

Daniel Davies E2 Assignment

by Daniel Davies

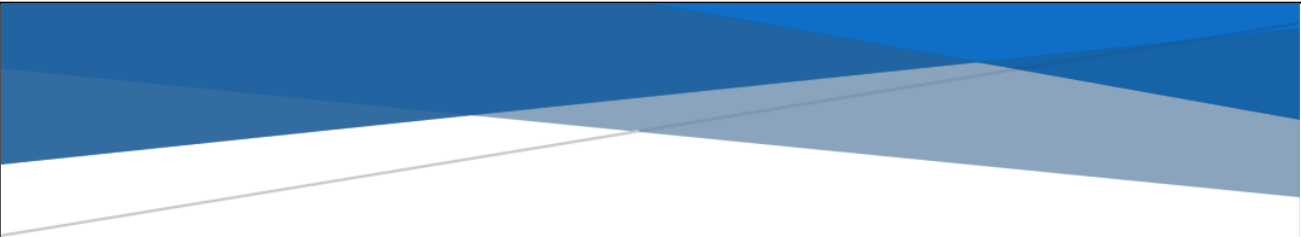
Submission date: 16-May-2023 08:06AM (UTC+0100)

Submission ID: 206381339

File name: 25596_Daniel_Davies_Daniel_Davies_E2_Assignment_497727_2019523285.docx (1.91M)

Word count: 2348

Character count: 11355



ELECTRONICS 2 – ACDA6001

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

Daniel Davies
2013713
27/04/2023

Contents

Introduction	2
Background Research	3
Information and Analysis.....	4
Brauns	4
Wallace Tree	4
Circuit Designs	7
Brauns	7
Wallace Tree	9
System Operation Flow Charts.....	11
Braun.....	11
Wallace Tree	12
FPGA Diagrams.....	13
Brauns	13
Wallace Tree	14
VHDL Codes	15
Brauns	15
Wallace Tree	16
Testing	25
Brauns	25
Wallace Tree	25
Final Comparisons.....	27
Conclusions and Recommendations	27
References.....	28

Introduction



In this assignment, I will be comparing the creation and operation of two different multipliers using VHDL code. Both multipliers are used to multiply two 8 bit numbers into a 16 bit output, the methods used are Brauns multiplier and Wallace Tree multiplier.

The report will show background research done, information and analysis of each multiplier, the circuit designs, the code used in VHDL and flowchart of the operation of each multiplier, as well as the FPGA diagrams. They will be compared at each step, followed by the testing and operation results, with a final comparison at the end along with recommendations for the future.

Background Research

A  multiplier is a type of electronic circuit that can be used to multiply two binary numbers, they work in a similar way to that of standard long multiplication in mathematics.

An example of this is below:

```
  123
x 567
=====
 861  (this is 123 x 7)
 738  (this is 123 x 6, shifted one position to the left)
+ 615  (this is 123 x 5, shifted two positions to the left)
=====
69741
```

A binary multiplier works in the same way as seen below:

```
  1001  (this is binary for decimal 9)
x 1110  (this is binary for decimal 14)
=====
 0000  (this is 1001 x 0)
 1001  (this is 1001 x 1, shifted one position to the left)
 1001  (this is 1001 x 1, shifted two positions to the left)
+ 1001  (this is 1001 x 1, shifted three positions to the left)
=====
01111110  (this is binary for decimal 126)
```

When using two unsigned integers of n bits in the multiplication, we can produce an output of $n \times 2$ bits by performing n one bit multiplications then adding the outputs together.

For example, in this report, 2 integers of 8 bits are multiplied together to get a 16 bit binary output.

Information and Analysis

Brauns

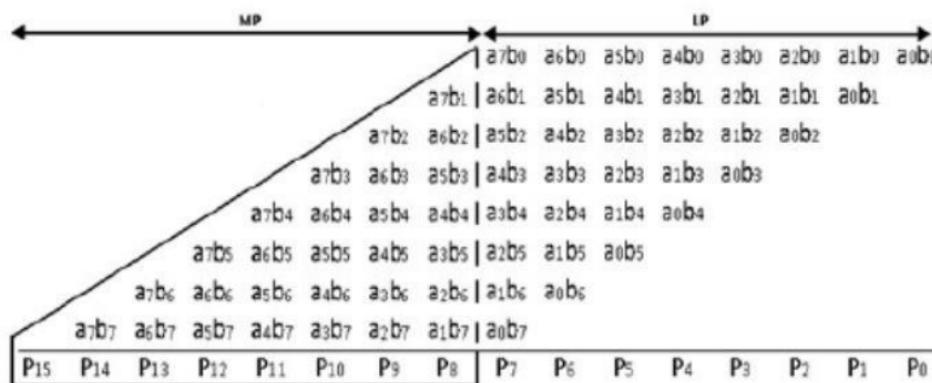
The Brauns multiplier was developed by Braun Edward Louis in 1963.

A Brauns multiplier is a type of parallel array multiplier. It is used for achieving high execution speed demands. It consists of a AND gates and adders in an iterative structure that has no requirement for logic registers.

The Brauns multiplier can only be designed for use with unsigned integers. It is used in DSP processors and also used in image processing.

In a Brauns multiplier, each full adder has 3 Inputs, A,B and Cin. A and B being the relevant input bits and Cin being the Carry input from the previous adder's output. The full adders have 2 outputs, Sum and Cout. Sum being the result of the adder and Cout being the carry that becomes an input for the next adder.

The half adders have 2 inputs, A and B, both being the relevant bit inputs, the outputs are Sum and Carry, which work in the same way as the full adders.



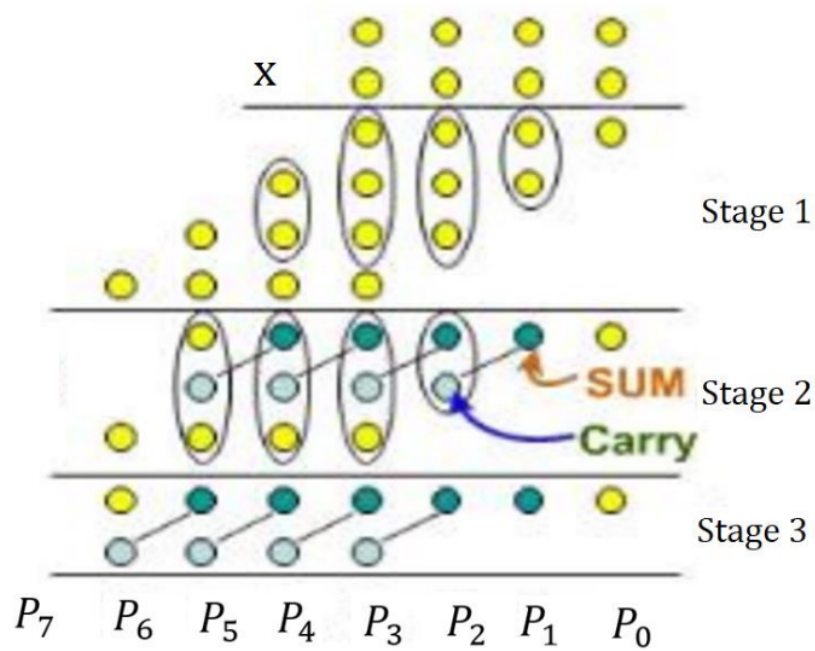
The above image shows the structure of how a Brauns multiplier works and what inputs are required to produce a given output bit. For example, for P0 to equal 1, A0B0 must equal 1. For P3 to equal 1, one of A3B0, A2B1, A1B2 or A0B3 must equal 1.

Wallace Tree

A Wallace Tree multiplier is a type of binary long multiplier, it uses half and full adders to sum two binary numbers in stages until only two numbers are left, a Wallace Tree multiplier will reduce as much as possible at each stage. It was devised by Chris Wallace, an Australian computer scientist in 1964. A Wallace Tree multiplier has three steps to it:

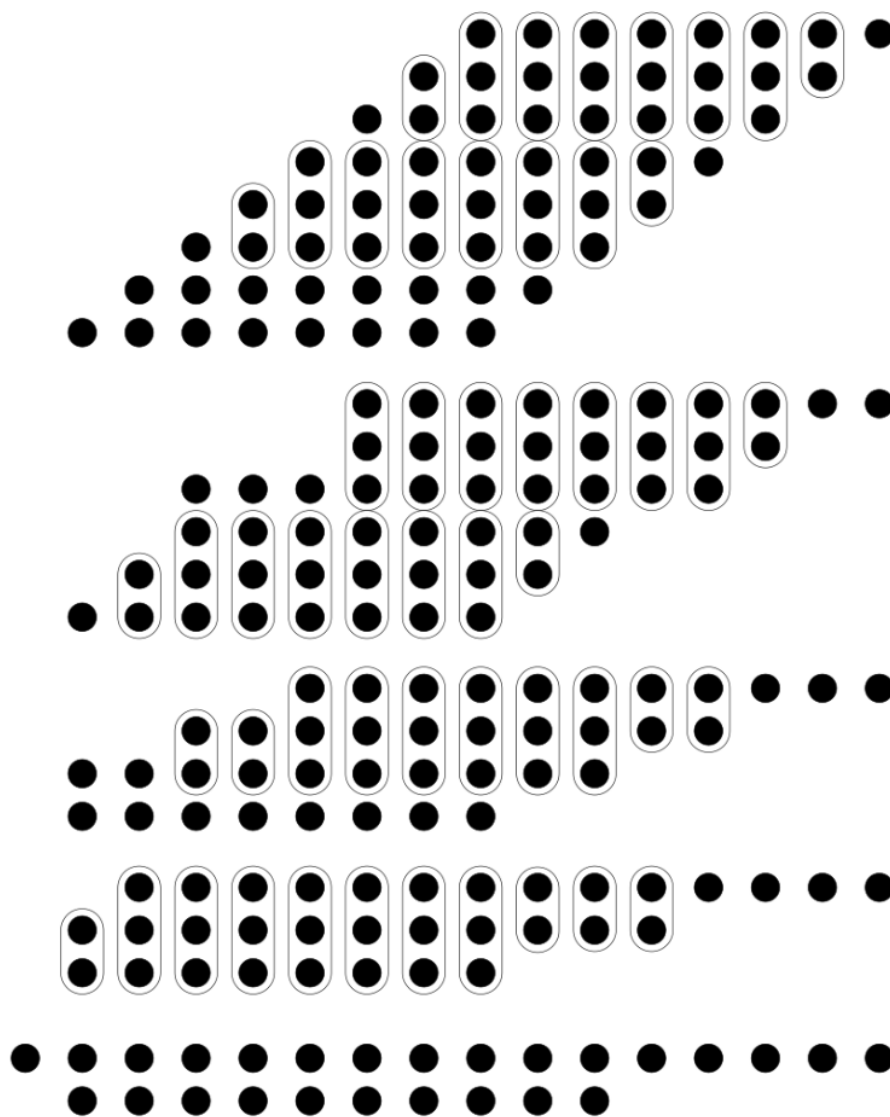
- Multiply each bit of one input by each bit of the other.
- Reduce the number of partial products down to two using half and full adders.
- The final product is calculated by the weighted sum of all of the products of its factors.

