



L6 – Electronics 2 (ACDA6001)

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

David Mathew Lewis

2219439

Electrical and Electronic Engineering

Table of Contents

1. Introduction	4
2. Background Research.....	4
Braun's Multiplier	4
Wallace Tree Multiplier.....	4
Applications of Multipliers	4
3. Flowchart	5
Braun's Diagram.....	6
Wallace Tree Diagram.....	7
4. Methodology.....	7
Requirements Analysis.....	7
Design Strategy	7
Braun's Multiplier	7
Wallace Tree Multiplier.....	7
VHDL Implementation.....	7
Synthesis and FPGA Implementation.....	8
Performance Evaluation.....	8
5. Algorithm Comparison	8
6. Simulation & Testing.....	8
Braun's Multiplier	8
Sample 1.....	9
Sample 2.....	10
Sample 3.....	10
Sample 4.....	11
Interpretation of Results.....	11
Wallace Tree Multiplier.....	11
Sample 1.....	12
Sample 2.....	13
Sample 3.....	14
Sample 4.....	14
Interpretation of Results.....	15
Performance Evaluation – Braun's Multiplier.....	15
Manual Delay Estimation	15

Vivado Timing Summary (Post Synthesis)	15
Power Consumption (Post Synthesis)	15
Report Utilisation	16
Summary of Braun's Performance	17
Performance Evaluation – Wallace Tree.....	17
Manual Delay Estimation	17
Vivado Timing Summary (Post Synthesis)	17
Power Consumption (Post Synthesis)	17
Report Utilisation	18
Summary of Wallace Tree Performance	19
Multiplier Performance Comparison	19
7. I/O Planning	20
I/O Planning Summary	21
8. Device View	21
Braun's Multiplier	21
Wallace Tree Multiplier.....	21
Comparison Summary.....	22
9. Schematics	23
10. Pins Layout	26
Braun's Multiplier	26
Wallace Tree	27
11. Recommendations	28
12. Conclusion.....	28
13. References	29
14. Appendices.....	29
Braun's Code	29
Wallace Tree Code	41

1. Introduction

In digital signal processing and embedded system applications, multiplication is a fundamental arithmetic operation, especially in high-speed and power-efficient systems. Efficient multiplier design is crucial in various computer systems including image processing, cryptography, and real-time control systems such as door entry systems. Among the many multiplication techniques available, Braun’s Multiplier and the Wallace Tree Multiplier are two prominent methods used in hardware implementations due to their parallelism and performance.

This assignment aims to critically evaluate, synthesise, and compare the performance of these two multiplier architectures using VHDL (VHSIC Hardware Description Language). Through the implementation of an 8-bit multiplier using both techniques, this project explores hardware design principles, timing considerations, and resource utilization. Additionally, this project synthesises the designs on FPGA using Xilinx Vivado to assess real-world performance metrics such as power consumption and logic utilisation.

2. Background Research

Braun’s Multiplier

Braun’s multiplier, also known as an array multiplier, is a regular structure that implements unsigned multiplication. It forms the product by generating all partial products simultaneously and summing them using a structured array of adders (typically full adders and half adders). The design is easy to implement and scales well, but it can suffer from increased propagation delay due to carry chains in the adder structure. [1]

Advantages	Disadvantages
Regular and modular structure, making it suitable for hardware implementation. Easier to layout on FPGA and ASICs.	Slower for large operand widths due to linear delay from carry propagation. Not as power-efficient as some optimised structures.
Good for moderate-speed applications.	

Figure 1 - Braun’s Advantages/Disadvantages Table

Wallace Tree Multiplier

The Wallace Tree multiplier is a fast multiplication technique that reduces the number of adding stages. It uses a tree-like structure of ‘carry’ adders to sum the partial products more efficiently. The method reduces the number of sequential addition levels, leading to reduced delay. [2]

Advantages	Disadvantages
Faster than array multipliers due to minimised adder depth. More efficient for high-performance systems.	Irregular structure can make it harder to implement and debug. More complex routing may lead to higher power consumption and area.

Figure 2 - Wallace Tree Advantages/Disadvantages Table

Applications of Multipliers

Multipliers are often found in equipment such as:

- Control systems (door entry, motor control etc).
- Image and video processing.
- Cryptographic algorithms.

- DSP applications.

Modern trends including multiplier design focus on:

- Lower power designs for embedded and portable devices.
- Higher-speed multipliers for real-time systems.
- Artificial intelligent computing where exact results aren't necessarily required.
- Pipelined/parallelised designs using FPGA. [3]

3. Flowchart

The flowchart below simply outlines the step-by-step process of performing an 8-bit multiplication using logic which is applicable to both Braun's and Wallace Tree multipliers.

- Start: The entry point of the system where the process begins.
- Input A & B: The system takes two 8-bit binary numbers as input. These are the multiplicand (A) and the multiplier (B).
- Generate Partial Products: Each bit of A is ANDed with each bit of B to create a matrix of partial products. This results in 64 partial products (8 rows $\times$ 8 columns).
- Sum Partial Products: The partial products are added together to create the final result.
 - o In Braun's, this is done through a structured array of adders.
 - o In Wallace Tree, the partial products are reduced layer-by-layer using full and half adders in the structure of a tree. [4]
- Get 16-bit Result: The result of multiplying two 8-bit numbers can be up to 16 bits long, so the final output is a 16-bit binary number.
- Complete.

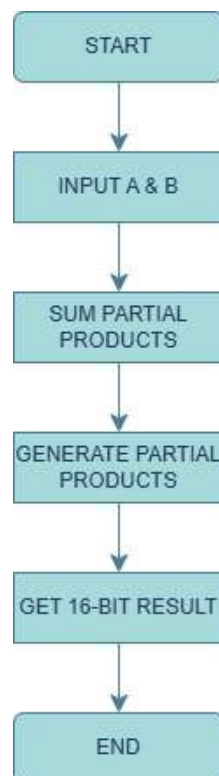


Figure 3 - Flowchart Explaining 8-bit Multiplication

Braun's Diagram

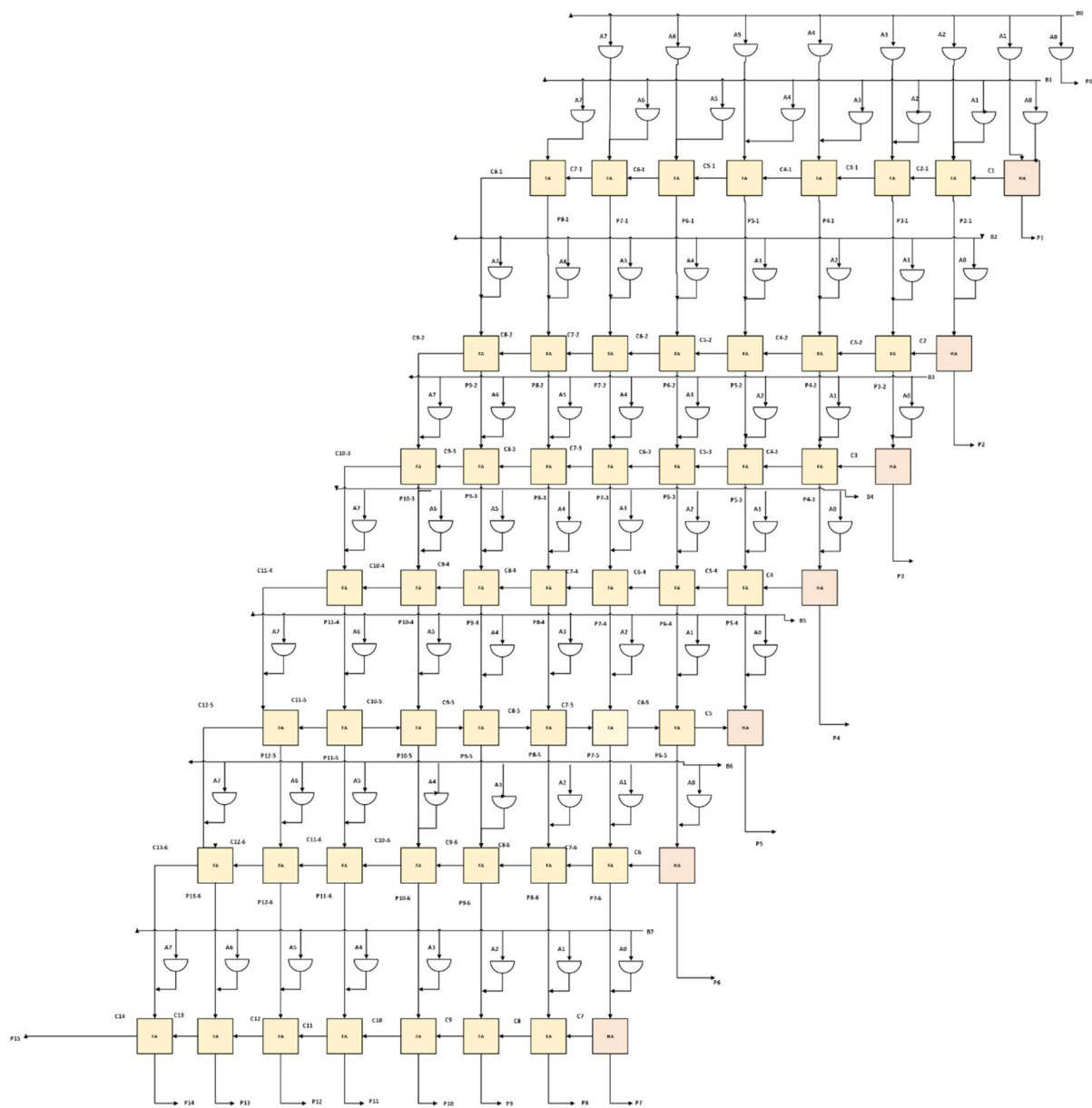


Figure 4 - Braun's Diagram

Wallace Tree Diagram

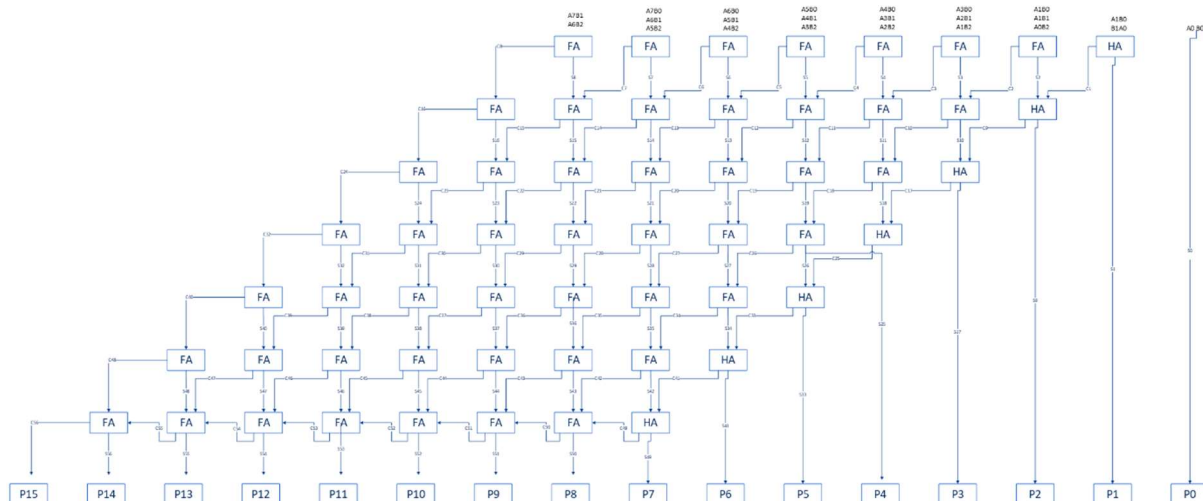


Figure 5 - Wallace Diagram

4. Methodology

Requirements Analysis

The goal is to design an 8-bit multiplier using two different architectures: Braun's and Wallace Tree Multipliers. Both designs will need too:

- Accept two x 8-bit inputs
- Output a 16-bit product.
- Written in VHDL.
- Synthesised using Vivado.

Design Strategy

Braun's Multiplier

- Partial products are generated using AND gates for each bit of the operands.
- A structure of half adders and full adders is used to add partial products.
- Components (AND gates, HAs, FAs) are implemented as separate VHDL entities and then integrated using port mapping.

Wallace Tree Multiplier

- Partial products are also generated using AND operations.
- The products are reduced using layers of carry-sum adders (using half and full adders).
- The first two rows are added using standard binary adder.

VHDL Implementation

- All basic components (AND gates, Half Adders, Full Adders) are coded in behavioural or structural VHDL.
- Designs are tested using VHDL testbenches to verify correctness.
- Port mapping is used to integrate components into the complete multiplier structure.
- Simulations are run in Vivado's Simulator to validate logical behaviour.

Synthesis and FPGA Implementation

- The VHDL designs are synthesized using Xilinx Vivado targeting a suitable FPGA (e.g., Basys 3 with Artix-7).
- Constraints are added for pin mapping and clock frequency.
- FPGA implementation results will be analysed to assess:
 - o Resource utilisation.
 - o Power consumption.
 - o Maximum operating frequency.

Performance Evaluation

- Delays, LUT usage and power are compared between Braun's and Wallace Tree designs.
- Four test samples are used to compare functional outputs and performance.
- Delay analysis is compared with synthesis reports to validate critical path findings.
- Strengths and limitations of each design are discussed with recommendations for improvement.

