

BEng Embedded  
Electronic Systems  
Electronics II  
ACDA6001  
Joshua Ofosu Asare  
2111739

## Contents

|       |                                                          |    |
|-------|----------------------------------------------------------|----|
| 1     | Introduction .....                                       | 3  |
| 2     | Background Research / Literature Review .....            | 3  |
| 3     | Task 1 – Braun Multiplier .....                          | 3  |
| 3.1   | Logic circuit diagram .....                              | 4  |
| 3.2   | Schematic .....                                          | 4  |
| 4     | Task 2 – Wallace Multiplier .....                        | 5  |
| 4.1   | Logic circuit diagram .....                              | 5  |
| 4.2   | Schematic .....                                          | 5  |
| 5     | Task 3 – Multiplier Evaluation .....                     | 6  |
| 5.1   | Test bench .....                                         | 6  |
| 5.2   | Longest delay .....                                      | 6  |
| 5.2.1 | Wallace tree multiplier combinational delay .....        | 7  |
| 5.2.2 | Braun multiplier combinational delay .....               | 8  |
| 5.2.3 | Calculated delays .....                                  | 9  |
| 5.2.4 | Multiplier delay comparison .....                        | 9  |
| 5.3   | Power consumption .....                                  | 9  |
| 5.3.1 | Braun multiplier power consumption .....                 | 10 |
| 5.3.2 | Wallace tree multiplier power consumption .....          | 10 |
| 5.3.3 | Braun vs Wallace tree multiplier power consumption ..... | 10 |
| 5.4   | Utilisation .....                                        | 11 |
| 5.4.1 | Braun multiplier utilisation .....                       | 11 |
| 5.4.2 | Wallace tree multiplier utilisation .....                | 11 |
| 6     | Conclusion .....                                         | 11 |
| 7     | Recommendations .....                                    | 12 |
| 8     | References .....                                         | 13 |

## 1 Introduction

Multipliers are essential in digital circuits for many tasks across different complexities. This assignment will critically evaluate the performance of a Braun and Wallace Tree multiplier simulated, synthesised and implemented on an xc7a15tcpg236-1 FPGA in Xilinx Vivado [1]. Multipliers are the backbone of complex digital signal processing algorithms such as Fast Fourier Transforms, Finite Impulse Responses [2], Infinite Impulse Responses and Convolutions [3], image/video processing and machine learning [4].

## 2 Background Research / Literature Review

Multiplication of decimal number, when done by hand, is often completed using long multiplication [5]. Long multiplication is the name applied to the summation of partial products, and is applicable to multiplication in number systems that do not use base 10 [6].

One multiplier that may be found in a digital circuit is a Braun multiplier [7] is comprised of “Carry Save Adder stages for generating partial products and one Ripple Carry Adder [6] stage which give final 4 MSB Product Bits.” The partial products are summed together to form the final product, similar to long multiplication [5]. Calculating partial products and summing them to create a final product is a method of multiplication that works in different number bases. As we are implementing a digital circuit, we must take into consideration that we are restricted to base 2.

The Wallace Tree multiplier is one such example of a parallel multiplier [8], which means it can generate parts of the final product in a non-linear fashion. It makes use of a carry look ahead adder enables faster computation of results by shortening propagation delay throughout the circuit [9]. Additionally, the Wallace tree multiplier has applications in non-FPGA digital circuits such as RISC-V processors [10].

## 3 Task 1 – Braun Multiplier

The Braun multiplier has been designed to take two unsigned numbers, a multiplicand and multiplier, represented by  $a$  and  $b$ , with the most significant and least significant bits being  $a_7$ ,  $b_7$  and  $a_0$ ,  $b_0$  respectively.

### 3.1 Logic circuit diagram

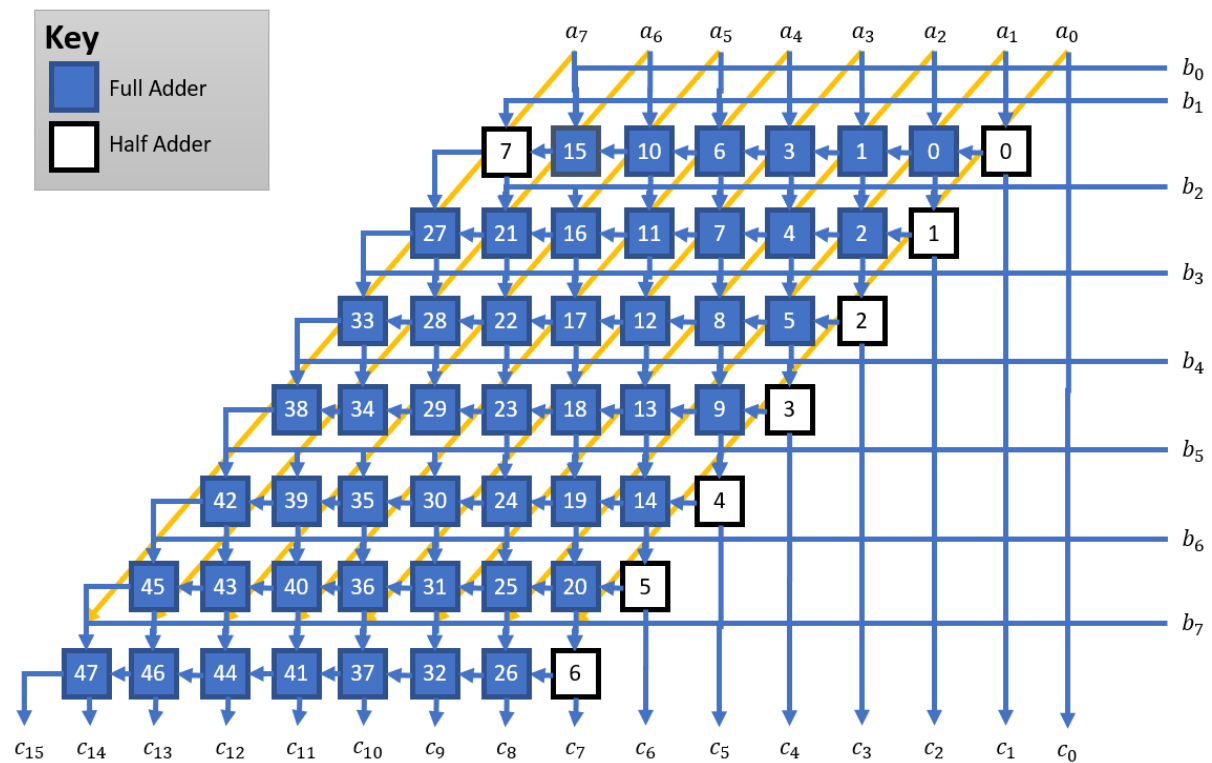


Figure 1: Circuit diagram of an 8-bit Braun multiplier

### 3.2 Schematic

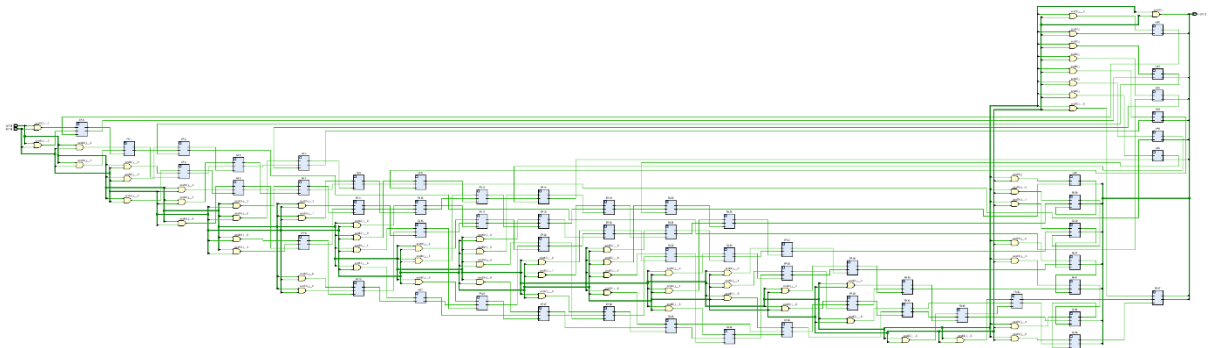


Figure 2: Post simulation schematic of an 8-bit Braun multiplier in Xilinx Vivado

