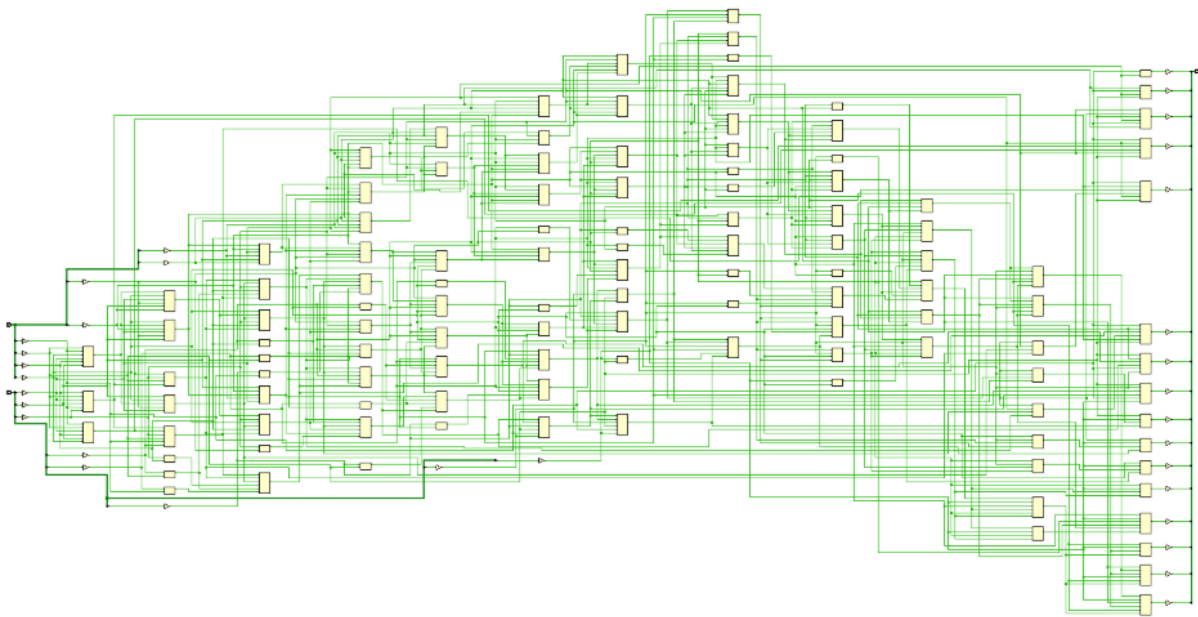


Electronics II – ACDA6001

Critical Evaluation of Braun's Multiplier and Wallace Tree Multiplier Using VHDL



Jayden Fox – 2316329

Table of Contents

Abstract	4
1. Introduction.....	5
2. Background Research	6
2.1 Braun's Array Multiplier	6
2.2 Wallace Tree Multiplier	6
2.3 Dadda multiplier	7
2.4 Applications	7
3. System Operation	8
3.1 Block Diagrams and Algorithms of Multipliers	11
4. Timing diagrams and schematics of the designs	16
4.1.1 Timing Performance – Braun's multiplier	16
4.1.2 Timing performance – Wallace tree multiplier.....	17
4.2 Longest Delay of the multipliers	17
4.3 Schematics	18
5. FPGA Design	21
6. Simulation results	25
6.1 8x8 Braun's multiplier simulation results	26
6.2 8x8 Wallace tree multiplier simulation results.....	28
6.3 Comparison, analysis and interpretation of results.....	30
7. Interpretation of utilisation and power reports	32
7.1 Utilisation Reports	32
7.2 Power Reports	33
8. Conclusion and recommendations for future improvements	35
Bibliography.....	36
Appendix.....	38
Appendix A. VHDL Codes	38
Appendix B. FPGA Simulation Results	58
Appendix C. Handwritten labels on 8x8 Braun's multiplier block diagram.	62

Table of Figures

Figure 1 – 8x8 Braun’s multiplier operation.	9
Figure 2 – 8x8 Wallace tree multiplier operation.	10
Figure 3 – 8x8 Braun’s multiplier block diagram.	12
Figure 4 - 8x8 Wallace tree multiplier block diagram.	13
Figure 5 - 8x8 Wallace tree multiplier dot diagram.	14
Figure 6 – 8x8 Wallace tree multiplication table	15
Figure 7 – 8x8 Braun’s multiplier timing summary after synthesis.	16
Figure 8 – 8x8 Braun’s multiplier timing summary after implementation.	16
Figure 9 - 8x8 Wallace tree multiplier timing summary after synthesis.	17
Figure 10 – 8x8 Wallace tree multiplier timing summary after implementation.	17
Figure 11 – 8x8 Braun’s multiplier RTL schematic.	18
Figure 12 - 8x8 Wallace tree multiplier RTL schematic.	19
Figure 13 – 8x8 Braun’s multiplier implementation schematic.	20
Figure 14 - 8x8 Wallace tree implementation schematic.	20
Figure 15 - 8x8 Braun's multiplier schematic of devices	21
Figure 16 - 8x8 Braun's multiplier I/O planning schematic.	22
Figure 17 - 8x8 Wallace tree multiplier schematic of devices.	23
Figure 18 - 8x8 Wallace tree multiplier I/O planning schematic.	24
Figure 19 - 'Force Clock' section of simulation tool.	25
Figure 20 - 8x8 Braun's multiplier low example simulation result.	26
Figure 21 - 8x8 Braun's multiplier medium example simulation result.	26
Figure 22 - 8x8 Braun's multiplier high example simulation result.	27
Figure 23 - 8x8 Braun's multiplier final example simulation result.	27
Figure 24 - 8x8 Wallace tree multiplier low example simulation result.	28
Figure 25 - 8x8 Wallace tree multiplier medium example simulation result.	28
Figure 26 - 8x8 Wallace tree multiplier high example simulation result.	29
Figure 27 - 8x8 Wallace tree multiplier final example simulation result.	29
Figure 28 – Verification of simulation results using calculator.	31
Figure 29 – 8x8 Braun’s multiplier utilisation report.	32
Figure 30 - 8x8 Wallace tree utilisation report.	32
Figure 31 – 8x8 Braun’s multiplier power report after synthesis.	33
Figure 32 – 8x8 Braun’s multiplier power report after implementation.	33
Figure 33 – 8x8 Wallace tree multiplier power report after synthesis.	34
Figure 34 – 8x8 Wallace tree multiplier power report after implementation.	34
Figure 35 – 8x8 Braun’s multiplier entity declaration.	38
Figure 36 – Half adder and full adder components.	38
Figure 37 – Partial product generation through AND gates.	39
Figure 38 – Carry signals declarations example.	40
Figure 39 – Sum signals declaration.	40

Figure 40 – Half adder component in use	41
Figure 41 – Full adder component in use.	41
Figure 42 – 15x10 result from Braun’s multiplier on FPGA board.....	58
Figure 43 - 15x10 result from Wallace tree multiplier on FPGA board.	58
Figure 44 – 9Fx7E result from Braun’s multiplier on FPGA board.	59
Figure 45 - 9Fx7E result from Wallace tree multiplier on FPGA board.....	59
Figure 46 - DExEF result from Braun’s multiplier on FPGA board.	60
Figure 47 - DExEF result from Wallace tree multiplier on FPGA board.	60
Figure 48 - FFxFF result from Braun’s multiplier on FPGA board.	61
Figure 49 - FFxFF result from Wallace tree multiplier on FPGA board.	61

Abstract

This report presents the design, implementation and critical evaluation of the 8x8 Braun’s and Wallace tree parallel multiplier architectures using VHDL to program a Xilinx Artix-7 FPGA. Both multipliers were designed to produce a 16-bit output from two 8-bit operands, and their performance in doing so was assessed through simulation, timing analysis and report evaluations. The Braun’s multiplier showed a simple, regular structure but a longer critical path due to ripple-carry propagation, while the Wallace tree multiplier achieved a faster operation through carry-save reduction whilst maintaining similar LUT and power usage to the Braun’s multiplier. The study shows that Wallace tree multipliers are better suited for high-speed FPGA applications, while Braun’s multipliers suit simple, low-power designs.

1. Introduction

Parallel multipliers are essential in modern digital systems to achieve high-speed computation, which is an important requirement in modern embedded electrical systems. This report investigates two parallel multiplier architectures, known as the Braun's multiplier and the Wallace Tree multiplier, and evaluates their suitability for FPGA (Field Programmable Gate Array) implementation using VHDL. Although both architectures generate a 16-bit product from two 8-bit input operands, they differ in structure, critical path depth and hardware utilisation. The design of the multipliers will be developed and simulated using the Xilinx Vivado Design Suite, targeting the Artix-7 FPGA (xc7a35tcpg236-1).

The aim of this report is to design, simulate and critically evaluate both multipliers. The report will include partial-product generation, VHDL designs, simulation results, FPGA synthesis and a detailed comparison of timing delay, critical path behaviour, power consumption and hardware utilisation of the two multipliers. By analysing the results of the strengths and limitations of each architecture, the report will provide a strong recommendation for selecting multipliers in embedded applications.

2. Background Research

Parallel multipliers have been thoroughly studied due to their importance in high-speed circuits. The two dominant architectures are the Braun's multiplier and the Wallace Tree multiplier, where both generate partial products in parallel but are different in how they reduce and accumulate these products. This leads to differences in speed, area and power consumption.

2.1 Braun's Array Multiplier

The Braun's multiplier, also known as an Array multiplier, is consistently described as a regular, grid-structure array made up of AND gates and ripple carry adders [1, 2]. Each bit of one operand is multiplied with each bit of the other operand to generate partial products, which are then summed in a fixed, predictable manner [3]. The routing of the Braun's multiplier is predictable, and it has a simple layout, making it popular for FPGA implementations [4].

Although the Braun's multiplier brings the advantage of being simple and predictable, they suffer from a long critical path as carries must propagate sequentially through multiple adder stages. Existing literature suggests they also bring a higher power consumption, as all adders toggle every cycle and there is no carry-save feature [5]. These disadvantages make the Braun's multiplier less suitable for high performance applications where speed is important.

2.2 Wallace Tree Multiplier

The Wallace tree multiplier is a high-speed digital circuit used in processors to multiply and was introduced by C.S.Wallace in 1964 [6, 7]. Instead of summing partial products sequentially, the Wallace tree uses a carry-save adder tree to reduce multiple partial products using many full adders and half adders [8]. The structure of the Wallace tree significantly reduces the number of sequential adder levels, resulting in shorter critical paths and higher operating frequencies [9, 10].

There are fewer active adders when operating compared to a Braun's multiplier, leading to lower power use and making it highly suitable for high-speed DSP applications [7]. Existing research suggests the drawbacks of Wallace tree multipliers are that they require more complex wiring and careful optimisation [11].

2.3 Dadda multiplier

In addition to the Braun's and Wallace tree multipliers, many other multiplier architectures have been developed to increase performance in speed, area and power consumption.

One popular alternative is the Dadda multiplier, which has a similar concept to the Wallace tree multiplier but uses a more structured and conservative reduction strategy [12]. The Dadda multiplier aims to minimise the total number of adders by delaying reduction wherever possible [13]. It does this by limiting the maximum number of bits per column at each stage. This results in less complex hardware and reduced area compared to the previously mentioned multipliers, whilst still achieving high performance and operating speeds [14]. However, the Dadda multiplier tends to be less intuitive due to its reduction process.

2.4 Applications

Braun's multipliers are typically used in low-complexity embedded systems where ease of implementation and predictable behaviour is preferred. On the other hand, Wallace tree and Dadda multipliers are preferred in performance critical applications such as finite impulse response (FIR) filtering, image processing, and cryptographic engines, where a reduced delay and a high performance is essential [15, 16].

3. System Operation

To explain the operation of each multiplier, flowcharts have been created. Figure 1 shows the operation of the 8x8 Braun's multiplier, where the Braun's multiplier generates partial products through 'AND' operations between the two input operands. The partial products are then arranged in a regular array based on their binary weights and added sequentially using full adders. After all partial products have been combined, the final carry stage will produce a 16-bit output product [3]. The Braun's architecture is easy to follow and has a regular layout, making it suitable for simple hardware designs [3].

As seen in Figure 2, the 8x8 Wallace tree multiplier operates by also generating all partial products through AND operations between the input operands. Each partial product is then assigned to a 'P' column, where these columns are then reduced in parallel using a Wallace tree structure consisting of half adders and full adders [6]. The adders decrease the number of operands at each stage whilst minimising any delays. This process is repeated until each column contains a maximum of 2 bits. A final carry addition stage then combines the remaining sums and carries to produce a 16-bit product output [8].

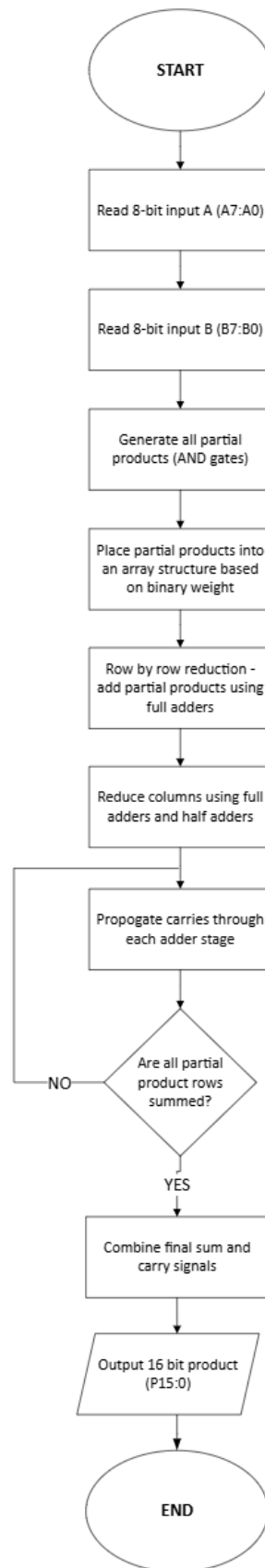


Figure 1 – 8x8 Braun's multiplier operation.

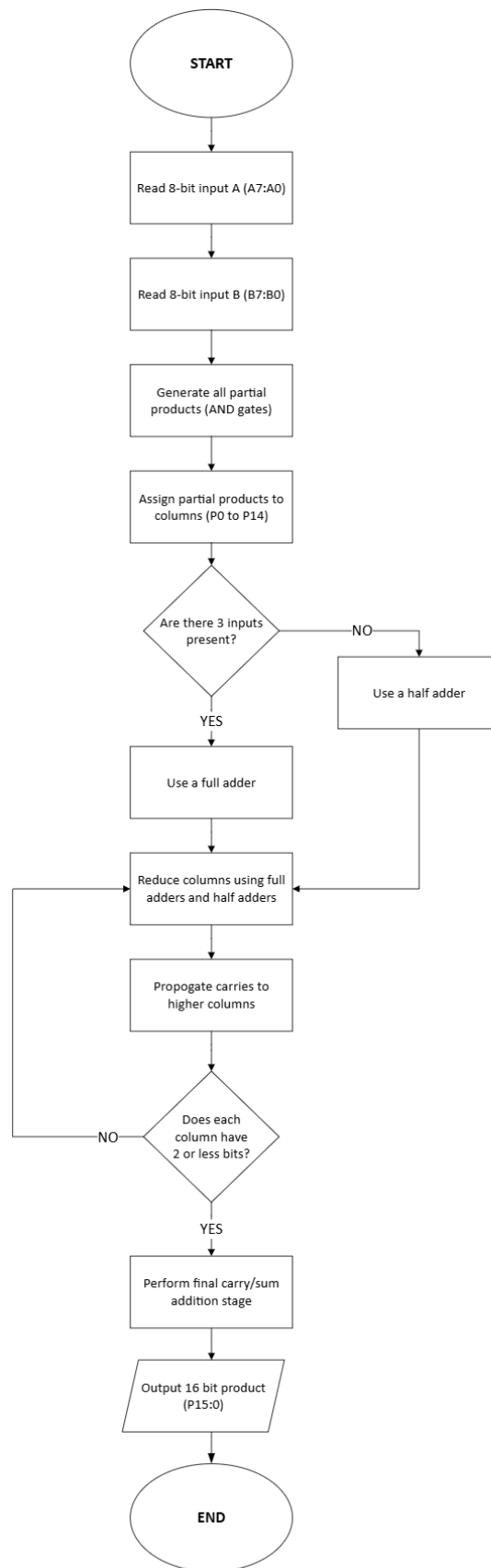


Figure 2 – 8x8 Wallace tree multiplier operation.

3.1 Block Diagrams and Algorithms of Multipliers

The block diagram for the 8x8 Braun's multipliers (seen in Figure 3) shows the operation of the multiplier begins by generating the partial products, which are then arranged by weight and summed using a regular array of full adders and half adders. Sum outputs progress vertically, while carries ripple through the array. The final product bits (P0:P15) are then shown at the bottom.

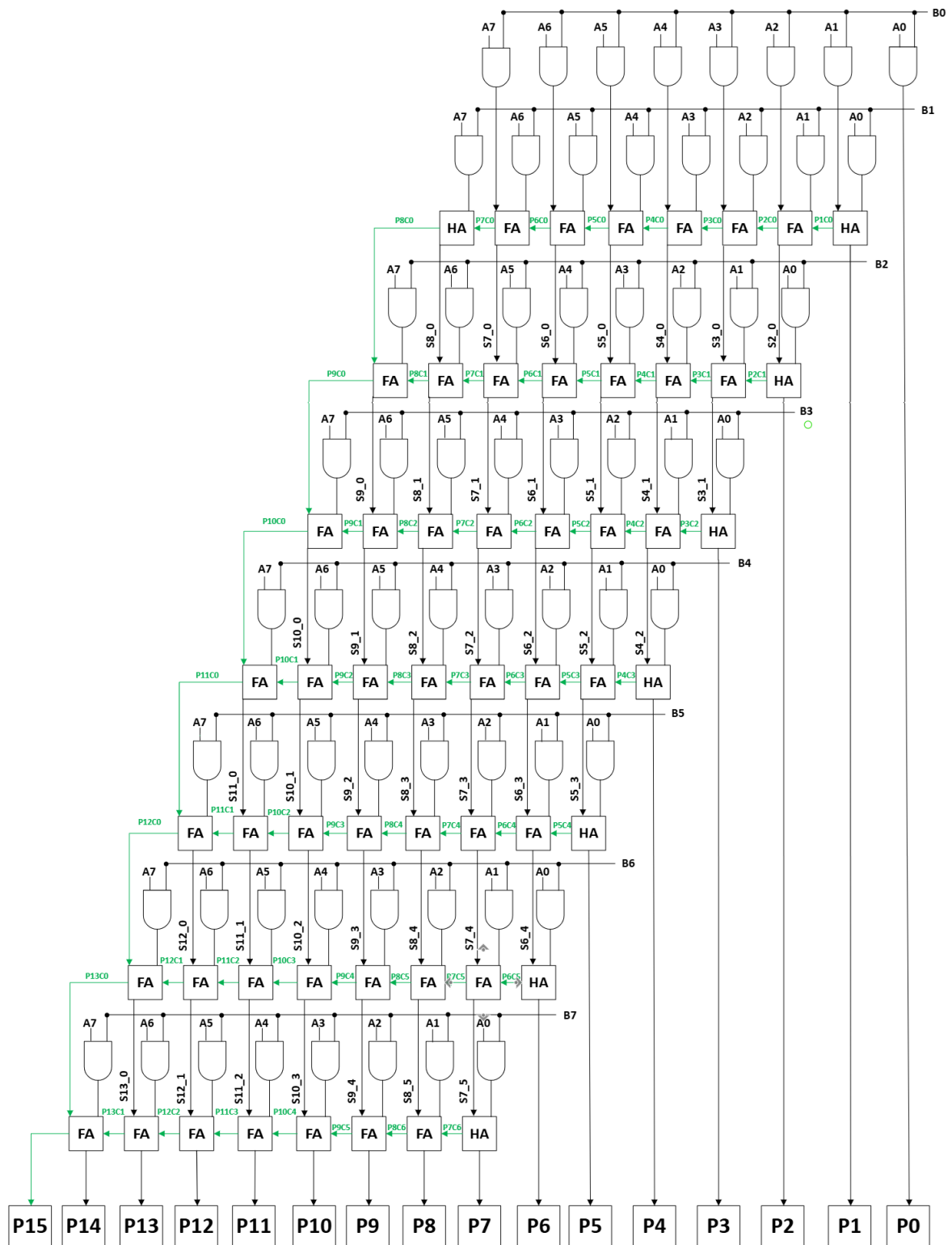


Figure 3 – 8x8 Braun's multiplier block diagram.

