

Electronics 2

ACDA6001

Assignment One

Jack Spillett

Eastman Chemical Company

Contents

Abstract.....	3
Background Research	4
Introduction.....	4
Multiplier Techniques.....	4
Recent Developments in Multiplier Techniques.....	5
Advantages and Disadvantages of each Multiplier.....	5
Applications of Binary Multipliers.....	6
VHDL	6
FPGA	7
Section One - 8X8 Braun's Multiplier	8
Logic Circuit Diagram.....	8
Component Declaration and Port Mapping.....	9
Signal Declaration and Source.....	10
Partial Product Reduction and Summation.....	11
Program Limitations.....	11
8X8 Brauns Test Results.....	12
8X8 Brauns Synthesis.....	14
Section Two - 8X8 Wallace Multiplier	16
Dot Diagram and Logic Circuit Diagram.....	16
Component Declaration and Port Mapping.....	18
Signal Declaration and Source.....	19
Partial Product Reduction and Summation.....	20
Program Limitations.....	20
8X8 Wallace Test Results.....	21
8X8 Wallace Synthesis.....	23
Braun's Testing on Hardware.....	25
Wallace Tree Testing on Hardware.....	26
Timing Constraints.....	27
Braun's Timing Delay Evidence	29
Wallace Tree Timing Delay Evidence	31
Delay Comparison between Multipliers	33
Braun's Power and Utilisation Evidence.....	33
Wallace Tree Power and Utilisation Evidence.....	35
Utilisation and Power Consumption Comparison.....	36
Conclusion.....	37
Appendicies.....	38
Appendix 1 - VHDL Code - Braun's.....	39
Appendix 2 - VHDL Code - Wallace Tree.....	43
Appendix 3 - References	46

Abstract

Multiplication is vital for digital processing, impacting a digital system's speed and power. This report presents the research, design, modeling, and performance analysis that went into producing and critically comparing Braun's and Wallace Tree multiplier techniques using VHDL.

This report begins by presenting background research on multiplier techniques, their applications and advantages; in summary, Braun's multiplier generates and sums partial products sequentially; it is simple but limited by propagation delays. In contrast, the Wallace Tree multiplier employs carry-save adders, improving speed by reducing sequential additions.

The second section details the design process for an 8-bit Braun multiplier, provides simulation results, and synthesises the system for FPGA design. The section afterwards presents the design and implementation of an 8×8 Wallace Tree multiplier using parallel multiplication, again providing simulation results, and the synthesis of the system for FPGA design. The last section shows the critical evaluation of their performance using Xilinx Vivado Software.

Assigning this task to students offered an opportunity to learn the fundamentals of VHDL programming and two binary multiplication approaches. The task also allows for a better understanding of the practical application of binary multiplication and where they are used.

Background Research

Introduction

Binary Multiplication is a fundamental operation in digital processing, impacting the performance of computing applications. Different multiplier architectures offer different benefits in terms of speed, power efficiency, and complexity. This section explores the various techniques, their advantages and disadvantages, their applications in real-world scenarios and recent advancements. Understanding these factors is vital in making informed decisions when designing and comparing the Braun's and Wallace Tree multiplier techniques later in this report.

Multiplier Techniques

Binary multiplication techniques include Array Multipliers, Booth Multipliers, Braun's Multipliers and Wallace Tree Multipliers. Array multipliers are simple and easy to implement but suffer from high latency due to the sequential addition of partial products. Booth multipliers improve signed number multiplication by reducing the number of partial products, making them more efficient for signed applications. Braun's multipliers, commonly used for unsigned multiplication, are structurally simple and work efficiently for small-scale operations. Wallace Tree multipliers enhance speed by employing hierarchical reduction techniques, reducing the number of sequential additions and improving performance [1],[2],[3].

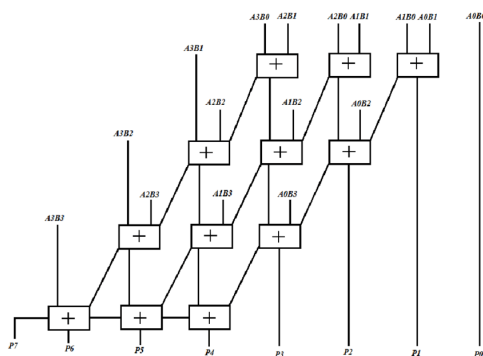


Figure 1 - Braun's Multiplier [4]

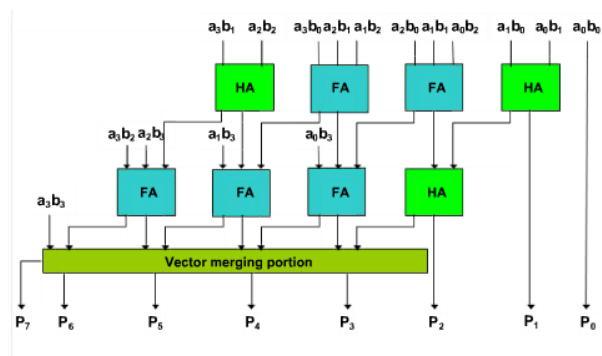


Figure 2 - Wallace Multiplier Architecture [5]

Choosing which multiplier is suitable depends on the required speed, allowable complexity, and power consumption requirements. Array multipliers are often used in small-scale applications where simplicity is key. Booth multipliers are preferred for signed arithmetic operations due to their efficiency in reducing partial products. Braun's multipliers are effective in basic digital circuits, while Wallace Tree multipliers are ideal for high-speed computing environments due to their parallel processing capabilities.

Recent Developments in Multiplier Techniques

Recent advancements in multiplier technology focus on enhancing speed, reducing power consumption, and reducing the resources required. Techniques such as modified Booth multipliers, hybrid multipliers and approximate computing methods have been introduced to improve efficiency in FPGA and ASIC implementations [6], [7]. These developments aim to balance trade-offs between power, speed, and accuracy, making them suitable for a wider range of applications.

Quantum multipliers are an emerging field of research, leveraging quantum computing principles to perform multiplications more efficiently than classical methods [9]. AI is also being used to optimise binary multipliers using machine learning algorithms based on specific computational requirements [8]. Additionally, low-power multipliers are being developed for energy-efficient applications such as basic IoT devices like smart watches [10].

Advantages and Disadvantages of each Multiplier

Braun's Multiplier:

Advantages: Simple structure, efficient for unsigned numbers, and easy to implement.

Disadvantages: High propagation delay due to sequential addition of partial products, limited scalability, and increased circuit complexity for large-bit operations.

Wallace Tree Multiplier:

Advantages: Faster computation due to parallel reduction of partial products, reduced delay, and improved efficiency in high-speed applications.

Disadvantages: More complex design, higher hardware resource consumption, and greater power requirements.

Booth Multiplier:

Advantages: Reduces the number of partial products, making it more efficient for signed multiplication and reducing overall computation time.

Disadvantages: Increased complexity in logic implementation and slightly higher latency for small-bit operations.

Array Multiplier:

Advantages: Simple and regular structure, making it easy to design and implement.

Disadvantages: High latency due to the carry propagation delay and inefficiency in large-bit multiplications.

Applications of Binary Multipliers

Binary multipliers are essential components in various computational fields, playing a crucial role in digital and embedded systems. Some applications include:

- Digital Signal Processing (DSP): Used in image processing, audio filtering, convolution operations, and real-time computations [11].
- Cryptography: Essential for encryption and decryption algorithms, ensuring secure data transmission and processing [8].
- Neural Networks and Machine Learning: Crucial for matrix multiplications, weight updates, and various AI computations in deep learning models [11].
- Embedded Systems: Used in microcontrollers for signal processing, control systems, and IoT applications requiring efficient computation with limited hardware resources [11].
- Graphics Processing: Fundamental in GPUs for accelerating image transformations, rendering, and pixel manipulations in gaming and animation [11].
- Scientific Computing: Integral to high-performance computing (HPC) environments, where rapid multiplication is necessary for simulations, modeling, and numerical analysis [9].

VHDL

VHDL (VHSIC Hardware Description Language) is a widely used language for modeling and simulating digital systems such as FPGAs and ASICs [12]. Developed in the 1980s through a U.S. Department of Defense initiative. It enables designers to describe digital circuits at multiple abstraction levels, including behavioral, dataflow, and structural [13].

VHDL supports both simulation and synthesis. Simulation allows verification of logic functionality before uploading to hardware, helping catch and correct design flaws early. Once validated, VHDL code can be synthesised into a gate-level netlist for implementation on an FPGA or ASIC [12],[13],[14].

The language encourages modular design through the use of entities and architectures; entities define I/O, while architectures describe internal logic. This structure makes it suitable for large, scalable systems. VHDL also supports both concurrent (parallel) and sequential (ordered) programming styles [13],[16].

A major advantage of VHDL is its versatility, capable of modeling anything from simple logic gates to complex processors. It also supports testbenches, which simulate and verify designs without hardware [14]. Its strict syntax helps minimize errors and improve readability, making it a preferred choice in high-reliability sectors such as aerospace and defense [16].

FPGA

FPGAs (Field-Programmable Gate Arrays) are 'circuits' which can be configured by the user after manufacturing to perform specific digital logic functions. Unlike fixed-function chips like CPUs or ASICs, FPGAs can be reprogrammed, allowing engineers to implement custom hardware designs without fabricating a new chip, making them ideal for prototyping.

FPGAs consist of configurable logic blocks, programmable interconnects and I/O blocks, the logic of which can be 'set' using VHDL or Verilog to define the behavior of the chip. FPGAs can work in parallel, meaning they can execute operations simultaneously, unlike CPUs. This makes FPGAs highly efficient for tasks like signal processing, encryption, and data analysis.

Modern FPGAs also include advanced features such as embedded processors, memory blocks and high-speed transceivers, enabling them to handle complex processing without the need for external components, i.e. memory. FPGAs are widely used in telecommunications and the automotive, aerospace, and finance industries due to their flexibility and speed.

Section One - 8X8 Braun's Multiplier

Logic Circuit Diagram

To write the code for an 8X8 Braun's Multiplier, a logic diagram was produced to reference which sum, carry, product, etc., is part of each row. The longest path can be seen in green, ready for when creating the timing constraints. A Braun's multiplier works by multiplying each bit in one row by every bit on the other row, and the total is then summed together using adders.

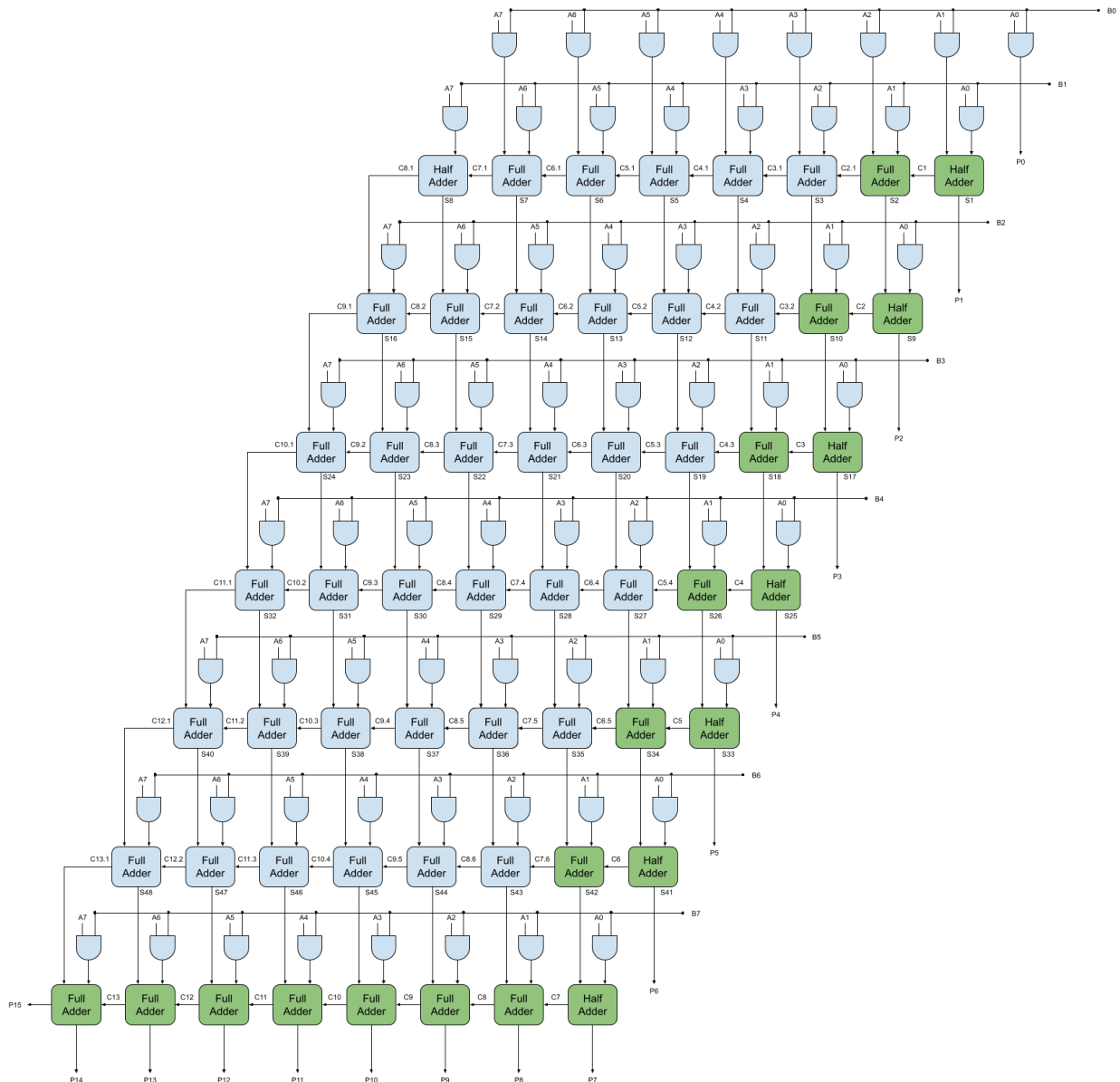


Figure 3 - 8X8 Braun's Multiplier Logic Diagram

Component Declaration and Port Mapping

First, the components were declared, describing the interface of the module i.e. specifying ports A and B as inputs and port P as an output. Afterwards, in the code, Port Mapping was used to connect the individual Full Adder and Half Adder programs to the 8X8 Brauns Program. Using port mapping made the design readable and scalable.

The full program can be found in Appendix 1

Component Declaration

```
entity Braun_8x8 is
PORT (A : in STD_LOGIC_VECTOR (7 downto 0);
      B : in STD_LOGIC_VECTOR (7 downto 0);
      P : out STD_LOGIC_VECTOR (15 downto 0));

end Braun_8x8;
```

Port Mapping

```
Component full_adder
port (A : in STD_LOGIC;
      B : in STD_LOGIC;
      Cin : in STD_LOGIC;
      S : out STD_LOGIC;
      Cout : out STD_LOGIC);
end component;

Component half_adder
port (A : in STD_LOGIC;
      B : in STD_LOGIC;
      S : out STD_LOGIC;
      Cout : out STD_LOGIC);
end component;
```

Signal Declaration and Source

After port mapping, the signals were declared for the Carry's and Inputs, then the main program was started, declaring what inputs made up each input signal.

Signal Declaration

```
signal A1B0 : STD_LOGIC;  
signal A2B0 : STD_LOGIC;  
signal A3B0 : STD_LOGIC;  
signal A4B0 : STD_LOGIC;  
signal A5B0 : STD_LOGIC;  
signal A6B0 : STD_LOGIC;  
signal A7B0 : STD_LOGIC;  
...  
  
signal C1   : STD_LOGIC;  
  
signal C2   : STD_LOGIC;  
signal C2_1 : STD_LOGIC;  
...
```

Signal Source

```
A1B0 <= A (1) AND B (0);  
A2B0 <= A (2) AND B (0);  
A3B0 <= A (3) AND B (0);  
A4B0 <= A (4) AND B (0);  
A5B0 <= A (5) AND B (0);  
A6B0 <= A (6) AND B (0);  
A7B0 <= A (7) AND B (0);  
...
```

Partial Product Reduction and Summation

The remainder of the code used the port mapping to synthesise the product, starting with P0, the least significant bit.

```
P0 : P(0) <= A(0) and B(0);

P1_0 : half_adder port map (A=>A1B0 , B=>A0B1 ,S=> P(1), Cout=>C1);

P21 : full_adder port map (A=>A2B0 , B=>A1B1 , Cin=>C1, S=>P2_1, Cout=>C2_1);
P2_0 : half_adder port map (A=>A0B2 , B=>P2_1 , S=>P(2), Cout=>C2);

P31 : full_adder port map (A=>A3B0 , B=>A2B1, Cin=>C2_1, S=>P3_1, Cout=>C3_1);
P32 : full_adder port map (A=>A1B2, B=>P3_1 , Cin=>C2, S=>P3_2, Cout=>C3_2);
P3_0 : half_adder port map (A=>A0B3, B=>P3_2, S=>P(3),Cout=>C3);

...
```

Program Limitations

There are several limitations with the Braun's multiplier and how it has been programmed. Firstly, the multiplier only works with unsigned integers, hence it can't handle negative numbers. Secondly, the main 'logic' of the program is bulky, with each signal and adder being declared individually, meaning that the program is hard to fault-find on. The program is entirely combinational, i.e. there are no registers or 'clocks', so each multiplication must finish in one cycle, before the next can start, limiting the maximum operating frequency. Although not an issue now, as the number of bits being multiplied increases, the delay will increase. Furthermore, all sums and carries are generated and added regardless of their value. Hence why it is less power-efficient compared to multipliers like the Booth or Wallace Tree. Finally, although the Braun structure is simple and repetitive, it is impractical to scale up beyond the 8X8. Excess area and power is used in practice, on a Field-Programmable Gate Array, without the improvements previously mentioned being made.

8X8 Brauns Test Results

Before testing, the inputs and outputs needed to be assigned using the I/O list, following the Basys 3 manual. Four simulations were run to test that the Braun's Multiplier had been programmed correctly. First, the clock is forced for both inputs, then a constant is set. In test one, this was 12 for A and 24 for B. The resultant output was 0288, indicating that the program is correct; however, to ensure this, three subsequent tests were done.