5. Algorithm Comparison

The operational comparison between Braun's and Wallace Tree multipliers reveals significant differences in design complexity and performance, as shown in Table 3 – Algorithm Comparison below [5],[6]:

Step	Braun's Multiplier	Wallace Tree Multiplier
1. Input Operands	A[7:0], B[7:0] (unsigned)	A[7:0], B[7:0] (unsigned)
2. Generate Partial Products	AND every bit of A with every bit of B (8x8 matrix)	AND every bit of A with every bit of B (8x8 matrix)
3. Grouping of Products	Structured in array rows (fixed layout)	Grouped by weight (bit position) for tree reduction
4. Addition Strategy	Adders in a triangular array structure	Carry-save adders reduce layers in parallel
5. Final Addition	Ripple through final row of adders	Fast adder (e.g., ripple carry) for final sum
6. Output Result	16-bit result P[15:0]	16-bit result P[15:0]
7. Design Complexity	Simple and regular, easy to implement	Complex but faster and efficient
8. Speed	Moderate (delayed by carry chains)	High (fewer addition stages)

Figure 6 – Table Comparing Algorithms

6. Simulation & Testing

Braun's Multiplier

The following table shows the simulation results for four different test cases, verifying the functionality of the Braun's multiplier implementation in VHDL. Each sample multiplies two unsigned 8-bit binary numbers and checks the 16-bit product output.

Sample	Input A (Hex)	Input B (Hex)	Decimal A × B	Output (Hex)	Output (Binary)	Status
--------	---------------	---------------	---------------	--------------	-----------------	--------

1	01	01	1	0001	0000 0000 0000 0001	✓ Pass
2	FF	FF	65025	FE01	1111 1110 0000 0001	✓ Pass
3	7F	7F	16129	3F01	0011 1111 0000 0001	✓ Pass
4	FF	80	32640	7F80	0111 1111 1000 0000	✓ Pass

Figure 7 - Output Table for Braun's Multiplier

Sample 1

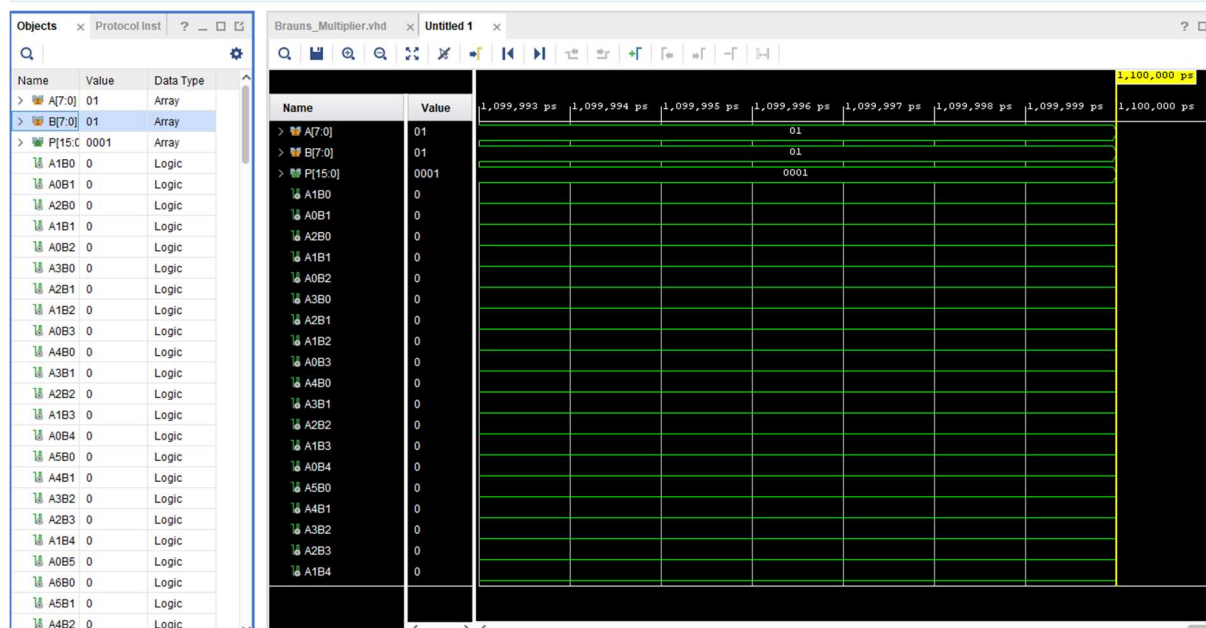


Figure 8 - Braun's Multiplier Sample 1

Sample 2

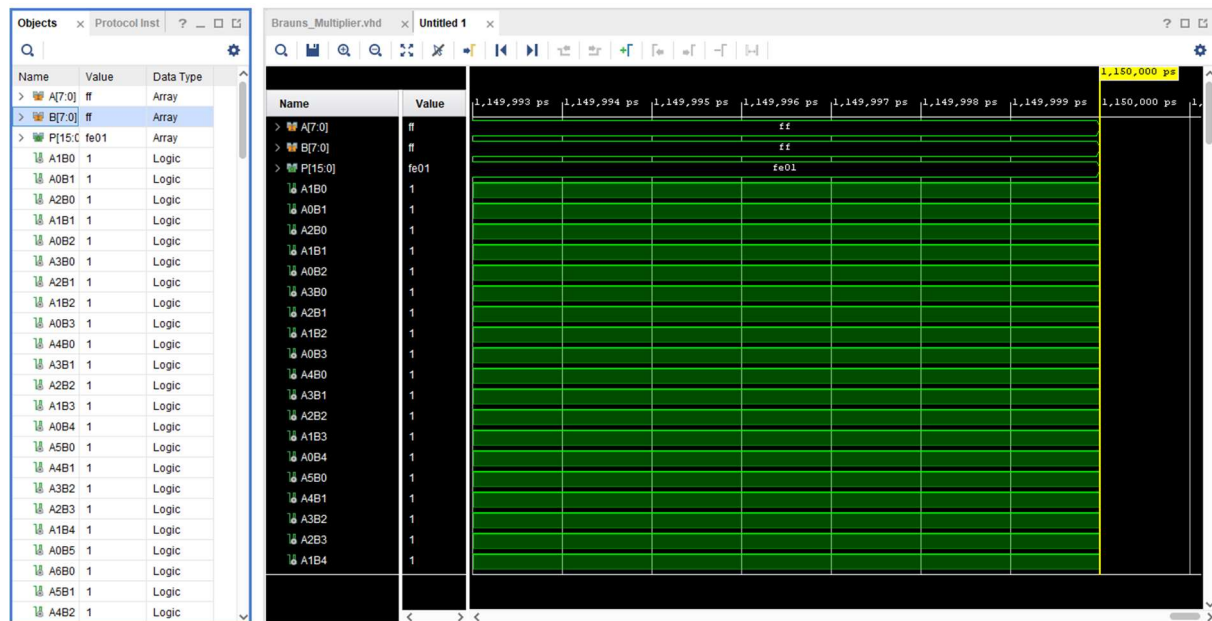


Figure 9 - Braun's Multiplier Sample 2

Sample 3

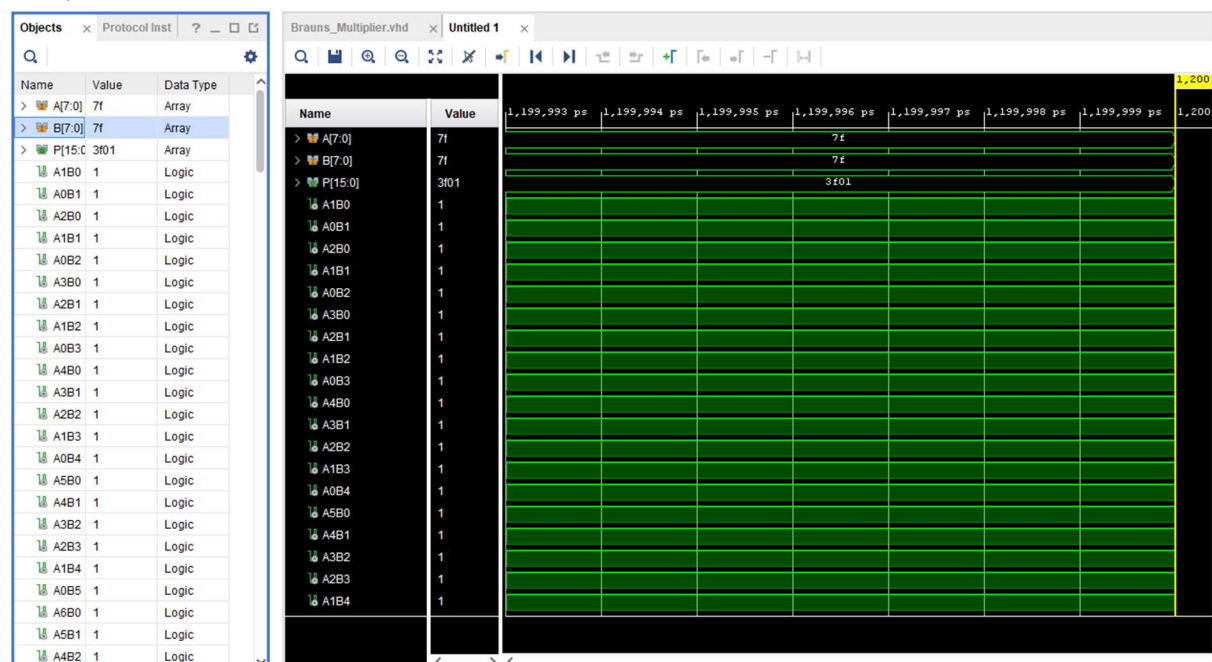


Figure 10 - Braun's Multiplier Sample 3

Sample 4

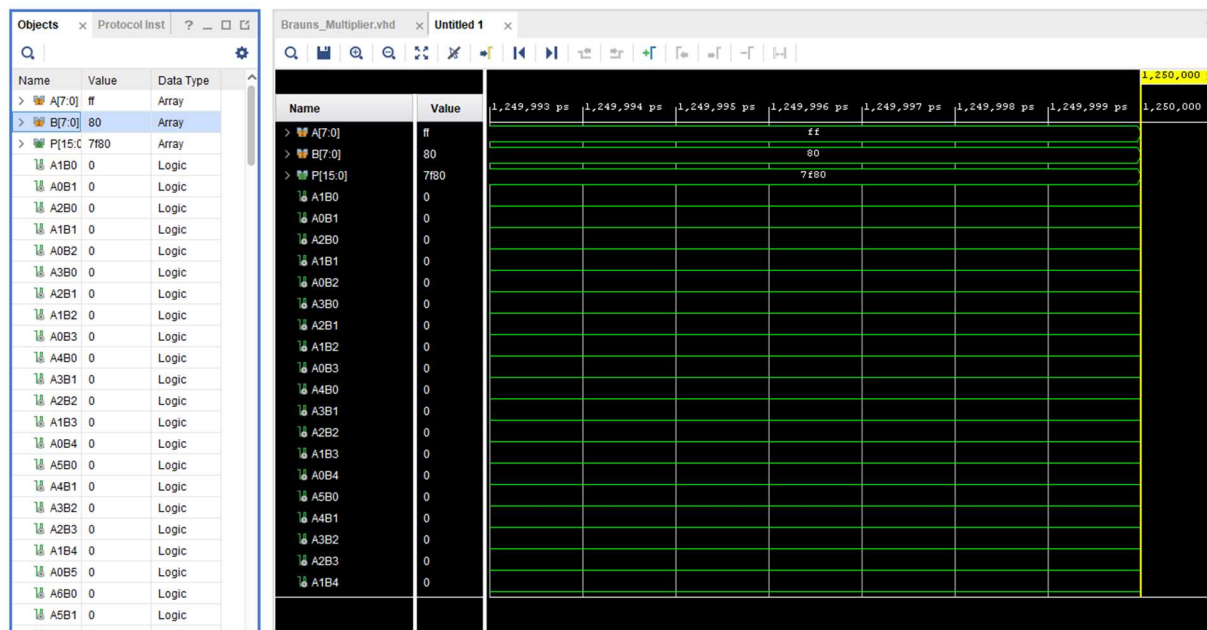


Figure 11 - Braun's Multiplier Sample 4

Interpretation of Results

- Sample 1 verified the multiplier's basic functionality with the minimal input case (1×1), producing the expected result of 1.
- Sample 2 tested the maximum 8-bit input values (255×255), correctly producing 65025 (FE01 in hex).
- Sample 3 evaluated mid-range symmetric inputs (127×127), yielding the accurate result of 16129 (3F01).
- Sample 4 combined a high-value operand (FF) with a power-of-two (80), resulting in 32640 (7F80), validating correct handling of higher-order bit multiplication.
- The multiplier demonstrated consistent and reliable operation across a range of input cases, from edge values to more complex combinations.
- No logical or arithmetic errors were observed in the simulation outputs, indicating that partial product generation and structured summation using full and half adders functioned as intended.

Wallace Tree Multiplier

Similarly to the Braun's multiplier, identical simulation tests were carried out to ensure an even comparison could be used between the two.