The operation of the 8x8 Wallace tree multiplier is displayed in the block diagram and dot diagram, shown in Figure 4 and Figure 5 respectively. The block diagram shows the physical implementation with the adders, whilst the dot diagram illustrates how column heights are reduced at each stage. The multiplier first generates the partial products and arranges them by weight as shown in level 1. The reduction process is then performed across levels 2 to 4 using full adders and half adders [11]. At each level, bits of equal weight are combined, where the sum outputs remain in the same column and carry outputs move to the next column. By level 5, the partial products are reduced to 2 rows, and a final 16-bit output is then produced.

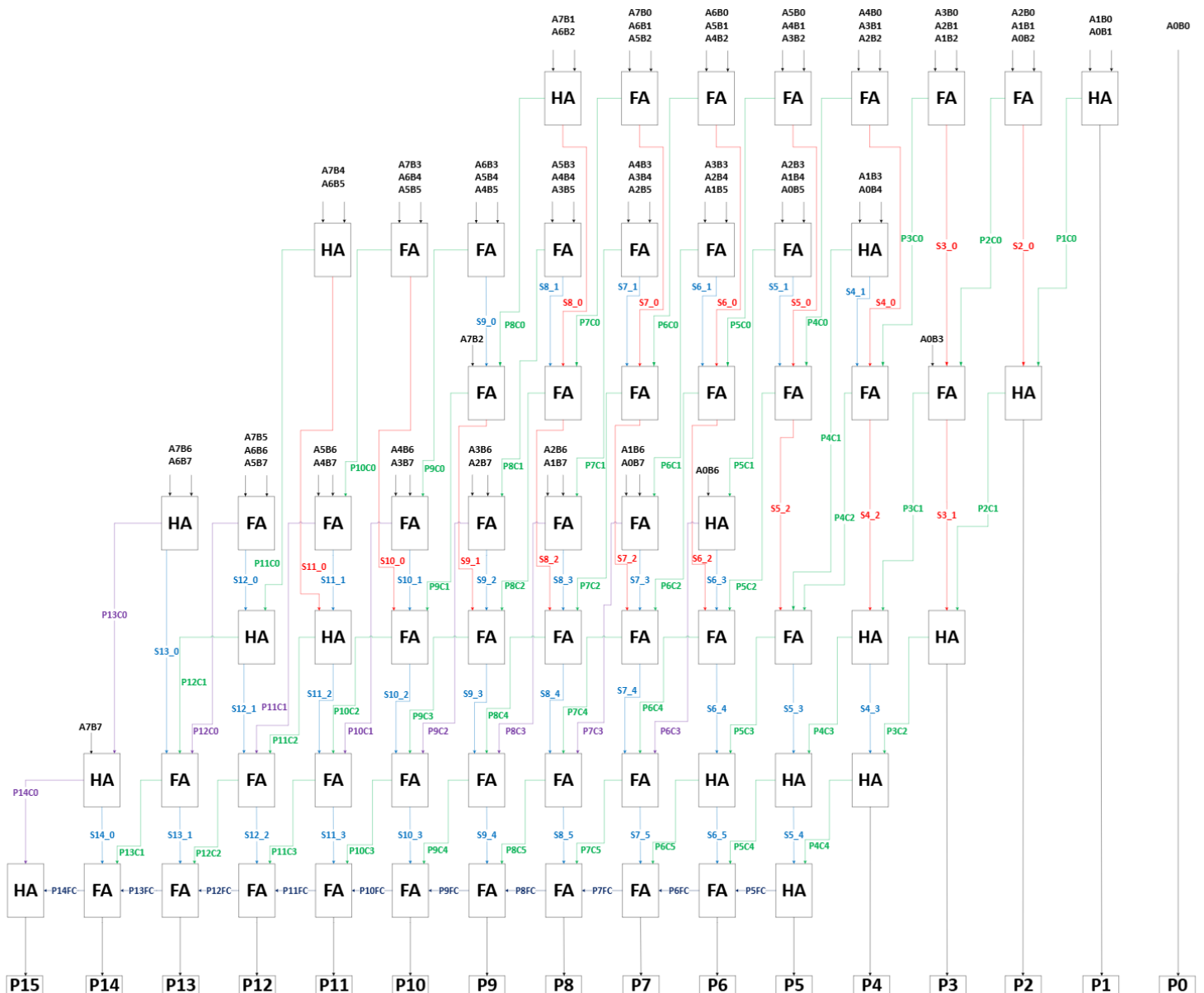
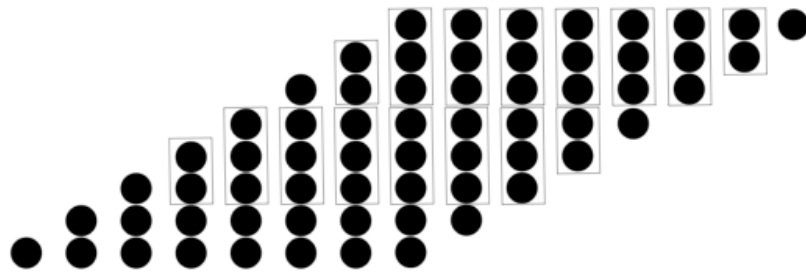
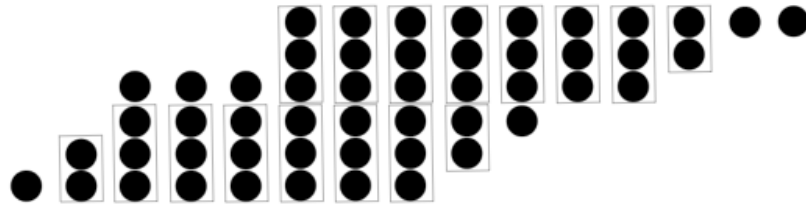


Figure 4 - 8x8 Wallace tree multiplier block diagram.

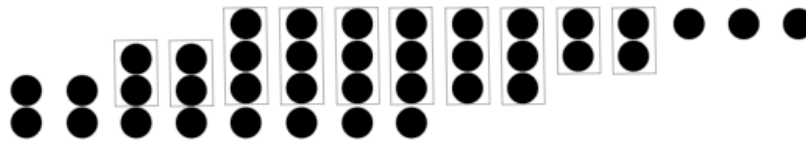
Level 1



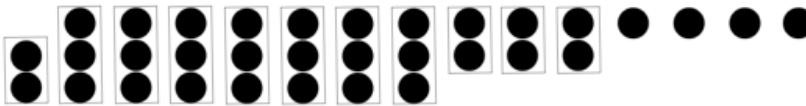
Level 2



Level 3



Level 4



Level 5



Figure 5 - 8x8 Wallace tree multiplier dot diagram.

[illegible]

4. Timing diagrams and schematics of the designs

4.1.1 Timing Performance – Braun's multiplier

After synthesis, the Worst Negative Slack (WNS) was shown to be 35.084ns (seen in Figure 7) whereas after implementation, the WNS was 27.496ns (seen in Figure 8). This indicates that the design meets timing expectations, with the critical path delay being below the target clock period. The Braun's multiplier demonstrates a very strong time performance both after synthesis and after implementation. The design is shown to be an efficient, low power Braun's multiplier which performs within timing requirements.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 35.084 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 7 – 8x8 Braun's multiplier timing summary after synthesis.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 27.496 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 8 – 8x8 Braun's multiplier timing summary after implementation.

4.1.2 Timing performance – Wallace tree multiplier

After synthesis, the WNS was 38.554ns with all timing constraints being met (shown in Figure 9). After implementation, the WNS drops to 32.928ns (shown in Figure 10). This shows the Wallace tree multiplier has an excellent timing performance both after synthesis and after implementation. Although the WNS drops after implementation, the design keeps a large positive timing margin. This confirms the effectiveness of the Wallace tree reduction in minimising the critical path and that it is well suited for FPGA implementation.

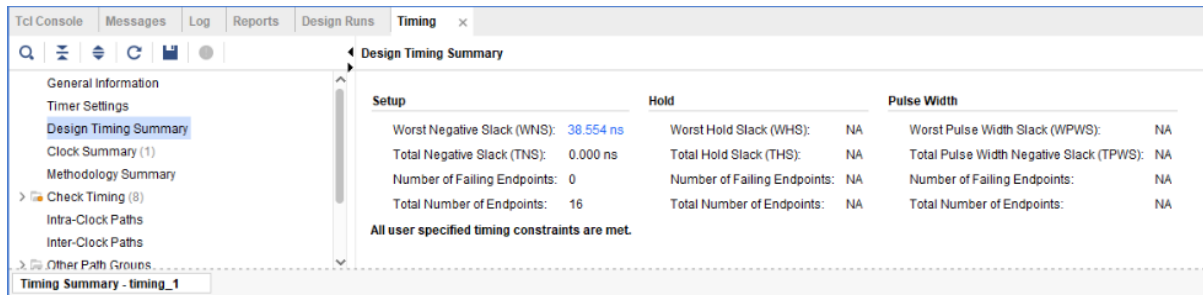


Figure 9 - 8x8 Wallace tree multiplier timing summary after synthesis.

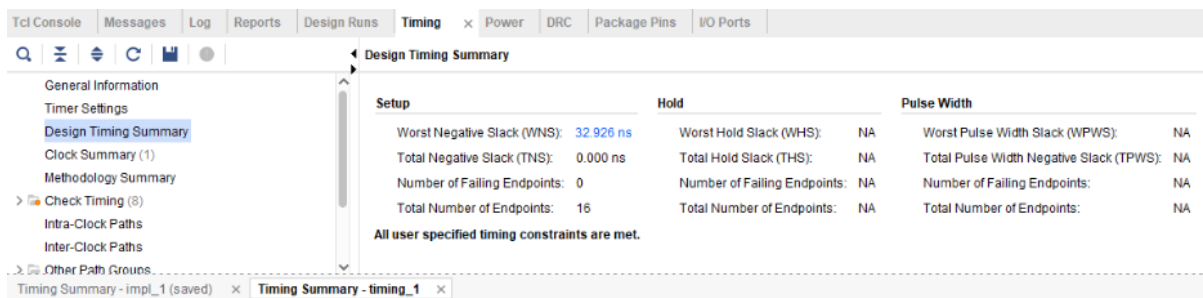


Figure 10 – 8x8 Wallace tree multiplier timing summary after implementation.

4.2 Longest Delay of the multipliers

The longest delay of the multipliers was calculated by treating the initial AND gate delay as 1ns and each full/half adder as 2ns. The following formula was used:

$$\text{Longest Delay} = \text{Delay of 1st AND gate} + (\text{number of adders} * 2\text{ns})$$

Therefore, the longest delay of the **8x8 Braun's multiplier** was:

$$\text{Longest Delay} = 1\text{ns} + (20 * 2\text{ns}) = 41\text{ns}$$

Also, the longest delay of the **8x8 Wallace tree multiplier** was:

$$\text{Longest Delay} = 1\text{ns} + (15 * 2\text{ns}) = 31\text{ns}$$

The Wallace tree is shown to be faster as it uses fewer adder stages, and most of the carry delays only occur in the final adders.

4.3 Schematics

The RTL schematics of both multipliers can be seen in Figure 11 and Figure 12. The RTL schematic of the Braun's multiplier shows a regular array structure formed by AND gates, full adders and half adders. This multiplier type has a simple layout, but results in a longer critical path due to sequential carry propagation. The RTL schematic of the Wallace tree multiplier displays a multi-level reduction structure, where partial products are reduced in parallel using half adders and full adders. Although the logic and routing is more complex than the Braun's multiplier, this structure significantly reduces the critical path and enables higher operating speeds. This same conclusion is also reached with the implementation schematics, seen in Figure 13 and Figure 14.

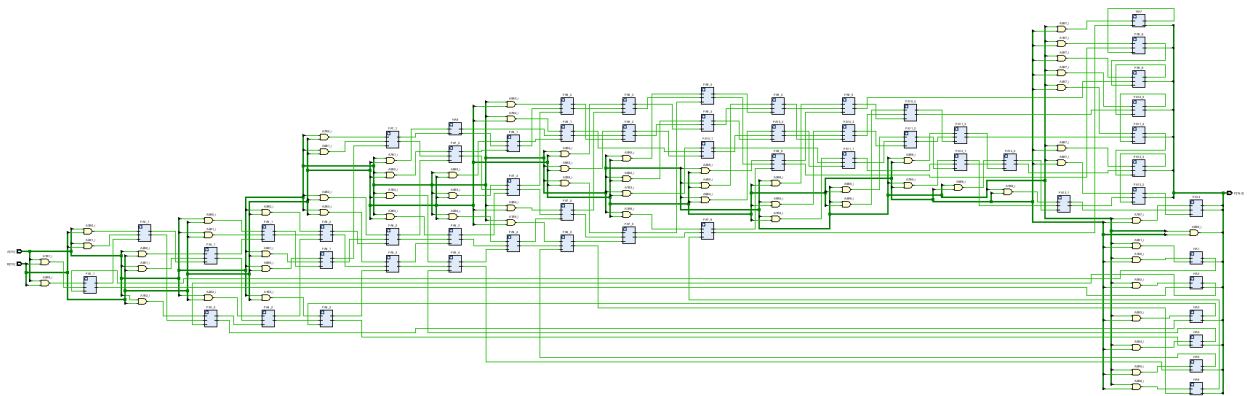


Figure 11 – 8x8 Braun's multiplier RTL schematic.

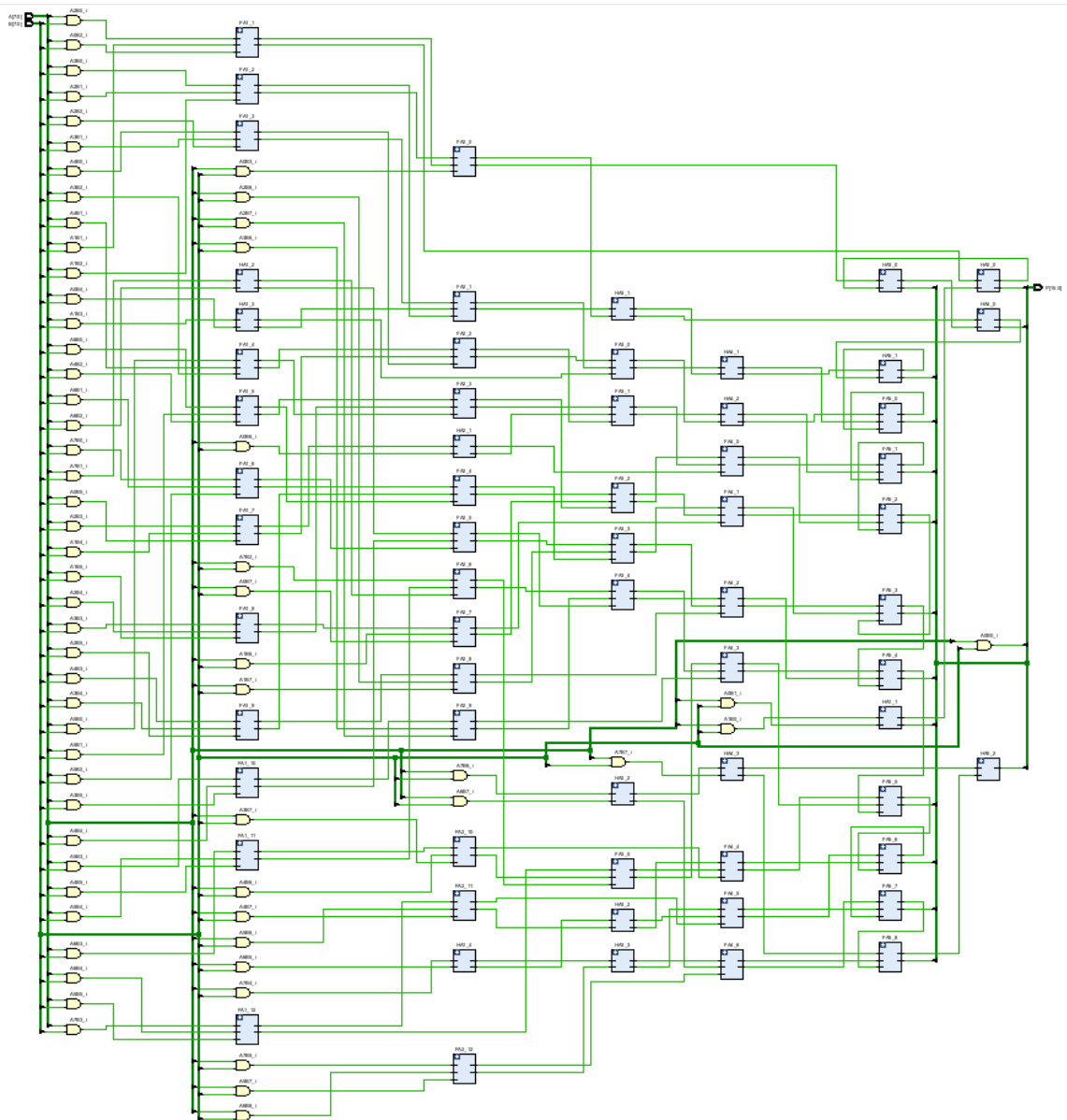


Figure 12 - 8x8 Wallace tree multiplier RTL schematic.