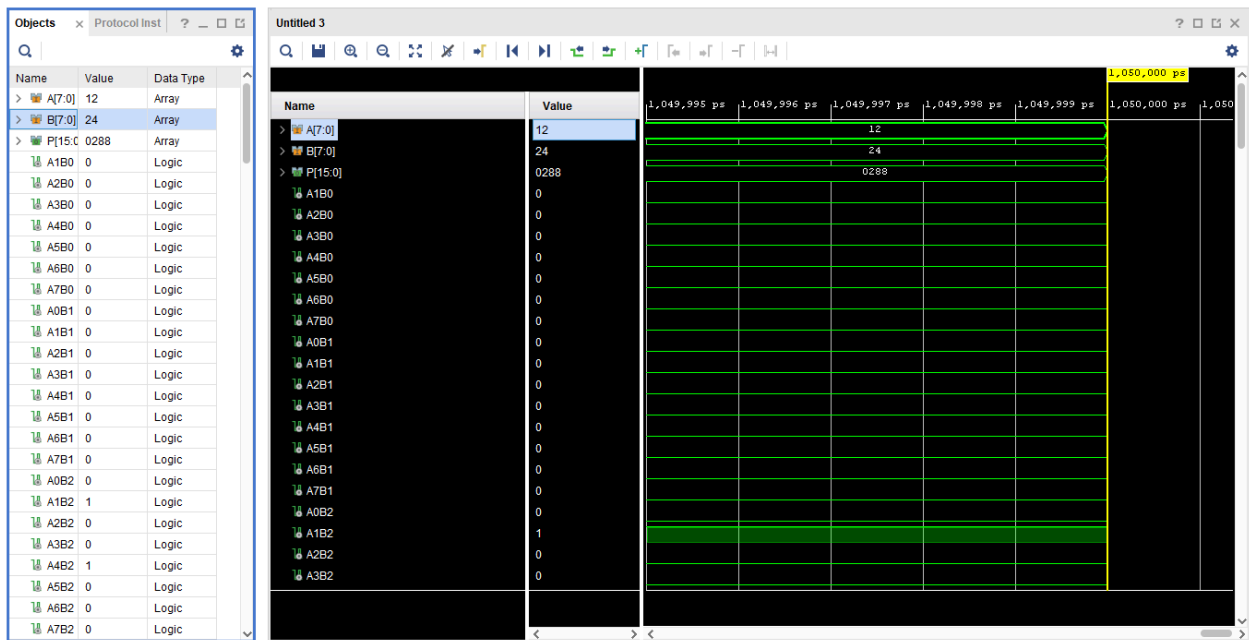


Figure 4 - 8X8 Braun's Multiplier Test One 12 x 24 = 0288

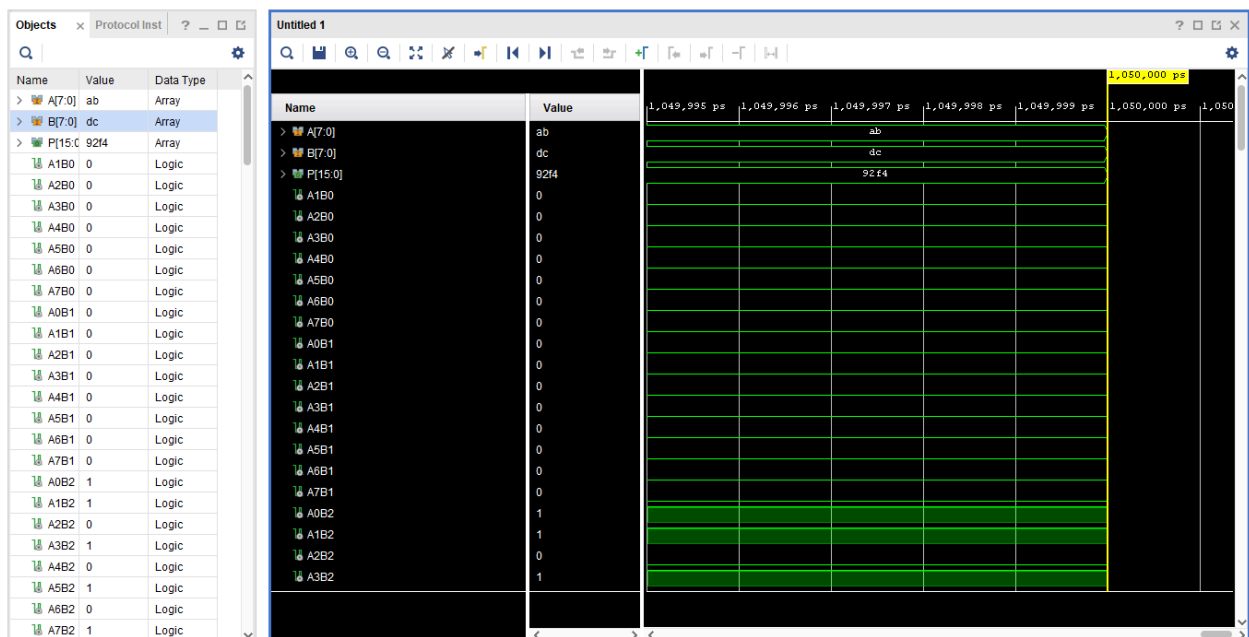


Figure 5 - 8X8 Braun's Multiplier Test Two ab x dc = 92f4

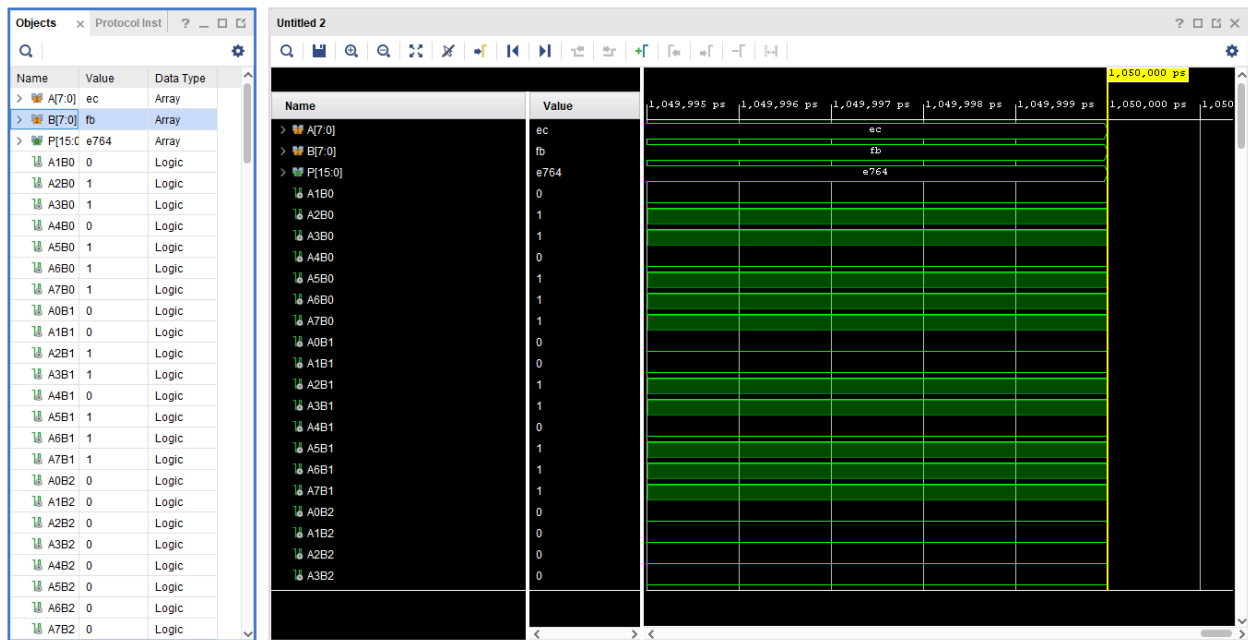


Figure 6 - 8X8 Braun's Multiplier Test Three $ec \times fb = e764$

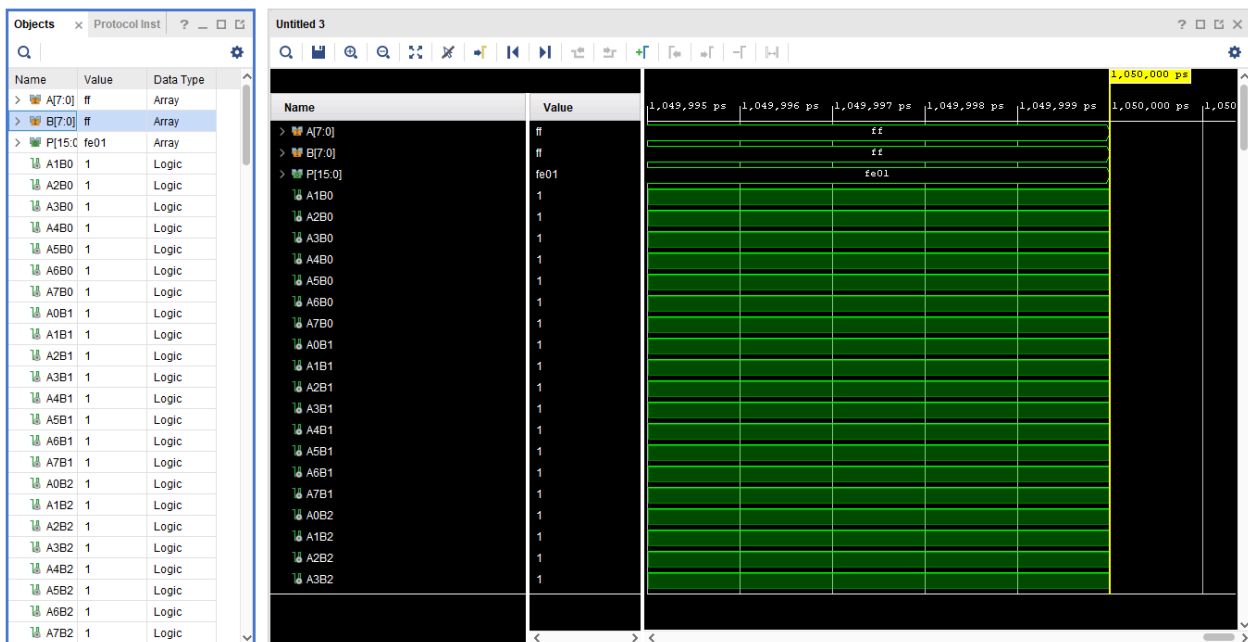


Figure 7 - 8X8 Braun's Multiplier Test Four $ff \times ff = fe01$

8X8 Brauns Synthesis

Having tested the program, Inputs and Outputs were set, for a Schematic, RTL Design, Device Diagram and I/O Planning Diagram to be produced.

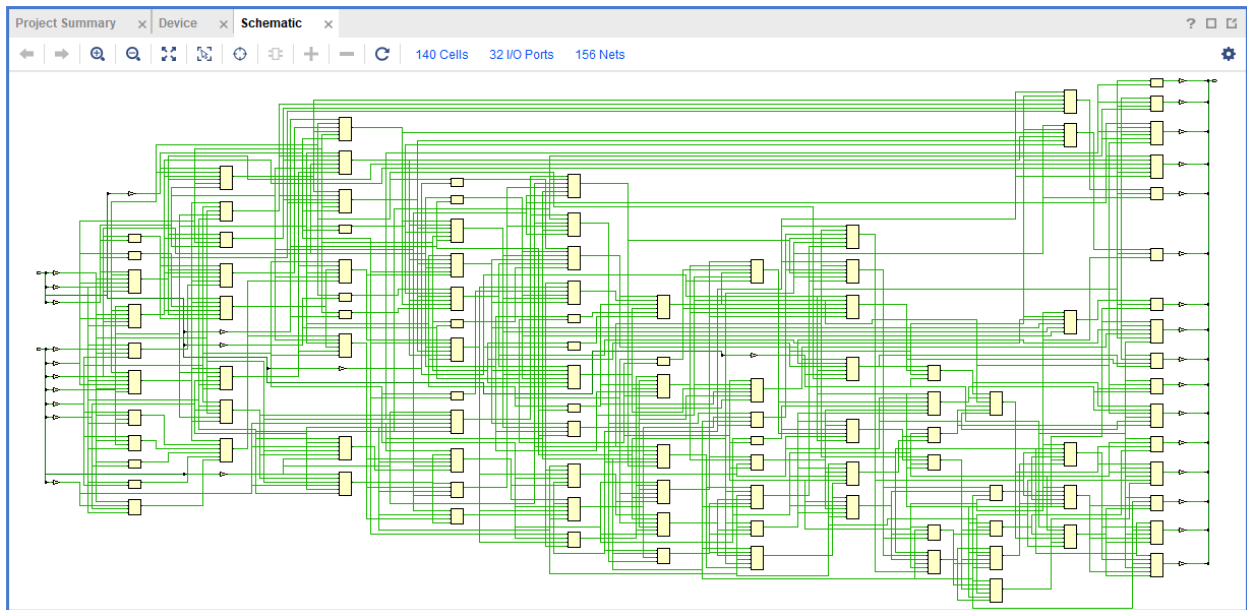


Figure 8 - 8X8 Braun's Multiplier Schematic

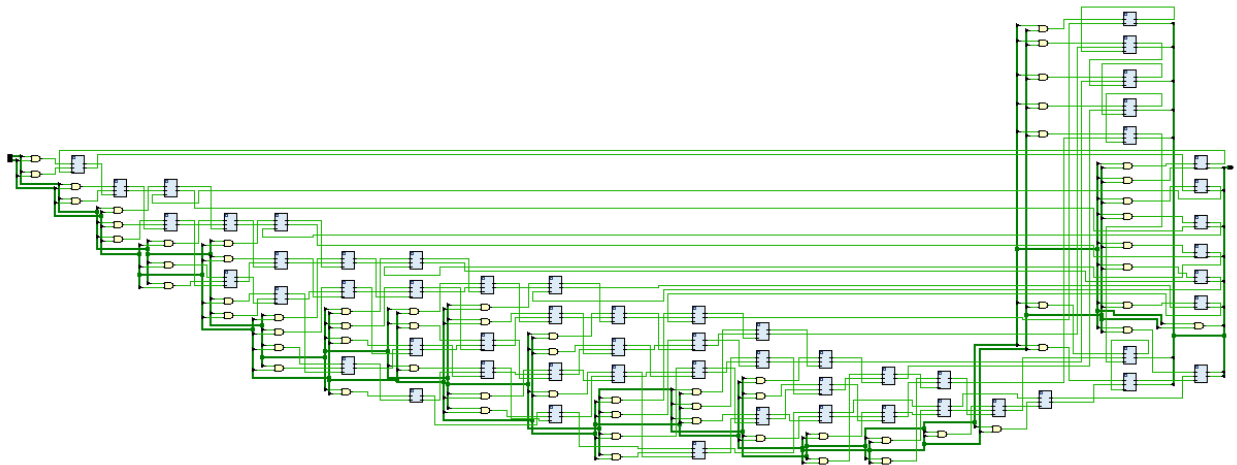


Figure 9 - 8X8 Braun's Multiplier RTL Design

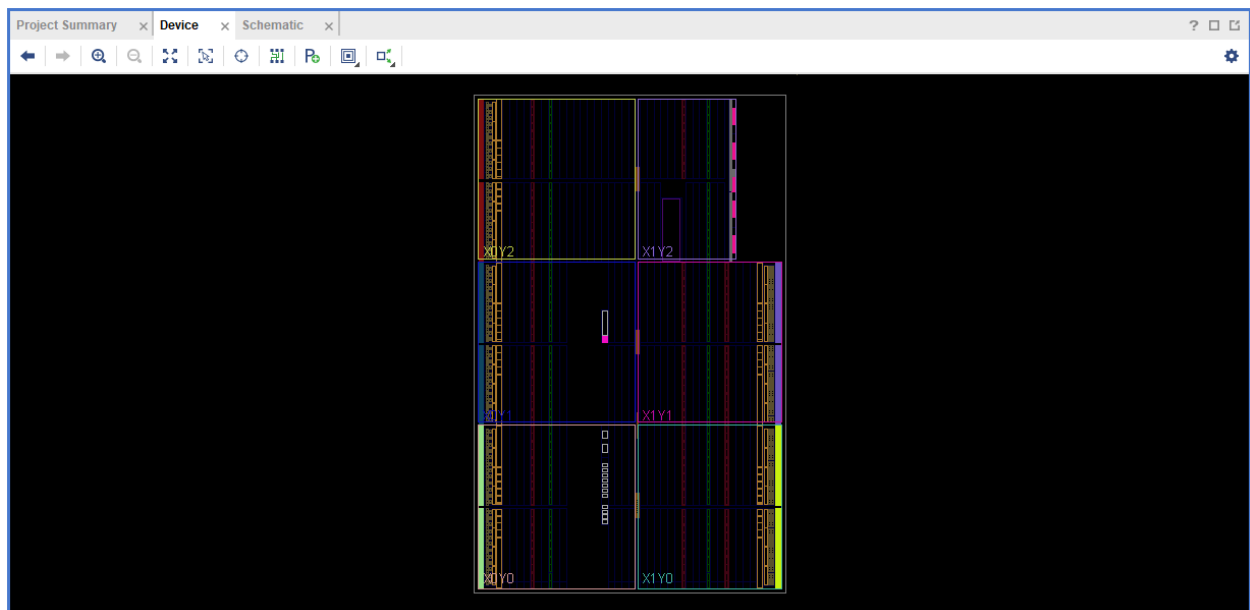


Figure 10 - 8X8 Braun's Multiplier Device Design

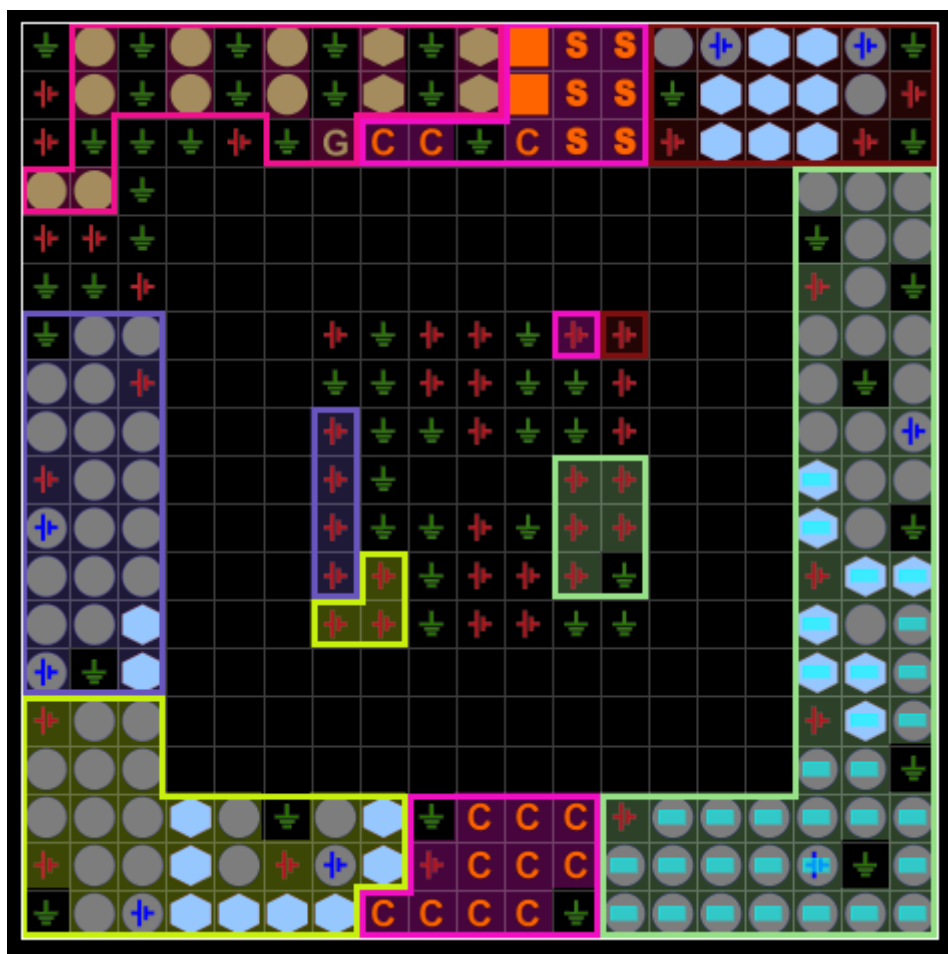


Figure 11 - 8X8 Braun's Multiplier I/O Planning

Section two - 8X8 Wallace Tree Multiplier

Dot Diagram and Logic Circuit Diagram

To write the code for an 8X8 Wallace Multiplier, a dot diagram and logic diagram were produced to reference which sum, carry, etc. is part of each row. The longest path can be seen in green, on the logic diagram (Figure 17), ready for when creating the timing constraints.