Sample	Input A (Hex)	Input B (Hex)	Decimal A × B	Output (Hex)	Output (Binary)	Status
1	01	01	1	0001	0000 0000 0000 0001	✓ Pass
2	FF	FF	65025	FE01	1111 1110 0000 0001	✓ Pass
3	7F	7F	16129	3F01	0011 1111 0000 0001	✓ Pass
4	FF	80	32640	7F80	0111 1111 1000 0000	✓ Pass

Figure 12 - Output Table for Wallace Tree

Sample 1

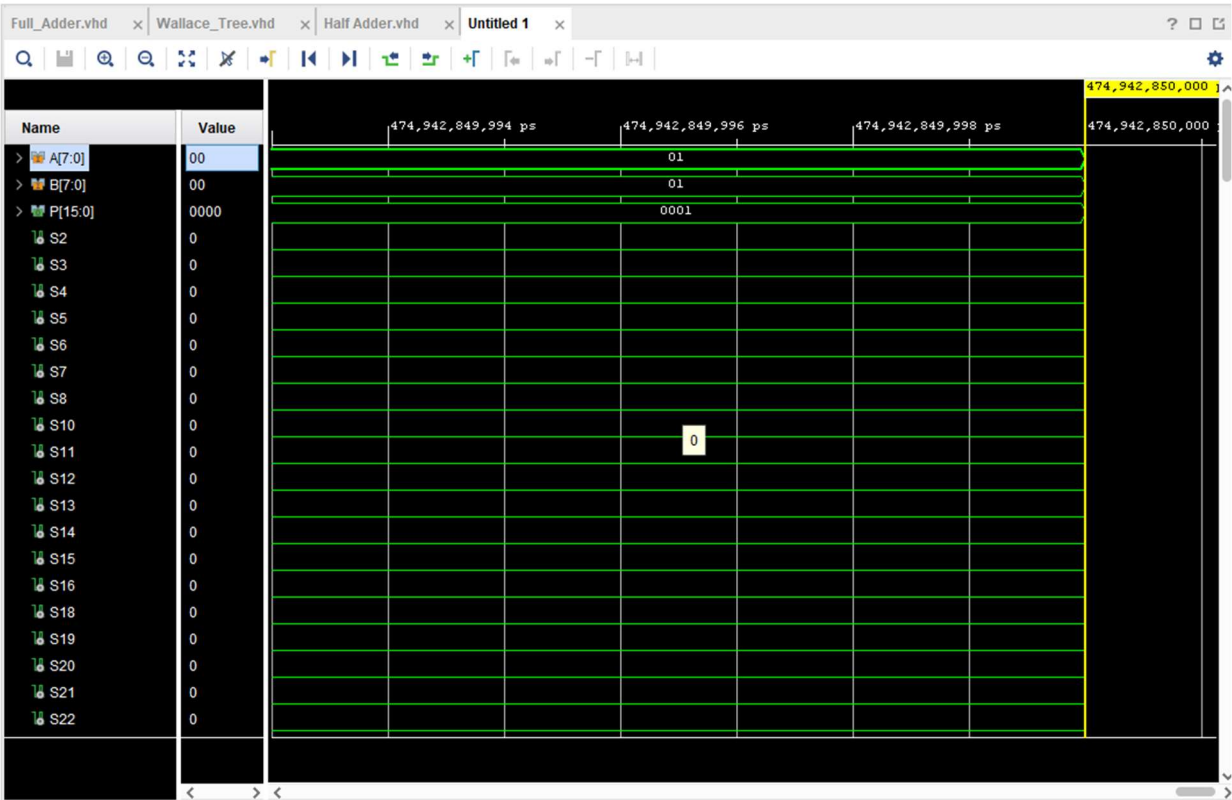


Figure 13 - Wallce Tree Sample 1

Sample 2

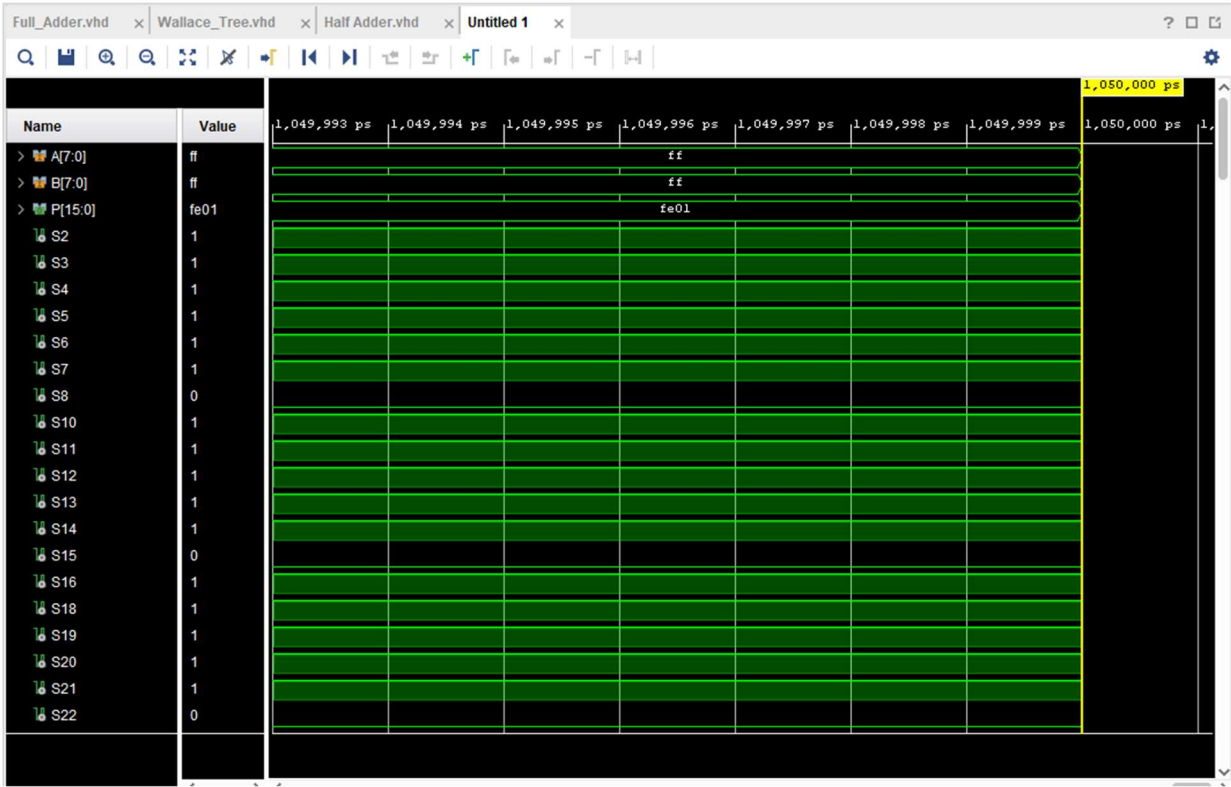


Figure 14 - Wallace Tree Sample 2

Sample 3

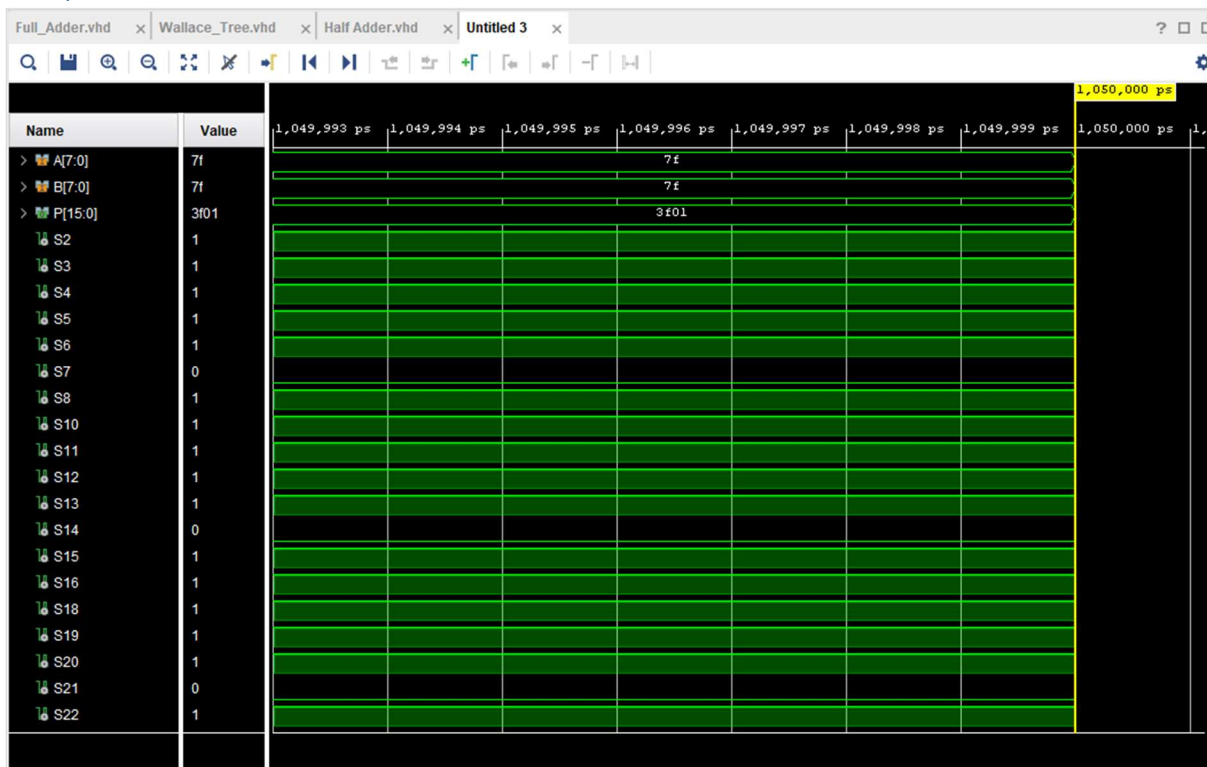


Figure 15 - Wallace Tree Sample 3

Sample 4

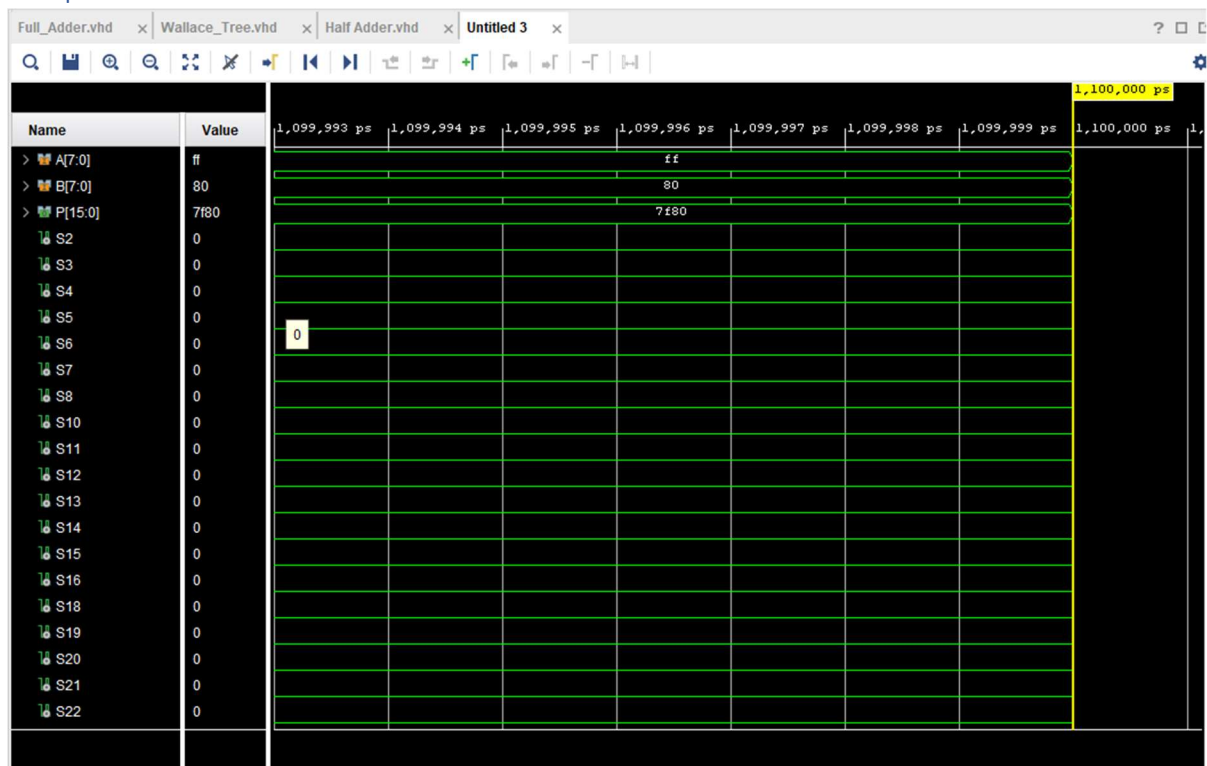


Figure 16 - Wallace Tree Sample 4

Interpretation of Results

- The Wallace Tree multiplier returned correct 16-bit products for all simulation inputs, confirming the design's functional correctness.
- Sample 1 served as a baseline, verifying successful multiplication of two minimal values.
- Sample 2 tested the architecture with maximum operand values, resulting in the correct product of 65025.
- Sample 3 used symmetric mid-range inputs (127×127), producing 16129, validating the Wallace Tree's multi-stage summation accuracy.
- Sample 4 demonstrated correct output for a combination of the maximum 8-bit value and a power-of-two ($255 \times 128 = 32640$).
- The tree structure's reduction of partial products and carry-save adders performed efficiently across all test cases.
- Simulation confirms that the implementation handles critical paths and input diversity without functional errors.

