

Multiplier Assignment

by Niall Rees

Submission date: 14-May-2023 09:40PM (UTC+0100)

Submission ID: 206146975

File name: 42910_Niall_Rees_Multiplier_Assignment_497727_445532353.docx (10.28M)

Word count: 5967

Character count: 31818

ACDA6001: Critical Evaluation of Braun's Multiplier and Wallace Tree Multiplier Using VHDL.

1.0 Introduction

Multipliers are a critical element in many applications in the area of computing. They are often dedicated units within hardware elements and perform demanding processes such as digital signal processing, fast Fourier transforms, image, and video processing. The demands placed on multipliers have led to extensive research into improving the performance with regards to speed and power consumption [1].

Multiplier circuits can be implemented in a variety of ways, this report focusses on the dominant methods of Braun and Wallace tree type multipliers. Utilising XILINX Vivado software, VHDL code can be developed for 8 x 8-bit unsigned binary multipliers using both design methods. From the VHDL code, the designs are synthesised and implemented onto FPGA where their timing, simulated behaviour and power requirements can be analysed.

Multiplier Design Research

Extensive research has been conducted to reduce the size and delay of binary multipliers implemented on to FPGA. With regards to the Braun multiplier, the total system delay is related to the number of full adders used and the size of the final adder blocks creating the product term. Suggestions for using row and column bypassing techniques are suggested in [2] along with fast adders. The results demonstrate how the changes effect FPGA utilisation when implemented on varying devices.

Implementation of Braun multipliers of varying $M \times N$ multiplier and Multiplicand sizes are implemented using VHDL in [3] where it is shown that the LUT utilisation remains the same across different hardware belonging to the Virtex FPGA devices assessed when the size of the multiplier increases.

Wallace suggested a fast method for binary multiplication with a tree type design [4], this is an efficient method to increase the speed of multiplication over a standard array type multiplier. Partial products are grouped into rows of three bits and fed into half and full adders in reduction stages. This methodology is discussed further in the report. Due to the higher speeds, research has been conducted to optimise tree type multipliers. A suggestion to use 4:2 compressors to group partial products is presented in [5] rather than 3:2 compressor full adders. This leads to less row reduction stages leading to less delay and number of interconnections.

In a greater search to reduce the size and efficiency of the Wallace Tree multiplier, a method for using reversible logic gates is suggested in [6].



The effect of using a reduced complexity Wallace Tree Multiplier is shown in [7]. The partial products are displayed in an inverted pyramid scheme before grouping and reduction. This results in fewer adders realised in the design, reducing delay and circuit complexity. It is shown that using Booth recoding and higher order compressors improves the speed of the multiplier.

Array type and Wallace tree multipliers are compared in [8] where it is stated that array type multipliers are slower than the Wallace tree and consume more power, however, they consume less area and have the simplest circuitry. The suggestion of using higher order compressors is given to improve delay in tree type multipliers.

2.0 Braun Multiplier

The Braun multiplier is an array type multiplier suitable for unsigned binary numbers. It consists of AND gates, half, and full adders. For an $N \times N$ bit input, a generated output of N^2 bits will be produced. Partial products are first generated using AND gates. These are then fed into an array of full and half adders. The sum can be taken directly as an element of the final product or fed into the next row of adders, with the carry out from one adder being fed into the subsequent adder in that row. This pattern is iterated until the final stage of full adders where the carry out bit is rippled through the final row producing the final product sum. As the number of bits increases, the number of AND gates and adders increase rapidly. This leads to more power consumption and area on an FPGA. This typically leads to Braun multipliers to be used in systems less than 16 bits [9]. The final ripple carry adder of a Braun multiplier is critical as it determines the overall delay of the system, this has led to methods of modifying the final stage to reduce delay [10]. A Braun array multiplier consists of $n(n-1)$ adders and n^2 and gates [11]. Therefore, to implement an 8bit x 8bit standard Braun multiplier, the requirements are as follows:

$$AND \text{ gates} = n^2$$

$$AND \text{ gates} = 8^2$$

$$AND \text{ gates} = 64$$

$$Adders = n(n-1)$$

$$Adders = 8(8-1)$$

$$Adders = 56$$

The generated partial products are created from the product of each bit of the multiplier by each bit of the multiplicand. These can be shown in table 1:

								A7	A6	A5	A4	A3	A2	A1	A0
							×	B7	B6	B5	B4	B3	B2	B1	B0
								A7B0	A6B0	A5B0	A4B0	A3B0	A2B0	A1B0	A0B0
							A7B0	A6B1	A5B1	A4B1	A3B1	A2B1	A1B1	A0B1	
							A7B1	A6B2	A5B2	A4B2	A3B2	A2B2	A1B2	A0B2	
							A7B2	A6B3	A5B3	A4B3	A3B3	A2B3	A1B3	A0B3	
							A7B3	A6B4	A5B4	A4B4	A3B4	A2B4	A1B4	A0B4	
							A7B4	A6B5	A5B5	A4B5	A3B5	A2B5	A1B5	A0B5	
							A7B5	A6B6	A5B6	A4B6	A3B6	A2B6	A1B6	A0B6	
							A7B6	A6B7	A5B7	A4B7	A3B7	A2B7	A1B7	A0B7	
P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Table 1 - Partial Products $N \times N$

2.1 Braun Multiplier Logic Diagram

To create the logic diagram for a Braun multiplier, half and full adder components need to be defined.

2.1.2 Half Adder

A half adder will take two inputs A and B. The outputs produced are a SUM bit and a Carry bit. Figure 1 displays the circuit diagram for a half adder.

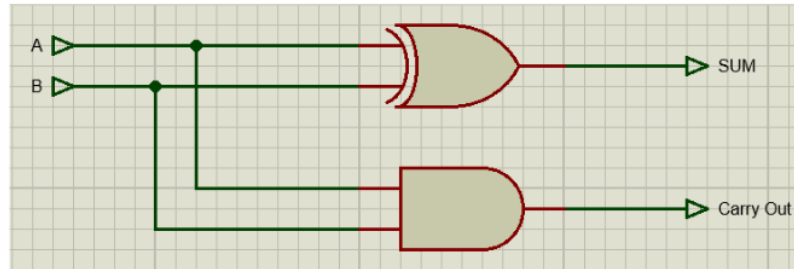


Figure 1 - Half Adder Circuit Diagram

$$SUM = A \text{ XOR } B$$

$$Carry = A \text{ AND } B$$

The output SUM and Carry bits are utilised in the subsequent adders in the multiplier circuit.

2.1.3 Full Adder

A full adder will accept 3 input bits, A, B and Carry in, and generate two output bits, SUM and Carry Out. Figure 2 displays the circuit diagram of a full adder circuit.

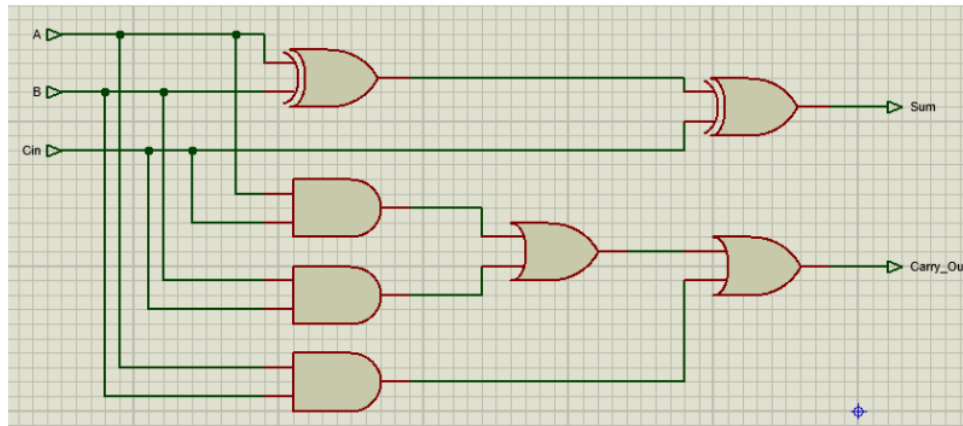


Figure 2 - Full Adder Circuit Diagram

$$SUM = A XOR B XOR C_{in}$$

$$Carry = (A AND B) OR (A AND C_{in}) OR (B AND C_{in})$$

Figure 3 displays the full logic diagram for an 8 x 8 Braun multiplier.

