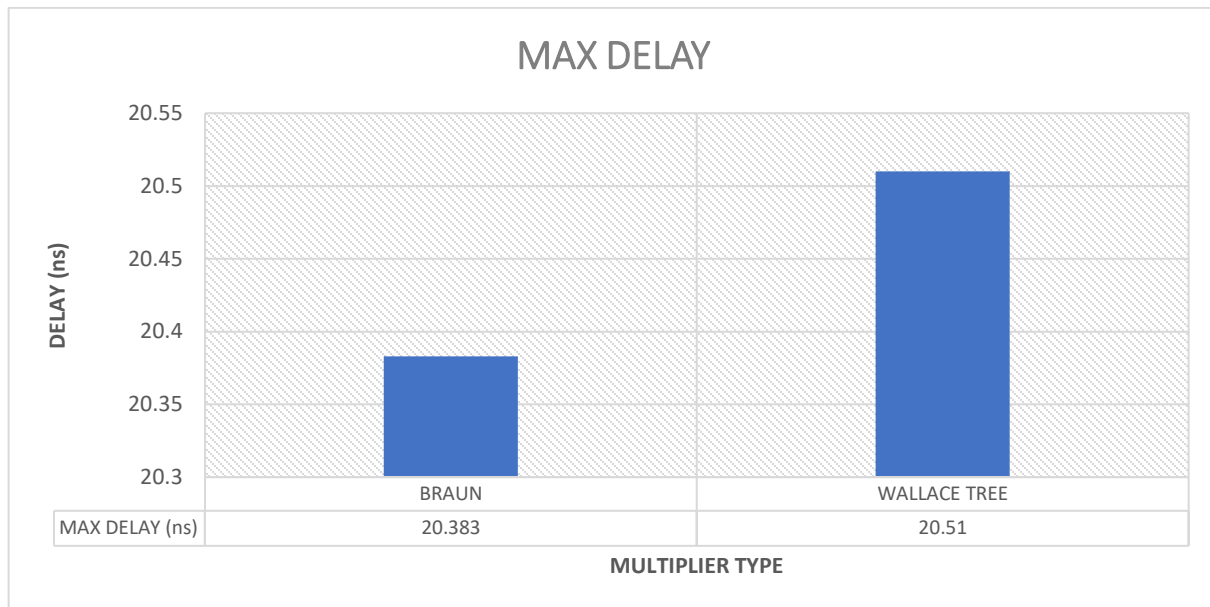


Figure 26 – Comparison of Max delay of Braun and Wallace Tree multiplier designs

Figure 27 shows the minimum delays for both multipliers. The Wallace tree design had the greatest minimum delay.

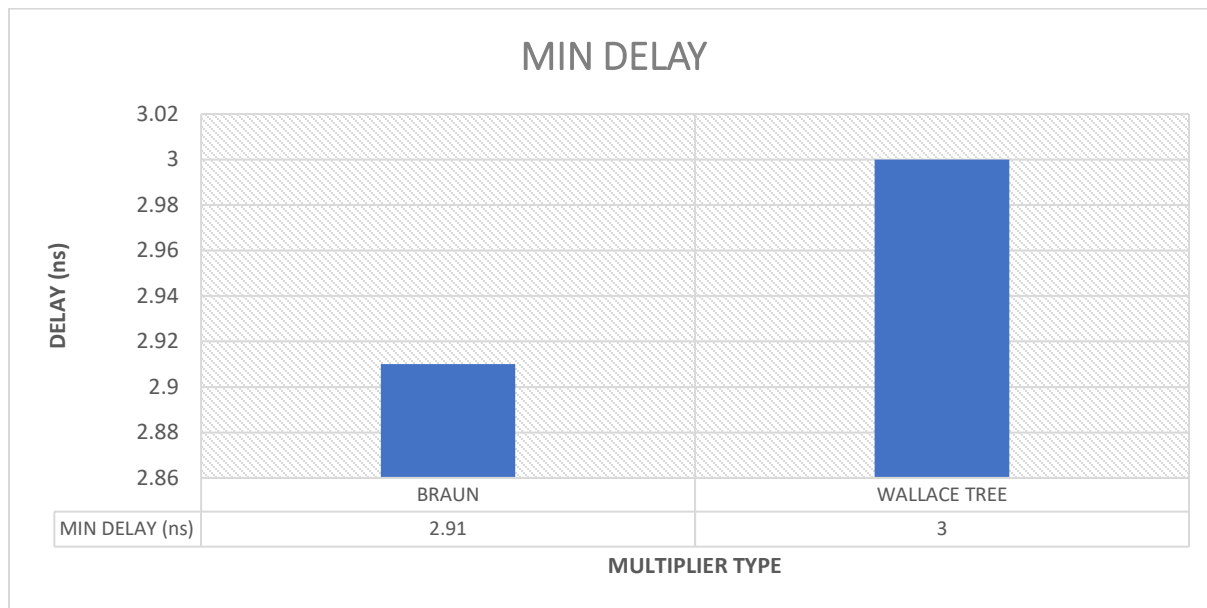


Figure 27– Comparison of Min delay of Braun and Wallace Tree multiplier designs

From the simulated delay timings, it is clear that the Wallace Tree was the slowest multiplier when it was synthesized.

8.6 POWER CONSUMPTION COMPARISON

Figure 28 shows the total power consumption of multiplier designs. The Wallace Tree multiplier design has a significantly higher power consumption than the Braun multiplier.

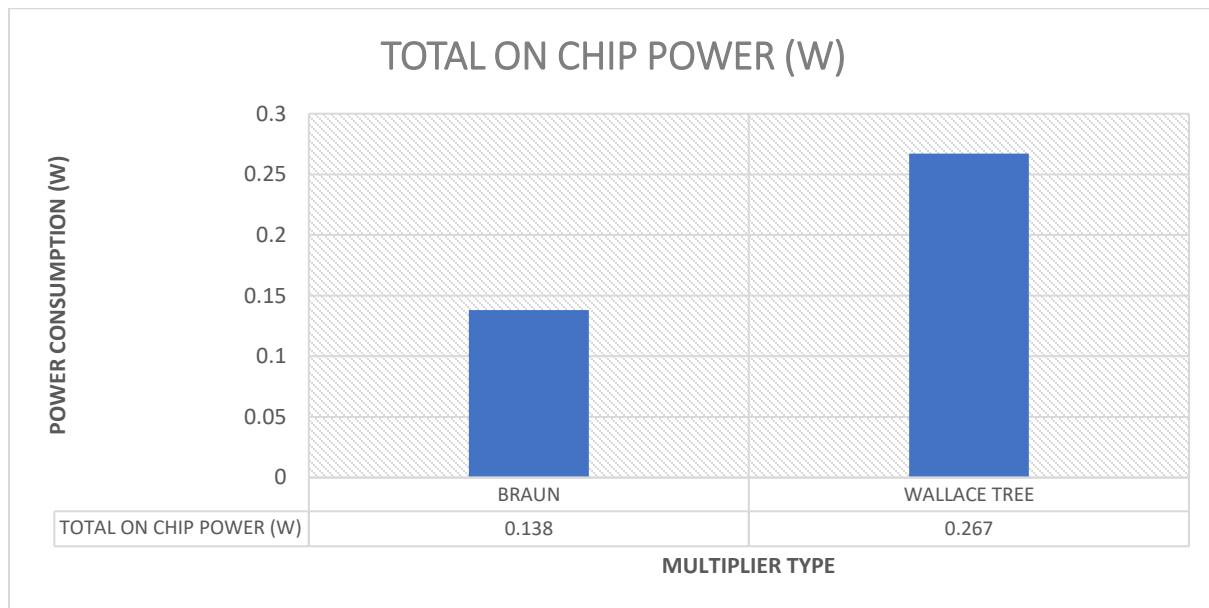


Figure 28 - Comparison of Power Consumption of Braun and Wallace Tree multiplier designs

A breakdown of power consumption of each multiplier design can be seen in Figure 29 and Figure 30. It is clear that the power usage of the Wallace tree multiplier is mostly used by the I/O within the FPGA.

