

ELECTRONICS 2 COURSEWORK

STEFAN-CRISTIAN DAITOIU

2317310

Introduction.....	3
Background Research.....	4
Design Methodology.....	5
Braun Multiplier Design.....	5
Wallace Tree Multiplier Design.....	6
Results.....	6
Timing Analysis.....	11
Critical Path Evaluation.....	12
Performance Comparison.....	13
Conclusion.....	14
Bibliography.....	15
Annex.....	16

Introduction

Digital multiplication is a fundamental arithmetic operation used within almost every modern digital electronic system. Applications such as microprocessors, digital signal processing (DSP), image processing, communication systems, and artificial intelligence hardware rely heavily on efficient multiplication circuits. Because multiplication is often one of the most computationally intensive operations, the architecture selected for its implementation has a direct impact on system performance, power consumption, and hardware resource utilisation. [1]

As digital systems continue to evolve, there is an increasing demand for arithmetic circuits that can operate at higher speeds while maintaining efficiency. Traditional multiplication techniques based on repeated addition are inherently slow because they require sequential processing. This leads to increased computation time, particularly for large bit-width operations. To address this limitation, parallel multiplier architectures have been developed. These architectures perform multiple operations simultaneously, significantly reducing computation time and improving overall system performance. [2]

This project focuses on the design, implementation, and evaluation of two widely used parallel multiplier architectures: the 8-bit Braun multiplier and the 8-bit Wallace Tree multiplier. Both designs operate using unsigned binary multiplication and follow the same initial process of generating partial products. Each bit of one operand is combined with each bit of the second operand using AND gates, producing a matrix of partial products. These are then summed using half adders and full adders to produce a 16-bit output. [3]

The Braun multiplier, also known as an array multiplier, is characterised by its regular and structured layout. This makes it relatively easy to design and implement using hardware description languages such as VHDL, and it is particularly well suited for FPGA-based systems. However, its main drawback is the propagation delay caused by carry signals passing through multiple sequential stages. [4]

In contrast, the Wallace Tree multiplier improves performance by reducing partial products in parallel rather than sequentially. This significantly reduces the number of addition stages required and improves speed. However, this benefit comes at the cost of increased design complexity and more difficult routing. [5]

Background Research

Binary multiplication follows the same basic principle as decimal multiplication, but it uses base-two arithmetic. Each bit of one operand is multiplied with each bit of the other operand to produce partial products.

For two 8-bit unsigned inputs, the operands are represented as:

A = A7A6A5A4A3A2A1A0

B = B7B6B5B4B3B2B1B0

The result is a 16-bit output.

Partial products are generated using the AND operation, and the efficiency of the multiplier depends largely on how these values are combined. Different architectures use different summation methods, which leads to trade-offs in speed, complexity, and power consumption. [6]

The Braun multiplier is one of the simplest parallel multiplier architectures. It consists of a regular array of AND gates, half adders, and full adders arranged in a structured grid. All partial products are generated at the same time and then added through multiple sequential stages. Its main advantage is its regular layout, which makes the design easier to understand, debug, and map onto FPGA hardware. However, because carry signals must propagate through several addition stages, the delay increases as the bit-width grows. [7]

The Wallace Tree multiplier improves performance by reducing partial products using a tree-like arrangement. Instead of relying on sequential addition, it uses parallel compression stages to reduce multiple bits at the same time. Full adders combine three bits into two outputs, while half adders combine two bits, and this process continues until only two rows remain. These final rows are then added to produce the output. This approach significantly reduces the critical path delay, making the Wallace Tree faster than array-based multipliers. However, the irregular structure increases design complexity and makes routing more difficult. [8]

Design Methodology

The project used a structural VHDL approach to model both multipliers, closely representing actual hardware by connecting smaller components such as AND gates, half adders, and full adders. The design process involved researching multiplier architectures, designing the circuit structures, implementing the VHDL models, generating partial products, performing the addition using the appropriate architecture, simulating the designs, synthesising them in Xilinx Vivado, and analysing the resulting hardware performance. Although both multipliers share the same initial stage of partial product generation, they differ in the way those partial products are combined to produce the final output.