Performance Evaluation – Braun's Multiplier

Manual Delay Estimation

Assuming:

- AND gate delay: 1 ns
- Adder delay (HA or FA): 2 ns

Manual Delay = 1 ns (1st AND gate) + 7 x 2 ns (HA) + 13 x 2ns (FA) = 41 ns

Vivado Timing Summary (Post Synthesis)

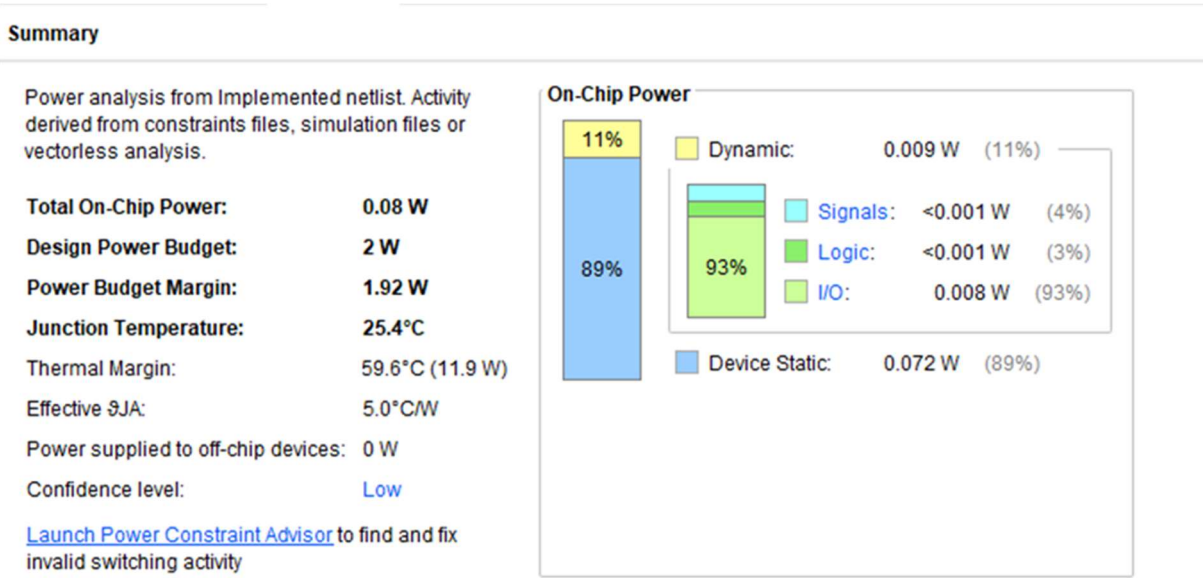
Vivado reports an actual delay budget of 13.5 ns for the worst-case path, which may involve optimised logic reduction during synthesis. This is lower than the manually estimated 31 ns due to parallel optimisations. All timing constraints met, confirming the design is timing-safe at the current clock setup.

Design Timing Summary					
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	13.508 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 17- Braun's Multiplier Timing Summary

Power Consumption (Post Synthesis)

Braun's multiplier is very power-efficient, especially in static conditions — ideal for moderate-speed, low-power embedded applications. IO Power dominates dynamic consumption (0.008 W from IO) whereas Logic and Signal Switching consume negligible power (<0.001 W).



Summary of Braun's Performance

Metric	Value
Manual Delay	~41 ns
Vivado Delay (WNS)	13.5 ns
Power	0.08 W (mostly static)
LUT Usage	87 (0.42%)
IO Usage	32 pins (30.2%)

Figure 21 - Braun's Performance Summary

Performance Evaluation – Wallace Tree

Manual Delay Estimation

Assuming:

- AND gate delay = 1 ns
- Adder delay (HA/FA) = 2 ns

Manual Delay = 1 ns (1st AND gate) + 7 x 2 ns (HA) + 7 x 2ns (FA) = 29 ns

Vivado Timing Summary (Post Synthesis)

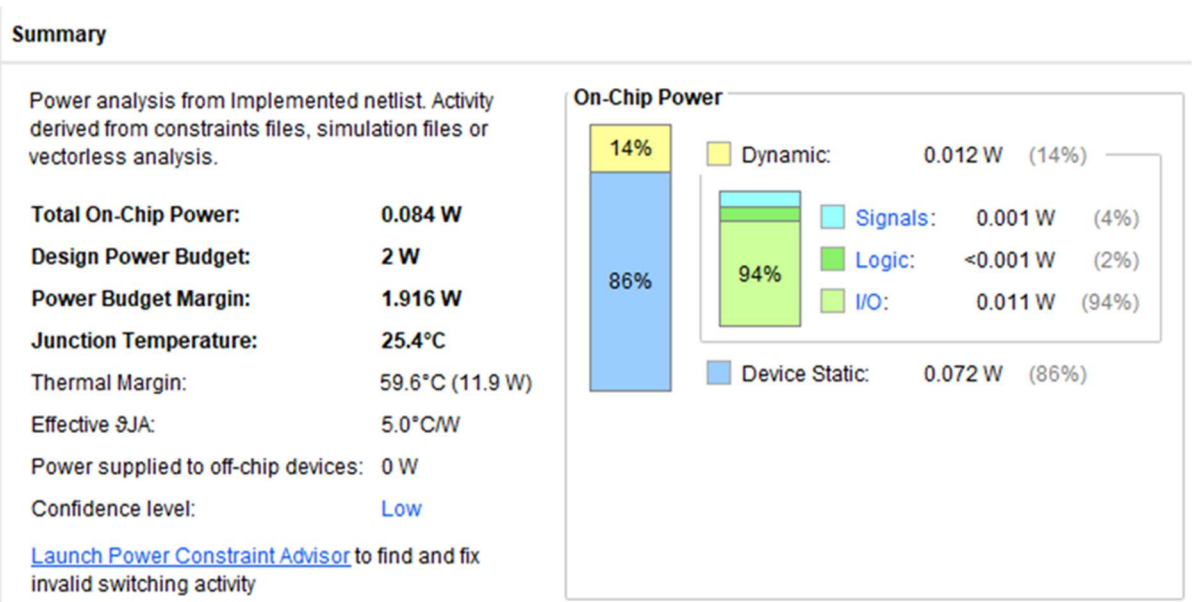
Vivado reports a shorter critical path than Braun's, confirming Wallace Tree's improved speed due to reduced adder depth.

Design Timing Summary					
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS): 6.078 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 22 - Wallace Tree Timing Summary

Power Consumption (Post Synthesis)

Slightly higher dynamic power compared to Braun (0.012 W vs 0.009 W) — expected due to Wallace Tree's increased switching activity in its deeper logic tree.



Summary of Wallace Tree Performance

Metric	Value
Manual Delay	~29 ns
Vivado Delay (WNS)	6.078 ns
Power	0.084 W
LUT Usage	86 (0.41%)
IO Usage	32 pins (30.2%)

Figure 26 - Wallace Tree Performance Summary

Multiplier Performance Comparison

Here's a side-by-side comparison table of Braun's vs. Wallace Tree Multiplier performance metrics.

	Metric	Braun's Multiplier	Wallace Tree Multiplier
1	Estimated Delay	41ns	29ns
2	Vivado Delay	13.508ns	6.078ns
3	Total Power	0.080W	0.084W
4	Dynamic Power	0.009W	0.012W
5	Static Power	0.072W	0.072W
6	LUT Usage (%)	87	86
7	LUT Utilisation	0.42%	0.41%
8	I/O Pin Usage	32	32
9	I/O Utilisation (%)	30.19%	30.19%

Figure 27 - Wallace/Braun's Performance Comparison

7. I/O Planning

The figure below shows the I/O Planning layout for the Braun's multiplier after synthesis in Vivado [7]. The diagram represents the physical pin assignments distributed along the periphery of the FPGA. This output confirms that the logical ports from the VHDL design have been successfully mapped to available I/O pins on the physical device. The outer grid area illustrates I/O blocks, while internal blocks show various configuration and logic elements.

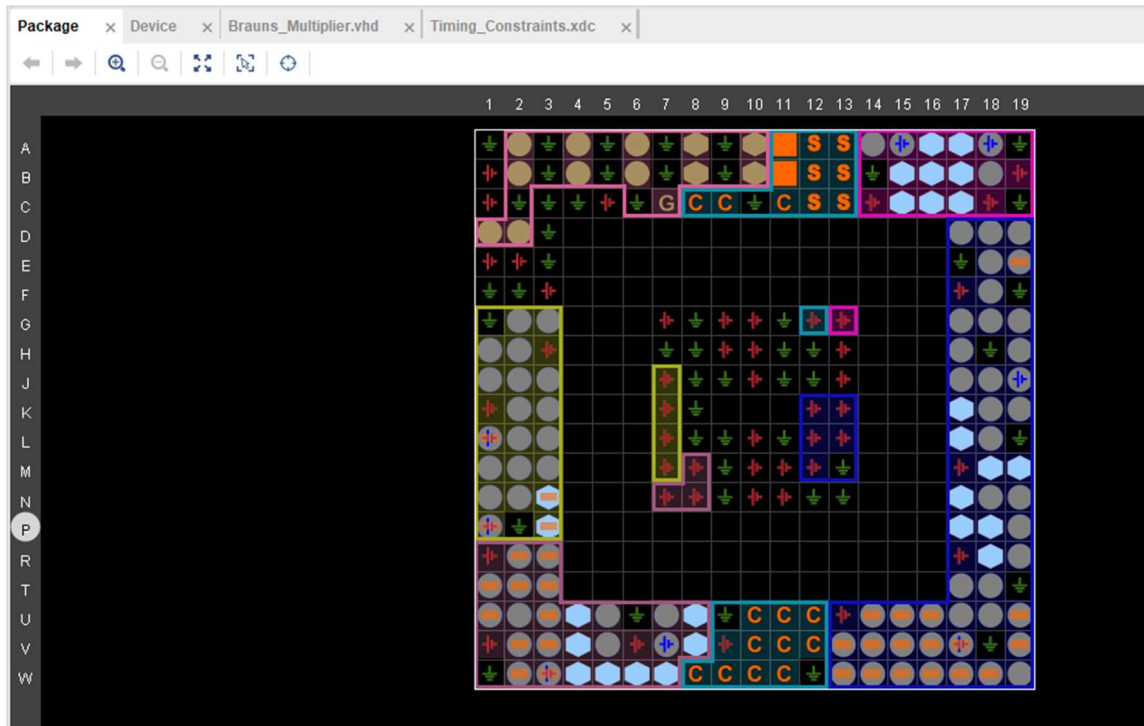


Figure 28 - Braun's IO Planning

The figure below presents the I/O Planning view for the Wallace Tree multiplier implementation post-synthesis. This visualization confirms that the 8-bit inputs and the 16-bit output (P[15:0]) have been successfully mapped to physical I/O pins on the FPGA device. Compared to the Braun implementation, the Wallace design shows a more compact and slightly asymmetrical layout of I/O cells yet remains evenly distributed along the outer edge of the FPGA.

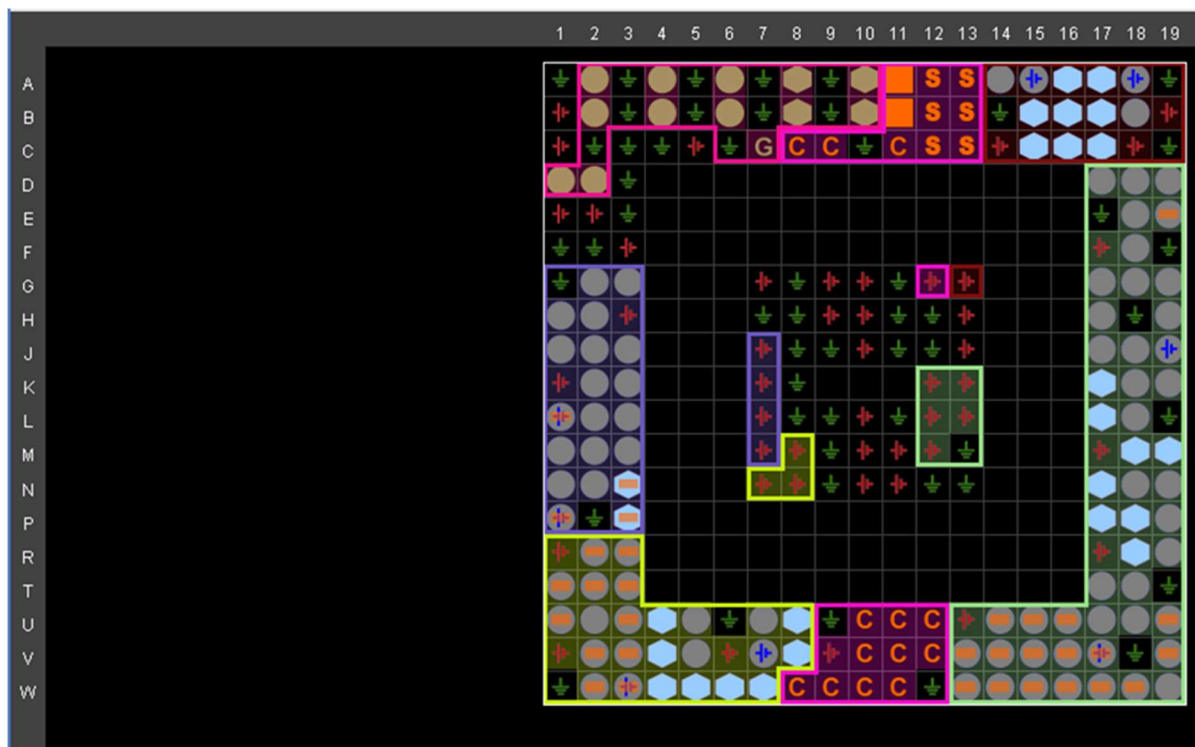


Figure 29 - Wallace Tree IO Planning

I/O Planning Summary

The I/O planning diagrams for Braun's and Wallace Tree multipliers reveal differences in how each design maps signals to the FPGA [8].