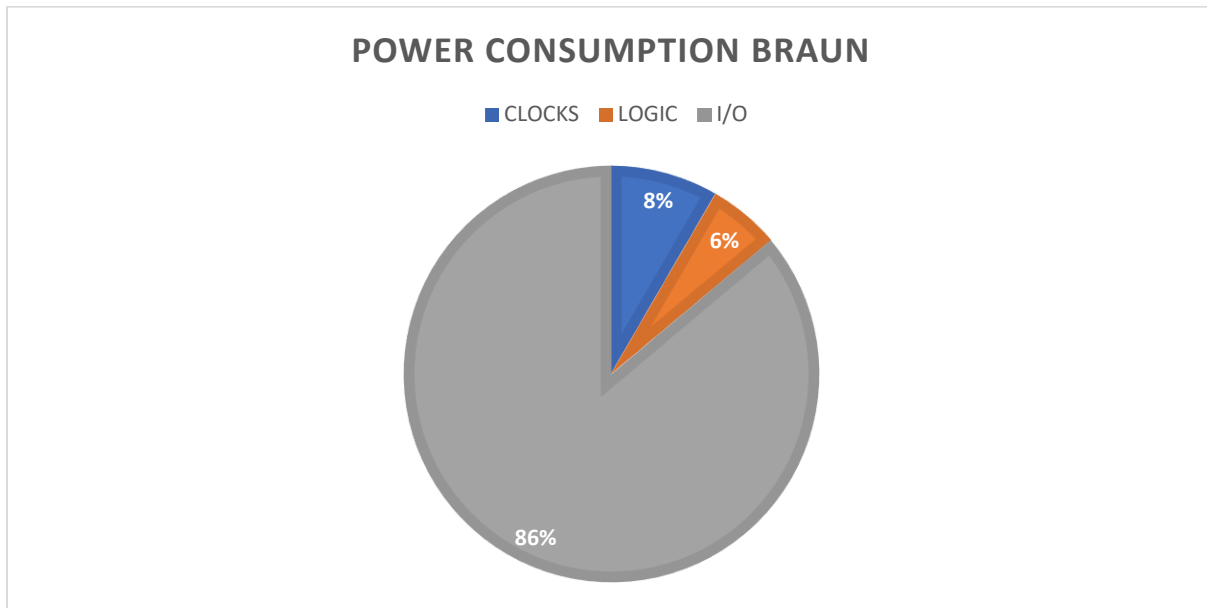


Figure 29 - Braun multiplier power consumption breakdown

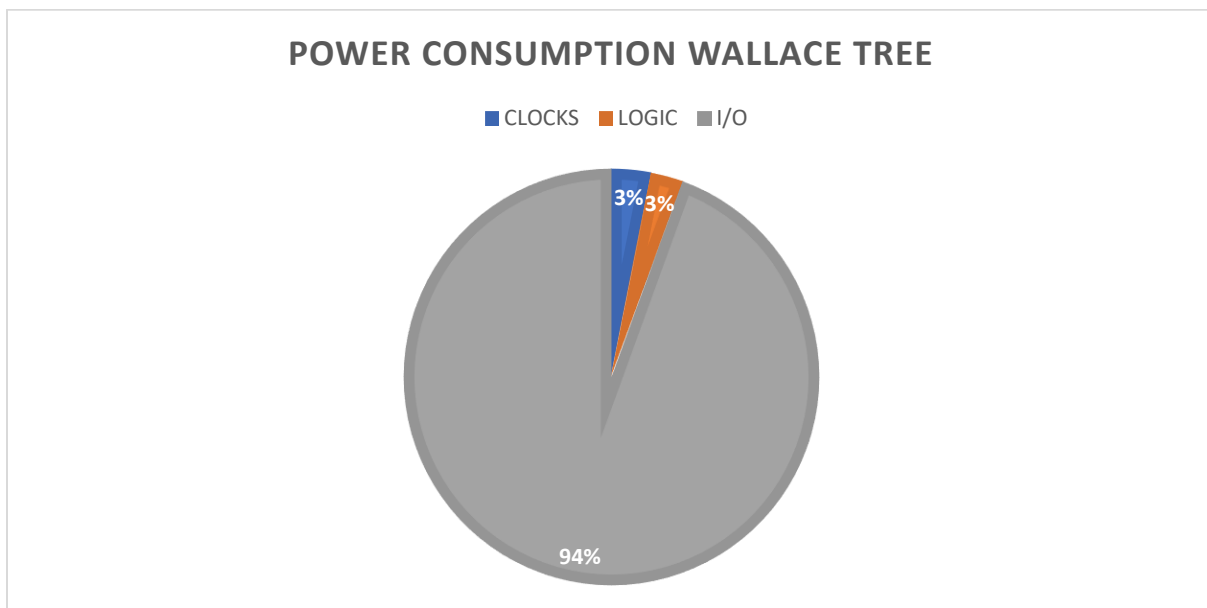


Figure 30 – Wallace Tree multiplier power consumption breakdown

Another comparison that can be done is to compare the number of look up tables (LUTs) used by each design. This is shown in Figure 31. The Wallace Tree has a higher number of LUTs used in its design. This is one thing that could account for the higher power consumption.

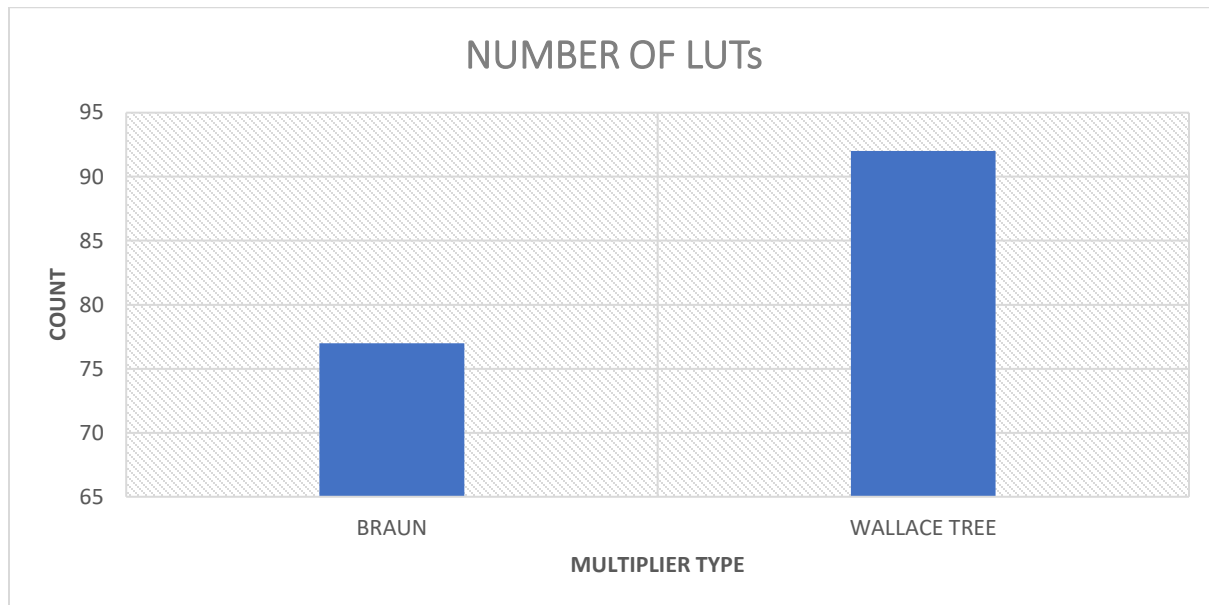


Figure 31 - Comparison of LUTs within Braun and Wallace Tree multiplier designs

9 CONCLUSION

It is clear that both multiplier designs were successful in their function. This was proved in the simulation section 6.4 and 7.5 where each combination of the multiplier was input and checked.

The data collected from the implementation reports after synthesis had some results that was not expected. For example, the Wallace Tree multiplier was not faster. Other studies such as “A survey paper on design and implementation of multipliers for digital system applications” stated that array multipliers such as the Wallace Tree variant are used to execute high speed designs while increasing the power consumption [11]. This was not reflected in the results of this report which show that while Wallace Tree multiplier did have a higher power consumption, it was the slower of the two.

It was noted that the Wallace tree multiplier did have a higher number of LUTs to realize the design so perhaps this could have been a factor that would slow the multiplier down.

Further study would be required to investigate the true cause of this unexpected result.

10 RECCOMENDATIONS

In this section, some possible improvements to the work will be discussed.

It would be possible to simplify the current VHDL code. If the half adders were replaced with full adders which did not have their third input connected, then it would reduce the overall components within the design.

If a multiplier were to be implemented in industry without the constraints of it having to be a Wallace Tree or a Braun, it could be implemented using the style shown in Figure 32. This would greatly reduce the amount of code required to realize the design and also vastly improve the readability. If an engineer were to review the VHDL code within this report, it may not be obvious how everything is connected without spending significant time looking at the code. Another benefit to having shorter code is that it is much easier to debug. This will reduce the time taken to get a design from concept to implementation. Timing constraints could still be used on the design to ensure that it operates within a given specification.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multiplier is
    generic (N: integer := 8);
    port (a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
          y: out STD_LOGIC_VECTOR(2*N-1 downto 0));
end;

architecture synth of multiplier is
begin
    y <= a * b;
end;
```

Figure 32 – Alternative multiplier VHDL implementation [8]

Another improvement that could be made is to program the implemented design on a real FPGA. This would be a full realization of the design and could potentially highlight issues with the design that would be found in a further debugging stage. The results would be a more robust design that is fully tested.

The FPGA device that was selected is perhaps not the most suitable for this application. While it does meet the minimum requirements to contain the designs specified in this report, it is far above the specification that is required. When analyzing the utilization of the FPGA in both designs it is clear that only a small percentage of the potential IOs and LUTs available in the device is being used. In both designs, only 1% of LUTs and 15% of IOs are being used. If a smaller FPGA was selected, then it may cost less and if it was used in a product, it would reduce the price of the product overall.

11 APPENDICES

11.1 FULL DOCUMENT DOWNLOAD LINKS

DOCUMENT NAME	NOTE	LINK
ELECTRONICS 2 ASSIGNMENT PROJECT FILES.7z	XILINX PROJECT FILES FOR BOTH MULTIPLIERS	link
BRAUN MULTIPLIER SCHEMATIC PROTEUS.PDF		link
BRAUN MULTIPLIER SCHEMATIC VIVADO ELABORATED.PDF		link
BRAUN MULTIPLIER SCHEMATIC VIVADO IMPLEMENTATION.PDF		link
BRAUN MULTIPLIER SCHEMATIC VIVADO IMPLEMENTATION_CRITICAL_PATH.PDF		link
WALLACE TREE MULTIPLIER SCHEMATIC PROTEUS.PDF		link
WALLACE TREE MULTIPLIER SCHEMATIC VIVADO ELABORATED.PDF		link
WALLACE TREE MULTIPLIER SCHEMATIC IMPLEMENTATION.PDF		link
WALLACE TREE MULTIPLIER SCHEMATIC IMPLEMENTATION_CRITICAL_PATH.PDF		link
braun and wallace tree multiplier checking.xlsx	Document used to verify all answers of multiplier	link

11.2 HALF ADDER COMPONENT CODE

```
-----  
-- University: UWTSD  
-- Engineer: Michael Howells  
-- Module Name: half_adder  
-- Description: half adder component  
-- Revision: 1  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity half_adder is  
port(  
    a: in std_logic;  
    b: in std_logic;  
    sum : out std_logic;  
    carry : out std_logic  
);  
end half_adder;
```

```
architecture Behavioral of half_adder is  
begin
```

```
    sum <= a xor b;  
    carry <= a and b;
```

```
end Behavioral;
```

11.3 FULL ADDER COMPONENT CODE

```
-----  
-- University: UWTSD  
-- Engineer: Michael Howells  
-- Module Name: full_adder  
-- Description: full adder component  
-- Revision: 1  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity full_adder is  
port(  
    a: in std_logic;  
    b: in std_logic;  
    carry_in: in std_logic;  
    sum : out std_logic;  
    carry_out : out std_logic  
);  
end full_adder;
```

```
architecture Behavioral of full_adder is  
begin  
    sum <= a xor b xor carry_in;  
    carry_out <= (a and b) or (a and carry_in) or (b and carry_in);  
  
end Behavioral;
```

11.4 BRAUN MULTIPLIER VHDL CODE

```
-----  
-- University: UWTSD  
-- Engineer: Michael Howells  
-- Module Name: multiplier_8_bit_wallace_tree  
-- Description: This is a 8 bit multiplier utilising the wallace tree design architecture. Implemented using  
the VHDL structural modeling.  
-- Revision: 1  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity multiplier_8bit_braun is  
port(  
    m: in std_logic_vector(7 downto 0);  
    n: in std_logic_vector(7 downto 0);  
    p: out std_logic_vector(15 downto 0)  
);  
end multiplier_8bit_braun;
```

```
architecture Behavioral of multiplier_8bit_braun is  
component half_adder  
port(  
    a: in std_logic;  
    b: in std_logic;  
    sum: out std_logic;  
    carry: out std_logic  
);  
end component;
```

```
component full_adder  
port(  
    a: in std_logic;  
    b: in std_logic;  
    carry_in: in std_logic;  
    sum : out std_logic;  
    carry_out : out std_logic  
);  
end component;
```

```
signal and0,and1,and2,and3,and4,and5,and6,and7,and8,and9: std_logic;  
signal and10,and11,and12,and13,and14,and15,and16,and17,and18,and19: std_logic;  
signal and20,and21,and22,and23,and24,and25,and26,and27,and28,and29: std_logic;  
signal and30,and31,and32,and33,and34,and35,and36,and37,and38,and39: std_logic;  
signal and40,and41,and42,and43,and44,and45,and46,and47,and48,and49: std_logic;  
signal and50,and51,and52,and53,and54,and55,and56,and57,and58,and59: std_logic;
```

```
signal and60,and61,and62,and63: std_logic;
```

```
--stage 1 signals
```

```
signal ha0_s,ha0_co: std_logic;  
signal ha1_s,ha1_co: std_logic;  
signal ha2_s,ha2_co: std_logic;  
signal ha3_s,ha3_co: std_logic;  
signal ha4_s,ha4_co: std_logic;  
signal ha5_s,ha5_co: std_logic;  
signal ha6_s,ha6_co: std_logic;
```