Figure 12 - 8X8 Wallace Multiplier Dot Diagram Stage 1

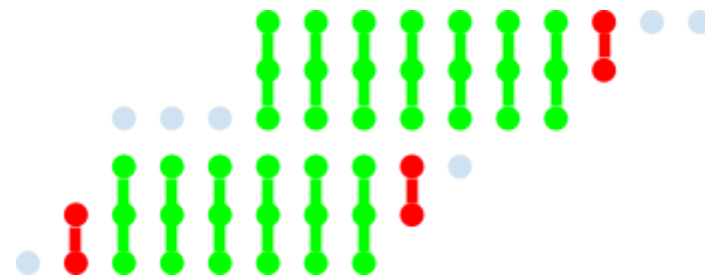


Figure 13 - 8X8 Wallace Multiplier Dot Diagram Stage 2



Figure 14 - 8X8 Wallace Multiplier Dot Diagram Stage 3



Figure 15 - 8X8 Wallace Multiplier Dot Diagram Stage 4



Figure 16 - 8X8 Wallace Multiplier Dot Diagram Final Stage

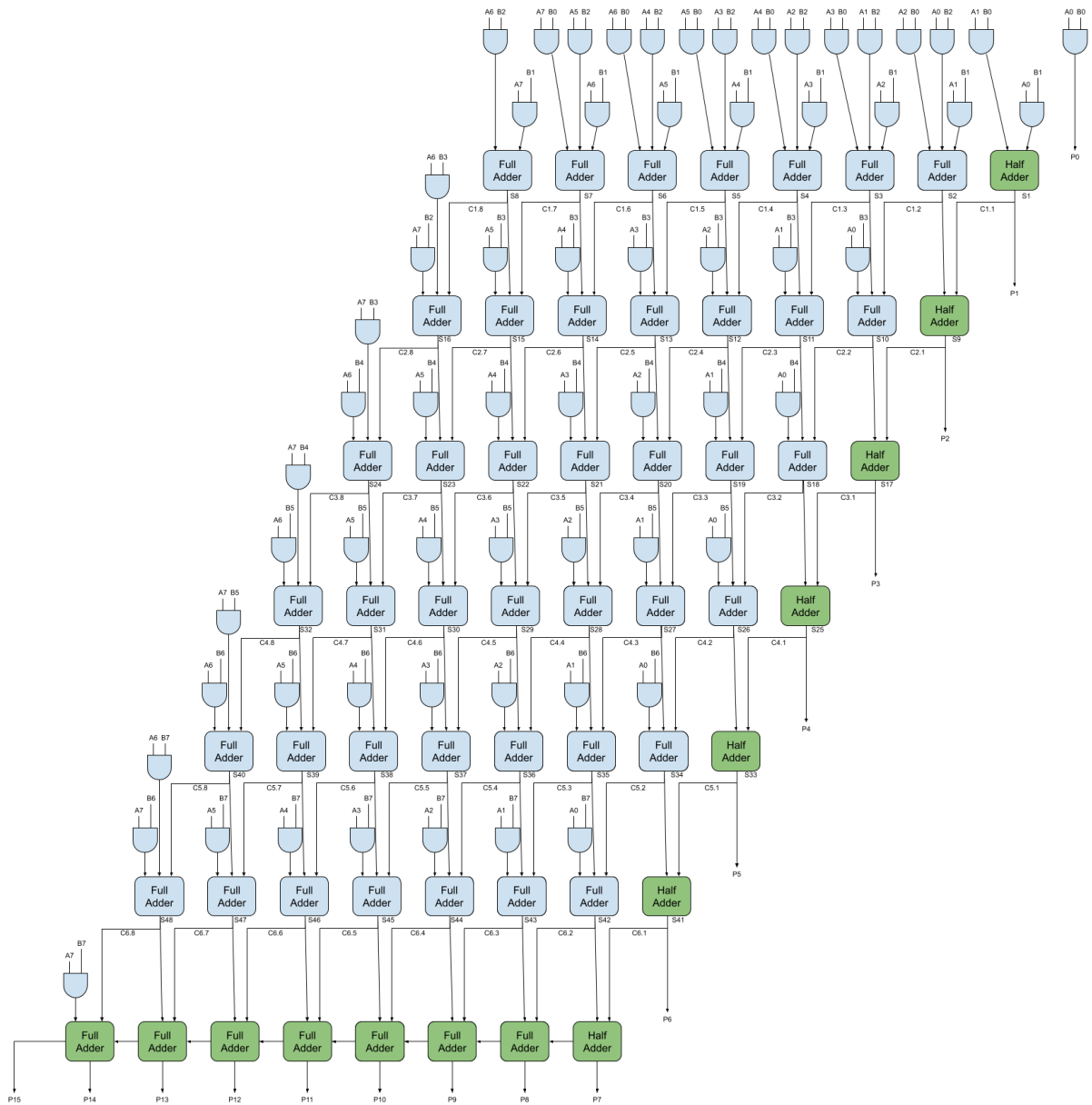


Figure 17 - 8X8 Wallace Multiplier Logic Diagram

Component Declaration and Port Mapping

Similarly to the Baruns, first, the components were declared, describing the interface of the module, i.e. ports A and B as inputs and port P as an output. Afterwards, in the code, Port Mapping was used to connect the individual Full Adder and Half Adder programs to the 8X8 Brauns Program. Using port mapping made the design readable and scalable.

The full program can be found in Appendix 2

Component Declaration

```
entity Wallace_8x8 is
PORT (A : in STD_LOGIC_VECTOR (7 downto 0);
      B : in STD_LOGIC_VECTOR (7 downto 0);
      P : out STD_LOGIC_VECTOR (15 downto 0));

end Wallace_8x8;
```

Port Mapping

```
Component full_adder
port (A : in STD_LOGIC;
      B : in STD_LOGIC;
      Cin : in STD_LOGIC;
      S : out STD_LOGIC;
      Cout : out STD_LOGIC);
end component;

Component half_adder
port (A : in STD_LOGIC;
      B : in STD_LOGIC;
      S : out STD_LOGIC;
      Cout : out STD_LOGIC);
end component;
```

Signal Declaration and Source

After port mapping, the signals were declared for the Carry's and Inputs, then the main program was started, declaring what inputs made up each input signal. However, unlike the Brauns, the signals were declared differently, to save time and space.

Signal Declaration

```
signal A1B0 : STD_LOGIC;  
signal A2B0 : STD_LOGIC;  
signal A3B0 : STD_LOGIC;  
signal A4B0 : STD_LOGIC;  
signal A5B0 : STD_LOGIC;  
signal A6B0 : STD_LOGIC;  
signal A7B0 : STD_LOGIC;  
...  
  
signal S2,S3,S4,S5,S6,S7...S47,S48: std_logic;  
  
signal C1,C2,C3,C4,C5,C6,C7...C54,C55: std_logic;
```

Signal Source

```
A1B0 <= A (1) AND B (0);  
A2B0 <= A (2) AND B (0);  
A3B0 <= A (3) AND B (0);  
A4B0 <= A (4) AND B (0);  
A5B0 <= A (5) AND B (0);  
A6B0 <= A (6) AND B (0);  
A7B0 <= A (7) AND B (0);  
...
```

Partial Product Reduction and Summation

Like the Brauns, the remainder of the code used the port mapping to synthesise the product, starting with P0, the least significant bit.

```
P0 : P(0) <= A(0) and B(0);
```

```
HA11 : half_adder port map (A=>A1B0,B=>A0B1,S=>P(1),Cout=>C1);  
FA12 : full_adder port map (A=>A2B0,B=>A1B1,Cin=>A0B2,S=>S2,Cout=>C2);  
FA13 : full_adder port map (A=>A3B0,B=>A2B1,Cin=>A1B2,S=>S3,Cout=>C3);  
FA14 : full_adder port map (A=>A4B0,B=>A3B1,Cin=>A2B2,S=>S4,Cout=>C4);  
FA15 : full_adder port map (A=>A5B0,B=>A4B1,Cin=>A3B2,S=>S5,Cout=>C5);  
FA16 : full_adder port map (A=>A6B0,B=>A5B1,Cin=>A4B2,S=>S6,Cout=>C6);  
FA17 : full_adder port map (A=>A7B0,B=>A6B1,Cin=>A5B2,S=>S7,Cout=>C7);  
HA18 : half_adder port map (A=>A7B1,B=>A6B2,S=>S8,Cout=>C8);
```

```
...
```

Program Limitations

There are a few limitations and considerations with the Wallace Tree multiplier and the program itself. Firstly, programming it was far more complicated compared to the Brauns due to the irregular reduction stages. However, this is why it has a shorter critical path and so is faster. Due to the multiple layers of half and full adders, the program logic is harder to understand and, as a result, debug, especially if this program were to be scaled up to a higher bit width. Next, just like the Brauns, as all sums, carries and partial product is declared, the program is even more difficult to read. Unlike the Brauns, the Wallace Tree Multiplier can support unsigned multiplication, although in this program, it has not been implemented, to reduce complexity. If it were to be implemented, additional logic would be needed. Finally, this program doesn't use pipelining or any form of clocked logic, meaning the multiplier remains fully combinational. As a result, similarly to the Brauns, the scalability and maximum clock frequency are limited since all operations must settle in a single cycle.

8X8 Wallace Test Results

Before testing, the inputs and outputs needed to be assigned using the I/O list, following the Basys 3 manual. To test that the Wallace Multiplier has been programmed correctly, four simulations were run. First, the clock was forced for both inputs, then a constant was set; in test one, this was ab for A and cd for B. The resultant output was 88ef, indicating that the program is correct; however, to ensure this, three subsequent tests were done.

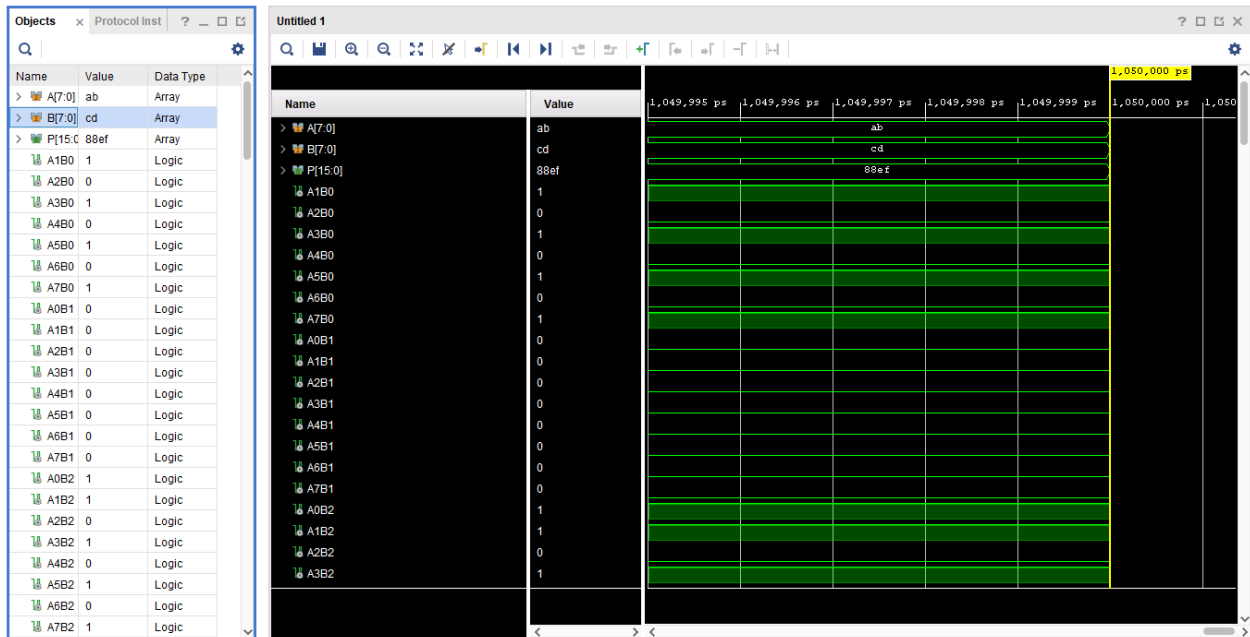


Figure 18 - 8X8 Wallace Multiplier Test One $ab \times cd = 88ef$

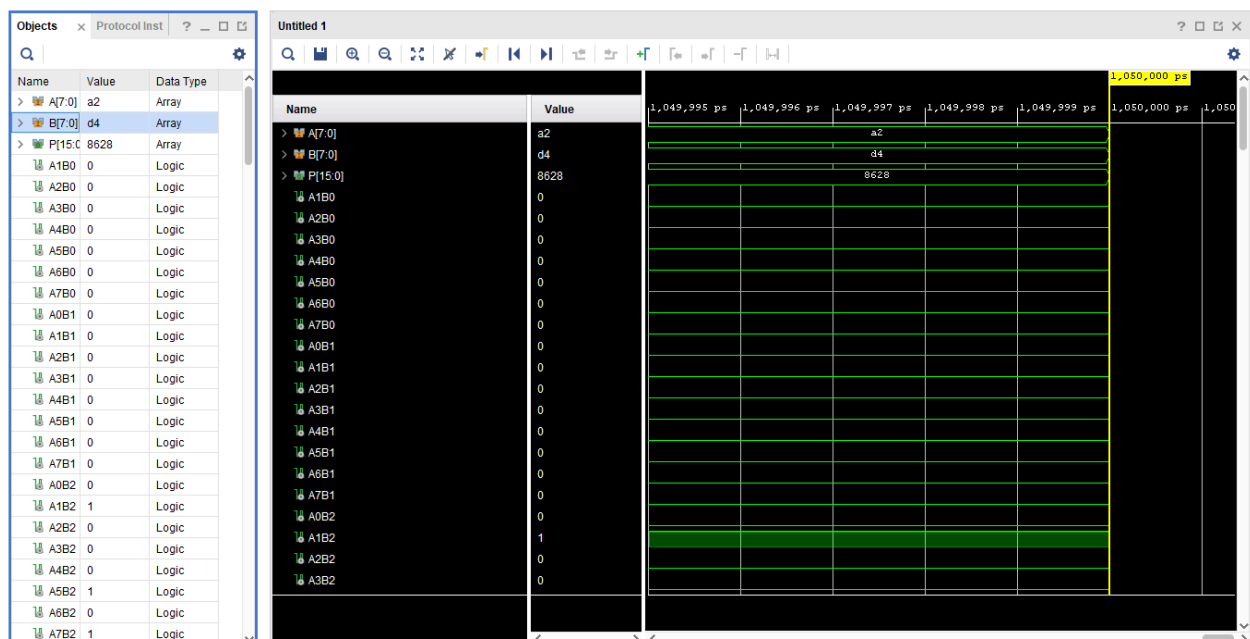


Figure 19 - 8X8 Wallace Multiplier Test Two $a2 \times d4 = 8628$

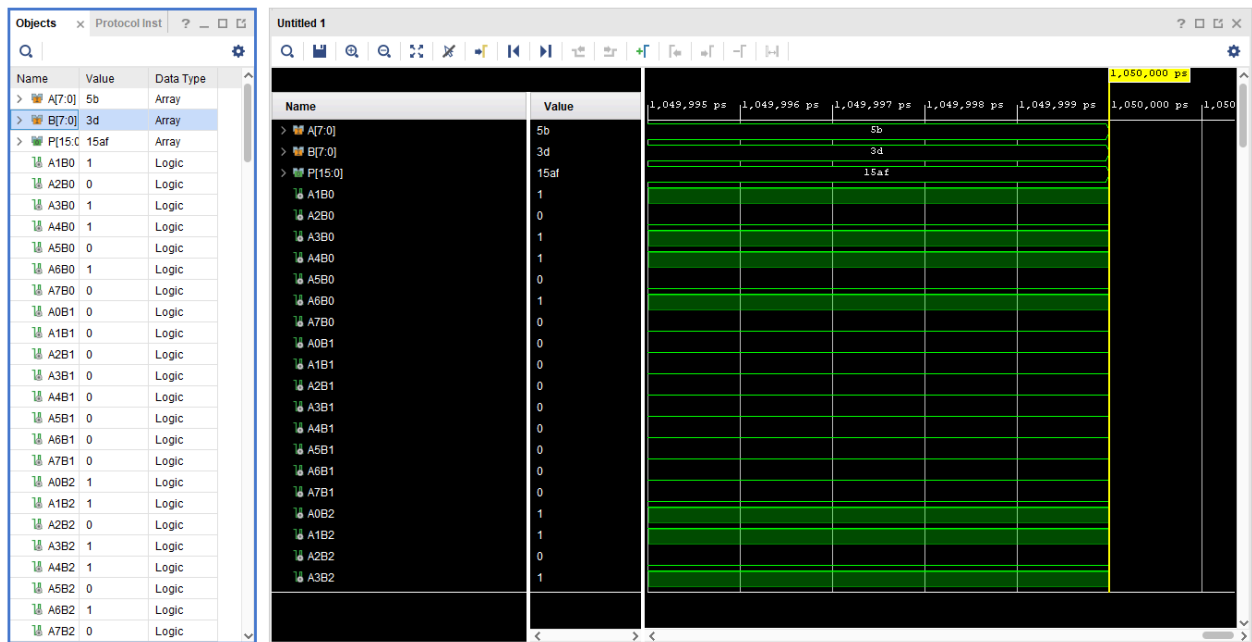


Figure 20 - 8X8 Wallace Multiplier Test Three 5b x 3d = 15af

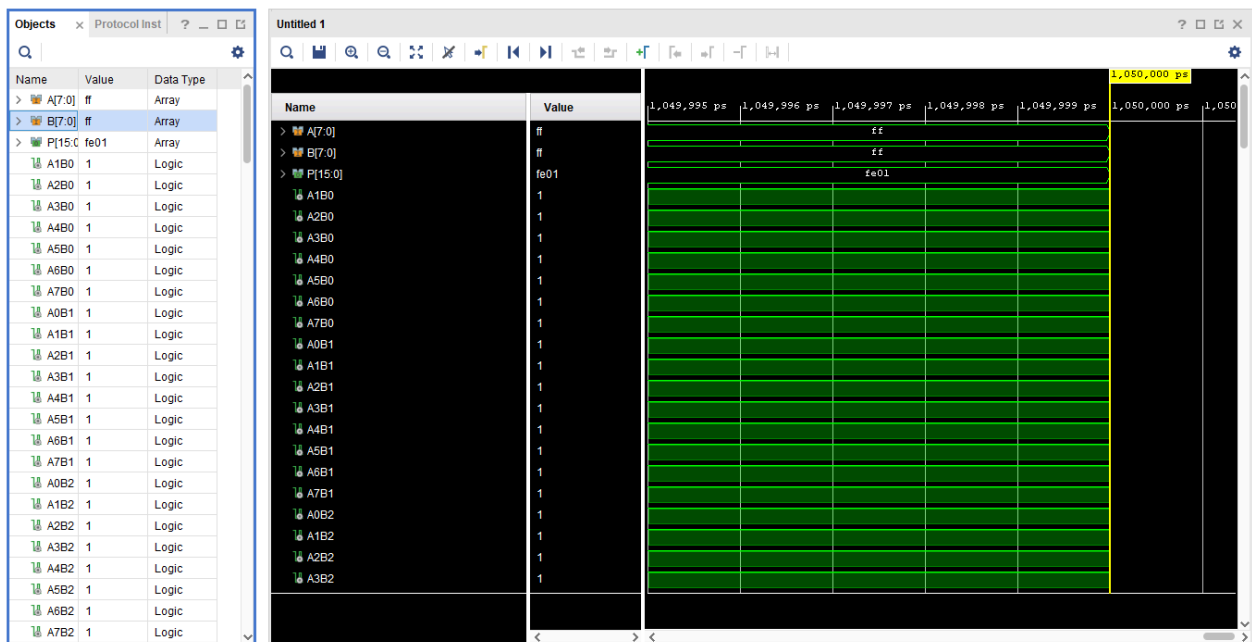


Figure 21 - 8X8 Wallace Multiplier Test Four ff x ff = fe01

8X8 Wallace Synthesis

Having tested the program, Inputs and Outputs were set, so that a Schematic, RTL Design, Device Diagram and I/O Planning Diagram could be produced.