Braun Multiplier Design

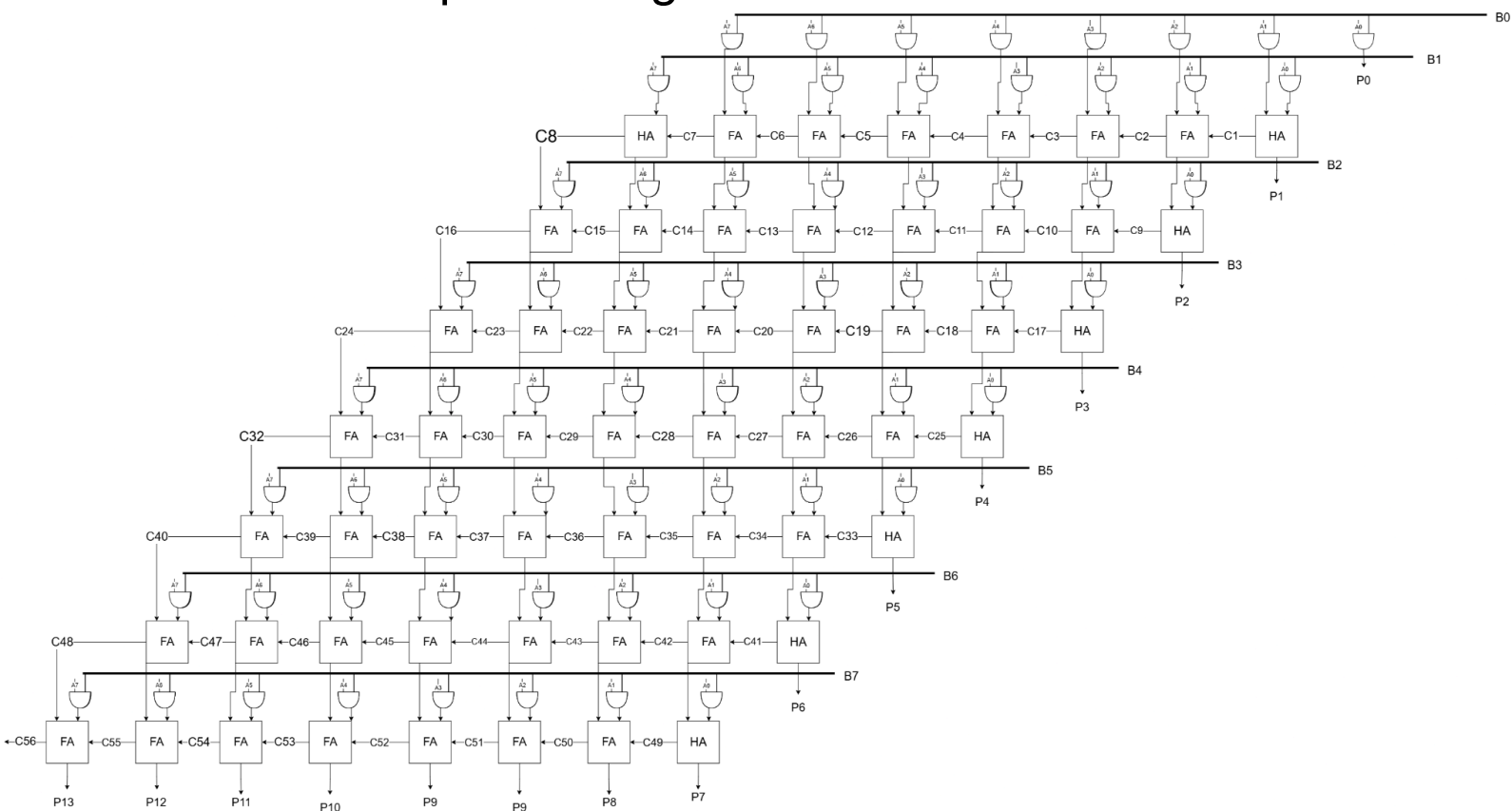


Figure 1: Braun design

The Braun multiplier generates partial products by ANDing each bit of input A with each bit of input B, producing 64 partial products arranged according to their binary significance. These values are then added together using a structured array in which half adders handle two input values, full adders handle three, and carry signals propagate diagonally through the circuit. The design was built using modular components connected through port mapping, which made it reusable and easier to debug. Simulation confirmed correct functionality.

Wallace Tree Multiplier Design

The Wallace Tree multiplier was implemented using the same structural VHDL approach as the Braun multiplier; however, the arrangement of the arithmetic components was different. Instead of adding partial products sequentially through an array structure, the Wallace design reduces the partial product matrix through multiple compression stages. Full adders and half adders were used to reduce the number of bits within each column until only two rows remained. These remaining rows were then combined to produce the final 16-bit multiplication output.

The main advantage of the Wallace Tree architecture is that the reduction process occurs in parallel, reducing the number of sequential addition stages required. This reduces the critical path delay compared with the Braun multiplier. However, the irregular arrangement of the reduction stages increases implementation complexity and makes debugging more difficult.

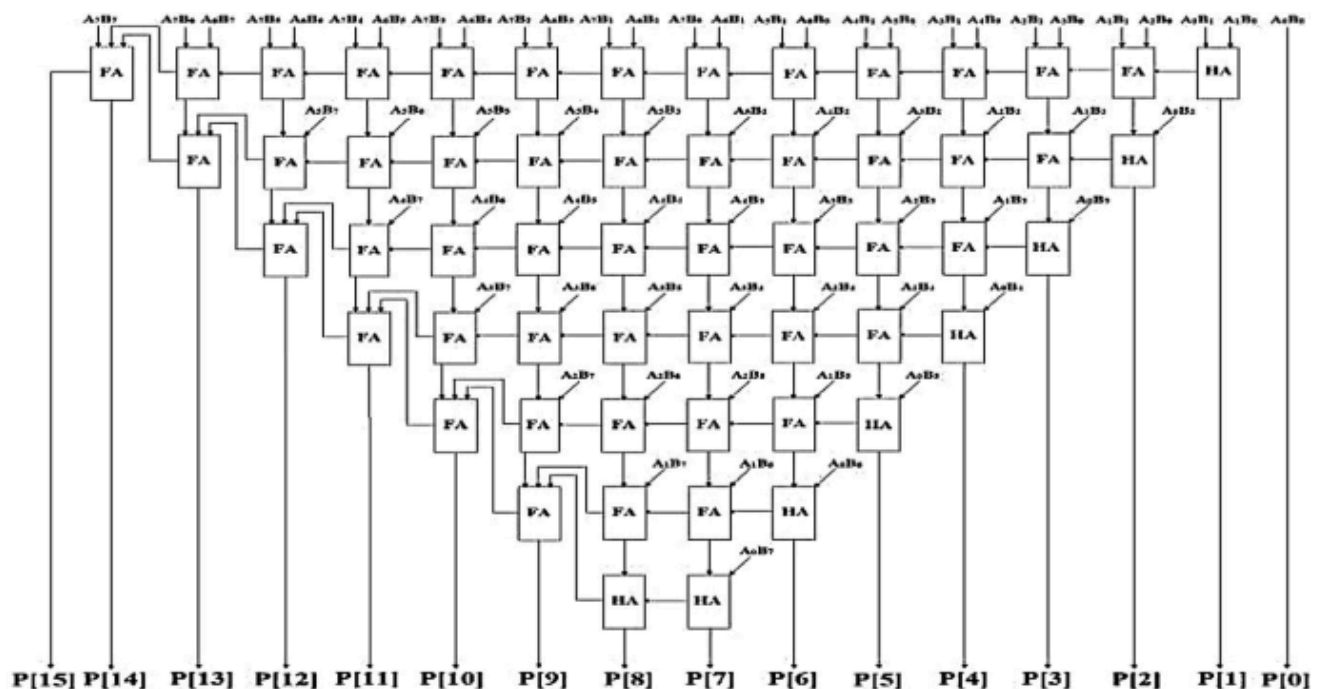


Figure 2: Wallace tree design [9]

Results

The Braun multiplier was successfully synthesised and functionally verified using Xilinx Vivado. Behavioural simulation was performed using multiple input combinations to confirm correct multiplication operation. The simulation results demonstrated that the multiplier generated the expected 16-bit output for the applied unsigned binary inputs.