```
--stage 2 signals
```

```
signal fa0_s,fa0_co: std_logic;  
signal fa1_s,fa1_co: std_logic;  
signal fa2_s,fa2_co: std_logic;  
signal fa3_s,fa3_co: std_logic;  
signal fa4_s,fa4_co: std_logic;  
signal fa5_s,fa5_co: std_logic;  
signal fa6_s,fa6_co: std_logic;
```

```
--stage 3 signals
```

```
signal fa7_s,fa7_co: std_logic;  
signal fa8_s,fa8_co: std_logic;  
signal fa9_s,fa9_co: std_logic;  
signal fa10_s,fa10_co: std_logic;  
signal fa11_s,fa11_co: std_logic;  
signal fa12_s,fa12_co: std_logic;  
signal fa13_s,fa13_co: std_logic;
```

```
--stage 4 signals
```

```
signal fa14_s,fa14_co: std_logic;  
signal fa15_s,fa15_co: std_logic;  
signal fa16_s,fa16_co: std_logic;  
signal fa17_s,fa17_co: std_logic;  
signal fa18_s,fa18_co: std_logic;  
signal fa19_s,fa19_co: std_logic;  
signal fa20_s,fa20_co: std_logic;
```

```
--stage 5 signals
```

```
signal fa21_s,fa21_co: std_logic;  
signal fa22_s,fa22_co: std_logic;  
signal fa23_s,fa23_co: std_logic;  
signal fa24_s,fa24_co: std_logic;  
signal fa25_s,fa25_co: std_logic;  
signal fa26_s,fa26_co: std_logic;  
signal fa27_s,fa27_co: std_logic;
```

```
--stage 6 signals
```

```
signal fa28_s,fa28_co: std_logic;
signal fa29_s,fa29_co: std_logic;
signal fa30_s,fa30_co: std_logic;
signal fa31_s,fa31_co: std_logic;
signal fa32_s,fa32_co: std_logic;
signal fa33_s,fa33_co: std_logic;
signal fa34_s,fa34_co: std_logic;
```

--stage 7 signals

```
signal fa35_s,fa35_co: std_logic;
signal fa36_s,fa36_co: std_logic;
signal fa37_s,fa37_co: std_logic;
signal fa38_s,fa38_co: std_logic;
signal fa39_s,fa39_co: std_logic;
signal fa40_s,fa40_co: std_logic;
signal fa41_s,fa41_co: std_logic;
```

--stage 8 signals

```
signal ha7_s,ha7_co: std_logic;
signal fa42_s,fa42_co: std_logic;
signal fa43_s,fa43_co: std_logic;
signal fa44_s,fa44_co: std_logic;
signal fa45_s,fa45_co: std_logic;
signal fa46_s,fa46_co: std_logic;
signal fa47_s,fa47_co: std_logic;
```

begin

--component setup

--stage 1 adders

```
ha0 : half_adder port map(and1,and2,ha0_s,ha0_co);
ha1 : half_adder port map(and3,and4,ha1_s,ha1_co);
ha2 : half_adder port map(and5,and6,ha2_s,ha2_co);
ha3 : half_adder port map(and7,and8,ha3_s,ha3_co);
ha4 : half_adder port map(and9,and10,ha4_s,ha4_co);
ha5 : half_adder port map(and11,and12,ha5_s,ha5_co);
ha6 : half_adder port map(and13,and14,ha6_s,ha6_co);
```

--stage 2 adders

```
fa0 : full_adder port map(and15,ha1_s,ha0_co,fa0_s,fa0_co);
fa1 : full_adder port map(and16,ha2_s,ha1_co,fa1_s,fa1_co);
fa2 : full_adder port map(and17,ha3_s,ha2_co,fa2_s,fa2_co);
fa3 : full_adder port map(and18,ha4_s,ha3_co,fa3_s,fa3_co);
fa4 : full_adder port map(and19,ha5_s,ha4_co,fa4_s,fa4_co);
fa5 : full_adder port map(and20,ha6_s,ha5_co,fa5_s,fa5_co);
fa6 : full_adder port map(and21,and22,ha6_co,fa6_s,fa6_co);
```

--stage 3 adders

```
fa7 : full_adder port map(and23,fa1_s,fa0_co,fa7_s,fa7_co);
fa8 : full_adder port map(and24,fa2_s,fa1_co,fa8_s,fa8_co);
fa9 : full_adder port map(and25,fa3_s,fa2_co,fa9_s,fa9_co);
fa10 : full_adder port map(and26,fa4_s,fa3_co,fa10_s,fa10_co);
fa11 : full_adder port map(and27,fa5_s,fa4_co,fa11_s,fa11_co);
fa12 : full_adder port map(and28,fa6_s,fa5_co,fa12_s,fa12_co);
fa13 : full_adder port map(and29,and30,fa6_co,fa13_s,fa13_co);
```

--stage 4 adders

```
fa14 : full_adder port map(and31,fa8_s ,fa7_co ,fa14_s,fa14_co);
fa15 : full_adder port map(and32,fa9_s ,fa8_co ,fa15_s,fa15_co);
fa16 : full_adder port map(and33,fa10_s,fa9_co ,fa16_s,fa16_co);
fa17 : full_adder port map(and34,fa11_s,fa10_co,fa17_s,fa17_co);
fa18 : full_adder port map(and35,fa12_s,fa11_co,fa18_s,fa18_co);
fa19 : full_adder port map(and36,fa13_s,fa12_co,fa19_s,fa19_co);
fa20 : full_adder port map(and37,and38 ,fa13_co,fa20_s,fa20_co);
```

--stage 5 adders

```
fa21 : full_adder port map(and39,fa15_s,fa14_co,fa21_s,fa21_co);
fa22 : full_adder port map(and40,fa16_s,fa15_co,fa22_s,fa22_co);
fa23 : full_adder port map(and41,fa17_s,fa16_co,fa23_s,fa23_co);
fa24 : full_adder port map(and42,fa18_s,fa17_co,fa24_s,fa24_co);
fa25 : full_adder port map(and43,fa19_s,fa18_co,fa25_s,fa25_co);
fa26 : full_adder port map(and44,fa20_s,fa19_co,fa26_s,fa26_co);
fa27 : full_adder port map(and45,and46 ,fa20_co,fa27_s,fa27_co);
```

--stage 6 adders

```
fa28 : full_adder port map(and47,fa22_s,fa21_co,fa28_s,fa28_co);
fa29 : full_adder port map(and48,fa23_s,fa22_co,fa29_s,fa29_co);
fa30 : full_adder port map(and49,fa24_s,fa23_co,fa30_s,fa30_co);
fa31 : full_adder port map(and50,fa25_s,fa24_co,fa31_s,fa31_co);
fa32 : full_adder port map(and51,fa26_s,fa25_co,fa32_s,fa32_co);
fa33 : full_adder port map(and52,fa27_s,fa26_co,fa33_s,fa33_co);
fa34 : full_adder port map(and53,and54 ,fa27_co,fa34_s,fa34_co);
```

--stage 7 adders

```
fa35 : full_adder port map(and55,fa29_s,fa28_co,fa35_s,fa35_co);
fa36 : full_adder port map(and56,fa30_s,fa29_co,fa36_s,fa36_co);
fa37 : full_adder port map(and57,fa31_s,fa30_co,fa37_s,fa37_co);
fa38 : full_adder port map(and58,fa32_s,fa31_co,fa38_s,fa38_co);
fa39 : full_adder port map(and59,fa33_s,fa32_co,fa39_s,fa39_co);
fa40 : full_adder port map(and60,fa34_s,fa33_co,fa40_s,fa40_co);
fa41 : full_adder port map(and61,and62 ,fa34_co,fa41_s,fa41_co);
```

--stage 8 adders

```
ha7 : half_adder port map(fa35_co,fa36_s ,ha7_s,ha7_co);
fa42 : full_adder port map(fa36_co,fa37_s ,ha7_co ,fa42_s,fa42_co);
fa43 : full_adder port map(fa37_co,fa38_s ,fa42_co,fa43_s,fa43_co);
```

```
fa44 : full_adder port map(fa38_co,fa39_s,fa43_co,fa44_s,fa44_co);
fa45 : full_adder port map(fa39_co,fa40_s,fa44_co,fa45_s,fa45_co);
fa46 : full_adder port map(fa40_co,fa41_s,fa45_co,fa46_s,fa46_co);
fa47 : full_adder port map(and63 ,fa41_co,fa46_co,fa47_s,fa47_co);
```

--stage 1 and gates

```
and0 <= n(0) and m(0);
and1 <= n(1) and m(0);
and2 <= n(0) and m(1);
and3 <= n(1) and m(1);
and4 <= n(0) and m(2);
and5 <= n(1) and m(2);
and6 <= n(0) and m(3);
and7 <= n(1) and m(3);
and8 <= n(0) and m(4);
and9 <= n(1) and m(4);
and10 <= n(0) and m(5);
and11 <= n(1) and m(5);
and12 <= n(0) and m(6);
and13 <= n(1) and m(6);
and14 <= n(0) and m(7);
```

--stage 2 and gates

```
and15 <= n(2) and m(0);
and16 <= n(2) and m(1);
and17 <= n(2) and m(2);
and18 <= n(2) and m(3);
and19 <= n(2) and m(4);
and20 <= n(2) and m(5);
and21 <= n(2) and m(6);
and22 <= n(1) and m(7);
```

--stage 3 and gates

```
and23 <= n(3) and m(0);
and24 <= n(3) and m(1);
and25 <= n(3) and m(2);
and26 <= n(3) and m(3);
and27 <= n(3) and m(4);
and28 <= n(3) and m(5);
and29 <= n(3) and m(6);
and30 <= n(2) and m(7);
```

--stage 4 and gates

```
and31 <= n(4) and m(0);
and32 <= n(4) and m(1);
and33 <= n(4) and m(2);
and34 <= n(4) and m(3);
and35 <= n(4) and m(4);
```

```
and36 <= n(4) and m(5);
and37 <= n(4) and m(6);
and38 <= n(3) and m(7);
```

--stage 5 and gates

```
and39 <= n(5) and m(0);
and40 <= n(5) and m(1);
and41 <= n(5) and m(2);
and42 <= n(5) and m(3);
and43 <= n(5) and m(4);
and44 <= n(5) and m(5);
and45 <= n(5) and m(6);
and46 <= n(4) and m(7);
```

--stage 6 and gates

```
and47 <= n(6) and m(0);
and48 <= n(6) and m(1);
and49 <= n(6) and m(2);
and50 <= n(6) and m(3);
and51 <= n(6) and m(4);
and52 <= n(6) and m(5);
and53 <= n(6) and m(6);
and54 <= n(5) and m(7);
```

--stage 7 and gates

```
and55 <= n(7) and m(0);
and56 <= n(7) and m(1);
and57 <= n(7) and m(2);
and58 <= n(7) and m(3);
and59 <= n(7) and m(4);
and60 <= n(7) and m(5);
and61 <= n(7) and m(6);
and62 <= n(6) and m(7);
```

--stage 8 connections