## 4 Task 2 – Wallace Multiplier

### 4.1 Logic circuit diagram

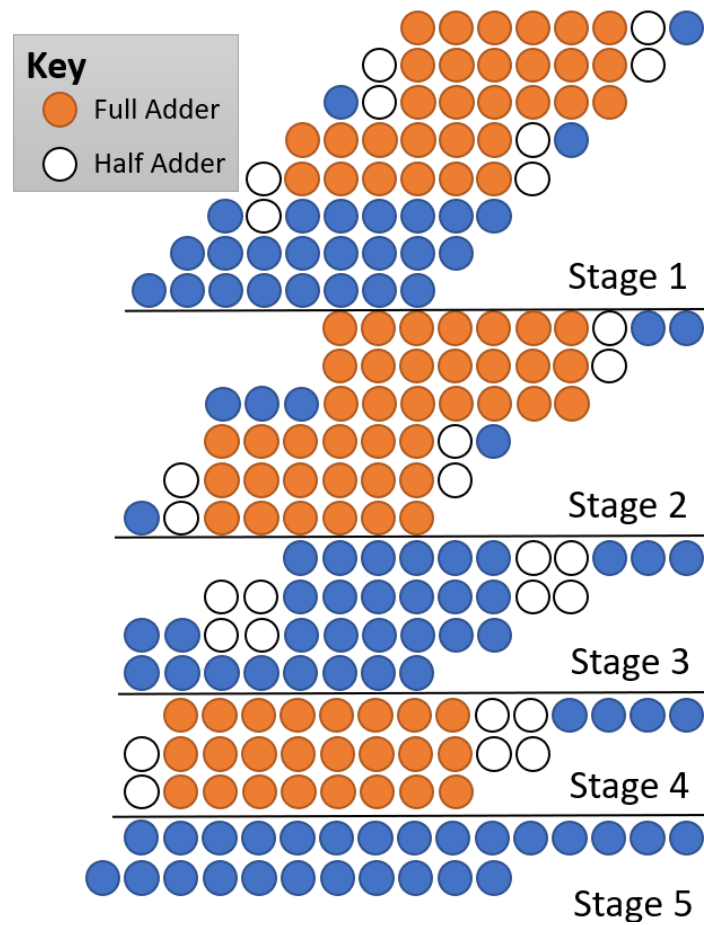


Figure 3: Circuit diagram of an 8-bit Wallace Tree multiplier

### 4.2 Schematic

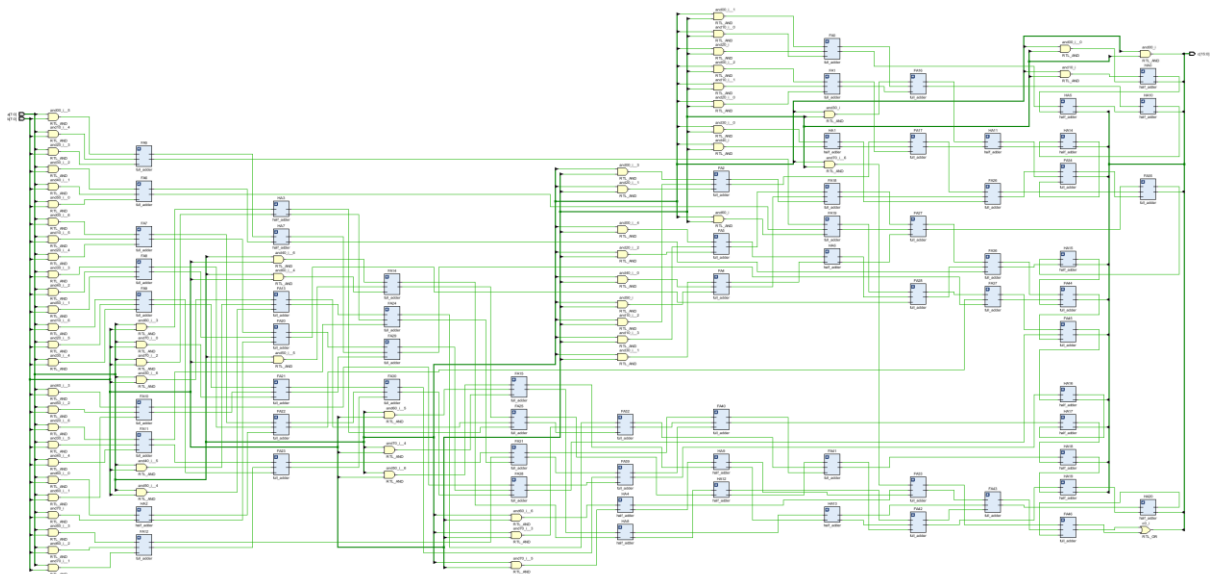


Figure 4: Post simulation schematic of an 8-bit Braun multiplier in Xilinx Vivado

## 5 Task 3 – Multiplier Evaluation

### 5.1 Test bench

The accuracy of the output from both multipliers was verified through the use of a test bench. Using the multiply operator “\*” to store the result of the two inputs multiplier together, a variable named “resultCheck” was created to store a comparison between the output of a multiplier and the internal multiply.

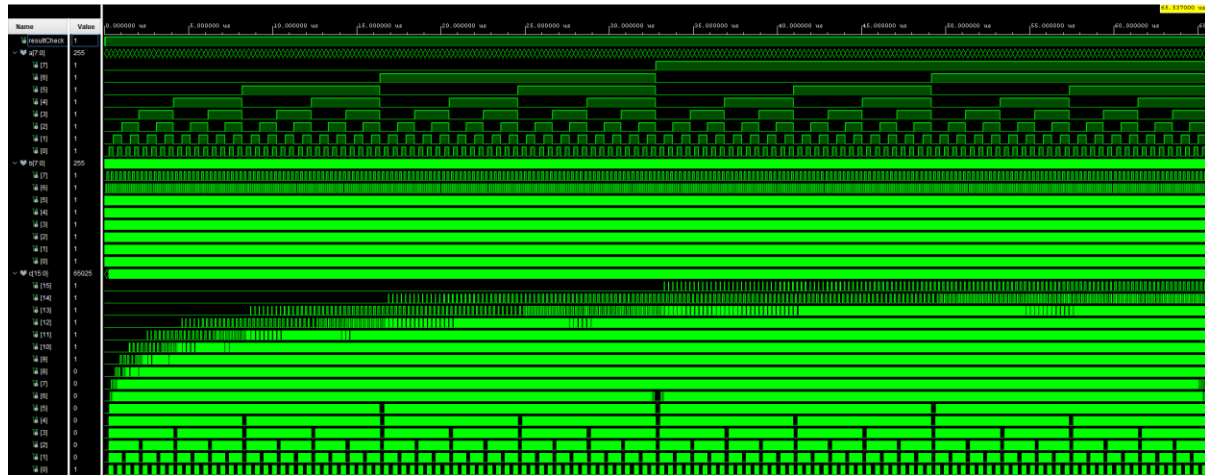


Figure 5: Screenshot of the simulation screen after running a Braun multiplier test bench

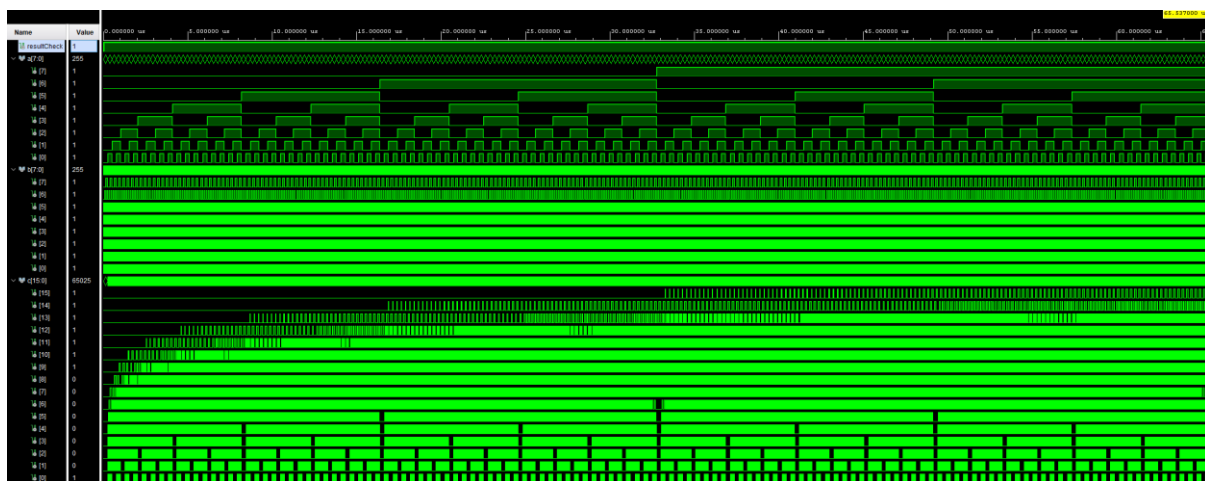


Figure 6: An identical output from the simulation of a Wallace Tree multiplier test bench

### 5.2 Longest delay

In order to calculate the longest delay, the “resultCheck” variable that was used in the test benches was removed prior to simulations that were used to retrieve timing values as it made use of integral multipliers to verify the output of the multipliers created in this assignment.