- Braun's multiplier shows a more evenly distributed and symmetrical layout, reflecting its regular array structure. This makes routing simpler and more predictable.
- Wallace Tree's layout is slightly more clustered and asymmetric, due to its complex carry-save adder structure. Despite this, it still fits efficiently within the I/O constraints.
- Both designs use 32 I/O pins, showing equal resource usage in terms of input/output connections.

Overall, Braun's design offers cleaner I/O organization, while Wallace Tree trades symmetry for performance without increasing I/O resource usage.

8. Device View

The device layout visualisations below highlight how each multiplier design is physically implemented across the FPGA's logic regions after synthesis in Vivado.

Braun's Multiplier

The Braun layout shows a wider horizontal distribution of logic resources across multiple clock and logic regions. This reflects the regular and structured grid-like logic of the array multiplier, where partial products are summed in aligned rows and columns. The layout appears more evenly dispersed, which helps reduce local congestion and improves routing simplicity.

Wallace Tree Multiplier

The Wallace Tree implementation exhibits a more vertically compact and centralized distribution of logic. This design condenses the carry-save adder structure into a narrower area, which is typical for

tree-based architectures that prioritize reduction depth over horizontal symmetry. Although more compact, it may result in denser routing within clustered regions.

Comparison Summary

While both designs occupy similar overall space, the Braun architecture spreads its logic more evenly, which can be advantageous for clarity and thermal distribution. The Wallace Tree concentrates logic for speed, trading off slightly higher local density in exchange for improved performance, as confirmed by its lower critical path delay.

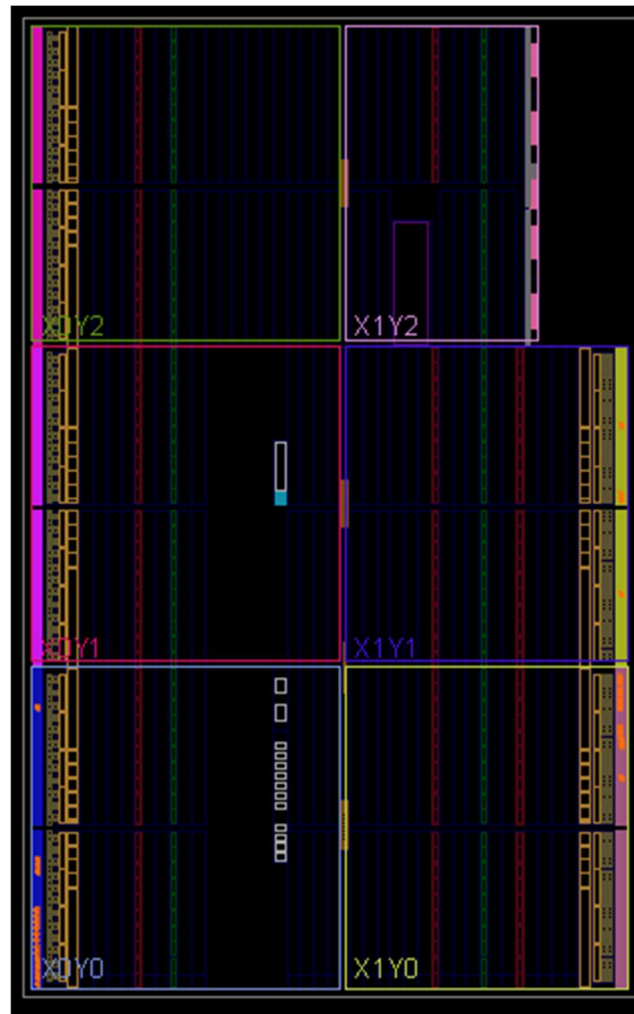


Figure 30 - Braun's Device View

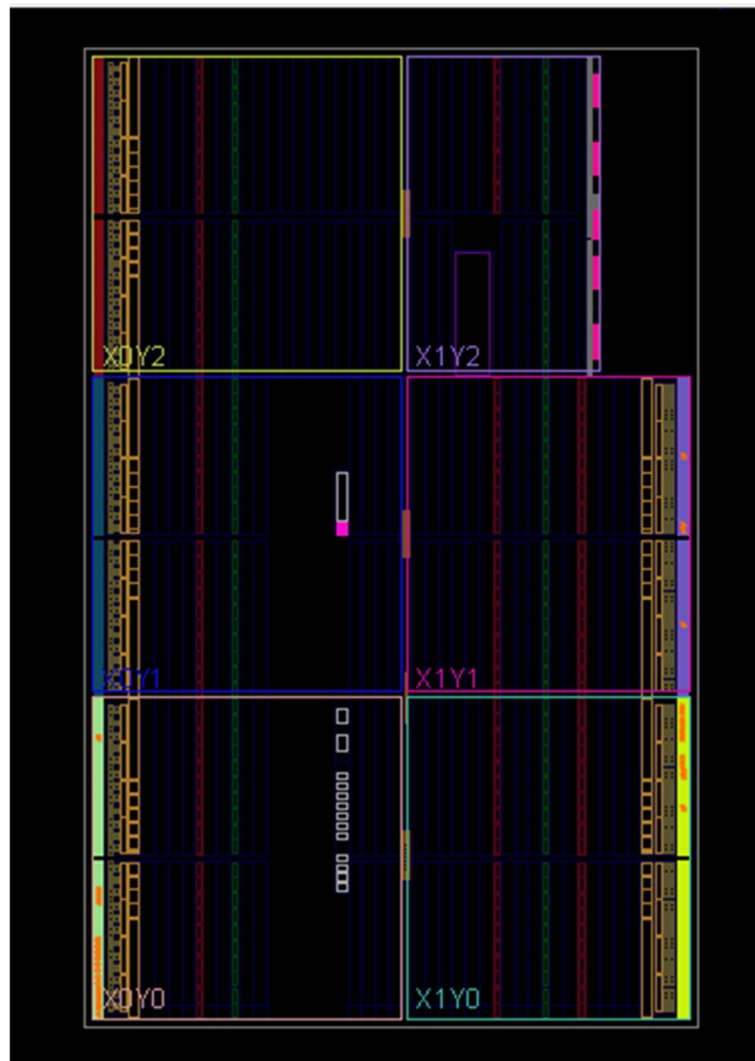


Figure 31 - Wallace Tree Device

9. Schematics

In digital design, schematic diagrams serve as visual representations of the underlying logic synthesized from HDL code. These schematics are automatically generated by design tools such as Xilinx Vivado and are essential for verifying the structure, connectivity, and function of the implemented logic. There are typically two types of schematics:

- RTL Schematic: Shows the high-level module interconnections as described in the VHDL source code.
- Technology Schematic: Displays the gate-level implementation of the design, including logic primitives such as lookup tables (LUTs), adders, and multiplexers.

The following schematics correspond to the technology-level views of both Braun's and Wallace Tree multipliers, offering insight into their internal logic structure and complexity after synthesis.