```
and63 <= n(7) and m(7);
```

--product assignment

```
p(0) <= and0;
p(1) <= ha0_s;
p(2) <= fa0_s;
p(3) <= fa7_s;
p(4) <= fa14_s;
p(5) <= fa21_s;
p(6) <= fa28_s;
p(7) <= fa35_s;
p(8) <= ha7_s;
p(9) <= fa42_s;
```

```
p(10) <= fa43_s;  
p(11) <= fa44_s;  
p(12) <= fa45_s;  
p(13) <= fa46_s;  
p(14) <= fa47_s;  
p(15) <= fa47_co;
```

```
end Behavioral;
```

11.5 BRAUN MULTIPLIER TEST BENCH VHDL CODE

```
-----  
-- University: UWTSD  
-- Engineer: Michael Howells  
-- Module Name: multiplier_8_bit_braun_tb  
-- Description: This test bench program will iterate over all input combinations of the 8bit braun  
multiplier. The output is also printed to the terminal.  
-- Revision: 1  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.NUMERIC_STD.ALL;
```

```
entity multiplier_8bit_braun_tb is  
end multiplier_8bit_braun_tb;
```

```
architecture Behavioral of multiplier_8bit_braun_tb is  
component multiplier_8bit_braun is  
port(  
    m: in std_logic_vector(7 downto 0);  
    n: in std_logic_vector(7 downto 0);  
    p: out std_logic_vector(15 downto 0)  
);  
end component;
```

```
--inputs  
signal m: std_logic_vector(7 downto 0);  
signal n: std_logic_vector(7 downto 0);
```

```
--outputs  
signal p: std_logic_vector(15 downto 0);
```

```
begin  
--device under test  
UUT: multiplier_8bit_braun port map (  
    m => m,  
    n => n,  
    p => p  
);
```

```
stimulus:process  
constant period: time:= 20 ns;
```

```
begin  
--for loop used to simulate all possible inputs for the multipliers  
--1111 1111 bin = 255
```

```

for int_m in 0 to 255 loop
  for int_n in 0 to 255 loop
    --value of a and b in loop are integers. The code below will convert their types to unsigned
    integers
    --and then to std_logic_vector types which can be used to drive the circuit.
    m <= std_logic_vector(to_unsigned(int_m, m'length));
    n <= std_logic_vector(to_unsigned(int_n, n'length));

    wait for period;
    --values of a and b are printed to terminal, used for debugging.
    report "m=" & integer'image(to_integer(unsigned(m))) & " n=" &
integer'image(to_integer(unsigned(n)))& " p=" & integer'image(to_integer(unsigned(p)));

    end loop;
  end loop;

wait for 2000ns;
wait;

end process;
end Behavioral;

```

11.6 WALLACE MULTIPLIER VHDL CODE

```
-----  
-- University: UWTSD  
-- Engineer: Michael Howells  
-- Module Name: multiplier_8_bit_wallace_tree  
-- Description: This is a 8 bit multiplier utilising the wallace tree design architecture. Implemented using  
the VHDL structural modeling.  
-- Revision: 1  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity multiplier_8bit_wallace_tree is  
port(  
    m: in std_logic_vector(7 downto 0);  
    n: in std_logic_vector(7 downto 0);  
    p: out std_logic_vector(15 downto 0)  
);  
end multiplier_8bit_wallace_tree;
```

```
architecture Behavioral of multiplier_8bit_wallace_tree is  
component half_adder  
port(  
    a: in std_logic;  
    b: in std_logic;  
    sum: out std_logic;  
    carry: out std_logic  
);  
end component;
```

```
component full_adder  
port(  
    a: in std_logic;  
    b: in std_logic;  
    carry_in: in std_logic;  
    sum : out std_logic;  
    carry_out : out std_logic  
);  
end component;
```

```
signal and0,and1,and2,and3,and4,and5,and6,and7,and8,and9: std_logic;  
signal and10,and11,and12,and13,and14,and15,and16,and17,and18,and19: std_logic;  
signal and20,and21,and22,and23,and24,and25,and26,and27,and28,and29: std_logic;  
signal and30,and31,and32,and33,and34,and35,and36,and37,and38,and39: std_logic;  
signal and40,and41,and42,and43,and44,and45,and46,and47,and48,and49: std_logic;  
signal and50,and51,and52,and53,and54,and55,and56,and57,and58,and59: std_logic;
```

```
signal and60,and61,and62,and63: std_logic;
```

```
--stage 1 signals
```

```
signal ha0_s,ha0_co: std_logic;
```

```
signal fa0_s,fa0_co: std_logic;
```

```
signal fa1_s,fa1_co: std_logic;
```

```
signal fa2_s,fa2_co: std_logic;
```

```
signal fa3_s,fa3_co: std_logic;
```

```
signal fa4_s,fa4_co: std_logic;
```

```
signal fa5_s,fa5_co: std_logic;
```

```
signal ha1_s,ha1_co: std_logic;
```

```
signal ha2_s,ha2_co: std_logic;
```

```
signal fa6_s,fa6_co: std_logic;
```

```
signal fa7_s,fa7_co: std_logic;
```

```
signal fa8_s,fa8_co: std_logic;
```

```
signal fa9_s,fa9_co: std_logic;
```

```
signal fa10_s,fa10_co: std_logic;
```

```
signal fa11_s,fa11_co: std_logic;
```

```
signal ha3_s,ha3_co: std_logic;
```

```
--stage 2 signals
```

```
signal ha4_s,ha4_co: std_logic;
```

```
signal fa12_s,fa12_co: std_logic;
```

```
signal fa13_s,fa13_co: std_logic;
```

```
signal fa14_s,fa14_co: std_logic;
```

```
signal fa15_s,fa15_co: std_logic;
```

```
signal fa16_s,fa16_co: std_logic;
```

```
signal fa17_s,fa17_co: std_logic;
```

```
signal fa18_s,fa18_co: std_logic;
```

```
signal ha5_s,ha5_co: std_logic;
```

```
signal fa19_s,fa19_co: std_logic;
```

```
signal fa20_s,fa20_co: std_logic;
```

```
signal fa21_s,fa21_co: std_logic;
```

```
signal fa22_s,fa22_co: std_logic;
```

```
signal fa23_s,fa23_co: std_logic;
```

```
signal fa24_s,fa24_co: std_logic;
```

```
signal ha6_s,ha6_co: std_logic;
```

```
--stage 3 signals
```

```
signal ha7_s,ha7_co: std_logic;  
signal ha8_s,ha8_co: std_logic;
```

```
signal fa25_s,fa25_co: std_logic;  
signal fa26_s,fa26_co: std_logic;  
signal fa27_s,fa27_co: std_logic;  
signal fa28_s,fa28_co: std_logic;  
signal fa29_s,fa29_co: std_logic;  
signal fa30_s,fa30_co: std_logic;
```

```
signal ha9_s,ha9_co: std_logic;  
signal ha10_s,ha10_co: std_logic;
```

```
--stage 4 signals
```

```
signal ha11_s,ha11_co: std_logic;  
signal ha12_s,ha12_co: std_logic;  
signal ha13_s,ha13_co: std_logic;
```

```
signal fa31_s,fa31_co: std_logic;  
signal fa32_s,fa32_co: std_logic;  
signal fa33_s,fa33_co: std_logic;  
signal fa34_s,fa34_co: std_logic;  
signal fa35_s,fa35_co: std_logic;  
signal fa36_s,fa36_co: std_logic;  
signal fa37_s,fa37_co: std_logic;
```

```
signal ha14_s,ha14_co: std_logic;
```

```
--stage 5 signals
```

```
signal ha15_s,ha15_co: std_logic;
```

```
signal fa38_s,fa38_co: std_logic;  
signal fa39_s,fa39_co: std_logic;  
signal fa40_s,fa40_co: std_logic;  
signal fa41_s,fa41_co: std_logic;  
signal fa42_s,fa42_co: std_logic;  
signal fa43_s,fa43_co: std_logic;  
signal fa44_s,fa44_co: std_logic;  
signal fa45_s,fa45_co: std_logic;  
signal fa46_s,fa46_co: std_logic;
```

```
signal ha16_s,ha16_co: std_logic;
```

```
begin
```

```
--component setup
```

```
--stage 1 adders
```

```
ha0 : half_adder port map(and1,and2,ha0_s,ha0_co);
```

```
fa0 : full_adder port map(and3,and4,and5,fa0_s,fa0_co);
fa1 : full_adder port map(and6,and7,and8,fa1_s,fa1_co);
fa2 : full_adder port map(and9,and10,and11,fa2_s,fa2_co);
fa3 : full_adder port map(and12,and13,and14,fa3_s,fa3_co);
fa4 : full_adder port map(and15,and16,and17,fa4_s,fa4_co);
fa5 : full_adder port map(and18,and19,and20,fa5_s,fa5_co);
```

```
ha1 : half_adder port map(and21,and22,ha1_s,ha1_co);
ha2 : half_adder port map(and23,and24,ha2_s,ha2_co);
```

```
fa6 : full_adder port map(and25,and26,and27,fa6_s,fa6_co);
fa7 : full_adder port map(and28,and29,and30,fa7_s,fa7_co);
fa8 : full_adder port map(and31,and32,and33,fa8_s,fa8_co);
fa9 : full_adder port map(and34,and35,and36,fa9_s,fa9_co);
fa10 : full_adder port map(and37,and38,and39,fa10_s,fa10_co);
fa11 : full_adder port map(and40,and41,and42,fa11_s,fa11_co);
```

```
ha3 : half_adder port map(and43,and44,ha3_s,ha3_co);
```

----stage 2 adders

```
ha4 : half_adder port map(ha0_co,fa0_s,ha4_s,ha4_co);
```

```
fa12 : full_adder port map(and45,fa1_s,fa0_co,fa12_s,fa12_co);
fa13 : full_adder port map(ha2_s,fa2_s,fa1_co,fa13_s,fa13_co);
fa14 : full_adder port map(fa3_s,fa6_s,fa2_co,fa14_s,fa14_co);
fa15 : full_adder port map(fa7_s,and46,fa4_s,fa15_s,fa15_co);
fa16 : full_adder port map(fa8_s,and47,fa5_s,fa16_s,fa16_co);
fa17 : full_adder port map(fa9_s,and48,fa5_co,fa17_s,fa17_co);
fa18 : full_adder port map(and49,and50,fa10_s,fa18_s,fa18_co);
```

```
ha5 : half_adder port map(fa6_co,fa3_co,ha5_s,ha5_co);
```

```
fa19 : full_adder port map(fa4_co,and51,fa7_co,fa19_s,fa19_co);
fa20 : full_adder port map(ha1_s,and52,fa8_co,fa20_s,fa20_co);
fa21 : full_adder port map(fa9_co,and53,ha1_co,fa21_s,fa21_co);
fa22 : full_adder port map(and54,and55,fa10_co,fa22_s,fa22_co);
fa23 : full_adder port map(and56,and57,fa11_co,fa23_s,fa23_co);
fa24 : full_adder port map(and58,and59,ha3_co,fa24_s,fa24_co);
```

```
ha6 : half_adder port map(and60,and61,ha6_s,ha6_co);
```

----stage 3 adders

```
ha7 : half_adder port map(ha4_co,fa12_s,ha7_s,ha7_co);
ha8 : half_adder port map(fa12_co,fa13_s,ha8_s,ha8_co);
```