For example, when the input values were $A = 03$ and $B = 05$, the multiplier produced an output of $000F$, which corresponds to the expected decimal multiplication result of 15. A

further test was carried out using A = 0F and B = 0F, producing an output of 00E1, confirming the expected result of 225. These results verified that the structural VHDL implementation of the Braun multiplier was operating correctly before FPGA implementation.

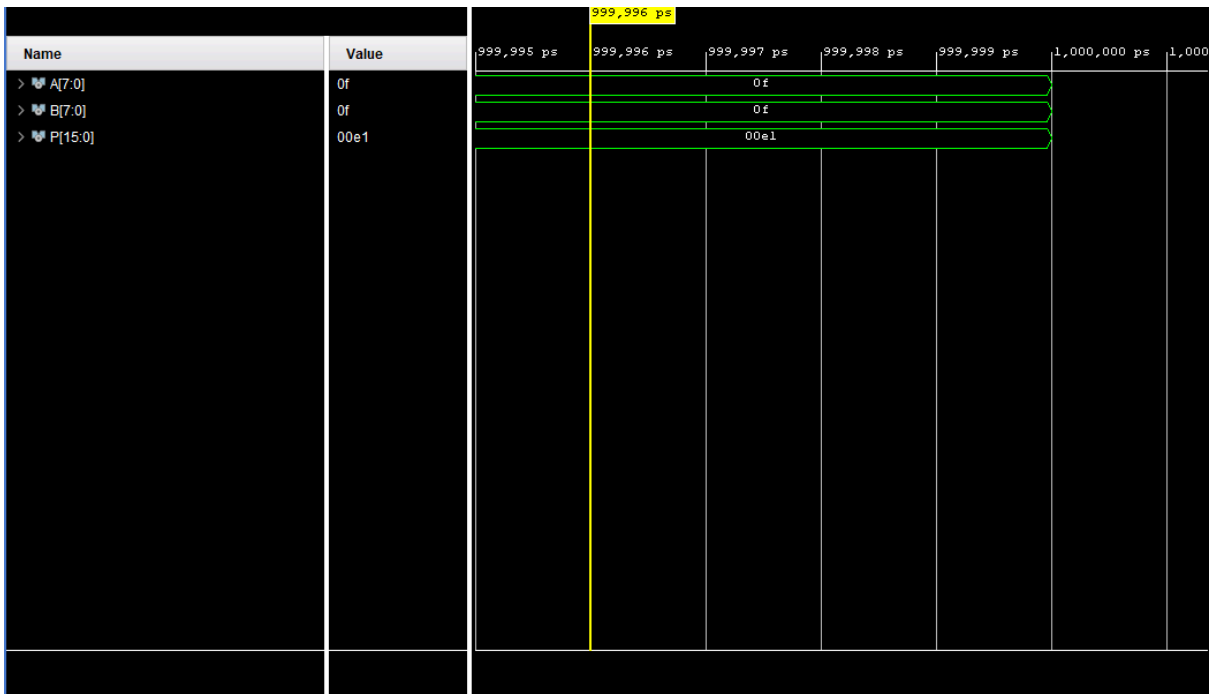


Figure 3: Braun result

Following successful simulation, the Braun multiplier was synthesised and implemented using the Xilinx Artix-7 FPGA device (xc7a35tcp236-1). The generated RTL schematic showed the expected regular array multiplier structure consisting of partial product generation and multiple addition stages. The synthesis reports provided information regarding FPGA resource utilisation and estimated power consumption, confirming that the design was suitable for FPGA implementation.