### 5.2.1 Wallace tree multiplier combinational delay

| Combinational Delays |         |           |                    |
|----------------------|---------|-----------|--------------------|
| From Port            | To Port | Max Delay | Max Process Corner |
| a[6]                 | d[14]   | 18.327    | SLOW               |
| b[5]                 | d[14]   | 18.157    | SLOW               |
| a[6]                 | d[15]   | 18.147    | SLOW               |
| b[5]                 | d[15]   | 17.977    | SLOW               |
| a[4]                 | d[14]   | 17.940    | SLOW               |
| a[5]                 | d[14]   | 17.795    | SLOW               |
| a[4]                 | d[15]   | 17.761    | SLOW               |
| b[4]                 | d[14]   | 17.677    | SLOW               |
| a[5]                 | d[15]   | 17.615    | SLOW               |
| b[4]                 | d[15]   | 17.497    | SLOW               |
| a[6]                 | d[12]   | 17.448    | SLOW               |
| a[7]                 | d[14]   | 17.353    | SLOW               |
| b[5]                 | d[12]   | 17.278    | SLOW               |
| a[6]                 | d[13]   | 17.237    | SLOW               |
| b[6]                 | d[14]   | 17.215    | SLOW               |
| a[7]                 | d[15]   | 17.174    | SLOW               |
| b[5]                 | d[13]   | 17.067    | SLOW               |
| a[4]                 | d[12]   | 17.062    | SLOW               |
| b[6]                 | d[15]   | 17.036    | SLOW               |
| a[5]                 | d[12]   | 16.916    | SLOW               |
| a[4]                 | d[13]   | 16.851    | SLOW               |
| b[5]                 | c[7]    | 16.809    | SLOW               |
| b[4]                 | d[12]   | 16.799    | SLOW               |
| b[3]                 | d[14]   | 16.768    | SLOW               |
| b[7]                 | d[14]   | 16.766    | SLOW               |
| a[6]                 | d[11]   | 16.747    | SLOW               |
| a[5]                 | d[13]   | 16.705    | SLOW               |
| a[5]                 | c[7]    | 16.689    | SLOW               |
| a[6]                 | d[10]   | 16.653    | SLOW               |
| b[3]                 | d[15]   | 16.588    | SLOW               |
| b[4]                 | d[13]   | 16.588    | SLOW               |
| b[7]                 | d[15]   | 16.586    | SLOW               |
| b[5]                 | d[11]   | 16.577    | SLOW               |
| a[4]                 | c[7]    | 16.559    | SLOW               |
| b[1]                 | d[14]   | 16.527    | SLOW               |
| b[5]                 | d[10]   | 16.483    | SLOW               |
| a[7]                 | d[12]   | 16.475    | SLOW               |
| b[2]                 | d[14]   | 16.449    | SLOW               |
| b[4]                 | c[7]    | 16.424    | SLOW               |
| a[2]                 | d[14]   | 16.392    | SLOW               |
| b[0]                 | d[14]   | 16.371    | SLOW               |
| a[4]                 | d[11]   | 16.361    | SLOW               |
| a[7]                 | d[13]   | 16.347    | SLOW               |
| b[1]                 | d[15]   | 16.347    | SLOW               |
| b[6]                 | d[12]   | 16.337    | SLOW               |
| a[1]                 | d[14]   | 16.301    | SLOW               |
| b[2]                 | d[15]   | 16.269    | SLOW               |
| a[4]                 | d[10]   | 16.266    | SLOW               |
| b[6]                 | d[13]   | 16.265    | SLOW               |

Figure 7: Maximum combinational delays between ports for a Wallace tree multiplier

## 5.2.2 Braun multiplier combinational delay

| Combinational Delays |         |           |                    |
|----------------------|---------|-----------|--------------------|
| From Port            | To Port | Max Delay | Max Process Corner |
| a[4]                 | c[12]   | 22.647    | SLOW               |
| a[4]                 | c[15]   | 22.608    | SLOW               |
| a[4]                 | c[13]   | 22.425    | SLOW               |
| a[5]                 | c[12]   | 22.369    | SLOW               |
| a[5]                 | c[15]   | 22.330    | SLOW               |
| a[5]                 | c[13]   | 22.147    | SLOW               |
| a[1]                 | c[12]   | 22.087    | SLOW               |
| a[2]                 | c[12]   | 22.070    | SLOW               |
| a[1]                 | c[15]   | 22.048    | SLOW               |
| a[2]                 | c[15]   | 22.031    | SLOW               |
| b[1]                 | c[12]   | 22.013    | SLOW               |
| b[1]                 | c[15]   | 21.974    | SLOW               |
| a[3]                 | c[12]   | 21.951    | SLOW               |
| b[0]                 | c[12]   | 21.950    | SLOW               |
| a[4]                 | c[14]   | 21.937    | SLOW               |
| a[3]                 | c[15]   | 21.912    | SLOW               |
| b[0]                 | c[15]   | 21.912    | SLOW               |
| a[1]                 | c[13]   | 21.864    | SLOW               |
| a[2]                 | c[13]   | 21.848    | SLOW               |
| b[1]                 | c[13]   | 21.791    | SLOW               |
| a[0]                 | c[12]   | 21.774    | SLOW               |
| a[0]                 | c[15]   | 21.735    | SLOW               |
| a[3]                 | c[13]   | 21.729    | SLOW               |
| b[0]                 | c[13]   | 21.728    | SLOW               |
| a[5]                 | c[14]   | 21.659    | SLOW               |
| a[0]                 | c[13]   | 21.552    | SLOW               |
| b[2]                 | c[12]   | 21.490    | SLOW               |
| a[4]                 | c[11]   | 21.465    | SLOW               |
| b[2]                 | c[15]   | 21.451    | SLOW               |
| a[1]                 | c[14]   | 21.377    | SLOW               |
| a[2]                 | c[14]   | 21.360    | SLOW               |
| b[1]                 | c[14]   | 21.303    | SLOW               |
| b[2]                 | c[13]   | 21.268    | SLOW               |
| a[3]                 | c[14]   | 21.241    | SLOW               |
| b[0]                 | c[14]   | 21.240    | SLOW               |
| a[5]                 | c[11]   | 21.187    | SLOW               |
| a[4]                 | c[10]   | 21.104    | SLOW               |
| a[0]                 | c[14]   | 21.064    | SLOW               |
| b[3]                 | c[12]   | 21.004    | SLOW               |
| b[3]                 | c[15]   | 20.965    | SLOW               |
| a[1]                 | c[11]   | 20.904    | SLOW               |
| a[2]                 | c[11]   | 20.888    | SLOW               |
| b[1]                 | c[11]   | 20.830    | SLOW               |
| a[5]                 | c[10]   | 20.826    | SLOW               |
| b[3]                 | c[13]   | 20.782    | SLOW               |
| b[2]                 | c[14]   | 20.780    | SLOW               |
| a[3]                 | c[11]   | 20.769    | SLOW               |
| b[0]                 | c[11]   | 20.768    | SLOW               |
| a[0]                 | c[11]   | 20.591    | SLOW               |

Figure 8: Maximum combinational delays between ports for a Wallace tree multiplier

### 5.2.3 Calculated delays

Delays for each multiplier that have been manually calculated below use the following assumptions:

- Each 'and gate' has a delay of 1ns;
- The processing delay in the half adder and full adder is 2ns;
- The propagation delay between logic elements is 0ns.