						$A_3 A_2 A_1 A_0$	
					X		
						$B_3 B_2 B_1 B_0$	
				$A_3 B_0$	$A_2 B_0$	$A_1 B_0$	$A_0 B_0$
			$A_3 B_1$	$A_2 B_1$	$A_1 B_1$	$A_0 B_1$	
		$A_3 B_2$	$A_2 B_2$	$A_1 B_2$	$A_0 B_2$		
	$A_3 B_3$	$A_2 B_3$	$A_1 B_3$	$A_0 B_3$	∨	∨	∨
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0



5

The Wallace Tree multiplier uses the previous stages as inputs for its next stage. Half adders have two inputs and two outputs, the inputs being for example A_0B_1 and A_1B_0 , with each of those being equal to either 1 or 0. The outputs are the Sum and Carry. The full adders have 3 inputs and 2 outputs, the inputs being dependant on the stage of the multiplier. At the first stage, the inputs are all binary inputs of the possible outcomes, for example they could be A_2B_0 , A_1B_1 and A_0B_2 , whereas at further stages, the inputs will be one of the possible outcomes, such as A_0B_3 and the other two will be the outputs from the previous connected adders. This again gives a Sum and Carry output which proceeds to be an input for the next stages.



The above dot diagram shows the stages for the Wallace Tree multiplier, the dots connected in threes show full adders, the dots connected in twos show half adders. There are 5 stages to a Wallace Tree multiplier, this ties in with the circuit diagram shown in the latter sections.

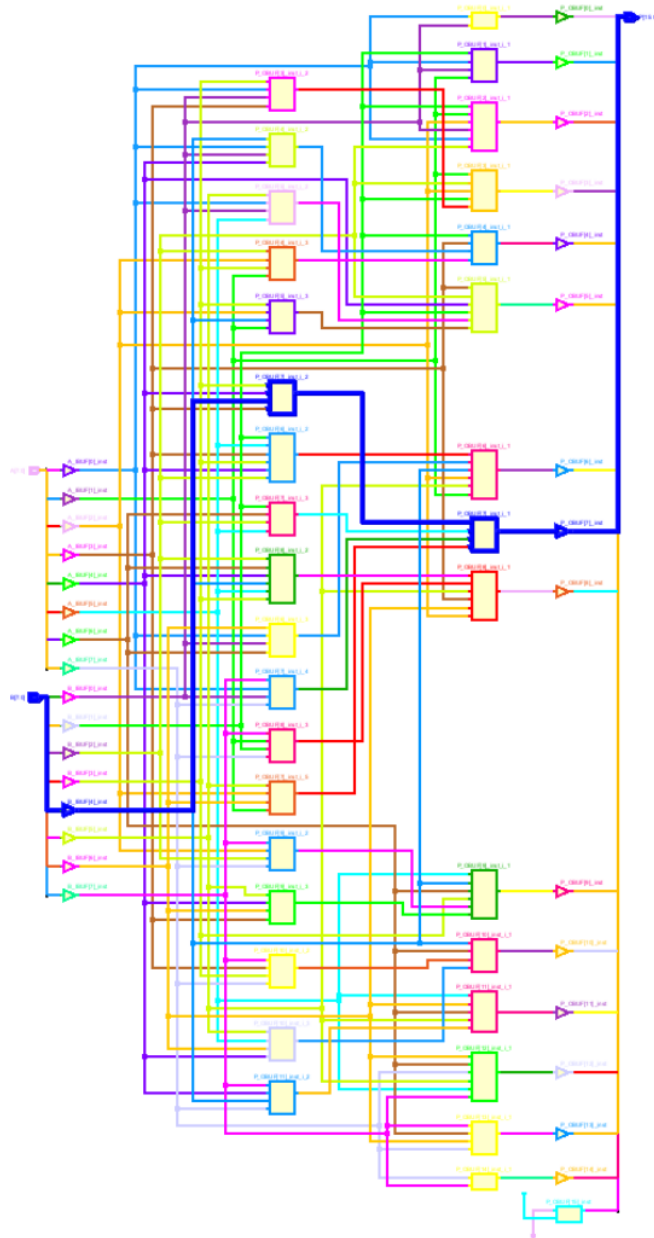
Circuit Designs

6

The below schematics show the critical paths of each multiplier. The critical path is the longest possible route from start to finish, in other words, the section that will cause the longest delay.

7

Brauns

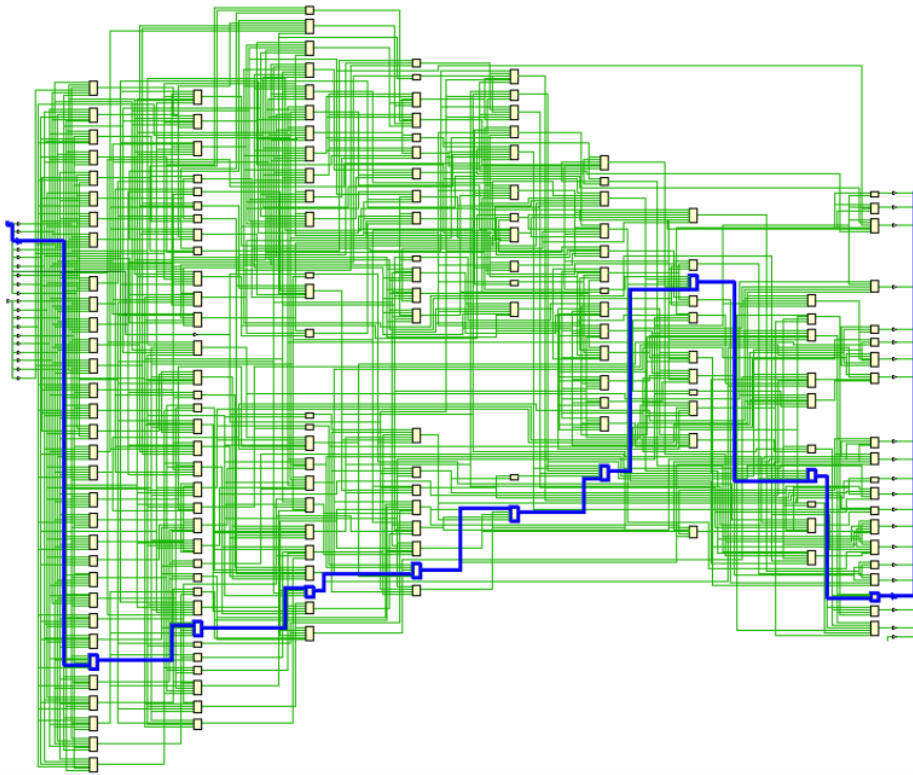


The delays for the Brauns multiplier are shown below. Path 9 is the highlighted critical path.

From Port	To Port	Max Delay	Max Process Corner	Min Delay	Min Process Corner
A[1]	P[8]	7.430	SLOW	3.003	FAST
A[2]	P[2]	6.172	SLOW	2.435	FAST
A[2]	P[3]	6.569	SLOW	2.592	FAST
A[2]	P[4]	7.143	SLOW	2.805	FAST
A[2]	P[5]	6.898	SLOW	2.708	FAST
A[2]	P[6]	6.288	SLOW	2.472	FAST
A[2]	P[7]	6.818	SLOW	2.665	FAST
A[2]	P[8]	6.410	SLOW	2.499	FAST
A[2]	P[9]	7.452	SLOW	3.022	FAST
A[3]	P[3]	6.934	SLOW	2.696	FAST
A[3]	P[4]	6.039	SLOW	2.348	FAST
A[3]	P[5]	6.677	SLOW	2.636	FAST
A[3]	P[6]	7.196	SLOW	2.846	FAST
A[3]	P[7]	7.351	SLOW	2.873	FAST
A[3]	P[8]	6.704	SLOW	2.633	FAST
A[3]	P[9]	6.970	SLOW	2.780	FAST

With the largest delay being 7.452ns.

Wallace Tree



8 This is the critical path of the Wallace tree multiplier.

The delays are below:

op1[2]	result[2]	6.993	SLOW	2.821	FAST
op1[2]	result[3]	7.927	SLOW	3.123	FAST
op1[2]	result[4]	9.224	SLOW	3.011	FAST
op1[2]	result[5]	9.370	SLOW	3.257	FAST
op1[2]	result[6]	9.748	SLOW	2.822	FAST
op1[2]	result[7]	10.690	SLOW	3.198	FAST
op1[2]	result[8]	10.519	SLOW	3.354	FAST
op1[2]	result[9]	10.705	SLOW	3.399	FAST
op1[2]	result[10]	11.480	SLOW	3.494	FAST
op1[2]	result[11]	11.856	SLOW	3.647	FAST
op1[2]	result[12]	11.861	SLOW	3.515	FAST
op1[2]	result[13]	11.945	SLOW	3.541	FAST
op1[2]	result[14]	12.059	SLOW	3.507	FAST
op1[2]	result[15]	12.487	SLOW	3.679	FAST
op1[2]	result[16]	12.867	SLOW	3.935	FAST
op1[2]	result[17]	13.867	SLOW	4.317	FAST
op1[2]	result[18]	13.723	SLOW	4.264	FAST
op1[2]	result[19]	13.681	SLOW	4.281	FAST

As can be seen, path 11 has the largest delay, at 13.867ns.

The largest delay between the two systems is the Wallace tree multiplier at 13.867ns compared to the 7.452ns on Brauns. This is due to the Wallace Tree multiplier having a higher requirement for the number of logic gates used.

To improve the speed of the system, it is possible to implement DSP48 in to the code. This enables pipeline registers for each DSP48 input and two registers for each DSP48 output. This will give maximum performance.

The code design should also be optimised, area optimisations will reduce resource usage of the design and speed optimisations will improve the systems timing so that the system will run at higher frequencies by optimising the critical path.

These can be done by using the following optimisations:

Speed Optimisations

Clock rate pipelining: This is a software option that tuns pipeline registers at faster clock rates when you have an oversampling factor that is larger than 1.

Distributed pipelining: This option retimes registers that have existing delays so that the system can evaluate what pipelines need faster clock rates for the targeted device.

Adaptive pipelining: This inserts pipeline registers at the input, output or both to create efficiently mapped blocks to DSP units. This considers the target device, frequency, multiplier word lengths and VHDL block property settings.

Loop unrolling: This is where a loop is unrolled by instantiating multiple loop body instances generated in the code.