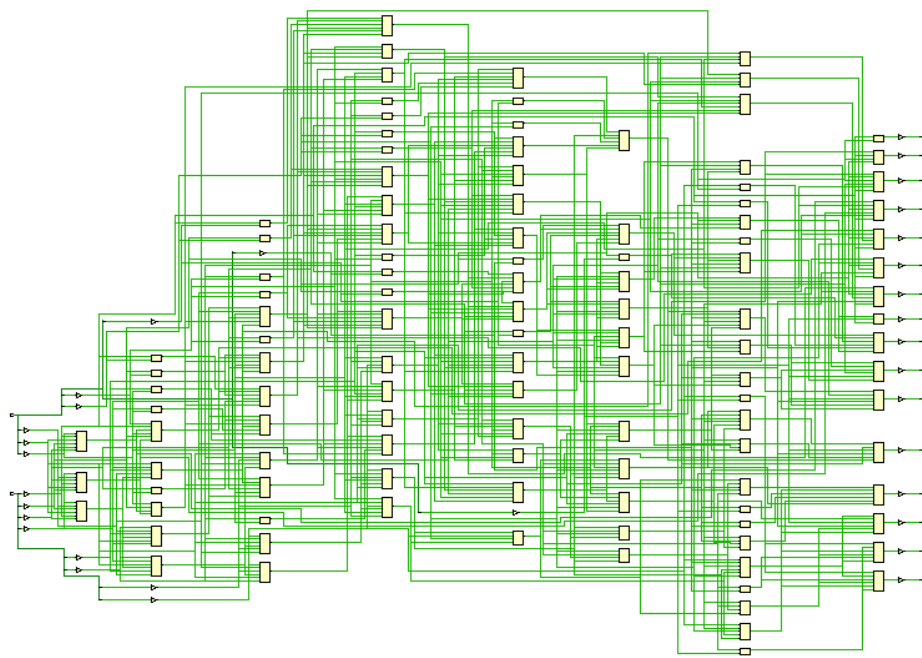


Figure 4: Braun RTL Schematic

Resource	Utilization	Available	Utilization %
LUT	89	20800	0.43
IO	32	106	30.19

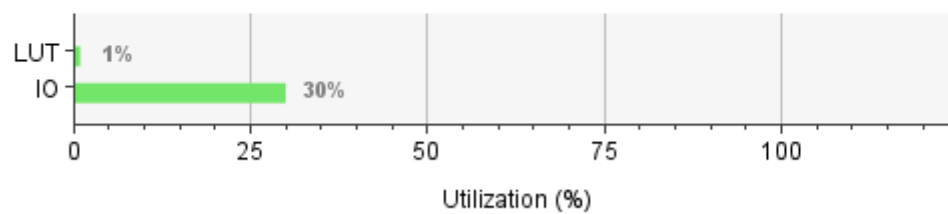


Figure 5: Braun utilisation

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power: **14.414 W (Junction temp exceeded!)**
Design Power Budget: **Not Specified**
Process: **typical**
Power Budget Margin: **N/A**
Junction Temperature: **97.1°C**
Thermal Margin: **-12.1°C (-2.3 W)**
Ambient Temperature: 25.0 °C
Effective θ_{JA} : 5.0°C/W
Power supplied to off-chip devices: 0 W
Confidence level: **Low**

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

On-Chip Power

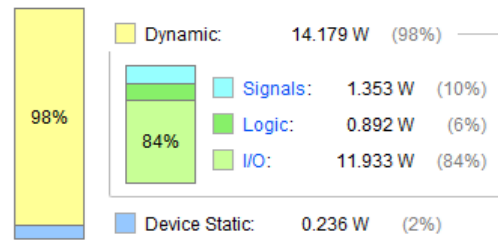


Figure 6: Braun power usage

The Wallace Tree multiplier was also successfully synthesised using Xilinx Vivado. The generated RTL schematic demonstrated the expected parallel reduction structure, where partial products are compressed using multiple arithmetic stages before the final addition process. The synthesis reports provided FPGA resource utilisation and power estimation results, allowing comparison between the two multiplier architectures.

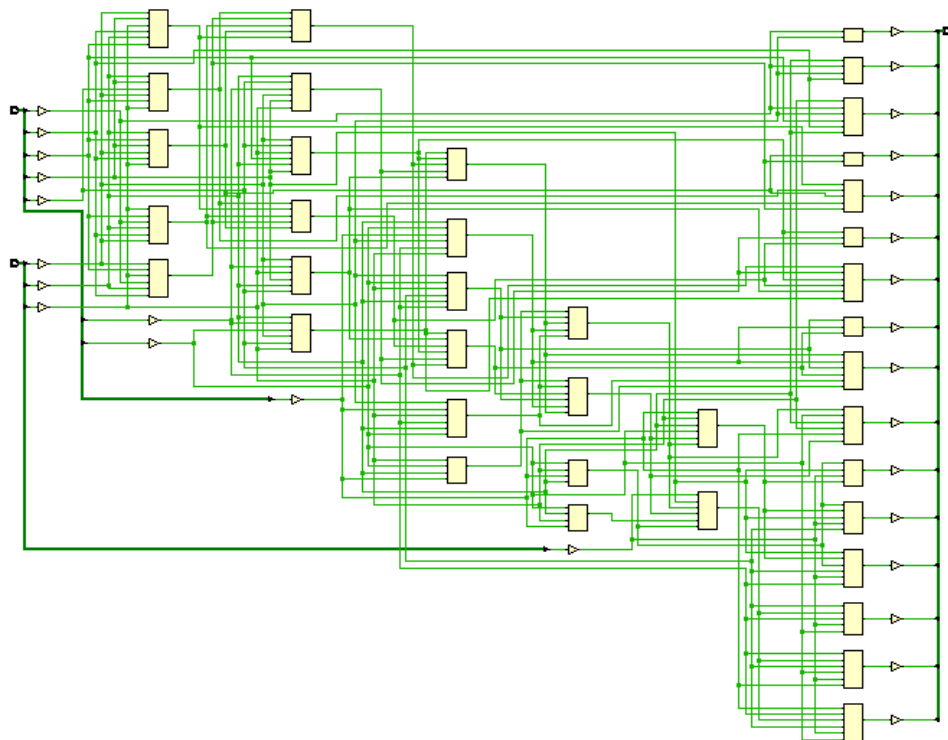


Figure 7: Wallace RTL Schematic

Resource	Utilization	Available	Utilization %
LUT	34	20800	0.16
IO	28	106	26.42

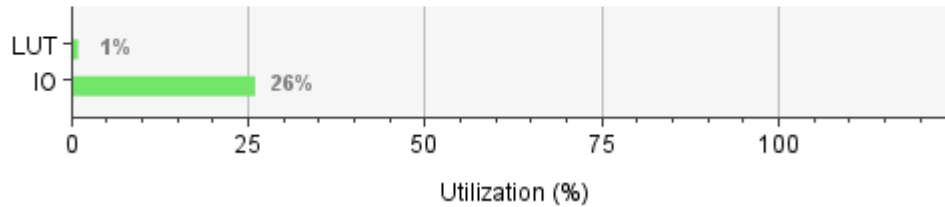


Figure 8: Wallace utilisation

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power: 7.733 W
Design Power Budget: Not Specified
Process: typical
Power Budget Margin: N/A
Junction Temperature: 63.7°C
 Thermal Margin: 21.3°C (4.2 W)
 Ambient Temperature: 25.0 °C
 Effective θ_{JA} : 5.0°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Low
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