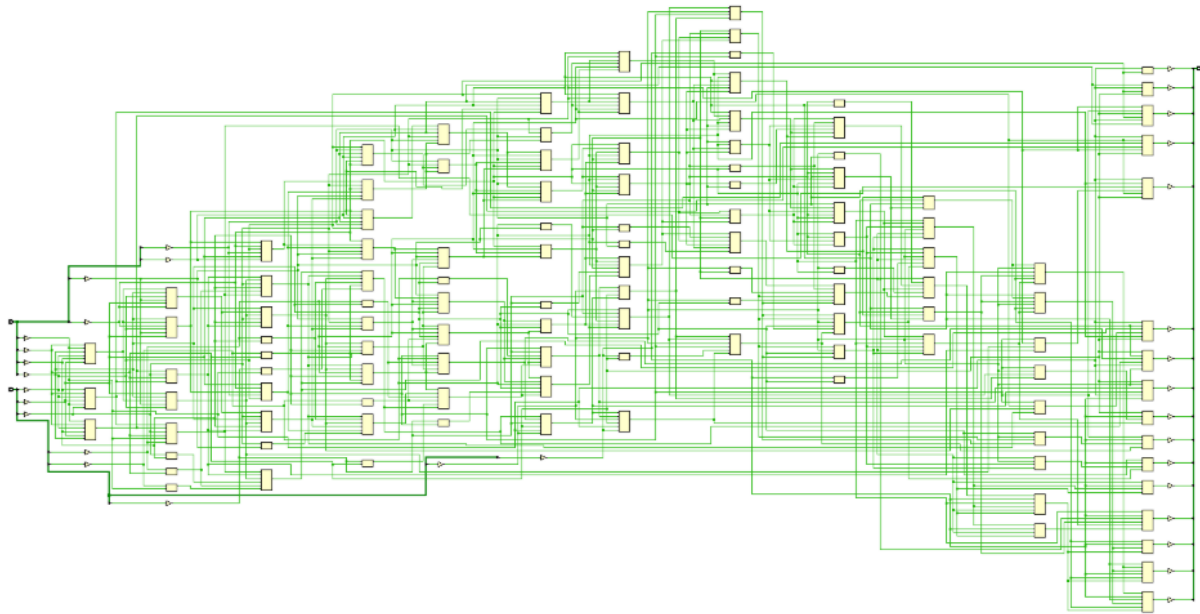


Figure 13 – 8x8 Braun's multiplier implementation schematic.

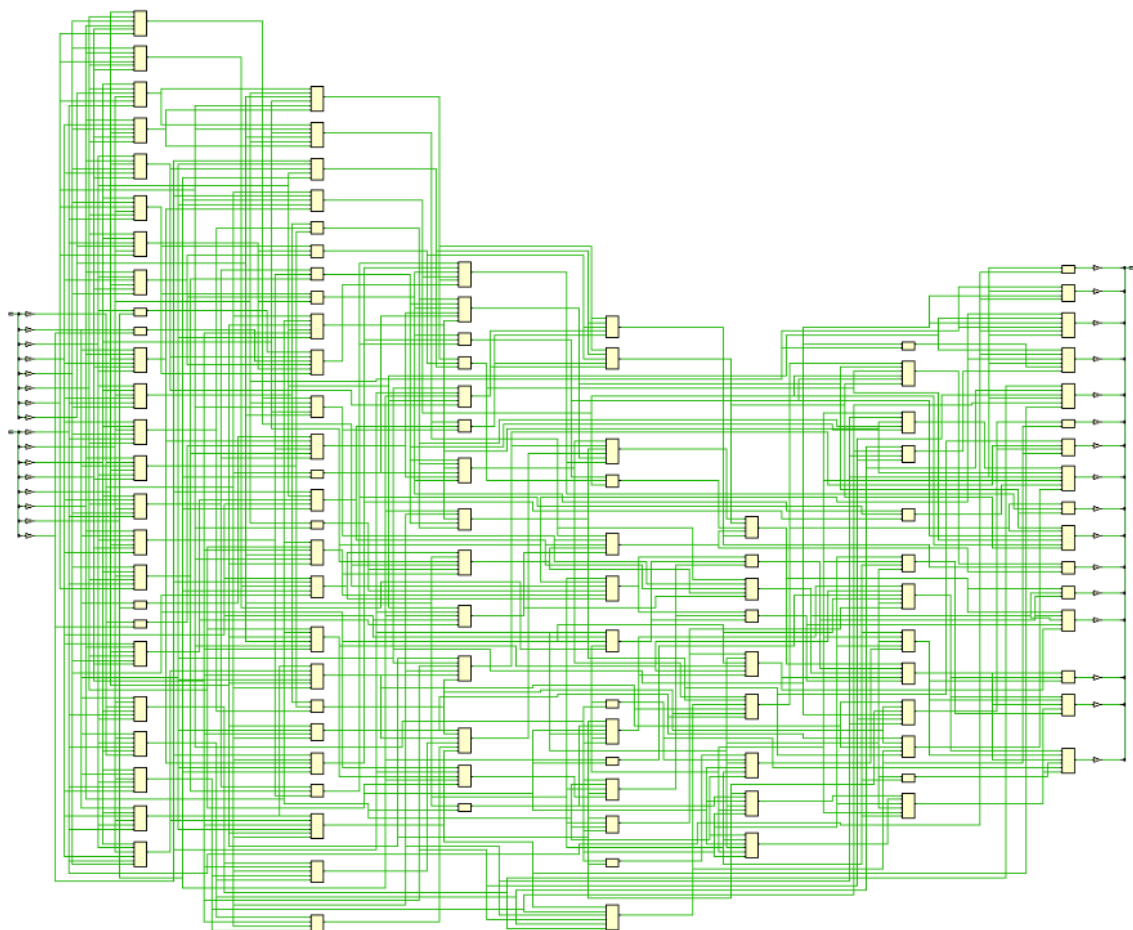


Figure 14 - 8x8 Wallace tree implementation schematic.

5. FPGA Design

In terms of the FPGA design of the 8x8 Braun's multiplier, Figure 15 shows the schematic of devices and Figure 16 shows the I/O planning schematic.

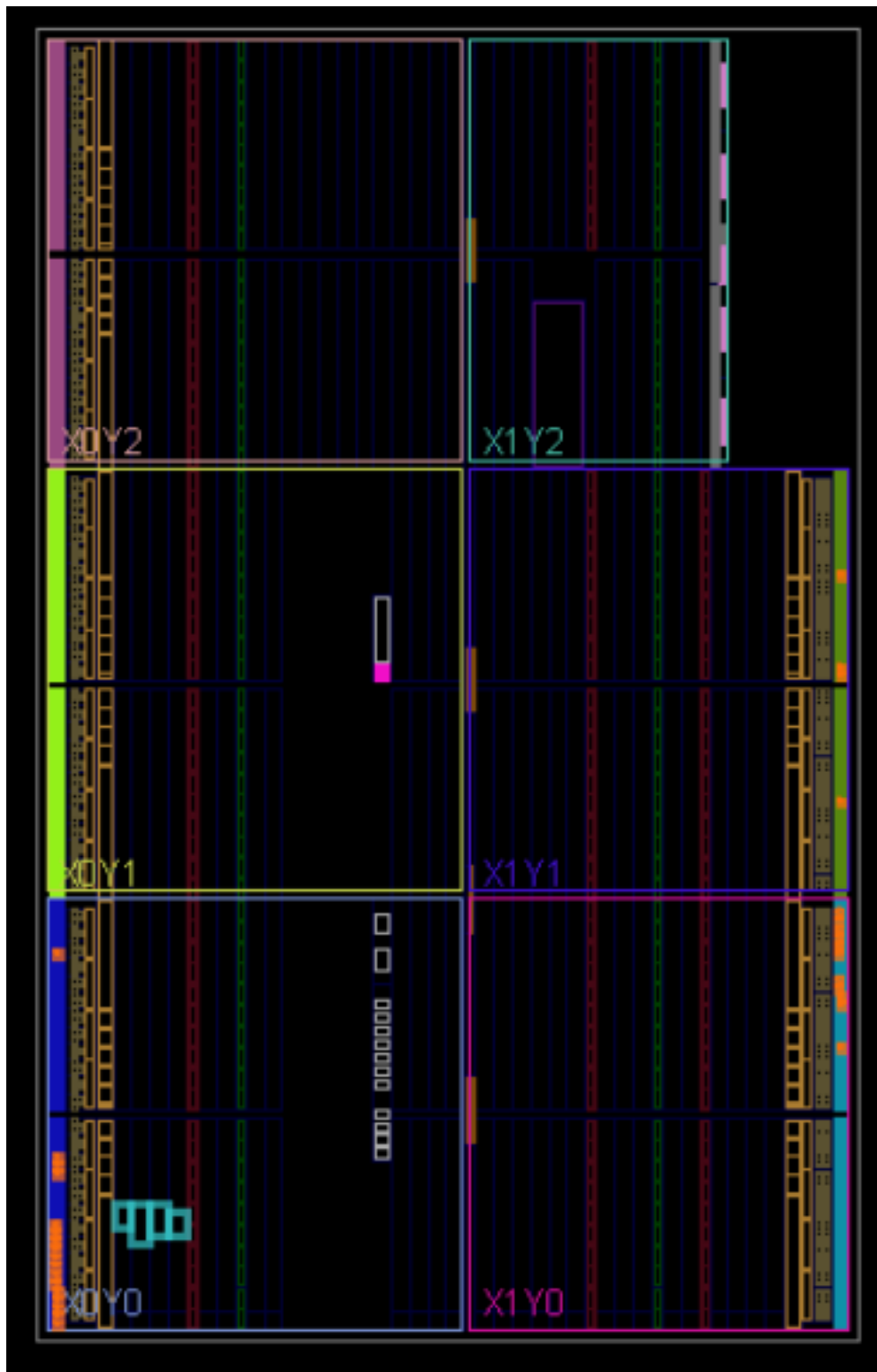


Figure 15 - 8x8 Braun's multiplier schematic of devices

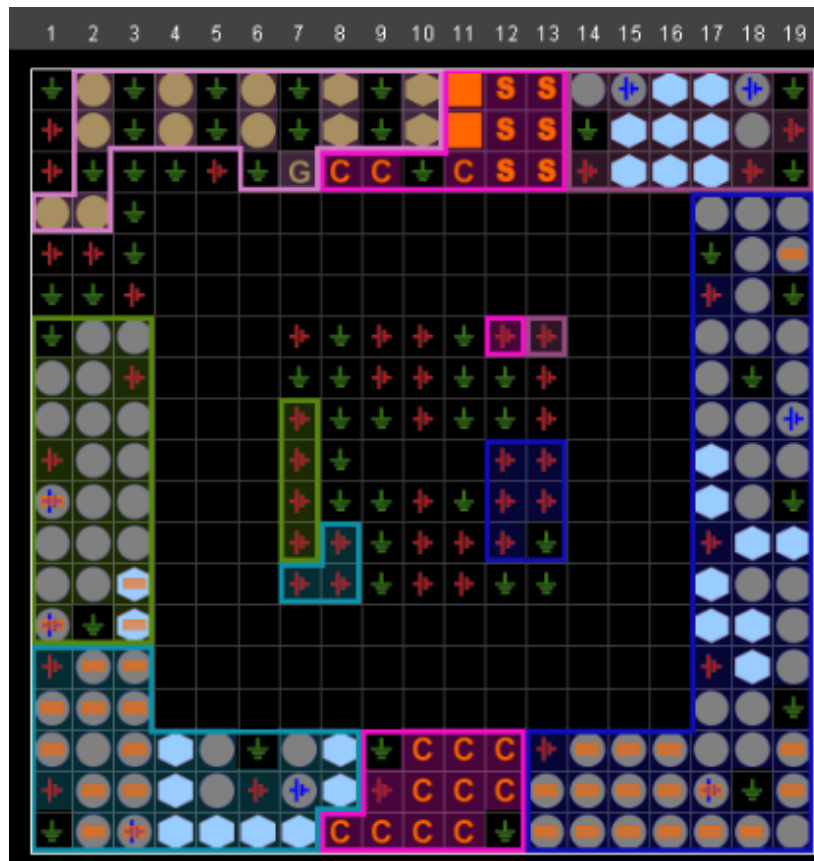


Figure 16 - 8x8 Braun's multiplier I/O planning schematic.

For the Wallace tree multiplier, the schematic of devices is shown in Figure 17 with the I/O planning schematic shown in Figure 18.

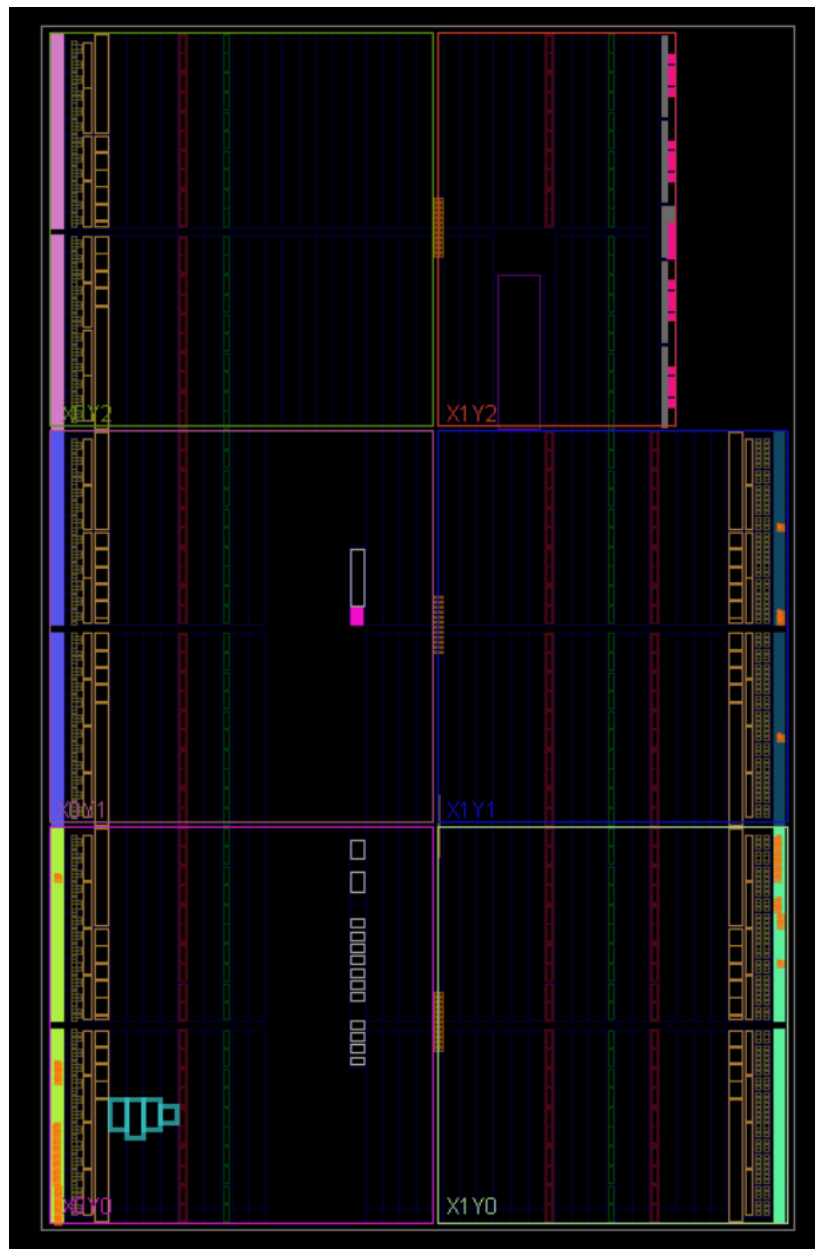


Figure 17 - 8x8 Wallace tree multiplier schematic of devices.

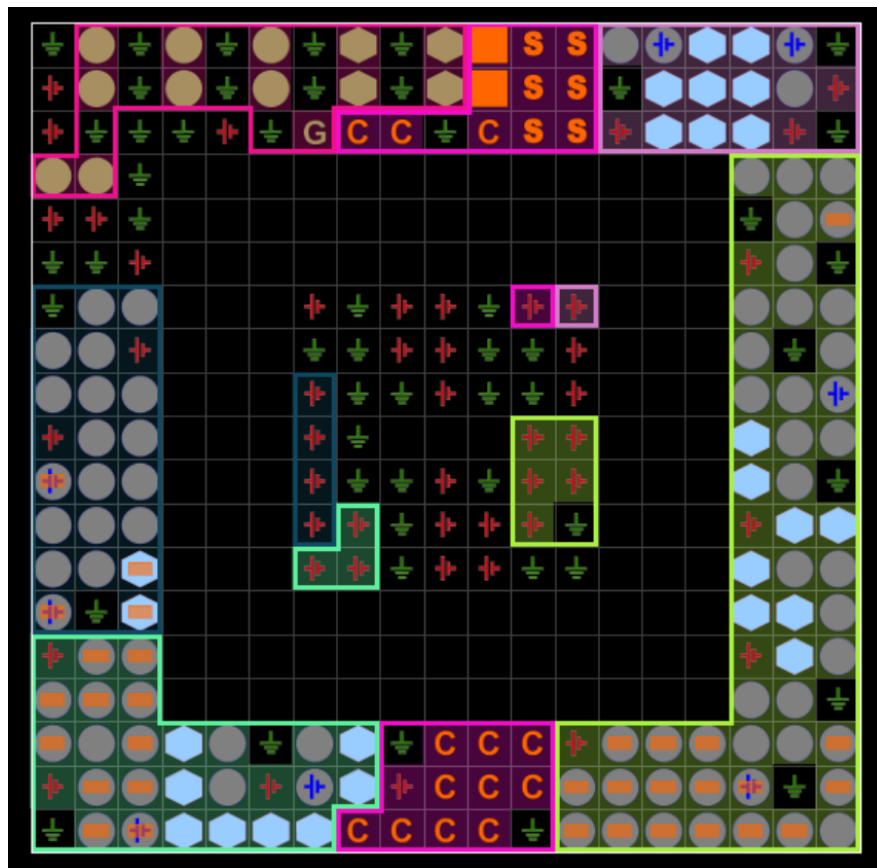
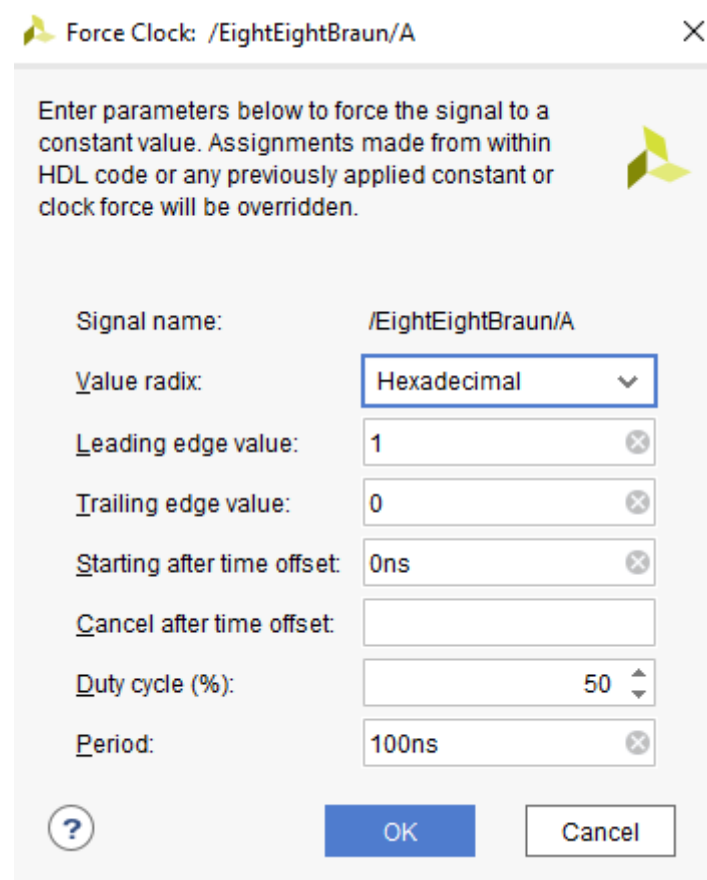


Figure 18 - 8x8 Wallace tree multiplier I/O planning schematic.