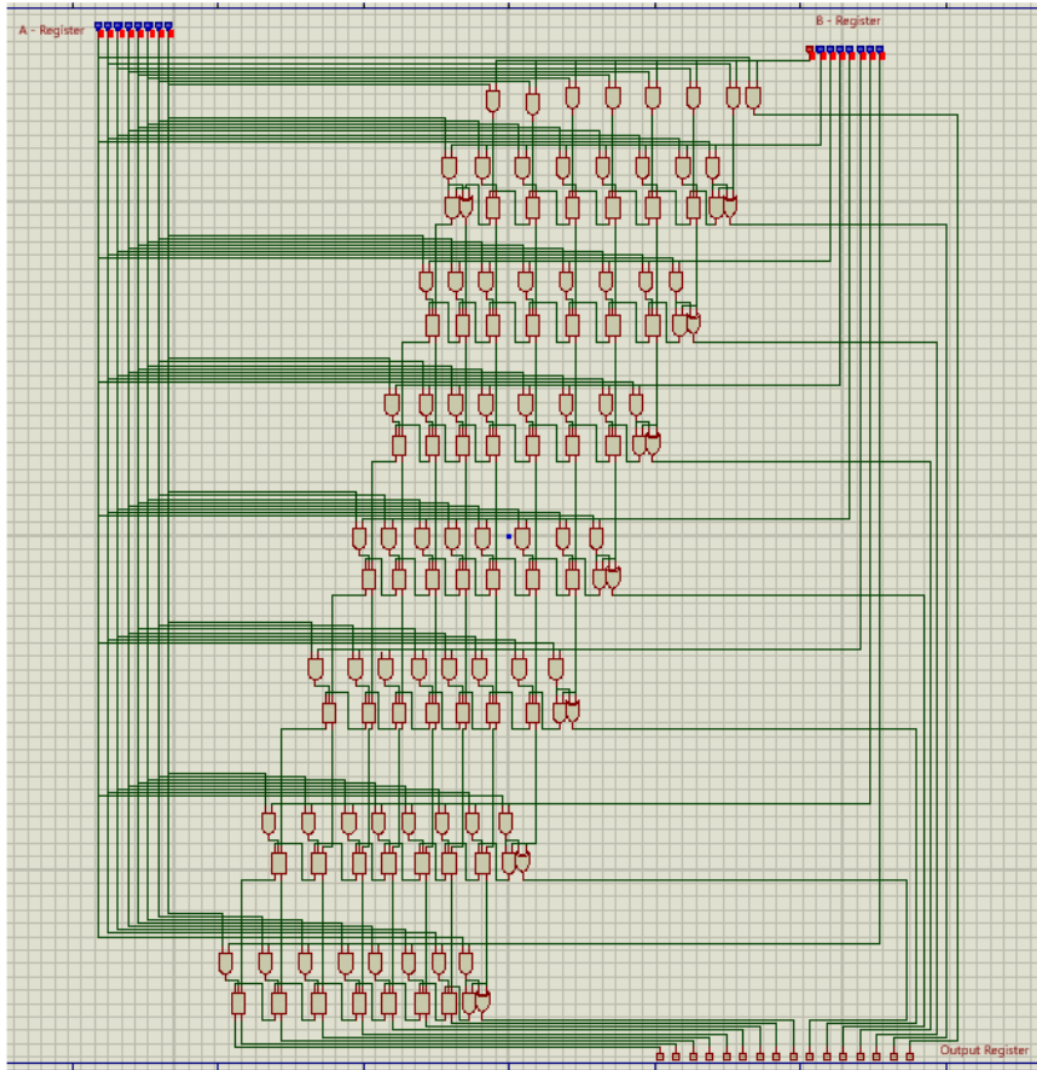
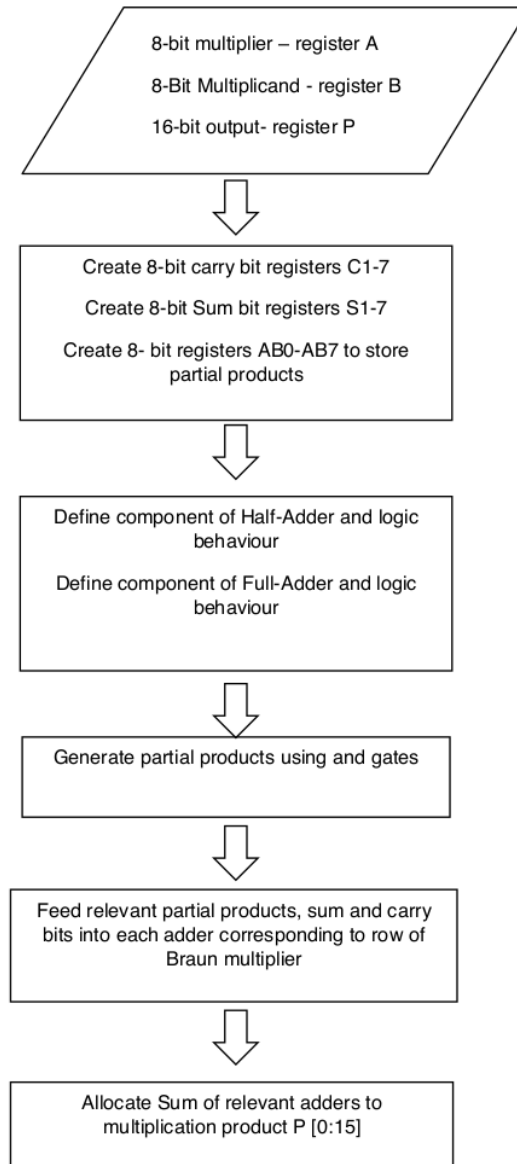


Figure 3 - 8x8 Braun Multiplier

2.2 VHDL Code

In order to implement the Braun multiplier in VHDL, the following program methodology has been followed.



As the Braun multiplier is a combination of partial products generated using AND gates, half and full adders, these entities have been defined and used through the program to avoid creating confusing equations for every output product. The partial products of $A_0 \rightarrow A_7$ and $B_0 \rightarrow B_7$ have been calculated and stored in arrays AB0 to AB7. Each adder equation can then

reference this lookup table, store its sum, and carry-out bits in the S1-7 and C1-7 registers defined.

2.2.1 VHDL Implementation

To simulate the behaviour of the Braun multiplier, VHDL code has been created using XILINX Vivado software. Using Vivado, timing diagrams can be extracted, and performance reports can be generated after synthesis. Appendix A contains the VHDL source code for a Braun multiplier.

To test the function of the multiplier, a test bench program is created to store multiplier and multiplicand data in inputs A and B. Multiplier test bench VHDL code can be seen in Appendix C. From the timing diagram the arithmetic is checked. Figures 4, 5 and 6 display the output timing diagrams.

```

○ A <= "00001010"; -- 10
○ B <= "00000110"; --6
--Expected Output: 60, 0000000000111100
○ wait for period;
○ A <= "00001010"; --10
○ B <= "00000010"; --4
--Expected Output: 40, 0000000000101000
○ wait for period;
○ A <= "00001010"; --10
○ B <= "00001110"; --14
--Expected Output: 140, 0000000010001100
○ wait for period;
○ A <= "00001010"; -- 10
○ B <= "00010010"; --18
--Expected Output: 180, 0000000010110100

```

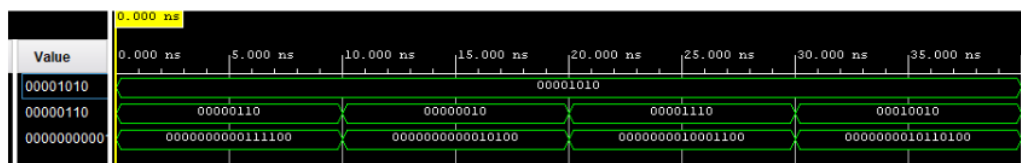
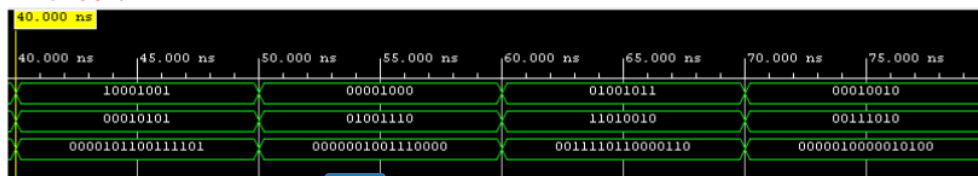


Figure 4 - Sample 1: Multiplier Output

```

○ wait for period;
○ A <= "10001001"; --137
○ B <= "00010101"; --21
--Expected Output: 2877, 0000101100111101
○ wait for period;
○ A <= "00001000"; --8
○ B <= "01001110"; --78
--Expected Output: 624, 0000001001110000
○ wait for period;
○ A <= "01001011"; -- 75
○ B <= "11010010"; --210
--Expected Output: 15750, 0011110110000110
○ wait for period;
○ A <= "00010010"; --18
○ B <= "00111010"; --58
--Expected Output: 1044, 0000010000010100

```



5 - Sample 2: Multiplier Output

```

○ wait for period;
○ A <= "00000111"; --7
○ B <= "01001110"; --78
--Expected Output: 546, 0000001000100010
○ wait for period;
○ A <= "10000111"; --135
○ B <= "01001010"; --74
--Expected Output: 9990, 0010011100000110
○ wait for period;
○ A <= "11100101"; --229
○ B <= "01111010"; --122
--Expected Output: 27938, 0110110100100010
○ wait for period;

```

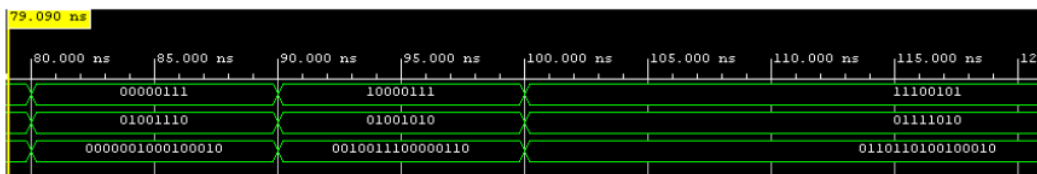


Figure 6 - Sample 3: Multiplier Output

3.0 Synthesised/Implemented Design Results

Using Vivado report generating tools, schematics, timing reports and power consumption data can be analysed from the implemented design. This synthesised design can then be implemented on to an FPGA device. For the Braun multiplier, the synthesised design is implemented using an Artix-7 xc7a50tcsg324-2L. This section displays the synthesised results for the Braun multiplier.

3.1 Braun Multiplier Schematic

Using Vivado RTL elaborated design, a schematic diagram of the circuit can be seen in figure 7. This schematic shows each and gate for the partial products feeding into the array of half and full adders.

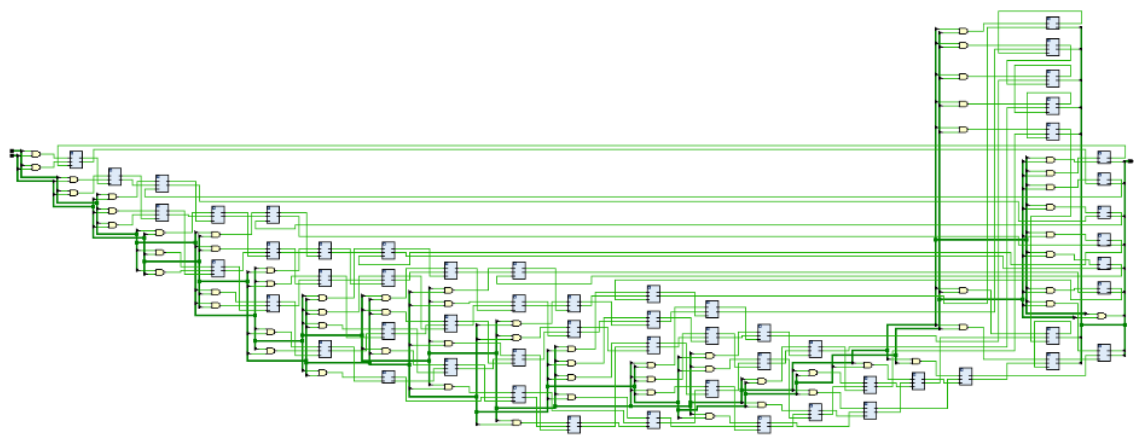


Figure 7 – Elaborated Design Schematic of Braun Multiplier

3.2 FPGA Utilisation Report

Figure 8 shows the utilisation report. This shows the number of LUT's, and IO needed to implement an 8x8 multiplier on to the FPGA.

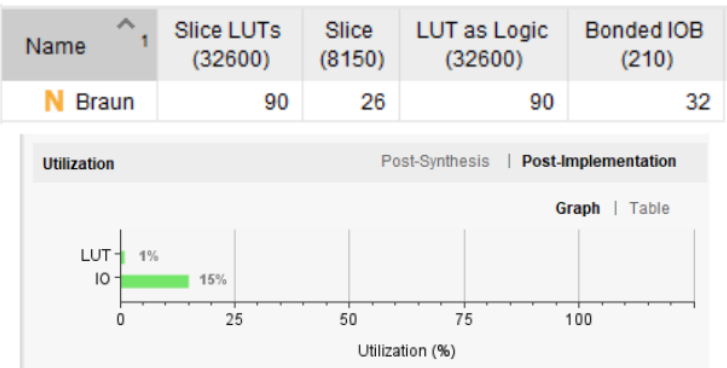


Figure 8 - Utilisation report for 8x8 Braun Multiplier

3.3 Multiplier Critical Delay Path

The critical delay of the multiplier is the longest delay between input and output through the combinational logic. Assuming approximate delay values for adders of $t_{adder} = 2ns$ and gates as $t_{gate} = 1ns$ the critical delay is $t_{delay} = t_{adder} + t_{gate}$. From the logic schematic, the longest path through the multiplier consists of one AND gate and 14 adders. The delay path can then be calculated as follows:

$$t_{delay} = t_{adder} + t_{gate}$$

$$t_{adder} = 14 \times 2$$

$$t_{adder} = 28ns$$

$$t_{gate} = 1ns$$

Therefore:

$$t_{delay} = 28 + 1$$

$$t_{delay} = 29ns$$

This figure is an approximation on assumed values. Xilinx Vivado provides a critical delay path timing report. To gain the combinational delay value from the software, a maximum delay between input and output has been set to 30ns utilising a 100ns virtual clock. The clock is required for timing analysis and not for circuit functionality due to it being purely combinational. From the implemented design the longest delay path is shown in figure 9.