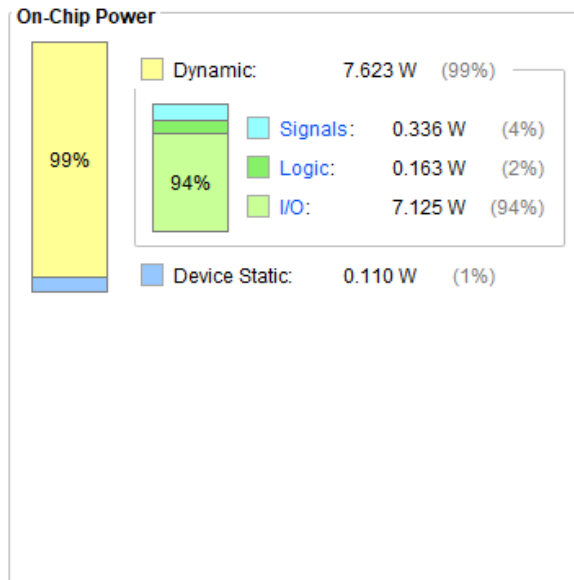


Figure 9: Wallace power

Behavioural simulation of the Wallace Tree multiplier was attempted; however, the simulation output contained unresolved ('U') values. This indicates that some internal signals were not fully assigned or connected correctly within the structural VHDL implementation. The issue is likely related to signal connections within the reduction and carry propagation stages. Although complete functional verification could not be achieved, successful synthesis confirms that the Wallace Tree design was structurally valid and capable of FPGA implementation. Therefore, the Wallace Tree multiplier was evaluated using its synthesis results and theoretical architectural advantages rather than simulation results.

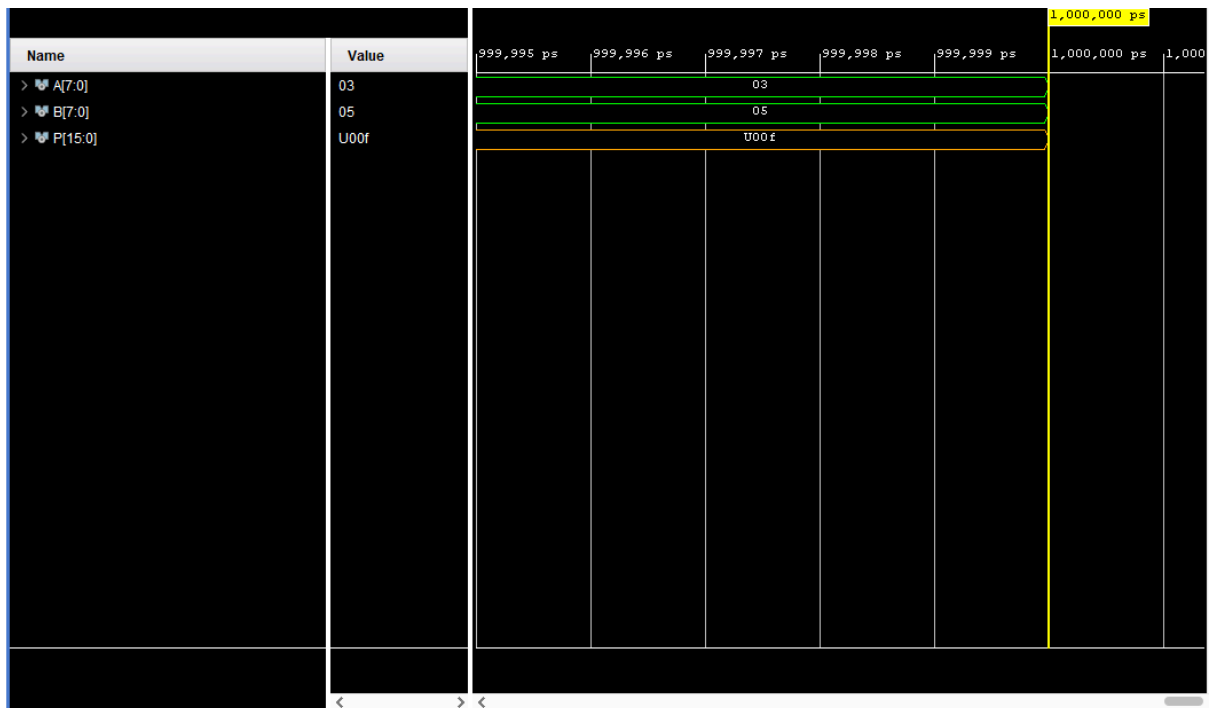


Figure 10: Wallace result

Timing Analysis

The timing analysis results for both the Braun and Wallace Tree multiplier implementations produced identical values. Vivado reported a Worst Negative Slack (WNS) of **inf** and a Total Negative Slack (TNS) of **0.00 ns** for both designs. The timing reports showed **16 total endpoints with 0 failing endpoints**, indicating that all analysed paths met the defined timing requirements.

The identical timing results are expected because both multiplier implementations were evaluated as unconstrained combinational designs. No clock signal or target operating frequency was defined, meaning Vivado was unable to perform a meaningful clock-based timing comparison between the two architectures. Instead, the timing report only confirms that both designs were successfully synthesised without any timing violations under the given constraints.

Although the Vivado timing summaries are identical, the theoretical timing behaviour of the two architectures differs. The Braun multiplier relies on a regular array structure where carry signals propagate through multiple full adder stages, resulting in a longer critical path. In contrast, the Wallace Tree multiplier reduces partial products through parallel compression stages, reducing the number of sequential logic levels and therefore providing a shorter theoretical critical path.

Therefore, while both designs achieved identical unconstrained timing reports, the architectural analysis indicates that the Wallace Tree multiplier should provide improved timing performance when implemented in a clocked system with defined timing constraints.