6. Simulation results

This section focuses on the simulation results of both multipliers. The simulations were conducted within the Vivado software. Under the 'Force Clock' section, both A and B were forced to have a leading edge of 1 and a trailing edge of 0 (as shown in Figure 19) . From this point, the 'Force Constant' tool was used to input different values on A and B to fully test the capabilities of the multipliers. Both multipliers were put under four simulation examples, with the results shown in Figures 20 to 27:

- Low example – 15 x 10
- Medium example – 9F x 7E
- High example – DE x EF
- Final example – FF x FF



Force Clock: /EightEightBraun/A

Enter parameters below to force the signal to a constant value. Assignments made from within HDL code or any previously applied constant or clock force will be overridden.

Signal name: /EightEightBraun/A

Value radix: Hexadecimal

Leading edge value: 1

Trailing edge value: 0

Starting after time offset: 0ns

Cancel after time offset:

Duty cycle (%): 50

Period: 100ns

OK Cancel

Figure 19 - 'Force Clock' section of simulation tool.

6.1 8x8 Braun's multiplier simulation results

- Low example – 15 x 10

Name	Value	1,049,995 ps	1,049,996 ps	1,049,997 ps	1,049,998 ps	1,049,999 ps
> A[7:0]	15	15				
> B[7:0]	10	10				
> P[15:0]	0150	0150				
A0B0	0					
A1B0	0					
A2B0	0					
A3B0	0					
A4B0	0					
A5B0	0					
A6B0	0					
A7B0	0					
A0B1	0					
A1B1	0					
A2B1	0					
A3B1	0					
A4B1	0					
A5B1	0					
A6B1	0					
A7B1	0					
A0B2	0					
A1B2	0					
A2B2	0					

Figure 20 - 8x8 Braun's multiplier low example simulation result.

- Medium example – 9F x 7E

Name	Value	1,049,995 ps	1,049,996 ps	1,049,997 ps	1,049,998 ps	1,049,999 ps
> A[7:0]	9f	9f				
> B[7:0]	7e	7e				
> P[15:0]	4e42	4e42				
A0B0	0					
A1B0	0					
A2B0	0					
A3B0	0					
A4B0	0					
A5B0	0					
A6B0	0					
A7B0	0					
A0B1	1					
A1B1	1					
A2B1	1					
A3B1	1					
A4B1	1					
A5B1	0					
A6B1	0					
A7B1	1					
A0B2	1					
A1B2	1					
A2B2	1					

Figure 21 - 8x8 Braun's multiplier medium example simulation result.

- High example – DE x EF

Name	Value	1,049,995 ps	1,049,996 ps	1,049,997 ps	1,049,998 ps	1,049,999 ps
> A[7:0]	de					
> B[7:0]	ef					
> P[15:0]	cf42					
A0B0	0					
A1B0	1					
A2B0	1					
A3B0	1					
A4B0	1					
A5B0	0					
A6B0	1					
A7B0	1					
A0B1	0					
A1B1	1					
A2B1	1					
A3B1	1					
A4B1	1					
A5B1	0					
A6B1	1					
A7B1	1					
A0B2	0					
A1B2	1					
A2B2	1					

Figure 22 - 8x8 Braun's multiplier high example simulation result.

- 4th example – FF x FF

Name	Value	1,049,995 ps	1,049,996 ps	1,049,997 ps	1,049,998 ps	1,049,999 ps
> A[7:0]	ff					
> B[7:0]	ff					
> P[15:0]	fe01					
A0B0	1					
A1B0	1					
A2B0	1					
A3B0	1					
A4B0	1					
A5B0	1					
A6B0	1					
A7B0	1					
A0B1	1					
A1B1	1					
A2B1	1					
A3B1	1					
A4B1	1					
A5B1	1					
A6B1	1					
A7B1	1					
A0B2	1					
A1B2	1					
A2B2	1					

Figure 23 - 8x8 Braun's multiplier final example simulation result.

6.2 8x8 Wallace tree multiplier simulation results

- Low example – 15 x 10

Name	Value	1,049,994 ps	1,049,995 ps	1,049,996 ps	1,049,997 ps	1,049,998 ps	1,049,999 ps
> A[7:0]	15	15					
> B[7:0]	10	10					
> P[15:0]	0150	0150					
A0B0	0						
A1B0	0						
A2B0	0						
A3B0	0						
A4B0	0						
A5B0	0						
A6B0	0						
A7B0	0						
A0B1	0						
A1B1	0						
A2B1	0						
A3B1	0						
A4B1	0						
A5B1	0						
A6B1	0						
A7B1	0						
A0B2	0						
A1B2	0						
A2B2	0						

Figure 24 - 8x8 Wallace tree multiplier low example simulation result.

- Medium Example – 9F x 7E

Name	Value	1,049,995 ps	1,049,996 ps	1,049,997 ps	1,049,998 ps	1,049,999 ps
> A[7:0]	9f	9f				
> B[7:0]	7e	7e				
> P[15:0]	4e42	4e42				
A0B0	0					
A1B0	0					
A2B0	0					
A3B0	0					
A4B0	0					
A5B0	0					
A6B0	0					
A7B0	0					
A0B1	1					
A1B1	1					
A2B1	1					
A3B1	1					
A4B1	1					
A5B1	0					
A6B1	0					
A7B1	1					
A0B2	1					
A1B2	1					
A2B2	1					

Figure 25 - 8x8 Wallace tree multiplier medium example simulation result.

- High Example – DE x EF

Name	Value	1,049,995 ps	1,049,996 ps	1,049,997 ps	1,049,998 ps	1,049,999 ps
> A[7:0]	de	de				
> B[7:0]	ef	ef				
> P[15:0]	cf42	cf42				
A0B0	0					
A1B0	1					
A2B0	1					
A3B0	1					
A4B0	1					
A5B0	0					
A6B0	1					
A7B0	1					
A0B1	0					
A1B1	1					
A2B1	1					
A3B1	1					
A4B1	1					
A5B1	0					
A6B1	1					
A7B1	1					
A0B2	0					
A1B2	1					
A2B2	1					

Figure 26 - 8x8 Wallace tree multiplier high example simulation result.

- Final Example – FF x FF

Name	Value	1,049,995 ps	1,049,996 ps	1,049,997 ps	1,049,998 ps	1,049,999 ps
> A[7:0]	ff	ff				
> B[7:0]	ff	ff				
> P[15:0]	fe01	fe01				
A0B0	1					
A1B0	1					
A2B0	1					
A3B0	1					
A4B0	1					
A5B0	1					
A6B0	1					
A7B0	1					
A0B1	1					
A1B1	1					
A2B1	1					
A3B1	1					
A4B1	1					
A5B1	1					
A6B1	1					
A7B1	1					
A0B2	1					
A1B2	1					
A2B2	1					

Figure 27 - 8x8 Wallace tree multiplier final example simulation result.

From the simulation examples, the Braun's multiplier achieved the following results shown in Table 1:

Table 1 – 8x8 Braun's multiplier simulation results summary.

Example	Value of A	Value of B	Result
Low	15	10	0150
Medium	9F	7E	4E42
High	DE	EF	CF42
Final	FF	FF	FE01

Similarly, the results from the Wallace tree multiplier's simulations can be seen in Table 2.

Table 2 – 8x8 Wallace tree multiplier simulation results summary.

Example	Value of A	Value of B	Result
Low	15	10	0150
Medium	9F	7E	4E42
High	DE	EF	CF42
Final	FF	FF	FE01

6.3 Comparison, analysis and interpretation of results

Functional testing was carried out on both the Braun's multiplier and the Wallace tree multiplier using the Vivado simulation environment. A range of input constants were forced to validate the function of the multipliers across low, medium and high operand values. From the simulation results, it is shown that both multipliers function as expected and give the desired output. Appendix B (Figures 42 to 49) show the FPGA results of these simulations and all of the results were validated using a hexadecimal calculator, seen in Figure 28. The FPGA results confirm that both multipliers behave consistently in hardware, but the internal operation of each multiplier differs, leading to the Wallace tree multiplier providing better scalability and timing performance.

The expected FPGA results can be seen in Table 3 below:

Table 3 – Expected 8x8 simulation results.

Value of A	Value of B	Expected Binary Result
15	10	0000 0000 1001 0110
9F	7E	0100 1110 0100 0010
DE	EF	1100 1111 0100 0010
FF	FF	1111 1111 0000 0001

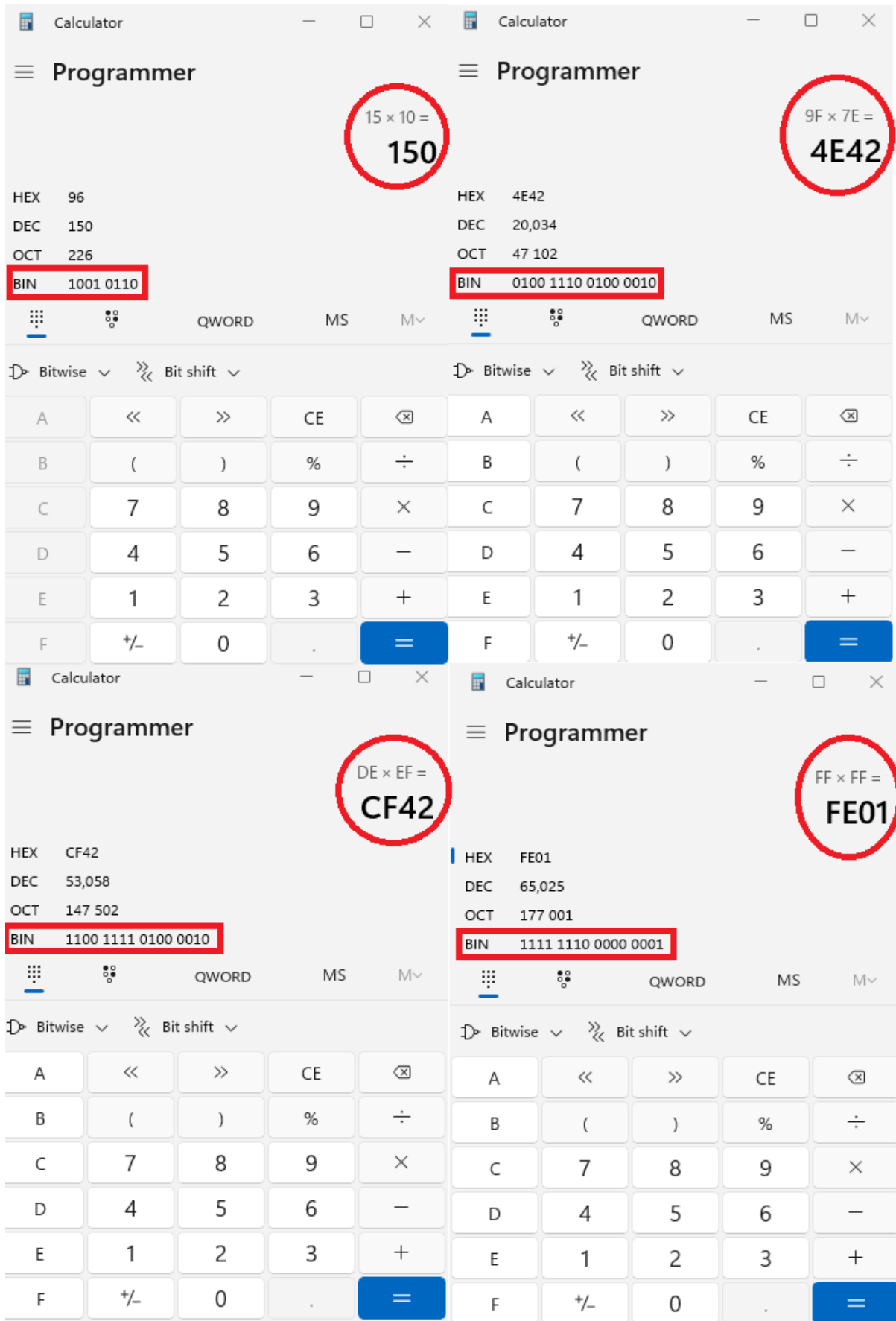


Figure 28 – Verification of simulation results using calculator.

7. Interpretation of utilisation and power reports

7.1 Utilisation Reports

In terms of the utilisation reports, Figure 29 shows the report of the Braun's multiplier and Figure 30 shows the report of the Wallace tree multiplier. The Braun's multiplier used 92 LUTs (Look-Up Tables) and the Wallace tree multiplier used 91 LUTs out of an available 20800, coming in at 0.44% usage. This meant that the logic used in the multipliers synthesizes very efficiently and there is room for further scaling (e.g. 16x16, 32x32 versions). Both multipliers used 32 input/output pins out of an available 106 (a 30.19% usage).

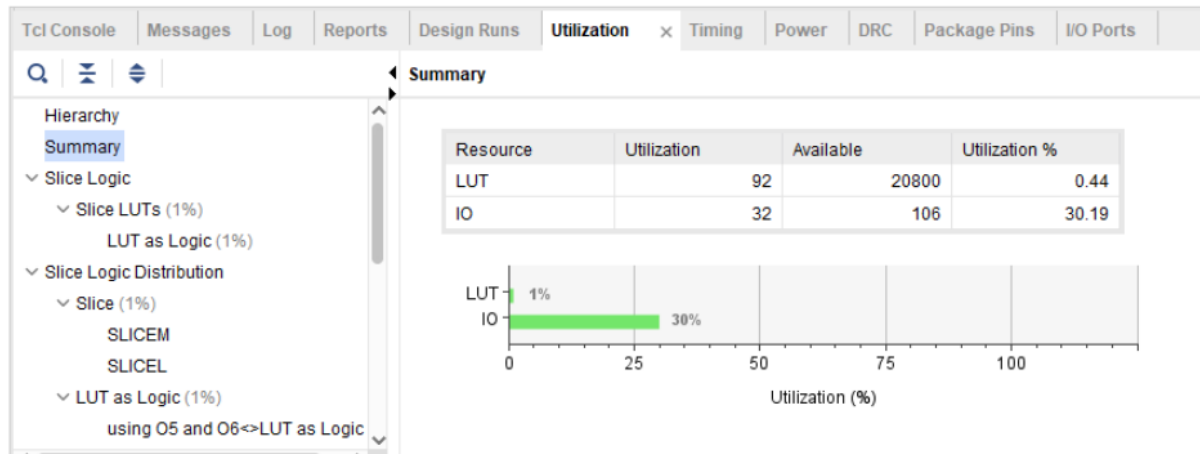


Figure 29 – 8x8 Braun's multiplier utilisation report.

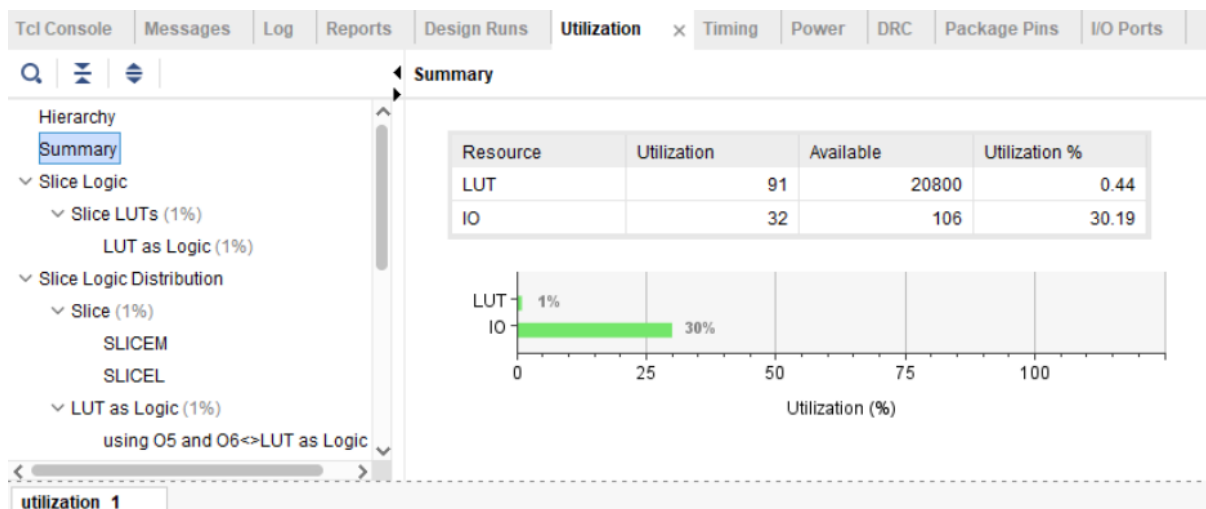


Figure 30 - 8x8 Wallace tree utilisation report.

7.2 Power Reports

Overall, both the synthesis and implementation power reports demonstrate that the 8x8 Braun's multiplier operates comfortably within the allocated 0.25W power budget. Figure 31 and Figure 32 show the power reports of both multipliers at the synthesis and implementation stages. The minimal difference between the two stages shows the suitability of the architecture for low power FPGA applications.