#### 5.2.3.1 Calculated delay in Braun multiplier

$$\text{and gate count} * t_g + \text{adder count} * t_{ad}$$

*Equation 1: Longest delay formula [11]*

Where  $t_g = 1ns$  and  $t_{ad} = 2ns$

$$1 * t_g + 20 * t_{ad} = \text{total delay}$$

$$1 * 1ns + 20 * 2ns = 41ns$$

#### 5.2.3.2 Calculated delay in Wallace tree multiplier

$$\text{and gate count} * t_g + \text{adder count} * t_{ad}$$

*Equation 2: Longest delay formula [11]*

Where  $t_g = 1ns$  and  $t_{ad} = 2ns$

$$1 * t_g + 12 * t_{ad} = \text{total delay}$$

$$1 * 1ns + 12 * 2ns = 25ns$$

### 5.2.4 Multiplier delay comparison

The delay in the Braun multiplier is calculated to be greater than that of the Wallace tree at 41ns vs 25ns, 64% slower. This highlights that there should be an expected difference in latency between the two multipliers, which is observed when comparing Figure 7 and Figure 8. The maximum delay from an input port to an output port for the Braun multiplier is 22.647ns, while the Wallace tree multiplier has a delay of 18.327ns.

## 5.3 Power consumption

The vast majority of the power consumed by the multiplier is device static which is a constant 0.070W and not related directly to the multiplier. The remaining power consumption is used by the multiplier and split into signals, the interconnects that run between logic, the logic itself, and I/O.

### 5.3.1 Braun multiplier power consumption

#### Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

**Total On-Chip Power:** 0.085 W  
**Design Power Budget:** 2 W  
**Power Budget Margin:** 1.915 W  
**Junction Temperature:** 25.4°C  
**Thermal Margin:** 59.6°C (11.8 W)  
**Effective  $\theta_{JA}$ :** 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

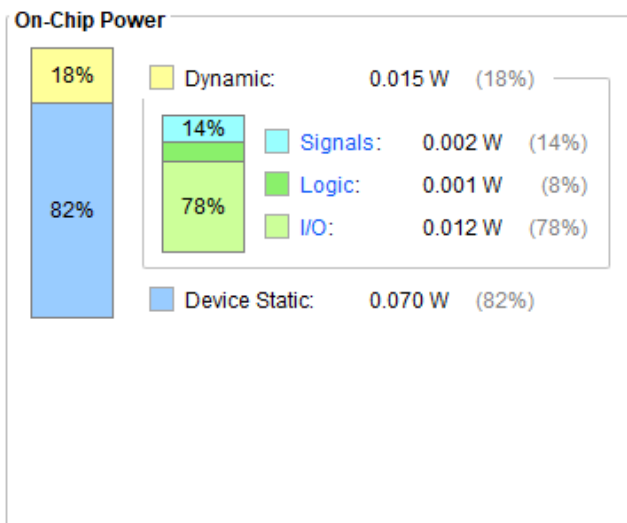


Figure 9: Power consumption from the implemented netlist for the Braun multiplier as shown in Xilinx Vivado

### 5.3.2 Wallace tree multiplier power consumption

#### Summary

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:** 0.084 W  
**Design Power Budget:** 2 W  
**Power Budget Margin:** 1.916 W  
**Junction Temperature:** 25.4°C  
**Thermal Margin:** 59.6°C (11.9 W)  
**Effective  $\theta_{JA}$ :** 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

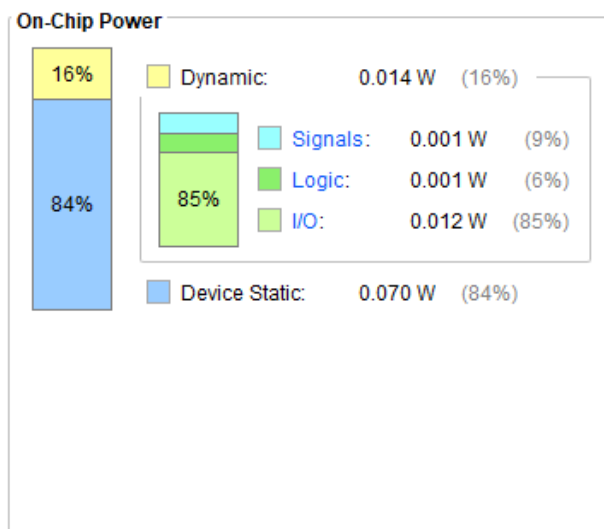


Figure 10: Power consumption from the implemented netlist for the Wallace tree multiplier as shown in Xilinx Vivado

### 5.3.3 Braun vs Wallace tree multiplier power consumption

The power consumption for the Braun and Wallace tree multipliers are shown above in Figure 9 and Figure 10 respectively. The overall power consumption for the two multipliers is almost identical, but analysing the differences highlights unexpected behaviour. Both multipliers have the same device I/O and device static power consumption, but unexpectedly the signals in the Braun multiplier draw more power than the Wallace tree multiplier. Overall, the Wallace tree uses 0.001W less power than the Braun multiplier.

## 5.4 Utilisation

The utilisation for both multipliers is roughly the same, with Xilinx Vivado [1] reporting 1% LUT usage and 30% IO (Input / Output) utilisation. For the purposes of this comparison, the IO utilisation may be disregarded as both multipliers use the same number of inputs and outputs, two 8-bit numbers are taken as inputs and a single 16-bit number is output.

The Braun multiplier uses 93 LUTs compared to 103 LUTs in the Wallace tree multiplier (0.89% vs 0.99%). If multipliers were to occupy the entire space of the FPGA, ~10% more Braun multipliers could be added.

### 5.4.1 Braun multiplier utilisation

#### Summary

|     |    |       |       |
|-----|----|-------|-------|
| LUT | 93 | 10400 | 0.89  |
| IO  | 32 | 106   | 30.19 |

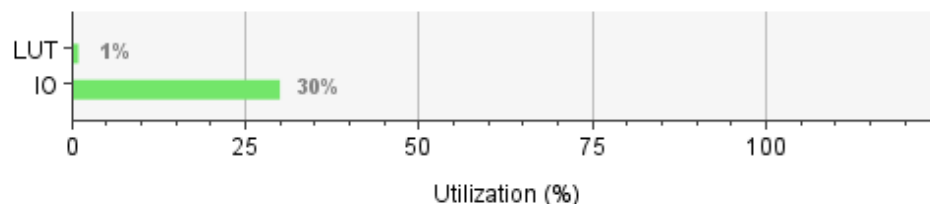


Figure 11: IO and LUT usage of a Braun multiplier implemented in Xilinx Vivado

### 5.4.2 Wallace tree multiplier utilisation

#### Summary

| Resource | Utilization | Available | Utilization % |
|----------|-------------|-----------|---------------|
| LUT      | 103         | 10400     | 0.99          |
| IO       | 32          | 106       | 30.19         |

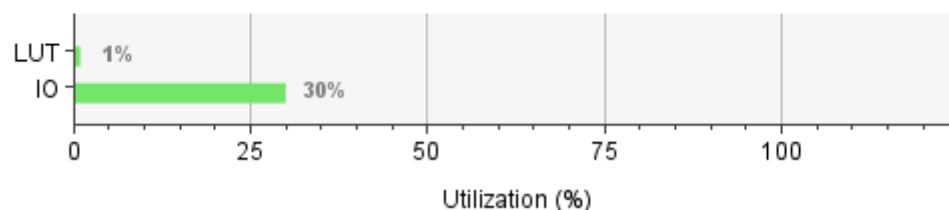


Figure 12: IO and LUT usage of a Wallace tree multiplier implemented in Xilinx Vivado

## 6 Conclusion

The Wallace tree multiplier is a faster multiplier that should be used when the design specification for a project requires a faster multiplier that is able to sacrifice the increase in power consumption.

The Braun multiplier is generally [7] suggested for projects where a more power efficient multiplier is required, but the additional speed of a Wallace tree multiplier is not required. However, in the

simulations conducted for this assignment, no significant difference was identified between the two despite several re-simulation. This could be due to redundant logic added in the VHDL files (see Annexes for VHDL source files used for simulation).

## 7 Recommendations

Careful consideration should be taken when choosing how to implement a multiplier, subject to the requirements of the larger design, to optimise for either power draw or multiplier speed. The simulation results examined in the above sections show that the Braun tree multiplier has a longer delay but lower complexity which makes it more suitable for a design where either power efficiency is important or design space is limited on the FPGA. Additional optimisations could be made to reduce the processing delay through the careful selection of input and output pins to reduce the distance signals have to travel. By reducing the distance signals are required to travel, the delay can be reduced. Further research into parallel multipliers and performing a similar comparison, where the power usage and longest delay is critically analysed could lead to the discovery of a more power or time efficient multiplier than the Braun or Wallace tree multipliers explored in this assignment.