Area Optimisations

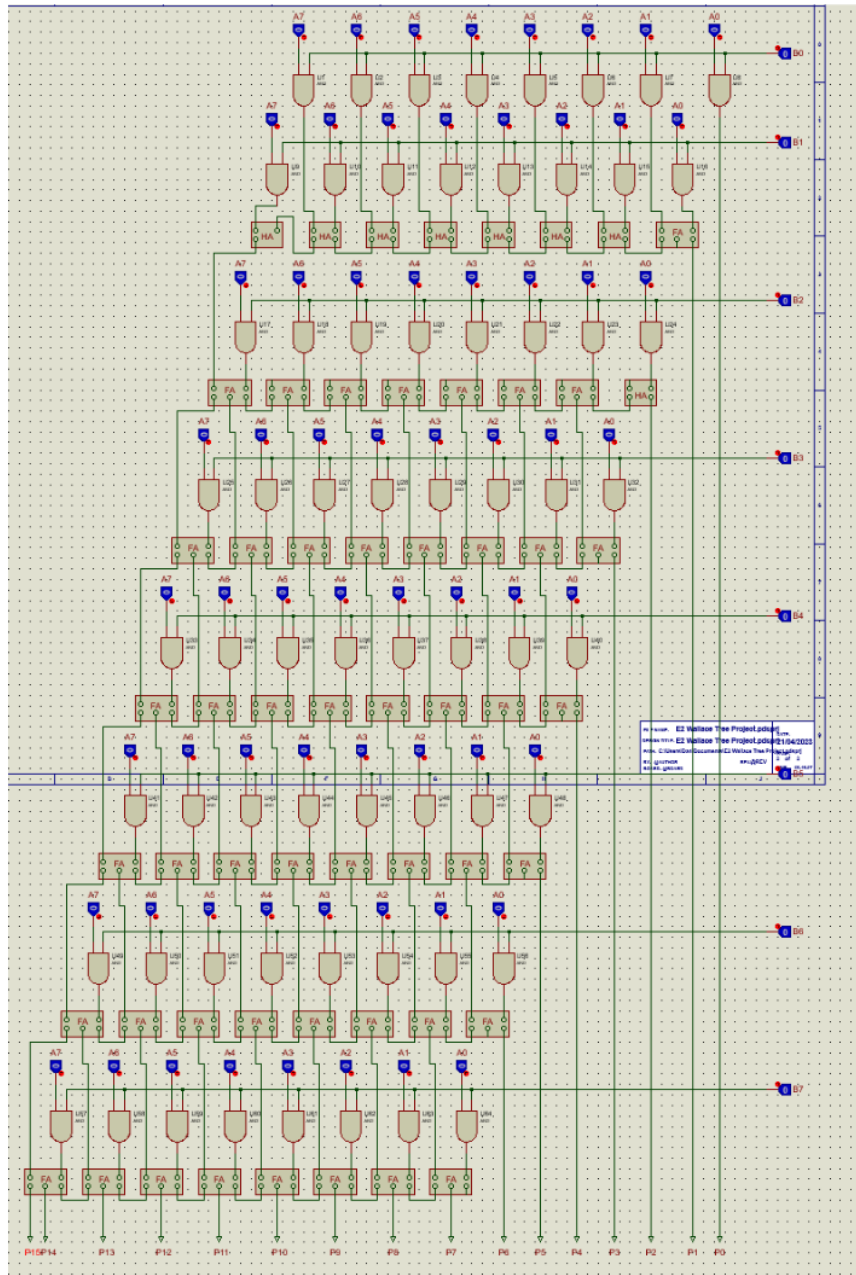
Resource sharing: This is an optimisation that will identify multiple equivalent resources and replace them with a single resource.

Streaming: This optimisation is used to split the vector data path into smaller data paths based on a streaming factor.

Loop streaming: This is where a loop is streamed by instantiating the loop body once and then reusing that instance for each iteration of the loop.

System Operation Flow Charts

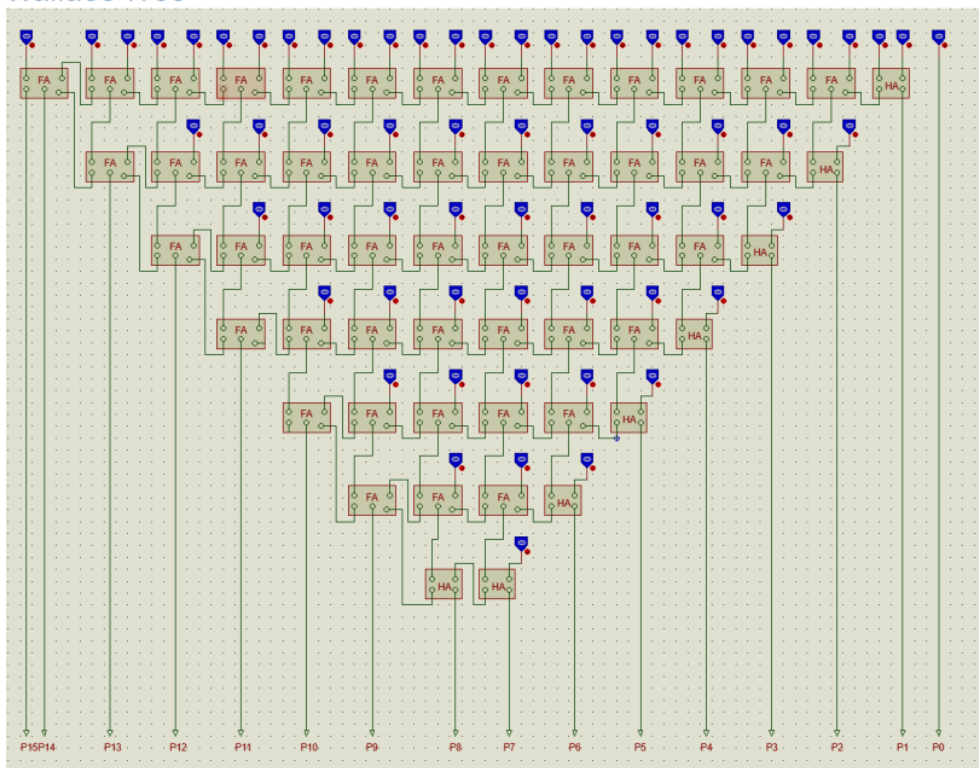
Braun



The operational flow chart follows the principle of long multiplication, from P0 to P15. Each of the outputs is added together to give the final 16 bit binary output.

As can be seen, the system is made up of AND gates, full adders and half adders. Each line dealing with one of the bits of the second 8 bit number along with each bit of the first 8 bit number, they then filter into an adder along with the sum output of the previous line.

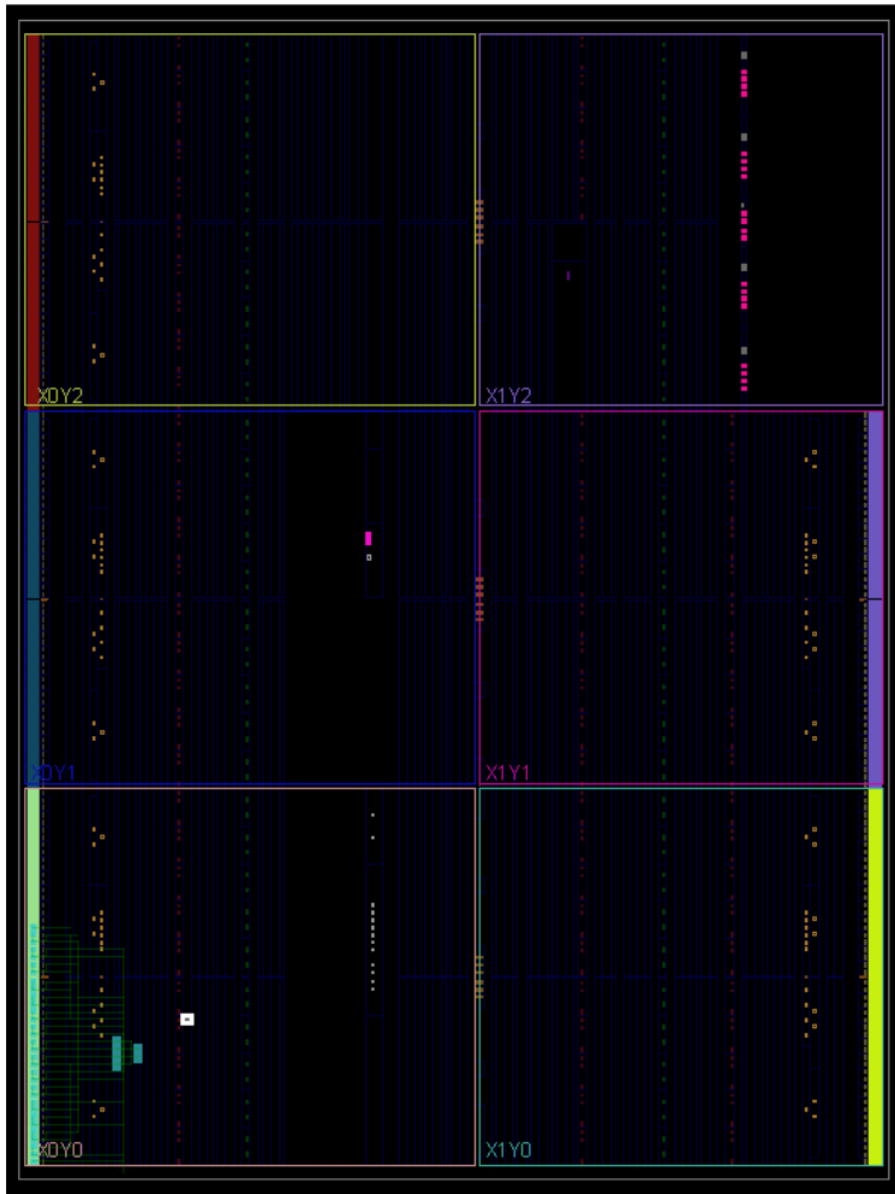
Wallace Tree



The Wallace Tree multiplier uses a series of full and half adders to condense the system at each level, as seen in the above diagram. This filters the inputs and outputs down to a final 16 bit number.