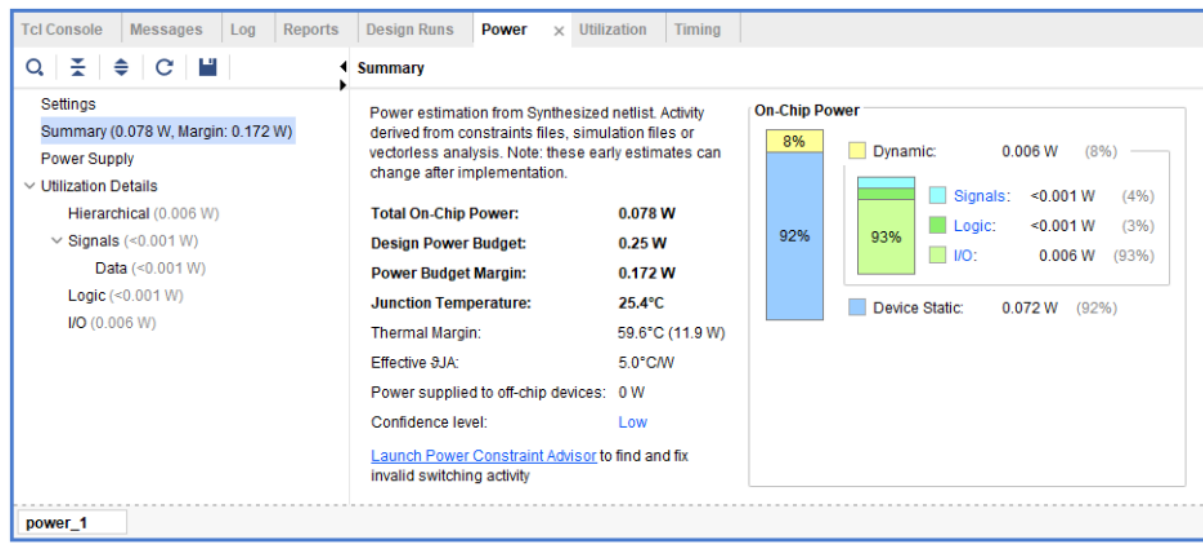


Figure 31 – 8x8 Braun's multiplier power report after synthesis.

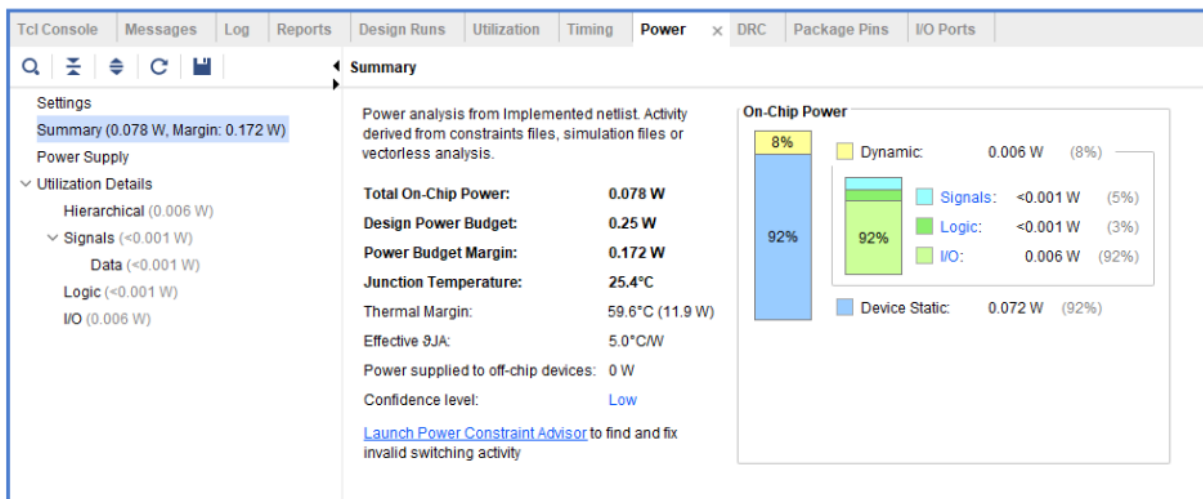


Figure 32 – 8x8 Braun's multiplier power report after implementation.

In terms of the 8x8 Wallace tree multiplier, the reports at the two stages can be seen in Figure 33 and Figure 34. The total on-chip power stays at 0.078W and similarly to the Braun's multiplier, most of the power consumption is down to the device static power. The Wallace tree multiplier is shown to operate comfortably within the power budget and shows the increased logic to achieve higher speeds does not introduce a significant power increase.

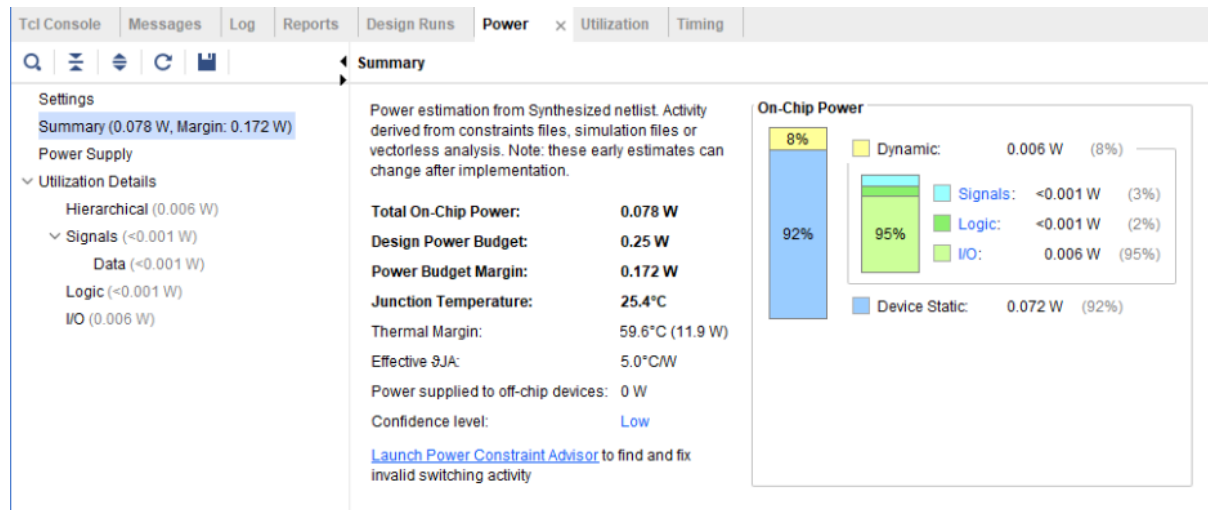


Figure 33 – 8x8 Wallace tree multiplier power report after synthesis.

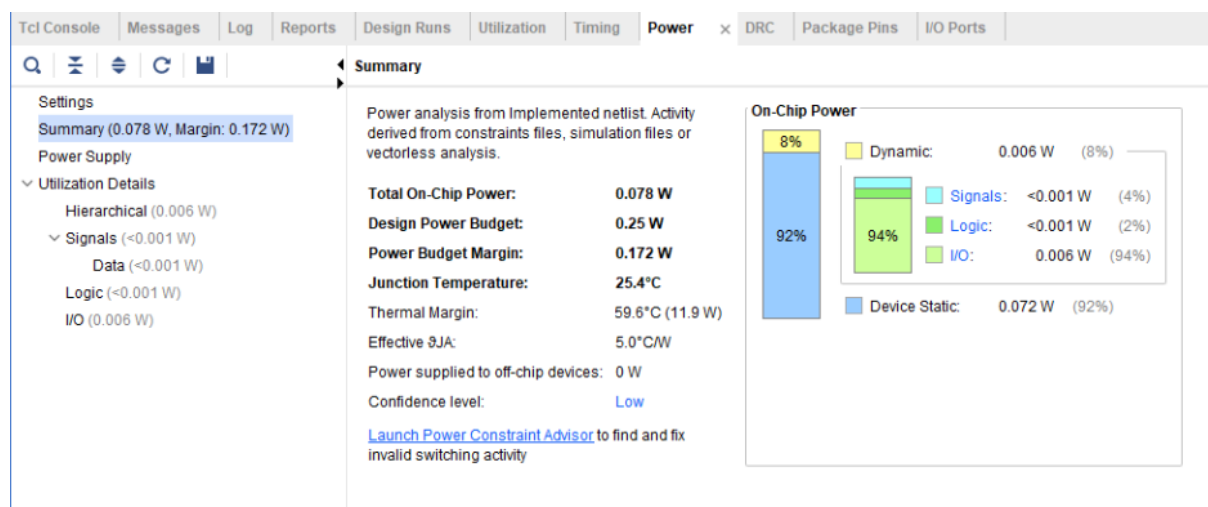


Figure 34 – 8x8 Wallace tree multiplier power report after implementation.

8. Conclusion and recommendations for future improvements

This report presented the design, simulation and evaluation of the 8x8 Braun's and Wallace tree parallel multiplier architectures. These architectures were implemented using VHDL and targeted to an Artix-7 FPGA using Xilinx Vivado. The Braun's multiplier had fewer carry and sum signals compared to the Wallace tree multiplier, but both architectures were verified through simulation and produced correct 16-bit outputs through simulation and on the FPGA board across a range of different test examples.

The Braun's multiplier demonstrated a simple and regular structure, with a predictable routing style and a basic implementation but did suffer from a longer critical path due to the ripple-carry propagation [1, 5]. On the other hand, the Wallace tree multiplier utilised carry-save reduction to minimise sequential adder stages [6]. This allowed the Wallace tree multiplier to be faster whilst still remaining in the specified power and utilisation ranges [9].

In terms of future work and improvements, expanding the designs to larger systems (e.g. 16x16 or 32x32) would allow more investigations of scalability and performance under increased complexity. There is also the possibility of replacing standard full adders with higher order compressors to allow more bits to be combined in a single reduction stage. This would reduce the number of reduction levels required, which will then shorten the critical path and improve operating speeds [14]. Another example of a future improvement is the use of booth encoding, which reduces the number of generated partial products [17]. This is done by encoding groups of bits together instead of producing one partial product per bit. The use of Booth encoding would shorten the critical path, and also improve speed and power efficiency [17].

Overall, the completed work shows that whilst Braun's multipliers are well suited for simple, low-power applications, Wallace tree multipliers provide a better performance for high-speed FPGA and DSP based systems. The multiplier choice should be made based on the speed, power and complexity requirements of the multiplier's application.

Bibliography

- [1] H. Chung, "Designing of Various types of multipliers and their implementation on FPGA," *Journal of Emerging Technologies and Innovative Research (JETIR)*, vol. 6, no. 6, pp. 1066-1070, 2019.
- [2] S. More, S. Shirol, A. Bandiwad and N. G. M, in *International Conference on Knowledge Engineering and Information Systems (KEIS-2025)*, Tumakuru, India,, 2025.
- [3] A. R. Bagyaveereswaran, "Braun's Multiplier Implementation using FPGA with Bypassing Techniques.," *International Journal of VLSI design & Communication Systems (VLSICS)*, vol. 2, no. 3, pp. 201-212, 2011.
- [4] S. Immareddy and A. Sundaramoorthy , "A survey paper on design and implementation of multipliers for digital system applications," *Artificial Intelligence Review*, vol. 55, pp. 4575-4603, 2022.
- [5] H. M. Brunda, H. Jayalaxmi, H. Arpitha, A. R. Patil and G. S. Manjushree, "Comparative Analysis of Various Multipliers Based on Performance," *Technology, Science and Society.*, vol. 2, pp. 100-108, 2025.
- [6] C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers* , Vols. EC-13, no. 1, pp. 14-17, 1964.
- [7] H. Jiang, C. Liu, N. Maheshwari, F. Lombardi and J. Han, "A comparative evaluation of approximate multipliers," in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, Beijing, China, 2016.
- [8] O. Gustafsson, A. G. Dempster and L. Wanhammar, "Multiplier blocks using carry-save adders," in *2004 IEEE International Symposium on Circuits and Systems*, Vancouver, Canada, 2004.
- [9] S. Abed, Y. Khalil, M. Modhaffar and I. Ahmad, "High-performance low-power approximate Wallace Tree multiplier," *International Journal of Circuit Theory and Applications*, vol. 46, no. 12, pp. 2334-2348, 2018.
- [10] R. Shanmuganathan and K. Brindhadevi, "Comparative analysis of various types of multipliers for effective low power," *Microelectronic Engineering*, vol. 214, pp. 28-37, 2019.
- [11] P. Mishra and S. Nayak, "A study on the Wallace tree multiplier," *International Journal of Advanced Research in Science and Engineering*, vol. 7, no. 5, pp. 100-105, 2018.

- [12] W. J. Townsend, E. E. Swartzlander and J. A. Abraham, "A comparison of Dadda and Wallace multiplier delays," in *SPIE 5205, Advanced Signal Processing Algorithms, Architectures, and Implementations XIII*, California, United States, 2003.
- [13] K. Bickerstaff, M. J. Shute and E. E. Swartzlander, "Reduced area multipliers," in *Proceedings of International Conference on Application Specific Array Processors (ASAP '93)*, Venice, Italy, 1993.
- [14] V. G. Oklobdzija, D. Villeger and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294-306, 1996.
- [15] S. Nagaleela, G. Shanthi, B. Manisha, P. Bharath and E. Praneeth, "Design of a DADDA Multiplier Using High Performance and Low Power Full Adder," in *14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, Delhi, India, 2023.
- [16] K. Bhardwaj, P. S. Mane and J. Henkel, "Power- and area-efficient Approximate Wallace Tree Multiplier for error-resilient systems," in *Fifteenth International Symposium on Quality Electronic Design*, California, USA, 2014.
- [17] W.-C. Yeh and C.-W. Jen, "High-speed Booth encoded parallel multiplier design," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 692-701, 2000.

Appendix

Appendix A. VHDL Codes

VHDL Code explanation

The first stage of the program imports standard logic types (std_logic_vector) to define the two 8-bit unsigned inputs A and B, as well as the 16-bit output P. Figure 35 shows the entity declaration stage that both multipliers follow, just with different entity names to suit the multiplier e.g. the Wallace tree multiplier has the entity name 'EightEightWallace'

```
entity EightEightBraun 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 EightEightBraun;
```

Figure 35 – 8x8 Braun's multiplier entity declaration.

The programs then declare the FullAdder and HalfAdder components, as shown in Figure 36. The half adder adds two bits to produce a sum and a carry. The full adder adds three bits, with the carry-in bit from a previous adder being considered as well as the two input bits.

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

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

Figure 36 – Half adder and full adder components.

A series of AND gates are used to be able to generate partial products, as shown in Figure 37. Each A_iB_j signal corresponds to $A(i)$ multiplied by $B(j)$, and these continue from $B(0)$ to $B(7)$.

```
-- B(0) row
A0B0 <= A(0) and B(0);
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);

-- B(1) row
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);

-- B(2) row
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);
```

Figure 37 – Partial product generation through AND gates.

In terms of carry signals, they are in a PxCy format where x is the output bit position and y is the carry index within that column e.g. P2C1. These are shown in Figure 38, and can be matched to the block diagram in Figure 3.

```
--P2
signal P2C0: std_logic;
signal P2C1: std_logic;

--P3
signal P3C0: std_logic;
signal P3C1: std_logic;
signal P3C2: std_logic;

--P4
signal P4C0: std_logic;
signal P4C1: std_logic;
signal P4C2: std_logic;
signal P4C3: std_logic;
```

Figure 38 – Carry signals declarations example.

The sum signals are intermediate sums, passed from one adder to the next within the same vertical column e.g. S2_1. These sum signals are shown in Figure 39.

```
--- intermediate signals
signal S2_0 : std_logic;

signal S3_0 : std_logic;
signal S3_1 : std_logic;

signal S4_0 : std_logic;
signal S4_1 : std_logic;
signal S4_2 : std_logic;

signal S5_0 : std_logic;
signal S5_1 : std_logic;
signal S5_2 : std_logic;
signal S5_3 : std_logic;
```

Figure 39 – Sum signals declaration.

The full adder and half adder components are then used to work through the Braun's multiplier and give the outputs P(0) to P(15). The half adder functions by assigning partial products to A and B, and then stating the sum signals and the carry-out. An example of the use of a half adder is seen in Figure 40.

```
-- P(1)
HA1: HalfAdder port map(A=>A1B0, B=>A0B1, Sout=>P(1), Cout=>P1C0);
```

Figure 40 – Half adder component in use

The full adder functions by also assigning partial products to A and B, but also considers a carry-in from a previous adder. The full adder also produces a sum signal and a carry-out, as shown in Figure 41. This process continues until a 16-bit output is produced.

```
FA2_1: FullAdder port map(A=>A2B0, B=>A1B1, Cin=>P1C0, Sout=>S2_0, Cout=>P2C0);
```

Figure 41 – Full adder component in use.