Finally, the simulations did not show a significant difference between the two multipliers power consumption. It is recommended that different variants of both multipliers are implemented in order to differentiate the two in terms of power consumption.

## 8 References

- [1] AMD, "Vivado Overview," AMD, 12 12 2023. [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado.html>. [Accessed 8 December 2023].
- [2] W. Green, "Multiplication with FPGA DSPs," Project F, 27 November 2021. [Online]. Available: <https://projectf.io/posts/multiplication-fpga-dsps/>. [Accessed 7 December 2023].
- [3] T. e. a. Pham-Dinh, "An FPGA-based Solution for Convolution," *Intelligence of Things: Technologies and Applications*, vol. 148, pp. 279-288, 2022.
- [4] S. R. M. S. a. A. K. S. Ullah, "High-Performance Accurate and Approximate Multipliers for FPGA-Based Hardware Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 211-224, 2022.
- [5] A. Watkins, "How to do long multiplication," 14 November 2023. [Online]. Available: <https://doodlelearning.com/maths/skills/multiplication/long-multiplication>. [Accessed 8 December 2023].
- [6] HEX Hero, "How to multiply Hexadecimal numbers," HEX Hero, [Online]. Available: <https://www.hexhero.com/tutorials/how-to-multiply-hexadecimal-numbers>. [Accessed 8 December 2023].
- [7] A. J. Thakur. M, "Kogge Stone Adder & It's Implementation on FPGA," *International Journal of Scientific & Engineering Research*, vol. 3, no. 10, 2012.
- [8] VLSI Verify, "Wallace Tree Multiplier," VLSI Verify, [Online]. Available: <https://vlsiverify.com/verilog/verilog-codes/wallace-tree-multiplier/>. [Accessed 11 December 2023].
- [9] S. Das, "Carry Look-Ahead Adder," Geeks for Geeks, 23 February 2023. [Online]. Available: <https://www.geeksforgeeks.org/carry-look-ahead-adder/>. [Accessed 13 December 2023].
- [10] e. a. Himanshu Bansal, "Wallace Tree Multiplier Designs: A Performance Comparison," *Innovative Systems Design and Engineering*, vol. 5, no. 5, 2014.
- [11] J. Devaraj, "Simulation, Synthesis, Implementation and Task 3 using Wallace Multiplier," 12 12 2023. [Online]. Available: <https://moodle.uwtsd.ac.uk/mod/resource/view.php?id=546644>. [Accessed 12 12 2023].
- [12] J. Wawrzynek and N. Weaver, "EECS151/251A Sping 2018 Digital Design and Integrated Circuits," [Online]. Available: <https://inst.eecs.berkeley.edu/~eecs151/sp18/files/Lecture21.pdf>. [Accessed 11 December 2023].
- [13] P. Cheung, "\$5 - MULTIPLIER CIRCUITS," [Online]. Available: [http://www.ee.ic.ac.uk/pcheung/teaching/ee3\\_DSD/5-multipliers.pdf](http://www.ee.ic.ac.uk/pcheung/teaching/ee3_DSD/5-multipliers.pdf). [Accessed 8 December 2023].

## Annex A. Braun Multiplier VHDL

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

entity BRAUN_8BIT is
    Port (a : in STD_LOGIC_VECTOR(7 downto 0);
          b : in STD_LOGIC_VECTOR(7 downto 0);
          c : out STD_LOGIC_VECTOR(15 downto 0));
end BRAUN_8BIT;

architecture b8 of BRAUN_8BIT is

    -- Generated by a python script I wrote, each signal vector (and{n})
    -- represents a horizontal array of and gates.
    signal and0, and1, and2, and3, and4, and5, and6, and7 :
    STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    -- Generated by a python script I wrote, these arrays hold the the
    -- carry out bits for each horizontal array of adders.
    signal adder0_c, adder1_c, adder2_c, adder3_c, adder4_c, adder5_c,
    adder6_c : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    -- Generated by a python script I wrote, these arrays hold the the sum
    -- out bits for each horizontal array of adders.
    signal adder0_s, adder1_s, adder2_s, adder3_s, adder4_s, adder5_s,
    adder6_s : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');

    component
    half_adder is
    port (a_ha : in STD_LOGIC;
          b_ha : in STD_LOGIC;
          sum_ha : out STD_LOGIC;
          carry : out STD_LOGIC);
    end component;

    component
    full_adder is
    port (a_fa : in STD_LOGIC;
          b_fa : in STD_LOGIC;
          carry_in : in STD_LOGIC;
          sum_fa : out STD_LOGIC;
          carry_out : out STD_LOGIC);
```

```

    end component;

begin
    -- Generated by a python script I wrote, each signal vector (and{n})
    represents a horizontal array of and gates.
    -- They start with a(7) as the STD_LOGIC_VECTOR above that defines the
    signals use downto instead of upto
    and0 <= (a(7) AND b(0), a(6) AND b(0), a(5) AND b(0), a(4) AND b(0),
a(3) AND b(0), a(2) AND b(0), a(1) AND b(0), a(0) AND b(0));
    and1 <= (a(7) AND b(1), a(6) AND b(1), a(5) AND b(1), a(4) AND b(1),
a(3) AND b(1), a(2) AND b(1), a(1) AND b(1), a(0) AND b(1));
    and2 <= (a(7) AND b(2), a(6) AND b(2), a(5) AND b(2), a(4) AND b(2),
a(3) AND b(2), a(2) AND b(2), a(1) AND b(2), a(0) AND b(2));
    and3 <= (a(7) AND b(3), a(6) AND b(3), a(5) AND b(3), a(4) AND b(3),
a(3) AND b(3), a(2) AND b(3), a(1) AND b(3), a(0) AND b(3));
    and4 <= (a(7) AND b(4), a(6) AND b(4), a(5) AND b(4), a(4) AND b(4),
a(3) AND b(4), a(2) AND b(4), a(1) AND b(4), a(0) AND b(4));
    and5 <= (a(7) AND b(5), a(6) AND b(5), a(5) AND b(5), a(4) AND b(5),
a(3) AND b(5), a(2) AND b(5), a(1) AND b(5), a(0) AND b(5));
    and6 <= (a(7) AND b(6), a(6) AND b(6), a(5) AND b(6), a(4) AND b(6),
a(3) AND b(6), a(2) AND b(6), a(1) AND b(6), a(0) AND b(6));
    and7 <= (a(7) AND b(7), a(6) AND b(7), a(5) AND b(7), a(4) AND b(7),
a(3) AND b(7), a(2) AND b(7), a(1) AND b(7), a(0) AND b(7));

    -- output for bit 0
    c(0) <= and0(0);
    -- output for bit 1
    HA0 : half_adder port map (a_ha => and0(1), b_ha => and1(0), sum_ha =>
c(1), carry => adder0_c(0));
    -- output for bit 2
    FA0 : full_adder port map (a_fa => and0(2), b_fa => and1(1), carry_in =>
adder0_c(0), sum_fa => adder0_s(1), carry_out => adder0_c(1));
    HA1 : half_adder port map (a_ha => and2(0), b_ha => adder0_s(1), sum_ha
=> c(2), carry => adder1_c(0));
    -- output for bit 3
    FA1 : full_adder port map (a_fa => and0(3), b_fa => and1(2), carry_in =>
adder0_c(1), sum_fa => adder0_s(2), carry_out => adder0_c(2));
    FA2 : full_adder port map (a_fa => and2(1), b_fa => adder0_s(2),
carry_in => adder1_c(0), sum_fa => adder1_s(1), carry_out => adder1_c(1));
    HA2 : half_adder port map (a_ha => and3(0), b_ha => adder1_s(1), sum_ha
=> c(3), carry => adder2_c(0));
    -- output for bit 4
    FA3 : full_adder port map (a_fa => and0(4), b_fa => and1(3), carry_in =>
adder0_c(2), sum_fa => adder0_s(3), carry_out => adder0_c(3));
    FA4 : full_adder port map (a_fa => and2(2), b_fa => adder0_s(3),
carry_in => adder1_c(1), sum_fa => adder1_s(2), carry_out => adder1_c(2));
    FA5 : full_adder port map (a_fa => and3(1), b_fa => adder1_s(2),
carry_in => adder2_c(0), sum_fa => adder2_s(1), carry_out => adder2_c(1));
    HA3 : half_adder port map (a_ha => and4(0), b_ha => adder2_s(1), sum_ha
=> c(4), carry => adder3_c(0));

```