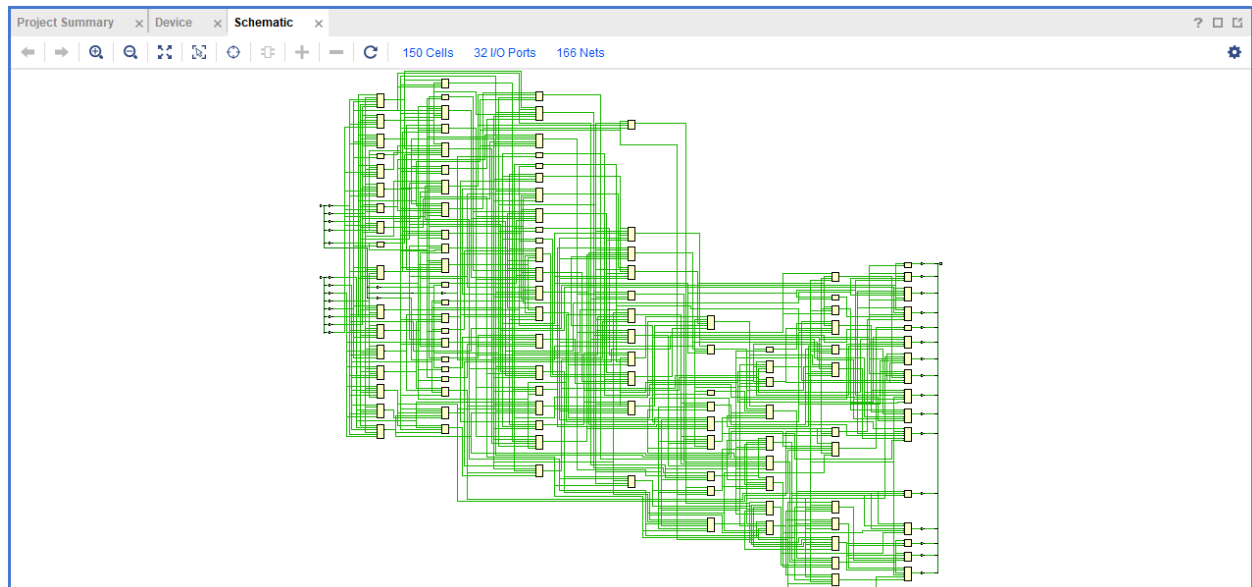


Figure 22 - 8X8 Wallace Multiplier Schematic

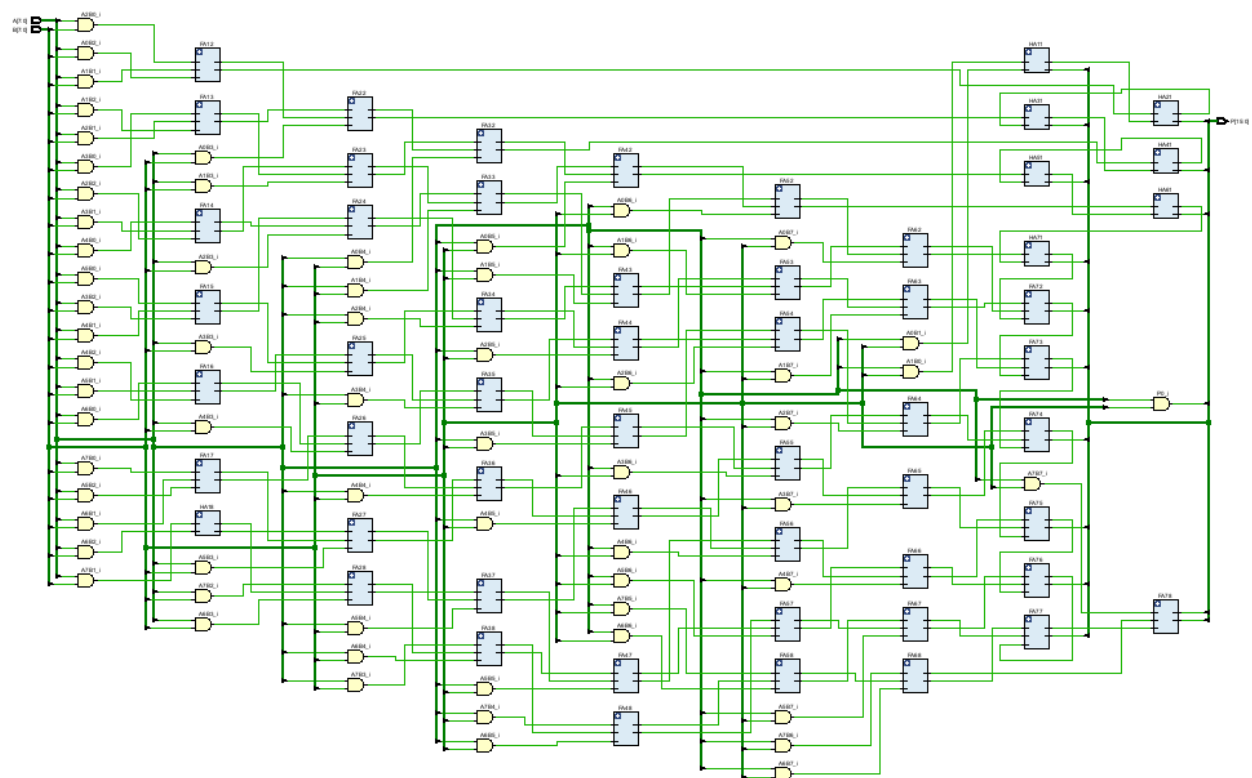


Figure 23 - 8X8 Wallace Multiplier Schematic

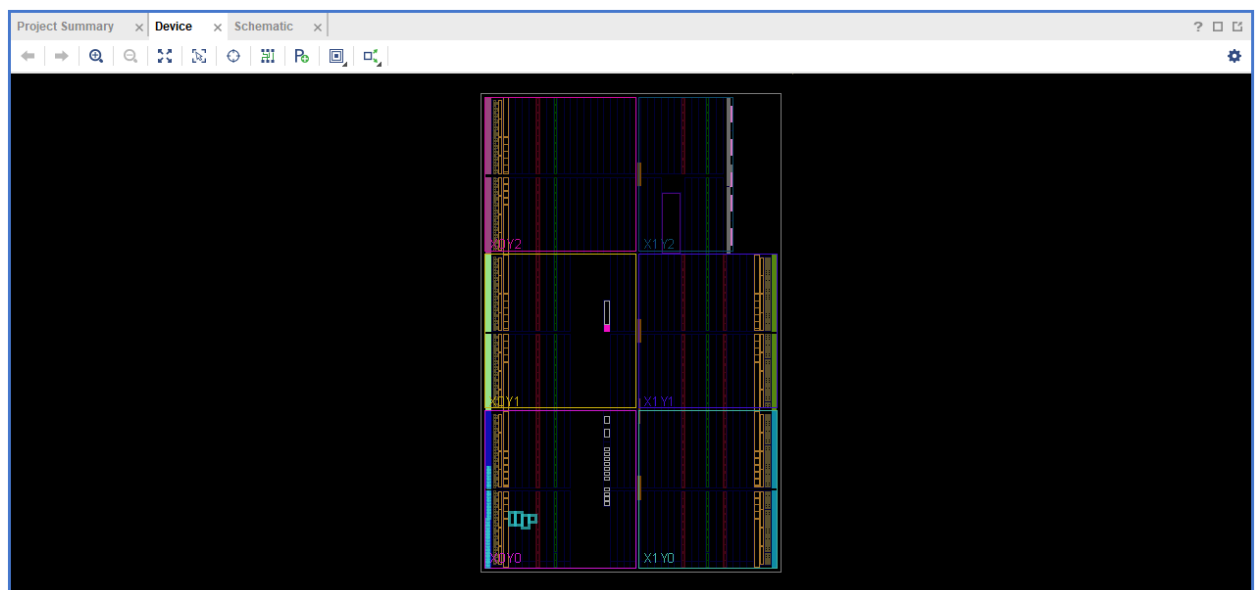


Figure 24 - 8X8 Wallace Multiplier Device Design

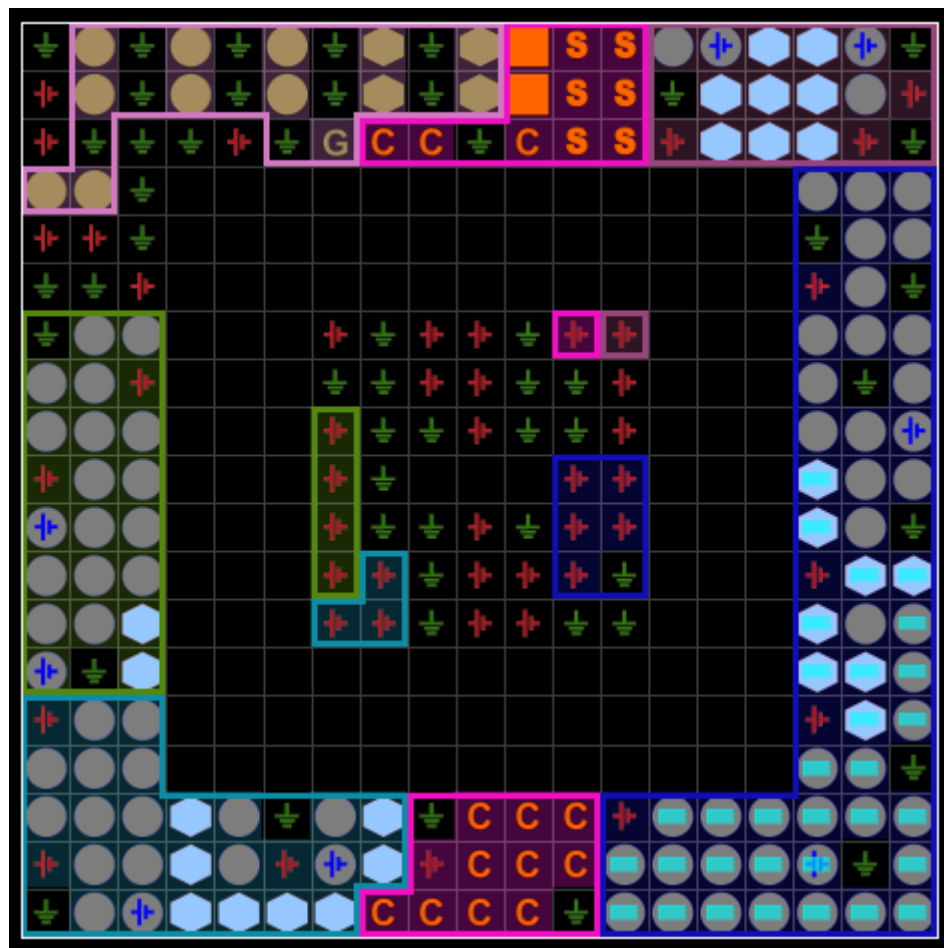


Figure 25 - 8X8 Wallace Multiplier I/O Planning

As a final test, the program was uploaded to the Basys 3 FPGA Board again using Vivado to generate a bitstream and upload the program.

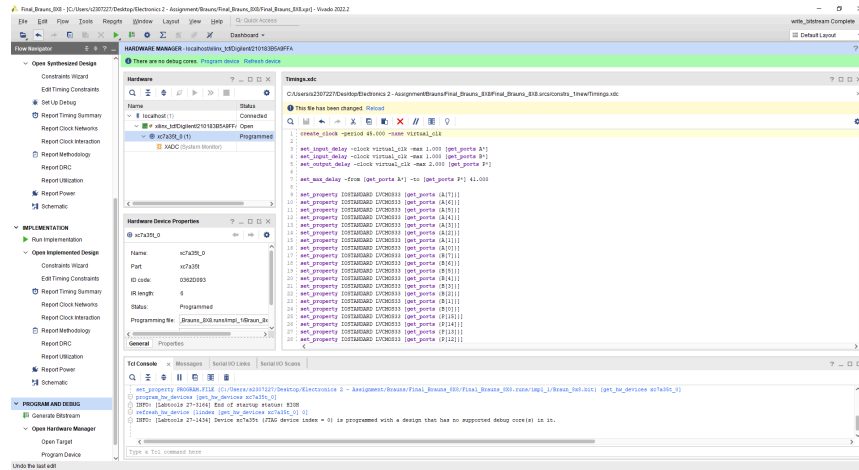


Figure 51 - Brauns Hardware Testing One

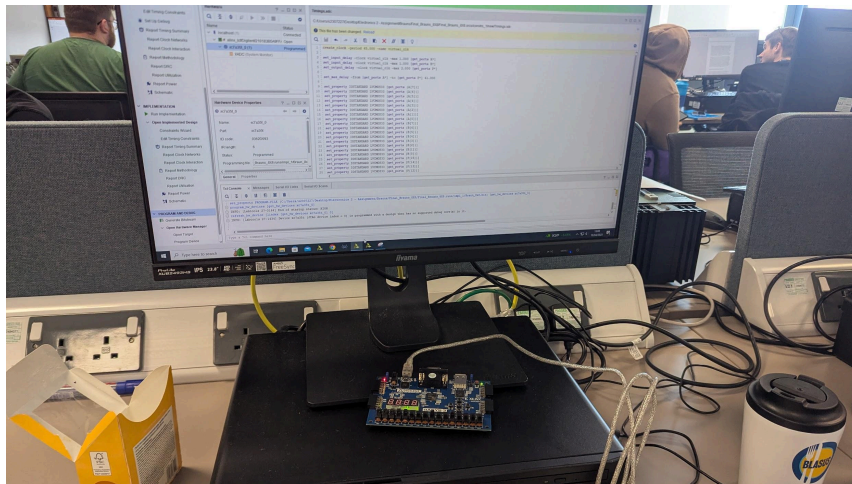


Figure 52 - Brauns Hardware Testing Two

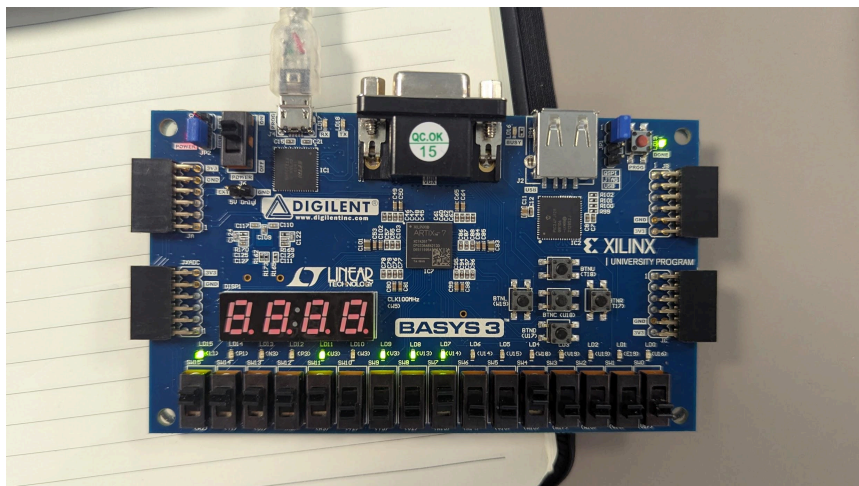


Figure 53 - Brauns Hardware Testing Three

Wallace Tree Testing on Hardware

As a final test, the program was uploaded to the Basys 3 FPGA Board again using Vivado to generate a bitstream and upload the program.

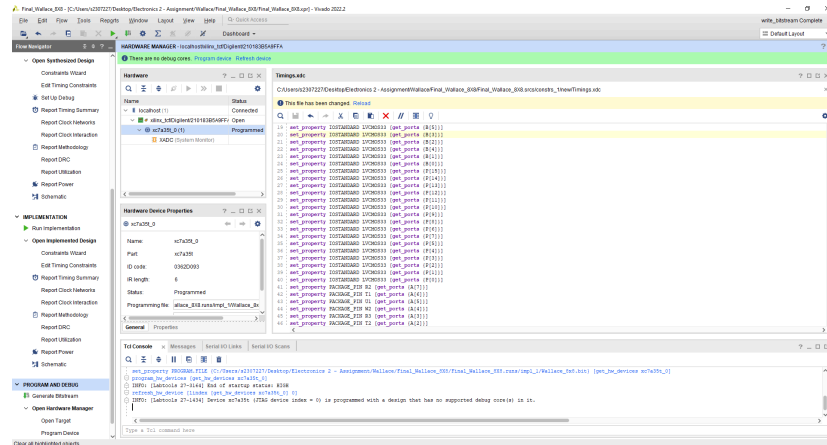


Figure 54 - Wallace Hardware Testing One

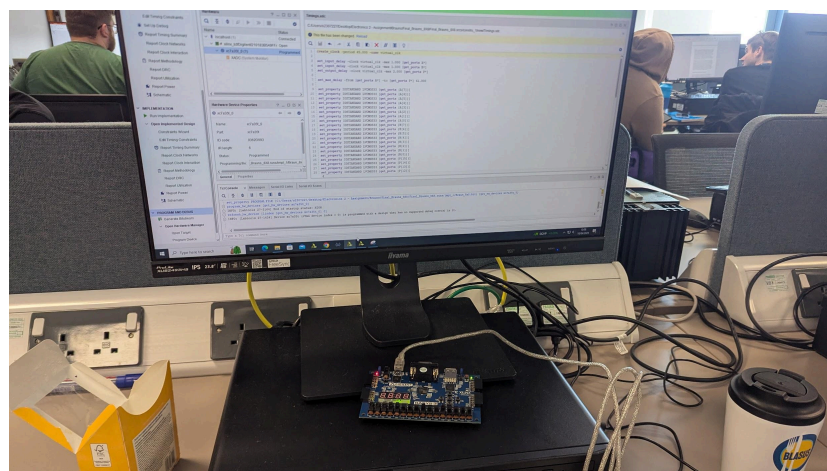


Figure 55 - Wallace Hardware Testing Two

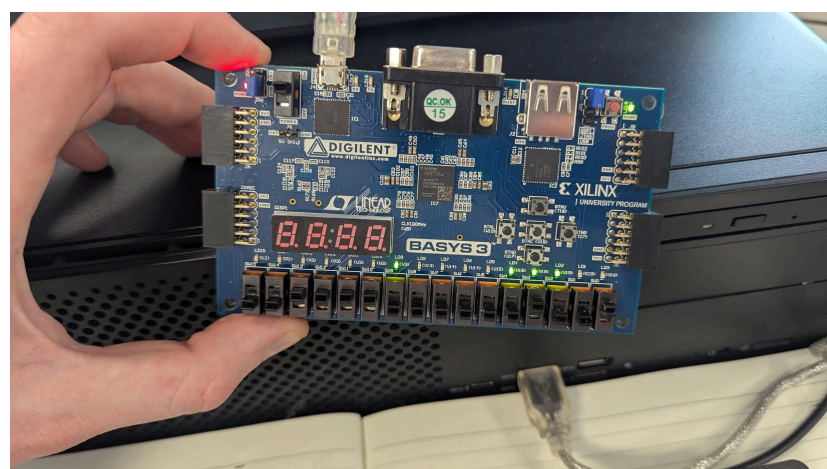


Figure 56 - Wallace Hardware Testing Three

Timing Constants

Before the performance of both multipliers could be done using the Vivado Software, timing constraints needed to be set for both the Brauns and the Wallace. To do this, the longest delay path was needed, see Figure 3 and Figure 17, to calculate the longest delay. By adding the delay of the first AND Gate (1ns) to the delay of the following Adders (2ns each).