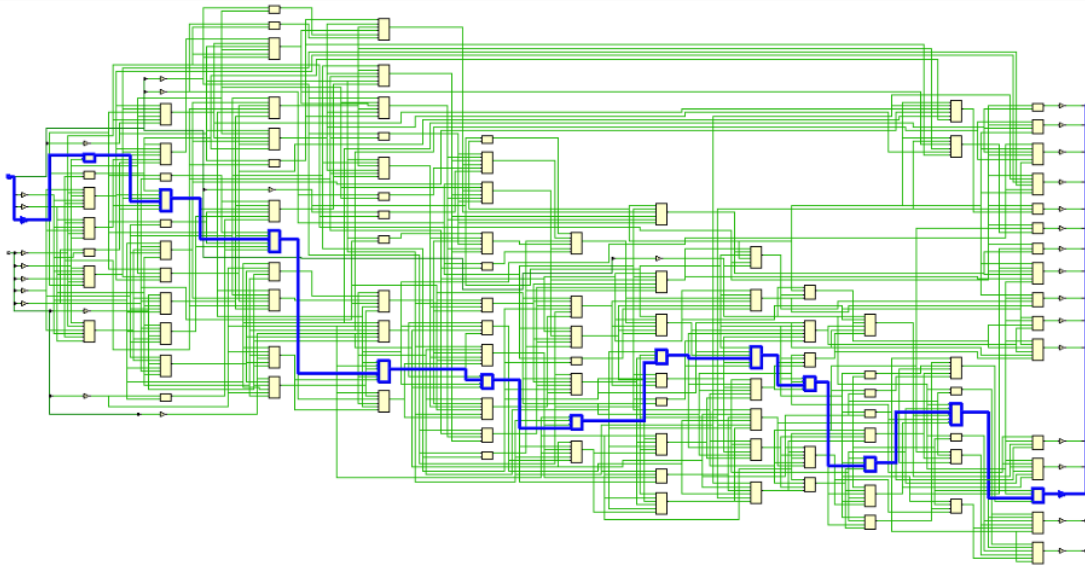


Figure 9 - Longest Delay Path of Braun Multiplier

Figure 10 displays the report for the longest combinational delay paths.

Name	Slack ^{^1}	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
↳ Path 1	23.250	14	13	16	B[2]	P[13]	16.725	5.446	11.280
↳ Path 2	23.411	14	13	16	B[2]	P[14]	16.564	5.418	11.146
↳ Path 3	23.447	14	13	16	B[2]	P[15]	16.528	5.417	11.111
↳ Path 4	23.988	13	12	16	B[2]	P[12]	15.987	5.338	10.649

Figure 10 - Combinational delay Path

From figure 10, the longest combinational delay is between B2 and P13 at 16.725ns.

3.4 Power Utilisation of FPGA

Using Vivado power utilisation reporting, the estimated power of the device can be seen for the Braun multiplier. Figure 11 shows the settings for the synthesised circuit tests and figure 12 shows the report results.

The screenshot shows the 'Report Power' dialog box in Vivado. The title bar says 'Report Power'. Below the title bar, it says 'Analyze power consumption based on the implemented design and part xc7a50tcs9324-2L'. The 'Results name' field is set to 'power_1'. There are four tabs: 'Environment', 'Power Supply', 'Switching', and 'Output'. The 'Environment' tab is selected. Under 'Device Settings', 'Temp grade' is set to 'extended' and 'Process' is set to 'maximum'. Under 'Environment Settings', 'Output Load' is 0 pF, 'Junction temperature' is 65.486 °C, 'Ambient temperature' is 65 °C, 'Effective RJA' is 4.335 °C/W, 'Airflow' is 500 LFM, 'Heat sink' is 'medium (Medium Profile)', 'θSA' is 4.6 °C/W, 'Board selection' is 'medium (10"x10")', 'Number of board layers' is '12to15 (12 to 15 Layers)', and 'θJB' is 6.8 °C/W. At the bottom, there is a 'Legend' section with three options: 'User Defined' (blue square), 'Calculated' (white square), and 'Default' (yellow square). There are also 'OK' and 'Cancel' buttons.

Figure 11 - Power simulation settings

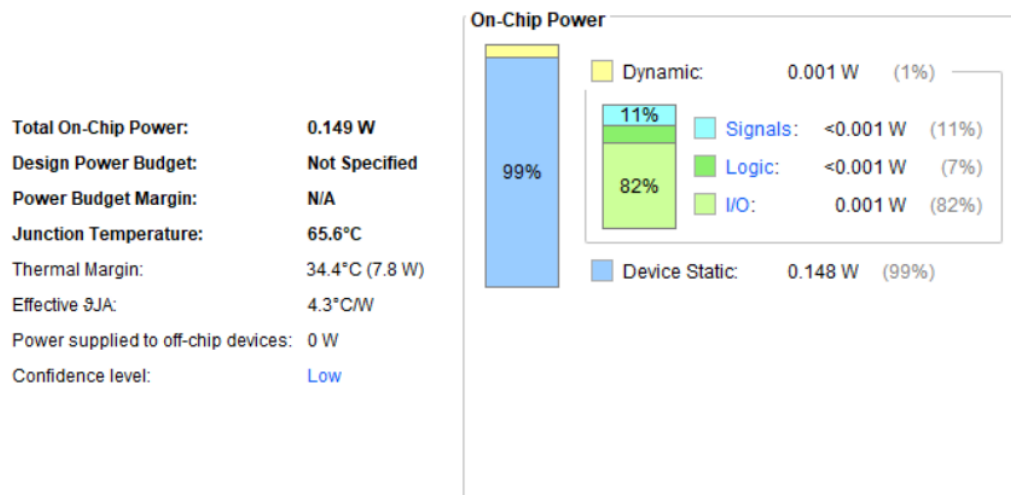


Figure 12 - Power Report

From figure 12, the total power used is 0.149W with 82% of the power being drawn by the IO of the circuit.

3.5 FPGA Implementation

Figure 13 displays the implemented design synthesised to FPGA. The routing between I/O blocks and the LUT's are highlighted.

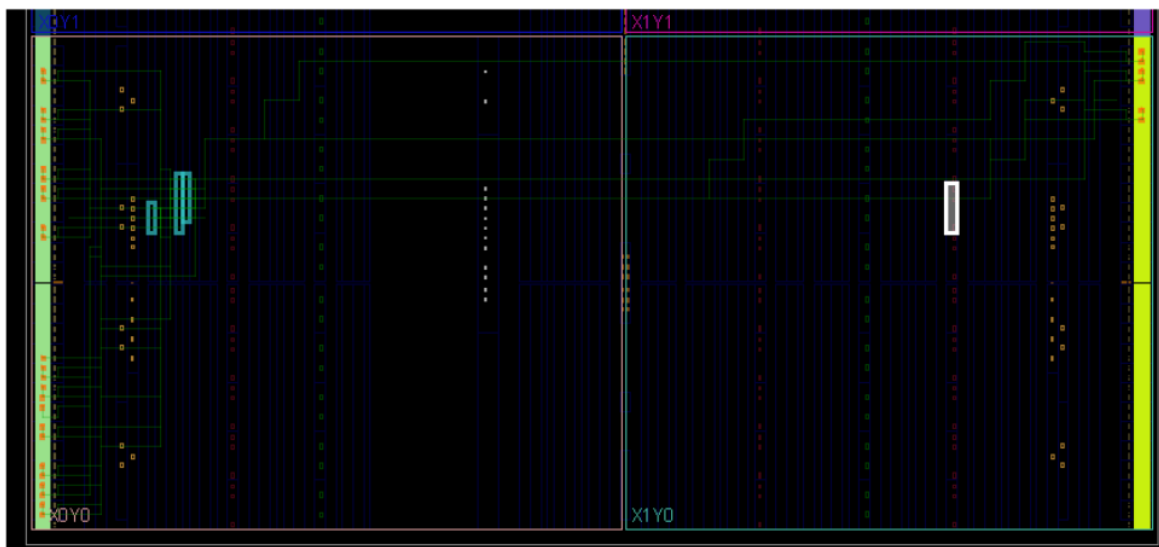


Figure 13 - FPGA Layout – Braun's Multiplier

4.0 Wallace Tree Multiplier

Wallace, an Australian computer scientist proposed a parallel binary multiplication method. This method utilises the simultaneous generation of summands along with a carry propagate adder stage to reduce the delay through the multiplier [4]. The Wallace tree multiplier follows the following steps to generate a final product from an $N \times N$ input, these steps are.

- Use of AND gates to generate a table of n^2 partial products. This is the same as Braun's multiplier shown in table 1.
- Row reduction stages then proceed. Rows of three bits tall are grouped. When three bits reside in this row, they are fed into a full adder. If there are two bits in a row of three, they are fed into a half adder.
- Any singular bits are passed on to the next row reduction level along with the Sum and Carry from the Adders.
- This row reduction is repeated until a row of only two bits remains. These are then summed using a carry propagate adder to generate the final product. [7]

This row reduction methodology is commonly illustrated in the form of a dot diagram. This displays the number of full, and half adders used in the row reduction levels. Figure 13 displays the dot diagram row reduction for an 8 x 8 unsigned binary multiplier.