```

-- output for bit 5
FA6 : full_adder port map (a_fa => and0(5), b_fa => and1(4), carry_in =>
adder0_c(3), sum_fa => adder0_s(4), carry_out => adder0_c(4));
FA7 : full_adder port map (a_fa => and2(3), b_fa => adder0_s(4),
carry_in => adder1_c(2), sum_fa => adder1_s(3), carry_out => adder1_c(3));
FA8 : full_adder port map (a_fa => and3(2), b_fa => adder1_s(3),
carry_in => adder2_c(1), sum_fa => adder2_s(2), carry_out => adder2_c(2));
FA9 : full_adder port map (a_fa => and4(1), b_fa => adder2_s(2),
carry_in => adder3_c(0), sum_fa => adder3_s(1), carry_out => adder3_c(1));
HA4 : half_adder port map (a_ha => and5(0), b_ha => adder3_s(1), sum_ha
=> c(5), carry => adder4_c(0));
-- output for bit 6
FA10 : full_adder port map (a_fa => and0(6), b_fa => and1(5), carry_in
=> adder0_c(4), sum_fa => adder0_s(5), carry_out => adder0_c(5));
FA11 : full_adder port map (a_fa => and2(4), b_fa => adder0_s(5),
carry_in => adder1_c(3), sum_fa => adder1_s(4), carry_out => adder1_c(4));
FA12 : full_adder port map (a_fa => and3(3), b_fa => adder1_s(4),
carry_in => adder2_c(2), sum_fa => adder2_s(3), carry_out => adder2_c(3));
FA13 : full_adder port map (a_fa => and4(2), b_fa => adder2_s(3),
carry_in => adder3_c(1), sum_fa => adder3_s(2), carry_out => adder3_c(2));
FA14 : full_adder port map (a_fa => and5(1), b_fa => adder3_s(2),
carry_in => adder4_c(0), sum_fa => adder4_s(1), carry_out => adder4_c(1));
HA5 : half_adder port map (a_ha => and6(0), b_ha => adder4_s(1), sum_ha
=> c(6), carry => adder5_c(0));
-- output for bit 7
FA15 : full_adder port map (a_fa => and0(7), b_fa => and1(6), carry_in
=> adder0_c(5), sum_fa => adder0_s(6), carry_out => adder0_c(6));
FA16 : full_adder port map (a_fa => and2(5), b_fa => adder0_s(6),
carry_in => adder1_c(4), sum_fa => adder1_s(5), carry_out => adder1_c(5));
FA17 : full_adder port map (a_fa => and3(4), b_fa => adder1_s(5),
carry_in => adder2_c(3), sum_fa => adder2_s(4), carry_out => adder2_c(4));
FA18 : full_adder port map (a_fa => and4(3), b_fa => adder2_s(4),
carry_in => adder3_c(2), sum_fa => adder3_s(3), carry_out => adder3_c(3));
FA19 : full_adder port map (a_fa => and5(2), b_fa => adder3_s(3),
carry_in => adder4_c(1), sum_fa => adder4_s(2), carry_out => adder4_c(2));
FA20 : full_adder port map (a_fa => and6(1), b_fa => adder4_s(2),
carry_in => adder5_c(0), sum_fa => adder5_s(1), carry_out => adder5_c(1));
HA6 : half_adder port map (a_ha => and7(0), b_ha => adder5_s(1), sum_ha
=> c(7), carry => adder6_c(0));

-- output for bit 8
HA7 : half_adder port map (a_ha => and1(7), b_ha => adder0_c(6), sum_ha
=> adder0_s(7), carry => adder0_c(7));
FA21 : full_adder port map (a_fa => and2(6), b_fa => adder0_s(7),
carry_in => adder1_c(5), sum_fa => adder1_s(6), carry_out => adder1_c(6));
FA22 : full_adder port map (a_fa => and3(5), b_fa => adder1_s(6),
carry_in => adder2_c(4), sum_fa => adder2_s(5), carry_out => adder2_c(5));
FA23 : full_adder port map (a_fa => and4(4), b_fa => adder2_s(5),
carry_in => adder3_c(3), sum_fa => adder3_s(4), carry_out => adder3_c(4));

```

```

FA24 : full_adder port map (a_fa => and5(3), b_fa => adder3_s(4),
carry_in => adder4_c(2), sum_fa => adder4_s(3), carry_out => adder4_c(3));
FA25 : full_adder port map (a_fa => and6(2), b_fa => adder4_s(3),
carry_in => adder5_c(1), sum_fa => adder5_s(2), carry_out => adder5_c(2));
FA26 : full_adder port map (a_fa => and7(1), b_fa => adder5_s(2),
carry_in => adder6_c(0), sum_fa => c(8), carry_out => adder6_c(1));
-- output for bit 9
FA27 : full_adder port map (a_fa => and2(7), b_fa => adder0_c(7),
carry_in => adder1_c(6), sum_fa => adder1_s(7), carry_out => adder1_c(7));
FA28 : full_adder port map (a_fa => and3(6), b_fa => adder1_s(7),
carry_in => adder2_c(5), sum_fa => adder2_s(6), carry_out => adder2_c(6));
FA29 : full_adder port map (a_fa => and4(5), b_fa => adder2_s(6),
carry_in => adder3_c(4), sum_fa => adder3_s(5), carry_out => adder3_c(5));
FA30 : full_adder port map (a_fa => and5(4), b_fa => adder3_s(5),
carry_in => adder4_c(3), sum_fa => adder4_s(4), carry_out => adder4_c(4));
FA31 : full_adder port map (a_fa => and6(3), b_fa => adder4_s(4),
carry_in => adder5_c(2), sum_fa => adder5_s(3), carry_out => adder5_c(3));
FA32 : full_adder port map (a_fa => and7(2), b_fa => adder5_s(3),
carry_in => adder6_c(1), sum_fa => c(9), carry_out => adder6_c(2));
-- output for bit 10
FA33 : full_adder port map (a_fa => and3(7), b_fa => adder1_c(7),
carry_in => adder2_c(6), sum_fa => adder2_s(7), carry_out => adder2_c(7));
FA34 : full_adder port map (a_fa => and4(6), b_fa => adder2_s(7),
carry_in => adder3_c(5), sum_fa => adder3_s(6), carry_out => adder3_c(6));
FA35 : full_adder port map (a_fa => and5(5), b_fa => adder3_s(6),
carry_in => adder4_c(4), sum_fa => adder4_s(5), carry_out => adder4_c(5));
FA36 : full_adder port map (a_fa => and6(4), b_fa => adder4_s(5),
carry_in => adder5_c(3), sum_fa => adder5_s(4), carry_out => adder5_c(4));
FA37 : full_adder port map (a_fa => and7(3), b_fa => adder5_s(4),
carry_in => adder6_c(2), sum_fa => c(10), carry_out => adder6_c(3));
-- output for bit 11
FA38 : full_adder port map (a_fa => and4(7), b_fa => adder2_c(7),
carry_in => adder3_c(6), sum_fa => adder3_s(7), carry_out => adder3_c(7));
FA39 : full_adder port map (a_fa => and5(6), b_fa => adder3_s(7),
carry_in => adder4_c(5), sum_fa => adder4_s(6), carry_out => adder4_c(6));
FA40 : full_adder port map (a_fa => and6(5), b_fa => adder4_s(6),
carry_in => adder5_c(4), sum_fa => adder5_s(5), carry_out => adder5_c(5));
FA41 : full_adder port map (a_fa => and7(4), b_fa => adder5_s(5),
carry_in => adder6_c(3), sum_fa => c(11), carry_out => adder6_c(4));
-- output for bit 12
FA42 : full_adder port map (a_fa => and5(7), b_fa => adder3_c(7),
carry_in => adder4_c(6), sum_fa => adder4_s(7), carry_out => adder4_c(7));
FA43 : full_adder port map (a_fa => and6(6), b_fa => adder4_s(7),
carry_in => adder5_c(5), sum_fa => adder5_s(6), carry_out => adder5_c(6));
FA44 : full_adder port map (a_fa => and7(5), b_fa => adder5_s(6),
carry_in => adder6_c(4), sum_fa => c(12), carry_out => adder6_c(5));
-- output for bit 13
FA45 : full_adder port map (a_fa => and6(7), b_fa => adder4_c(7),
carry_in => adder5_c(6), sum_fa => adder5_s(7), carry_out => adder5_c(7));

```