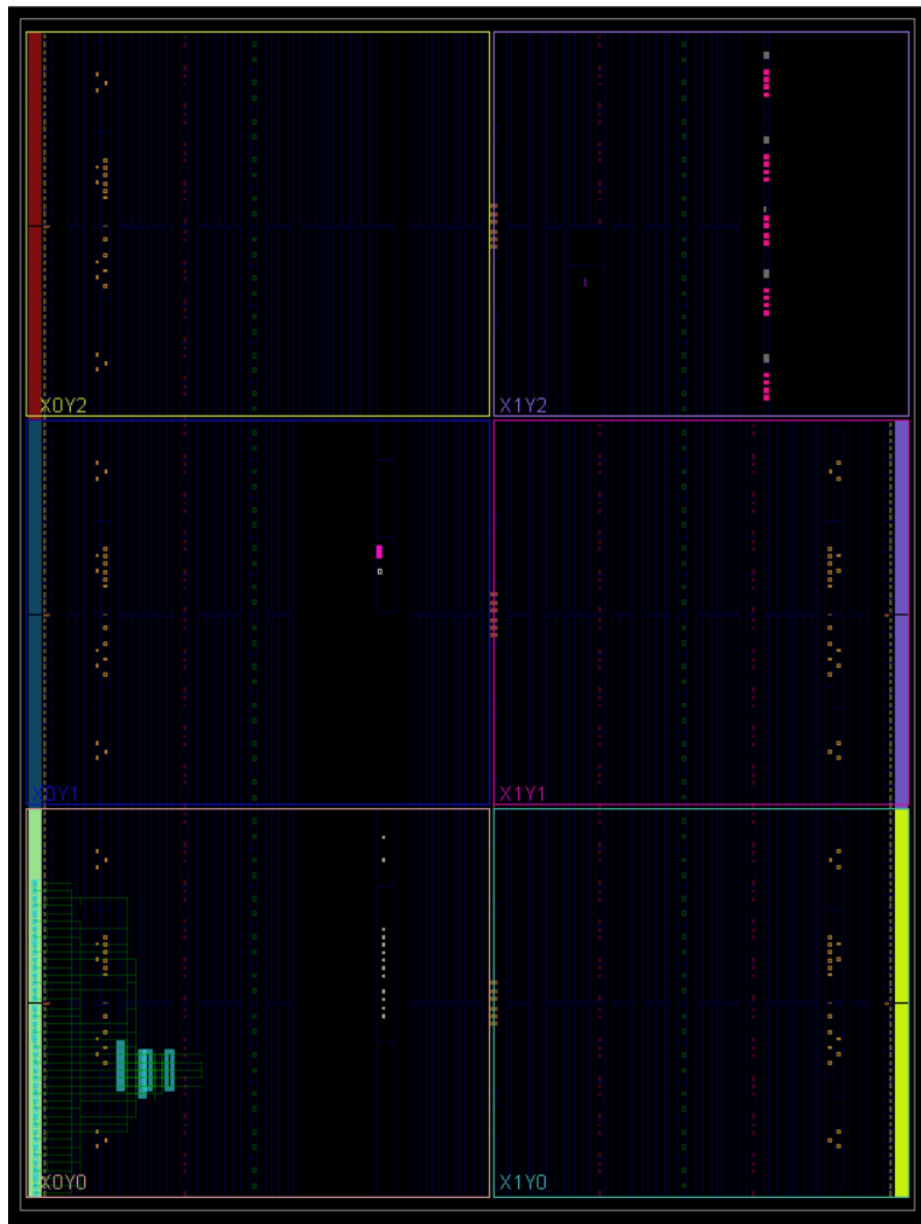
FPGA Diagrams

Brauns



This is the FPGA diagram for the Brauns multiplier, this shows each of the configurable logic blocks surrounded by the system of programmable interconnects. The highlighted blocks are those used by the designed system, along with the pathways that connect each input and output. This gives a visual representation of what the physical chip would look like and how it would be connected if built to be able to carry out the created multiplier code.

Wallace Tree



This is the FPGA diagram for the Wallace Tree multiplier, this shows each of the configurable logic blocks surrounded by the system of programmable interconnects. The highlighted blocks are those used by the designed system. The highlighted paths are the connections between each input and output.

VHDL Codes

Brauns

```
library ieee;
use ieee.std_logic_1164.all;
entity multiply is
port (A, B : in bit_vector(7 downto 0);
      P : out bit_vector(15 downto 0));
end multiply;
architecture dataflow of multiply is
begin
    P(0) <= A(0) AND B(0);
    P(1) <= (A(1) AND B(0)) XOR (A(0) AND B(1));
    P(2) <= ((A(2) AND B(0)) XOR (A(1) AND B(1))) XOR (A(0) AND B(2));
    P(3) <= ((A(3) AND B(0)) XOR (A(2) AND B(1))) XOR (A(1) AND B(2)) XOR (A(0) AND B(3));
    P(4) <= ((A(4) AND B(0)) XOR (A(3) AND B(1))) XOR (A(2) AND B(2)) XOR (A(1) AND B(3)) XOR (A(0) AND B(4));
    P(5) <= ((A(5) AND B(0)) XOR (A(4) AND B(1))) XOR (A(3) AND B(2)) XOR (A(2) AND B(3)) XOR (A(1) AND B(4)) XOR (A(0) AND B(5));
    P(6) <= ((A(6) AND B(0)) XOR (A(5) AND B(1))) XOR (A(4) AND B(2)) XOR (A(3) AND B(3)) XOR (A(2) AND B(4)) XOR (A(1) AND B(5)) XOR (A(0) AND B(6));
    P(7) <= ((A(7) AND B(0)) XOR (A(6) AND B(1))) XOR (A(5) AND B(2)) XOR (A(4) AND B(3)) XOR (A(3) AND B(4)) XOR (A(2) AND B(5)) XOR (A(1) AND B(6)) XOR (A(0) AND B(7));
    P(8) <= ((A(7) AND B(1)) XOR (A(6) AND B(2)) XOR (A(5) AND B(3)) XOR (A(4) AND B(4)) XOR (A(3) AND B(5)) XOR (A(2) AND B(6)) XOR (A(1) AND B(7)));
    P(9) <= ((A(7) AND B(2)) XOR (A(6) AND B(3)) XOR (A(5) AND B(4)) XOR (A(4) AND B(5)) XOR (A(3) AND B(6)) XOR (A(2) AND B(7)));
    P(10) <= ((A(7) AND B(3)) XOR (A(6) AND B(4)) XOR (A(5) AND B(5)) XOR (A(4) AND B(6)) XOR (A(3) AND B(7)));
    P(11) <= ((A(7) AND B(4)) XOR (A(6) AND B(5)) XOR (A(5) AND B(6)) XOR (A(4) AND B(7)));
    P(12) <= ((A(7) AND B(5)) XOR (A(6) AND B(6)) XOR (A(5) AND B(7)));
    P(13) <= ((A(7) AND B(6)) XOR (A(6) AND B(7)));
    P(14) <= ((A(7) AND B(7)));
end architecture;
```

As you can see from this code, the outputs follow the flow chart of how the multiplier works and works very similarly to long multiplication. Each output is made up of AND and XOR operation gates with all possibilities given, the possibilities are determined by the table shown in the information section of this report.

Wallace Tree

10

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity walltree_mul is
    port (
        op1 : in std_logic_vector(9 downto 0 );
        op2 : in std_logic_vector(9 downto 0 );
        result : out std_logic_vector(20 downto 0 )
    );
end walltree_mul;
architecture rtl of walltree_mul is
    -- signals for the partial products

    signal pp0 : std_logic_vector(9 downto 0 );
    signal pp1 : std_logic_vector(9 downto 0 );
    signal pp2 : std_logic_vector(9 downto 0 );
    signal pp3 : std_logic_vector(9 downto 0 );
    signal pp4 : std_logic_vector(9 downto 0 );
    signal pp5 : std_logic_vector(9 downto 0 );
    signal pp6 : std_logic_vector(9 downto 0 );
    signal pp7 : std_logic_vector(9 downto 0 );
    signal pp8 : std_logic_vector(9 downto 0 );
    signal pp9 : std_logic_vector(9 downto 0 );

    -- internal signals

    -- layer : 1

    -- csa : 0
    signal sigCSA_sum_0 : std_logic_vector (11 downto 0);
    signal sigCSA_cry_0 : std_logic_vector (11 downto 0);
    -- csa : 1
    signal sigCSA_sum_1 : std_logic_vector (11 downto 0);
    signal sigCSA_cry_1 : std_logic_vector (11 downto 0);
    -- csa : 2
    signal sigCSA_sum_2 : std_logic_vector (11 downto 0);
    signal sigCSA_cry_2 : std_logic_vector (11 downto 0);

    -- layer : 2

    -- csa : 3
    signal sigCSA_sum_3 : std_logic_vector (14 downto 0);
    signal sigCSA_cry_3 : std_logic_vector (14 downto 0);
```

```

-- csa : 4
    signal sigCSA_sum_4 : std_logic_vector (14 downto 0);
    signal sigCSA_cry_4 : std_logic_vector (14 downto 0);

-- layer : 3

-- csa : 5
    signal sigCSA_sum_5 : std_logic_vector (18 downto 0);
    signal sigCSA_cry_5 : std_logic_vector (18 downto 0);

-- layer : 4

-- csa : 6
    signal sigCSA_sum_6 : std_logic_vector (19 downto 0);
    signal sigCSA_cry_6 : std_logic_vector (19 downto 0);

-- layer : 5

-- csa : 7
    signal sigCSA_sum_7 : std_logic_vector (20 downto 0);
    signal sigCSA_cry_7 : std_logic_vector (20 downto 0);
--sigCSA_sum_7
--sigCSA_cry_7
-- all csa structures generated
-- carry signal for the final RCA

    signal carry_rca    : std_logic_vector( 19 downto 0 );
begin

    pp0(0) <= op1(0) and op2(0);
    pp0(1) <= op1(0) and op2(1);
    pp0(2) <= op1(0) and op2(2);
    pp0(3) <= op1(0) and op2(3);
    pp0(4) <= op1(0) and op2(4);
    pp0(5) <= op1(0) and op2(5);
    pp0(6) <= op1(0) and op2(6);
    pp0(7) <= op1(0) and op2(7);
    pp0(8) <= op1(0) and op2(8);
    pp0(9) <= op1(0) and op2(9);
    ppl(0) <= op1(1) and op2(0);
    ppl(1) <= op1(1) and op2(1);
    ppl(2) <= op1(1) and op2(2);
    ppl(3) <= op1(1) and op2(3);

```

```

pp1(4) <= op1(1) and op2(4);
pp1(5) <= op1(1) and op2(5);
pp1(6) <= op1(1) and op2(6);
pp1(7) <= op1(1) and op2(7);
pp1(8) <= op1(1) and op2(8);
pp1(9) <= op1(1) and op2(9);
pp2(0) <= op1(2) and op2(0);
pp2(1) <= op1(2) and op2(1);
pp2(2) <= op1(2) and op2(2);
pp2(3) <= op1(2) and op2(3);
pp2(4) <= op1(2) and op2(4);
pp2(5) <= op1(2) and op2(5);
pp2(6) <= op1(2) and op2(6);
pp2(7) <= op1(2) and op2(7);
pp2(8) <= op1(2) and op2(8);
pp2(9) <= op1(2) and op2(9);
pp3(0) <= op1(3) and op2(0);
pp3(1) <= op1(3) and op2(1);
pp3(2) <= op1(3) and op2(2);
pp3(3) <= op1(3) and op2(3);
pp3(4) <= op1(3) and op2(4);
pp3(5) <= op1(3) and op2(5);
pp3(6) <= op1(3) and op2(6);
pp3(7) <= op1(3) and op2(7);
pp3(8) <= op1(3) and op2(8);
pp3(9) <= op1(3) and op2(9);
pp4(0) <= op1(4) and op2(0);
pp4(1) <= op1(4) and op2(1);
pp4(2) <= op1(4) and op2(2);
pp4(3) <= op1(4) and op2(3);
pp4(4) <= op1(4) and op2(4);
pp4(5) <= op1(4) and op2(5);
pp4(6) <= op1(4) and op2(6);
pp4(7) <= op1(4) and op2(7);
pp4(8) <= op1(4) and op2(8);
pp4(9) <= op1(4) and op2(9);
pp5(0) <= op1(5) and op2(0);
pp5(1) <= op1(5) and op2(1);
pp5(2) <= op1(5) and op2(2);
pp5(3) <= op1(5) and op2(3);
pp5(4) <= op1(5) and op2(4);
pp5(5) <= op1(5) and op2(5);
pp5(6) <= op1(5) and op2(6);
pp5(7) <= op1(5) and op2(7);
...

```