Worst Negative Slack (WNS): inf
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 16

Figure 11: Braun multiplier timing summary

Worst Negative Slack (WNS): inf
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 16

Figure 12: Wallace timing summary

Critical Path Evaluation

The critical path of the Braun multiplier begins from the partial product generation stage and continues through the longest chain of full adders within the multiplier array. The carry signals must propagate across multiple addition stages before reaching the most significant output bits. This sequential carry propagation is the main factor limiting the operating speed of the Braun multiplier.

In comparison, the Wallace Tree multiplier reduces the number of sequential arithmetic operations by compressing partial products in parallel. The critical path passes through the reduction stages and the final addition stage rather than through a long chain of carry-propagating adders. This reduced logic depth allows the Wallace Tree multiplier to achieve improved timing performance.

From an engineering perspective, the difference between the two architectures is caused by the method used to combine partial products. The Braun multiplier prioritises simplicity and regularity, whereas the Wallace Tree multiplier prioritises speed by increasing parallel processing.

Performance Comparison

The synthesis and implementation results demonstrate the different design priorities of the Braun and Wallace Tree multiplier architectures.

The Braun multiplier provides a simple and predictable hardware structure. The regular arrangement of AND gates, half adders and full adders makes the design easier to understand, implement and debug. This structure is beneficial for smaller designs where simplicity and reliability are important. However, the carry propagation through multiple addition stages increases the critical path delay and limits the achievable operating frequency.

The Wallace Tree multiplier provides improved timing performance by reducing partial products through parallel compression stages. Although the architecture requires more complex routing and a more complicated hardware arrangement, fewer sequential addition stages are required. This reduces propagation delay and makes the Wallace Tree multiplier more suitable for high-performance applications such as digital signal processing, communication systems and artificial intelligence hardware.

The Vivado implementation results allow comparison of the hardware requirements of both architectures. The Wallace Tree multiplier may require additional resources due to its increased number of interconnections and reduction stages. However, this additional complexity provides improved computational speed. The Braun multiplier requires a simpler implementation but sacrifices performance due to its longer critical path.

Overall, the results demonstrate the trade-off between hardware simplicity and computational performance. The Braun multiplier is suitable for applications where low design complexity is required, while the Wallace Tree multiplier is more appropriate where high-speed multiplication is the main design requirement.

Conclusion

This project successfully designed, implemented and analysed two 8-bit unsigned parallel multiplier architectures using structural VHDL and Xilinx Vivado Design Suite. The Braun multiplier was successfully verified through behavioural simulation, where the applied test cases produced the expected multiplication outputs. The design was then synthesised for the Xilinx Artix-7 FPGA, with Vivado providing RTL schematic generation, resource utilisation analysis and power estimation results.

The Wallace Tree multiplier was also successfully synthesised and implemented using the same FPGA target device. The generated RTL schematic demonstrated the expected parallel reduction architecture, where partial products are compressed through multiple addition stages. However, behavioural simulation of the Wallace Tree multiplier produced unresolved output values due to remaining internal signal connection issues. Despite this limitation, the synthesis results confirmed that the design structure was suitable for FPGA implementation, allowing architectural analysis and comparison with the Braun multiplier.

The comparison between the two architectures demonstrated the fundamental trade-off between hardware simplicity and computational performance. The Braun multiplier provides a regular and straightforward design structure, making it easier to implement and verify. However, its sequential carry propagation results in a longer critical path and increased propagation delay. In contrast, the Wallace Tree multiplier reduces the number of sequential addition stages through parallel partial product compression, resulting in improved theoretical speed performance and better scalability for larger multiplier designs.

Overall, the Wallace Tree multiplier is more suitable for high-performance applications where processing speed is a priority, such as digital signal processing and artificial intelligence hardware. The Braun multiplier remains a practical solution for applications where simplicity, reliability and ease of implementation are more important than maximum operating speed. Future improvements could include correcting the Wallace Tree simulation issues, implementing faster adder architectures and introducing pipelining techniques to further improve speed and power efficiency.

Bibliography

- [1] IEEE, "Multiplying Circuits," *IEEE Technology Navigator*.
<https://technav.ieee.org/topic/multiplying-circuits/>
- [2] Lilac Education Press, "Design and FPGA Implementation of High-Performance Parallel Multipliers Using Radix Booth Algorithm, Wallace Tree, and Carry Look-Ahead Adder for Optimal Speed, Area, and Power Efficiency," *Lilac Education Press (LEP)*.
<https://press.lilaceducation.com/fpga-implementation-high-performance-parallel-multipliers/>
- [3] S. Rani, A. Kumar, V. Singla, and R. Singla, "Performance Analysis of Different 8X8 Bit CMOS Multiplier Using 65Nm Technology," *International Journal of Computer Applications*, vol. 148, no. 13, pp. 1–6, Aug. 2016, doi: 10.5120/ijca2016911049.
- [4] A. Sharma, A. Mittal, A. Singh, and R. Kapoor, "Design and Implementation of Braun Multiplier Using Verilog: Review," *2024 Asia Pacific Conference on Innovation in Technology (Apcit)*, pp. 1–6, July 2024, doi: 10.1109/apcit62007.2024.10673483.
- [5] I. Hussain, R. Sah, M. Kumar, and P. Student, "Performance Comparison of Wallace Multiplier Architectures," *International Journal of Innovative Research in Science, Engineering and Technology (an ISO)*, vol. 3297, no. 1, 2007, doi: 10.15680/IJRSET.2015.0401052.
- [6] S. Nair and A. Saraf, "A Review Paper on Comparison of Multipliers Based on Performance Parameters General Terms."
- [7] "Design and Analysis of Braun Multiplier Using Cmos/Gdi Technique for Signal Processing," *International Journal of Emerging Trends in Engineering Research*, vol. 8, no. 10, pp. 6917–6924, Oct. 2020, doi: 10.30534/ijeter/2020/488102020.
- [8] "Wallace Tree," Sept. 2021. [Online]. Available:
https://en.wikipedia.org/wiki/Wallace_tree
- [9] "Call for Papers – Ijert," *Ijert.org*, 2025.
<https://www.ijert.org/research/design-and-performance-analysis-of-multipliers-using-different-logic-styles-ijertv4is040896.pdf>