```

    FA46 : full_adder port map (a_fa => and7(6), b_fa => adder5_s(7),
carry_in => adder6_c(5), sum_fa => c(13), carry_out => adder6_c(6));
    -- output for bit 14 and bit 15
    FA47 : full_adder port map (a_fa => and7(7), b_fa => adder5_c(7),
carry_in => adder6_c(6), sum_fa => c(14), carry_out => c(15));

end b8;

```

## Annex B. Wallace Multiplier VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity wallace_8bit is
    Port (a : in STD_LOGIC_VECTOR(7 downto 0);
          b : in STD_LOGIC_VECTOR(7 downto 0);
          c : out STD_LOGIC_VECTOR(15 downto 0));
end wallace_8bit;

architecture w8 of wallace_8bit is

    -- Generated by a python script I wrote, each signal vector (and{n})
    represents a horizontal array of and gates.
    signal and0, and1, and2, and3, and4, and5, and6, and7 :
STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
    -- Signals to take the sums and carry of the adders
    signal c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13,
c14, c15, c16, c17, c18, c19, c20, c21, c22, c23, c24, c25, c26, c27, c28,
c29, c30, c31, c32, c33, c34, c35, c36, c37, c38, c39, c40, c41, c42, c43,
c44, c45, c46, c47, c48, c49, c50, c51, c52, c53, c54, c55, c56, c57, c58,
c59, c60, c61, c62, c63, c64, c65, c66, c67 : STD_LOGIC := '0';
    signal s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13,
s14, s15, s16, s17, s18, s19, s20, s21, s22, s23, s24, s25, s26, s27, s28,
s29, s30, s31, s32, s33, s34, s35, s36, s37, s38, s39, s40, s41, s42, s43,
s44, s45, s46, s47, s48, s49, s50, s51, s52, s53, s54, s55, s56, s57, s58,
s59, s60, s61, s62, s63, s64, s65, s66, s67 : STD_LOGIC := '0';

    component
half_adder is
    port (a_ha : in STD_LOGIC;
          b_ha : in STD_LOGIC;
          sum_ha : out STD_LOGIC;
          carry : out STD_LOGIC);
end component;

    component

```

```

full_adder is
port (a_fa : in STD_LOGIC;
      b_fa : in STD_LOGIC;
      carry_in : in STD_LOGIC;
      sum_fa : out STD_LOGIC;
      carry_out : out STD_LOGIC);
end component;

begin

    and0 <= (a(7) AND b(0), a(6) AND b(0), a(5) AND b(0), a(4) AND b(0),
a(3) AND b(0), a(2) AND b(0), a(1) AND b(0), a(0) AND b(0));
    and1 <= (a(7) AND b(1), a(6) AND b(1), a(5) AND b(1), a(4) AND b(1),
a(3) AND b(1), a(2) AND b(1), a(1) AND b(1), a(0) AND b(1));
    and2 <= (a(7) AND b(2), a(6) AND b(2), a(5) AND b(2), a(4) AND b(2),
a(3) AND b(2), a(2) AND b(2), a(1) AND b(2), a(0) AND b(2));
    and3 <= (a(7) AND b(3), a(6) AND b(3), a(5) AND b(3), a(4) AND b(3),
a(3) AND b(3), a(2) AND b(3), a(1) AND b(3), a(0) AND b(3));
    and4 <= (a(7) AND b(4), a(6) AND b(4), a(5) AND b(4), a(4) AND b(4),
a(3) AND b(4), a(2) AND b(4), a(1) AND b(4), a(0) AND b(4));
    and5 <= (a(7) AND b(5), a(6) AND b(5), a(5) AND b(5), a(4) AND b(5),
a(3) AND b(5), a(2) AND b(5), a(1) AND b(5), a(0) AND b(5));
    and6 <= (a(7) AND b(6), a(6) AND b(6), a(5) AND b(6), a(4) AND b(6),
a(3) AND b(6), a(2) AND b(6), a(1) AND b(6), a(0) AND b(6));
    and7 <= (a(7) AND b(7), a(6) AND b(7), a(5) AND b(7), a(4) AND b(7),
a(3) AND b(7), a(2) AND b(7), a(1) AND b(7), a(0) AND b(7));

    -- output for bit 0

    c(0) <= and0(0);

    -- First stage

    HA0 : half_adder port map (a_ha => and1(0), b_ha => and0(1), sum_ha => s0,
carry => c0);
    FA0 : full_adder port map (a_fa => and0(2), b_fa => and2(0), carry_in =>
and1(1), sum_fa => s1, carry_out => c1);
    FA1 : full_adder port map (a_fa => and0(3), b_fa => and1(2), carry_in =>
and2(1), sum_fa => s2, carry_out => c2);
    FA2 : full_adder port map (a_fa => and0(4), b_fa => and1(3), carry_in =>
and2(2), sum_fa => s3, carry_out => c3);

    HA1 : half_adder port map (a_ha => and3(1), b_ha => and4(0), sum_ha => s4,
carry => c4);
    FA3 : full_adder port map (a_fa => and0(5), b_fa => and1(4), carry_in =>
and2(3), sum_fa => s5, carry_out => c5);
    FA4 : full_adder port map (a_fa => and3(2), b_fa => and4(1), carry_in =>
and5(0), sum_fa => s6, carry_out => c6);
    FA5 : full_adder port map (a_fa => and0(6), b_fa => and1(5), carry_in =>
and2(4), sum_fa => s7, carry_out => c7);

```

```

FA6 : full_adder port map (a_fa => and3(3), b_fa => and4(2), carry_in =>
and5(1), sum_fa => s8, carry_out => c8);
FA7 : full_adder port map (a_fa => and0(7), b_fa => and1(6), carry_in =>
and2(5), sum_fa => s9, carry_out => c9);
FA8 : full_adder port map (a_fa => and3(4), b_fa => and4(3), carry_in =>
and5(2), sum_fa => s10, carry_out => c10);

```

```

HA2 : half_adder port map (a_ha => and6(1), b_ha => and7(0), sum_ha =>
s11, carry => c11);
FA9 : full_adder port map (a_fa => and1(7), b_fa => and2(6), carry_in =>
and3(5), sum_fa => s12, carry_out => c12);
FA10 : full_adder port map (a_fa => and4(4), b_fa => and5(3), carry_in =>
and6(2), sum_fa => s13, carry_out => c13);
FA11 : full_adder port map (a_fa => and2(7), b_fa => and3(6), carry_in =>
and4(5), sum_fa => s14, carry_out => c14);
FA12 : full_adder port map (a_fa => and5(4), b_fa => and6(3), carry_in =>
and7(2), sum_fa => s15, carry_out => c15);
FA13 : full_adder port map (a_fa => and3(7), b_fa => and4(6), carry_in =>
and5(5), sum_fa => s16, carry_out => c16);

```

```

HA3 : half_adder port map (a_ha => and6(4), b_ha => and7(3), sum_ha =>
s17, carry => c17);
FA14 : full_adder port map (a_fa => and4(7), b_fa => and5(6), carry_in =>
and6(5), sum_fa => s18, carry_out => c18);
FA15 : full_adder port map (a_fa => and5(7), b_fa => and6(6), carry_in =>
and7(5), sum_fa => s19, carry_out => c19);

```

```

HA4 : half_adder port map (a_ha => and6(7), b_ha => and7(6), sum_ha =>
s20, carry => c20);

```

-- Second stage

```

HA5 : half_adder port map (a_ha => s1, b_ha => c0, sum_ha => s21, carry
=> c21);
FA16 : full_adder port map (a_fa => and3(0), b_fa => c1, carry_in => s2,
sum_fa => s22, carry_out => c22);
FA17 : full_adder port map (a_fa => s3, b_fa => s4, carry_in => c2, sum_fa
=> s23, carry_out => c23);
FA18 : full_adder port map (a_fa => s5, b_fa => s6, carry_in => c3, sum_fa
=> s24, carry_out => c24);
FA19 : full_adder port map (a_fa => s7, b_fa => s8, carry_in => and6(0),
sum_fa => s25, carry_out => c25);

```

```

HA6 : half_adder port map (a_ha => c5, b_ha => c6, sum_ha => s26, carry
=> c26);
FA20 : full_adder port map (a_fa => s9, b_fa => s10, carry_in => s11,
sum_fa => s27, carry_out => c27);