```

pp5(8) <= opl(5) and op2(8);
pp5(9) <= opl(5) and op2(9);
pp6(0) <= opl(6) and op2(0);
pp6(1) <= opl(6) and op2(1);
pp6(2) <= opl(6) and op2(2);
pp6(3) <= opl(6) and op2(3);
pp6(4) <= opl(6) and op2(4);
pp6(5) <= opl(6) and op2(5);
pp6(6) <= opl(6) and op2(6);
pp6(7) <= opl(6) and op2(7);
pp6(8) <= opl(6) and op2(8);
pp6(9) <= opl(6) and op2(9);
pp7(0) <= opl(7) and op2(0);
pp7(1) <= opl(7) and op2(1);
pp7(2) <= opl(7) and op2(2);
pp7(3) <= opl(7) and op2(3);
pp7(4) <= opl(7) and op2(4);
pp7(5) <= opl(7) and op2(5);
pp7(6) <= opl(7) and op2(6);
pp7(7) <= opl(7) and op2(7);
pp7(8) <= opl(7) and op2(8);
pp7(9) <= opl(7) and op2(9);
pp8(0) <= opl(8) and op2(0);
pp8(1) <= opl(8) and op2(1);
pp8(2) <= opl(8) and op2(2);
pp8(3) <= opl(8) and op2(3);
pp8(4) <= opl(8) and op2(4);
pp8(5) <= opl(8) and op2(5);
pp8(6) <= opl(8) and op2(6);
pp8(7) <= opl(8) and op2(7);
pp8(8) <= opl(8) and op2(8);
pp8(9) <= opl(8) and op2(9);
pp9(0) <= opl(9) and op2(0);
pp9(1) <= opl(9) and op2(1);
pp9(2) <= opl(9) and op2(2);
pp9(3) <= opl(9) and op2(3);
pp9(4) <= opl(9) and op2(4);
pp9(5) <= opl(9) and op2(5);
pp9(6) <= opl(9) and op2(6);
pp9(7) <= opl(9) and op2(7);
pp9(8) <= opl(9) and op2(8);
pp9(9) <= opl(9) and op2(9);

```

```

-- csa : 0
-- generating sigCSA_sum_0 and sigCSA_cry_0

sigCSA_sum_0(0) <= pp0(0) xor '0' xor '0' ;
sigCSA_cry_0(0) <= ( pp0(0) and '0' ) or ( '0' and ( pp0(0) xor '0' ) ) ;
sigCSA_sum_0(1) <= pp0(1) xor ppl(0) xor '0' ;
sigCSA_cry_0(1) <= ( pp0(1) and ppl(0) ) or ( '0' and ( pp0(1) xor ppl(0) ) ) ;
sigCSA_sum_0(2) <= pp0(2) xor ppl(1) xor pp2(0) ;
sigCSA_cry_0(2) <= ( pp0(2) and ppl(1) ) or ( pp2(0) and ( pp0(2) xor ppl(1) ) ) ;
sigCSA_sum_0(3) <= pp0(3) xor ppl(2) xor pp2(1) ;
sigCSA_cry_0(3) <= ( pp0(3) and ppl(2) ) or ( pp2(1) and ( pp0(3) xor ppl(2) ) ) ;
sigCSA_sum_0(4) <= pp0(4) xor ppl(3) xor pp2(2) ;
sigCSA_cry_0(4) <= ( pp0(4) and ppl(3) ) or ( pp2(2) and ( pp0(4) xor ppl(3) ) ) ;
sigCSA_sum_0(5) <= pp0(5) xor ppl(4) xor pp2(3) ;
sigCSA_cry_0(5) <= ( pp0(5) and ppl(4) ) or ( pp2(3) and ( pp0(5) xor ppl(4) ) ) ;
sigCSA_sum_0(6) <= pp0(6) xor ppl(5) xor pp2(4) ;
sigCSA_cry_0(6) <= ( pp0(6) and ppl(5) ) or ( pp2(4) and ( pp0(6) xor ppl(5) ) ) ;
sigCSA_sum_0(7) <= pp0(7) xor ppl(6) xor pp2(5) ;
sigCSA_cry_0(7) <= ( pp0(7) and ppl(6) ) or ( pp2(5) and ( pp0(7) xor ppl(6) ) ) ;
sigCSA_sum_0(8) <= pp0(8) xor ppl(7) xor pp2(6) ;
sigCSA_cry_0(8) <= ( pp0(8) and ppl(7) ) or ( pp2(6) and ( pp0(8) xor ppl(7) ) ) ;
sigCSA_sum_0(9) <= pp0(9) xor ppl(8) xor pp2(7) ;
sigCSA_cry_0(9) <= ( pp0(9) and ppl(8) ) or ( pp2(7) and ( pp0(9) xor ppl(8) ) ) ;
sigCSA_sum_0(10) <= '0' xor ppl(9) xor pp2(8) ;
sigCSA_cry_0(10) <= ( '0' and ppl(9) ) or ( pp2(8) and ( '0' xor ppl(9) ) ) ;
sigCSA_sum_0(11) <= '0' xor '0' xor pp2(9) ;
sigCSA_cry_0(11) <= ( '0' and '0' ) or ( pp2(9) and ( '0' xor '0' ) ) ;

-- csa : 1
-- generating sigCSA_sum_1 and sigCSA_cry_1

sigCSA_sum_1(0) <= pp3(0) xor '0' xor '0' ;
sigCSA_cry_1(0) <= ( pp3(0) and '0' ) or ( '0' and ( pp3(0) xor '0' ) ) ;
sigCSA_sum_1(1) <= pp3(1) xor pp4(0) xor '0' ;
sigCSA_cry_1(1) <= ( pp3(1) and pp4(0) ) or ( '0' and ( pp3(1) xor pp4(0) ) ) ;
sigCSA_sum_1(2) <= pp3(2) xor pp4(1) xor pp5(0) ;
sigCSA_cry_1(2) <= ( pp3(2) and pp4(1) ) or ( pp5(0) and ( pp3(2) xor pp4(1) ) ) ;
sigCSA_sum_1(3) <= pp3(3) xor pp4(2) xor pp5(1) ;
sigCSA_cry_1(3) <= ( pp3(3) and pp4(2) ) or ( pp5(1) and ( pp3(3) xor pp4(2) ) ) ;
sigCSA_sum_1(4) <= pp3(4) xor pp4(3) xor pp5(2) ;
sigCSA_cry_1(4) <= ( pp3(4) and pp4(3) ) or ( pp5(2) and ( pp3(4) xor pp4(3) ) ) ;
sigCSA_sum_1(5) <= pp3(5) xor pp4(4) xor pp5(3) ;
sigCSA_cry_1(5) <= ( pp3(5) and pp4(4) ) or ( pp5(3) and ( pp3(5) xor pp4(4) ) ) ;
sigCSA_sum_1(6) <= pp3(6) xor pp4(5) xor pp5(4) ;
sigCSA_cry_1(6) <= ( pp3(6) and pp4(5) ) or ( pp5(4) and ( pp3(6) xor pp4(5) ) ) ;
sigCSA_sum_1(7) <= pp3(7) xor pp4(6) xor pp5(5) ;
sigCSA_cry_1(7) <= ( pp3(7) and pp4(6) ) or ( pp5(5) and ( pp3(7) xor pp4(6) ) ) ;
sigCSA_sum_1(8) <= pp3(8) xor pp4(7) xor pp5(6) ;
sigCSA_cry_1(8) <= ( pp3(8) and pp4(7) ) or ( pp5(6) and ( pp3(8) xor pp4(7) ) ) ;
sigCSA_sum_1(9) <= pp3(9) xor pp4(8) xor pp5(7) ;
sigCSA_cry_1(9) <= ( pp3(9) and pp4(8) ) or ( pp5(7) and ( pp3(9) xor pp4(8) ) ) ;
sigCSA_sum_1(10) <= '0' xor pp4(9) xor pp5(8) ;
sigCSA_cry_1(10) <= ( '0' and pp4(9) ) or ( pp5(8) and ( '0' xor pp4(9) ) ) ;
sigCSA_sum_1(11) <= '0' xor '0' xor pp5(9) ;
sigCSA_cry_1(11) <= ( '0' and '0' ) or ( pp5(9) and ( '0' xor '0' ) ) ;

```

```

-- csa : 2
-- generating sigCSA_sum_2 and sigCSA_cry_2

sigCSA_sum_2(0) <= pp6(0) xor '0' xor '0' ;
sigCSA_cry_2(0) <= ( pp6(0) and '0' ) or ( '0' and ( pp6(0) xor '0' ) ) ;
sigCSA_sum_2(1) <= pp6(1) xor pp7(0) xor '0' ;
sigCSA_cry_2(1) <= ( pp6(1) and pp7(0) ) or ( '0' and ( pp6(1) xor pp7(0) ) ) ;
sigCSA_sum_2(2) <= pp6(2) xor pp7(1) xor pp8(0) ;
sigCSA_cry_2(2) <= ( pp6(2) and pp7(1) ) or ( pp8(0) and ( pp6(2) xor pp7(1) ) ) ;
sigCSA_sum_2(3) <= pp6(3) xor pp7(2) xor pp8(1) ;
sigCSA_cry_2(3) <= ( pp6(3) and pp7(2) ) or ( pp8(1) and ( pp6(3) xor pp7(2) ) ) ;
sigCSA_sum_2(4) <= pp6(4) xor pp7(3) xor pp8(2) ;
sigCSA_cry_2(4) <= ( pp6(4) and pp7(3) ) or ( pp8(2) and ( pp6(4) xor pp7(3) ) ) ;
sigCSA_sum_2(5) <= pp6(5) xor pp7(4) xor pp8(3) ;
sigCSA_cry_2(5) <= ( pp6(5) and pp7(4) ) or ( pp8(3) and ( pp6(5) xor pp7(4) ) ) ;
sigCSA_sum_2(6) <= pp6(6) xor pp7(5) xor pp8(4) ;
sigCSA_cry_2(6) <= ( pp6(6) and pp7(5) ) or ( pp8(4) and ( pp6(6) xor pp7(5) ) ) ;
sigCSA_sum_2(7) <= pp6(7) xor pp7(6) xor pp8(5) ;
sigCSA_cry_2(7) <= ( pp6(7) and pp7(6) ) or ( pp8(5) and ( pp6(7) xor pp7(6) ) ) ;
sigCSA_sum_2(8) <= pp6(8) xor pp7(7) xor pp8(6) ;
sigCSA_cry_2(8) <= ( pp6(8) and pp7(7) ) or ( pp8(6) and ( pp6(8) xor pp7(7) ) ) ;
sigCSA_sum_2(9) <= pp6(9) xor pp7(8) xor pp8(7) ;
sigCSA_cry_2(9) <= ( pp6(9) and pp7(8) ) or ( pp8(7) and ( pp6(9) xor pp7(8) ) ) ;
sigCSA_sum_2(10) <= '0' xor pp7(9) xor pp8(8) ;
sigCSA_cry_2(10) <= ( '0' and pp7(9) ) or ( pp8(8) and ( '0' xor pp7(9) ) ) ;
sigCSA_sum_2(11) <= '0' xor '0' xor pp8(9) ;
sigCSA_cry_2(11) <= ( '0' and '0' ) or ( pp8(9) and ( '0' xor '0' ) ) ;