```
fa25 : full_adder port map(ha2_co,fa14_s,fa13_co,fa25_s,fa25_co);
fa26 : full_adder port map(fa15_s,ha5_s,fa14_co,fa26_s,fa26_co);
```

```
fa27 : full_adder port map(fa16_s,fa19_s,fa15_co,fa27_s,fa27_co);
fa28 : full_adder port map(fa17_s,fa20_s,fa16_co,fa28_s,fa28_co);
fa29 : full_adder port map(fa18_s,fa21_s,fa17_co,fa29_s,fa29_co);
fa30 : full_adder port map(fa11_s,fa22_s,fa18_co,fa30_s,fa30_co);
```

```
ha9 : half_adder port map(ha3_s,fa23_s,ha9_s,ha9_co);
ha10 : half_adder port map(fa24_s,and62,ha10_s,ha10_co);
```

----stage 4 adders

```
ha11 : half_adder port map(ha7_co,ha8_s,ha11_s,ha11_co);
ha12 : half_adder port map(ha8_co,fa25_s,ha12_s,ha12_co);
ha13 : half_adder port map(fa25_co,fa26_s,ha13_s,ha13_co);
```

```
fa31 : full_adder port map(ha5_co,fa27_s,fa26_co,fa31_s,fa31_co);
fa32 : full_adder port map(fa19_co,fa28_s,fa27_co,fa32_s,fa32_co);
fa33 : full_adder port map(fa20_co,fa29_s,fa28_co,fa33_s,fa33_co);
fa34 : full_adder port map(fa21_co,fa30_s,fa29_co,fa34_s,fa34_co);
fa35 : full_adder port map(fa22_co,ha9_s,fa30_co,fa35_s,fa35_co);
fa36 : full_adder port map(fa23_co,ha10_s,ha9_co,fa36_s,fa36_co);
fa37 : full_adder port map(fa24_co,ha6_s,ha10_co,fa37_s,fa37_co);
```

```
ha14 : half_adder port map(ha6_co,and63,ha14_s,ha14_co);
```

----stage 5 adders

```
ha15 : half_adder port map(ha11_co,ha12_s,ha15_s,ha15_co);
```

```
fa38 : full_adder port map(ha12_co,ha13_s,ha15_co,fa38_s,fa38_co);
fa39 : full_adder port map(ha13_co,fa31_s,fa38_co,fa39_s,fa39_co);
fa40 : full_adder port map(fa31_co,fa32_s,fa39_co,fa40_s,fa40_co);
fa41 : full_adder port map(fa32_co,fa33_s,fa40_co,fa41_s,fa41_co);
fa42 : full_adder port map(fa33_co,fa34_s,fa41_co,fa42_s,fa42_co);
fa43 : full_adder port map(fa34_co,fa35_s,fa42_co,fa43_s,fa43_co);
fa44 : full_adder port map(fa35_co,fa36_s,fa43_co,fa44_s,fa44_co);
fa45 : full_adder port map(fa36_co,fa37_s,fa44_co,fa45_s,fa45_co);
fa46 : full_adder port map(fa37_co,ha14_s,fa45_co,fa46_s,fa46_co);
```

```
ha16 : half_adder port map(fa46_co,ha14_co,ha16_s,ha16_co);
```

--stage 1 and gates

```
and0 <= n(0) and m(0);
```

```
and1 <= n(1) and m(0);
```

```
and2 <= n(0) and m(1);
```

```
and3 <= n(2) and m(0);
```

```
and4 <= n(1) and m(1);
```

```
and5 <= n(0) and m(2);
```

and6 <= n(2) and m(1);
and7 <= n(1) and m(2);
and8 <= n(0) and m(3);

and9 <= n(2) and m(2);
and10 <= n(1) and m(3);
and11 <= n(0) and m(4);

and12 <= n(2) and m(3);
and13 <= n(1) and m(4);
and14 <= n(0) and m(5);

and15 <= n(2) and m(4);
and16 <= n(1) and m(5);
and17 <= n(0) and m(6);

and18 <= n(2) and m(5);
and19 <= n(1) and m(6);
and20 <= n(0) and m(7);

and21 <= n(2) and m(6);
and22 <= n(1) and m(7);

and23 <= n(4) and m(0);
and24 <= n(3) and m(1);

and25 <= n(5) and m(0);
and26 <= n(4) and m(1);
and27 <= n(3) and m(2);

and28 <= n(5) and m(1);
and29 <= n(4) and m(2);
and30 <= n(3) and m(3);

and31 <= n(5) and m(2);
and32 <= n(4) and m(3);
and33 <= n(3) and m(4);

and34 <= n(5) and m(3);
and35 <= n(4) and m(4);
and36 <= n(3) and m(5);

and37 <= n(5) and m(4);
and38 <= n(4) and m(5);
and39 <= n(3) and m(6);

and40 <= n(5) and m(5);

```
and41 <= n(4) and m(6);
and42 <= n(3) and m(7);
```

```
and43 <= n(5) and m(6);
and44 <= n(4) and m(7);
```

```
--stage 2 and gates
```

```
and45 <= n(3) and m(0);
```

```
and46 <= n(6) and m(0);
```

```
and47 <= n(6) and m(1);
```

```
and48 <= n(6) and m(2);
```

```
and49 <= n(6) and m(3);
```

```
and50 <= n(2) and m(7);
```

```
and51 <= n(7) and m(0);
```

```
and52 <= n(7) and m(1);
```

```
and53 <= n(7) and m(2);
```

```
and54 <= n(7) and m(3);
```

```
and55 <= n(6) and m(4);
```

```
and56 <= n(7) and m(4);
```

```
and57 <= n(6) and m(5);
```

```
and58 <= n(7) and m(5);
```

```
and59 <= n(6) and m(6);
```

```
and60 <= n(7) and m(6);
```

```
and61 <= n(6) and m(7);
```

```
--stage 3 and gates
```

```
and62 <= n(5) and m(7);
```

```
--stage 4 and gates
```

```
and63 <= n(7) and m(7);
```

```
--product assignment
```

```
p(0) <= and0;
```

```
p(1) <= ha0_s;
```

```
p(2) <= ha4_s;
```

```
p(3) <= ha7_s;
```

```
p(4) <= ha11_s;
```

```
p(5) <= ha15_s;  
p(6) <= fa38_s;  
p(7) <= fa39_s;  
p(8) <= fa40_s;  
p(9) <= fa41_s;  
p(10) <= fa42_s;  
p(11) <= fa43_s;  
p(12) <= fa44_s;  
p(13) <= fa45_s;  
p(14) <= fa46_s;  
p(15) <= ha16_s;
```

```
end Behavioral;
```

11.7 WALLACE MULTIPLIER TEST BENCH VHDL CODE

```
-----  
-- University: UWTSD  
-- Engineer: Michael Howells  
-- Module Name: multiplier_8_bit_wallace_tree_tb  
-- Description: This test bench program will iterate over all input combinations of the 8bit wallace tree  
multiplier. The output is also printed to the terminal.  
-- Revision: 1  
-----
```

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.NUMERIC_STD.ALL;
```

```
entity multiplier_8bit_wallace_tree_tb is  
end multiplier_8bit_wallace_tree_tb;
```

```
architecture Behavioral of multiplier_8bit_wallace_tree_tb is  
component multiplier_8bit_wallace_tree is  
port(  
    m: in std_logic_vector(7 downto 0);  
    n: in std_logic_vector(7 downto 0);  
    p: out std_logic_vector(15 downto 0)  
);  
end component;
```

```
--inputs  
signal m: std_logic_vector(7 downto 0);  
signal n: std_logic_vector(7 downto 0);
```

```
--outputs  
signal p: std_logic_vector(15 downto 0);
```

```
begin  
--device under test  
UUT: multiplier_8bit_wallace_tree port map (  
    m => m,  
    n => n,  
    p => p  
);
```

```
stimulus:process  
constant period: time:= 20 ns;
```

```
begin  
--for loop used to simulate all possible inputs for the multipliers  
--1111 1111 bin = 255
```

```

for int_m in 0 to 255 loop
  for int_n in 0 to 255 loop
    --value of a and b in loop are integers. The code below will convert their types to unsigned
    integers
    --and then to std_logic_vector types which can be used to drive the circuit.
    m <= std_logic_vector(to_unsigned(int_m, m'length));
    n <= std_logic_vector(to_unsigned(int_n, n'length));

    wait for period;
    --values of a and b are printed to terminal, used for debugging.
    report "m=" & integer'image(to_integer(unsigned(m))) & " n=" &
integer'image(to_integer(unsigned(n)))& " p=" & integer'image(to_integer(unsigned(p)));

    end loop;
  end loop;

wait for 2000ns;
wait;

end process;
end Behavioral;

```

11.8 IMPLEMENTATION POWER SETTINGS

Device		Environment	
Part:	xc7a15tlcsg324-2L	Output Load:	0 pF
Temp grade:	extended	Ambient temperature:	25.0 °C
Process:	typical	Airflow:	250 LFM
Characterization:	Production	Heat sink:	medium (Medium Profile)
		θ _{SA} :	4.6 °C/W
		Board selection:	medium (10"x10")
		Number of board layers:	12to15 (12 to 15 Layers)
		θ _{JB} :	6.8 °C/W
		Board temperature:	25.0 °C

Figure 33 - Implementation Power Settings

11.9 BRAUN MULTIPLIER POWER SUMMARY

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	0.138 W
Design Power Budget:	Not Specified
Power Budget Margin:	N/A
Junction Temperature:	25.7°C
Thermal Margin:	74.3°C (15.5 W)
Effective θ_{JA} :	4.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

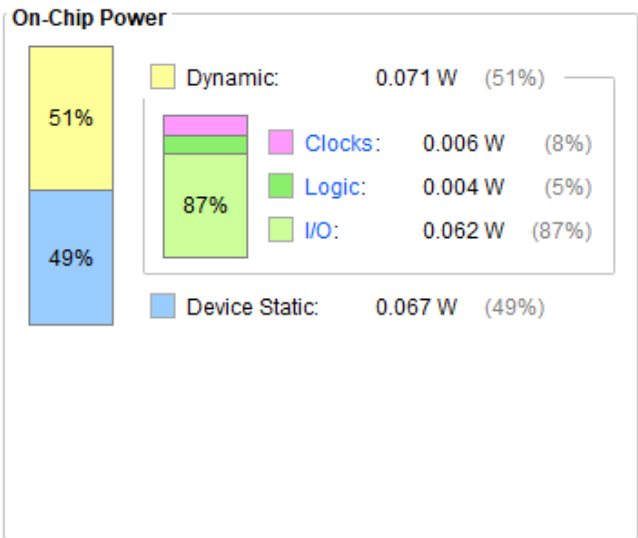


Figure 34 - Braun Power Consumption

11.10 WALLACE TREE MULTIPLIER POWER SUMMARY

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	0.267 W
Design Power Budget:	Not Specified
Power Budget Margin:	N/A
Junction Temperature:	26.3°C
Thermal Margin:	73.7°C (15.3 W)
Effective θ_{JA} :	4.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