Annex

Braun code

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

entity braun_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 braun_multiplier;

architecture Behavioral of braun_multiplier is

component full_adder

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

end component;

component half_adder

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

end component;

-- Partial product signals

```

signal A0B0,A1B0,A2B0,A3B0,A4B0,A5B0,A6B0,A7B0 : STD_LOGIC;
signal A0B1,A1B1,A2B1,A3B1,A4B1,A5B1,A6B1,A7B1 : STD_LOGIC;
signal A0B2,A1B2,A2B2,A3B2,A4B2,A5B2,A6B2,A7B2 : STD_LOGIC;
signal A0B3,A1B3,A2B3,A3B3,A4B3,A5B3,A6B3,A7B3 : STD_LOGIC;
signal A0B4,A1B4,A2B4,A3B4,A4B4,A5B4,A6B4,A7B4 : STD_LOGIC;
signal A0B5,A1B5,A2B5,A3B5,A4B5,A5B5,A6B5,A7B5 : STD_LOGIC;
signal A0B6,A1B6,A2B6,A3B6,A4B6,A5B6,A6B6,A7B6 : STD_LOGIC;
signal A0B7,A1B7,A2B7,A3B7,A4B7,A5B7,A6B7,A7B7 : STD_LOGIC;

```

-- Intermediate sums

```

signal P2_1 : STD_LOGIC;
signal P3_1,P3_2 : STD_LOGIC;
signal P4_1,P4_2,P4_3 : STD_LOGIC;
signal P5_1,P5_2,P5_3,P5_4 : STD_LOGIC;
signal P6_1,P6_2,P6_3,P6_4,P6_5 : STD_LOGIC;
signal P7_1,P7_2,P7_3,P7_4,P7_5,P7_6 : STD_LOGIC;
signal P8_1,P8_2,P8_3,P8_4,P8_5,P8_6 : STD_LOGIC;
signal P9_1,P9_2,P9_3,P9_4,P9_5 : STD_LOGIC;
signal P10_1,P10_2,P10_3,P10_4 : STD_LOGIC;
signal P11_1,P11_2,P11_3 : STD_LOGIC;
signal P12_1,P12_2 : STD_LOGIC;
signal P13_1 : STD_LOGIC;

```

-- Carry signals

```

signal C1 : STD_LOGIC;
signal C2_1,C2 : STD_LOGIC;
signal C3_1,C3_2,C3 : STD_LOGIC;
signal C4_1,C4_2,C4_3,C4 : STD_LOGIC;
signal C5_1,C5_2,C5_3,C5_4,C5 : STD_LOGIC;
signal C6_1,C6_2,C6_3,C6_4,C6_5,C6 : STD_LOGIC;
signal C7_1,C7_2,C7_3,C7_4,C7_5,C7_6,C7 : STD_LOGIC;
signal C8_1,C8_2,C8_3,C8_4,C8_5,C8_6,C8 : STD_LOGIC;
signal C9_1,C9_2,C9_3,C9_4,C9_5,C9 : STD_LOGIC;
signal C10_1,C10_2,C10_3,C10_4,C10 : STD_LOGIC;
signal C11_1,C11_2,C11_3,C11 : STD_LOGIC;
signal C12_1,C12_2,C12 : STD_LOGIC;
signal C13_1,C13 : STD_LOGIC;
signal C14 : STD_LOGIC;

```

begin

-- Partial Product Generation

```

A0B0 <= A(0) AND B(0);
A1B0 <= A(1) AND B(0);
A2B0 <= A(2) AND B(0);
A3B0 <= A(3) AND B(0);

```

A4B0 <= A(4) AND B(0);
A5B0 <= A(5) AND B(0);
A6B0 <= A(6) AND B(0);
A7B0 <= A(7) AND B(0);

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

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

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

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

A0B5 <= A(0) AND B(5);
A1B5 <= A(1) AND B(5);

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

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

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

```
-- First multiplication column
P(0) <= A0B0;
```

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

```
-- Column 2
P21: full_adder
port map(
    A => A2B0,
    B => A1B1,
    Cin => C1,
    S => P2_1,
    Cout => C2_1
```

```
);
```

```
P2_0: half_adder
```

```
port map(
```

```
  A => A0B2,
```

```
  B => P2_1,
```

```
  S => P(2),
```

```
  Cout => C2
```

```
);
```

```
-- Column 3
```

```
P31: full_adder
```

```
port map(
```

```
  A => A3B0,
```

```
  B => A2B1,
```

```
  Cin => C2_1,
```

```
  S => P3_1,
```

```
  Cout => C3_1
```

```
);
```

```
P32: full_adder
```

```
port map(
```

```
  A => A1B2,
```

```
  B => P3_1,
```

```
  Cin => C2,
```

```
  S => P3_2,
```

```
  Cout => C3_2
```

```
);
```

```
P3_0: half_adder
```

```
port map(
```

```
  A => A0B3,
```

```
  B => P3_2,
```

```
  S => P(3),
```

```
  Cout => C3
```

```
);
```

```
-- Column 4
```

```
P41: full_adder
```

```
port map(
```

```
  A => A4B0,
```

```
  B => A3B1,
```

```
  Cin => C3_1,
```

```
    S => P4_1,  
    Cout => C4_1  
);
```

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

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

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

```
-- Column 5  
P51: full_adder  
port map(  
    A => A5B0,  
    B => A4B1,  
    Cin => C4_1,  
    S => P5_1,  
    Cout => C5_1  
);
```

```
P52: full_adder  
port map(  
    A => A3B2,
```

```
    B => P5_1,  
    Cin => C4_2,  
    S => P5_2,  
    Cout => C5_2  
);
```