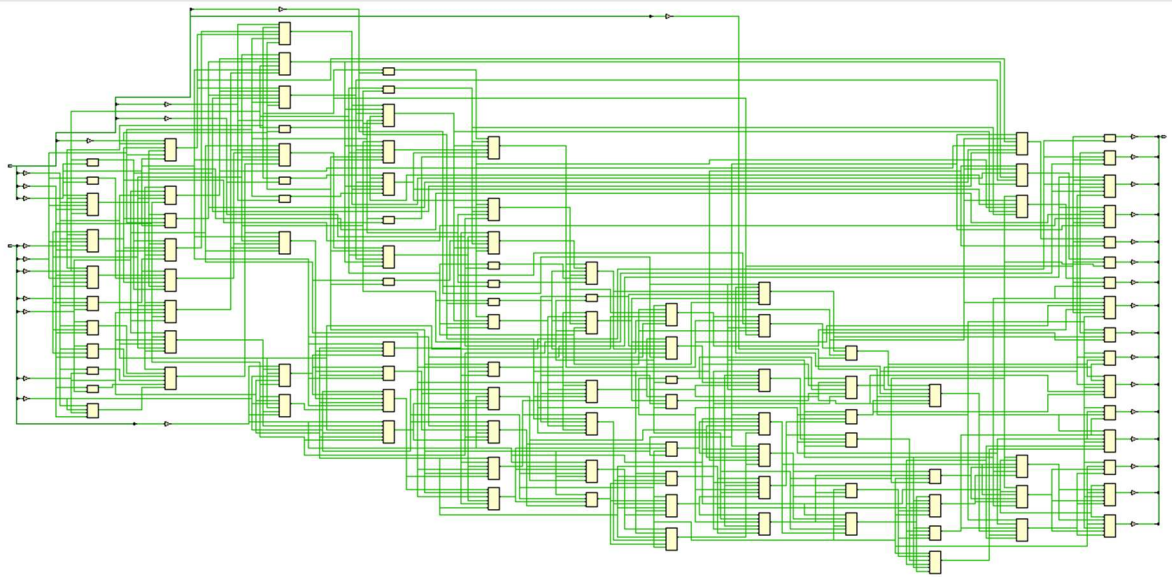


Figure 32 - Braun's Schematic

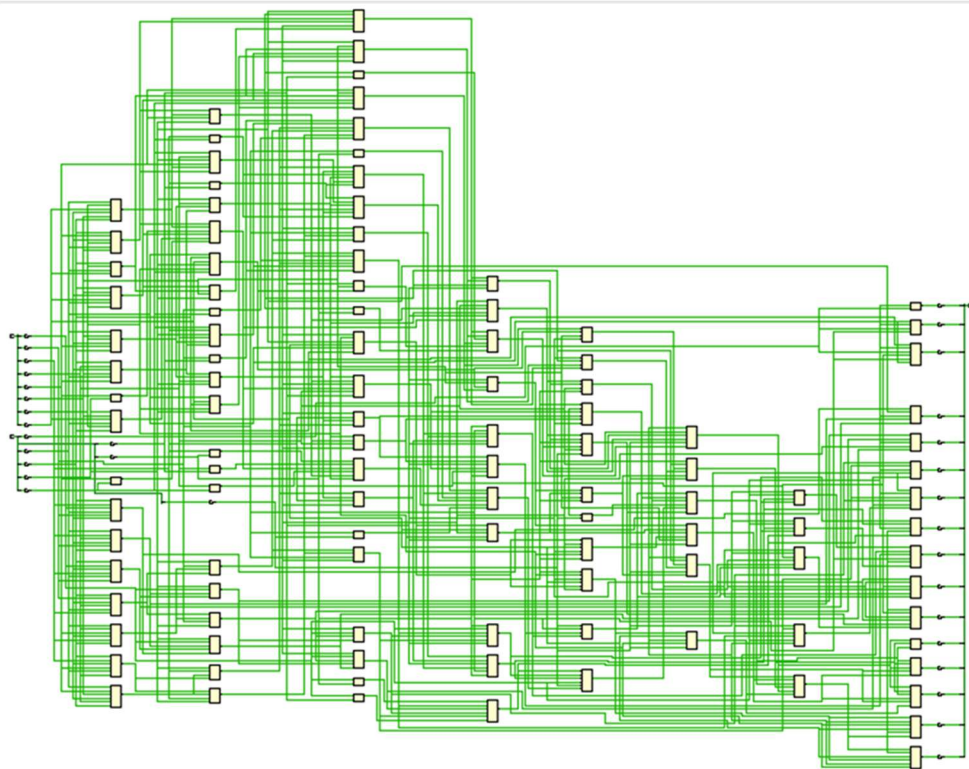


Figure 33 - Wallace Tree Schematic

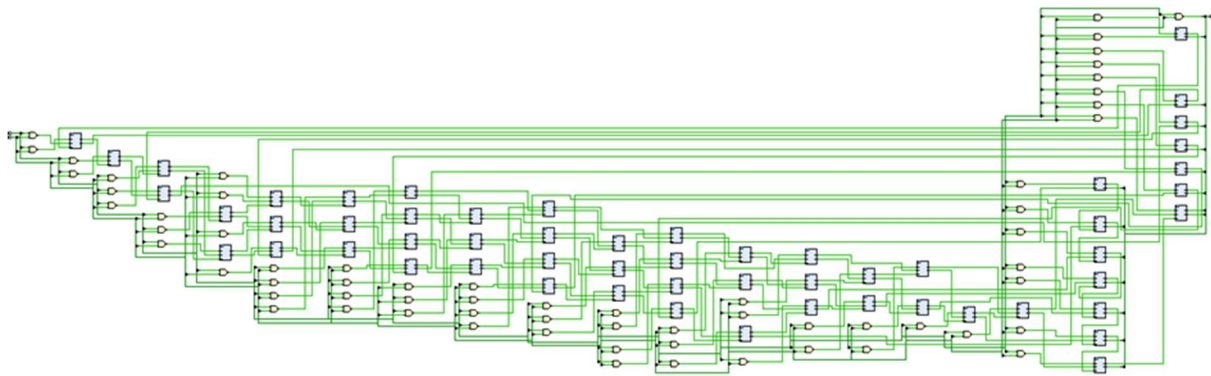


Figure 34 - Braun's RTL Schematic

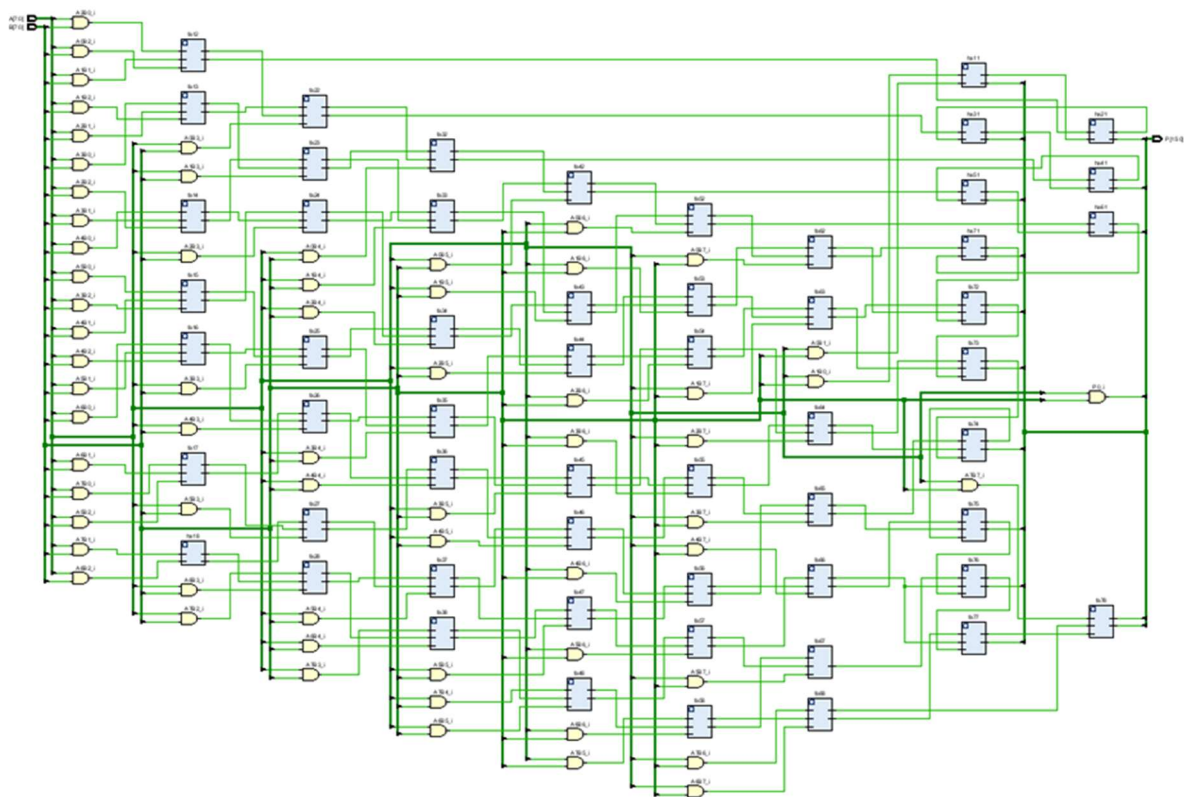


Figure 35 - Wallace Tree RTL Schematic

10. Pins Layout

Braun's Multiplier

▼  A (8)	IN		
 A[7]	IN		R2
 A[6]	IN		T1
 A[5]	IN		U1
 A[4]	IN		W2
 A[3]	IN		R3
 A[2]	IN		T2
 A[1]	IN		T3
 A[0]	IN		V2
▼  B (8)	IN		
 B[7]	IN		W13
 B[6]	IN		W14
 B[5]	IN		V15
 B[4]	IN		W15
 B[3]	IN		W17
 B[2]	IN		W16
 B[1]	IN		V16
 B[0]	IN		V17

▼  P (16)	OUT		
☒ P[15]	OUT		L1
☒ P[14]	OUT		P1
☒ P[13]	OUT		N3
☒ P[12]	OUT		P3
☒ P[11]	OUT		U3
☒ P[10]	OUT		W3
☒ P[9]	OUT		V3
☒ P[8]	OUT		V13
☒ P[7]	OUT		V14
☒ P[6]	OUT		U14
☒ P[5]	OUT		U15
☒ P[4]	OUT		W18
☒ P[3]	OUT		V19
☒ P[2]	OUT		U19
☒ P[1]	OUT		E19
☒ P[0]	OUT		U16

Figure 36 - Brauns Pinout

Wallace Tree

▼  A (8)	IN		
☒ A[7]	IN		R2
☒ A[6]	IN		T1
☒ A[5]	IN		U1
☒ A[4]	IN		W2
☒ A[3]	IN		R3
☒ A[2]	IN		T2
☒ A[1]	IN		T3
☒ A[0]	IN		V2
▼  B (8)	IN		
☒ B[7]	IN		W13
☒ B[6]	IN		W14
☒ B[5]	IN		V15
☒ B[4]	IN		W15
☒ B[3]	IN		W17
☒ B[2]	IN		W16
☒ B[1]	IN		V16
☒ B[0]	IN		V17

 P (16)	OUT		
 P[15]	OUT		L1
 P[14]	OUT		P1
 P[13]	OUT		N3
 P[12]	OUT		P3
 P[11]	OUT		U3
 P[10]	OUT		W3
 P[9]	OUT		V3
 P[8]	OUT		V13
 P[7]	OUT		V14
 P[6]	OUT		U14
 P[5]	OUT		U15
 P[4]	OUT		W18
 P[3]	OUT		V19
 P[2]	OUT		U19
 P[1]	OUT		E19
 P[0]	OUT		U16

Figure 37 - Wallace Tree Pinout

11. Recommendations

Based on the design, simulation, and synthesis results of both Braun's and Wallace Tree multipliers, the following recommendations are proposed:

- For speed-critical applications, the Wallace Tree Multiplier is recommended due to its significantly lower delay (6.078 ns vs. 13.508 ns). Its tree-structured summation stages make it more suitable for high-performance digital signal processing (DSP) or real-time computing systems.
- For area- or power-sensitive applications, the Braun Multiplier is more appropriate. Despite being slower, it uses slightly fewer LUTs (87 vs. 86) and has lower dynamic power consumption (0.009 W vs. 0.012 W). Its regular structure is easier to debug and implement on smaller FPGAs or ASICs.

To improve overall design efficiency:

- Pipelining could be applied to both architectures to reduce critical path delays and increase clock speed.
- Approximate computing techniques may be introduced in non-critical sections for further area/power savings, especially in AI or multimedia applications.
- Custom VHDL component reuse (e.g., parameterized adders) can make the design more modular and scalable.

12. Conclusion

This assignment successfully explored and compared two prominent unsigned multiplier architectures — Braun's Multiplier and the Wallace Tree Multiplier — using VHDL and FPGA implementation tools.

Both multipliers were:

- Designed structurally in VHDL
- Simulated and functionally verified with multiple test samples
- Synthesised using Xilinx Vivado and implemented on FPGA
- Analysed in terms of timing, power, area, and performance

The Braun multiplier proved to be highly regular, efficient in area, and easy to implement, while the Wallace Tree offered better performance through reduced critical delay. I/O planning and device layout diagrams demonstrated successful synthesis and placement on hardware.

Through quantitative and visual analysis, it is evident that design choice should be guided by application-specific requirements — Braun's for simplicity and Wallace Tree for speed. Both designs show potential for future improvement and integration into more complex systems [9].

13. References

[1] S. P. Dandamudi, *Fundamentals of Computer Organization and Design*. New York, NY, USA: Springer, 2003.

[2] Wallace, C. S. (1964). "A Suggestion for a Fast Multiplier". *IEEE Transactions on Electronic Computers*, EC-13(1), 14-17.

[3] Parhami, B. (2010). *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press.

[4] M. Morris Mano and M. D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog*, 6th ed., Pearson, 2017.

[5] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed., Oxford, U.K.: Oxford University Press, 2010.

[6] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, Feb. 1964.

[7] Xilinx Inc., *Vivado Design Suite User Guide: I/O and Clock Planning (UG899)*, v2023.1, May 2023. [Online]. Available: <https://www.xilinx.com>

[8] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3rd ed., New York, NY, USA: McGraw-Hill, 2009.

[9] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed., Oxford, U.K.: Oxford University Press, 2010.

14. Appendices

Braun's Code

-- Company: UWTSD

-- Engineer: David Mathew Lewis

-- Design Name: 8x8 Brauns Multiplier

-- Module Name: Braun - Behavioral

-- Project Name: Electronics II

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity Brauns_Multiplier 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 Brauns_Multiplier;

architecture Behavioral of Brauns_Multiplier is

signal A1B0 : std_logic;

signal A0B1 : std_logic;

signal A2B0 : std_logic;

signal A1B1 : std_logic;

signal A0B2 : std_logic;

signal A3B0 : std_logic;

signal A2B1 : std_logic;

signal A1B2 : std_logic;

signal A0B3 : std_logic;

signal A4B0 : std_logic;

signal A3B1 : std_logic;

signal A2B2 : std_logic;

```
signal A1B3 : std_logic;  
signal A0B4 : std_logic;  
signal A5B0 : std_logic;  
signal A4B1 : std_logic;  
signal A3B2 : std_logic;  
signal A2B3 : std_logic;  
signal A1B4 : std_logic;  
signal A0B5 : std_logic;  
signal A6B0 : std_logic;  
signal A5B1 : std_logic;  
signal A4B2 : std_logic;  
signal A3B3 : std_logic;  
signal A2B4 : std_logic;  
signal A1B5 : std_logic;  
signal A0B6 : std_logic;  
signal A7B0 : std_logic;  
signal A6B1 : std_logic;  
signal A5B2 : std_logic;  
signal A4B3 : std_logic;  
signal A3B4 : std_logic;  
signal A2B5 : std_logic;  
signal A1B6 : std_logic;  
signal A0B7 : std_logic;  
signal A7B1 : std_logic;  
signal A6B2 : std_logic;  
signal A5B3 : std_logic;  
signal A4B4 : std_logic;  
signal A3B5 : std_logic;  
signal A2B6 : std_logic;  
signal A1B7 : std_logic;  
signal A7B2 : std_logic;
```

```
signal A6B3 : std_logic;
signal A5B4 : std_logic;
signal A4B5 : std_logic;
signal A3B6 : std_logic;
signal A2B7 : std_logic;
signal A7B3 : std_logic;
signal A6B4 : std_logic;
signal A5B5 : std_logic;
signal A4B6 : std_logic;
signal A3B7 : std_logic;
signal A7B4 : std_logic;
signal A6B5 : std_logic;
signal A5B6 : std_logic;
signal A4B7 : std_logic;
signal A7B5 : std_logic;
signal A6B6 : std_logic;
signal A5B7 : std_logic;
signal A7B6 : std_logic;
signal A6B7 : std_logic;
signal A7B7 : std_logic;
signal S0 : std_logic;--P0
signal S1 : std_logic;--P1
signal C1 : std_logic;--P2
signal S2 : std_logic;
signal C2 : std_logic;--P3
signal S3 : std_logic;
signal C9 : std_logic;
signal S10 : std_logic;
signal C3 : std_logic;--P4
signal S4 : std_logic;
signal C10 : std_logic;
```

```
signal S11 : std_logic;
signal C17 : std_logic;
signal S18 : std_logic;
signal C4 : std_logic;--P5
signal S5 : std_logic;
signal C11 : std_logic;
signal S12 : std_logic;
signal C18 : std_logic;
signal S19 : std_logic;
signal C25 : std_logic;
signal S26 : std_logic;
signal C5 : std_logic;--P6
signal S6 : std_logic;
signal C12 : std_logic;
signal S13 : std_logic;
signal C19 : std_logic;
signal S20 : std_logic;
signal C26 : std_logic;
signal S27 : std_logic;
signal C33 : std_logic;
signal S34 : std_logic;
signal C6 : std_logic;--P7
signal S7 : std_logic;
signal C13 : std_logic;
signal S14 : std_logic;
signal C20 : std_logic;
signal S21 : std_logic;
signal C27 : std_logic;
signal S28 : std_logic;
signal C34 : std_logic;
signal S35 : std_logic;
```

```
signal C41 : std_logic;
signal S42 : std_logic;
signal C7 : std_logic;--P8
signal S8 : std_logic;
signal C14 : std_logic;
signal S15 : std_logic;
signal C21 : std_logic;
signal S22 : std_logic;
signal C28 : std_logic;
signal S29 : std_logic;
signal C35 : std_logic;
signal S36 : std_logic;
signal C42 : std_logic;
signal S43 : std_logic;
signal C49 : std_logic;
signal C8 : std_logic;--P9
signal C15 : std_logic;
signal S16 : std_logic;
signal C22 : std_logic;
signal S23 : std_logic;
signal C29 : std_logic;
signal S30 : std_logic;
signal C36 : std_logic;
signal S37 : std_logic;
signal C43 : std_logic;
signal S44 : std_logic;
signal C50 : std_logic;
signal C16 : std_logic;--P10
signal C23 : std_logic;
signal S24 : std_logic;
signal C30 : std_logic;
```

```
signal S31 : std_logic;
signal C37 : std_logic;
signal S38 : std_logic;
signal C44 : std_logic;
signal S45 : std_logic;
signal C51 : std_logic;
signal C24 : std_logic;--P11
signal C31 : std_logic;
signal S32 : std_logic;
signal C38 : std_logic;
signal S39 : std_logic;
signal C45 : std_logic;
signal S46 : std_logic;
signal C52 : std_logic;
signal C32 : std_logic;--P12
signal C39 : std_logic;
signal S40 : std_logic;
signal C46 : std_logic;
signal S47 : std_logic;
signal C53 : std_logic;
signal C40 : std_logic;--P13
signal C47 : std_logic;
signal S48 : std_logic;
signal C54 : std_logic;
signal C48 : std_logic;--P14
signal C55 : std_logic;
signal C56 : std_logic;--p15
```

component Full_Adder is

Port (A : in STD_LOGIC;

```
B : in STD_LOGIC;  
Cin : in STD_LOGIC;  
S : out STD_LOGIC;  
Cout : out STD_LOGIC);
```

```
end component;
```

```
component Half_Adder is
```

```
Port ( A : in STD_LOGIC;  
      B : in STD_LOGIC;  
      S: out STD_LOGIC;  
      Cout : out STD_LOGIC);
```

```
end component;
```

```
begin
```

```
A1B0 <= A(1) and B(0);  
A0B1 <= A(0) and B(1);  
A2B0 <= A(2) and B(0);  
A1B1 <= A(1) and B(1);  
A0B2 <= A(0) and B(2);  
A3B0 <= A(3) and B(0);  
A2B1 <= A(2) and B(1);  
A1B2 <= A(1) and B(2);  
A0B3 <= A(0) and B(3);  
A4B0 <= A(4) and B(0);  
A3B1 <= A(3) and B(1);
```