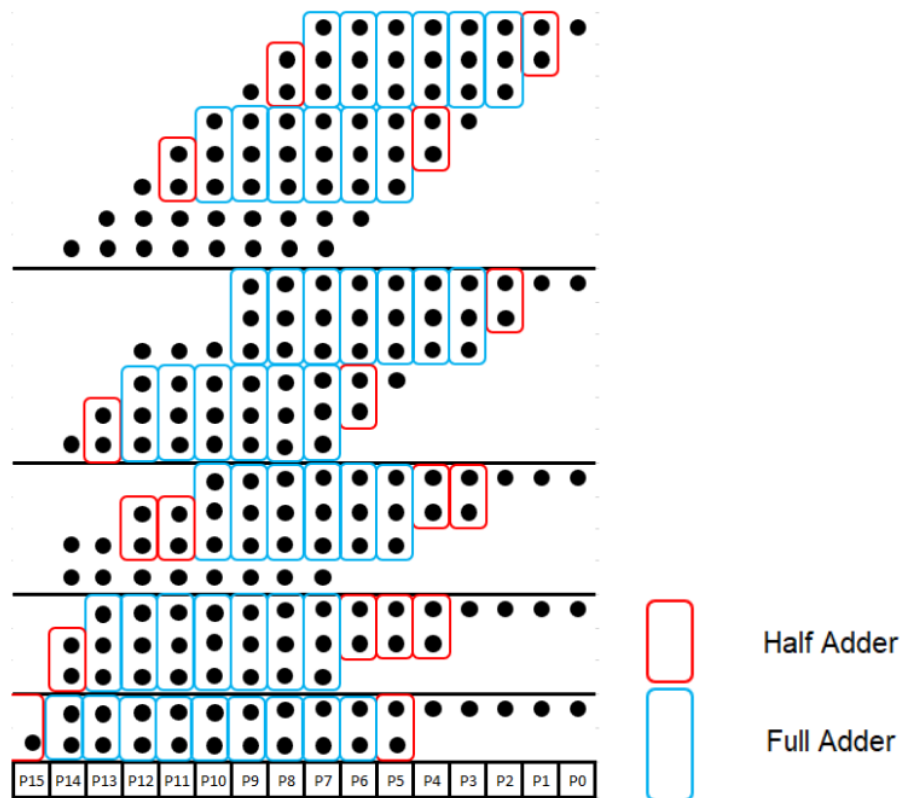
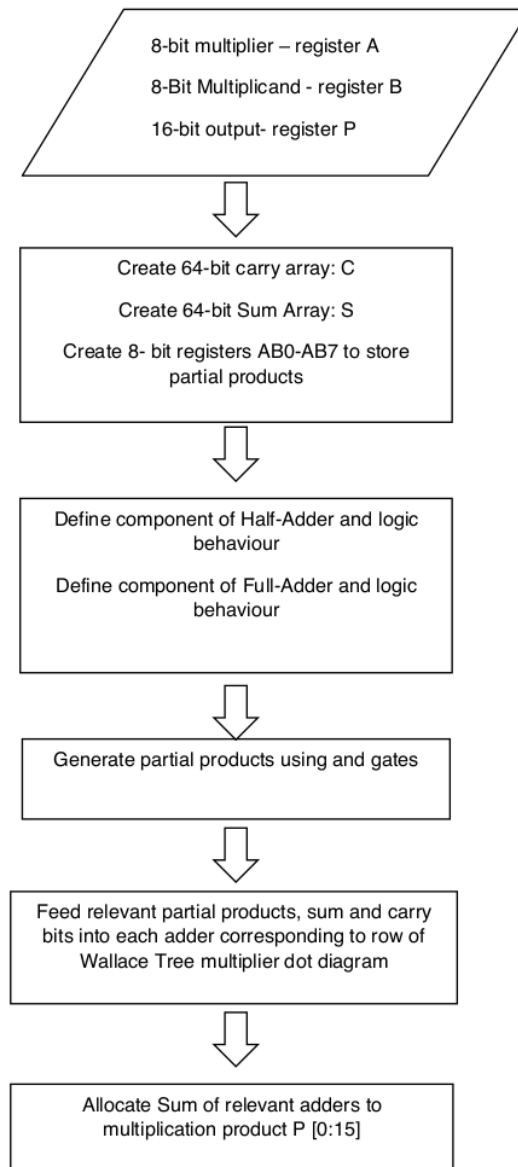


Figure 14 - Wallace tree dot diagram

From the dot diagram, to implement a Wallace tree 8 x 8 multiplier, 47 full adders and 17 half adders are required.

4.1 Wallace Tree VHDL Code

To implement the Wallace Tree multiplier in VHDL code, the following programming methodology has been followed.



Similar to the Braun Multiplier code methodology in section 3, the VHDL for the Wallace tree multiplier has been written as a behavioural model. Half and full adders have been defined as components utilising port maps to make placement of the relevant partial products and carry/save bits easier. Each section of code is defined in stages corresponding with the row reduction stages shown in the dot diagram in figure 14. The VHDL code created for the Wallace Tree can be seen in Appendix B.

4.2 Wallace Tree Behavioural Simulation

The behavioural simulation for the VHDL code utilises the multiplier test bench from appendix C, with the unit under test defined as the Wallace tree multiplier. Figures 15 – 17 show the results of the simulation.

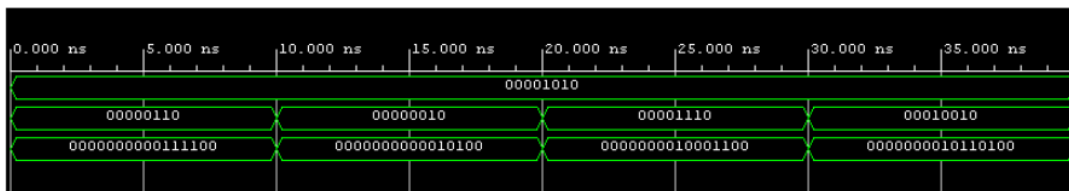


Figure 15 - Sample 1

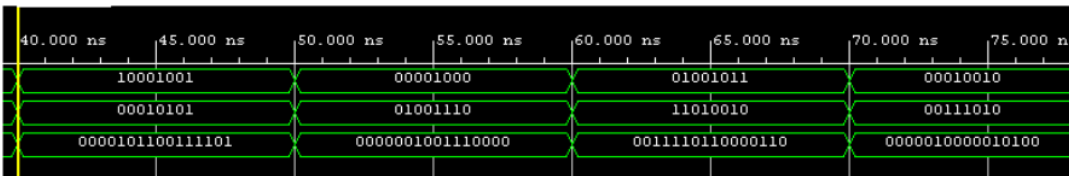


Figure 16 - Sample 2

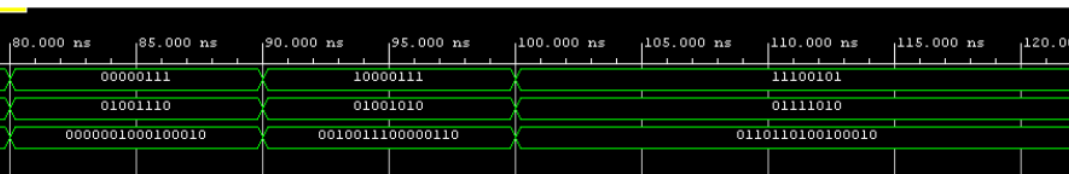


Figure 17 - Sample 3

From the behavioural simulation, it can be seen that the multiplier successfully computes a 16- bit product from two unsigned 8-bit numbers.

4.3 Synthesised/ Implemented Results

Using Vivado XILINX, the Wallace tree VHDL has been synthesised on to an Artix-7 xc7a50tcs9324-2L. Using the report generation tools, power, utilisation and timing information has been extracted.

4.3.1 Wallace Tree Schematic Diagram

Figure 18 displays the logic circuit diagram for the Wallace tree multiplier.

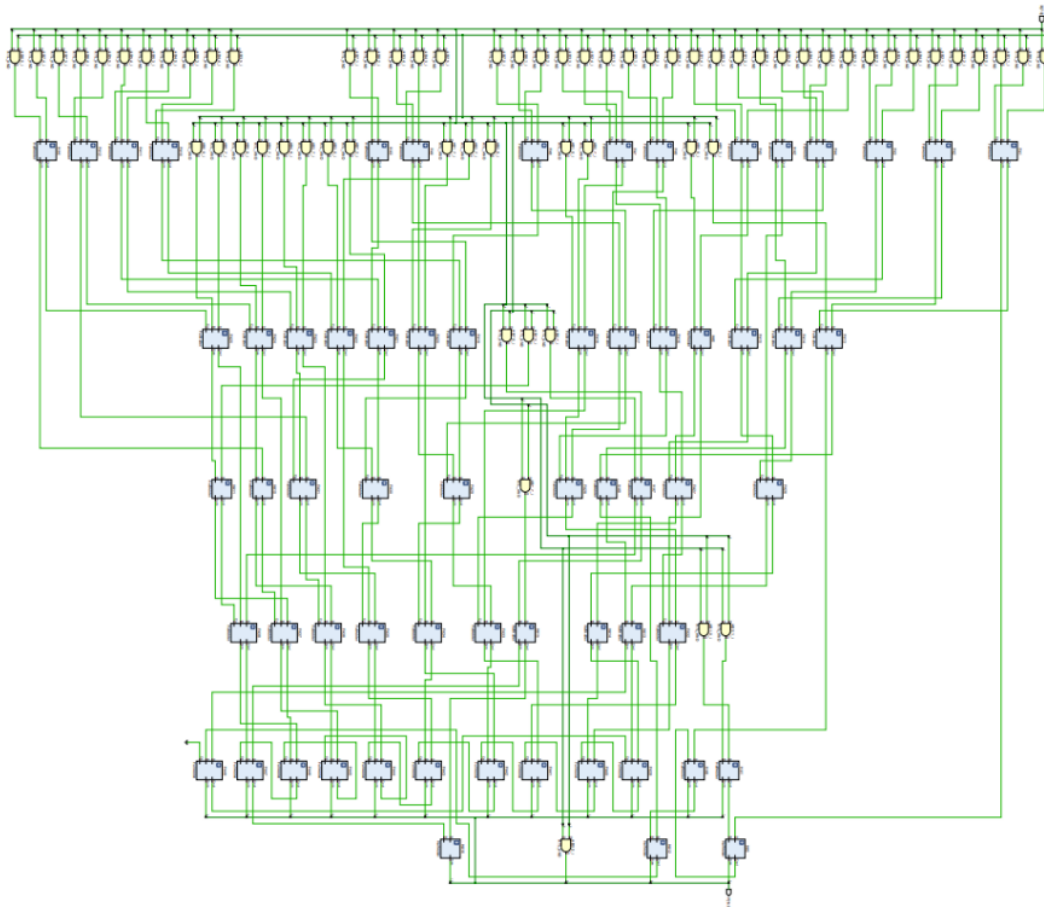


Figure 18 Wallace Tree Logic Diagram

4.3.2 FPGA Utilisation Report

Figure 19 displays the total FPGA utilisation.

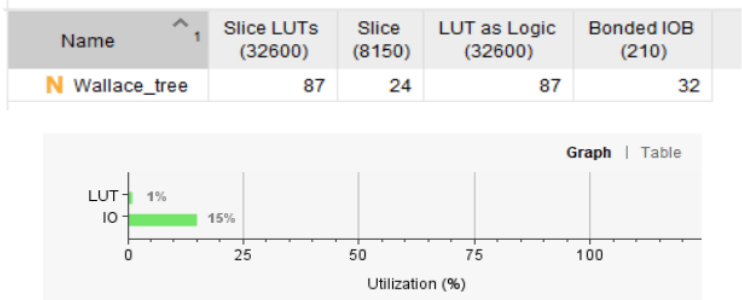


Figure 19 - FPGA Utilisation Report

4.3.3 Wallace Tree Critical Delay Path

Figure 20 shows the critical delay path of the multiplier when synthesised on to FPGA. Figure 21 shows the maximum combinational delay path. A virtual clock of 100ns is configured to carry out the analysis in Vivado with a 30ns max delay parameter between input and outputs. These constraints allow an approximation of critical delay to be calculated.

