



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Electronics II

By

Shaun Farthing

Lecturer: Dr Gary Jones

Module code: ACDA6001

Assignment submitted as part of the
requirements for the award of

BEng (Hons) Electrical and Electronics
Engineering

2021

Table of Contents

Introduction	3
Literature review.....	4
Types of multiplier	5
Design of 12 bit x 4 bit Braun multiplier	7
Parallel multiplier layout.....	9
Half adder.....	10
Half adder behavioural model	12
Full adder	14
Full adder behavioural model	17
Line 0.....	19
Line 0 behavioural model.....	25
Line x	26
Line x behavioural model	35
12x4 multiplier	36
12 bit x 4 bit multiplier behavioural model	40
Part 2 – Design, simulate and synthesise the FSM as VHDL and discrete gate solution	42
Discrete gate solution	42
Schematic of discrete gate solution.....	44
FSM design using VHDL.....	46
References	47
 Figure 1 - Multiplier layout overview.....	 9
Figure 2 - Half adder circuit diagram and half adder truth table.....	10
Figure 3 - Half adder VHDL code	10
Figure 4 - Half adder test bench.....	11
Figure 5 - Half adder, A=0, B=0, Sum=0, Carry=0.....	12
Figure 6 - Half adder, A=0, B=1 Sum=1, Carry=0.....	12
Figure 7 - Half adder, A=1, B=0, Sum=1, Carry=0.....	13
Figure 8 - Half adder, A=1, B=1, Sum=0, Carry=1.....	13
Figure 9 - Full adder circuit diagram and full adder truth table	14
Figure 10 - Full adder VHDL code.....	14
Figure 11 - Full adder test bench	16
Figure 12 - Full adder, A=0, B=0, Cin=0, Sum=0, Cout=0	17
Figure 13 - Full adder, A=0, B=1, Cin=1, Sum=0, Cout=1	18
Figure 14 - Full adder, A=1, B=1, Cin=1, Sum=1, Cout=1	18

Figure 15 - Line 0 VHDL code	19
Figure 16 - Line 0 test bench	24
Figure 17 - Line 0 when B is 0	25
Figure 18 - Line 0 when B is 1	25
Figure 19 - Line x VHDL code	28
Figure 20 – Line x hierarchy	29
Figure 21 – Line x test bench	34
Figure 22 - Line x when b = 0	35
Figure 23 - Line x when b = 1	35
Figure 24 - Multi12x4_full multiplier entity	36
Figure 25 - Line 0 component	37
Figure 26 - Line x component	38
Figure 27 - Multiplier assigned signals	39
Figure 28 - Multiplier port maps	39
Figure 29 - Multiplier (1220*4 = 4880)	40
Figure 30 - Multiplier (1652*2 = 3304)	40
Figure 31 - Multiplier (1672*9 = 15048)	41
Figure 32 - Largest multiplication (4095*15=61425)	41
Figure 33 - State diagram	42
Figure 34 - State table	43
Figure 35 - FSM displayed as a discrete gate solution	45
Figure 36 - FSM in VHDL	46

Introduction

This first part of this document reviews some of the different types of multipliers used in DSP. The document then focuses on the design, simulation and synthetisation of a 12 bit x 4 bit unsigned number multiplier based upon a simple parallel Braun multiplier. Xilinx software will be used to produce the multiplier in VHDL form. The second part of this paper shows the design, simulation and synthetisation of a finite state machine in VHDL form and discrete gate solution based on information provided in the state diagram.

Literature review

A binary multiplier is a combinational logic circuit or digital device used for multiplying two binary numbers. The two numbers are more specifically known as multiplicand and multiplier and the result is known as the product.

The multiplicand and multiplier can be of various bit sizes. The product bit size depends on the bit size of the multiplicand and multiplier. The bit size of the product is equal to the sum of the bit size of the multiplier and multiplicand.

Binary multiplication is the same as decimal multiplication. Binary multiplication of more than 1 – bit numbers contains 2 steps. The first step is single bit multiplication known as partial product and the second step is adding all partial products into a single product. [1]

An efficient multiplier should have the following characteristics: [2]

Accuracy – the multiplier should produce the correct result.

Speed – the multiplier should perform the operation at high speed.

Area – the multiplier should contain as less slices and look up tables (LUT) as possible.

Power – the multiplier should consume as little power as possible.