$A2B2 \leq A(2) \text{ and } B(2);$

$A1B3 \leq A(1) \text{ and } B(3);$

$A0B4 \leq A(0) \text{ and } B(4);$

$A5B0 \leq A(5) \text{ and } B(0);$

$A4B1 \leq A(4) \text{ and } B(1);$

$A3B2 \leq A(3) \text{ and } B(2);$

$A2B3 \leq A(2) \text{ and } B(3);$

$A2B1 \leq A(2) \text{ and } B(1);$

$A1B4 \leq A(1) \text{ and } B(4);$

$A0B5 \leq A(0) \text{ and } B(5);$

$A6B0 \leq A(6) \text{ and } B(0);$

$A5B1 \leq A(5) \text{ and } B(1);$

$A4B2 \leq A(4) \text{ and } B(2);$

$A3B3 \leq A(3) \text{ and } B(3);$

$A2B4 \leq A(2) \text{ and } B(4);$

$A1B5 \leq A(1) \text{ and } B(5);$

$A0B6 \leq A(0) \text{ and } B(6);$

$A7B0 \leq A(7) \text{ and } B(0);$

$A6B1 \leq A(6) \text{ and } B(1);$

$A5B2 \leq A(5) \text{ and } B(2);$

$A4B3 \leq A(4) \text{ and } B(3);$

$A3B4 \leq A(3) \text{ and } B(4);$

$A2B5 \leq A(2) \text{ and } B(5);$

$A1B6 \leq A(1) \text{ and } B(6);$

$A0B7 \leq A(0) \text{ and } B(7);$

$A7B1 \leq A(7) \text{ and } B(1);$

$A6B2 \leq A(6) \text{ and } B(2);$

$A5B3 \leq A(5) \text{ and } B(3);$

$A4B4 \leq A(4) \text{ and } B(4);$

$A3B5 \leq A(3) \text{ and } B(5);$

$A2B6 \leq A(2) \text{ and } B(6);$

A1B7 <= A(1) and B(7);

A7B2 <= A(7) and B(2);

A6B3 <= A(6) and B(3);

A5B4 <= A(5) and B(4);

A4B5 <= A(4) and B(5);

A3B6 <= A(3) and B(6);

A2B7 <= A(2) and B(7);

A7B3 <= A(7) and B(3);

A6B4 <= A(6) and B(4);

A5B5 <= A(5) and B(5);

A4B6 <= A(4) and B(6);

A3B7 <= A(3) and B(7);

A7B4 <= A(7) and B(4);

A6B5 <= A(6) and B(5);

A5B6 <= A(5) and B(6);

A4B7 <= A(4) and B(7);

A7B5 <= A(7) and B(5);

A6B6 <= A(6) and B(6);

A5B7 <= A(5) and B(7);

A7B6 <= A(7) and B(6);

A6B7 <= A(6) and B(7);

A7B7 <= A(7) and B(7);

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

P1 : Half_Adder port map (A=>A1B0 , B=>A0B1 , S=> P(1) , Cout=> C1);--P1

S2_1 : Full_adder port map (A=>A2B0 , B=>A1B1 , Cin=> C1, S=>S2 ,Cout=> C2);--P2

P2: Half_Adder port map (A=>A0B2 , B=>S2 , S=>P(2) , cout=>C9);

S3_1 : Full_adder port map (A=>A3B0 , B=>A2B1 , Cin=>C2 , S=>S3 , Cout=>C3);--P3

S10_1 : Full_adder port map (A=>A1B2 , B=>S3 , Cin=>C9 , S=>S10 , Cout=>C10);

P3 : Half_Adder port map (A=>A0B3 , B=>S10 , S=>P(3) , Cout=>C17);

S4_1 : Full_Adder port map (A=>A4B0 , B=>A3B1 , Cin=>C3 , S=>S4 , Cout=>C4);--P4

S11_1 : Full_Adder port map (A=>A2B2 , B=>S4 , Cin=>C10 , S=>S11 , Cout=>C11);

S18_1 : Full_Adder port map (A=>A1B3 , B=>S11 , Cin=>C17 , S=>S18 , Cout=>C18);

P4 : Half_Adder port map (A=>A0B4 , B=>S18 , S=>P(4) , Cout=>C25);

S5_1 : Full_Adder port map (A=>A5B0 , B=>A4B1 , Cin=>C4 , S=>S5 , Cout=>C5);--P5

S12_1 : Full_Adder port map (A=>A3B2 , B=>S5 , Cin=>C11 , S=>S12 , Cout=>C12);

S19_1 : Full_Adder port map (A=>A2B3 , B=>S12 , Cin=>C18 , S=>S19 , Cout=>C19);

S26_1 : Full_Adder port map (A=>A1B4 , B=>S19 , Cin=>C25 , S=>S26 , Cout=>C26);

P5 : Half_adder port map (A=>A0B5 , B=>S26 , S=>P(5) , Cout=>C33);

S6_1 : Full_adder port map (A=>A6B0 , B=>A5B1 , Cin=>C5 , S=>S6 , Cout=>C6);--P6

S13_1 : Full_adder port map (A=>A4B2 , B=>S6 , Cin=>C12 , S=>S13 , Cout=>C13);

S20_1 : Full_adder port map (A=>A3B3 , B=>S13 , Cin=>C19 , S=>S20 , Cout=>C20);

S27_1 : Full_adder port map (A=>A2B4 , B=>S20 , Cin=>C26 , S=>S27 , Cout=>C27);

S34_1 : Full_adder port map (A=>A1B5 , B=>S27 , Cin=>C33 , S=>S34 , Cout=>C34);

P6 : Half_adder port map (A=>A0B6 , B=>S34 , S=>P(6) , Cout=>C41);

S7_1 : Full_adder port map (A=>A7B0 , B=>A6B1 , Cin=>C6 , S=>S7 , Cout=>C7);--P7

S14_1 : Full_adder port map (A=>A5B2 , B=>S7 , Cin=>C13 , S=>S14 , Cout=>C14);

S21_1 : Full_adder port map (A=>A4B3 , B=>S14 , Cin=>C20 , S=>S21 , Cout=>C21);

S28_1 : Full_adder port map (A=>A3B4 , B=>S21 , Cin=>C27 , S=>S28 , Cout=>C28);

S35_1 : Full_adder port map (A=>A2B5 , B=>S28 , Cin=>C34 , S=>S35 , Cout=>C35);

S42_1 : Full_adder port map (A=>A1B6 , B=>S35 , Cin=>C41 , S=>S42 , Cout=>C42);

P7 : Half_adder port map (A=>A0B7 , B=>S42 , S=>P(7) , Cout=>C49);

S8_1 : Half_Adder port map (A=>A7B1 , B=>C7 , S=>S8 , Cout=>C8);--P8

S15_1 : Full_adder port map (A=>A6B2 , B=>S8 , Cin=>C14 , S=>S15 , Cout=>C15);

S22_1 : Full_adder port map (A=>A5B3 , B=>S15 , Cin=>C21 , S=>S22 , Cout=>C22);
S29_1 : Full_adder port map (A=>A4B4 , B=>S22 , Cin=>C28 , S=>S29 , Cout=>C29);
S36_1 : Full_adder port map (A=>A3B5 , B=>S29 , Cin=>C35 , S=>S36 , Cout=>C36);
S43_1 : Full_adder port map (A=>A2B6 , B=>S36 , Cin=>C42 , S=>S43 , Cout=>C43);
P8 : Full_adder port map (A=>A1B7 , B=>S43 , Cin=>C49 , S=>P(8) , Cout=>C50);
S16_1 : Full_adder port map (A=>A7B2 , B=>C8 , Cin=>C15 , S=>S16 , Cout=>C16);--P9
S23_1 : Full_adder port map (A=>A6B3 , B=>S16 , Cin=>C22 , S=>S23 , Cout=>C23);
S30_1 : Full_adder port map (A=>A5B4 , B=>S23 , Cin=>C29 , S=>S30 , Cout=>C30);
S37_1 : Full_adder port map (A=>A4B5 , B=>S30 , Cin=>C36 , S=>S37 , Cout=>C37);
S44_1 : Full_adder port map (A=>A3B6 , B=>S37 , Cin=>C43 , S=>S44 , Cout=>C44);
P9 : Full_adder port map (A=>A2B7 , B=>S44 , Cin=>C50 , S=>P(9) , Cout=>C51);

S24_1 : Full_adder port map (A=>A7B3 , B=>C16 , Cin=>C23 , S=>S24 , Cout=>C24);--P10
S31_1 : Full_adder port map (A=>A6B4 , B=>S24 , Cin=>C30 , S=>S31 , Cout=>C31);
S38_1 : Full_adder port map (A=>A5B5 , B=>S31 , Cin=>C37 , S=>S38 , Cout=>C38);
S45_1 : Full_adder port map (A=>A4B6 , B=>S38 , Cin=>C44 , S=>S45 , Cout=>C45);
P10 : Full_adder port map (A=>A3B7 , B=>S45 , Cin=>C51 , S=>P(10) , Cout=>C52);

S32_1 : Full_adder port map (A=>A7B4 , B=>C24 , Cin=>C31 , S=>S32 , Cout=>C32);--P11
S39_1 : Full_adder port map (A=>A6B5 , B=>S32 , Cin=>C38 , S=>S39 , Cout=>C39);
S46_1 : Full_adder port map (A=>A5B6 , B=>S39 , Cin=>C45 , S=>S46 , Cout=>C46);
P11 : Full_adder port map (A=>A4B7 , B=>S46 , Cin=>C52 , S=>P(11) , Cout=>C53);

S40_1 : Full_adder port map (A=>A7B5 , B=>C32 , Cin=>C39 , S=>S40 , Cout=>C40);--P12
S47_1 : Full_adder port map (A=>A6B6 , B=>S40 , Cin=>C46 , S=>S47 , Cout=>C47);
P12 : Full_adder port map (A=>A5B7 , B=>S47 , Cin=>C53 , S=>P(12) , Cout=>C54);

S48_1 : Full_adder port map (A=>A7B6 , B=>C40 , Cin=>C47 , S=>S48 , Cout=>C48);--P13
P13 : Full_adder port map (A=>A6B7 , B=>S48 , Cin=>C54 , S=>P(13) , Cout=>C55);

P14 : Full_adder port map (A=>A7B7 , B=>C48 , Cin=>C55 , S=>P(14) , Cout=>C56);--P14

P15 : P(15) <= C56;--P15

end Behavioral;

Wallace Tree Code

-- Company: UWTSD

-- Engineer: David Mathew Lewis

-- Design Name: 8x8 Wallace Tree Multiplier

-- Module Name: Wallace Tree - Behavioral

-- Project Name: Electronics II

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity Wallace_Tree is

Port (A : in STD_LOGIC_VECTOR (7 downto 0) ;

B : in STD_LOGIC_VECTOR (7 downto 0) ;

P : out STD_LOGIC_VECTOR (15 downto 0)) ;

end Wallace_Tree;

architecture Behavioral of Wallace_Tree is

component Full_Adder is

Port (A : in STD_LOGIC;

B : in STD_LOGIC;

Cin : in STD_LOGIC;

S : out STD_LOGIC;