```

```

HA7 : half_adder port map (a_ha => c7, b_ha => c8, sum_ha => s28, carry
=> c28);

```

```

FA21 : full_adder port map (a_fa => s12, b_fa => s13, carry_in => and7(1),
sum_fa => s29, carry_out => c29);
FA22 : full_adder port map (a_fa => c9, b_fa => c10, carry_in => c11,
sum_fa => s30, carry_out => c30);
FA23 : full_adder port map (a_fa => s14, b_fa => s15, carry_in => c12,
sum_fa => s31, carry_out => c31);
FA24 : full_adder port map (a_fa => s16, b_fa => s17, carry_in => c14,
sum_fa => s32, carry_out => c32);
FA25 : full_adder port map (a_fa => s18, b_fa => c16, carry_in => c17,
sum_fa => s33, carry_out => c33);

```

```

HA8  : half_adder port map (a_ha => s19, b_ha => c18, sum_ha => s34, carry
=> c34);
HA9  : half_adder port map (a_ha => s20, b_ha => c19, sum_ha => s35, carry
=> c35);

```

-- Third stage

```

HA10 : half_adder port map (a_ha => s22, b_ha => c21, sum_ha => s36,
carry => c36);
HA11 : half_adder port map (a_ha => c22, b_ha => s23, sum_ha => s37,
carry => c37);

```

```

FA26 : full_adder port map (a_fa => c23, b_fa => s24, carry_in => c4,
sum_fa => s38, carry_out => c38);
FA27 : full_adder port map (a_fa => c24, b_fa => s25, carry_in => s26,
sum_fa => s39, carry_out => c39);
FA28 : full_adder port map (a_fa => c25, b_fa => c26, carry_in => s27,
sum_fa => s40, carry_out => c40);
FA29 : full_adder port map (a_fa => c27, b_fa => c28, carry_in => s29,
sum_fa => s41, carry_out => c41);
FA30 : full_adder port map (a_fa => c29, b_fa => c30, carry_in => s31,
sum_fa => s42, carry_out => c42);
FA31 : full_adder port map (a_fa => c31, b_fa => c15, carry_in => s32,
sum_fa => s43, carry_out => c43);
FA32 : full_adder port map (a_fa => c32, b_fa => and7(4), carry_in => s33,
sum_fa => s44, carry_out => c44);

```

```

HA12 : half_adder port map (a_ha => s34, b_ha => c33, sum_ha => s45, carry
=> c45);
HA13 : half_adder port map (a_ha => c34, b_ha => s35, sum_ha => s46, carry
=> c46);

```

```

FA33 : full_adder port map (a_fa => and7(7), b_fa => c20, carry_in => c35,
sum_fa => s47, carry_out => c47);

```

-- Fourth stage

```

HA14 : half_adder port map (a_ha => c36, b_ha => s37, sum_ha => s48, carry
=> c48);

```

```

FA34 : full_adder port map (a_fa => s38, b_fa => c37, carry_in => c48,
sum_fa => s49, carry_out => c49);
FA35 : full_adder port map (a_fa => s39, b_fa => c38, carry_in => c49,
sum_fa => s50, carry_out => c50);
FA36 : full_adder port map (a_fa => c39, b_fa => s40, carry_in => s28,
sum_fa => s51, carry_out => c51);
FA37 : full_adder port map (a_fa => c40, b_fa => s30, carry_in => s41,
sum_fa => s52, carry_out => c52);
FA38 : full_adder port map (a_fa => c13, b_fa => c41, carry_in => s42,
sum_fa => s53, carry_out => c53);
FA39 : full_adder port map (a_fa => s43, b_fa => c42, carry_in => c53,
sum_fa => s54, carry_out => c54);
FA40 : full_adder port map (a_fa => c43, b_fa => s44, carry_in => c54,
sum_fa => s55, carry_out => c55);
FA41 : full_adder port map (a_fa => s45, b_fa => c44, carry_in => c55,
sum_fa => s56, carry_out => c56);
FA42 : full_adder port map (a_fa => c45, b_fa => s46, carry_in => c56,
sum_fa => s57, carry_out => c57);
FA43 : full_adder port map (a_fa => s47, b_fa => c46, carry_in => c57,
sum_fa => s58, carry_out => c58);

```

-- Fifth and final stage

```

HA15 : half_adder port map (a_ha => c50, b_ha => s51, sum_ha => s59, carry
=> c59);
FA44 : full_adder port map (a_fa => c51, b_fa => s52, carry_in => c59,
sum_fa => s60, carry_out => c60);
FA45 : full_adder port map (a_fa => c52, b_fa => s53, carry_in => c60,
sum_fa => s61, carry_out => c61);

```

```

HA16 : half_adder port map (a_ha => s54, b_ha => c61, sum_ha => s62, carry
=> c62);
HA17 : half_adder port map (a_ha => s55, b_ha => c62, sum_ha => s63, carry
=> c63);
HA18 : half_adder port map (a_ha => s56, b_ha => c63, sum_ha => s64, carry
=> c64);
HA19 : half_adder port map (a_ha => s57, b_ha => c64, sum_ha => s65, carry
=> c65);
HA20 : half_adder port map (a_ha => s58, b_ha => c65, sum_ha => s66, carry
=> c66);

```

```

FA46 : full_adder port map (a_fa => c47, b_fa => c58, carry_in => c66,
sum_fa => s67, carry_out => c67);

```

```

c(1) <= s0;
c(2) <= s21;
c(3) <= s36;
c(4) <= s48;
c(5) <= s49;
c(6) <= s50;

```

```

c(7) <= s59;
c(8) <= s60;
c(9) <= s61;
c(10) <= s62;
c(11) <= s63;
c(12) <= s64;
c(13) <= s65;
c(14) <= s66;
c(15) <= s67 or c67;

```

```
end w8;
```

## Annex C. Half adder VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity half_adder is
    Port ( a_ha : in STD_LOGIC;
           b_ha : in STD_LOGIC;
           sum_ha : out STD_LOGIC;
           carry : out STD_LOGIC);
end half_adder;

```

```

architecture ha_a of half_adder is
begin
    sum_ha <= a_ha XOR b_ha;
    carry <= a_ha AND b_ha;
end ha_a;

```

## Annex D. Full adder VHDL

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder is
    Port ( a_fa : in STD_LOGIC;
           b_fa : in STD_LOGIC;
           carry_in : in STD_LOGIC;
           sum_fa : out STD_LOGIC;
           carry_out : out STD_LOGIC);
end full_adder;

architecture fa_a of full_adder is
begin
    sum_fa <= a_fa XOR b_fa XOR carry_in;
    carry_out <= (a_fa AND b_fa) OR (b_fa AND carry_in) OR (a_fa AND
carry_in);
end fa_a;

```

## Annex E. Python script to create signals

MakeAndarray.py

```

def andGates(BITS, signalName, TABSPACE):
    vhdl = ""

    vhdl += "%ssignal " % (' ' * TABSPACE)
    for i in range(BITS):
        vhdl += "%s%s, " % (signalName, i)

    vhdl += ': STD_LOGIC_VECTOR(%s downto 0) := (others => \'0\');\n' %
(BITS - 1)

    for i in range(BITS):
        vhdl += "%s%s%s <= (" % (' ' * TABSPACE, signalName, i)
        for j in range(BITS):
            vhdl += "a(%s) AND b(%s), " % (BITS - 1 - j, i)
        vhdl = vhdl[:-2]
        vhdl += ");\n"

    vhdl = vhdl[:-1]

    return vhdl

def halfAdder(count, a, a_m, b, b_m, s, s_m, c, c_m):
    HA = "HA%s : half_adder port map (%s => %s, %s => %s, %s => %s, %s =>
%s);" % (count, a, a_m, b, b_m, s, s_m, c, c_m)
    return HA

if __name__ == "__main__":

    print(halfAdder(0, 'a_ha', r'and0(%s)', r'b_ha', r'and0(%s)',
r'sum_ha', r'adder%s_s(%s)', r'carry_ha', r'adder%s_c(%s)'))

    ret = andGates(8, "and", 2)
    print(ret)

MakeSignals.py

raw = r'%s%s, '

o = ''
p = ''
for i in range(68):
    o+= raw % ('c', i)
for i in range(68):
    p+= raw % ('s', i)

print('signal ' + o[:-2] + " : STD_LOGIC := '0';")
print('signal ' + p[:-2] + " : STD_LOGIC := '0';")

```