-- csa : 3
-- generating sigCSA_sum_3 and sigCSA_cry_3

sigCSA_sum_3(0) <= sigCSA_sum_0(0) xor '0' xor '0' ;
sigCSA_cry_3(0) <= ( sigCSA_sum_0(0) and '0' ) or ( '0' and ( sigCSA_sum_0(0) xor '0' ) ) ;
sigCSA_sum_3(1) <= sigCSA_sum_0(1) xor sigCSA_cry_0(0) xor '0' ;
sigCSA_cry_3(1) <= ( sigCSA_sum_0(1) and sigCSA_cry_0(0) ) or ( '0' and ( sigCSA_sum_0(1) xor sigCSA_cry_0(0) ) ) ;
sigCSA_sum_3(2) <= sigCSA_sum_0(2) xor sigCSA_cry_0(1) xor '0' ;
sigCSA_cry_3(2) <= ( sigCSA_sum_0(2) and sigCSA_cry_0(1) ) or ( '0' and ( sigCSA_sum_0(2) xor sigCSA_cry_0(1) ) ) ;
sigCSA_sum_3(3) <= sigCSA_sum_0(3) xor sigCSA_cry_0(2) xor sigCSA_sum_1(0) ;
sigCSA_cry_3(3) <= ( sigCSA_sum_0(3) and sigCSA_cry_0(2) ) or ( sigCSA_sum_1(0) and ( sigCSA_sum_0(3) xor sigCSA_cry_0(2) ) ) ;
sigCSA_sum_3(4) <= sigCSA_sum_0(4) xor sigCSA_cry_0(3) xor sigCSA_sum_1(1) ;
sigCSA_cry_3(4) <= ( sigCSA_sum_0(4) and sigCSA_cry_0(3) ) or ( sigCSA_sum_1(1) and ( sigCSA_sum_0(4) xor sigCSA_cry_0(3) ) ) ;
sigCSA_sum_3(5) <= sigCSA_sum_0(5) xor sigCSA_cry_0(4) xor sigCSA_sum_1(2) ;
sigCSA_cry_3(5) <= ( sigCSA_sum_0(5) and sigCSA_cry_0(4) ) or ( sigCSA_sum_1(2) and ( sigCSA_sum_0(5) xor sigCSA_cry_0(4) ) ) ;
sigCSA_sum_3(6) <= sigCSA_sum_0(6) xor sigCSA_cry_0(5) xor sigCSA_sum_1(3) ;
sigCSA_cry_3(6) <= ( sigCSA_sum_0(6) and sigCSA_cry_0(5) ) or ( sigCSA_sum_1(3) and ( sigCSA_sum_0(6) xor sigCSA_cry_0(5) ) ) ;
sigCSA_sum_3(7) <= sigCSA_sum_0(7) xor sigCSA_cry_0(6) xor sigCSA_sum_1(4) ;
sigCSA_cry_3(7) <= ( sigCSA_sum_0(7) and sigCSA_cry_0(6) ) or ( sigCSA_sum_1(4) and ( sigCSA_sum_0(7) xor sigCSA_cry_0(6) ) ) ;
sigCSA_sum_3(8) <= sigCSA_sum_0(8) xor sigCSA_cry_0(7) xor sigCSA_sum_1(5) ;
sigCSA_cry_3(8) <= ( sigCSA_sum_0(8) and sigCSA_cry_0(7) ) or ( sigCSA_sum_1(5) and ( sigCSA_sum_0(8) xor sigCSA_cry_0(7) ) ) ;
sigCSA_sum_3(9) <= sigCSA_sum_0(9) xor sigCSA_cry_0(8) xor sigCSA_sum_1(6) ;
sigCSA_cry_3(9) <= ( sigCSA_sum_0(9) and sigCSA_cry_0(8) ) or ( sigCSA_sum_1(6) and ( sigCSA_sum_0(9) xor sigCSA_cry_0(8) ) ) ;
sigCSA_sum_3(10) <= sigCSA_sum_0(10) xor sigCSA_cry_0(9) xor sigCSA_sum_1(7) ;
sigCSA_cry_3(10) <= ( sigCSA_sum_0(10) and sigCSA_cry_0(9) ) or ( sigCSA_sum_1(7) and ( sigCSA_sum_0(10) xor sigCSA_cry_0(9) ) ) ;
sigCSA_sum_3(11) <= sigCSA_sum_0(11) xor sigCSA_cry_0(10) xor sigCSA_sum_1(8) ;
sigCSA_cry_3(11) <= ( sigCSA_sum_0(11) and sigCSA_cry_0(10) ) or ( sigCSA_sum_1(8) and ( sigCSA_sum_0(11) xor sigCSA_cry_0(10) ) ) ;
sigCSA_sum_3(12) <= '0' xor sigCSA_cry_0(11) xor sigCSA_sum_1(9) ;
sigCSA_cry_3(12) <= ( '0' and sigCSA_cry_0(11) ) or ( sigCSA_sum_1(9) and ( '0' xor sigCSA_cry_0(11) ) ) ;
sigCSA_sum_3(13) <= '0' xor '0' xor sigCSA_sum_1(10) ;
sigCSA_cry_3(13) <= ( '0' and '0' ) or ( sigCSA_sum_1(10) and ( '0' xor '0' ) ) ;
sigCSA_sum_3(14) <= '0' xor '0' xor sigCSA_sum_1(11) ;
sigCSA_cry_3(14) <= ( '0' and '0' ) or ( sigCSA_sum_1(11) and ( '0' xor '0' ) ) ;

```

```

-- csa : 4
-- generating sigCSA_sum_4 and sigCSA_cry_4

sigCSA_sum_4(0) <= sigCSA_cry_1(0) xor '0' xor '0' ;
sigCSA_cry_4(0) <= ( sigCSA_cry_1(0) and '0' ) or ( '0' and ( sigCSA_cry_1(0) xor '0' ) ) ;
sigCSA_sum_4(1) <= sigCSA_cry_1(1) xor '0' xor '0' ;
sigCSA_cry_4(1) <= ( sigCSA_cry_1(1) and '0' ) or ( '0' and ( sigCSA_cry_1(1) xor '0' ) ) ;
sigCSA_sum_4(2) <= sigCSA_cry_1(2) xor sigCSA_sum_2(0) xor '0' ;
sigCSA_cry_4(2) <= ( sigCSA_cry_1(2) and sigCSA_sum_2(0) ) or ( '0' and ( sigCSA_cry_1(2) xor sigCSA_sum_2(0) ) ) ;
sigCSA_sum_4(3) <= sigCSA_cry_1(3) xor sigCSA_sum_2(1) xor sigCSA_cry_2(0) ;
sigCSA_cry_4(3) <= ( sigCSA_cry_1(3) and sigCSA_sum_2(1) ) or ( sigCSA_cry_2(0) and ( sigCSA_cry_1(3) xor sigCSA_sum_2(1) ) ) ;
sigCSA_sum_4(4) <= sigCSA_cry_1(4) xor sigCSA_sum_2(2) xor sigCSA_cry_2(1) ;
sigCSA_cry_4(4) <= ( sigCSA_cry_1(4) and sigCSA_sum_2(2) ) or ( sigCSA_cry_2(1) and ( sigCSA_cry_1(4) xor sigCSA_sum_2(2) ) ) ;
sigCSA_sum_4(5) <= sigCSA_cry_1(5) xor sigCSA_sum_2(3) xor sigCSA_cry_2(2) ;
sigCSA_cry_4(5) <= ( sigCSA_cry_1(5) and sigCSA_sum_2(3) ) or ( sigCSA_cry_2(2) and ( sigCSA_cry_1(5) xor sigCSA_sum_2(3) ) ) ;
sigCSA_sum_4(6) <= sigCSA_cry_1(6) xor sigCSA_sum_2(4) xor sigCSA_cry_2(3) ;
sigCSA_cry_4(6) <= ( sigCSA_cry_1(6) and sigCSA_sum_2(4) ) or ( sigCSA_cry_2(3) and ( sigCSA_cry_1(6) xor sigCSA_sum_2(4) ) ) ;
sigCSA_sum_4(7) <= sigCSA_cry_1(7) xor sigCSA_sum_2(5) xor sigCSA_cry_2(4) ;
sigCSA_cry_4(7) <= ( sigCSA_cry_1(7) and sigCSA_sum_2(5) ) or ( sigCSA_cry_2(4) and ( sigCSA_cry_1(7) xor sigCSA_sum_2(5) ) ) ;
sigCSA_sum_4(8) <= sigCSA_cry_1(8) xor sigCSA_sum_2(6) xor sigCSA_cry_2(5) ;
sigCSA_cry_4(8) <= ( sigCSA_cry_1(8) and sigCSA_sum_2(6) ) or ( sigCSA_cry_2(5) and ( sigCSA_cry_1(8) xor sigCSA_sum_2(6) ) ) ;
sigCSA_sum_4(9) <= sigCSA_cry_1(9) xor sigCSA_sum_2(7) xor sigCSA_cry_2(6) ;
sigCSA_cry_4(9) <= ( sigCSA_cry_1(9) and sigCSA_sum_2(7) ) or ( sigCSA_cry_2(6) and ( sigCSA_cry_1(9) xor sigCSA_sum_2(7) ) ) ;
sigCSA_sum_4(10) <= sigCSA_cry_1(10) xor sigCSA_sum_2(8) xor sigCSA_cry_2(7) ;
sigCSA_cry_4(10) <= ( sigCSA_cry_1(10) and sigCSA_sum_2(8) ) or ( sigCSA_cry_2(7) and ( sigCSA_cry_1(10) xor sigCSA_sum_2(8) ) ) ;
sigCSA_sum_4(11) <= sigCSA_cry_1(11) xor sigCSA_sum_2(9) xor sigCSA_cry_2(8) ;
sigCSA_cry_4(11) <= ( sigCSA_cry_1(11) and sigCSA_sum_2(9) ) or ( sigCSA_cry_2(8) and ( sigCSA_cry_1(11) xor sigCSA_sum_2(9) ) ) ;
sigCSA_sum_4(12) <= '0' xor sigCSA_sum_2(10) xor sigCSA_cry_2(9) ;
sigCSA_cry_4(12) <= ( '0' and sigCSA_sum_2(10) ) or ( sigCSA_cry_2(9) and ( '0' xor sigCSA_sum_2(10) ) ) ;
sigCSA_sum_4(13) <= '0' xor sigCSA_sum_2(11) xor sigCSA_cry_2(10) ;
sigCSA_cry_4(13) <= ( '0' and sigCSA_sum_2(11) ) or ( sigCSA_cry_2(10) and ( '0' xor sigCSA_sum_2(11) ) ) ;
sigCSA_sum_4(14) <= '0' xor '0' xor sigCSA_cry_2(11) ;
sigCSA_cry_4(14) <= ( '0' and '0' ) or ( sigCSA_cry_2(11) and ( '0' xor '0' ) ) ;