Full 8x8 Braun's multiplier VHDL code:

```
22 | library IEEE;
23 | use IEEE.STD_LOGIC_1164.ALL;
24 |
25 | -- Uncomment the following library declaration if using
26 | -- arithmetic functions with Signed or Unsigned values
27 | --use IEEE.NUMERIC_STD.ALL;
28 |
29 | -- Uncomment the following library declaration if instantiating
30 | -- any Xilinx leaf cells in this code.
31 | --library UNISIM;
32 | --use UNISIM.VComponents.all;
33 |
34 | entity EightEightBraun is
35 |     Port (
36 |         A : in STD_LOGIC_VECTOR (7 downto 0) ;
37 |         B : in STD_LOGIC_VECTOR (7 downto 0);
38 |         P : out STD_LOGIC_VECTOR (15 downto 0));
39 | end EightEightBraun;
40 |
41 | architecture Behavioral of EightEightBraun is
42 |
43 |     component FullAdder
44 |     port (
45 |         A : in STD_LOGIC;
46 |         B : in STD_LOGIC;
47 |         Cin : in STD_LOGIC;
48 |         Sout : out STD_LOGIC;
49 |         Cout : out STD_LOGIC);
50 | end component;
51 |
52 |     component HalfAdder
53 |     port (
54 |         A : in STD_LOGIC;
55 |         B : in STD_LOGIC;
56 |         Sout : out STD_LOGIC;
57 |         Cout : out STD_LOGIC);
58 | end component;
59 |
60 |     signal A0B0,A1B0,A2B0,A3B0,A4B0,A5B0,A6B0,A7B0 : std_logic;
61 |     signal A0B1,A1B1,A2B1,A3B1,A4B1,A5B1,A6B1,A7B1 : std_logic;
62 |     signal A0B2,A1B2,A2B2,A3B2,A4B2,A5B2,A6B2,A7B2 : std_logic;
63 |     signal A0B3,A1B3,A2B3,A3B3,A4B3,A5B3,A6B3,A7B3 : std_logic;
64 |     signal A0B4,A1B4,A2B4,A3B4,A4B4,A5B4,A6B4,A7B4 : std_logic;
65 |     signal A0B5,A1B5,A2B5,A3B5,A4B5,A5B5,A6B5,A7B5 : std_logic;
66 |     signal A0B6,A1B6,A2B6,A3B6,A4B6,A5B6,A6B6,A7B6 : std_logic;
67 |     signal A0B7,A1B7,A2B7,A3B7,A4B7,A5B7,A6B7,A7B7 : std_logic;
```

```

68 |
69 | --Carry
70 | --P1
71 | signal P1C0: std_logic;
72 |
73 | --P2
74 | signal P2C0: std_logic;
75 | signal P2C1: std_logic;
76 |
77 | --P3
78 | signal P3C0: std_logic;
79 | signal P3C1: std_logic;
80 | signal P3C2: std_logic;
81 |
82 | --P4
83 | signal P4C0: std_logic;
84 | signal P4C1: std_logic;
85 | signal P4C2: std_logic;
86 | signal P4C3: std_logic;
87 |
88 | --P5
89 | signal P5C0: std_logic;
90 | signal P5C1: std_logic;
91 | signal P5C2: std_logic;
92 | signal P5C3: std_logic;
93 | signal P5C4: std_logic;
94 |
95 | --P6
96 | signal P6C0: std_logic;
97 | signal P6C1: std_logic;
98 | signal P6C2: std_logic;
99 | signal P6C3: std_logic;
100 | signal P6C4: std_logic;
101 | signal P6C5: std_logic;
102 |
103 | --P7
104 | signal P7C0: std_logic;
105 | signal P7C1: std_logic;
106 | signal P7C2: std_logic;
107 | signal P7C3: std_logic;
108 | signal P7C4: std_logic;
109 | signal P7C5: std_logic;
110 | signal P7C6: std_logic;
111 |
112 | --P8
113 | signal P8C0: std_logic;
114 | signal P8C1: std_logic;
115 | signal P8C2: std_logic;
116 | signal P8C3: std_logic;
117 | signal P8C4: std_logic;
118 | signal P8C5: std_logic;
119 | signal P8C6: std_logic;

```

```
120 :
121 : --P9
122 : signal P9C0: std_logic;
123 : signal P9C1: std_logic;
124 : signal P9C2: std_logic;
125 : signal P9C3: std_logic;
126 : signal P9C4: std_logic;
127 : signal P9C5: std_logic;
128 :
129 : --P10
130 : signal P10C0: std_logic;
131 : signal P10C1: std_logic;
132 : signal P10C2: std_logic;
133 : signal P10C3: std_logic;
134 : signal P10C4: std_logic;
135 :
136 : --P11
137 : signal P11C0: std_logic;
138 : signal P11C1: std_logic;
139 : signal P11C2: std_logic;
140 : signal P11C3: std_logic;
141 :
142 : --P12
143 : signal P12C0: std_logic;
144 : signal P12C1: std_logic;
145 : signal P12C2: std_logic;
146 :
147 : --P13
148 : signal P13C0: std_logic;
149 : signal P13C1: std_logic;
```

```
152  --- intermediate signals
153  signal S2_0 : std_logic;
154
155  signal S3_0 : std_logic;
156  signal S3_1 : std_logic;
157
158  signal S4_0 : std_logic;
159  signal S4_1 : std_logic;
160  signal S4_2 : std_logic;
161
162  signal S5_0 : std_logic;
163  signal S5_1 : std_logic;
164  signal S5_2 : std_logic;
165  signal S5_3 : std_logic;
166
167  signal S6_0 : std_logic;
168  signal S6_1 : std_logic;
169  signal S6_2 : std_logic;
170  signal S6_3 : std_logic;
171  signal S6_4 : std_logic;
172
173  signal S7_0 : std_logic;
174  signal S7_1 : std_logic;
175  signal S7_2 : std_logic;
176  signal S7_3 : std_logic;
177  signal S7_4 : std_logic;
178  signal S7_5 : std_logic;
179
180  signal S8_0 : std_logic;
181  signal S8_1 : std_logic;
182  signal S8_2 : std_logic;
183  signal S8_3 : std_logic;
184  signal S8_4 : std_logic;
185  signal S8_5 : std_logic;
186
187  signal S9_0 : std_logic;
188  signal S9_1 : std_logic;
189  signal S9_2 : std_logic;
190  signal S9_3 : std_logic;
191  signal S9_4 : std_logic;
192
193  signal S10_0 : std_logic;
194  signal S10_1 : std_logic;
195  signal S10_2 : std_logic;
196  signal S10_3 : std_logic;
197
198  signal S11_0 : std_logic;
199  signal S11_1 : std_logic;
200  signal S11_2 : std_logic;
201
202  signal S12_0 : std_logic;
203  signal S12_1 : std_logic;
```

```

204 signal S13_0 : std_logic;
205
206 begin
207
208     -- B(0) row
209     A0B0 <= A(0) and B(0);
210     A1B0 <= A(1) and B(0);
211     A2B0 <= A(2) and B(0);
212     A3B0 <= A(3) and B(0);
213     A4B0 <= A(4) and B(0);
214     A5B0 <= A(5) and B(0);
215     A6B0 <= A(6) and B(0);
216     A7B0 <= A(7) and B(0);
217
218     -- B(1) row
219     A0B1 <= A(0) and B(1);
220     A1B1 <= A(1) and B(1);
221     A2B1 <= A(2) and B(1);
222     A3B1 <= A(3) and B(1);
223     A4B1 <= A(4) and B(1);
224     A5B1 <= A(5) and B(1);
225     A6B1 <= A(6) and B(1);
226     A7B1 <= A(7) and B(1);
227
228     -- B(2) row
229     A0B2 <= A(0) and B(2);
230     A1B2 <= A(1) and B(2);
231     A2B2 <= A(2) and B(2);
232     A3B2 <= A(3) and B(2);
233     A4B2 <= A(4) and B(2);
234     A5B2 <= A(5) and B(2);
235     A6B2 <= A(6) and B(2);
236     A7B2 <= A(7) and B(2);
237
238     -- B(3) row
239     A0B3 <= A(0) and B(3);
240     A1B3 <= A(1) and B(3);
241     A2B3 <= A(2) and B(3);
242     A3B3 <= A(3) and B(3);
243     A4B3 <= A(4) and B(3);
244     A5B3 <= A(5) and B(3);
245     A6B3 <= A(6) and B(3);
246     A7B3 <= A(7) and B(3);

```

```
248      -- B(4) row
249      A0B4 <= A(0) and B(4);
250      A1B4 <= A(1) and B(4);
251      A2B4 <= A(2) and B(4);
252      A3B4 <= A(3) and B(4);
253      A4B4 <= A(4) and B(4);
254      A5B4 <= A(5) and B(4);
255      A6B4 <= A(6) and B(4);
256      A7B4 <= A(7) and B(4);
257
258      -- B(5) row
259      A0B5 <= A(0) and B(5);
260      A1B5 <= A(1) and B(5);
261      A2B5 <= A(2) and B(5);
262      A3B5 <= A(3) and B(5);
263      A4B5 <= A(4) and B(5);
264      A5B5 <= A(5) and B(5);
265      A6B5 <= A(6) and B(5);
266      A7B5 <= A(7) and B(5);
267
268      -- B(6) row
269      A0B6 <= A(0) and B(6);
270      A1B6 <= A(1) and B(6);
271      A2B6 <= A(2) and B(6);
272      A3B6 <= A(3) and B(6);
273      A4B6 <= A(4) and B(6);
274      A5B6 <= A(5) and B(6);
275      A6B6 <= A(6) and B(6);
276      A7B6 <= A(7) and B(6);
277
278      -- B(7) row
279      A0B7 <= A(0) and B(7);
280      A1B7 <= A(1) and B(7);
281      A2B7 <= A(2) and B(7);
282      A3B7 <= A(3) and B(7);
283      A4B7 <= A(4) and B(7);
284      A5B7 <= A(5) and B(7);
285      A6B7 <= A(6) and B(7);
286      A7B7 <= A(7) and B(7);
```

```

288 -- P(0)
289 P(0) <= A0B0;
290
291 -- P(1)
292 HA1: HalfAdder port map(A=>A1B0, B=>A0B1, Sout=>P(1), Cout=>P1C0);
293
294 -- P(2)
295 FA2_1: FullAdder port map(A=>A2B0, B=>A1B1, Cin=>P1C0, Sout=>S2_0, Cout=>P2C0);
296 HA2: HalfAdder port map(A=>A0B2, B=>S2_0, Sout=>P(2), Cout=>P2C1);
297
298 -- P(3)
299 FA3_1: FullAdder port map(A=>A3B0, B=>A2B1, Cin=>P2C0, Sout=>S3_0, Cout=>P3C0);
300 FA3_2: FullAdder port map(A=>A1B2, B=>S3_0, Cin=>P2C1, Sout=>S3_1, Cout=>P3C1);
301 HA3: HalfAdder port map(A=>A0B3, B=>S3_1, Sout=>P(3), Cout=>P3C2);
302
303 -- P(4)
304 FA4_1: FullAdder port map(A=>A4B0, B=>A3B1, Cin=>P3C0, Sout=>S4_0, Cout=>P4C0);
305 FA4_2: FullAdder port map(A=>A2B2, B=>S4_0, Cin=>P3C1, Sout=>S4_1, Cout=>P4C1);
306 FA4_3: FullAdder port map(A=>A1B3, B=>S4_1, Cin=>P3C2, Sout=>S4_2, Cout=>P4C2);
307 HA4: HalfAdder port map(A=>A0B4, B=>S4_2, Sout=>P(4), Cout=>P4C3);
308
309 -- P(5)
310 FA5_1: FullAdder port map(A=>A5B0, B=>A4B1, Cin=>P4C0, Sout=>S5_0, Cout=>P5C0);
311 FA5_2: FullAdder port map(A=>A3B2, B=>S5_0, Cin=>P4C1, Sout=>S5_1, Cout=>P5C1);
312 FA5_3: FullAdder port map(A=>A2B3, B=>S5_1, Cin=>P4C2, Sout=>S5_2, Cout=>P5C2);
313 FA5_4: FullAdder port map(A=>A1B4, B=>S5_2, Cin=>P4C3, Sout=>S5_3, Cout=>P5C3);
314 HA5: HalfAdder port map(A=>A0B5, B=>S5_3, Sout=>P(5), Cout=>P5C4);
315
316 -- P(6)
317 FA6_1: FullAdder port map(A=>A6B0, B=>A5B1, Cin=>P5C0, Sout=>S6_0, Cout=>P6C0);
318 FA6_2: FullAdder port map(A=>A4B2, B=>S6_0, Cin=>P5C1, Sout=>S6_1, Cout=>P6C1);
319 FA6_3: FullAdder port map(A=>A3B3, B=>S6_1, Cin=>P5C2, Sout=>S6_2, Cout=>P6C2);
320 FA6_4: FullAdder port map(A=>A2B4, B=>S6_2, Cin=>P5C3, Sout=>S6_3, Cout=>P6C3);
321 FA6_5: FullAdder port map(A=>A1B5, B=>S6_3, Cin=>P5C4, Sout=>S6_4, Cout=>P6C4);
322 HA6: HalfAdder port map(A=>A0B6, B=>S6_4, Sout=>P(6), Cout=>P6C5);
323
324 -- P(7)
325 FA7_1: FullAdder port map(A=>A7B0, B=>A6B1, Cin=>P6C0, Sout=>S7_0, Cout=>P7C0);
326 FA7_2: FullAdder port map(A=>A5B2, B=>S7_0, Cin=>P6C1, Sout=>S7_1, Cout=>P7C1);
327 FA7_3: FullAdder port map(A=>A4B3, B=>S7_1, Cin=>P6C2, Sout=>S7_2, Cout=>P7C2);
328 FA7_4: FullAdder port map(A=>A3B4, B=>S7_2, Cin=>P6C3, Sout=>S7_3, Cout=>P7C3);
329 FA7_5: FullAdder port map(A=>A2B5, B=>S7_3, Cin=>P6C4, Sout=>S7_4, Cout=>P7C4);
330 FA7_6: FullAdder port map(A=>A1B6, B=>S7_4, Cin=>P6C5, Sout=>S7_5, Cout=>P7C5);
331 HA7: HalfAdder port map(A=>A0B7, B=>S7_5, Sout=>P(7), Cout=>P7C6);
332

```

```

333 -- P(8)
334 HA8: HalfAdder port map(A=>A7B1, B=>P7C0, Sout=>S8_0, Cout=>P8C0);
335 FA8_1: FullAdder port map(A=>A6B2, B=>S8_0, Cin=>P7C1, Sout=>S8_1, Cout=>P8C1);
336 FA8_2: FullAdder port map(A=>A5B3, B=>S8_1, Cin=>P7C2, Sout=>S8_2, Cout=>P8C2);
337 FA8_3: FullAdder port map(A=>A4B4, B=>S8_2, Cin=>P7C3, Sout=>S8_3, Cout=>P8C3);
338 FA8_4: FullAdder port map(A=>A3B5, B=>S8_3, Cin=>P7C4, Sout=>S8_4, Cout=>P8C4);
339 FA8_5: FullAdder port map(A=>A2B6, B=>S8_4, Cin=>P7C5, Sout=>S8_5, Cout=>P8C5);
340 FA8_6: FullAdder port map(A=>A1B7, B=>S8_5, Cin=>P7C6, Sout=>P(8), Cout=>P8C6);
341
342 -- P(9)
343 FA9_1: FullAdder port map(A=>A7B2, B=>P8C0, Cin=>P8C1, Sout=>S9_0, Cout=>P9C0);
344 FA9_2: FullAdder port map(A=>A6B3, B=>S9_0, Cin=>P8C2, Sout=>S9_1, Cout=>P9C1);
345 FA9_3: FullAdder port map(A=>A5B4, B=>S9_1, Cin=>P8C3, Sout=>S9_2, Cout=>P9C2);
346 FA9_4: FullAdder port map(A=>A4B5, B=>S9_2, Cin=>P8C4, Sout=>S9_3, Cout=>P9C3);
347 FA9_5: FullAdder port map(A=>A3B6, B=>S9_3, Cin=>P8C5, Sout=>S9_4, Cout=>P9C4);
348 FA9_6: FullAdder port map(A=>A2B7, B=>S9_4, Cin=>P8C6, Sout=>P(9), Cout=>P9C5);
349
350 -- P(10)
351 FA10_1: FullAdder port map(A=>A7B3, B=>P9C0, Cin=>P9C1, Sout=>S10_0, Cout=>P10C0);
352 FA10_2: FullAdder port map(A=>A6B4, B=>S10_0, Cin=>P9C2, Sout=>S10_1, Cout=>P10C1);
353 FA10_3: FullAdder port map(A=>A5B5, B=>S10_1, Cin=>P9C3, Sout=>S10_2, Cout=>P10C2);
354 FA10_4: FullAdder port map(A=>A4B6, B=>S10_2, Cin=>P9C4, Sout=>S10_3, Cout=>P10C3);
355 FA10_5: FullAdder port map(A=>A3B7, B=>S10_3, Cin=>P9C5, Sout=>P(10), Cout=>P10C4);
356
357 -- P(11)
358 FA11_1: FullAdder port map(A=>A7B4, B=>P10C0, Cin=>P10C1, Sout=>S11_0, Cout=>P11C0);
359 FA11_2: FullAdder port map(A=>A6B5, B=>S11_0, Cin=>P10C2, Sout=>S11_1, Cout=>P11C1);
360 FA11_3: FullAdder port map(A=>A5B6, B=>S11_1, Cin=>P10C3, Sout=>S11_2, Cout=>P11C2);
361 FA11_4: FullAdder port map(A=>A4B7, B=>S11_2, Cin=>P10C4, Sout=>P(11), Cout=>P11C3);
362
363 -- P(12)
364 FA12_1: FullAdder port map(A=>A7B5, B=>P11C0, Cin=>P11C1, Sout=>S12_0, Cout=>P12C0);
365 FA12_2: FullAdder port map(A=>A6B6, B=>S12_0, Cin=>P11C2, Sout=>S12_1, Cout=>P12C1);
366 FA12_3: FullAdder port map(A=>A5B7, B=>S12_1, Cin=>P11C3, Sout=>P(12), Cout=>P12C2);
367
368 -- P(13)
369 FA13_1: FullAdder port map(A=>A7B6, B=>P12C0, Cin=>P12C1, Sout=>S13_0, Cout=>P13C0);
370 FA13_2: FullAdder port map(A=>A6B7, B=>S13_0, Cin=>P12C2, Sout=>P(13), Cout=>P13C1);
371
372 -- P(14) and P(15)
373 FA14: FullAdder port map(A=>A7B7, B=>P13C0, Cin=>P13C1, Sout=>P(14), Cout=>P(15));
374
375 end Behavioral;

```

Full 8x8 Wallace tree multiplier VHDL Code

```
22 : library IEEE;
23 : use IEEE.STD_LOGIC_1164.ALL;
24 :
25 : -- Uncomment the following library declaration if using
26 : -- arithmetic functions with Signed or Unsigned values
27 : --use IEEE.NUMERIC_STD.ALL;
28 :
29 : -- Uncomment the following library declaration if instantiating
30 : -- any Xilinx leaf cells in this code.
31 : --library UNISIM;
32 : --use UNISIM.VComponents.all;
33 :
34 : entity EightEightWallace is
35 :     Port (
36 :         A : in STD_LOGIC_VECTOR (7 downto 0) ;
37 :         B : in STD_LOGIC_VECTOR (7 downto 0);
38 :         P : out STD_LOGIC_VECTOR (15 downto 0));
39 : end EightEightWallace;
40 :
41 : architecture Behavioral of EightEightWallace is
42 :
43 :     component FullAdder
44 :     port (
45 :         A : in STD_LOGIC;
46 :         B : in STD_LOGIC;
47 :         Cin : in STD_LOGIC;
48 :         Sout : out STD_LOGIC;
49 :         Cout : out STD_LOGIC);
50 :     end component;
51 :
52 :     component HalfAdder
53 :     port (
54 :         A : in STD_LOGIC;
55 :         B : in STD_LOGIC;
56 :         Sout : out STD_LOGIC;
57 :         Cout : out STD_LOGIC);
58 :     end component;
59 :
60 :     signal A0B0,A1B0,A2B0,A3B0,A4B0,A5B0,A6B0,A7B0 : std_logic;
61 :     signal A0B1,A1B1,A2B1,A3B1,A4B1,A5B1,A6B1,A7B1 : std_logic;
62 :     signal A0B2,A1B2,A2B2,A3B2,A4B2,A5B2,A6B2,A7B2 : std_logic;
63 :     signal A0B3,A1B3,A2B3,A3B3,A4B3,A5B3,A6B3,A7B3 : std_logic;
64 :     signal A0B4,A1B4,A2B4,A3B4,A4B4,A5B4,A6B4,A7B4 : std_logic;
65 :     signal A0B5,A1B5,A2B5,A3B5,A4B5,A5B5,A6B5,A7B5 : std_logic;
66 :     signal A0B6,A1B6,A2B6,A3B6,A4B6,A5B6,A6B6,A7B6 : std_logic;
67 :     signal A0B7,A1B7,A2B7,A3B7,A4B7,A5B7,A6B7,A7B7 : std_logic;
```

```

69  --Carry
70  --P1
71  signal P1C0: std_logic;
72
73  --P2
74  signal P2C0: std_logic;
75  signal P2C1: std_logic;
76
77  --P3
78  signal P3C0: std_logic;
79  signal P3C1: std_logic;
80  signal P3C2: std_logic;
81
82  --P4
83  signal P4C0: std_logic;
84  signal P4C1: std_logic;
85  signal P4C2: std_logic;
86  signal P4C3: std_logic;
87  signal P4C4: std_logic;
88
89  --P5
90  signal P5C0: std_logic;
91  signal P5C1: std_logic;
92  signal P5C2: std_logic;
93  signal P5C3: std_logic;
94  signal P5C4: std_logic;
95
96  --P6
97  signal P6C0: std_logic;
98  signal P6C1: std_logic;
99  signal P6C2: std_logic;
100 signal P6C3: std_logic;
101 signal P6C4: std_logic;
102 signal P6C5: std_logic;
103
104  --P7
105 signal P7C0: std_logic;
106 signal P7C1: std_logic;
107 signal P7C2: std_logic;
108 signal P7C3: std_logic;
109 signal P7C4: std_logic;
110 signal P7C5: std_logic;
111
112  --P8
113 signal P8C0: std_logic;
114 signal P8C1: std_logic;
115 signal P8C2: std_logic;
116 signal P8C3: std_logic;
117 signal P8C4: std_logic;
118 signal P8C5: std_logic;