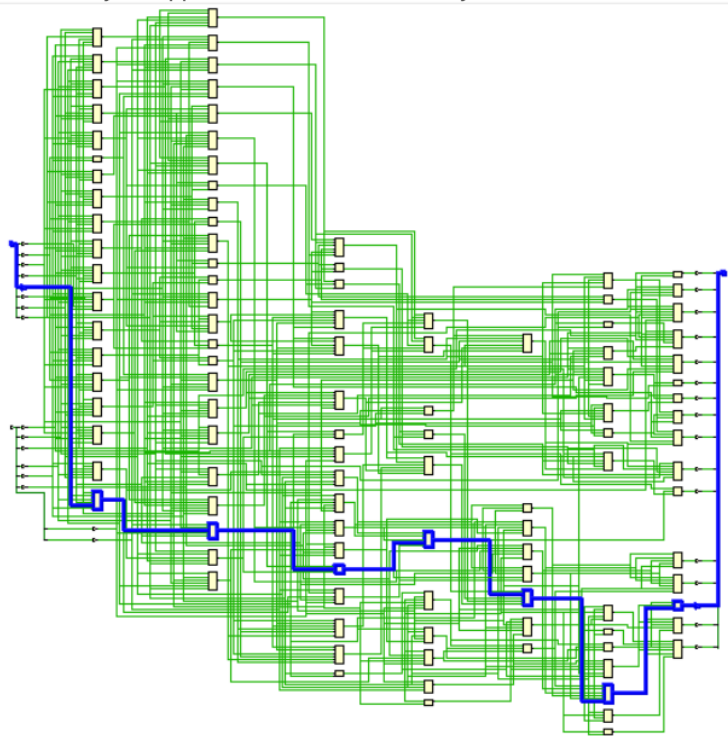


Figure 20 - Maximum delay path

Name	Slack ^{^1}	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	28.265	9	8	16	A[4]	P[13]	11.710	4.533	7.177	100.0
Path 2	28.458	9	8	16	A[4]	P[14]	11.517	4.348	7.169	100.0
Path 3	28.539	9	8	16	A[4]	P[15]	11.436	4.362	7.074	100.0
Path 4	28.875	8	7	16	A[4]	P[12]	11.100	4.257	6.843	100.0

Figure 21 - Combinational Delay

From figure 21, the longest combinational delay path is between A [4] and P [13] at 11.710ns.

4.3.4 FPGA Power Reporting

Using Vivado power utilisation reporting, the estimated power of the device can be seen for the Wallace Tree multiplier. Figure 22 shows the settings for the synthesised circuit tests and figure 23 shows the report results.

Figure 22 - Power Simulation Settings

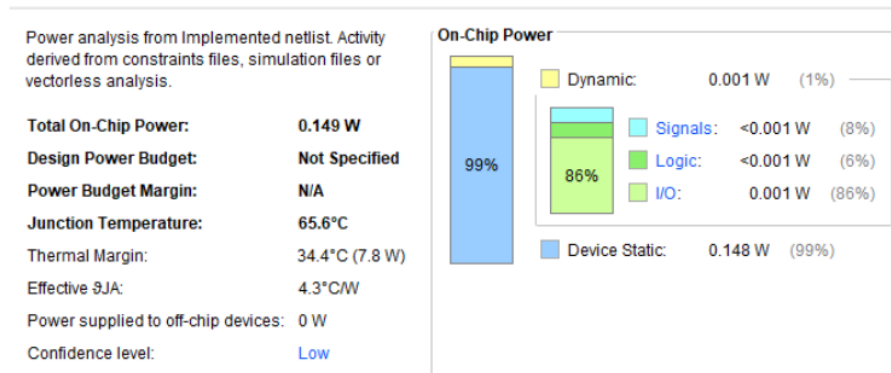


Figure 23 - Wallace Tree Power Report

11

From these reports, 86% of the power required for the Wallace tree is derived from the IO.

4.3.4 FPGA Implemented IO Routing

Figure 24 displays the IO routing on the FPGA package.

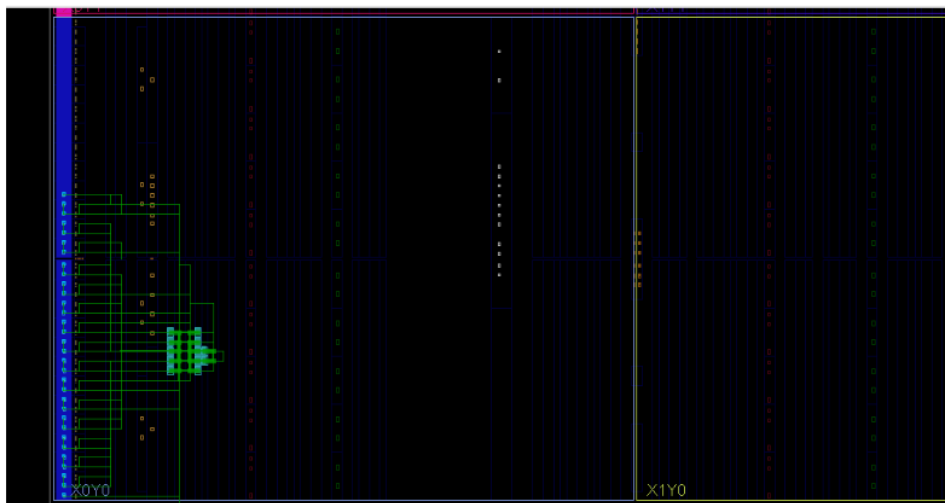


Figure 24 - Implemented Design on FPGA

12

5.0 Comparison of Multiplier Circuits

From the simulation reports generated, the performance characteristics can be compared between the Braun and Wallace tree multipliers. In this section, combinational delays, logic utilisation and power requirements for both multipliers are compared.

5.1 Delay Comparison

Table 2 displays the delay parameter comparison between the simulated 8x8 Wallace and Braun multipliers.

Delay Type	Braun 8 x 8	Wallace 8 x 8
Max Combinational Delay	16.725ns	11.710ns
Logic Delay	5.446ns	4.533ns
Net Delay	11.280ns	7.177ns

Table 2 - Delay Comparison Table

From Table 2, the Wallace tree multiplier is 4.15ns faster than the 8 x 8 Braun architecture. This equates to a 30% improvement in delay through the multiplier.

From the research section in the report, the delay reduction follows typical behaviour for a Wallace tree versus a Braun type multiplier. The delay could be improved by implementing higher order compressors.

5.2 FPGA Utilisation Comparison

Parameter	Braun 8 x 8	Wallace 8 x 8
Slice LUT	90	87
Slices	26	24
LUT as Logic	90	87
Cells	147	143
Nets	163	159
I/O	32	32

Table 3 - Hardware Utilisation

Table 3 displays the FPGA utilisation for both multiplier types. The number of LUT's are higher in the Braun type multiplier as it uses more logic adders to complete the arithmetic. IO remains the same as they both compute an 8 x 8 binary multiplication. The number of cells and nets are reduced in the Wallace tree circuit, however from the schematic diagram in figure 20, the routing is more complex compared to the Braun multiplier in figure 9.

5.3 FPGA Power Comparison

Parameter	Braun 8 x 8	Wallace 8 x 8
Total Power	0.149W	0.149W
Signals (% of Dynamic)	11%	8%
Logic (% of Dynamic)	7%	6%
I/O (% of Dynamic)	82%	86%
IO Utilisation	15%	15%
LUT Utilisation	1%	1%

Table 4 - Power Comparison

From the simulations carried out, the power consumption reports generated that both multipliers draw the same power. This is due to the implemented design having arbitrary initial simulation data for power reporting. What can be seen is the percentage of the dynamic power taken by differing aspects of the designs on the FPGA. The Wallace tree required less power from the logic and signals over the Braun method. Typically, Wallace tree multipliers use less power over Braun multipliers due to the lower number of adders and all of the summands being created simultaneously. The shorter delay results in lower power. The complexity of the Wallace tree design may lead to power dissipation issues in reality due to the large area they occupy. IO utilisation remains the same between both methods due to the constant size of the inputs and output.

6.0 Conclusion

From the simulated VHDL code multipliers, the Wallace tree multiplier is the fastest at 11.710ns and consumes the lowest power. The Wallace Tree uses a lower number of LUT's at 87 vs 90 for a Braun Multiplier. The Braun multiplier works at a slower speed but has reduced complexity, however its stability may suffer when expanded to larger multiplications.

As multipliers are striving to reduce LUT usage, increase speeds, and reduce power, methods to improve tree type multipliers such as Wallace and Dadda Tree are being researched. This includes using higher order compressors to reduce computation times.

Bibliography

- [1] A. Y. H. Khaldoun M. Mhaidat, "A New Efficient Reduction Scheme to Implement Tree Multipliers on FPGAs," in *9th International Design and Test Symposium*, Algeria, 2014.
- [2] B. V. Anitha R, "Braun's Multiplier Implementation using FPGA with Bypassing Techniques," *International Journal of VLSI design & Communication Systems*, vol. 2, no. 3, pp. 201-212, 2011.
- [3] M. H. A. M. M. N. Muhammad H. Rais, "FPGA Realization of Braun's Multipliers," in *Proceedings of The International Multi Conference of Engineers and Computer Scientists*, Hong Kong, 2012.
- [4] C. Wallace, "A Suggestion for a Fast Multiplier," *IEEE TRANSACTIONS ON ELECTRONIC COMPUTERS*, vol. 13, no. 1, pp. 14-17, 1964.
- [5] T. K. M. P.N.V.K. Hasini, "A Novel high-speed transistorized 8x8 Multiplier using 4-2 Compressors," *International Journal of Engineering Research and General Science*, vol. 3, no. 2, pp. 359-365, 2015.
- [6] S. B. G. Hemalatha K N, "Efficient Design of Compact 8-bit Wallace Tree Multiplier Using Reversible Logic," *International Journal of Engineering and Manufacturing*, vol. 4, pp. 29-36, 2022.
- [7] J. K. G. S. T. S. Himanshu Bansal, "Wallace Tree Multiplier Designs: A Performance Comparison Review," *Innovative Systems Design and Engineering*, vol. 5, no. 5, pp. 60-67, 2014.
- [8] "A Review Paper on Comparison of Multipliers based on Performance Parameters," *International Conference on Advances in Science and Technology*, Vols. ICASST-2014, no. 3, pp. 6-9, 2015.
- [9] C. A. MADHU GUDHIMETLA, "Comparison of Different Types of Multipliers With Respect to Speed, Area, and Power," *International Journal of Industrial Electronics and Electrical Engineering*, vol. 5, no. 12, pp. 83-88, 2017.
- [10] K. A. K. D. S. R. R. S. R. H. R. S. C. R. V. Kunjan D. Shinde, "A Novel Approach to Design Braun Array Multiplier Using Paralell Prefix Adders for Parallel Processing Architectures," in *Soft Computing Systems*, Singapore, Springer Singapore, 2018, pp. 602-614.
- [11] P. K. Rajesh Kumar Dwivedi, "VHDL Implementation of Fastest Braun's Multiplier," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 3, no. 5, pp. 2843-2846, 2015.