Brauns: 1ns + (20 x 2ns) = 41ns

<pre>create_clock -period 45.000 -name virtual_clk set_input_delay -clock virtual_clk -max 1.000 [get_ports A*] set_input_delay -clock virtual_clk -max 1.000 [get_ports B*] set_output_delay -clock virtual_clk -max 2.000 [get_ports P*] set_max_delay -from [get_ports A*] -to [get_ports P*] 41.000 set_property IOSTANDARD LVCMOS33 [get_ports {A[7]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[6]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[5]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[4]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[3]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[2]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[1]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[0]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[7]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[6]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[5]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[4]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[3]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[2]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[1]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[0]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[15]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[14]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[13]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[12]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[11]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[10]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[9]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[8]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[7]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[6]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[5]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[4]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[3]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[2]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[1]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[0]}]</pre>	<pre>set_property PACKAGE_PIN R2 [get_ports {A[7]}] set_property PACKAGE_PIN T1 [get_ports {A[6]}] set_property PACKAGE_PIN W2 [get_ports {A[4]}] set_property PACKAGE_PIN U1 [get_ports {A[5]}] set_property PACKAGE_PIN R3 [get_ports {A[3]}] set_property PACKAGE_PIN T2 [get_ports {A[2]}] set_property PACKAGE_PIN T3 [get_ports {A[1]}] set_property PACKAGE_PIN V2 [get_ports {A[0]}] set_property PACKAGE_PIN W13 [get_ports {B[7]}] set_property PACKAGE_PIN W14 [get_ports {B[6]}] set_property PACKAGE_PIN V15 [get_ports {B[5]}] set_property PACKAGE_PIN W15 [get_ports {B[4]}] set_property PACKAGE_PIN W17 [get_ports {B[3]}] set_property PACKAGE_PIN W16 [get_ports {B[2]}] set_property PACKAGE_PIN V16 [get_ports {B[1]}] set_property PACKAGE_PIN V17 [get_ports {B[0]}] set_property PACKAGE_PIN L1 [get_ports {P[15]}] set_property PACKAGE_PIN P1 [get_ports {P[14]}] set_property PACKAGE_PIN N3 [get_ports {P[13]}] set_property PACKAGE_PIN P3 [get_ports {P[12]}] set_property PACKAGE_PIN U3 [get_ports {P[11]}] set_property PACKAGE_PIN W3 [get_ports {P[10]}] set_property PACKAGE_PIN V3 [get_ports {P[9]}] set_property PACKAGE_PIN V13 [get_ports {P[8]}] set_property PACKAGE_PIN U14 [get_ports {P[6]}] set_property PACKAGE_PIN W18 [get_ports {P[4]}] set_property PACKAGE_PIN V19 [get_ports {P[3]}] set_property PACKAGE_PIN U19 [get_ports {P[2]}] set_property PACKAGE_PIN V14 [get_ports {P[7]}] set_property PACKAGE_PIN U15 [get_ports {P[5]}] set_property PACKAGE_PIN E19 [get_ports {P[1]}] set_property PACKAGE_PIN U16 [get_ports {P[0]}]</pre>
--	---

Figure 26 - 8X8 Brauns Timing Constraints

Wallace: 1ns + (14 x 2ns) = 29ns

<pre> create_clock -period 45.000 -name virtual_clk set_input_delay -clock virtual_clk -max 1.000 [get_ports A*] set_input_delay -clock virtual_clk -max 1.000 [get_ports B*] set_output_delay -clock virtual_clk -max 2.000 [get_ports P*] set_max_delay -from [get_ports A*] -to [get_ports P*] 29.000 set_property IOSTANDARD LVCMOS33 [get_ports {A[7]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[6]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[5]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[4]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[3]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[2]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[1]}] set_property IOSTANDARD LVCMOS33 [get_ports {A[0]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[7]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[6]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[5]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[3]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[2]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[4]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[1]}] set_property IOSTANDARD LVCMOS33 [get_ports {B[0]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[15]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[14]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[13]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[12]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[11]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[10]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[9]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[8]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[6]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[7]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[5]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[4]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[3]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[2]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[1]}] set_property IOSTANDARD LVCMOS33 [get_ports {P[0]}] </pre>	<pre> set_property PACKAGE_PIN R2 [get_ports {A[7]}] set_property PACKAGE_PIN T1 [get_ports {A[6]}] set_property PACKAGE_PIN U1 [get_ports {A[5]}] set_property PACKAGE_PIN W2 [get_ports {A[4]}] set_property PACKAGE_PIN R3 [get_ports {A[3]}] set_property PACKAGE_PIN T2 [get_ports {A[2]}] set_property PACKAGE_PIN T3 [get_ports {A[1]}] set_property PACKAGE_PIN V2 [get_ports {A[0]}] set_property PACKAGE_PIN W13 [get_ports {B[7]}] set_property PACKAGE_PIN W14 [get_ports {B[6]}] set_property PACKAGE_PIN V15 [get_ports {B[5]}] set_property PACKAGE_PIN W15 [get_ports {B[4]}] set_property PACKAGE_PIN W17 [get_ports {B[3]}] set_property PACKAGE_PIN W16 [get_ports {B[2]}] set_property PACKAGE_PIN V16 [get_ports {B[1]}] set_property PACKAGE_PIN V17 [get_ports {B[0]}] set_property PACKAGE_PIN L1 [get_ports {P[15]}] set_property PACKAGE_PIN P1 [get_ports {P[14]}] set_property PACKAGE_PIN N3 [get_ports {P[13]}] set_property PACKAGE_PIN P3 [get_ports {P[12]}] set_property PACKAGE_PIN U3 [get_ports {P[11]}] set_property PACKAGE_PIN W3 [get_ports {P[10]}] set_property PACKAGE_PIN V3 [get_ports {P[9]}] set_property PACKAGE_PIN V13 [get_ports {P[8]}] set_property PACKAGE_PIN V14 [get_ports {P[7]}] set_property PACKAGE_PIN U14 [get_ports {P[6]}] set_property PACKAGE_PIN U15 [get_ports {P[5]}] set_property PACKAGE_PIN W18 [get_ports {P[4]}] set_property PACKAGE_PIN V19 [get_ports {P[3]}] set_property PACKAGE_PIN U19 [get_ports {P[2]}] set_property PACKAGE_PIN E19 [get_ports {P[1]}] set_property PACKAGE_PIN U16 [get_ports {P[0]}] </pre>
--	---

Figure 27 - 8X8 Wallace Timing Constraints

Braun's Timing Delay Evidence

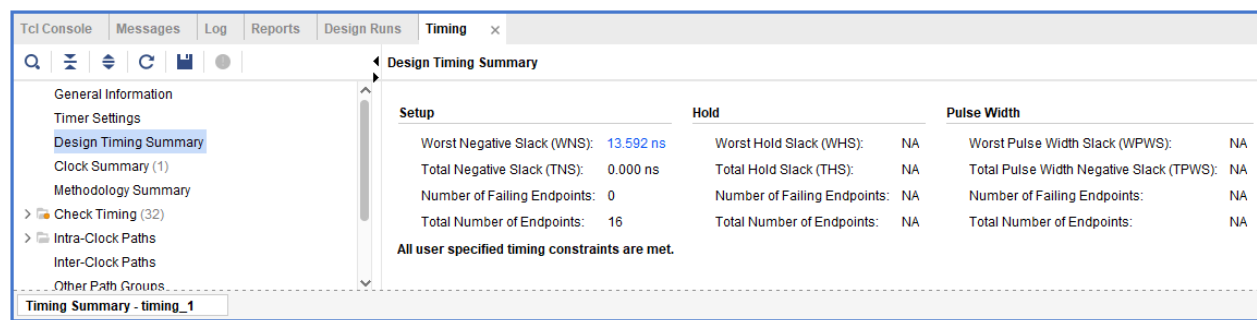


Figure 28 - 8X8 Brauns Timing Results Overview

From Port	To Port	Max Delay	Max Process Corner
A[0]	P[0]	11.905	SLOW
A[0]	P[1]	11.419	SLOW
A[0]	P[2]	11.247	SLOW
A[0]	P[3]	11.307	SLOW
A[0]	P[4]	13.708	SLOW
A[0]	P[5]	14.659	SLOW
A[0]	P[6]	16.385	SLOW
A[0]	P[7]	16.392	SLOW
A[0]	P[8]	17.843	SLOW
A[0]	P[9]	19.962	SLOW
A[0]	P[10]	20.134	SLOW
A[0]	P[11]	21.285	SLOW
A[0]	P[12]	22.252	SLOW
A[0]	P[13]	22.678	SLOW
A[0]	P[14]	23.127	SLOW
A[0]	P[15]	23.678	SLOW
A[1]	P[1]	11.516	SLOW
A[1]	P[2]	11.411	SLOW
A[1]	P[3]	11.950	SLOW
A[1]	P[4]	14.111	SLOW
A[1]	P[5]	15.302	SLOW
A[1]	P[6]	16.755	SLOW
A[1]	P[7]	17.035	SLOW
A[1]	P[8]	18.486	SLOW
A[1]	P[9]	20.333	SLOW
A[1]	P[10]	20.505	SLOW
A[1]	P[11]	21.656	SLOW
A[1]	P[12]	22.622	SLOW
A[1]	P[13]	23.048	SLOW
A[1]	P[14]	23.497	SLOW
A[1]	P[15]	24.048	SLOW

Figure 29 - 8X8 Brauns Timing Results 1

A[2]	P[2]	10.787	SLOW
A[2]	P[3]	11.519	SLOW
A[2]	P[4]	13.598	SLOW
A[2]	P[5]	14.871	SLOW
A[2]	P[6]	16.314	SLOW
A[2]	P[7]	16.604	SLOW
A[2]	P[8]	18.055	SLOW
A[2]	P[9]	19.853	SLOW
A[2]	P[10]	20.025	SLOW
A[2]	P[11]	21.176	SLOW
A[2]	P[12]	22.142	SLOW
A[2]	P[13]	22.568	SLOW
A[2]	P[14]	23.017	SLOW
A[2]	P[15]	23.568	SLOW
A[3]	P[3]	11.163	SLOW
A[3]	P[4]	14.413	SLOW
A[3]	P[5]	14.912	SLOW
A[3]	P[6]	17.090	SLOW
A[3]	P[7]	17.057	SLOW
A[3]	P[8]	18.537	SLOW
A[3]	P[9]	20.667	SLOW
A[3]	P[10]	20.839	SLOW
A[3]	P[11]	21.990	SLOW
A[3]	P[12]	22.956	SLOW
A[3]	P[13]	23.383	SLOW
A[3]	P[14]	23.832	SLOW
A[3]	P[15]	24.383	SLOW
A[4]	P[4]	13.505	SLOW
A[4]	P[5]	14.825	SLOW
A[4]	P[6]	16.268	SLOW
A[4]	P[7]	16.558	SLOW
A[4]	P[8]	18.009	SLOW

Figure 30 - 8X8 Brauns Timing Results 2

A[4]	P[9]	20.126	SLOW
A[4]	P[10]	20.298	SLOW
A[4]	P[11]	21.449	SLOW
A[4]	P[12]	22.415	SLOW
A[4]	P[13]	22.841	SLOW
A[4]	P[14]	23.290	SLOW
A[4]	P[15]	23.841	SLOW
A[5]	P[5]	13.070	SLOW
A[5]	P[6]	15.482	SLOW
A[5]	P[7]	15.450	SLOW
A[5]	P[8]	16.978	SLOW
A[5]	P[9]	19.164	SLOW
A[5]	P[10]	19.336	SLOW
A[5]	P[11]	20.487	SLOW
A[5]	P[12]	21.453	SLOW
A[5]	P[13]	21.879	SLOW
A[5]	P[14]	22.328	SLOW
A[5]	P[15]	22.879	SLOW
A[6]	P[6]	15.203	SLOW
A[6]	P[7]	15.156	SLOW
A[6]	P[8]	16.951	SLOW
A[6]	P[9]	19.137	SLOW
A[6]	P[10]	19.309	SLOW
A[6]	P[11]	20.460	SLOW
A[6]	P[12]	21.426	SLOW
A[6]	P[13]	21.853	SLOW
A[6]	P[14]	22.301	SLOW
A[6]	P[15]	22.853	SLOW
A[7]	P[7]	14.324	SLOW
A[7]	P[8]	16.162	SLOW
A[7]	P[9]	18.306	SLOW
A[7]	P[10]	18.478	SLOW
A[7]	P[11]	19.629	SLOW

Figure 31 - 8X8 Brauns Timing Results 3

✓ A[7]	✓ P[12]	20.595	SLOW
✓ A[7]	✓ P[13]	21.021	SLOW
✓ A[7]	✓ P[14]	21.470	SLOW
✓ A[7]	✓ P[15]	22.021	SLOW
✓ B[0]	✓ P[0]	9.629	SLOW
✓ B[0]	✓ P[1]	9.139	SLOW
✓ B[0]	✓ P[2]	9.154	SLOW
✓ B[0]	✓ P[3]	10.151	SLOW
✓ B[0]	✓ P[4]	12.979	SLOW
✓ B[0]	✓ P[5]	13.687	SLOW
✓ B[0]	✓ P[6]	15.656	SLOW
✓ B[0]	✓ P[7]	15.623	SLOW
✓ B[0]	✓ P[8]	17.103	SLOW
✓ B[0]	✓ P[9]	19.233	SLOW
✓ B[0]	✓ P[10]	19.406	SLOW
✓ B[0]	✓ P[11]	20.557	SLOW
✓ B[0]	✓ P[12]	21.523	SLOW
✓ B[0]	✓ P[13]	21.949	SLOW
✓ B[0]	✓ P[14]	22.398	SLOW
✓ B[0]	✓ P[15]	22.949	SLOW
✓ B[1]	✓ P[1]	9.074	SLOW
✓ B[1]	✓ P[2]	8.765	SLOW
✓ B[1]	✓ P[3]	9.815	SLOW
✓ B[1]	✓ P[4]	12.738	SLOW
✓ B[1]	✓ P[5]	13.351	SLOW
✓ B[1]	✓ P[6]	15.414	SLOW
✓ B[1]	✓ P[7]	15.382	SLOW
✓ B[1]	✓ P[8]	16.965	SLOW
✓ B[1]	✓ P[9]	19.151	SLOW
✓ B[1]	✓ P[10]	19.323	SLOW
✓ B[1]	✓ P[11]	20.474	SLOW
✓ B[1]	✓ P[12]	21.440	SLOW

Figure 32 - 8X8 Brauns
Timing Results 4

✓ B[1]	✓ P[13]	21.866	SLOW
✓ B[1]	✓ P[14]	22.315	SLOW
✓ B[1]	✓ P[15]	22.867	SLOW
✓ B[2]	✓ P[2]	8.650	SLOW
✓ B[2]	✓ P[3]	10.503	SLOW
✓ B[2]	✓ P[4]	12.241	SLOW
✓ B[2]	✓ P[5]	13.855	SLOW
✓ B[2]	✓ P[6]	15.298	SLOW
✓ B[2]	✓ P[7]	15.588	SLOW
✓ B[2]	✓ P[8]	17.039	SLOW
✓ B[2]	✓ P[9]	18.536	SLOW
✓ B[2]	✓ P[10]	18.708	SLOW
✓ B[2]	✓ P[11]	19.859	SLOW
✓ B[2]	✓ P[12]	20.825	SLOW
✓ B[2]	✓ P[13]	21.251	SLOW
✓ B[2]	✓ P[14]	21.700	SLOW
✓ B[2]	✓ P[15]	22.251	SLOW
✓ B[3]	✓ P[3]	9.035	SLOW
✓ B[3]	✓ P[4]	11.177	SLOW
✓ B[3]	✓ P[5]	12.030	SLOW
✓ B[3]	✓ P[6]	13.795	SLOW
✓ B[3]	✓ P[7]	13.948	SLOW
✓ B[3]	✓ P[8]	15.399	SLOW
✓ B[3]	✓ P[9]	17.427	SLOW
✓ B[3]	✓ P[10]	17.599	SLOW
✓ B[3]	✓ P[11]	18.750	SLOW
✓ B[3]	✓ P[12]	19.716	SLOW
✓ B[3]	✓ P[13]	20.143	SLOW
✓ B[3]	✓ P[14]	20.591	SLOW
✓ B[3]	✓ P[15]	21.143	SLOW
✓ B[4]	✓ P[4]	9.680	SLOW
✓ B[4]	✓ P[5]	11.078	SLOW
✓ B[4]	✓ P[6]	13.801	SLOW

Figure 33 - 8X8 Brauns
Timing Results 5

✓ B[4]	✓ P[7]	14.092	SLOW
✓ B[4]	✓ P[8]	15.543	SLOW
✓ B[4]	✓ P[9]	16.968	SLOW
✓ B[4]	✓ P[10]	16.902	SLOW
✓ B[4]	✓ P[11]	18.053	SLOW
✓ B[4]	✓ P[12]	19.019	SLOW
✓ B[4]	✓ P[13]	19.445	SLOW
✓ B[4]	✓ P[14]	19.894	SLOW
✓ B[4]	✓ P[15]	20.445	SLOW
✓ B[5]	✓ P[5]	9.931	SLOW
✓ B[5]	✓ P[6]	12.874	SLOW
✓ B[5]	✓ P[7]	13.165	SLOW
✓ B[5]	✓ P[8]	14.616	SLOW
✓ B[5]	✓ P[9]	16.040	SLOW
✓ B[5]	✓ P[10]	15.975	SLOW
✓ B[5]	✓ P[11]	17.126	SLOW
✓ B[5]	✓ P[12]	18.092	SLOW
✓ B[5]	✓ P[13]	18.518	SLOW
✓ B[5]	✓ P[14]	18.967	SLOW
✓ B[5]	✓ P[15]	19.518	SLOW
✓ B[6]	✓ P[6]	9.777	SLOW
✓ B[6]	✓ P[7]	13.307	SLOW
✓ B[6]	✓ P[8]	15.148	SLOW
✓ B[6]	✓ P[9]	16.743	SLOW
✓ B[6]	✓ P[10]	16.754	SLOW
✓ B[6]	✓ P[11]	17.903	SLOW
✓ B[6]	✓ P[12]	18.869	SLOW
✓ B[6]	✓ P[13]	19.296	SLOW
✓ B[6]	✓ P[14]	19.745	SLOW
✓ B[6]	✓ P[15]	20.296	SLOW
✓ B[7]	✓ P[7]	9.769	SLOW
✓ B[7]	✓ P[8]	11.403	SLOW
✓ B[7]	✓ P[9]	12.828	SLOW

Figure 34 - 8X8 Brauns
Timing Results 6

✓ B[7]	✓ P[10]	12.700	SLOW
✓ B[7]	✓ P[11]	13.773	SLOW
✓ B[7]	✓ P[12]	14.808	SLOW
✓ B[7]	✓ P[13]	14.881	SLOW
✓ B[7]	✓ P[14]	15.325	SLOW
✓ B[7]	✓ P[15]	15.870	SLOW

Figure 35 - 8X8 Brauns Timing Results 7

Wallace Tree Timing Delay Evidence

Timing			
Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 6.090 ns	Worst Hold Slack (WHS): NA	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): NA	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints: 0	Number of Failing Endpoints: NA	Number of Failing Endpoints:	NA
Total Number of Endpoints: 16	Total Number of Endpoints: NA	Total Number of Endpoints:	NA
All user specified timing constraints are met.			