```

```
120  --P9
121  signal P9C0: std_logic;
122  signal P9C1: std_logic;
123  signal P9C2: std_logic;
124  signal P9C3: std_logic;
125  signal P9C4: std_logic;
126
127  --P10
128  signal P10C0: std_logic;
129  signal P10C1: std_logic;
130  signal P10C2: std_logic;
131  signal P10C3: std_logic;
132
133  --P11
134  signal P11C0: std_logic;
135  signal P11C1: std_logic;
136  signal P11C2: std_logic;
137  signal P11C3: std_logic;
138
139  --P12
140  signal P12C0: std_logic;
141  signal P12C1: std_logic;
142  signal P12C2: std_logic;
143
144  --P13
145  signal P13C0: std_logic;
146  signal P13C1: std_logic;
147
148  --P14
149  signal P14C0: std_logic;
150
151  --FINAL PRODUCT CARRIES
152  signal P5FC: std_logic;
153  signal P6FC: std_logic;
154  signal P7FC: std_logic;
155  signal P8FC: std_logic;
156  signal P9FC: std_logic;
157  signal P10FC: std_logic;
158  signal P11FC: std_logic;
159  signal P12FC: std_logic;
160  signal P13FC: std_logic;
161  signal P14FC: std_logic;
162  signal P15FC: std_logic;
```

```
164  --intermediate signals
165  signal S2_0 : std_logic;
166  signal S2_1 : std_logic;
167
168  signal S3_0 : std_logic;
169  signal S3_1 : std_logic;
170  signal S3_2 : std_logic;
171
172  signal S4_0 : std_logic;
173  signal S4_1 : std_logic;
174  signal S4_2 : std_logic;
175  signal S4_3 : std_logic;
176  signal S4_4 : std_logic;
177
178  signal S5_0 : std_logic;
179  signal S5_1 : std_logic;
180  signal S5_2 : std_logic;
181  signal S5_3 : std_logic;
182  signal S5_4 : std_logic;
183
184  signal S6_0 : std_logic;
185  signal S6_1 : std_logic;
186  signal S6_2 : std_logic;
187  signal S6_3 : std_logic;
188  signal S6_4 : std_logic;
189  signal S6_5 : std_logic;
190
191  signal S7_0 : std_logic;
192  signal S7_1 : std_logic;
193  signal S7_2 : std_logic;
194  signal S7_3 : std_logic;
195  signal S7_4 : std_logic;
196  signal S7_5 : std_logic;
197
198  signal S8_0 : std_logic;
199  signal S8_1 : std_logic;
200  signal S8_2 : std_logic;
201  signal S8_3 : std_logic;
202  signal S8_4 : std_logic;
203  signal S8_5 : std_logic;
204
205  signal S9_0 : std_logic;
206  signal S9_1 : std_logic;
207  signal S9_2 : std_logic;
208  signal S9_3 : std_logic;
209  signal S9_4 : std_logic;
210
211  signal S10_0 : std_logic;
212  signal S10_1 : std_logic;
213  signal S10_2 : std_logic;
214  signal S10_3 : std_logic;
```

```

225     signal S13_0 : std_logic;
226     signal S13_1 : std_logic;
227
228     signal S14_0 : std_logic;
229
230     begin
231
232         -- B(0) row
233         A0B0 <= A(0) and B(0);
234         A1B0 <= A(1) and B(0);
235         A2B0 <= A(2) and B(0);
236         A3B0 <= A(3) and B(0);
237         A4B0 <= A(4) and B(0);
238         A5B0 <= A(5) and B(0);
239         A6B0 <= A(6) and B(0);
240         A7B0 <= A(7) and B(0);
241
242         -- B(1) row
243         A0B1 <= A(0) and B(1);
244         A1B1 <= A(1) and B(1);
245         A2B1 <= A(2) and B(1);
246         A3B1 <= A(3) and B(1);
247         A4B1 <= A(4) and B(1);
248         A5B1 <= A(5) and B(1);
249         A6B1 <= A(6) and B(1);
250         A7B1 <= A(7) and B(1);
251
252         -- B(2) row
253         A0B2 <= A(0) and B(2);
254         A1B2 <= A(1) and B(2);
255         A2B2 <= A(2) and B(2);
256         A3B2 <= A(3) and B(2);
257         A4B2 <= A(4) and B(2);
258         A5B2 <= A(5) and B(2);
259         A6B2 <= A(6) and B(2);
260         A7B2 <= A(7) and B(2);
261
262         -- B(3) row
263         A0B3 <= A(0) and B(3);
264         A1B3 <= A(1) and B(3);
265         A2B3 <= A(2) and B(3);
266         A3B3 <= A(3) and B(3);
267         A4B3 <= A(4) and B(3);
268         A5B3 <= A(5) and B(3);
269         A6B3 <= A(6) and B(3);
270         A7B3 <= A(7) and B(3);

```

```
272  -- B(4) row
273  A0B4 <= A(0) and B(4);
274  A1B4 <= A(1) and B(4);
275  A2B4 <= A(2) and B(4);
276  A3B4 <= A(3) and B(4);
277  A4B4 <= A(4) and B(4);
278  A5B4 <= A(5) and B(4);
279  A6B4 <= A(6) and B(4);
280  A7B4 <= A(7) and B(4);
281
282  -- B(5) row
283  A0B5 <= A(0) and B(5);
284  A1B5 <= A(1) and B(5);
285  A2B5 <= A(2) and B(5);
286  A3B5 <= A(3) and B(5);
287  A4B5 <= A(4) and B(5);
288  A5B5 <= A(5) and B(5);
289  A6B5 <= A(6) and B(5);
290  A7B5 <= A(7) and B(5);
291
292  -- B(6) row
293  A0B6 <= A(0) and B(6);
294  A1B6 <= A(1) and B(6);
295  A2B6 <= A(2) and B(6);
296  A3B6 <= A(3) and B(6);
297  A4B6 <= A(4) and B(6);
298  A5B6 <= A(5) and B(6);
299  A6B6 <= A(6) and B(6);
300  A7B6 <= A(7) and B(6);
301
302  -- B(7) row
303  A0B7 <= A(0) and B(7);
304  A1B7 <= A(1) and B(7);
305  A2B7 <= A(2) and B(7);
306  A3B7 <= A(3) and B(7);
307  A4B7 <= A(4) and B(7);
308  A5B7 <= A(5) and B(7);
309  A6B7 <= A(6) and B(7);
310  A7B7 <= A(7) and B(7);
```

```

312 -- LEVEL 1
313 -- P(0)
314 P(0) <= A0B0;
315
316 --1st row level 1
317 HA1_0 : HalfAdder port map(A=>A1B0,B=>A0B1,Sout=>P(1),Cout=>P1C0);
318 FA1_0 : FullAdder port map(A=>A2B0,B=>A1B1,Cin=>A0B2,Sout=>S2_0,Cout=>P2C0);
319 FA1_1 : FullAdder port map(A=>A3B0,B=>A2B1,Cin=>A1B2,Sout=>S3_0,Cout=>P3C0);
320 FA1_2 : FullAdder port map(A=>A4B0,B=>A3B1,Cin=>A2B2,Sout=>S4_0,Cout=>P4C0);
321 FA1_3 : FullAdder port map(A=>A5B0,B=>A4B1,Cin=>A3B2,Sout=>S5_0,Cout=>P5C0);
322 FA1_4 : FullAdder port map(A=>A6B0,B=>A5B1,Cin=>A4B2,Sout=>S6_0,Cout=>P6C0);
323 FA1_5 : FullAdder port map(A=>A7B0,B=>A6B1,Cin=>A5B2,Sout=>S7_0,Cout=>P7C0);
324 HA1_1 : HalfAdder port map(A=>A7B1,B=>A6B2,Sout=>S8_0,Cout=>P8C0);
325
326 --2nd row level 1
327 HA1_2 : HalfAdder port map(A=>A1B3,B=>A0B4,Sout=>S4_1,Cout=>P4C1);
328 FA1_6 : FullAdder port map(A=>A2B3,B=>A1B4,Cin=>A0B5,Sout=>S5_1,Cout=>P5C1);
329 FA1_7 : FullAdder port map(A=>A3B3,B=>A2B4,Cin=>A1B5,Sout=>S6_1,Cout=>P6C1);
330 FA1_8 : FullAdder port map(A=>A4B3,B=>A3B4,Cin=>A2B5,Sout=>S7_1,Cout=>P7C1);
331 FA1_9 : FullAdder port map(A=>A5B3,B=>A4B4,Cin=>A3B5,Sout=>S8_1,Cout=>P8C1);
332 FA1_10 : FullAdder port map(A=>A6B3,B=>A5B4,Cin=>A4B5,Sout=>S9_0,Cout=>P9C0);
333 FA1_11 : FullAdder port map(A=>A7B3,B=>A6B4,Cin=>A5B5,Sout=>S10_0,Cout=>P10C0);
334 HA1_3 : HalfAdder port map(A=>A7B4,B=>A6B5,Sout=>S11_0,Cout=>P11C0);
335
336 --LEVEL 2
337
338 --1st row level 2
339 HA2_0 : HalfAdder port map(A=>S2_0,B=>P1C0,Sout=>P(2),Cout=>P2C1);
340 FA2_0 : FullAdder port map(A=>S3_0,B=>P2C0,Cin=>A0B3,Sout=>S3_1,Cout=>P3C1);
341 FA2_1 : FullAdder port map(A=>S4_0,B=>S4_1,Cin=>P3C0,Sout=>S4_2,Cout=>P4C2);
342 FA2_2 : FullAdder port map(A=>S5_0,B=>S5_1,Cin=>P4C0,Sout=>S5_2,Cout=>P5C2);
343 FA2_3 : FullAdder port map(A=>S6_0,B=>S6_1,Cin=>P5C0,Sout=>S6_2,Cout=>P6C2);
344 FA2_4 : FullAdder port map(A=>S7_0,B=>S7_1,Cin=>P6C0,Sout=>S7_2,Cout=>P7C2);
345 FA2_5 : FullAdder port map(A=>S8_0,B=>S8_1,Cin=>P7C0,Sout=>S8_2,Cout=>P8C2);
346 FA2_6 : FullAdder port map(A=>A7B2,B=>S9_0,Cin=>P8C0,Sout=>S9_1,Cout=>P9C1);
347
348 --2nd row level 2
349 HA2_1 : HalfAdder port map(A=>P5C1,B=>A0B6,Sout=>S6_3,Cout=>P6C3);
350 FA2_7 : FullAdder port map(A=>P6C1,B=>A1B6,Cin=>A0B7,Sout=>S7_3,Cout=>P7C3);
351 FA2_8 : FullAdder port map(A=>P7C1,B=>A2B6,Cin=>A1B7,Sout=>S8_3,Cout=>P8C3);
352 FA2_9 : FullAdder port map(A=>P8C1,B=>A3B6,Cin=>A2B7,Sout=>S9_2,Cout=>P9C2);
353 FA2_10 : FullAdder port map(A=>P9C0,B=>A4B6,Cin=>A3B7,Sout=>S10_1,Cout=>P10C1);
354 FA2_11 : FullAdder port map(A=>P10C0,B=>A5B6,Cin=>A4B7,Sout=>S11_1,Cout=>P11C1);
355 FA2_12 : FullAdder port map(A=>A7B5,B=>A6B6,Cin=>A5B7,Sout=>S12_0,Cout=>P12C0);
356 HA2_2 : HalfAdder port map(A=>A7B6,B=>A6B7,Sout=>S13_0,Cout=>P13C0);

```

```

358  --LEVEL 3
359  HA3_0 : HalfAdder port map (A=>S3_1,B=>P2C1,Sout=>P(3),Cout=>P3C2);
360  HA3_1 : HalfAdder port map (A=>S4_2,B=>P3C1,Sout=>S4_3,Cout=>P4C3);
361  FA3_0 : FullAdder port map (A=>S5_2,B=>P4C2,Cin=>P4C1,Sout=>S5_3,Cout=>P5C3);
362  FA3_1 : FullAdder port map (A=>S6_2,B=>S6_3,Cin=>P5C2,Sout=>S6_4,Cout=>P6C4);
363  FA3_2 : FullAdder port map (A=>S7_2,B=>S7_3,Cin=>P6C2,Sout=>S7_4,Cout=>P7C4);
364  FA3_3 : FullAdder port map (A=>S8_2,B=>S8_3,Cin=>P7C2,Sout=>S8_4,Cout=>P8C4);
365  FA3_4 : FullAdder port map (A=>S9_1,B=>S9_2,Cin=>P8C2,Sout=>S9_3,Cout=>P9C3);
366  FA3_5 : FullAdder port map (A=>S10_0,B=>S10_1,Cin=>P9C1,Sout=>S10_2,Cout=>P10C2);
367  HA3_2 : HalfAdder port map (A=>S11_0,B=>S11_1,Sout=>S11_2,Cout=>P11C2);
368  HA3_3 : HalfAdder port map (A=>P11C0,B=>S12_0,Sout=>S12_1,Cout=>P12C1);
369
370  --LEVEL 4
371  HA4_0 : HalfAdder port map (A=>S4_3,B=>P3C2,Sout=>P(4),Cout=>P4C4);
372  HA4_1 : HalfAdder port map (A=>S5_3,B=>P4C3,Sout=>S5_4,Cout=>P5C4);
373  HA4_2 : HalfAdder port map (A=>S6_4,B=>P5C3,Sout=>S6_5,Cout=>P6C5);
374  FA4_0 : FullAdder port map (A=>S7_4,B=>P6C4,Cin=>P6C3,Sout=>S7_5,Cout=>P7C5);
375  FA4_1 : FullAdder port map (A=>S8_4,B=>P7C4,Cin=>P7C3,Sout=>S8_5,Cout=>P8C5);
376  FA4_2 : FullAdder port map (A=>S9_3,B=>P8C4,Cin=>P8C3,Sout=>S9_4,Cout=>P9C4);
377  FA4_3 : FullAdder port map (A=>S10_2,B=>P9C3,Cin=>P9C2,Sout=>S10_3,Cout=>P10C3);
378  FA4_4 : FullAdder port map (A=>S11_2,B=>P10C2,Cin=>P10C1,Sout=>S11_3,Cout=>P11C3);
379  FA4_5 : FullAdder port map (A=>S12_1,B=>P11C2,Cin=>P11C1,Sout=>S12_2,Cout=>P12C2);
380  FA4_6 : FullAdder port map (A=>P12C1,B=>S13_0,Cin=>P12C0,Sout=>S13_1,Cout=>P13C1);
381  HA4_3 : HalfAdder port map (A=>P13C0,B=>A7B7,Sout=>S14_0,Cout=>P14C0);
382
383  --LEVEL 5
384  HA5_0 : HalfAdder port map (A=>S5_4,B=>P4C4,Sout=>P(5),Cout=>P5FC);
385  FA5_0 : FullAdder port map (A=>S6_5,B=>P5C4,Cin=>P5FC,Sout=>P(6),Cout=>P6FC);
386  FA5_1 : FullAdder port map (A=>S7_5,B=>P6C5,Cin=>P6FC,Sout=>P(7),Cout=>P7FC);
387  FA5_2 : FullAdder port map (A=>S8_5,B=>P7C5,Cin=>P7FC,Sout=>P(8),Cout=>P8FC);
388  FA5_3 : FullAdder port map (A=>S9_4,B=>P8C5,Cin=>P8FC,Sout=>P(9),Cout=>P9FC);
389  FA5_4 : FullAdder port map (A=>S10_3,B=>P9C4,Cin=>P9FC,Sout=>P(10),Cout=>P10FC);
390  FA5_5 : FullAdder port map (A=>S11_3,B=>P10C3,Cin=>P10FC,Sout=>P(11),Cout=>P11FC);
391  FA5_6 : FullAdder port map (A=>S12_2,B=>P11C3,Cin=>P11FC,Sout=>P(12),Cout=>P12FC);
392  FA5_7 : FullAdder port map (A=>S13_1,B=>P12C2,Cin=>P12FC,Sout=>P(13),Cout=>P13FC);
393  FA5_8 : FullAdder port map (A=>S14_0,B=>P13C1,Cin=>P13FC,Sout=>P(14),Cout=>P14FC);
394  HA5_1 : HalfAdder port map (A=>P14C0,B=>P14FC,Sout=>P(15),Cout=>P15FC);
395
396
397  end Behavioral;

```

Appendix B. FPGA Simulation Results

15 x 10 Braun

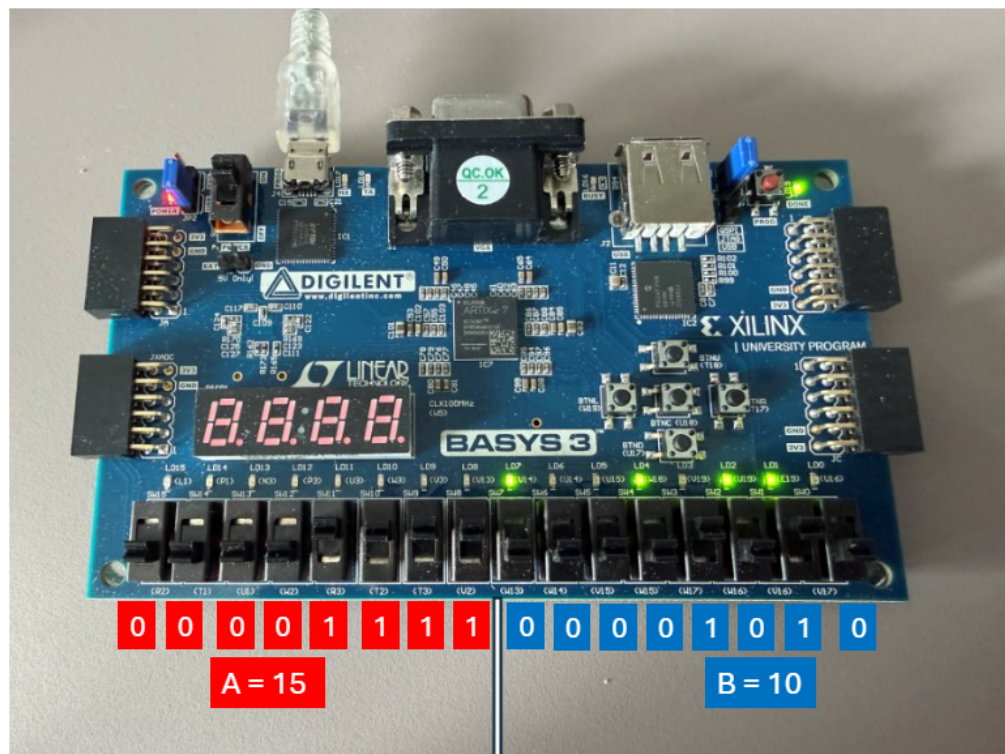


Figure 42 – 15x10 result from Braun’s multiplier on FPGA board.