-- csa : 5
-- generating sigCSA_sum_5 and sigCSA_cry_5

sigCSA_sum_5(0) <= sigCSA_sum_3(0) xor '0' xor '0' ;
sigCSA_cry_5(0) <= ( sigCSA_sum_3(0) and '0' ) or ( '0' and ( sigCSA_sum_3(0) xor '0' ) ) ;
sigCSA_sum_5(1) <= sigCSA_sum_3(1) xor sigCSA_cry_3(0) xor '0' ;
sigCSA_cry_5(1) <= ( sigCSA_sum_3(1) and sigCSA_cry_3(0) ) or ( '0' and ( sigCSA_sum_3(1) xor sigCSA_cry_3(0) ) ) ;
sigCSA_sum_5(2) <= sigCSA_sum_3(2) xor sigCSA_cry_3(1) xor '0' ;
sigCSA_cry_5(2) <= ( sigCSA_sum_3(2) and sigCSA_cry_3(1) ) or ( '0' and ( sigCSA_sum_3(2) xor sigCSA_cry_3(1) ) ) ;
sigCSA_sum_5(3) <= sigCSA_sum_3(3) xor sigCSA_cry_3(2) xor '0' ;
sigCSA_cry_5(3) <= ( sigCSA_sum_3(3) and sigCSA_cry_3(2) ) or ( '0' and ( sigCSA_sum_3(3) xor sigCSA_cry_3(2) ) ) ;
sigCSA_sum_5(4) <= sigCSA_sum_3(4) xor sigCSA_cry_3(3) xor sigCSA_sum_4(0) ;
sigCSA_cry_5(4) <= ( sigCSA_sum_3(4) and sigCSA_cry_3(3) ) or ( sigCSA_sum_4(0) and ( sigCSA_sum_3(4) xor sigCSA_cry_3(3) ) ) ;
sigCSA_sum_5(5) <= sigCSA_sum_3(5) xor sigCSA_cry_3(4) xor sigCSA_sum_4(1) ;
sigCSA_cry_5(5) <= ( sigCSA_sum_3(5) and sigCSA_cry_3(4) ) or ( sigCSA_sum_4(1) and ( sigCSA_sum_3(5) xor sigCSA_cry_3(4) ) ) ;
sigCSA_sum_5(6) <= sigCSA_sum_3(6) xor sigCSA_cry_3(5) xor sigCSA_sum_4(2) ;
sigCSA_cry_5(6) <= ( sigCSA_sum_3(6) and sigCSA_cry_3(5) ) or ( sigCSA_sum_4(2) and ( sigCSA_sum_3(6) xor sigCSA_cry_3(5) ) ) ;
sigCSA_sum_5(7) <= sigCSA_sum_3(7) xor sigCSA_cry_3(6) xor sigCSA_sum_4(3) ;
sigCSA_cry_5(7) <= ( sigCSA_sum_3(7) and sigCSA_cry_3(6) ) or ( sigCSA_sum_4(3) and ( sigCSA_sum_3(7) xor sigCSA_cry_3(6) ) ) ;
sigCSA_sum_5(8) <= sigCSA_sum_3(8) xor sigCSA_cry_3(7) xor sigCSA_sum_4(4) ;
sigCSA_cry_5(8) <= ( sigCSA_sum_3(8) and sigCSA_cry_3(7) ) or ( sigCSA_sum_4(4) and ( sigCSA_sum_3(8) xor sigCSA_cry_3(7) ) ) ;
sigCSA_sum_5(9) <= sigCSA_sum_3(9) xor sigCSA_cry_3(8) xor sigCSA_sum_4(5) ;
sigCSA_cry_5(9) <= ( sigCSA_sum_3(9) and sigCSA_cry_3(8) ) or ( sigCSA_sum_4(5) and ( sigCSA_sum_3(9) xor sigCSA_cry_3(8) ) ) ;
sigCSA_sum_5(10) <= sigCSA_sum_3(10) xor sigCSA_cry_3(9) xor sigCSA_sum_4(6) ;
sigCSA_cry_5(10) <= ( sigCSA_sum_3(10) and sigCSA_cry_3(9) ) or ( sigCSA_sum_4(6) and ( sigCSA_sum_3(10) xor sigCSA_cry_3(9) ) ) ;
sigCSA_sum_5(11) <= sigCSA_sum_3(11) xor sigCSA_cry_3(10) xor sigCSA_sum_4(7) ;
sigCSA_cry_5(11) <= ( sigCSA_sum_3(11) and sigCSA_cry_3(10) ) or ( sigCSA_sum_4(7) and ( sigCSA_sum_3(11) xor sigCSA_cry_3(10) ) ) ;
sigCSA_sum_5(12) <= sigCSA_sum_3(12) xor sigCSA_cry_3(11) xor sigCSA_sum_4(8) ;
sigCSA_cry_5(12) <= ( sigCSA_sum_3(12) and sigCSA_cry_3(11) ) or ( sigCSA_sum_4(8) and ( sigCSA_sum_3(12) xor sigCSA_cry_3(11) ) ) ;
sigCSA_sum_5(13) <= sigCSA_sum_3(13) xor sigCSA_cry_3(12) xor sigCSA_sum_4(9) ;
sigCSA_cry_5(13) <= ( sigCSA_sum_3(13) and sigCSA_cry_3(12) ) or ( sigCSA_sum_4(9) and ( sigCSA_sum_3(13) xor sigCSA_cry_3(12) ) ) ;
sigCSA_sum_5(14) <= sigCSA_sum_3(14) xor sigCSA_cry_3(13) xor sigCSA_sum_4(10) ;
sigCSA_cry_5(14) <= ( sigCSA_sum_3(14) and sigCSA_cry_3(13) ) or ( sigCSA_sum_4(10) and ( sigCSA_sum_3(14) xor sigCSA_cry_3(13) ) ) ;
sigCSA_sum_5(15) <= '0' xor sigCSA_cry_3(14) xor sigCSA_sum_4(11) ;
sigCSA_cry_5(15) <= ( '0' and sigCSA_cry_3(14) ) or ( sigCSA_sum_4(11) and ( '0' xor sigCSA_cry_3(14) ) ) ;
sigCSA_sum_5(16) <= '0' xor '0' xor sigCSA_sum_4(12) ;
sigCSA_cry_5(16) <= ( '0' and '0' ) or ( sigCSA_sum_4(12) and ( '0' xor '0' ) ) ;
sigCSA_sum_5(17) <= '0' xor '0' xor sigCSA_sum_4(13) ;
sigCSA_cry_5(17) <= ( '0' and '0' ) or ( sigCSA_sum_4(13) and ( '0' xor '0' ) ) ;
sigCSA_sum_5(18) <= '0' xor '0' xor sigCSA_sum_4(14) ;
sigCSA_cry_5(18) <= ( '0' and '0' ) or ( sigCSA_sum_4(14) and ( '0' xor '0' ) ) ;

```

23