Cout : out STD_LOGIC);

end component;

component Half_Adder is

```
Port ( A : in STD_LOGIC;  
      B : in STD_LOGIC;  
      Cout : out STD_LOGIC;  
      S : out STD_LOGIC);
```

end component;

signal

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

signal

C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,C16,C17,C18,C19,C20,C21,C22,C23,C24,C25,C
26,C27,C28,C29,C30,C31,C32,C33,C34,C35,C36,C37,C38,C39,C40,C41,C42,C43,C44,C45,C46,C47,C48,
C49,C50,C51,C52,C53,C54,C55 : STD_LOGIC;

signal A1B0 : STD_LOGIC;

signal A2B0 : STD_LOGIC;

signal A3B0 : STD_LOGIC;

signal A4B0 : STD_LOGIC;

signal A5B0 : STD_LOGIC;

signal A6B0 : STD_LOGIC;

signal A7B0 : STD_LOGIC;

signal A0B1 : STD_LOGIC;

signal A1B1 : STD_LOGIC;

signal A2B1 : STD_LOGIC;

signal A3B1 : STD_LOGIC;

signal A4B1 : STD_LOGIC;

signal A5B1 : STD_LOGIC;

signal A6B1 : STD_LOGIC;

signal A7B1 : STD_LOGIC;

signal A0B2 : STD_LOGIC;

signal A1B2 : STD_LOGIC;

signal A2B2 : STD_LOGIC;

signal A3B2 : STD_LOGIC;

signal A4B2 : STD_LOGIC;

signal A5B2 : STD_LOGIC;

signal A6B2 : STD_LOGIC;

signal A7B2 : STD_LOGIC;

signal A0B3 : STD_LOGIC;

signal A1B3 : STD_LOGIC;

signal A2B3 : STD_LOGIC;

signal A3B3 : STD_LOGIC;

signal A4B3 : STD_LOGIC;

signal A5B3 : STD_LOGIC;

signal A6B3 : STD_LOGIC;

signal A7B3 : STD_LOGIC;

signal A0B4 : STD_LOGIC;

signal A1B4 : STD_LOGIC;

signal A2B4 : STD_LOGIC;

signal A3B4 : STD_LOGIC;

signal A4B4 : STD_LOGIC;

signal A5B4 : STD_LOGIC;

signal A6B4 : STD_LOGIC;

signal A7B4 : STD_LOGIC;

signal A0B5 : STD_LOGIC;

signal A1B5 : STD_LOGIC;

signal A2B5 : STD_LOGIC;

signal A3B5 : STD_LOGIC;

signal A4B5 : STD_LOGIC;

signal A5B5 : STD_LOGIC;

signal A6B5 : STD_LOGIC;

signal A7B5 : STD_LOGIC;

signal A0B6 : STD_LOGIC;

signal A1B6 : STD_LOGIC;

signal A2B6 : STD_LOGIC;

signal A3B6 : STD_LOGIC;

signal A4B6 : STD_LOGIC;

signal A5B6 : STD_LOGIC;

```
signal A6B6 : STD_LOGIC;
```

```
signal A7B6 : STD_LOGIC;
```

```
signal A0B7 : STD_LOGIC;
```

```
signal A1B7 : STD_LOGIC;
```

```
signal A2B7 : STD_LOGIC;
```

```
signal A3B7 : STD_LOGIC;
```

```
signal A4B7 : STD_LOGIC;
```

```
signal A5B7 : STD_LOGIC;
```

```
signal A6B7 : STD_LOGIC;
```

```
signal A7B7 : STD_LOGIC;
```

```
begin
```

```
A1B0 <= A(1) and B(0);
```

```
A2B0 <= A(2) and B(0);
```

```
A3B0 <= A(3) and B(0);
```

```
A4B0 <= A(4) and B(0);
```

```
A5B0 <= A(5) and B(0);
```

```
A6B0 <= A(6) and B(0);
```

```
A7B0 <= A(7) and B(0);
```

$A0B1 \leq A(0) \text{ and } B(1);$

$A1B1 \leq A(1) \text{ and } B(1);$

$A2B1 \leq A(2) \text{ and } B(1);$

$A3B1 \leq A(3) \text{ and } B(1);$

$A4B1 \leq A(4) \text{ and } B(1);$

$A5B1 \leq A(5) \text{ and } B(1);$

$A6B1 \leq A(6) \text{ and } B(1);$

$A7B1 \leq A(7) \text{ and } B(1);$

$A0B2 \leq A(0) \text{ and } B(2);$

$A1B2 \leq A(1) \text{ and } B(2);$

$A2B2 \leq A(2) \text{ and } B(2);$

$A3B2 \leq A(3) \text{ and } B(2);$

$A4B2 \leq A(4) \text{ and } B(2);$

$A5B2 \leq A(5) \text{ and } B(2);$

$A6B2 \leq A(6) \text{ and } B(2);$

$A7B2 \leq A(7) \text{ and } B(2);$

$A0B3 \leq A(0) \text{ and } B(3);$

$A1B3 \leq A(1) \text{ and } B(3);$

$A2B3 \leq A(2) \text{ and } B(3);$

$A3B3 \leq A(3) \text{ and } B(3);$

$A4B3 \leq A(4) \text{ and } B(3);$

$A5B3 \leq A(5) \text{ and } B(3);$

$A6B3 \leq A(6) \text{ and } B(3);$

$A7B3 \leq A(7) \text{ and } B(3);$

$A0B4 \leq A(0) \text{ and } B(4);$

$A1B4 \leq A(1) \text{ and } B(4);$

$A2B4 \leq A(2) \text{ and } B(4);$

$A3B4 \leq A(3) \text{ and } B(4);$

$A4B4 \leq A(4) \text{ and } B(4);$

$A5B4 \leq A(5) \text{ and } B(4);$

$A6B4 \leq A(6) \text{ and } B(4);$

$A7B4 \leq A(7) \text{ and } B(4);$

$A0B5 \leq A(0) \text{ and } B(5);$

$A1B5 \leq A(1) \text{ and } B(5);$

$A2B5 \leq A(2) \text{ and } B(5);$

$A3B5 \leq A(3) \text{ and } B(5);$

$A4B5 \leq A(4) \text{ and } B(5);$

$A5B5 \leq A(5) \text{ and } B(5);$

$A6B5 \leq A(6) \text{ and } B(5);$

$A7B5 \leq A(7) \text{ and } B(5);$

$A0B6 \leq A(0) \text{ and } B(6);$

$A1B6 \leq A(1) \text{ and } B(6);$

$A2B6 \leq A(2) \text{ and } B(6);$

$A3B6 \leq A(3) \text{ and } B(6);$

$A4B6 \leq A(4) \text{ and } B(6);$

$A5B6 \leq A(5) \text{ and } B(6);$

$A6B6 \leq A(6) \text{ and } B(6);$

$A7B6 \leq A(7) \text{ and } B(6);$

$A0B7 \leq A(0) \text{ and } B(7);$

$A1B7 \leq A(1) \text{ and } B(7);$

$A2B7 \leq A(2) \text{ and } B(7);$

$A3B7 \leq A(3) \text{ and } B(7);$

$A4B7 \leq A(4) \text{ and } B(7);$

$A5B7 \leq A(5) \text{ and } B(7);$

$A6B7 \leq A(6) \text{ and } B(7);$

$A7B7 \leq A(7) \text{ and } B(7);$

--first stage

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

ha11 : Half_Adder port map(A=>A1B0,B=>A0B1,S=>P(1),Cout=>C1);

fa12 : Full_Adder port map(A=>A2B0, B=>A1B1, Cin=>A0B2, S=>S2, Cout=>C2);

fa13 : Full_Adder port map(A=>A3B0, B=>A2B1, Cin=>A1B2, S=>S3, Cout=>C3);

fa14 : Full_Adder port map(A=>A4B0, B=>A3B1, Cin=>A2B2, S=>S4, Cout=>C4);

fa15 : Full_Adder port map(A=>A5B0, B=>A4B1, Cin=>A3B2, S=>S5, Cout=>C5);

fa16 : Full_Adder port map(A=>A6B0, B=>A5B1, Cin=>A4B2, S=>S6, Cout=>C6);

fa17 : Full_Adder port map(A=>A7B0, B=>A6B1, Cin=>A5B2, S=>S7, Cout=>C7);

ha18 : Half_Adder port map(A=>A7B1, B=>A6B2, S=>S8, Cout=>C8);

--second stage

ha21 : Half_Adder port map(A=>S2, B=>C1, S=>P(2), Cout=>C9);

fa22 : Full_Adder port map(A=>S3, B=>C2, Cin=>A0B3, S=>S10, Cout=>C10);

fa23 : Full_Adder port map(A=>S4, B=>C3, Cin=>A1B3, S=>S11, Cout=>C11);

fa24 : full_Adder port map(A=>S5, B=>C4, Cin=>A2B3, S=>S12, Cout=>C12);

fa25 : Full_Adder port map(A=>S6, B=>C5, Cin=>A3B3, S=>S13, Cout=>C13);

fa26 : Full_Adder port map(A=>S7, B=>C6, Cin=>A4B3, S=>S14, Cout=>C14);

fa27 : Full_Adder port map(A=>S8, B=>C7, Cin=>A5B3, S=>S15, Cout=>C15);

fa28 : Full_Adder port map(A=>A7B2, B=>C8, Cin=>A6B3, S=>S16, Cout=>C16);

--third stage

ha31 : Half_Adder port map(A=>S10, B=>C9, S=>P(3), Cout=>C17);

fa32 : Full_Adder port map(A=>S11, B=>C10, Cin=>A0B4, S=>S18, Cout=>C18);

fa33 : Full_Adder port map(A=>S12, B=>C11, Cin=>A1B4, S=>S19, Cout=>C19);

fa34 : Full_Adder port map(A=>S13, B=>C12, Cin=>A2B4, S=>S20, Cout=>C20);

```
fa35 : Full_Adder port map(A=>S14, B=>C13, Cin=>A3B4, S=>S21, Cout=>C21);
fa36 : Full_Adder port map(A=>S15, B=>C14, Cin=>A4B4, S=>S22, Cout=>C22);
fa37 : Full_Adder port map(A=>S16, B=>C15, Cin=>A5B4, S=>S23, Cout=>C23);
fa38 : Full_Adder port map(A=>A7B3, B=>C16, Cin=>A6B4, S=>S24, Cout=>C24);
```

--fourth stage

```
ha41 : Half_Adder port map(A=>S18, B=>C17, S=>P(4), Cout=>C25);
fa42 : Full_Adder port map(A=>S19, B=>C18, Cin=>A0B5, S=>S26, Cout=>C26);
fa43 : Full_Adder port map(A=>S20, B=>C19, Cin=>A1B5, S=>S27, Cout=>C27);
fa44 : Full_Adder port map(A=>S21, B=>C20, Cin=>A2B5, S=>S28, Cout=>C28);
fa45 : Full_Adder port map(A=>S22, B=>C21, Cin=>A3B5, S=>S29, Cout=>C29);
fa46 : Full_Adder port map(A=>S23, B=>C22, Cin=>A4B5, S=>S30, Cout=>C30);
fa47 : Full_Adder port map(A=>S24, B=>C23, Cin=>A5B5, S=>S31, Cout=>C31);
fa48 : Full_Adder port map(A=>A7B4, B=>C24, Cin=>A6B5, S=>S32, Cout=>C32);
```

--fifth stage

```
ha51 : Half_Adder port map(A=>S26, B=>C25, S=>P(5), Cout=>C33);
fa52 : Full_Adder port map(A=>S27, B=>C26, Cin=>A0B6, S=>S34, Cout=>C34);
fa53 : Full_Adder port map(A=>S28, B=>C27, Cin=>A1B6, S=>S35, Cout=>C35);
fa54 : Full_Adder port map(A=>S29, B=>C28, Cin=>A2B6, S=>S36, Cout=>C36);
fa55 : Full_Adder port map(A=>S30, B=>C29, Cin=>A3B6, S=>S37, Cout=>C37);
fa56 : Full_Adder port map(A=>S31, B=>C30, Cin=>A4B6, S=>S38, Cout=>C38);
fa57 : Full_Adder port map(A=>S32, B=>C31, Cin=>A5B6, S=>S39, Cout=>C39);
fa58 : Full_Adder port map(A=>A7B5, B=>C32, Cin=>A6B6, S=>S40, Cout=>C40);
```

-- sixth stage

```
ha61 : Half_Adder port map(A=>S34, B=>C33, S=>P(6), Cout=>C41);
fa62 : Full_Adder port map(A=>S35, B=>C34, Cin=>A0B7, S=>S42, Cout=>C42);
```

```
fa63 : Full_Adder port map(A=>S36, B=>C35, Cin=>A1B7, S=>S43, Cout=>C43);
fa64 : Full_Adder port map(A=>S37, B=>C36, Cin=>A2B7, S=>S44, Cout=>C44);
fa65 : Full_Adder port map(A=>S38, B=>C37, Cin=>A3B7, S=>S45, Cout=>C45);
fa66 : Full_Adder port map(A=>S39, B=>C38, Cin=>A4B7, S=>S46, Cout=>C46);
fa67 : Full_Adder port map(A=>S40, B=>C39, Cin=>A5B7, S=>S47, Cout=>C47);
fa68 : Full_Adder port map(A=>A7B6, B=>C40, Cin=>A6B7, S=>S48, Cout=>C48);
```

-- seventh stage

```
ha71 : Half_Adder port map(A=>S42, B=>C41, S=>P(7), Cout=>C49);
fa72 : Full_Adder port map(A=>S43, B=>C42, Cin=>C49, S=>P(8), Cout=>C50);
fa73 : Full_Adder port map(A=>S44, B=>C43, Cin=>C50, S=>P(9), Cout=>C51);
fa74 : Full_Adder port map(A=>S45, B=>C44, Cin=>C51, S=>P(10), Cout=>C52);
fa75 : Full_Adder port map(A=>S46, B=>C45, Cin=>C52, S=>P(11), Cout=>C53);
fa76 : Full_Adder port map(A=>S47, B=>C46, Cin=>C53, S=>P(12), Cout=>C54);
fa77 : Full_Adder port map(A=>S48, B=>C46, Cin=>C54, S=>P(13), Cout=>C55);
fa78 : Full_Adder port map(A=>A7B7, B=>C48, Cin=>C55, S=>P(14), Cout=>P(15));
```

end Behavioral;