Types of multiplier

Braun array multiplier

The Braun array multiplier is a simple parallel multiplier that is commonly known as a carry array multiplier. This is only suitable for unsigned numbers. It consists of an array of AND gates and adders in iterative structure and does not require logic registers. It is also known as an additive multiplier, as it does not add additional operand to the result of the multiplication. [3]

If the quantity of bits increase, the number of adders and AND gates components also increases, which may result in increased power consumption, occupy more area and take more time, hence, it is only suitable for less than 16 – bit. A negative of the Braun multiplier is delay, 't' is dependent on the delay of the full adder and also a final adder in the last row. The speed and power of the full adder is very important for large arrays. [3]

Wallace tree multiplier

The Wallace tree multiplier designs capitalize of the concept of the carry save adder. It is like multiplier designs in the sense that they generate and they sum partial products in parallel, but they improve the critical path through the multiplier. [3] Wallace tree reduces the number of partial products and use carry select adder for the addition of partial products. The Wallace tree has three steps, 1) multiply each bit of multiplier with same bit position of multiplicand. Depending on the position of the multiplier bits generated partial products have different weights. 2) Reduce the number of partial products to two by using layers of full and half adders. 3) After the second step, we get two rows of sum and carry, add these rows with conventional adders.

As long as there are three or more rows with the same weight, add the following layers, 1) Take any three rows with the same weights and input them into a full adder. The result will be an output row of the same weight i.e. sum and an output row with a higher weight for each three input wires i.e. carry. 2) If there are two rows of the same weight left, input them into a half adder. 3) If there is just one row left, connect it to the next layer. The advantage of the Wallace tree multiplier is that there is small delay. A disadvantage is that it is complex to layout as it has irregular wires. [2]

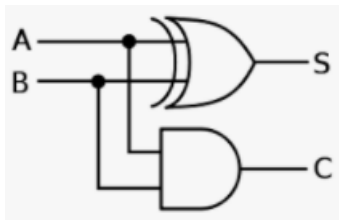
Booth multiplier

In the concept of multiplication it is often convenient to convert a two's – compliment number into a signed – digit form. The multiplication method using the multiplier in the signed – digit form is known as the Booths' method. [4] The Booth multiplication algorithm has low power complexity and it also consumes low power. When the radix number increases it becomes cumbersome. It is used for arithmetic operations for DSP applications such as filtering and Fourier transform. To achieve the high execution speed, parallel array multipliers are more widely used. [3]

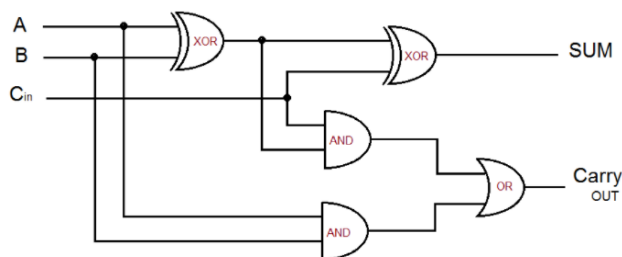
Design of 12 bit x 4 bit Braun multiplier

The 12 bit x 4 bit Braun multiplier will be made up of five different elements. The first element is the half adder, the second is the full adder, the third is line0 and the fourth is linex. These four elements, once developed as VHDL modules can be reused. The fifth element is the 12x4 multiplier.

Half adder – the half adder is a combinational arithmetic circuit that adds two input numbers (A and B) and produces the sum bit (S) and a carry bit (C) as the outputs.



Full adder – the full adder has three inputs and produces two outputs. There are two inputs in the form of A and B, the third is the carry input (C_{in}). The outputs are an output carry (C_{out}) and the sum (S) output.



Line 0 – line 0 contains a series of AND gates, the signals between the gates are set using the 'Port Map' function.

Line x – line x contains one half adder and eleven full adders. This line is reusable, the input and outputs are designated using the 'Port Map' function.

Multi12x4 – the multi12x4 module can be constructed when the previous four elements have been constructed. The other four modules are copied into the main multiplier module as components.

Port Map is again used to map the inputs and outputs to the correct connection point.

When each of the modules has been created in the VHDL format, the code must be synthesised. The synthesis process checks code syntax, analyses the hierarchy of the design, recognizes the functionality described, and ensures that your design is optimized for the device architecture you selected. The resulting netlist is saved to an NGC file for Xilinx Synthesis Technology (XST) [5]

On successful completion of the synthesis, a test bench must be created. The purpose of the test bench is to simulate the design, you need both the design under test (DUT) or unit under test (UUT) and the stimulus provided by the test bench. A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. A test bench can be as simple as a file with clock and input data or a more complicated file that includes error checking, file input and output, and conditional testing. [6]

Parallel multiplier layout

To aid with the design, a layout diagram was produced.