Figure 36 - 8X8 Wallace Timing Results Overview

From Port	To Port	Max Delay	Max Process Corner
A[0]	P[0]	11.974	SLOW
A[0]	P[1]	11.501	SLOW
A[0]	P[2]	10.481	SLOW
A[0]	P[3]	12.582	SLOW
A[0]	P[4]	13.089	SLOW
A[0]	P[5]	13.345	SLOW
A[0]	P[6]	14.806	SLOW
A[0]	P[7]	14.394	SLOW
A[0]	P[8]	15.633	SLOW
A[0]	P[9]	16.369	SLOW
A[0]	P[10]	16.086	SLOW
A[0]	P[11]	15.986	SLOW
A[0]	P[12]	18.126	SLOW
A[0]	P[13]	18.525	SLOW
A[0]	P[14]	18.159	SLOW
A[0]	P[15]	18.599	SLOW
A[1]	P[1]	10.809	SLOW
A[1]	P[2]	10.661	SLOW
A[1]	P[3]	13.044	SLOW
A[1]	P[4]	13.482	SLOW
A[1]	P[5]	14.002	SLOW
A[1]	P[6]	15.494	SLOW
A[1]	P[7]	15.523	SLOW
A[1]	P[8]	16.320	SLOW
A[1]	P[9]	17.056	SLOW
A[1]	P[10]	16.992	SLOW
A[1]	P[11]	16.892	SLOW
A[1]	P[12]	19.031	SLOW
A[1]	P[13]	19.431	SLOW
A[1]	P[14]	19.065	SLOW
A[1]	P[15]	19.505	SLOW
A[2]	P[2]	11.096	SLOW
A[2]	P[3]	12.836	SLOW

Figure 37 - 8X8 Wallace Timing Results 1

A[2]	P[4]	13.862	SLOW
A[2]	P[5]	14.381	SLOW
A[2]	P[6]	15.873	SLOW
A[2]	P[7]	15.902	SLOW
A[2]	P[8]	16.700	SLOW
A[2]	P[9]	17.436	SLOW
A[2]	P[10]	17.372	SLOW
A[2]	P[11]	17.272	SLOW
A[2]	P[12]	19.411	SLOW
A[2]	P[13]	19.811	SLOW
A[2]	P[14]	19.445	SLOW
A[2]	P[15]	19.885	SLOW
A[3]	P[3]	12.489	SLOW
A[3]	P[4]	13.373	SLOW
A[3]	P[5]	13.927	SLOW
A[3]	P[6]	15.419	SLOW
A[3]	P[7]	15.635	SLOW
A[3]	P[8]	16.245	SLOW
A[3]	P[9]	16.981	SLOW
A[3]	P[10]	16.917	SLOW
A[3]	P[11]	16.817	SLOW
A[3]	P[12]	18.957	SLOW
A[3]	P[13]	19.356	SLOW
A[3]	P[14]	18.990	SLOW
A[3]	P[15]	19.430	SLOW
A[4]	P[4]	12.640	SLOW
A[4]	P[5]	13.193	SLOW
A[4]	P[6]	14.685	SLOW
A[4]	P[7]	14.991	SLOW
A[4]	P[8]	15.512	SLOW
A[4]	P[9]	16.357	SLOW
A[4]	P[10]	16.444	SLOW
A[4]	P[11]	16.344	SLOW

Figure 38 - 8X8 Wallace Timing Results 2

A[4]	P[12]	18.483	SLOW
A[4]	P[13]	18.883	SLOW
A[4]	P[14]	18.517	SLOW
A[4]	P[15]	18.957	SLOW
A[5]	P[5]	13.140	SLOW
A[5]	P[6]	14.724	SLOW
A[5]	P[7]	15.260	SLOW
A[5]	P[8]	15.604	SLOW
A[5]	P[9]	16.993	SLOW
A[5]	P[10]	16.713	SLOW
A[5]	P[11]	16.613	SLOW
A[5]	P[12]	18.753	SLOW
A[5]	P[13]	19.152	SLOW
A[5]	P[14]	18.786	SLOW
A[5]	P[15]	19.226	SLOW
A[6]	P[6]	13.986	SLOW
A[6]	P[7]	14.826	SLOW
A[6]	P[8]	15.138	SLOW
A[6]	P[9]	16.934	SLOW
A[6]	P[10]	16.380	SLOW
A[6]	P[11]	16.192	SLOW
A[6]	P[12]	18.332	SLOW
A[6]	P[13]	18.731	SLOW
A[6]	P[14]	18.365	SLOW
A[6]	P[15]	18.805	SLOW
A[7]	P[7]	14.113	SLOW
A[7]	P[8]	14.424	SLOW
A[7]	P[9]	17.558	SLOW
A[7]	P[10]	17.004	SLOW
A[7]	P[11]	16.516	SLOW
A[7]	P[12]	18.655	SLOW
A[7]	P[13]	19.055	SLOW
A[7]	P[14]	18.689	SLOW

Figure 39 - 8X8 Wallace Timing Results 3

 A[7]	 P[15]	19.129	SLOW
 B[0]	 P[0]	9.529	SLOW
 B[0]	 P[1]	9.057	SLOW
 B[0]	 P[2]	9.006	SLOW
 B[0]	 P[3]	11.753	SLOW
 B[0]	 P[4]	12.446	SLOW
 B[0]	 P[5]	13.000	SLOW
 B[0]	 P[6]	14.492	SLOW
 B[0]	 P[7]	14.966	SLOW
 B[0]	 P[8]	15.319	SLOW
 B[0]	 P[9]	16.299	SLOW
 B[0]	 P[10]	16.145	SLOW
 B[0]	 P[11]	16.045	SLOW
 B[0]	 P[12]	18.185	SLOW
 B[0]	 P[13]	18.584	SLOW
 B[0]	 P[14]	18.218	SLOW
 B[0]	 P[15]	18.658	SLOW
 B[1]	 P[1]	9.227	SLOW
 B[1]	 P[2]	8.769	SLOW
 B[1]	 P[3]	11.198	SLOW
 B[1]	 P[4]	12.507	SLOW
 B[1]	 P[5]	13.026	SLOW
 B[1]	 P[6]	14.518	SLOW
 B[1]	 P[7]	14.547	SLOW
 B[1]	 P[8]	15.345	SLOW
 B[1]	 P[9]	16.148	SLOW
 B[1]	 P[10]	16.016	SLOW
 B[1]	 P[11]	15.916	SLOW
 B[1]	 P[12]	18.056	SLOW
 B[1]	 P[13]	18.455	SLOW
 B[1]	 P[14]	18.090	SLOW
 B[1]	 P[15]	18.530	SLOW
 B[2]	 P[2]	8.476	SLOW

Figure 40 - 8X8 Wallace
Timing Results 4

 B[2]	 P[3]	11.564	SLOW
 B[2]	 P[4]	12.492	SLOW
 B[2]	 P[5]	13.012	SLOW
 B[2]	 P[6]	14.504	SLOW
 B[2]	 P[7]	14.533	SLOW
 B[2]	 P[8]	15.331	SLOW
 B[2]	 P[9]	16.544	SLOW
 B[2]	 P[10]	16.002	SLOW
 B[2]	 P[11]	15.902	SLOW
 B[2]	 P[12]	18.042	SLOW
 B[2]	 P[13]	18.441	SLOW
 B[2]	 P[14]	18.075	SLOW
 B[2]	 P[15]	18.515	SLOW
 B[3]	 P[3]	10.318	SLOW
 B[3]	 P[4]	11.711	SLOW
 B[3]	 P[5]	12.262	SLOW
 B[3]	 P[6]	13.754	SLOW
 B[3]	 P[7]	14.014	SLOW
 B[3]	 P[8]	14.581	SLOW
 B[3]	 P[9]	16.234	SLOW
 B[3]	 P[10]	15.854	SLOW
 B[3]	 P[11]	15.685	SLOW
 B[3]	 P[12]	17.825	SLOW
 B[3]	 P[13]	18.224	SLOW
 B[3]	 P[14]	17.858	SLOW
 B[3]	 P[15]	18.298	SLOW
 B[4]	 P[4]	10.515	SLOW
 B[4]	 P[5]	10.764	SLOW
 B[4]	 P[6]	13.514	SLOW
 B[4]	 P[7]	14.050	SLOW
 B[4]	 P[8]	14.394	SLOW
 B[4]	 P[9]	15.685	SLOW
 B[4]	 P[10]	15.230	SLOW

Figure 41 - 8X8 Wallace
Timing Results 5

 B[4]	 P[10]	15.230	SLOW
 B[4]	 P[11]	15.129	SLOW
 B[4]	 P[12]	17.269	SLOW
 B[4]	 P[13]	17.668	SLOW
 B[4]	 P[14]	17.303	SLOW
 B[4]	 P[15]	17.743	SLOW
 B[5]	 P[5]	9.689	SLOW
 B[5]	 P[6]	13.051	SLOW
 B[5]	 P[7]	13.594	SLOW
 B[5]	 P[8]	13.931	SLOW
 B[5]	 P[9]	14.625	SLOW
 B[5]	 P[10]	15.060	SLOW
 B[5]	 P[11]	14.960	SLOW
 B[5]	 P[12]	17.100	SLOW
 B[5]	 P[13]	17.499	SLOW
 B[5]	 P[14]	17.133	SLOW
 B[5]	 P[15]	17.573	SLOW
 B[6]	 P[6]	10.343	SLOW
 B[6]	 P[7]	11.996	SLOW
 B[6]	 P[8]	12.340	SLOW
 B[6]	 P[9]	13.579	SLOW
 B[6]	 P[10]	14.910	SLOW
 B[6]	 P[11]	14.810	SLOW
 B[6]	 P[12]	16.950	SLOW
 B[6]	 P[13]	17.349	SLOW
 B[6]	 P[14]	16.984	SLOW
 B[6]	 P[15]	17.424	SLOW
 B[7]	 P[7]	10.895	SLOW
 B[7]	 P[8]	12.210	SLOW
 B[7]	 P[9]	13.527	SLOW
 B[7]	 P[10]	14.617	SLOW
 B[7]	 P[11]	14.516	SLOW
 B[7]	 P[12]	16.656	SLOW

Figure 42 - 8X8 Wallace
Timing Results 6

 B[7]	 P[13]	17.055	SLOW
 B[7]	 P[14]	16.690	SLOW
 B[7]	 P[15]	17.130	SLOW

Figure 43 - 8X8 Wallace Timing Results 7

Delay Comparison between Multipliers

Variable	Braun's Multiplier	Wallace Tree Multiplier
Worst Negative Slack	-13.592 ns	- 6.090 ns
Maximum Maximum Delay Path	24.383 ns (A3 to P15)	19.811 ns (A2 to P13)
Minimum Maximum Delay Path	8.650 ns (B2 to P2)	8.476 ns (B2 to P2)

Table 1 - Timing Results Summary

Firstly, comparing the results, the Wallace Tree Multiplier has been proven to have a shorter critical path, 6.090 ns compared to the Braun's Multiplier with a Worst Negative Slack of 13.592 ns, indicating an improvement of 7.502 ns. Secondly, the Wallace Tree Multiplier is proven quicker again as the Maximum Delay Path is 4.572 ns faster than the Braun's Multiplier. Furthermore, the Wallace Tree Multiplier not only has a faster Minimum Maximum Delay Path by 0.174 ns, but it also has a lower average Maximum Delay Path, compared to the Braun's Multiplier. Finally, the difference in the Maximum and Minimum Maximum Delay Path is smaller for the Wallace Tree Multiplier, not the Brauns, indicating a better balance.

As mentioned previously, these differences are a result of the architectural differences between the multipliers. The Braun's Multiplier uses a more regular structure with a series of ripple-carry adders, which have resulted in longer carry propagation paths. In contrast, the Wallace Tree Multiplier reduces partial products in parallel using a 'tree' of adders, hence the name, thus reducing the length of the critical path and distributing the delays more evenly.

Braun's Power and Utilisation Evidence

Name	Slice LUTs (20800)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)
Braun_8x8	87	27	87	32

Figure 44 - 8X8 Brauns Utilisation Graph 1

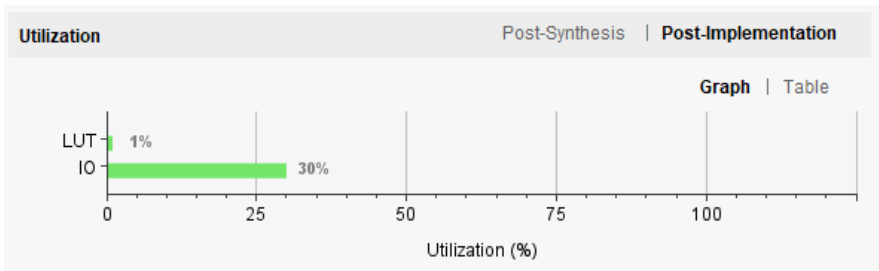


Figure 44 - 8X8 Brauns Utilisation Graph 2

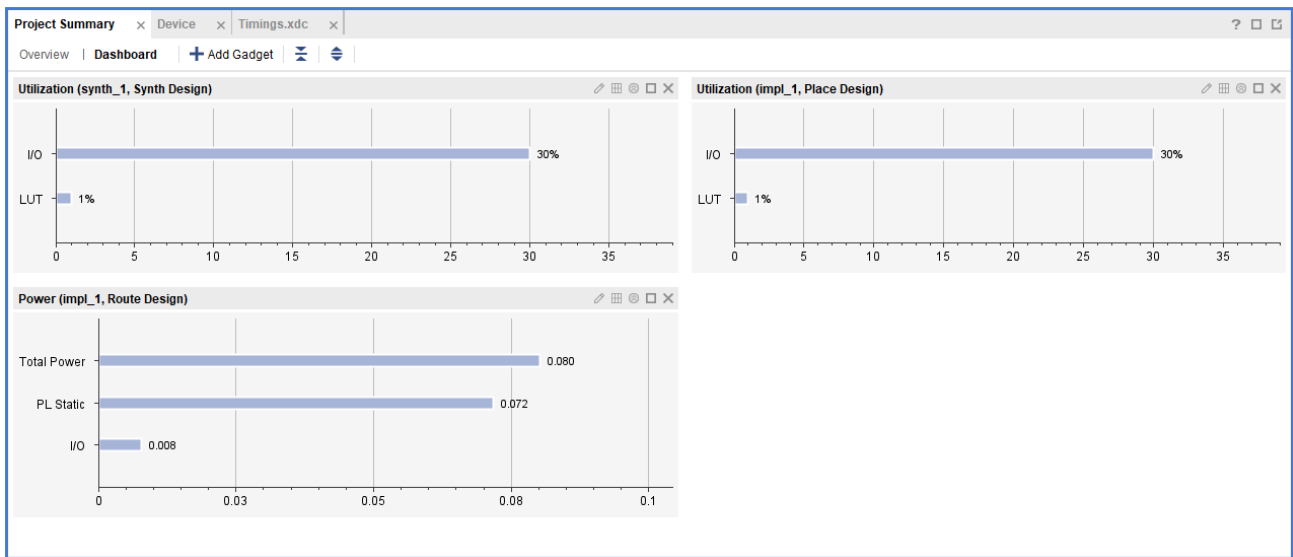


Figure 45 - 8X8 Brauns Utilisation Graphs 3,4 and 5

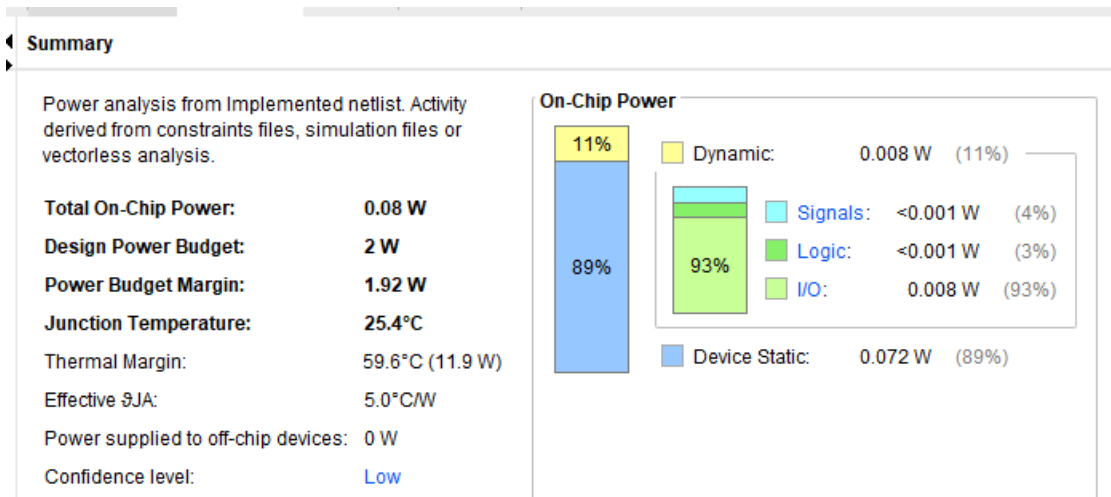


Figure 46 - 8X8 Braun's Power Consumption Evidence

Wallace Tree Power and Utilisation Evidence

Name	Slice LUTs (10400)	Slice (8150)	LUT as Logic (10400)	Bonded IOB (106)
N Wallace_8x8	91	25	91	32

Figure 47 - 8X8 Wallace Utilisation Graph 1

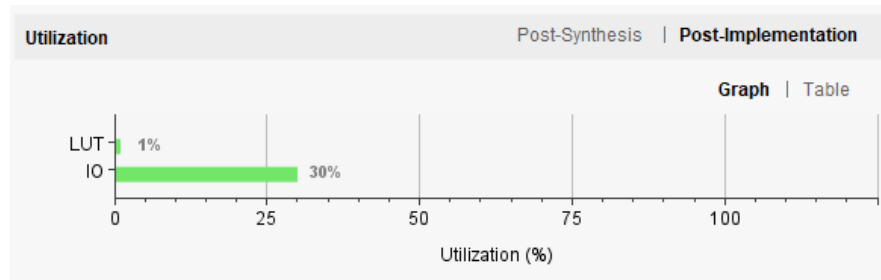


Figure 48 - 8X8 Wallace Utilisation Graph 2



Figure 49 - 8X8 Wallace Utilisation Graph 3,4 and 5

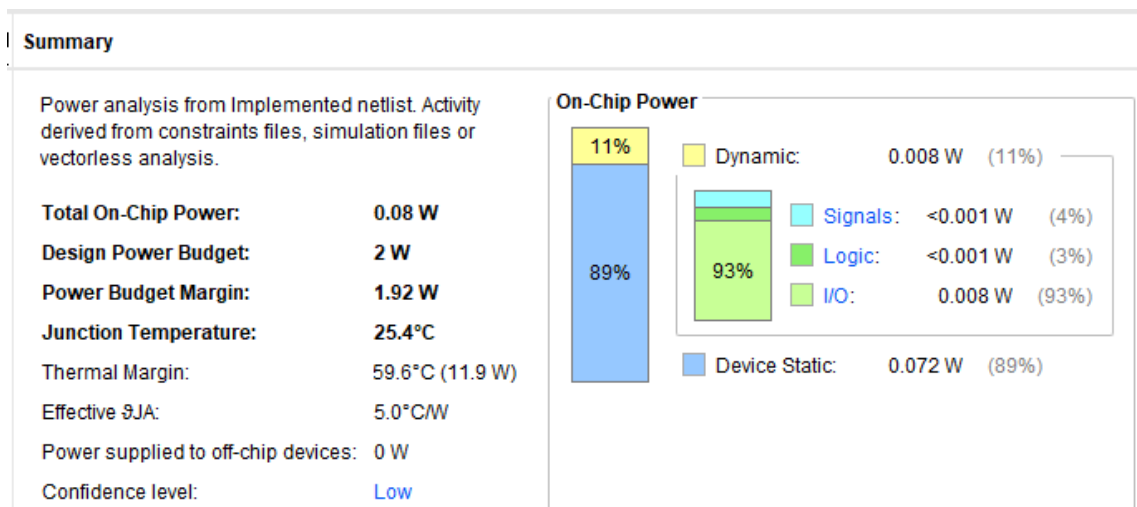


Figure 50 - 8X8 Wallace Power Consumption Evidence

Utilisation and Power Consumption Comparison

Despite their differing architectures, there are minimal differences between the Braun's Multiplier and the Wallace Tree Multiplier. The Wallace Tree uses slightly more Slice LUTs (91 compared to 87), a result of the increase in logic complexity. Both designs have a similar Bonded IOB count (27 for Braun's and 25 for Wallace), showing comparable external interfacing requirements. The Wallace Tree Multiplier uses 91 LUTs as logic and occupies 25 slices. In comparison, the Braun Multiplier uses 87 LUTs as logic but occupies 32 slices. This further proves that Braun's Multiplier uses slightly fewer logic elements, and thus is less efficient in terms of slice utilisation. Both multipliers used 1% LUT Utilisation and 30% I/O Utilisation, showing nearly identical resource usage and FPGA footprint.