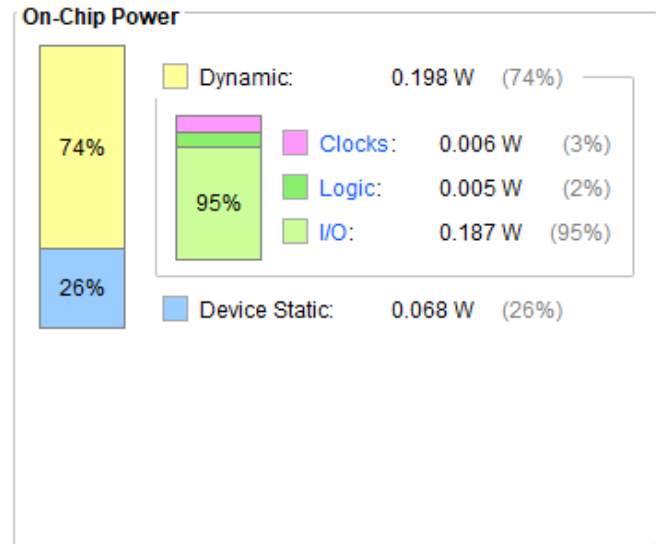


Figure 35 - Wallace tree power summary

11.11 BRAUN MULTIPLIER UTILISATION SUMMARY

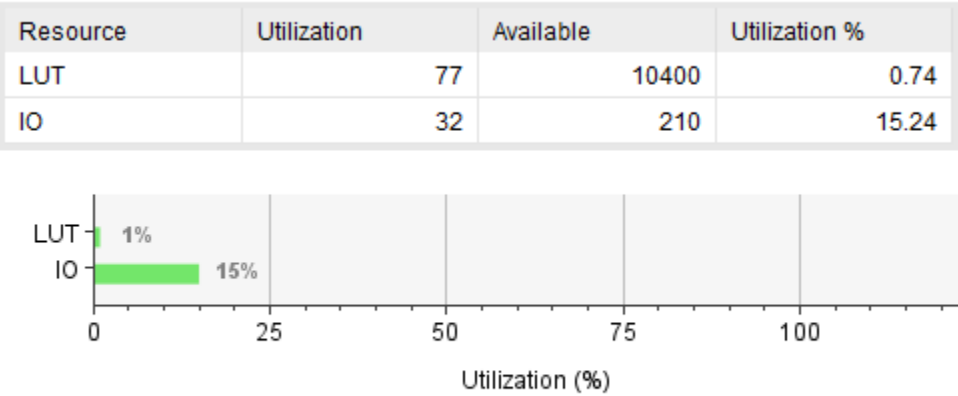


Figure 36 – Braun Utilization

11.12 WALLACE TREE MULTIPLIER UTILISATION SUMMARY

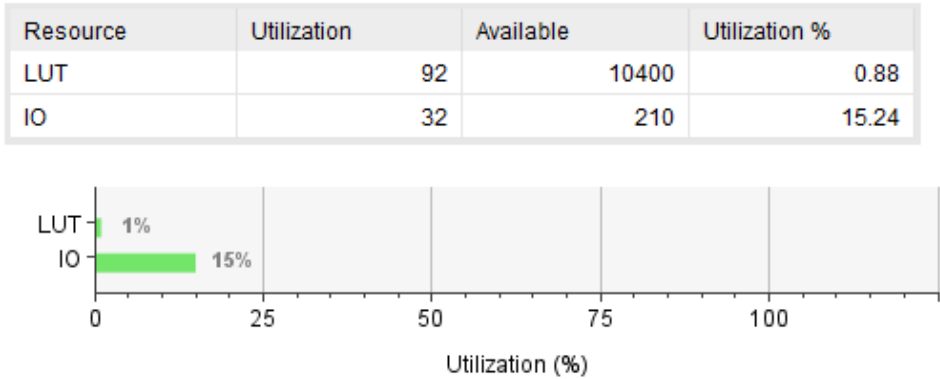


Figure 37 – Wallace tree utilization

11.13 BRAUN MULTIPLIER FULL COMBINATIONAL DELAY

From Port	To Port	Max Delay	Max Process Corner	Min Delay	Min Process Corner
m[0]	p[0]	8.26	SLOW	3.45	FAST
m[0]	p[1]	7.89	SLOW	3.31	FAST
m[0]	p[10]	13.49	SLOW	5.41	FAST
m[0]	p[11]	13.77	SLOW	5.57	FAST
m[0]	p[12]	14.86	SLOW	6.19	FAST
m[0]	p[13]	15.55	SLOW	6.47	FAST
m[0]	p[14]	13.57	SLOW	5.28	FAST
m[0]	p[15]	13.91	SLOW	5.36	FAST
m[0]	p[2]	7.22	SLOW	2.97	FAST
m[0]	p[3]	8.66	SLOW	3.50	FAST
m[0]	p[4]	8.96	SLOW	3.42	FAST
m[0]	p[5]	9.09	SLOW	3.65	FAST
m[0]	p[6]	12.30	SLOW	5.00	FAST
m[0]	p[7]	12.40	SLOW	5.08	FAST
m[0]	p[8]	12.75	SLOW	5.21	FAST
m[0]	p[9]	12.48	SLOW	4.94	FAST
m[1]	p[1]	7.89	SLOW	3.31	FAST
m[1]	p[10]	16.23	SLOW	5.45	FAST
m[1]	p[11]	16.50	SLOW	5.61	FAST
m[1]	p[12]	17.59	SLOW	6.22	FAST
m[1]	p[13]	18.28	SLOW	6.50	FAST
m[1]	p[14]	16.30	SLOW	5.31	FAST
m[1]	p[15]	16.64	SLOW	5.40	FAST
m[1]	p[2]	7.09	SLOW	2.93	FAST
m[1]	p[3]	8.54	SLOW	3.47	FAST
m[1]	p[4]	9.83	SLOW	3.64	FAST
m[1]	p[5]	11.16	SLOW	3.69	FAST
m[1]	p[6]	14.30	SLOW	5.43	FAST
m[1]	p[7]	15.14	SLOW	5.45	FAST
m[1]	p[8]	15.48	SLOW	5.50	FAST
m[1]	p[9]	15.22	SLOW	5.02	FAST
m[2]	p[10]	16.29	SLOW	4.51	FAST
m[2]	p[11]	16.56	SLOW	4.80	FAST
m[2]	p[12]	17.65	SLOW	5.41	FAST
m[2]	p[13]	18.34	SLOW	5.69	FAST
m[2]	p[14]	16.36	SLOW	4.50	FAST
m[2]	p[15]	16.70	SLOW	4.59	FAST
m[2]	p[2]	7.16	SLOW	2.91	FAST
m[2]	p[3]	9.07	SLOW	3.49	FAST

m[2]	p[4]	9.90	SLOW	3.70	FAST
m[2]	p[5]	11.22	SLOW	3.71	FAST
m[2]	p[6]	14.36	SLOW	5.23	FAST
m[2]	p[7]	15.20	SLOW	5.26	FAST
m[2]	p[8]	15.54	SLOW	5.34	FAST
m[2]	p[9]	15.28	SLOW	4.41	FAST
m[3]	p[10]	15.80	SLOW	4.39	FAST
m[3]	p[11]	16.07	SLOW	4.48	FAST
m[3]	p[12]	17.16	SLOW	5.06	FAST
m[3]	p[13]	17.85	SLOW	5.33	FAST
m[3]	p[14]	15.87	SLOW	4.15	FAST
m[3]	p[15]	16.21	SLOW	4.23	FAST
m[3]	p[3]	7.67	SLOW	3.17	FAST
m[3]	p[4]	9.41	SLOW	3.38	FAST
m[3]	p[5]	10.73	SLOW	3.39	FAST
m[3]	p[6]	13.87	SLOW	5.20	FAST
m[3]	p[7]	14.71	SLOW	5.23	FAST
m[3]	p[8]	15.05	SLOW	5.28	FAST
m[3]	p[9]	14.79	SLOW	4.48	FAST
m[4]	p[10]	15.41	SLOW	4.82	FAST
m[4]	p[11]	15.69	SLOW	4.52	FAST
m[4]	p[12]	16.78	SLOW	5.25	FAST
m[4]	p[13]	17.47	SLOW	5.52	FAST
m[4]	p[14]	15.49	SLOW	4.34	FAST
m[4]	p[15]	15.83	SLOW	4.42	FAST
m[4]	p[4]	8.82	SLOW	3.69	FAST
m[4]	p[5]	10.13	SLOW	3.86	FAST
m[4]	p[6]	13.27	SLOW	5.34	FAST
m[4]	p[7]	14.11	SLOW	5.45	FAST
m[4]	p[8]	14.69	SLOW	5.52	FAST
m[4]	p[9]	14.40	SLOW	4.96	FAST
m[5]	p[10]	18.33	SLOW	6.47	FAST
m[5]	p[11]	18.61	SLOW	6.32	FAST
m[5]	p[12]	19.69	SLOW	6.70	FAST
m[5]	p[13]	20.38	SLOW	6.51	FAST
m[5]	p[14]	18.40	SLOW	5.41	FAST
m[5]	p[15]	18.74	SLOW	5.13	FAST
m[5]	p[5]	12.80	SLOW	5.76	FAST
m[5]	p[6]	15.92	SLOW	7.16	FAST
m[5]	p[7]	16.76	SLOW	7.16	FAST
m[5]	p[8]	17.60	SLOW	7.20	FAST

m[5]	p[9]	17.32	SLOW	6.58	FAST
m[6]	p[10]	16.70	SLOW	6.34	FAST
m[6]	p[11]	16.97	SLOW	6.31	FAST
m[6]	p[12]	18.06	SLOW	6.79	FAST
m[6]	p[13]	18.75	SLOW	6.54	FAST
m[6]	p[14]	16.77	SLOW	5.29	FAST
m[6]	p[15]	17.11	SLOW	5.14	FAST
m[6]	p[6]	14.12	SLOW	6.83	FAST
m[6]	p[7]	15.00	SLOW	6.93	FAST
m[6]	p[8]	15.97	SLOW	6.98	FAST
m[6]	p[9]	15.69	SLOW	6.35	FAST
m[7]	p[10]	16.94	SLOW	6.55	FAST
m[7]	p[11]	17.22	SLOW	6.53	FAST
m[7]	p[12]	18.31	SLOW	6.68	FAST
m[7]	p[13]	19.00	SLOW	6.47	FAST
m[7]	p[14]	17.02	SLOW	4.98	FAST
m[7]	p[15]	17.36	SLOW	4.97	FAST
m[7]	p[7]	14.75	SLOW	7.15	FAST
m[7]	p[8]	16.22	SLOW	7.20	FAST
m[7]	p[9]	15.93	SLOW	6.57	FAST
n[0]	p[0]	7.84	SLOW	3.27	FAST
n[0]	p[1]	7.46	SLOW	3.13	FAST
n[0]	p[10]	16.06	SLOW	5.17	FAST
n[0]	p[11]	16.33	SLOW	5.14	FAST
n[0]	p[12]	17.42	SLOW	5.69	FAST
n[0]	p[13]	18.11	SLOW	5.55	FAST
n[0]	p[14]	16.13	SLOW	4.35	FAST
n[0]	p[15]	16.47	SLOW	4.10	FAST
n[0]	p[2]	7.56	SLOW	3.11	FAST
n[0]	p[3]	8.82	SLOW	3.56	FAST
n[0]	p[4]	9.53	SLOW	3.75	FAST
n[0]	p[5]	10.86	SLOW	3.78	FAST
n[0]	p[6]	14.00	SLOW	5.59	FAST
n[0]	p[7]	14.83	SLOW	5.63	FAST
n[0]	p[8]	15.33	SLOW	5.73	FAST
n[0]	p[9]	15.04	SLOW	5.28	FAST
n[1]	p[1]	7.42	SLOW	3.15	FAST
n[1]	p[10]	16.26	SLOW	5.28	FAST
n[1]	p[11]	16.54	SLOW	5.18	FAST
n[1]	p[12]	17.62	SLOW	5.81	FAST
n[1]	p[13]	18.31	SLOW	5.53	FAST