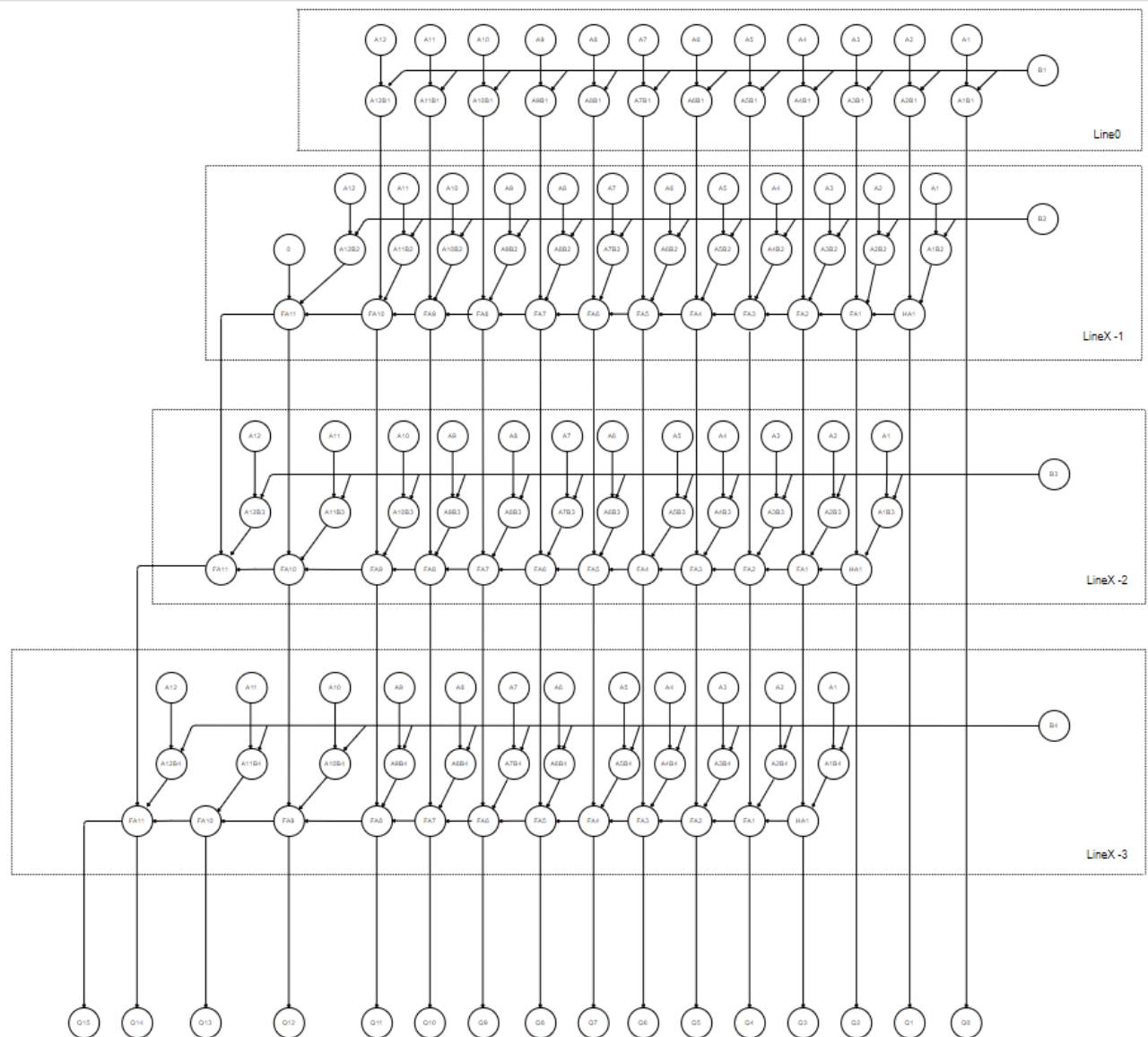
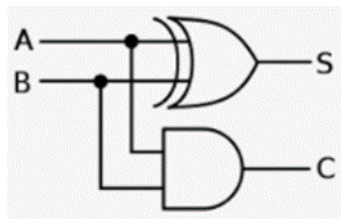


Figure 1 - Multiplier layout overview

Half adder

The half adder circuit is a combinational arithmetic circuit that adds two input numbers (A and B) and produces the sum bit (S) and a carry bit (C) as the outputs. It has no carry in input, so, it can only be used where there is no carry in input, and this will reduce hardware size slightly.