The power usage for both Multipliers was 0.073W, 0.07W due to Device Static and 0.003W due to I/O usage. Both multipliers have a Junction Temperature of 25.4°C (suggesting similar thermal performances under identical operating conditions) and a Dynamic Power Usage of 11% consisting of IO, Logic and Signal usage from a 2W allocated power budget. The main difference in power would be seen for a higher demand application, as an 8X8 multiplier alone is not rough enough to show a difference on the Vivado software, with an accuracy limited to 0.001W. The Braun's Multiplier would be shown to use more power for Logic and Signal Processing compared to the Wallace Multiplier. A result of the Braun's Multiplier being less refined, due to its linear and ripple-based carry structure.

Although both differences are relatively small, they reinforce that the Wallace Tree Multiplier is a better approach both in terms of timing performance but also power efficiency. Its structure reduces switching delays, contributing to slightly lower logic and signal power usage.

Conclusion

In summary, both the Braun's and Wallace Tree Multipliers simulated offer as solutions for unsigned 8-bit multipliers, but their architectural differences result in distinct trade-offs. Firstly, the Braun's Multiplier, while simpler and easier to implement, becomes difficult to manage and inefficient as the bit width increases. As it is fully combinational, using ripple-carry structures, its scalability, speed, and power efficiency are limited. Conversely, the Wallace Tree Multiplier is more complicated to program due to its irregular adder stages, but this results in significantly better timing performance, lower power consumption, with a more balanced signal delay. This perfectly ties in with the initial research on Braun's and Wallace Tree Multipliers.

From a practical standpoint, both designs use similar resources and have a similar power consumption. However, the Wallace Tree's structured reduction approach makes it more refined and better suited for higher-performance applications. While both multipliers serve well in small-scale, low-demand scenarios, the Wallace Tree offer improved performance for practical applications like use on a larger scale FPGA implementations.

Appendices

Appendix One -VHDL Code - Braun's

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

entity Braun_8x8 is
PORT (A : in STD_LOGIC_VECTOR (7 downto 0);
      B : in STD_LOGIC_VECTOR (7 downto 0);
      P : out STD_LOGIC_VECTOR (15 downto 0));

end Braun_8x8;

architecture Behavioral of Braun_8x8 is
```

```
signal A1B0 : STD_LOGIC;
signal A2B0 : STD_LOGIC;
signal A3B0 : STD_LOGIC;
signal A4B0 : STD_LOGIC;
signal A5B0 : STD_LOGIC;
signal A6B0 : STD_LOGIC;
signal A7B0 : STD_LOGIC;
```

```
signal A0B1 : STD_LOGIC;
signal A1B1 : STD_LOGIC;
signal A2B1 : STD_LOGIC;
signal A3B1 : STD_LOGIC;
signal A4B1 : STD_LOGIC;
signal A5B1 : STD_LOGIC;
signal A6B1 : STD_LOGIC;
signal A7B1 : STD_LOGIC;
```

```
signal A0B2 : STD_LOGIC;
signal A1B2 : STD_LOGIC;
signal A2B2 : STD_LOGIC;
signal A3B2 : STD_LOGIC;
signal A4B2 : STD_LOGIC;
signal A5B2 : STD_LOGIC;
signal A6B2 : STD_LOGIC;
signal A7B2 : STD_LOGIC;
```

```
signal A0B3 : STD_LOGIC;
signal A1B3 : STD_LOGIC;
signal A2B3 : STD_LOGIC;
signal A3B3 : STD_LOGIC;
signal A4B3 : STD_LOGIC;
signal A5B3 : STD_LOGIC;
signal A6B3 : STD_LOGIC;
signal A7B3 : STD_LOGIC;
```

```
signal A0B4 : STD_LOGIC;
signal A1B4 : STD_LOGIC;
signal A2B4 : STD_LOGIC;
signal A3B4 : STD_LOGIC;
signal A4B4 : STD_LOGIC;
signal A5B4 : STD_LOGIC;
signal A6B4 : STD_LOGIC;
signal A7B4 : STD_LOGIC;
```

```
signal A0B5 : STD_LOGIC;
signal A1B5 : STD_LOGIC;
signal A2B5 : STD_LOGIC;
signal A3B5 : STD_LOGIC;
signal A4B5 : STD_LOGIC;
signal A5B5 : STD_LOGIC;
signal A6B5 : STD_LOGIC;
signal A7B5 : STD_LOGIC;
```

```
signal A0B6 : STD_LOGIC;
signal A1B6 : STD_LOGIC;
signal A2B6 : STD_LOGIC;
signal A3B6 : STD_LOGIC;
signal A4B6 : STD_LOGIC;
signal A5B6 : STD_LOGIC;
signal A6B6 : STD_LOGIC;
signal A7B6 : STD_LOGIC;
```

```
signal A0B7 : STD_LOGIC;
```

signal A1B7 : STD_LOGIC;
signal A2B7 : STD_LOGIC;
signal A3B7 : STD_LOGIC;
signal A4B7 : STD_LOGIC;
signal A5B7 : STD_LOGIC;
signal A6B7 : STD_LOGIC;
signal A7B7 : STD_LOGIC;

signal C1 : STD_LOGIC;

signal C2 : STD_LOGIC;
signal C2_1 : STD_LOGIC;

signal C3 : STD_LOGIC;
signal C3_1 : STD_LOGIC;
signal C3_2 : STD_LOGIC;

signal C4 : STD_LOGIC;
signal C4_1 : STD_LOGIC;
signal C4_2 : STD_LOGIC;
signal C4_3 : STD_LOGIC;

signal C5 : STD_LOGIC;
signal C5_1 : STD_LOGIC;
signal C5_2 : STD_LOGIC;
signal C5_3 : STD_LOGIC;
signal C5_4 : STD_LOGIC;

signal C6 : STD_LOGIC;
signal C6_1 : STD_LOGIC;
signal C6_2 : STD_LOGIC;
signal C6_3 : STD_LOGIC;
signal C6_4 : STD_LOGIC;
signal C6_5 : STD_LOGIC;

signal C7 : STD_LOGIC;
signal C7_1 : STD_LOGIC;
signal C7_2 : STD_LOGIC;
signal C7_3 : STD_LOGIC;
signal C7_4 : STD_LOGIC;
signal C7_5 : STD_LOGIC;
signal C7_6 : STD_LOGIC;

signal C8 : STD_LOGIC;
signal C8_1 : STD_LOGIC;
signal C8_2 : STD_LOGIC;
signal C8_3 : STD_LOGIC;
signal C8_4 : STD_LOGIC;
signal C8_5 : STD_LOGIC;
signal C8_6 : STD_LOGIC;

signal C9 : STD_LOGIC;
signal C9_1 : STD_LOGIC;
signal C9_2 : STD_LOGIC;
signal C9_3 : STD_LOGIC;
signal C9_4 : STD_LOGIC;
signal C9_5 : STD_LOGIC;

signal C10 : STD_LOGIC;
signal C10_1 : STD_LOGIC;
signal C10_2 : STD_LOGIC;
signal C10_3 : STD_LOGIC;
signal C10_4 : STD_LOGIC;

signal C11 : STD_LOGIC;
signal C11_1 : STD_LOGIC;
signal C11_2 : STD_LOGIC;
signal C11_3 : STD_LOGIC;

signal C12 : STD_LOGIC;
signal C12_1 : STD_LOGIC;
signal C12_2 : STD_LOGIC;

signal C13 : STD_LOGIC;
signal C13_1 : STD_LOGIC;

signal C14 : STD_LOGIC;

signal P2_1 : STD_LOGIC;

signal P3_1 : STD_LOGIC;
signal P3_2 : STD_LOGIC;

signal P4_1 : STD_LOGIC;
signal P4_2 : STD_LOGIC;

```

signal P4_3 : STD_LOGIC;

signal P5_1 : STD_LOGIC;
signal P5_2 : STD_LOGIC;
signal P5_3 : STD_LOGIC;
signal P5_4 : STD_LOGIC;

signal P6_1 : STD_LOGIC;
signal P6_2 : STD_LOGIC;
signal P6_3 : STD_LOGIC;
signal P6_4 : STD_LOGIC;
signal P6_5 : STD_LOGIC;

signal P7_1 : STD_LOGIC;
signal P7_2 : STD_LOGIC;
signal P7_3 : STD_LOGIC;
signal P7_4 : STD_LOGIC;
signal P7_5 : STD_LOGIC;
signal P7_6 : STD_LOGIC;

signal P8_1 : STD_LOGIC;
signal P8_2 : STD_LOGIC;
signal P8_3 : STD_LOGIC;
signal P8_4 : STD_LOGIC;
signal P8_5 : STD_LOGIC;
signal P8_6 : STD_LOGIC;

signal P9_1 : STD_LOGIC;
signal P9_2 : STD_LOGIC;
signal P9_3 : STD_LOGIC;
signal P9_4 : STD_LOGIC;
signal P9_5 : STD_LOGIC;

signal P10_1 : STD_LOGIC;
signal P10_2 : STD_LOGIC;
signal P10_3 : STD_LOGIC;
signal P10_4 : STD_LOGIC;

signal P11_1 : STD_LOGIC;
signal P11_2 : STD_LOGIC;
signal P11_3 : STD_LOGIC;

signal P12_1 : STD_LOGIC;
signal P12_2 : STD_LOGIC;

signal P13_1 : STD_LOGIC;

```

```

Component full_adder
port (A : in STD_LOGIC;
      B : in STD_LOGIC;
      Cin : in STD_LOGIC;
      S : out STD_LOGIC;
      Cout : out STD_LOGIC);
end component;

```

```

Component half_adder
port (A : in STD_LOGIC;
      B : in STD_LOGIC;
      S : out STD_LOGIC;
      Cout : out STD_LOGIC);
end component;

```

```

begin

```

```

A1B0 <= A (1) AND B (0);
A2B0 <= A (2) AND B (0);
A3B0 <= A (3) AND B (0);
A4B0 <= A (4) AND B (0);
A5B0 <= A (5) AND B (0);
A6B0 <= A (6) AND B (0);
A7B0 <= A (7) AND B (0);

A0B1 <= A (0) AND B (1);
A1B1 <= A (1) AND B (1);
A2B1 <= A (2) AND B (1);
A3B1 <= A (3) AND B (1);
A4B1 <= A (4) AND B (1);
A5B1 <= A (5) AND B (1);
A6B1 <= A (6) AND B (1);
A7B1 <= A (7) AND B (1);

A0B2 <= A (0) AND B (2);
A1B2 <= A (1) AND B (2);
A2B2 <= A (2) AND B (2);

```

A3B2 <= A (3) AND B (2);
A4B2 <= A (4) AND B (2);
A5B2 <= A (5) AND B (2);
A6B2 <= A (6) AND B (2);
A7B2 <= A (7) AND B (2);

A0B3 <= A (0) AND B (3);
A1B3 <= A (1) AND B (3);
A2B3 <= A (2) AND B (3);
A3B3 <= A (3) AND B (3);
A4B3 <= A (4) AND B (3);
A5B3 <= A (5) AND B (3);
A6B3 <= A (6) AND B (3);
A7B3 <= A (7) AND B (3);

A0B4 <= A (0) AND B (4);
A1B4 <= A (1) AND B (4);
A2B4 <= A (2) AND B (4);
A3B4 <= A (3) AND B (4);
A4B4 <= A (4) AND B (4);
A5B4 <= A (5) AND B (4);
A6B4 <= A (6) AND B (4);
A7B4 <= A (7) AND B (4);

A0B5 <= A (0) AND B (5);
A1B5 <= A (1) AND B (5);
A2B5 <= A (2) AND B (5);
A3B5 <= A (3) AND B (5);
A4B5 <= A (4) AND B (5);
A5B5 <= A (5) AND B (5);
A6B5 <= A (6) AND B (5);
A7B5 <= A (7) AND B (5);

A0B6 <= A (0) AND B (6);
A1B6 <= A (1) AND B (6);
A2B6 <= A (2) AND B (6);
A3B6 <= A (3) AND B (6);
A4B6 <= A (4) AND B (6);
A5B6 <= A (5) AND B (6);
A6B6 <= A (6) AND B (6);
A7B6 <= A (7) AND B (6);

A0B7 <= A (0) AND B (7);
A1B7 <= A (1) AND B (7);
A2B7 <= A (2) AND B (7);
A3B7 <= A (3) AND B (7);
A4B7 <= A (4) AND B (7);
A5B7 <= A (5) AND B (7);
A6B7 <= A (6) AND B (7);
A7B7 <= A (7) AND B (7);

P0 : P(0) <= A(0) and B(0);

P1_0 : half_adder port map (A=>A1B0 , B=>A0B1 ,S=> P(1), Cout=>C1);

P21 : full_adder port map (A=>A2B0 , B=>A1B1 , Cin=>C1, S=>P2_1, Cout=>C2_1);

P2_0 : half_adder port map (A=>A0B2 , B=>P2_1 , S=>P(2), Cout=>C2);

P31 : full_adder port map (A=>A3B0 , B=>A2B1, Cin=>C2_1, S=>P3_1, Cout=>C3_1);

P32 : full_adder port map (A=>A1B2, B=>P3_1 , Cin=>C2, S=>P3_2, Cout=>C3_2);

P3_0 : half_adder port map (A=>A0B3, B=>P3_2, S=>P(3),Cout=>C3);

P41 : full_adder port map (A=>A4B0 , B=>A3B1, Cin=>C3_1, S=>P4_1, Cout=>C4_1);

P42 : full_adder port map (A=>A2B2, B=>P4_1 , Cin=>C3_2, S=>P4_2, Cout=>C4_2);

P43 : full_adder port map (A=>A1B3, B=>P4_2 , Cin=>C3, S=>P4_3, Cout=>C4_3);

P4_0 : half_adder port map (A=>A0B4, B=>P4_3, S=>P(4),Cout=>C4);

P51 : full_adder port map (A=>A5B0 , B=>A4B1, Cin=>C4_1, S=>P5_1, Cout=>C5_1);

P52 : full_adder port map (A=>A3B2, B=>P5_1 , Cin=>C4_2, S=>P5_2, Cout=>C5_2);

P53 : full_adder port map (A=>A2B3, B=>P5_2 , Cin=>C4_3, S=>P5_3, Cout=>C5_3);

P54 : full_adder port map (A=>A1B4, B=>P5_3 , Cin=>C4, S=>P5_4, Cout=>C5_4);

P5_0 : half_adder port map (A=>A0B5, B=>P5_4, S=>P(5),Cout=>C5);

P61 : full_adder port map (A=>A6B0 , B=>A5B1, Cin=>C5_1, S=>P6_1, Cout=>C6_1);

P62 : full_adder port map (A=>A4B2, B=>P6_1 , Cin=>C5_2, S=>P6_2, Cout=>C6_2);

P63 : full_adder port map (A=>A3B3, B=>P6_2 , Cin=>C5_3, S=>P6_3, Cout=>C6_3);

P64 : full_adder port map (A=>A2B4, B=>P6_3 , Cin=>C5_4, S=>P6_4, Cout=>C6_4);

P65 : full_adder port map (A=>A1B5, B=>P6_4 , Cin=>C5, S=>P6_5, Cout=>C6_5);

P6_0 : half_adder port map (A=>A0B6, B=>P6_5, S=>P(6),Cout=>C6);

P71 : full_adder port map (A=>A7B0 , B=>A6B1, Cin=>C6_1, S=>P7_1, Cout=>C7_1);

P72 : full_adder port map (A=>A5B2, B=>P7_1 , Cin=>C6_2, S=>P7_2, Cout=>C7_2);

P73 : full_adder port map (A=>A4B3, B=>P7_2 , Cin=>C6_3, S=>P7_3, Cout=>C7_3);

P74 : full_adder port map (A=>A3B4, B=>P7_3 , Cin=>C6_4, S=>P7_4, Cout=>C7_4);

P75 : full_adder port map (A=>A2B5, B=>P7_4 , Cin=>C6_5, S=>P7_5, Cout=>C7_5);

P76 : full_adder port map (A=>A1B6, B=>P7_5 , Cin=>C6, S=>P7_6, Cout=>C7_6);

P7_0 : half_adder port map (A=>A0B7, B=>P7_6, S=>P(7), Cout=>C7);

P81 : half_adder port map (A=>A7B1 , B=>C7_1, S=>P8_1, Cout=>C8_1);
P82 : full_adder port map (A=>A6B2, B=>P8_1 , Cin=>C7_2, S=>P8_2, Cout=>C8_2);
P83 : full_adder port map (A=>A5B3, B=>P8_2 , Cin=>C7_3, S=>P8_3, Cout=>C8_3);
P84 : full_adder port map (A=>A4B4, B=>P8_3 , Cin=>C7_4, S=>P8_4, Cout=>C8_4);
P85 : full_adder port map (A=>A3B5, B=>P8_4 , Cin=>C7_5, S=>P8_5, Cout=>C8_5);
P86 : full_adder port map (A=>A2B6, B=>P8_5 , Cin=>C7_6, S=>P8_6, Cout=>C8_6);
P8_0 : full_adder port map (A=>A1B7, B=>P8_6, Cin=>C7, S=>P(8), Cout=>C8);

P91 : full_adder port map (A=>C8_1, B=>A7B2 , Cin=>C8_2, S=>P9_1, Cout=>C9_1);
P92 : full_adder port map (A=>A6B3, B=>P9_1 , Cin=>C8_3, S=>P9_2, Cout=>C9_2);
P93 : full_adder port map (A=>A5B4, B=>P9_2 , Cin=>C8_4, S=>P9_3, Cout=>C9_3);
P94 : full_adder port map (A=>A4B5, B=>P9_3 , Cin=>C8_5, S=>P9_4, Cout=>C9_4);
P95 : full_adder port map (A=>A3B6, B=>P9_4 , Cin=>C8_6, S=>P9_5, Cout=>C9_5);
P9_0 : full_adder port map (A=>A2B7, B=>P9_5, Cin=>C8, S=>P(9), Cout=>C9);

P101 : full_adder port map (A=>C9_1, B=>A7B3 , Cin=>C9_2, S=>P10_1, Cout=>C10_1);
P102 : full_adder port map (A=>A6B4, B=>P10_1 , Cin=>C9_3, S=>P10_2, Cout=>C10_2);
P103 : full_adder port map (A=>A5B5, B=>P10_2 , Cin=>C9_4, S=>P10_3, Cout=>C10_3);
P104 : full_adder port map (A=>A4B6, B=>P10_3 , Cin=>C9_5, S=>P10_4, Cout=>C10_4);
P10_0 : full_adder port map (A=>A3B7, B=>P10_4, Cin=>C9, S=>P(10), Cout=>C10);

P111 : full_adder port map (A=>C10_1, B=>A7B4 , Cin=>C10_2, S=>P11_1, Cout=>C11_1);
P112 : full_adder port map (A=>A6B5, B=>P11_1 , Cin=>C10_3, S=>P11_2, Cout=>C11_2);
P113 : full_adder port map (A=>A5B6, B=>P11_2 , Cin=>C10_4, S=>P11_3, Cout=>C11_3);
P11_0 : full_adder port map (A=>A4B7, B=>P11_3, Cin=>C10, S=>P(11), Cout=>C11);

P121 : full_adder port map (A=>C11_1, B=>A7B5 , Cin=>C11_2, S=>P12_1, Cout=>C12_1);
P122 : full_adder port map (A=>A6B6, B=>P12_1 , Cin=>C11_3, S=>P12_2, Cout=>C12_2);
P12_0 : full_adder port map (A=>A5B7, B=>P12_2, Cin=>C11, S=>P(12), Cout=>C12);

P131 : full_adder port map (A=>C12_1 , B=>A7B6, Cin=>C12_2, S=>P13_1, Cout=>C13_1);
P13_0 : full_adder port map (A=>A6B7 , B=>P13_1, Cin=>C12, S=>P(13), Cout=>C13);

P14 : full_adder port map (A=>A7B7 , B=>C13_1, Cin=>C13, S=>P(14), Cout=> C14);
P15: P(15) <= C14;

end Behavioral;

Appendix Two -VHDL Code - Wallace Tree

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