Appendix A: Braun Multiplier VHDL Code

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

entity Braun is
Port (A:in bit_vector(7 downto 0); --m multiplier
      B:in bit_vector(7 downto 0); -- n Multiplicand
      P:out bit_vector (15 downto 0)); -- P Product Output Terms
end Braun;

architecture Behavioural of Braun is
signal C1,C2,C3,C4,C5,C6,C7: bit_vector(8 downto 0); -- Carry bit arrays from adders,
corresponding to rows of the array multiplier

signal S1,S2,S3,S4,S5,S6,S7: bit_vector(7 downto 0); -- Sum bit arrays form adders,
corresponding to columns of the array multiplier

signal AB0,AB1,AB2,AB3,AB4,AB5,AB6,AB7: bit_vector(7 downto 0); -- These AB vectors store
all of the and terms of the multiplier and multiplicand terms to create each partial
product, e.g A(0) and B(0)

component FullAdder --Full adder component to build model
Port (A,B,Cin: in bit;
      Cout,Sum: out bit);
end component;

component HalfAdder -- Half adder component to build model
Port(A,B: in bit;
      Cout,Sum: out bit);
end component;

begin

AB0(0) <= A(0)and B(0); AB1(0) <= A(1)and B(0); AB2(0) <= A(2)and B(0); AB3(0) <= A(3)and
B(0);
AB0(1) <= A(0)and B(1); AB1(1) <= A(1)and B(1); AB2(1) <= A(2)and B(1); AB3(1) <= A(3)and
B(1);
AB0(2) <= A(0)and B(2); AB1(2) <= A(1)and B(2); AB2(2) <= A(2)and B(2); AB3(2) <= A(3)and
B(2);
AB0(3) <= A(0)and B(3); AB1(3) <= A(1)and B(3); AB2(3) <= A(2)and B(3); AB3(3) <= A(3)and
B(3);
AB0(4) <= A(0)and B(4); AB1(4) <= A(1)and B(4); AB2(4) <= A(2)and B(4); AB3(4) <= A(3)and
B(4);
```

AB0(5) <= A(0)and B(5); AB1(5) <= A(1)and B(5); AB2(5) <= A(2)and B(5); AB3(5) <= A(3)and B(5);

AB0(6) <= A(0)and B(6); AB1(6) <= A(1)and B(6); AB2(6) <= A(2)and B(6); AB3(6) <= A(3)and B(6);

AB0(7) <= A(0)and B(7); AB1(7) <= A(1)and B(7); AB2(7) <= A(2)and B(7); AB3(7) <= A(3)and B(7);

AB4(0) <= A(4)and B(0); AB5(0) <= A(5)and B(0); AB6(0) <= A(6)and B(0); AB7(0) <= A(7)and B(0);

AB4(1) <= A(4)and B(1); AB5(1) <= A(5)and B(1); AB6(1) <= A(6)and B(1); AB7(1) <= A(7)and B(1);

AB4(2) <= A(4)and B(2); AB5(2) <= A(5)and B(2); AB6(2) <= A(6)and B(2); AB7(2) <= A(7)and B(2);

AB4(3) <= A(4)and B(3); AB5(3) <= A(5)and B(3); AB6(3) <= A(6)and B(3); AB7(3) <= A(7)and B(3);

AB4(4) <= A(4)and B(4); AB5(4) <= A(5)and B(4); AB6(4) <= A(6)and B(4); AB7(4) <= A(7)and B(4);

AB4(5) <= A(4)and B(5); AB5(5) <= A(5)and B(5); AB6(5) <= A(6)and B(5); AB7(5) <= A(7)and B(5);

AB4(6) <= A(4)and B(6); AB5(6) <= A(5)and B(6); AB6(6) <= A(6)and B(6); AB7(6) <= A(7)and B(6);

AB4(7) <= A(4)and B(7); AB5(7) <= A(5)and B(7); AB6(7) <= A(6)and B(7); AB7(7) <= A(7)and B(7);

HA1: HalfAdder port map (A=> AB0(1),B => AB1(0),Cout=> C1(0),Sum => S1(0));

HA2: HalfAdder port map (A=> AB0(2),B => S1(1),Cout=> C2(0),Sum => S2(0));

HA3: HalfAdder port map (A=> AB0(3),B => S2(1),Cout=> C3(0),Sum => S3(0));

HA4: HalfAdder port map (A=> AB0(4),B => S3(1),Cout=> C4(0),Sum => S4(0));

HA5: HalfAdder port map (A=> AB0(5),B => S4(1),Cout=> C5(0),Sum => S5(0));

HA6: HalfAdder port map (A=> AB0(6),B => S5(1),Cout=> C6(0),Sum => S6(0));

HA7: HalfAdder port map (A=> AB0(7),B => S6(1),Cout=> C7(0),Sum => S7(0));

HA8: HalfAdder port map (A=> AB7(1),B => C1(6),Cout=> C1(7),Sum => S1(7));

FA1: FullAdder port map (A=> AB1(1),B => AB2(0),Cin=> C1(0),Cout=> C1(1),Sum => S1(1));

FA2: FullAdder port map (A=> AB1(2),B => S1(2),Cin => C2(0),Cout=> C2(1),Sum => S2(1));

FA3: FullAdder port map (A=> AB1(3),B => S2(2),Cin => C3(0),Cout=> C3(1),Sum => S3(1));

FA4: FullAdder port map (A=> AB1(4),B => S3(2),Cin => C4(0),Cout=> C4(1),Sum => S4(1));

FA5: FullAdder port map (A=> AB1(5),B => S4(2),Cin => C5(0),Cout=> C5(1),Sum => S5(1));

FA6: FullAdder port map (A=> AB1(6),B => S5(2),Cin => C6(0),Cout=> C6(1),Sum => S6(1));

FA7: FullAdder port map (A=> AB1(7),B => S6(2),Cin => C7(0),Cout=> C7(1),Sum => S7(1));

```

FA8: FullAdder port map (A=> AB2(1),B => AB3(0),Cin=> C1(1),Cout=> C1(2),Sum => S1(2));
FA9: FullAdder port map (A=> AB2(2),B => S1(3),Cin=> C2(1),Cout=> C2(2),Sum => S2(2));
FA10: FullAdder port map (A=> AB2(3),B => S2(3),Cin=> C3(1),Cout=> C3(2),Sum => S3(2));
FA11: FullAdder port map (A=> AB2(4),B => S3(3),Cin=> C4(1),Cout=> C4(2),Sum => S4(2));
FA12: FullAdder port map (A=> AB2(5),B => S4(3),Cin=> C5(1),Cout=> C5(2),Sum => S5(2));
FA13: FullAdder port map (A=> AB2(6),B => S5(3),Cin=> C6(1),Cout=> C6(2),Sum => S6(2)); --
FA14: FullAdder port map (A=> AB2(7),B => S6(3),Cin=> C7(1),Cout=> C7(2),Sum => S7(2));

FA15: FullAdder port map (A=> AB3(1),B => AB4(0),Cin=> C1(2),Cout=> C1(3),Sum => S1(3));
FA16: FullAdder port map (A=> AB3(2),B => S1(4),Cin=> C2(2),Cout=> C2(3),Sum => S2(3));
FA17: FullAdder port map (A=> AB3(3),B => S2(4),Cin=> C3(2),Cout=> C3(3),Sum => S3(3));
FA18: FullAdder port map (A=> AB3(4),B => S3(4),Cin=> C4(2),Cout=> C4(3),Sum => S4(3));
FA19: FullAdder port map (A=> AB3(5),B => S4(4),Cin=> C5(2),Cout=> C5(3),Sum => S5(3));
FA20: FullAdder port map (A=> AB3(6),B => S5(4),Cin=> C6(2),Cout=> C6(3),Sum => S6(3));
FA21: FullAdder port map (A=> AB3(7),B => S6(4),Cin=> C7(2),Cout=> C7(3),Sum => S7(3));

FA22: FullAdder port map (A=> AB4(1),B => AB5(0),Cin=> C1(3),Cout=> C1(4),Sum => S1(4));
FA23: FullAdder port map (A=> AB4(2),B => S1(5),Cin=> C2(3),Cout=> C2(4),Sum => S2(4));
FA24: FullAdder port map (A=> AB4(3),B => S2(5),Cin=> C3(3),Cout=> C3(4),Sum => S3(4));
FA25: FullAdder port map (A=> AB4(4),B => S3(5),Cin=> C4(3),Cout=> C4(4),Sum => S4(4));
FA26: FullAdder port map (A=> AB4(5),B => S4(5),Cin=> C5(3),Cout=> C5(4),Sum => S5(4));
FA27: FullAdder port map (A=> AB4(6),B => S5(5),Cin=> C6(3),Cout=> C6(4),Sum => S6(4));
FA28: FullAdder port map (A=> AB4(7),B => S6(5),Cin=> C7(3),Cout=> C7(4),Sum => S7(4));

FA29: FullAdder port map (A=> AB5(1),B => AB6(0),Cin=> C1(4),Cout=> C1(5),Sum => S1(5));
FA30: FullAdder port map (A=> AB5(2),B => S1(6),Cin=> C2(4),Cout=> C2(5),Sum => S2(5));
FA31: FullAdder port map (A=> AB5(3),B => S2(6),Cin=> C3(4),Cout=> C3(5),Sum => S3(5));
FA32: FullAdder port map (A=> AB5(4),B => S3(6),Cin=> C4(4),Cout=> C4(5),Sum => S4(5));
FA33: FullAdder port map (A=> AB5(5),B => S4(6),Cin=> C5(4),Cout=> C5(5),Sum => S5(5));
FA34: FullAdder port map (A=> AB5(6),B => S5(6),Cin=> C6(4),Cout=> C6(5),Sum => S6(5));
FA35: FullAdder port map (A=> AB5(7),B => S6(6),Cin=> C7(4),Cout=> C7(5),Sum => S7(5));

FA36: FullAdder port map (A=> AB6(1),B => AB7(0),Cin=> C1(5),Cout=> C1(6),Sum => S1(6));
FA37: FullAdder port map (A=> AB6(2),B => S1(7),Cin=> C2(5),Cout=> C2(6),Sum => S2(6));
FA38: FullAdder port map (A=> AB6(3),B => S2(7),Cin=> C3(5),Cout=> C3(6),Sum => S3(6));
FA39: FullAdder port map (A=> AB6(4),B => S3(7),Cin=> C4(5),Cout=> C4(6),Sum => S4(6));

```

```

FA40: FullAdder port map (A=> AB6(5),B => S4(7),Cin=> C5(5),Cout=> C5(6),Sum => S5(6));
FA41: FullAdder port map (A=> AB6(6),B => S5(7),Cin=> C6(5),Cout=> C6(6),Sum => S6(6));
FA42: FullAdder port map (A=> AB6(7),B => S6(7),Cin=> C7(5),Cout=> C7(6),Sum => S7(6));