15 x 10 Wallace

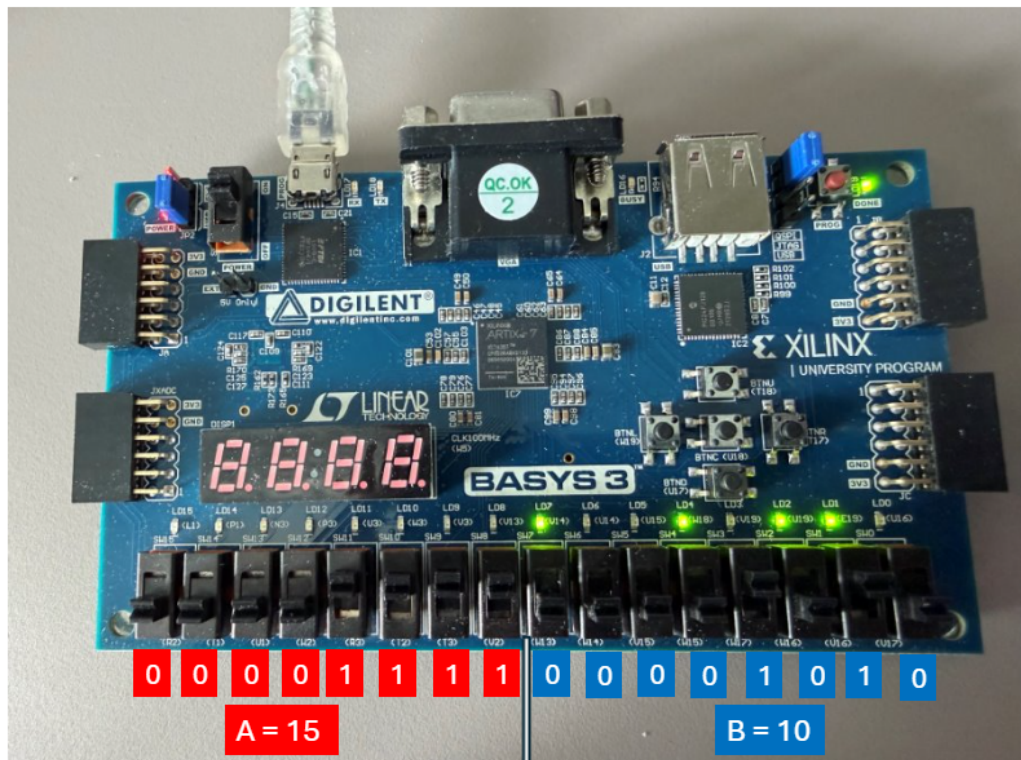


Figure 43 - 15x10 result from Wallace tree multiplier on FPGA board.

9F x 7E Braun

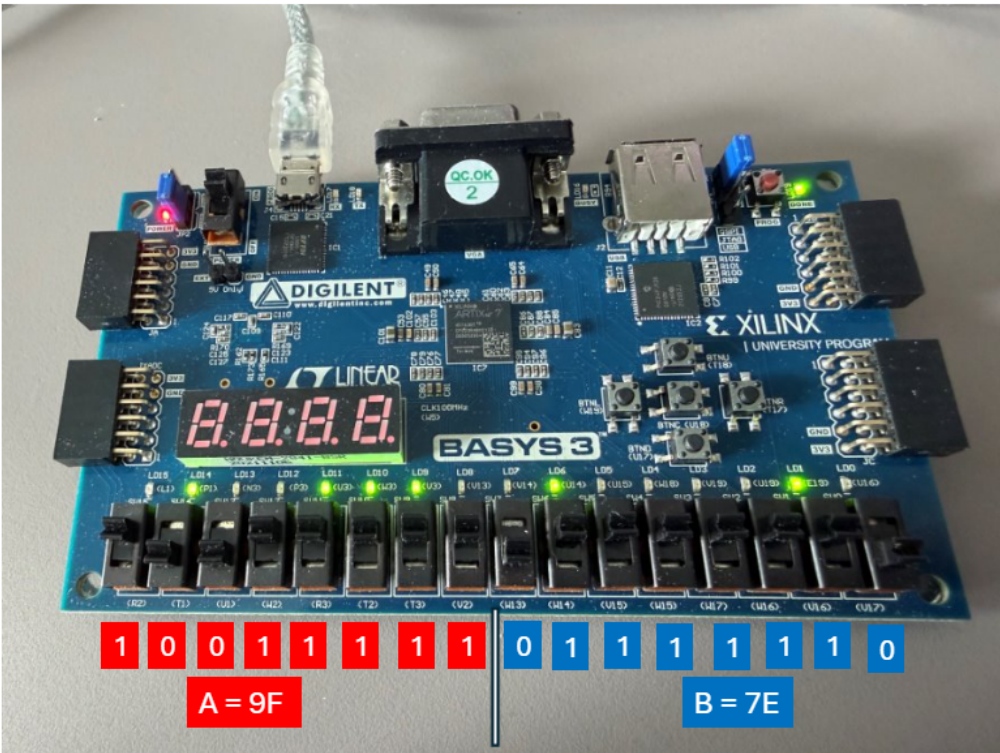


Figure 44 – 9F x 7E result from Braun’s multiplier on FPGA board.

9F x 7E Wallace

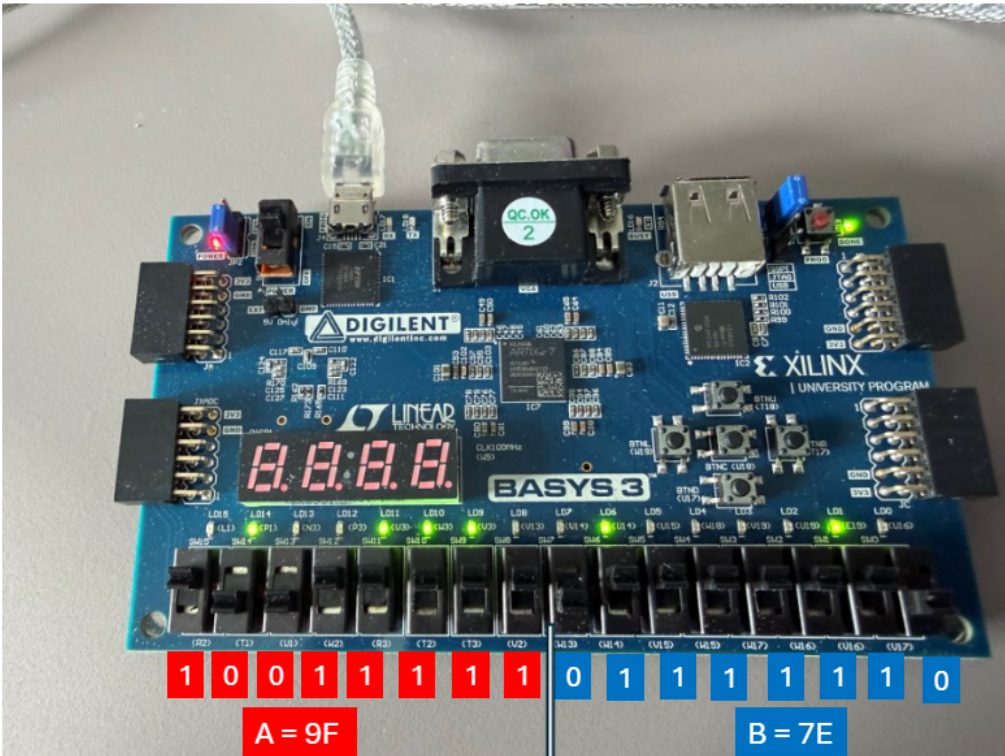


Figure 45 - 9F x 7E result from Wallace tree multiplier on FPGA board.

DE x EF Braun

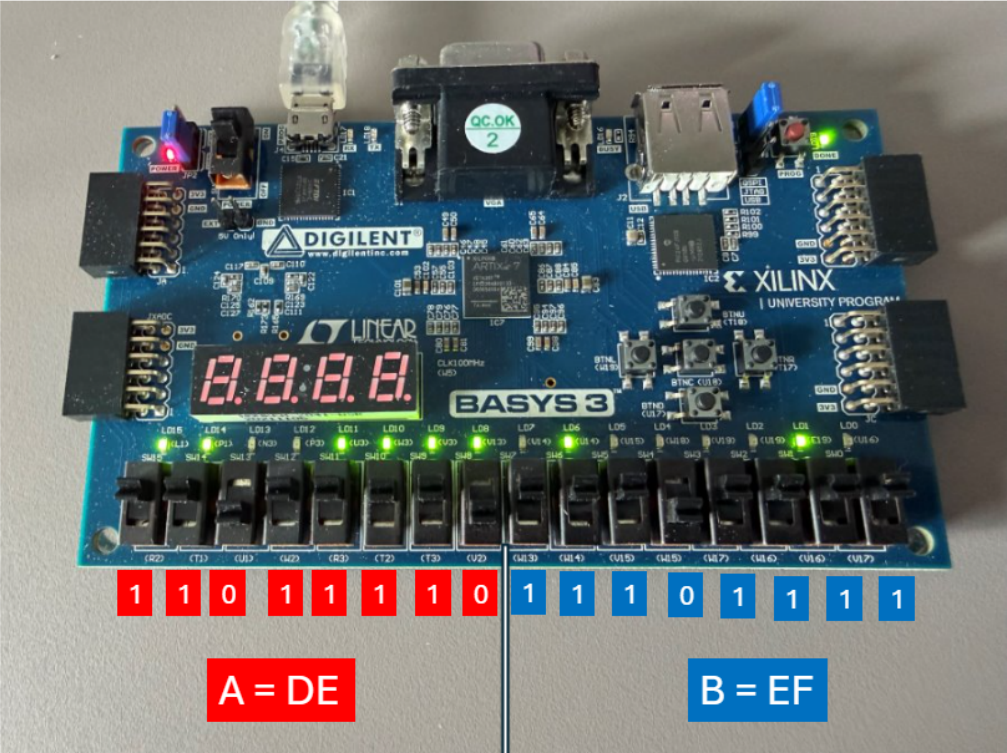


Figure 46 - DExEF result from Braun’s multiplier on FPGA board.

DE x EF Wallace

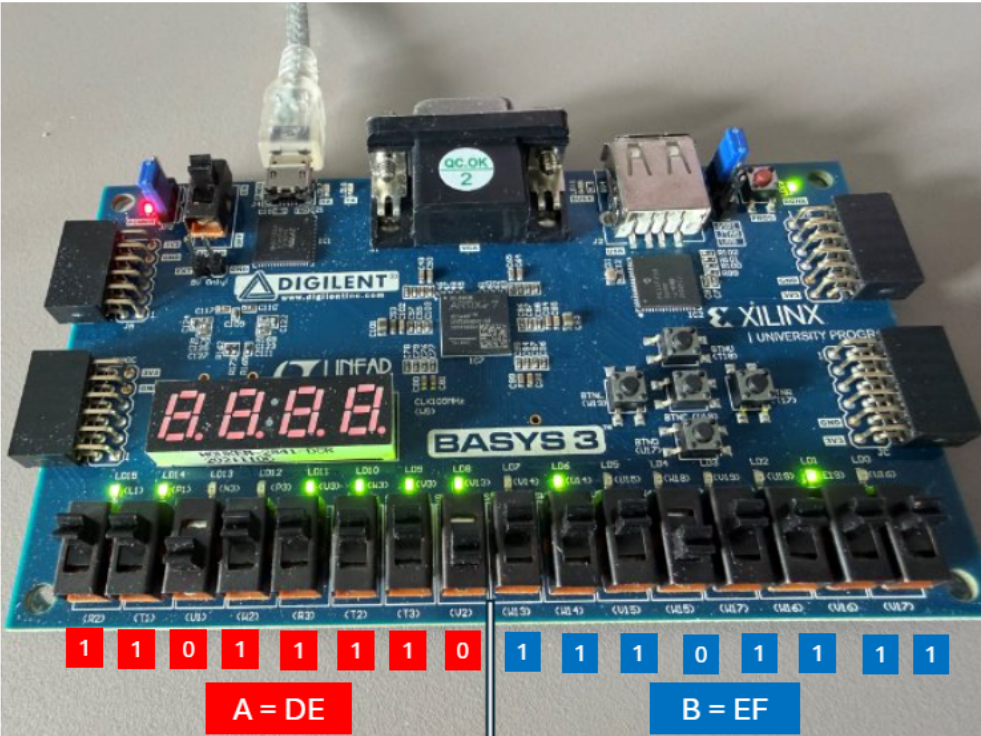


Figure 47 - DExEF result from Wallace tree multiplier on FPGA board.

FF x FF Braun

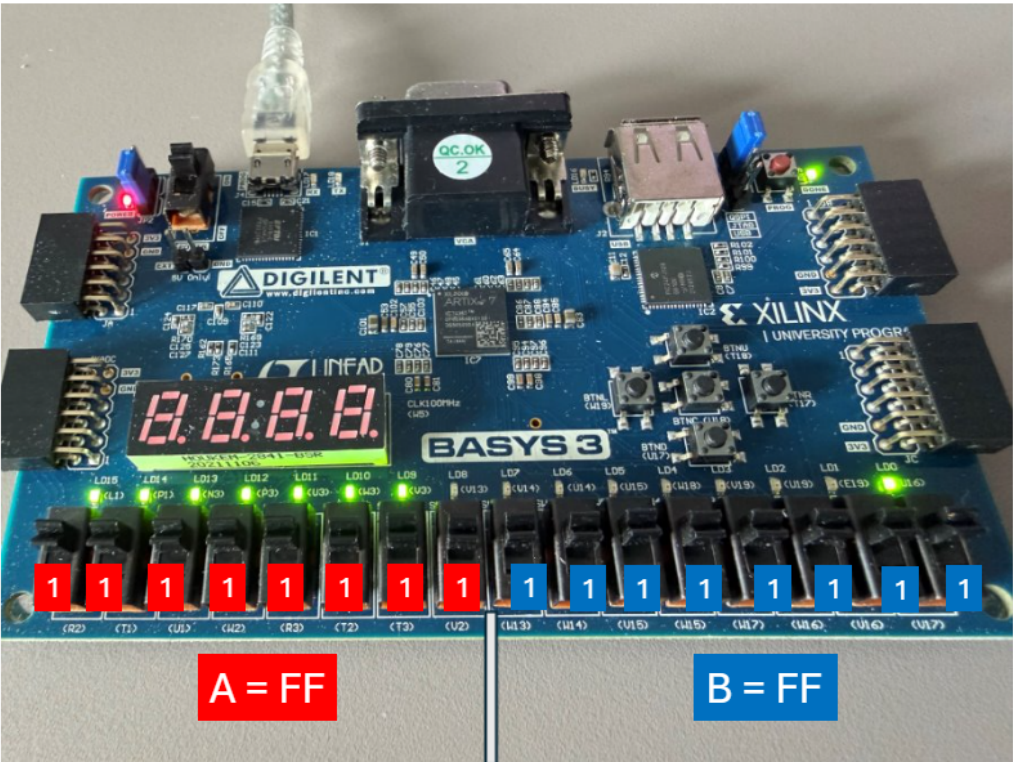


Figure 48 - FFxFF result from Braun’s multiplier on FPGA board.

FF x FF Wallace

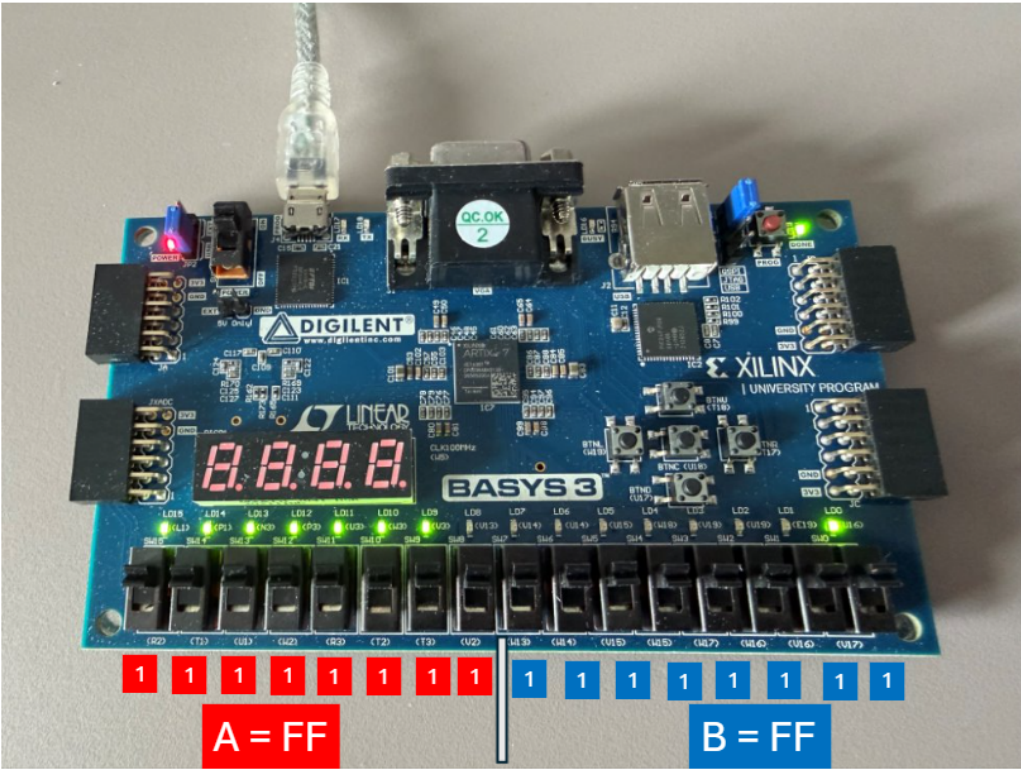


Figure 49 - FFxFF result from Wallace tree multiplier on FPGA board.

Appendix C. Handwritten labels on 8x8 Braun's multiplier block diagram.