```
entity Wallace_8x8 is
PORT (A : in STD_LOGIC_VECTOR (7 downto 0);
      B : in STD_LOGIC_VECTOR (7 downto 0);
      P : out STD_LOGIC_VECTOR (15 downto 0));
```

```
end Wallace_8x8;
```

```
architecture Behavioral of Wallace_8x8 is
```

```
signal A1B0 : STD_LOGIC;
signal A2B0 : STD_LOGIC;
signal A3B0 : STD_LOGIC;
signal A4B0 : STD_LOGIC;
signal A5B0 : STD_LOGIC;
signal A6B0 : STD_LOGIC;
signal A7B0 : STD_LOGIC;
```

```
signal A0B1 : STD_LOGIC;
signal A1B1 : STD_LOGIC;
signal A2B1 : STD_LOGIC;
signal A3B1 : STD_LOGIC;
signal A4B1 : STD_LOGIC;
signal A5B1 : STD_LOGIC;
signal A6B1 : STD_LOGIC;
signal A7B1 : STD_LOGIC;
```

```
signal A0B2 : STD_LOGIC;
signal A1B2 : STD_LOGIC;
signal A2B2 : STD_LOGIC;
signal A3B2 : STD_LOGIC;
signal A4B2 : STD_LOGIC;
signal A5B2 : STD_LOGIC;
signal A6B2 : STD_LOGIC;
signal A7B2 : STD_LOGIC;
```

```
signal A0B3 : STD_LOGIC;
signal A1B3 : STD_LOGIC;
signal A2B3 : STD_LOGIC;
signal A3B3 : STD_LOGIC;
signal A4B3 : STD_LOGIC;
signal A5B3 : STD_LOGIC;
signal A6B3 : STD_LOGIC;
signal A7B3 : STD_LOGIC;
```

```
signal A0B4 : STD_LOGIC;
signal A1B4 : STD_LOGIC;
signal A2B4 : STD_LOGIC;
signal A3B4 : STD_LOGIC;
signal A4B4 : STD_LOGIC;
signal A5B4 : STD_LOGIC;
signal A6B4 : STD_LOGIC;
signal A7B4 : STD_LOGIC;
```

```
signal A0B5 : STD_LOGIC;
signal A1B5 : STD_LOGIC;
signal A2B5 : STD_LOGIC;
signal A3B5 : STD_LOGIC;
signal A4B5 : STD_LOGIC;
signal A5B5 : STD_LOGIC;
signal A6B5 : STD_LOGIC;
signal A7B5 : STD_LOGIC;
```

```
signal A0B6 : STD_LOGIC;
signal A1B6 : STD_LOGIC;
signal A2B6 : STD_LOGIC;
signal A3B6 : STD_LOGIC;
signal A4B6 : STD_LOGIC;
signal A5B6 : STD_LOGIC;
signal A6B6 : STD_LOGIC;
signal A7B6 : STD_LOGIC;
```

```
signal A0B7 : STD_LOGIC;
signal A1B7 : STD_LOGIC;
signal A2B7 : STD_LOGIC;
signal A3B7 : STD_LOGIC;
signal A4B7 : STD_LOGIC;
signal A5B7 : STD_LOGIC;
```

```
signal A6B7 : STD_LOGIC;
signal A7B7 : STD_LOGIC;
```

```
signal
```

```
S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S20,S21,S22,S23,S24,S25,S26,S27,S28,S29,S30,S31,S32,S33,S34,S35,S36,S37,S38,S39,S40,S41,
S42,S43,S44,S45,S46,S47,S48: std_logic;
```

```
signal
```

```
C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,C16,C17,C18,C19,C20,C21,C22,C23,C24,C25,C26,C27,C28,C29,C30,C31,C32,C33,C34,C35,C36,C37,C38,C39,C
40,C41,C42,C43,C44,C45,C46,C47,C48,C49,C50,C51,C52,C53,C54,C55: std_logic;
```

```
Component full_adder
```

```
port (A : in STD_LOGIC;
B : in STD_LOGIC;
Cin : in STD_LOGIC;
S : out STD_LOGIC;
Cout : out STD_LOGIC);
end component;
```

```
Component half_adder
```

```
port (A : in STD_LOGIC;
B : in STD_LOGIC;
S : out STD_LOGIC;
Cout : out STD_LOGIC);
end component;
```

```
begin
```

```
A1B0 <= A (1) AND B (0);
A2B0 <= A (2) AND B (0);
A3B0 <= A (3) AND B (0);
A4B0 <= A (4) AND B (0);
A5B0 <= A (5) AND B (0);
A6B0 <= A (6) AND B (0);
A7B0 <= A (7) AND B (0);
```

```
A0B1 <= A (0) AND B (1);
A1B1 <= A (1) AND B (1);
A2B1 <= A (2) AND B (1);
A3B1 <= A (3) AND B (1);
A4B1 <= A (4) AND B (1);
A5B1 <= A (5) AND B (1);
A6B1 <= A (6) AND B (1);
A7B1 <= A (7) AND B (1);
```

```
A0B2 <= A (0) AND B (2);
A1B2 <= A (1) AND B (2);
A2B2 <= A (2) AND B (2);
A3B2 <= A (3) AND B (2);
A4B2 <= A (4) AND B (2);
A5B2 <= A (5) AND B (2);
A6B2 <= A (6) AND B (2);
A7B2 <= A (7) AND B (2);
```

```
A0B3 <= A (0) AND B (3);
A1B3 <= A (1) AND B (3);
A2B3 <= A (2) AND B (3);
A3B3 <= A (3) AND B (3);
A4B3 <= A (4) AND B (3);
A5B3 <= A (5) AND B (3);
A6B3 <= A (6) AND B (3);
A7B3 <= A (7) AND B (3);
```

```
A0B4 <= A (0) AND B (4);
A1B4 <= A (1) AND B (4);
A2B4 <= A (2) AND B (4);
A3B4 <= A (3) AND B (4);
A4B4 <= A (4) AND B (4);
A5B4 <= A (5) AND B (4);
A6B4 <= A (6) AND B (4);
A7B4 <= A (7) AND B (4);
```

```
A0B5 <= A (0) AND B (5);
A1B5 <= A (1) AND B (5);
A2B5 <= A (2) AND B (5);
A3B5 <= A (3) AND B (5);
A4B5 <= A (4) AND B (5);
A5B5 <= A (5) AND B (5);
A6B5 <= A (6) AND B (5);
A7B5 <= A (7) AND B (5);
```

```
A0B6 <= A (0) AND B (6);
```

A1B6 <= A (1) AND B (6);
A2B6 <= A (2) AND B (6);
A3B6 <= A (3) AND B (6);
A4B6 <= A (4) AND B (6);
A5B6 <= A (5) AND B (6);
A6B6 <= A (6) AND B (6);
A7B6 <= A (7) AND B (6);

A0B7 <= A (0) AND B (7);
A1B7 <= A (1) AND B (7);
A2B7 <= A (2) AND B (7);
A3B7 <= A (3) AND B (7);
A4B7 <= A (4) AND B (7);
A5B7 <= A (5) AND B (7);
A6B7 <= A (6) AND B (7);
A7B7 <= A (7) AND B (7);

P0 : P(0) <= A(0) and B(0);

HA11 : half_adder port map (A=>A1B0,B=>A0B1,S=>P(1),Cout=>C1);
FA12 : full_adder port map (A=>A2B0,B=>A1B1,Cin=>A0B2,S=>S2,Cout=>C2);
FA13 : full_adder port map (A=>A3B0,B=>A2B1,Cin=>A1B2,S=>S3,Cout=>C3);
FA14 : full_adder port map (A=>A4B0,B=>A3B1,Cin=>A2B2,S=>S4,Cout=>C4);
FA15 : full_adder port map (A=>A5B0,B=>A4B1,Cin=>A3B2,S=>S5,Cout=>C5);
FA16 : full_adder port map (A=>A6B0,B=>A5B1,Cin=>A4B2,S=>S6,Cout=>C6);
FA17 : full_adder port map (A=>A7B0,B=>A6B1,Cin=>A5B2,S=>S7,Cout=>C7);
HA18 : half_adder port map (A=>A7B1,B=>A6B2,S=>S8,Cout=>C8);

HA21 : half_adder port map(A=>S2,B=>C1,S=>P(2),Cout=>C9);
FA22 : full_adder port map(A=>S3,B=>C2,Cin=>A0B3,S=>S10,Cout=>C10);
FA23 : full_adder port map(A=>S4,B=>C3,Cin=>A1B3,S=>S11,Cout=>C11);
FA24 : full_adder port map(A=>S5,B=>C4,Cin=>A2B3,S=>S12,Cout=>C12);
FA25 : full_adder port map(A=>S6,B=>C5,Cin=>A3B3,S=>S13,Cout=>C13);
FA26 : full_adder port map(A=>S7,B=>C6,Cin=>A4B3,S=>S14,Cout=>C14);
FA27 : full_adder port map(A=>S8,B=>C7,Cin=>A5B3,S=>S15,Cout=>C15);
FA28 : full_adder port map(A=>A7B2,B=>C8,Cin=>A6B3,S=>S16,Cout=>C16);

HA31 : half_adder port map(A=>S10,B=>C9,S=>P(3),Cout=>C17);
FA32 : full_adder port map(A=>S11,B=>C10,Cin=>A0B4,S=>S18,Cout=>C18);
FA33 : full_adder port map(A=>S12,B=>C11,Cin=>A1B4,S=>S19,Cout=>C19);
FA34 : full_adder port map(A=>S13,B=>C12,Cin=>A2B4,S=>S20,Cout=>C20);
FA35 : full_adder port map(A=>S14,B=>C13,Cin=>A3B4,S=>S21,Cout=>C21);
FA36 : full_adder port map(A=>S15,B=>C14,Cin=>A4B4,S=>S22,Cout=>C22);
FA37 : full_adder port map(A=>S16,B=>C15,Cin=>A5B4,S=>S23,Cout=>C23);
FA38 : full_adder port map(A=>A7B3,B=>C16,Cin=>A6B4,S=>S24,Cout=>C24);

HA41 : half_adder port map(A=>S18,B=>C17,S=>P(4),Cout=>C25);
FA42 : full_adder port map(A=>S19,B=>C18,Cin=>A0B5,S=>S26,Cout=>C26);
FA43 : full_adder port map(A=>S20,B=>C19,Cin=>A1B5,S=>S27,Cout=>C27);
FA44 : full_adder port map(A=>S21,B=>C20,Cin=>A2B5,S=>S28,Cout=>C28);
FA45 : full_adder port map(A=>S22,B=>C21,Cin=>A3B5,S=>S29,Cout=>C29);
FA46 : full_adder port map(A=>S23,B=>C22,Cin=>A4B5,S=>S30,Cout=>C30);
FA47 : full_adder port map(A=>S24,B=>C23,Cin=>A5B5,S=>S31,Cout=>C31);
FA48 : full_adder port map(A=>A7B4,B=>C24,Cin=>A6B5,S=>S32,Cout=>C32);

HA51 : half_adder port map(A=>S26,B=>C25,S=>P(5),Cout=>C33);
FA52 : full_adder port map(A=>S27,B=>C26,Cin=>A0B6,S=>S34,Cout=>C34);
FA53 : full_adder port map(A=>S28,B=>C27,Cin=>A1B6,S=>S35,Cout=>C35);
FA54 : full_adder port map(A=>S29,B=>C28,Cin=>A2B6,S=>S36,Cout=>C36);
FA55 : full_adder port map(A=>S30,B=>C29,Cin=>A3B6,S=>S37,Cout=>C37);
FA56 : full_adder port map(A=>S31,B=>C30,Cin=>A4B6,S=>S38,Cout=>C38);
FA57 : full_adder port map(A=>S32,B=>C31,Cin=>A5B6,S=>S39,Cout=>C39);
FA58 : full_adder port map(A=>A7B5,B=>C32,Cin=>A6B6,S=>S40,Cout=>C40);

HA61 : half_adder port map(A=>S34,B=>C33,S=>P(6),Cout=>C41);
FA62 : full_adder port map(A=>S35,B=>C34,Cin=>A0B7,S=>S42,Cout=>C42);
FA63 : full_adder port map(A=>S36,B=>C35,Cin=>A1B7,S=>S43,Cout=>C43);
FA64 : full_adder port map(A=>S37,B=>C36,Cin=>A2B7,S=>S44,Cout=>C44);
FA65 : full_adder port map(A=>S38,B=>C37,Cin=>A3B7,S=>S45,Cout=>C45);
FA66 : full_adder port map(A=>S39,B=>C38,Cin=>A4B7,S=>S46,Cout=>C46);
FA67 : full_adder port map(A=>S40,B=>C39,Cin=>A5B7,S=>S47,Cout=>C47);
FA68 : full_adder port map(A=>A7B6,B=>C40,Cin=>A6B7,S=>S48,Cout=>C48);

HA71 : half_adder port map(A=>S42,B=>C41,S=>P(7),Cout=>C49);
FA72 : full_adder port map(A=>S43,B=>C42,Cin=>C49,S=>P(8),Cout=>C50);
FA73 : full_adder port map(A=>S44,B=>C43,Cin=>C50,S=>P(9),Cout=>C51);
FA74 : full_adder port map(A=>S45,B=>C44,Cin=>C51,S=>P(10),Cout=>C52);
FA75 : full_adder port map(A=>S46,B=>C45,Cin=>C52,S=>P(11),Cout=>C53);
FA76 : full_adder port map(A=>S47,B=>C46,Cin=>C53,S=>P(12),Cout=>C54);
FA77 : full_adder port map(A=>S48,B=>C47,Cin=>C54,S=>P(13),Cout=>C55);
FA78 : full_adder port map(A=>A7B7,B=>C48,Cin=>C55,S=>P(14),Cout=>P(15));

end Behavioral;

Appendix Three - References

[1]

N. S., "Design and Analysis of 8-bit Array, Carry Save Array, Braun, Wallace Tree and Vedic Multipliers," *International Journal of Psychosocial Rehabilitation*, vol. 24, no. 3, pp. 2687–2697, Mar. 2020, doi: <https://doi.org/10.37200/ijpr/v24i3/pr2020305>.

[2]

1MADHU GUDHIMETLA, "COMPARISON OF DIFFERENT TYPES OF MULTIPLIERS WITH RESPECT TO SPEED, AREA AND POWER 1 MADHU GUDHIMETLA, 2 CM ANANDA," Oct. 2017. Available: https://www.digitalxplore.org/up_proc/pdf/315-150546881948-53.pdf

[3]

Mansaf Alam, *Digital logic design*. Prentice-Hall Of India, 2016.

[4]

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*, vol. 3, no. 3, pp. 19–27, Jul. 2019, doi: <https://doi.org/10.33166/aetic.2019.03.003>.

[5]

S. S A Marimuthu, "Design of Area-Delay-Power Efficient Adaptive Filter using Wallace Tree Multiplier," *International Journal of Scientific Engineering and Research*, vol. 2, no. 4, pp. 121–125, Apr. 2014, doi: <https://doi.org/10.70729/ij2013241>.

[6]

B. Liu, D. Chen, C. Xiong, and K. Xing, "New methods for binary multiplication," *International Journal of Computer Mathematics*, vol. 82, no. 1, pp. 13–22, Jan. 2005, doi: <https://doi.org/10.1080/00207160412331290676>.

[7]

V. Thamizharasan and V. Parthipan, "Design of efficient binary multiplier architecture using hybrid compressor with FPGA implementation," *Scientific Reports*, vol. 14, no. 1, Apr. 2024, doi: <https://doi.org/10.1038/s41598-024-58482-0>.

[8]

R. Pandey, N. Srivastava, Neeraj Kumar Singh, and K. Tyagi, *Quantum Computing: A Shift from Bits to Qubits*. Springer Nature, 2023.

[9]

K. Jang, W. Kim, S. Lim, Y. Kang, Y. Yang, and H. Seo, "Quantum Binary Field Multiplication with Optimized Toffoli Depth and Extension to Quantum Inversion," *Sensors*, vol. 23, no. 6, pp. 3156–3156, Mar. 2023, doi: <https://doi.org/10.3390/s23063156>.

[10]

K. Madhuri, C. Bhargav, T. Chakrapani, and K. Sudhakar, "Design of Low Power Binary Multiplier," *International Journal of Research and Scientific Innovation (IJRSI)* |, vol. IV, no. 12, 2017, Accessed: Mar. 11, 2025. [Online]. Available: <https://rsisinternational.org/journals/ijrsi/digital-library/volume-4-issue-12/18-22.pdf>

[11]

S. Immareddy and A. Sundaramoorthy, "A survey paper on design and implementation of multipliers for digital system applications," *Artificial Intelligence Review*, Jan. 2022, doi: <https://doi.org/10.1007/s10462-021-10113-0>.

[12]

S. D. Brown and Z. G. Vranesic, *Fundamentals of digital logic with VHDL design*. Chennai: McGraw-Hill Education (India) Private Limited, 2012.

[13]

V. A. Pedroni, *Circuit Design with VHDL*. The Mit Press, 2019.

[14]

D. L. Perry, *Vhdl*. New York: McGraw-Hill, 1998.

[15]

R. Waxman, L. Saunders, and H. Carter, "VHDL links design, test, and maintenance," *IEEE Spectrum*, vol. 26, no. 5, pp. 40–44, May 1989, doi: <https://doi.org/10.1109/6.30762>.

[16]

L. John, *Digital systems design using vhdI, international edition*. Cengage Learning Custom P, 2017.

[17]

DIGILENT, "Basys 3™ FPGA Board Reference Manual," *DIGILENT*, Apr. 08, 2016.

https://digilent.com/reference/_media/basys3:basys3_rm.pdf?srsItd=AfmBOoqBX9FHz0PzVqjRiFX5vINIP24ppR2JroHScrGxMzu9VftpvA3