FA44: FullAdder port map (A=> AB7(2),B => C1(7),Cin=> C2(6),Cout=> C2(7),Sum => S2(7));
FA45: FullAdder port map (A=> AB7(3),B => C2(7),Cin=> C3(6),Cout=> C3(7),Sum => S3(7));
FA46: FullAdder port map (A=> AB7(4),B => C3(7),Cin=> C4(6),Cout=> C4(7),Sum => S4(7));
FA47: FullAdder port map (A=> AB7(5),B => C4(7),Cin=> C5(6),Cout=> C5(7),Sum => S5(7));
FA48: FullAdder port map (A=> AB7(6),B => C5(7),Cin=> C6(6),Cout=> C6(7),Sum => S6(7));
FA49: FullAdder port map (A=> AB7(7),B => C6(7),Cin=> C7(6),Cout=> C7(7),Sum => S7(7));

-- Partial Product output, 16 bit binary unsigned number
P(0) <= AB0(0);
P(1) <= S1(0);
P(2) <= S2(0);
P(3) <= S3(0);
P(4) <= S4(0);
P(5) <= S5(0);
P(6) <= S6(0);
P(7) <= S7(0);
P(8) <= S7(1);
P(9) <= S7(2);
P(10) <= S7(3);
P(11) <= S7(4);
P(12) <= S7(5);
P(13) <= S7(6);
P(14) <= S7(7);
P(15) <= C7(7);

end Behavioural;

entity FullAdder is
Port (A,B,Cin: in bit;
      Cout,Sum: out bit);
end FullAdder;

architecture equations of FullAdder is

```

```
begin
Sum <= A xor B xor Cin;
Cout <= ((A and B) or (B and Cin) or (A and Cin));
end equations;

entity HalfAdder is
Port (A,B: in bit;
      Cout,Sum: out bit);
end HalfAdder;

architecture equations of HalfAdder is
begin
Sum<= A xor B;
Cout <= A and B;
end equations;
```

Appendix B: Wallace Tree VHDL Code

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Wallace_tree is
Port (A:in bit_vector(7 downto 0); --m multiplier
      B:in bit_vector(7 downto 0); -- n Multiplicand
      P:out bit_vector (15 downto 0)); -- P Product Output Terms
end Wallace_tree;

architecture Behavioral of Wallace_tree is

signal C: bit_vector(64 downto 1); -- Carry bit arrays from adders, corresponding to rows
of the array multiplier

signal S: bit_vector(64 downto 1); -- Sum bit arrays form adders, corresponding to columns
of the array multiplier

signal AB0,AB1,AB2,AB3,AB4,AB5,AB6,AB7: bit_vector(7 downto 0); -- These AB vectors store
all of the and terms of the multiplier and multiplicand terms to create each partial
product, e.g A(0) and B(0)

component FullAdder --Full adder component to build model
Port (A,B,Cin: in bit;
      Cout,Sum: out bit);
end component;

component HalfAdder -- Half adder component to build model
Port(A,B: in bit;
      Cout,Sum: out bit);
```

```

end component;

begin
--Stage 1 Creating partial products
AB0(0) <= A(0)and B(0); AB1(0) <= A(1)and B(0); AB2(0) <= A(2)and B(0); AB3(0) <= A(3)and
B(0);
AB0(1) <= A(0)and B(1); AB1(1) <= A(1)and B(1); AB2(1) <= A(2)and B(1); AB3(1) <= A(3)and
B(1);
AB0(2) <= A(0)and B(2); AB1(2) <= A(1)and B(2); AB2(2) <= A(2)and B(2); AB3(2) <= A(3)and
B(2);
AB0(3) <= A(0)and B(3); AB1(3) <= A(1)and B(3); AB2(3) <= A(2)and B(3); AB3(3) <= A(3)and
B(3);
AB0(4) <= A(0)and B(4); AB1(4) <= A(1)and B(4); AB2(4) <= A(2)and B(4); AB3(4) <= A(3)and
B(4);
AB0(5) <= A(0)and B(5); AB1(5) <= A(1)and B(5); AB2(5) <= A(2)and B(5); AB3(5) <= A(3)and
B(5);
AB0(6) <= A(0)and B(6); AB1(6) <= A(1)and B(6); AB2(6) <= A(2)and B(6); AB3(6) <= A(3)and
B(6);
AB0(7) <= A(0)and B(7); AB1(7) <= A(1)and B(7); AB2(7) <= A(2)and B(7); AB3(7) <= A(3)and
B(7);

AB4(0) <= A(4)and B(0); AB5(0) <= A(5)and B(0); AB6(0) <= A(6)and B(0); AB7(0) <= A(7)and
B(0);
AB4(1) <= A(4)and B(1); AB5(1) <= A(5)and B(1); AB6(1) <= A(6)and B(1); AB7(1) <= A(7)and
B(1);
AB4(2) <= A(4)and B(2); AB5(2) <= A(5)and B(2); AB6(2) <= A(6)and B(2); AB7(2) <= A(7)and
B(2);
AB4(3) <= A(4)and B(3); AB5(3) <= A(5)and B(3); AB6(3) <= A(6)and B(3); AB7(3) <= A(7)and
B(3);
AB4(4) <= A(4)and B(4); AB5(4) <= A(5)and B(4); AB6(4) <= A(6)and B(4); AB7(4) <= A(7)and
B(4);
AB4(5) <= A(4)and B(5); AB5(5) <= A(5)and B(5); AB6(5) <= A(6)and B(5); AB7(5) <= A(7)and
B(5);
AB4(6) <= A(4)and B(6); AB5(6) <= A(5)and B(6); AB6(6) <= A(6)and B(6); AB7(6) <= A(7)and
B(6);
AB4(7) <= A(4)and B(7); AB5(7) <= A(5)and B(7); AB6(7) <= A(6)and B(7); AB7(7) <= A(7)and
B(7);

--Stage 2:
HA1: HalfAdder port map(A=> AB1(0),B => AB0(1),Cout=> C(1),Sum => S(1));
FA1: FullAdder port map (A=> AB2(0),B => AB1(1),Cin=> AB0(2),Cout=> C(2),Sum => S(2));
FA2: FullAdder port map (A=> AB3(0),B => AB2(1),Cin=> AB1(2),Cout=> C(3),Sum => S(3));
FA3: FullAdder port map (A=> AB4(0),B => AB3(1),Cin=> AB2(2),Cout=> C(4),Sum => S(4));

```

```

HA2: HalfAdder port map(A=> AB1(3),B => AB0(4),Cout=> C(5),Sum => S(5));
FA4: FullAdder port map (A=> AB5(0),B => AB4(1),Cin=> AB3(2),Cout=> C(6),Sum => S(6));
FA5: FullAdder port map (A=> AB2(3),B => AB1(4),Cin=> AB0(5),Cout=> C(7),Sum => S(7));
FA6: FullAdder port map (A=> AB6(0),B => AB5(1),Cin=> AB4(2),Cout=> C(8),Sum => S(8));
FA7: FullAdder port map (A=> AB3(3),B => AB2(4),Cin=> AB1(5),Cout=> C(9),Sum => S(9));
FA8: FullAdder port map (A=> AB7(0),B => AB6(1),Cin=> AB5(2),Cout=> C(10),Sum => S(10));
FA9: FullAdder port map (A=> AB4(3),B => AB3(4),Cin=> AB2(5),Cout=> C(11),Sum => S(11));
HA3: HalfAdder port map(A=> AB7(1),B => AB6(2),Cout=> C(12),Sum => S(12));
FA10: FullAdder port map (A=> AB5(3),B => AB4(4),Cin=> AB3(5),Cout=> C(13),Sum => S(13));
FA11: FullAdder port map (A=> AB6(3),B => AB5(4),Cin=> AB4(5),Cout=> C(14),Sum => S(14));
FA12: FullAdder port map (A=> AB7(3),B => AB6(4),Cin=> AB5(5),Cout=> C(15),Sum => S(15));
HA4: HalfAdder port map(A=> AB7(4),B => AB6(5),Cout=> C(16),Sum => S(16));

--Stage 3:
HA5: HalfAdder port map(A=> S(2),B => C(1),Cout=> C(17),Sum => S(17));
FA13: FullAdder port map (A=> S(3),B => AB0(3),Cin=> C(2) ,Cout=> C(18),Sum => S(18));
FA14: FullAdder port map (A=> S(4),B => S(5),Cin=> C(3) ,Cout=> C(19),Sum => S(19));
FA15: FullAdder port map (A=> S(6),B => S(7),Cin=> C(4) ,Cout=> C(20),Sum => S(20));
FA16: FullAdder port map (A=> S(8),B => S(9),Cin=> C(6) ,Cout=> C(21),Sum => S(21));
HA6: HalfAdder port map(A=> C(7),B => AB0(6),Cout=> C(22),Sum => S(22));
FA17: FullAdder port map (A=> S(10),B => S(11),Cin=> C(8) ,Cout=> C(23),Sum => S(23));
FA18: FullAdder port map (A=> C(9),B => AB0(7),Cin=>AB1(6),Cout=> C(24),Sum => S(24));
FA19: FullAdder port map (A=> S(12),B => S(13),Cin=> C(10) ,Cout=> C(25),Sum => S(25));
FA20: FullAdder port map (A=> AB1(7),B => C(11),Cin=> AB2(6),Cout=> C(26),Sum => S(26));
FA21: FullAdder port map (A=> AB7(2),B => S(14),Cin=> C(12) ,Cout=> C(27),Sum => S(27));
FA22: FullAdder port map (A=> AB2(7),B => AB3(6), Cin=> C(13),Cout=> C(28),Sum => S(28));
FA23: FullAdder port map (A=> AB3(7),B => AB4(6),Cin=> C(14),Cout=> C(29),Sum => S(29));
FA24: FullAdder port map (A=> AB4(7),B => AB5(6),Cin=> C(15),Cout=> C(30),Sum => S(30));
FA25: FullAdder port map (A=> AB5(7),B => AB6(6),Cin=> C(16),Cout=> C(31),Sum => S(31));
HA7: HalfAdder port map(A=> AB7(6),B => AB6(7),Cout=> C(32),Sum => S(32));

--Stage 4:
HA8: HalfAdder port map(A=> S(18),B => C(17),Cout=> C(33),Sum => S(33));
HA9: HalfAdder port map(A=> S(19),B => C(18),Cout=> C(34),Sum => S(34));
FA26: FullAdder port map (A=> S(20),B => C(5) ,Cin=> C(19),Cout=> C(35),Sum => S(35));
FA27: FullAdder port map (A=> S(21),B => S(22),Cin=> C(20),Cout=> C(36),Sum => S(36));
FA28: FullAdder port map (A=> S(23),B => S(24),Cin=> C(21),Cout=> C(37),Sum => S(37));