n[1]	p[14]	16.33	SLOW	4.33	FAST
n[1]	p[15]	16.67	SLOW	4.08	FAST
n[1]	p[2]	7.62	SLOW	3.15	FAST
n[1]	p[3]	9.09	SLOW	3.73	FAST
n[1]	p[4]	9.56	SLOW	3.78	FAST
n[1]	p[5]	10.82	SLOW	3.94	FAST
n[1]	p[6]	13.96	SLOW	5.60	FAST
n[1]	p[7]	14.79	SLOW	5.71	FAST
n[1]	p[8]	15.53	SLOW	5.81	FAST
n[1]	p[9]	15.25	SLOW	5.40	FAST
n[2]	p[10]	15.98	SLOW	5.14	FAST
n[2]	p[11]	16.25	SLOW	5.12	FAST
n[2]	p[12]	17.34	SLOW	5.67	FAST
n[2]	p[13]	18.03	SLOW	5.56	FAST
n[2]	p[14]	16.05	SLOW	4.46	FAST
n[2]	p[15]	16.39	SLOW	4.18	FAST
n[2]	p[2]	7.29	SLOW	2.99	FAST
n[2]	p[3]	8.98	SLOW	3.55	FAST
n[2]	p[4]	9.58	SLOW	3.68	FAST
n[2]	p[5]	10.91	SLOW	3.76	FAST
n[2]	p[6]	14.05	SLOW	5.57	FAST
n[2]	p[7]	14.89	SLOW	5.62	FAST
n[2]	p[8]	15.23	SLOW	5.72	FAST
n[2]	p[9]	14.97	SLOW	5.26	FAST
n[3]	p[10]	15.26	SLOW	5.17	FAST
n[3]	p[11]	15.53	SLOW	5.07	FAST
n[3]	p[12]	16.62	SLOW	5.53	FAST
n[3]	p[13]	17.31	SLOW	5.14	FAST
n[3]	p[14]	15.33	SLOW	3.93	FAST
n[3]	p[15]	15.67	SLOW	3.69	FAST
n[3]	p[3]	8.36	SLOW	3.48	FAST
n[3]	p[4]	8.87	SLOW	3.25	FAST
n[3]	p[5]	10.19	SLOW	3.70	FAST
n[3]	p[6]	13.33	SLOW	5.48	FAST
n[3]	p[7]	14.16	SLOW	5.55	FAST
n[3]	p[8]	14.51	SLOW	5.65	FAST
n[3]	p[9]	14.24	SLOW	5.19	FAST
n[4]	p[10]	13.62	SLOW	4.57	FAST
n[4]	p[11]	13.90	SLOW	4.68	FAST
n[4]	p[12]	14.99	SLOW	5.26	FAST
n[4]	p[13]	15.68	SLOW	5.12	FAST

n[4]	p[14]	13.70	SLOW	4.02	FAST
n[4]	p[15]	14.04	SLOW	3.74	FAST
n[4]	p[4]	7.89	SLOW	3.28	FAST
n[4]	p[5]	8.58	SLOW	3.32	FAST
n[4]	p[6]	11.70	SLOW	5.00	FAST
n[4]	p[7]	12.53	SLOW	5.11	FAST
n[4]	p[8]	12.88	SLOW	5.21	FAST
n[4]	p[9]	12.61	SLOW	4.59	FAST
n[5]	p[10]	12.77	SLOW	4.52	FAST
n[5]	p[11]	13.05	SLOW	4.74	FAST
n[5]	p[12]	14.14	SLOW	5.32	FAST
n[5]	p[13]	14.83	SLOW	5.15	FAST
n[5]	p[14]	12.85	SLOW	3.95	FAST
n[5]	p[15]	13.19	SLOW	3.70	FAST
n[5]	p[5]	7.56	SLOW	3.12	FAST
n[5]	p[6]	10.16	SLOW	4.62	FAST
n[5]	p[7]	11.07	SLOW	4.82	FAST
n[5]	p[8]	12.05	SLOW	4.90	FAST
n[5]	p[9]	11.76	SLOW	4.53	FAST
n[6]	p[10]	15.22	SLOW	5.94	FAST
n[6]	p[11]	15.50	SLOW	5.91	FAST
n[6]	p[12]	16.58	SLOW	6.55	FAST
n[6]	p[13]	17.27	SLOW	6.45	FAST
n[6]	p[14]	15.29	SLOW	5.36	FAST
n[6]	p[15]	15.63	SLOW	5.07	FAST
n[6]	p[6]	12.76	SLOW	6.27	FAST
n[6]	p[7]	13.41	SLOW	6.54	FAST
n[6]	p[8]	14.49	SLOW	6.61	FAST
n[6]	p[9]	14.21	SLOW	5.96	FAST
n[7]	p[10]	0.00		5.68	FAST
n[7]	p[11]	0.00		5.95	FAST
n[7]	p[12]	0.00		6.57	FAST
n[7]	p[13]	0.00		6.63	FAST
n[7]	p[14]	0.00		5.27	FAST
n[7]	p[15]	0.00		5.01	FAST
n[7]	p[7]	0.00		6.05	FAST
n[7]	p[8]	0.00		6.23	FAST
n[7]	p[9]	0.00		5.57	FAST

11.14 WALLACE TREE FULL COMBINATIONAL DELAY

From Port	To Port	Max Delay	Max Process Corner	Min Delay	Min Process Corner
m[0]	p[0]	9.84	SLOW	3.54	FAST
m[0]	p[1]	9.47	SLOW	3.41	FAST
m[0]	p[2]	9.05	SLOW	3.16	FAST
m[0]	p[3]	10.50	SLOW	3.72	FAST
m[0]	p[4]	11.44	SLOW	3.68	FAST
m[0]	p[5]	10.87	SLOW	3.42	FAST
m[0]	p[6]	15.12	SLOW	5.40	FAST
m[0]	p[7]	14.89	SLOW	5.26	FAST
m[0]	p[8]	15.70	SLOW	5.63	FAST
m[0]	p[9]	14.59	SLOW	4.90	FAST
m[0]	p[10]	15.18	SLOW	5.16	FAST
m[0]	p[11]	16.35	SLOW	5.69	FAST
m[0]	p[12]	17.10	SLOW	6.10	FAST
m[0]	p[13]	18.27	SLOW	6.56	FAST
m[0]	p[14]	15.93	SLOW	5.23	FAST
m[0]	p[15]	15.73	SLOW	5.13	FAST
m[1]	p[1]	9.47	SLOW	3.41	FAST
m[1]	p[2]	8.66	SLOW	3.02	FAST
m[1]	p[3]	11.23	SLOW	3.99	FAST
m[1]	p[4]	12.16	SLOW	4.10	FAST
m[1]	p[5]	12.14	SLOW	3.52	FAST
m[1]	p[6]	15.85	SLOW	5.91	FAST
m[1]	p[7]	15.62	SLOW	5.70	FAST
m[1]	p[8]	16.35	SLOW	5.63	FAST
m[1]	p[9]	15.24	SLOW	5.04	FAST
m[1]	p[10]	15.83	SLOW	5.29	FAST
m[1]	p[11]	17.08	SLOW	5.83	FAST
m[1]	p[12]	17.83	SLOW	6.23	FAST
m[1]	p[13]	18.99	SLOW	6.70	FAST
m[1]	p[14]	16.66	SLOW	5.37	FAST
m[1]	p[15]	16.46	SLOW	5.27	FAST
m[2]	p[2]	8.74	SLOW	3.00	FAST
m[2]	p[3]	10.62	SLOW	3.49	FAST
m[2]	p[4]	11.55	SLOW	3.77	FAST
m[2]	p[5]	11.25	SLOW	3.59	FAST
m[2]	p[6]	15.25	SLOW	5.42	FAST
m[2]	p[7]	15.12	SLOW	5.28	FAST
m[2]	p[8]	16.48	SLOW	5.63	FAST
m[2]	p[9]	15.37	SLOW	4.77	FAST

m[2]	p[10]	15.95	SLOW	5.03	FAST
m[2]	p[11]	16.62	SLOW	5.41	FAST
m[2]	p[12]	17.37	SLOW	5.91	FAST
m[2]	p[13]	18.54	SLOW	6.37	FAST
m[2]	p[14]	16.20	SLOW	5.04	FAST
m[2]	p[15]	16.00	SLOW	4.94	FAST
m[3]	p[3]	10.09	SLOW	3.62	FAST
m[3]	p[4]	11.03	SLOW	3.76	FAST
m[3]	p[5]	11.07	SLOW	3.61	FAST
m[3]	p[6]	14.72	SLOW	5.40	FAST
m[3]	p[7]	14.49	SLOW	5.20	FAST
m[3]	p[8]	15.60	SLOW	5.61	FAST
m[3]	p[9]	14.49	SLOW	4.96	FAST
m[3]	p[10]	15.08	SLOW	5.08	FAST
m[3]	p[11]	15.95	SLOW	5.24	FAST
m[3]	p[12]	16.70	SLOW	5.62	FAST
m[3]	p[13]	17.86	SLOW	6.09	FAST
m[3]	p[14]	15.52	SLOW	4.75	FAST
m[3]	p[15]	15.32	SLOW	4.65	FAST
m[4]	p[4]	11.18	SLOW	3.98	FAST
m[4]	p[5]	11.00	SLOW	3.93	FAST
m[4]	p[6]	14.36	SLOW	5.52	FAST
m[4]	p[7]	14.13	SLOW	5.10	FAST
m[4]	p[8]	15.31	SLOW	5.53	FAST
m[4]	p[9]	14.20	SLOW	4.92	FAST
m[4]	p[10]	14.79	SLOW	5.05	FAST
m[4]	p[11]	15.77	SLOW	5.24	FAST
m[4]	p[12]	16.52	SLOW	5.48	FAST
m[4]	p[13]	17.69	SLOW	5.39	FAST
m[4]	p[14]	15.35	SLOW	4.37	FAST
m[4]	p[15]	15.15	SLOW	4.17	FAST
m[5]	p[5]	13.30	SLOW	5.47	FAST
m[5]	p[6]	17.22	SLOW	7.27	FAST
m[5]	p[7]	16.98	SLOW	7.05	FAST
m[5]	p[8]	18.44	SLOW	7.24	FAST
m[5]	p[9]	17.33	SLOW	6.51	FAST
m[5]	p[10]	17.92	SLOW	6.54	FAST
m[5]	p[11]	18.59	SLOW	6.64	FAST
m[5]	p[12]	19.34	SLOW	7.12	FAST
m[5]	p[13]	20.51	SLOW	7.02	FAST
m[5]	p[14]	18.17	SLOW	5.40	FAST