```

-- the final output
result(0) <= sigCSA_sum_7(0);
result(1) <= sigCSA_sum_7(1) xor sigCSA_cry_7(0) xor '0' ;
carry_rca(0) <= sigCSA_sum_7(1) and sigCSA_cry_7(0);
result(2) <= sigCSA_sum_7(2) xor sigCSA_cry_7(1) xor carry_rca(0);
carry_rca(1) <= ( sigCSA_sum_7(2) and sigCSA_cry_7(1)) or ( carry_rca(0) and ( sigCSA_sum_7(2) xor sigCSA_cry_7(1)));
result(3) <= sigCSA_sum_7(3) xor sigCSA_cry_7(2) xor carry_rca(1);
carry_rca(2) <= ( sigCSA_sum_7(3) and sigCSA_cry_7(2)) or ( carry_rca(1) and ( sigCSA_sum_7(3) xor sigCSA_cry_7(2)));
result(4) <= sigCSA_sum_7(4) xor sigCSA_cry_7(3) xor carry_rca(2);
carry_rca(3) <= ( sigCSA_sum_7(4) and sigCSA_cry_7(3)) or ( carry_rca(2) and ( sigCSA_sum_7(4) xor sigCSA_cry_7(3)));
result(5) <= sigCSA_sum_7(5) xor sigCSA_cry_7(4) xor carry_rca(3);
carry_rca(4) <= ( sigCSA_sum_7(5) and sigCSA_cry_7(4)) or ( carry_rca(3) and ( sigCSA_sum_7(5) xor sigCSA_cry_7(4)));
result(6) <= sigCSA_sum_7(6) xor sigCSA_cry_7(5) xor carry_rca(4);
carry_rca(5) <= ( sigCSA_sum_7(6) and sigCSA_cry_7(5)) or ( carry_rca(4) and ( sigCSA_sum_7(6) xor sigCSA_cry_7(5)));
result(7) <= sigCSA_sum_7(7) xor sigCSA_cry_7(6) xor carry_rca(5);
carry_rca(6) <= ( sigCSA_sum_7(7) and sigCSA_cry_7(6)) or ( carry_rca(5) and ( sigCSA_sum_7(7) xor sigCSA_cry_7(6)));
result(8) <= sigCSA_sum_7(8) xor sigCSA_cry_7(7) xor carry_rca(6);
carry_rca(7) <= ( sigCSA_sum_7(8) and sigCSA_cry_7(7)) or ( carry_rca(6) and ( sigCSA_sum_7(8) xor sigCSA_cry_7(7)));
result(9) <= sigCSA_sum_7(9) xor sigCSA_cry_7(8) xor carry_rca(7);
carry_rca(8) <= ( sigCSA_sum_7(9) and sigCSA_cry_7(8)) or ( carry_rca(7) and ( sigCSA_sum_7(9) xor sigCSA_cry_7(8)));
result(10) <= sigCSA_sum_7(10) xor sigCSA_cry_7(9) xor carry_rca(8);
carry_rca(9) <= ( sigCSA_sum_7(10) and sigCSA_cry_7(9)) or ( carry_rca(8) and ( sigCSA_sum_7(10) xor sigCSA_cry_7(9)));
result(11) <= sigCSA_sum_7(11) xor sigCSA_cry_7(10) xor carry_rca(9);
carry_rca(10) <= ( sigCSA_sum_7(11) and sigCSA_cry_7(10)) or ( carry_rca(9) and ( sigCSA_sum_7(11) xor sigCSA_cry_7(10)));
result(12) <= sigCSA_sum_7(12) xor sigCSA_cry_7(11) xor carry_rca(10);
carry_rca(11) <= ( sigCSA_sum_7(12) and sigCSA_cry_7(11)) or ( carry_rca(10) and ( sigCSA_sum_7(12) xor sigCSA_cry_7(11)));
result(13) <= sigCSA_sum_7(13) xor sigCSA_cry_7(12) xor carry_rca(11);
carry_rca(12) <= ( sigCSA_sum_7(13) and sigCSA_cry_7(12)) or ( carry_rca(11) and ( sigCSA_sum_7(13) xor sigCSA_cry_7(12)));
result(14) <= sigCSA_sum_7(14) xor sigCSA_cry_7(13) xor carry_rca(12);
carry_rca(13) <= ( sigCSA_sum_7(14) and sigCSA_cry_7(13)) or ( carry_rca(12) and ( sigCSA_sum_7(14) xor sigCSA_cry_7(13)));
result(15) <= sigCSA_sum_7(15) xor sigCSA_cry_7(14) xor carry_rca(13);
carry_rca(14) <= ( sigCSA_sum_7(15) and sigCSA_cry_7(14)) or ( carry_rca(13) and ( sigCSA_sum_7(15) xor sigCSA_cry_7(14)));
result(16) <= sigCSA_sum_7(16) xor sigCSA_cry_7(15) xor carry_rca(14);
carry_rca(15) <= ( sigCSA_sum_7(16) and sigCSA_cry_7(15)) or ( carry_rca(14) and ( sigCSA_sum_7(16) xor sigCSA_cry_7(15)));
result(17) <= sigCSA_sum_7(17) xor sigCSA_cry_7(16) xor carry_rca(15);
carry_rca(16) <= ( sigCSA_sum_7(17) and sigCSA_cry_7(16)) or ( carry_rca(15) and ( sigCSA_sum_7(17) xor sigCSA_cry_7(16)));
result(18) <= sigCSA_sum_7(18) xor sigCSA_cry_7(17) xor carry_rca(16);
carry_rca(17) <= ( sigCSA_sum_7(18) and sigCSA_cry_7(17)) or ( carry_rca(16) and ( sigCSA_sum_7(18) xor sigCSA_cry_7(17)));
result(19) <= sigCSA_sum_7(19) xor sigCSA_cry_7(18) xor carry_rca(17);
carry_rca(18) <= ( sigCSA_sum_7(19) and sigCSA_cry_7(18)) or ( carry_rca(17) and ( sigCSA_sum_7(19) xor sigCSA_cry_7(18)));
result(20) <= sigCSA_sum_7(20) xor sigCSA_cry_7(19) xor carry_rca(18);
carry_rca(19) <= ( sigCSA_sum_7(20) and sigCSA_cry_7(19)) or ( carry_rca(18) and ( sigCSA_sum_7(20) xor sigCSA_cry_7(19)));
result(20) <= sigCSA_cry_7(19) xor carry_rca(18);
end rtl;

```

This is the code written to create a Wallace Tree multiplier in VHDL, this is most likely over complicated to able to achieve the desired output but with a lack of knowledge on coding in VHDL, this was the only way I was able to accomplish it. The code is made up using inputs, outputs, Half Adders and Full Adders to achieve the required 16 bit output.

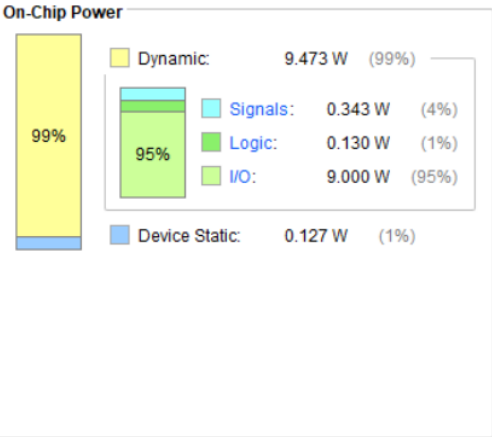
Testing

Brauns

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:	9.599 W
Design Power Budget:	Not Specified
Power Budget Margin:	N/A
Junction Temperature:	70.9°C
Thermal Margin:	29.1°C (6.0 W)
Effective θ_{JA} :	4.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

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



As can be seen, the Brauns multiplier uses 9.599W of total on chip power, junction temperature is 70.9°C. I/O uses most of the power at 9W with signals and logic using the remainder of the power. Although there is a 0.127W of device static.

There is a minimal timing delay on the Brauns multiplier.

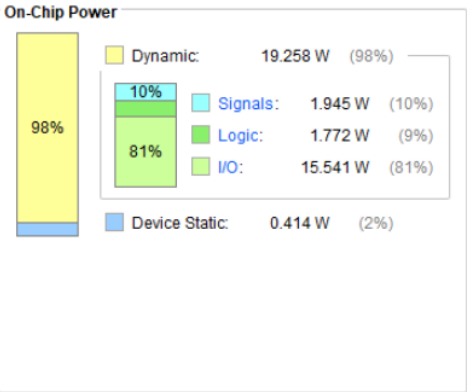
Wallace Tree

Summary

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

Total On-Chip Power:	19.672 W
Design Power Budget:	Not Specified
Power Budget Margin:	N/A
Junction Temperature:	119.0°C
Thermal Margin:	-19.0°C (-3.8 W)
Effective θ_{JA} :	4.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

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



The power consumption of the Wallace tree multiplier is just under 20W, at 19.672W, there is a high junction temperature of 119°C. The I/O uses a massive amount of the power consumption at 15.541W with signals at 1.945W and Logic at 1.772W. There is also 0.414W of device static.

The timing delay on the Wallace tree is slightly larger than that of the Brauns multiplier.

The Wallace Tree multiplier output efficiency in terms of both temperature and total power should in theory be less than what the software is providing in its report. This is because a Wallace Tree multiplier should use less power than what it is using in general and should in theory be less than what is used by the Brauns multiplier.

Final Comparisons

The overall comparison of the two solutions shows that the Brauns multiplier is the better option for an 8x8 multiplier, it has a lower power requirement, faster speed and operates at a lower temperature than the Wallace Tree multiplier, although the Wallace Tree multiplier uses just under double the I/O power consumption it has more than six times the power consumption in signals and over ten times more logic power consumption. The code is also much shorter for the Brauns multiplier meaning the process is faster, it also has a much shorter critical path, this means the longest possible route in the code is shorter than that of the Wallace Tree multiplier, with the Brauns multiplier having a longest possible delay of 5.317 seconds compared to the Wallace Tree multiplier having a longest possible delay of 13.732 seconds. The Brauns multiplier code was also far simpler to write and is much easier to understand compared to the Wallace Tree multiplier code. Overall Brauns multiplier is the better option as it is simpler to code, understand and more efficient and faster compared to the Wallace Tree multiplier.

I believe this is wrong as the Wallace Tree multiplier should in theory be more efficient in every way compared the Brauns multiplier and should be the better option.

Conclusions and Recommendations

In conclusion, the systems work but I believe there are a few improvements that should be made, such as those listed in the testing section of the report. Also, as the Wallace Tree multiplier appears to be wrong in terms of coding, this has brought up faults where it does not give the efficiency expected of it in terms of power and timings. In the future, the Wallace Tree multiplier would be written differently to try and achieve the required operation efficiency.

For future work in the aspects of coding, I would ensure some of the techniques that can reduce the delay, would be used in other programs. I would also ensure that the code is written in a better and more efficient way, such as by condensing it. But this would only be able to be done once my knowledge of coding in VHDL was improved upon doing more research.

References

- “Array multiplier in digital logic,” *GeeksforGeeks*, 30-Dec-2019. [Online]. Available: <https://www.geeksforgeeks.org/array-multiplier-in-digital-logic/>. [Accessed: 09-Apr-2023].
- S. P. Parkash, “Design and analysis of Low Power Braun multiplier architecture,” *Design and Analysis of Low Power Braun Multiplier Architecture*, 2018. [Online]. Available: <https://acadpubl.eu/hub/2018-119-16/2/454.pdf>. [Accessed: 09-Apr-2023].
- S. Laxman, *FPGA Implementation of Different Multiplier Architectures*, 6th ed., vol. 2. Bangalore, India: International Journal of Emerging Technology and Advanced Engineering, 2012.

Daniel Davies E2 Assignment

GRADEMARK REPORT

FINAL GRADE

53/100

GENERAL COMMENTS

Instructor

a. Introduction to Multipliers/ Various Multipliers/ Advantages and Disadvantages/ Applications of Multipliers (8) :4 Marks

Very shallow writing on the introduction. Needs to address more on the types, benefits of Multipliers.

b. Background Research (8) :3 Marks

Very Limited essential reading. No evidence of wider reading. More academic credible literatures from peer reviewed journals need to be referred to the work.

c. Diversity of sources and Referencing Skills(4) :2 Marks
Referencing conventions used largely absent.

a. Task 1: Identifying the correct Multiplier (4), Architecture of Braun's Multiplier(6) :5 Marks

Good! The correct Multiplier is identified. Key concepts were discussed. The Architecture diagram is missing but the systematic multiplication is seen.

b. Task 2: Dot Diagram(4), Architecture of Wallace Multiplier(6) :5 Marks

The dot diagram is seen but architecture was missing. Assumptions on Full adders and half adders were done

a. Finding Critical Path, Longest Delay, Min and Max Delay,

**Power Consumption(12). :8
Marks**

Critical evaluation on finding the Power consumption and delays were made. The conclusion is controversial when you talk about delays.

b.

Innovative thinking in problem solving and feature additions(8) :4 Marks

Generalised statements made with scant evidence.

a. Simulation (7) :4 Marks

Good Try- The simulation results were found with some unexcepted results. The Simulations results are correct only for LSBs for Braun's Multiplier and Expected results were found for Wallace multipliers.

b. VHDL

Coding's (10) :5 Marks

The codes are written for Braun's Multiplier has some errors but works and gives minimal results.

b. Circuit Schematic (4) :3 Marks

The screen dump provided for the Circuit schematic is clear.

d. FPGA

Synthesis (4) :3 Marks

Synthesis is provided. The procedures to capture FPGA synthesis were done perfectly

a. Logical order of ideas, Flow of ideas in the report from the beginning to end with appropriate chapters, sub chapters and continuation(7) :3 Marks

Mostly ordered presentation and structure in which relevant ideas / concepts are reasonably expressed

b.

Definitions,

Acronyms and Abbreviations (3) :2
Marks

Definitions, Acronyms and Abbreviations are captured. **C.**

Professionalism, Spelling and Grammar, Complete Report, Proper Presentation, Quality of the

report(5) :2

Marks

A

satisfactory level of professionalism is seen in the report. Quality of the report needs to be improved.

Total

Marks :53

Marks

Findings and Simulations

Braun's Multiplier

Codes runs without errors or warnings. The code is created

using Expressions. Accuracy of the output is limited. The Simulations results are correct only for LSBs.

Wallace Multiplier

Signal Declaration is made. Here assumptions are made with respect to the sum and carry.

The results are obtained at LSB level and MASs are obtained with few errors.

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

PAGE 1

PAGE 2

PAGE 3



Comment 1

Very shallow writing on the introduction. Needs to address more on the types, benefits of Multipliers.

PAGE 4



Comment 2

More academic credible literatures from peer reviewed journals need to be referred to the work.

PAGE 5



Comment 3

Good! The correct Multiplier is identified.



Comment 4

The Architecture diagram is missing but the systematic multiplication is seen.

PAGE 6

PAGE 7



Comment 5

The dot diagram is seen but architecture was missing.

PAGE 8



Comment 6

Should have used topic, sub topic numbers



Comment 7

The screen dump provided for the Circuit schematic is clear.

PAGE 9

PAGE 10



Comment 8

No Fig no and Fig names found

PAGE 11

PAGE 12

PAGE 13

PAGE 14



Comment 9

Synthesis is provided.

PAGE 15

PAGE 16

PAGE 17



Comment 10

Expected results were found for Wallace multipliers.

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26



Comment 11

Critical evaluation on finding the Power consumption were made.

PAGE 27

PAGE 28



Comment 12

Controversial Statement.

PAGE 29