```

```

FA29: FullAdder port map (A=> S(25),B => S(26),Cin=> C(23),Cout=> C(38),Sum => S(38));
FA30: FullAdder port map (A=> S(27),B => S(28),Cin=> C(25),Cout=> C(39),Sum => S(39));
FA31: FullAdder port map (A=> S(15),B => S(29),Cin=> C(27),Cout=> C(40),Sum => S(40));
HA10: HalfAdder port map(A=> S(16),B => S(30),Cout=> C(41),Sum => S(41));
HA11: HalfAdder port map(A=> AB7(5),B => S(31),Cout=> C(42),Sum => S(42));
--Stage 5:
HA12: HalfAdder port map(A=> S(34),B => C(33),Cout=> C(43),Sum => S(43));
HA13: HalfAdder port map(A=> S(35),B => C(34),Cout=> C(44),Sum => S(44));
HA14: HalfAdder port map(A=> S(36),B => C(35),Cout=> C(45),Sum => S(45));
FA32: FullAdder port map (A=> S(37),B => C(22),Cin=> C(36),Cout=> C(46),Sum => S(46));
FA33: FullAdder port map (A=> S(38),B => C(24),Cin=> C(37),Cout=> C(47),Sum => S(47));
FA34: FullAdder port map (A=> S(39),B => C(26),Cin=> C(38),Cout=> C(48),Sum => S(48));
FA35: FullAdder port map (A=> S(40),B => C(28),Cin=> C(39),Cout=> C(49),Sum => S(49));
FA36: FullAdder port map (A=> S(41),B => C(29),Cin=> C(40),Cout=> C(50),Sum => S(50));
FA37: FullAdder port map (A=> S(42),B => C(30),Cin=> C(41),Cout=> C(51),Sum => S(51));
FA38: FullAdder port map (A=> S(32),B => C(31),Cin=> C(42),Cout=> C(52),Sum => S(52));
HA15: HalfAdder port map(A=> AB7(7),B => C(32),Cout=> C(53),Sum => S(53));

--Final Carry Propagate Adder with full adder
FA48: FullAdder port map (A=> S(44),B => C(43),Cin=> '0',Cout=> C(54),Sum => S(54));
--HA16: HalfAdder port map(A=> S(44),B => C(43),Cout=> C(54),Sum => S(54));
FA39: FullAdder port map (A=> S(45),B => C(44),Cin=> C(54),Cout=> C(55),Sum => S(55));
FA40: FullAdder port map (A=> S(46),B => C(45),Cin=> C(55),Cout=> C(56),Sum => S(56));
FA41: FullAdder port map (A=> S(47),B => C(46),Cin=> C(56),Cout=> C(57),Sum => S(57));
FA42: FullAdder port map (A=> S(48),B => C(47),Cin=> C(57),Cout=> C(58),Sum => S(58));
FA43: FullAdder port map (A=> S(49),B => C(48),Cin=> C(58),Cout=> C(59),Sum => S(59));
FA44: FullAdder port map (A=> S(50),B => C(49),Cin=> C(59),Cout=> C(60),Sum => S(60));
FA45: FullAdder port map (A=> S(51),B => C(50),Cin=> C(60),Cout=> C(61),Sum => S(61));
FA46: FullAdder port map (A=> S(52),B => C(51),Cin=> C(61),Cout=> C(62),Sum => S(62));
FA47: FullAdder port map (A=> S(53),B => C(52),Cin=> C(62),Cout=> C(63),Sum => S(63));
HA16: HalfAdder port map(A=> C(53),B => C(63),Cout=> C(64),Sum => S(64));

-- Partial Product output, 16 bit binary unsigned number
P(0) <= AB0(0);
P(1) <= S(1);
P(2) <= S(17);
P(3) <= S(33);

```

```
P(4) <= S(43);
```

```
P(5) <= S(54);
```

```
P(6) <= S(55);
```

```
P(7) <= S(56);
```

```
P(8) <= S(57);
```

```
P(9) <= S(58);
```

```
P(10) <= S(59);
```

```
P(11) <= S(60);
```

```
P(12) <= S(61);
```

```
P(13) <= S(62);
```

```
P(14) <= S(63);
```

```
P(15) <= S(64);
```



```
end Behavioral;
```

```
entity FullAdder is
```

```
Port (A,B,Cin: in bit;
```

```
      Cout,Sum: out bit);
```

```
end FullAdder;
```

```
architecture equations of FullAdder is
```

```
begin
```

```
Sum <= A xor B xor Cin;
```

```
Cout <= ((A and B) or (B and Cin) or (A and Cin));
```

```
end equations;
```

```
entity HalfAdder is
```

```
Port (A,B: in bit;
```

```
      Cout,Sum: out bit);
```

```
end HalfAdder;
```

```
architecture equations of HalfAdder is
```

```
begin
```

```
Sum<= A xor B;
```

```
Cout <= A and B;
```

```
end equations;
```

Appendix C: Multiplier Test Bench Code

```
entity Multiplier_tb is
-- Port ( );
end Multiplier_tb;

architecture tb of Multiplier_tb is
component Wallace_tree
Port (A:in bit_vector(7 downto 0); --m multiplier
      B:in bit_vector(7 downto 0); -- n Multiplicand
      P:out bit_vector (15 downto 0)); -- P Product Output Terms
end component;
signal A: bit_vector (7 downto 0);
signal B: bit_vector (7 downto 0);
signal P: bit_vector (15 downto 0);
begin
UUT: Wallace_tree Port map(
A =>A,
B =>B,
P =>P);
Force:process
constant period: time := 10 ns;
begin
A <= "00001010"; -- 10
B <= "00000110"; --6
--Expected Output: 60, 000000000111100
wait for period;
A <= "00001010"; --10
B <= "00000010"; --4
--Expected Output: 40, 000000000101000
wait for period;
A <= "00001010"; --10
B <= "00001110"; --14
--Expected Output: 140, 000000010001100
wait for period;
A <= "00001010"; -- 10
```

```

B <= "00010010"; --18
--Expected Output: 180, 0000000010110100
wait for period;
A <= "10001001"; --137
B <= "00010101"; --21
--Expected Output: 2877, 0000101100111101
wait for period;
A <= "00001000"; --8
B <= "01001110"; --78
--Expected Output: 624, 0000001001110000
wait for period;
A <= "01001011"; -- 75
B <= "11010010"; --210
--Expected Output: 15750, 0011110110000110
wait for period;
A <= "00010010"; --18
B <= "00111010"; --58
--Expected Output: 1044, 0000010000010100
wait for period;
A <= "00000111"; --7
B <= "01001110"; --78
--Expected Output: 546, 0000001000100010
wait for period;
A <= "10000111"; --135
B <= "01001010"; --74
--Expected Output: 9990, 0010011100000110
wait for period;
A <= "11100101"; --229
B <= "01111010"; --122
--Expected Output: 27938, 0110110100100010
wait for period;
wait;
end process;
end tb;

```

Multiplier Assignment

GRADEMARK REPORT

FINAL GRADE

77 / 100

GENERAL COMMENTS

Instructor

a. Introduction to Multipliers/ Various Multipliers/ Advantages and Disadvantages/ Applications of Multipliers (8) Marks 6

Introduction was provided with a clear idea on both the Multipliers. Significance of the Multipliers were captured. Major points are stated clearly and are well supported. It could be better if a brief writings were made addressing different multipliers.

b.

Background Research (8) : Marks

Engagement with an extensive range of relevant and credible literature.

c. Diversity of sources and Referencing Skills(4) Marks :4

High-level referencing skills consistently applied.

a. Task 1:

Identifying the correct Multiplier (4), Architecture of Braun's Multiplier(6) Marks :8

The correct Multiplier is identified. The concepts of Braun's Multiplier is up to the expected depth. Architecture of 8X8 Braun's is well captured. Assumptions of partial signals made is very clear.

b. Task 2:

Dot Diagram(4), Architecture of Wallace Multiplier(6) Marks :7

The Dot Diagram is well Captured. Extensive usage of Full Adders and Half adders were Presented. It is very clear the you have a good knowledge on choosing the correct adder in your architecture. It could be better if the architecture diagram is seen.

a. Finding

**Critical Path, Longest Delay,
Min and Max Delay, Power
Consumption(12).
Marks**

:10

Well Done- Critical evaluation on finding the Critical path, longest min, max delay, Power consumption were made. The range slightly differs with varies respect to the no. of LUTs and I/O used. Based upon the Evidence Based judgements the solutions you have stated are within the expected range (for delay). All the needed screen dumps were found which evidently proves the work of yours. It is good that you have managed to compare the all the criteria's with both the Multipliers.

**b. Innovative
thinking in problem solving and feature
additions(8)
Marks**

:4

Mostly valid arguments and logical judgements on Examining the evidences of the design , its accuracy, precision, relevance, completeness.

a.

Simulation (7)

:7 Marks

Well done -

The simulation results are well captured by using appropriate screen dumps. Minimum 4 samples were found. The usage of test bench were flawless.

b. VHDL

Coding's

(10)

:8 Marks

Good - Both the codes are relevant and produces the expected outputs and results without any errors. The products are matching with the Manual Calculations.

b. Circuit

**Schematic (4)
Marks**

:4

Good- The screen dump provided for the Circuit schematic is neat and precise in both the cases of multipliers. It covers most of the Blocks.

Synthesis**(4)****:3 Marks**

Synthesis is provided. It clearly shows that the you have a very good experience in obtaining the synthesis and Implementation reports. The procedures to capture FPGA synthesis were done perfectly

a. Logical order of ideas, Flow of ideas in the report from the beginning to end with appropriate chapters, sub chapters and continuation(7)

:5**Marks**

Work is coherent, very fluent and is presented proficiently.

b. Definitions, Acronyms and Abbreviations (3)

:2**Marks**

Sufficient Definitions, Acronyms and Abbreviations are captured.

c. Professionalism, Spelling and Grammar, Complete Report, Proper Presentation, Quality of the report(5)

:3**Marks**

High level of professionalism is maintained in the complete report. Proper Presentation and the Quality of the report is appreciable.

Total Marks:**77%****Finding from Codes and Simulations:**

Port and signal declarations are found. Too many expressions were found which may reduce the efficiency of the multiplier. Code runs without error and also produces expected results .Test bench Code is also found.

[Click here to view the feed back Report](#)

PAGE 1

PAGE 2



Comment 1

Introduction was provided with a clear idea on both the Multipliers.

PAGE 3



Comment 2

It could be better if a brief writings were made addressing different multipliers.



Comment 3

The concepts of Braun's Multiplier is up to the expected depth.



Comment 4

The correct Multiplier is identified.

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8



Comment 5

Well done - The simulation results are well captured by using appropriate screen dumps.

PAGE 9

PAGE 10

PAGE 11

PAGE 12



Comment 6

Extensive usage of Full Adders and Half adders were Presented.



Comment 7

Good- The Dot Diagram is well Captured.



Comment 8

Good Job!



Comment 9

Schematics are found



Comment 10

Well Done- Critical evaluation on finding the Critical path, longest delay, were made. The range slightly differs with varies respect to the no. of LUTs and I/O used. Based upon the Evidence Based judgements the solutions you have stated are within the expected range.



Comment 11

Critical evaluation on finding the Power consumption were made.



Comment 12

FPGAs are seen



Comment 13

Right!

PAGE 21



Comment 14

This is unusual.

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32



Comment 15

Both the codes are relevant and produces the expected outputs and results without any errors.

PAGE 33

PAGE 34