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

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

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

```
-- Column 6  
P61: full_adder  
port map(  
    A => A6B0,  
    B => A5B1,  
    Cin => C5_1,  
    S => P6_1,  
    Cout => C6_1  
);
```

```
P62: full_adder
```

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

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

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

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

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

```
-- Column 7
P71: full_adder
port map(
  A => A7B0,
  B => A6B1,
  Cin => C6_1,
  S => P7_1,
  Cout => C7_1
);
```

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

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

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

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

```
);
```

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

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

```
-- Column 8  
P81: full_adder  
port map(  
  A => A7B1,  
  B => A6B2,  
  Cin => C7_1,  
  S => P8_1,  
  Cout => C8_1  
);
```

```
P82: full_adder  
port map(  
  A => A5B3,  
  B => P8_1,  
  Cin => C7_2,  
  S => P8_2,  
  Cout => C8_2  
);
```

```
P83: full_adder  
port map(  
  A => A4B4,  
  B => P8_2,  
  Cin => C7_3,
```

```
    S => P8_3,  
    Cout => C8_3  
);
```

```
P84: full_adder  
port map(  
    A => A3B5,  
    B => P8_3,  
    Cin => C7_4,  
    S => P8_4,  
    Cout => C8_4  
);
```

```
P85: full_adder  
port map(  
    A => A2B6,  
    B => P8_4,  
    Cin => C7_5,  
    S => P8_5,  
    Cout => C8_5  
);
```

```
P86: full_adder  
port map(  
    A => A1B7,  
    B => P8_5,  
    Cin => C7_6,  
    S => P8_6,  
    Cout => C8_6  
);
```

```
P8_0: full_adder  
port map(  
    A => A7B0,  
    B => P8_6,  
    Cin => C7,  
    S => P(8),  
    Cout => C8  
);
```

```
-- Column 9  
P91: full_adder  
port map(  
    A => A8B0,  
    B => P8_7,  
    Cin => C8,  
    S => P(9),  
    Cout => C9  
);
```

```
    A => A7B2,  
    B => P8_1,  
    Cin => C8_1,  
    S => P9_1,  
    Cout => C9_1  
);
```

```
P92: full_adder  
port map(  
    A => A6B3,  
    B => P9_1,  
    Cin => C8_2,  
    S => P9_2,  
    Cout => C9_2  
);
```

```
P93: full_adder  
port map(  
    A => A5B4,  
    B => P9_2,  
    Cin => C8_3,  
    S => P9_3,  
    Cout => C9_3  
);
```

```
P94: full_adder  
port map(  
    A => A4B5,  
    B => P9_3,  
    Cin => C8_4,  
    S => P9_4,  
    Cout => C9_4  
);
```

```
P95: full_adder  
port map(  
    A => A3B6,  
    B => P9_4,  
    Cin => C8_5,  
    S => P9_5,  
    Cout => C9_5  
);
```

```
P96: full_adder
port map(
  A => A2B7,
  B => P9_5,
  Cin => C8_6,
  S => P(9),
  Cout => C9
);
```

```
-- Column 10
P101: full_adder
port map(
  A => A7B3,
  B => P9_1,
  Cin => C9_1,
  S => P10_1,
  Cout => C10_1
);
```

```
P102: full_adder
port map(
  A => A6B4,
  B => P10_1,
  Cin => C9_2,
  S => P10_2,
  Cout => C10_2
);
```

```
P103: full_adder
port map(
  A => A5B5,
  B => P10_2,
  Cin => C9_3,
  S => P10_3,
  Cout => C10_3
);
```

```
P104: full_adder
port map(
  A => A4B6,
  B => P10_3,
  Cin => C9_4,
  S => P10_4,
  Cout => C10_4
);
```

```
P105: full_adder
port map(
  A => A3B7,
  B => P10_4,
  Cin => C9_5,
  S => P(10),
  Cout => C10
);
```

```
-- Column 11
P111: full_adder
port map(
  A => A7B4,
  B => P10_1,
  Cin => C10_1,
  S => P11_1,
  Cout => C11_1
);
```

```
P112: full_adder
port map(
  A => A6B5,
  B => P11_1,
  Cin => C10_2,
  S => P11_2,
  Cout => C11_2
);
```

```
P113: full_adder
port map(
  A => A5B6,
  B => P11_2,
  Cin => C10_3,
  S => P11_3,
  Cout => C11_3
);
```

```
P114: full_adder
port map(
  A => A4B7,
  B => P11_3,
  Cin => C10_4,
```

```
    S => P(11),  
    Cout => C11  
);
```

```
-- Column 12  
P121: full_adder  
port map(  
    A => A7B5,  
    B => P11_1,  
    Cin => C11_1,  
    S => P12_1,  
    Cout => C12_1  
);
```

```
P122: full_adder  
port map(  
    A => A6B6,  
    B => P12_1,  
    Cin => C11_2,  
    S => P12_2,  
    Cout => C12_2  
);
```

```
P123: full_adder  
port map(  
    A => A5B7,  
    B => P12_2,  
    Cin => C11_3,  
    S => P(12),  
    Cout => C12  
);
```

```
-- Column 13  
P131: full_adder  
port map(  
    A => A7B6,  
    B => P12_1,  
    Cin => C12_1,  
    S => P13_1,  
    Cout => C13_1  
);
```

```
P132: full_adder  
port map(  
    A => A6B7,  
    B => P13_1,  
    Cin => C12_2,  
    S => P13_2,  
    Cout => C13_2  
);
```

```
    A => A6B7,  
    B => P13_1,  
    Cin => C12_2,  
    S => P(13),  
    Cout => C13  
);
```

```
-- Column 14 and 15  
P141: full_adder  
port map(  
    A => A7B7,  
    B => P13_1,  
    Cin => C13,  
    S => P(14),  
    Cout => C14  
);
```

```
P(15) <= C14;  
  
end Behavioral;
```