m[5]	p[15]	17.97	SLOW	5.81	FAST
m[6]	p[6]	16.39	SLOW	7.16	FAST
m[6]	p[7]	17.00	SLOW	6.94	FAST
m[6]	p[8]	17.94	SLOW	6.95	FAST
m[6]	p[9]	16.77	SLOW	6.18	FAST
m[6]	p[10]	17.16	SLOW	6.27	FAST
m[6]	p[11]	18.36	SLOW	6.48	FAST
m[6]	p[12]	19.11	SLOW	6.73	FAST
m[6]	p[13]	20.27	SLOW	6.47	FAST
m[6]	p[14]	17.94	SLOW	5.22	FAST
m[6]	p[15]	17.74	SLOW	5.42	FAST
m[7]	p[7]	16.81	SLOW	7.36	FAST
m[7]	p[8]	17.74	SLOW	7.05	FAST
m[7]	p[9]	16.58	SLOW	6.28	FAST
m[7]	p[10]	16.97	SLOW	6.37	FAST
m[7]	p[11]	18.17	SLOW	6.60	FAST
m[7]	p[12]	18.92	SLOW	6.88	FAST
m[7]	p[13]	20.08	SLOW	6.57	FAST
m[7]	p[14]	17.74	SLOW	5.41	FAST
m[7]	p[15]	17.54	SLOW	5.12	FAST
n[0]	p[0]	9.19	SLOW	3.29	FAST
n[0]	p[1]	8.82	SLOW	3.15	FAST
n[0]	p[2]	9.14	SLOW	3.21	FAST
n[0]	p[3]	10.03	SLOW	3.48	FAST
n[0]	p[4]	11.39	SLOW	3.61	FAST
n[0]	p[5]	11.22	SLOW	3.56	FAST
n[0]	p[6]	15.00	SLOW	5.56	FAST
n[0]	p[7]	14.86	SLOW	5.36	FAST
n[0]	p[8]	16.23	SLOW	5.76	FAST
n[0]	p[9]	15.12	SLOW	5.07	FAST
n[0]	p[10]	15.70	SLOW	5.27	FAST
n[0]	p[11]	16.37	SLOW	5.85	FAST
n[0]	p[12]	17.12	SLOW	6.26	FAST
n[0]	p[13]	18.29	SLOW	6.73	FAST
n[0]	p[14]	15.95	SLOW	5.39	FAST
n[0]	p[15]	15.75	SLOW	5.29	FAST
n[1]	p[1]	9.23	SLOW	3.35	FAST
n[1]	p[2]	8.58	SLOW	3.00	FAST
n[1]	p[3]	10.68	SLOW	3.69	FAST
n[1]	p[4]	11.61	SLOW	3.89	FAST
n[1]	p[5]	12.14	SLOW	3.76	FAST

n[1]	p[6]	15.51	SLOW	5.68	FAST
n[1]	p[7]	15.28	SLOW	5.54	FAST
n[1]	p[8]	16.60	SLOW	5.90	FAST
n[1]	p[9]	15.49	SLOW	5.14	FAST
n[1]	p[10]	16.08	SLOW	5.23	FAST
n[1]	p[11]	16.75	SLOW	5.44	FAST
n[1]	p[12]	17.50	SLOW	6.05	FAST
n[1]	p[13]	18.67	SLOW	6.51	FAST
n[1]	p[14]	16.33	SLOW	5.18	FAST
n[1]	p[15]	16.13	SLOW	5.07	FAST
n[2]	p[2]	8.87	SLOW	3.08	FAST
n[2]	p[3]	10.32	SLOW	3.55	FAST
n[2]	p[4]	11.33	SLOW	3.75	FAST
n[2]	p[5]	11.73	SLOW	3.62	FAST
n[2]	p[6]	15.10	SLOW	5.62	FAST
n[2]	p[7]	15.02	SLOW	5.48	FAST
n[2]	p[8]	16.05	SLOW	5.82	FAST
n[2]	p[9]	14.94	SLOW	5.10	FAST
n[2]	p[10]	15.53	SLOW	5.20	FAST
n[2]	p[11]	16.38	SLOW	5.32	FAST
n[2]	p[12]	17.13	SLOW	5.85	FAST
n[2]	p[13]	18.30	SLOW	6.31	FAST
n[2]	p[14]	15.96	SLOW	4.98	FAST
n[2]	p[15]	15.76	SLOW	4.88	FAST
n[3]	p[3]	9.40	SLOW	3.36	FAST
n[3]	p[4]	10.65	SLOW	3.50	FAST
n[3]	p[5]	10.75	SLOW	3.44	FAST
n[3]	p[6]	15.22	SLOW	5.46	FAST
n[3]	p[7]	15.10	SLOW	5.26	FAST
n[3]	p[8]	16.45	SLOW	5.52	FAST
n[3]	p[9]	15.34	SLOW	4.85	FAST
n[3]	p[10]	15.92	SLOW	4.94	FAST
n[3]	p[11]	16.60	SLOW	5.30	FAST
n[3]	p[12]	17.34	SLOW	5.72	FAST
n[3]	p[13]	18.51	SLOW	5.63	FAST
n[3]	p[14]	16.17	SLOW	4.60	FAST
n[3]	p[15]	15.97	SLOW	4.41	FAST
n[4]	p[4]	10.97	SLOW	3.97	FAST
n[4]	p[5]	11.05	SLOW	3.40	FAST
n[4]	p[6]	14.52	SLOW	5.25	FAST
n[4]	p[7]	14.52	SLOW	5.11	FAST

n[4]	p[8]	15.75	SLOW	5.35	FAST
n[4]	p[9]	14.64	SLOW	4.71	FAST
n[4]	p[10]	15.23	SLOW	4.86	FAST
n[4]	p[11]	15.90	SLOW	5.25	FAST
n[4]	p[12]	16.65	SLOW	5.55	FAST
n[4]	p[13]	17.81	SLOW	5.46	FAST
n[4]	p[14]	15.48	SLOW	3.88	FAST
n[4]	p[15]	15.27	SLOW	4.17	FAST
n[5]	p[5]	11.10	SLOW	4.04	FAST
n[5]	p[6]	14.66	SLOW	5.55	FAST
n[5]	p[7]	14.54	SLOW	5.17	FAST
n[5]	p[8]	15.89	SLOW	5.45	FAST
n[5]	p[9]	14.78	SLOW	4.80	FAST
n[5]	p[10]	15.36	SLOW	4.96	FAST
n[5]	p[11]	16.03	SLOW	5.35	FAST
n[5]	p[12]	16.78	SLOW	5.63	FAST
n[5]	p[13]	17.95	SLOW	5.53	FAST
n[5]	p[14]	15.61	SLOW	4.39	FAST
n[5]	p[15]	15.41	SLOW	4.31	FAST
n[6]	p[6]	16.13	SLOW	7.17	FAST
n[6]	p[7]	16.34	SLOW	6.95	FAST
n[6]	p[8]	17.36	SLOW	7.00	FAST
n[6]	p[9]	16.25	SLOW	6.14	FAST
n[6]	p[10]	16.84	SLOW	6.39	FAST
n[6]	p[11]	17.78	SLOW	6.81	FAST
n[6]	p[12]	18.53	SLOW	7.02	FAST
n[6]	p[13]	19.70	SLOW	6.87	FAST
n[6]	p[14]	17.36	SLOW	5.18	FAST
n[6]	p[15]	17.16	SLOW	5.65	FAST
n[7]	p[7]	15.54	SLOW	6.87	FAST
n[7]	p[8]	17.17	SLOW	6.84	FAST
n[7]	p[9]	16.69	SLOW	6.09	FAST
n[7]	p[10]	17.29	SLOW	6.33	FAST
n[7]	p[11]	18.55	SLOW	6.70	FAST
n[7]	p[12]	19.30	SLOW	7.11	FAST
n[7]	p[13]	20.46	SLOW	6.58	FAST
n[7]	p[14]	18.13	SLOW	5.40	FAST
n[7]	p[15]	17.93	SLOW	5.54	FAST

12 REFERENCES

- [1] M. V. T. A.L Siridhara, "Performance Evaluation of Parallel Multipliers," *Journal of Engineering and Applied Sciences*, vol. 12, p. 20, 2017.
- [2] P. Horowitz and W. Hill, "DIGITAL LOGIC," in *The Art of Electronics - Third Edition*, Cambridge University Press, 2016, pp. 703-707.
- [3] J. Bird, "Chapter 10 - Binary, octal and hexadecimal numbers," in *HIGHER ENGINEERING MATHEMATICS - SEVENTH EDITION*, Routledge, 2014, pp. 90 - 91.
- [4] D. M. Harris and S. L.Harris, "1.4 Number Systems," in *Digital Design and Computer Architecture - 2nd Edition*, Morgan Kaufmann, 2012, p. 15.
- [5] P. Horowitz and W. Hill, "DIGITAL LOGIC - 10.1.4 Gates and truth tables," in *The Art of Electronics - Third Edition*, Cambridge University Press, 2016, pp. 709 - 710.
- [6] P. M. Albert and A. B. Jerald, in *Digital Computer Electronics - Third Edition*, McGraw Hill Education (India) Private Limited, 1993, pp. 24,38.
- [7] P. M. Albert and A. B. Jerald, "ARITHMETIC-LOGIC UNITS," in *Digital Computer Electronics - Third Edition*, McGraw Hill Education (India) Private Limited, 1993, p. 81.
- [8] D. M. Harris and S. L.Harris, "Digital Building Blocks," in *Digital Design and Computer Architecture - 2nd Edition*, Morgan Kaufmann, 2012, pp. 246-247.
- [9] M. Sadeghi, M. Zahedi and M. Ali, "The Cascade Carry Array Multiplier – A Novel Structure of Digital Unsigned Multipliers for Low-Power Consumption and Ultra-Fast Applications," *Annals of Emerging Technologies in Computing (AETiC)*, vol. 3, no. 3, p. 22, 2019.
- [10] Elprocus, "4x4 Array Multiplier : Construction, Working and Applications," Elprocus, [Online]. Available: <https://www.elprocus.com/4x4-array-multiplier-and-its-working/>. [Accessed 16 04 2023].
- [11] Srikanth Immareddy and Aunmetha Sundaramoorthy, "A survey paper on design and implementation of multipliers," *Artificial Intelligence Review*, no. 55, 2022.
- [12] Cadence, "Hardware Description Languages: VHDL vs Verilog, and Their Functional Uses," Cadence, [Online]. Available: <https://resources.pcb.cadence.com/blog/2020-hardware-description-languages-vhdl-vs-verilog-and-their-functional-uses>. [Accessed 16 04 2023].
- [13] XILINX, "LUT," XILINX, [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/yeo1504034293627.html#:~:text=The%20LUT%20is%20the%20basic,functions%20to%20yield%20output%20values.. [Accessed 01 05 2023].

- [14] XILINX, "Vivado ML Overview," XILINX, [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado.html#overview>. [Accessed 1 05 2023].