A	B	Sum	Carry out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 2 - Half adder circuit diagram and half adder truth table

The half adder VHDL code is detailed below. Line 32 details the name of the entity. Lines 33-36 designates the inputs and outputs. Line 39 starts the architecture body, this describes the relationship between the ports. Line 43 shows that the sum output is assigned an XOR gate with ports A and B inputs. Line 44 details that the carry output is assigned to AND gate with A and B inputs.

```

19 -----
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23 -- Uncomment the following library declaration if using
24 -- arithmetic functions with Signed or Unsigned values
25 --use IEEE.NUMERIC_STD.ALL;
26
27 -- Uncomment the following library declaration if instantiating
28 -- any Xilinx primitives in this code.
29 --library UNISIM;
30 --use UNISIM.VComponents.all;
31
32 entity halfadder is
33     Port ( A : in  STD_LOGIC;
34           B : in  STD_LOGIC;
35           Sum : out STD_LOGIC;
36           Carry : out STD_LOGIC);
37 end halfadder;
38
39 architecture Behavioral of halfadder is
40
41 begin
42
43     Sum <= A xor B;
44     Carry <= A and B;
45
46 end Behavioral;
47
48

```

Figure 3 - Half adder VHDL code

When the half adder VHDL logic has been synthesized a VHDL test bench was created. The purpose of the test bench is to simulate the design, you need both the design under test (DUT) or unit under test (UUT) and the stimulus provided by the test bench. A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. [6]

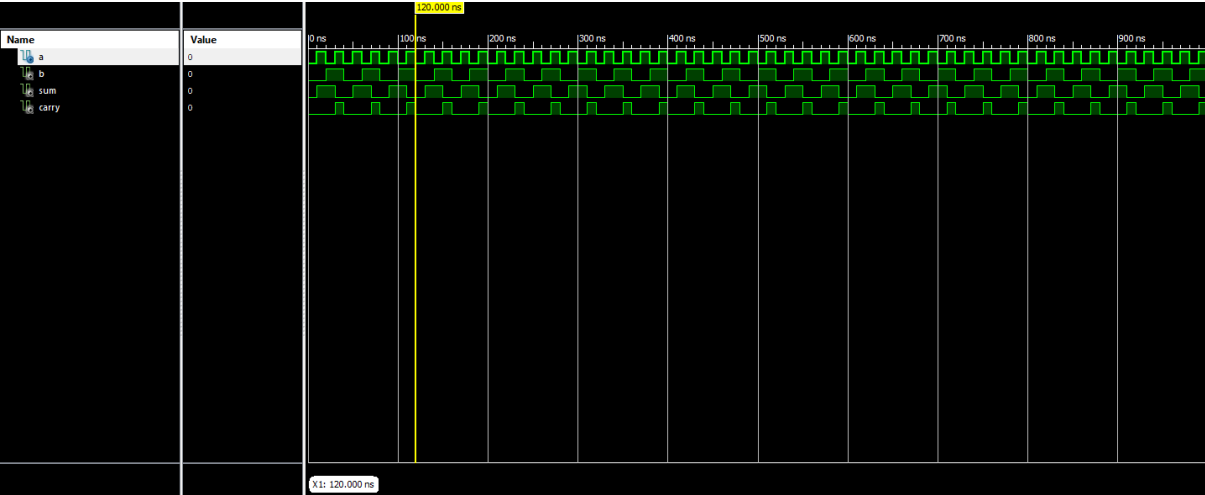
The code generates input stimuli to the input ports, the circuit will then react and generate outputs as required. In lines 70-89 of the test bench, the input stimuli is generated on inputs A and B using the stimproc*: process begin command, this command switches the input state from 0 to 1. In this case, A is switched every 10ns and B is switched every 20ns.

```
61 BEGIN
62   -- Instantiate the Unit Under Test (UUT)
63   uut: halfadder PORT MAP (
64     A => A,
65     B => B,
66     Sum => Sum,
67     Carry => Carry
68   );
69
70   -- Stimulus process
71   stim_proc0: process
72   begin
73     -- hold reset state for 100 ns.
74     wait for 10 ns;
75
76     A <= not A;
77
78   end process;
79
80   stim_proc1: process
81   begin
82     -- hold reset state for 100 ns.
83     wait for 20 ns;
84
85     B <= not B;
86
87   end process;
88
89 END;
90
```

Figure 4 - Half adder test bench

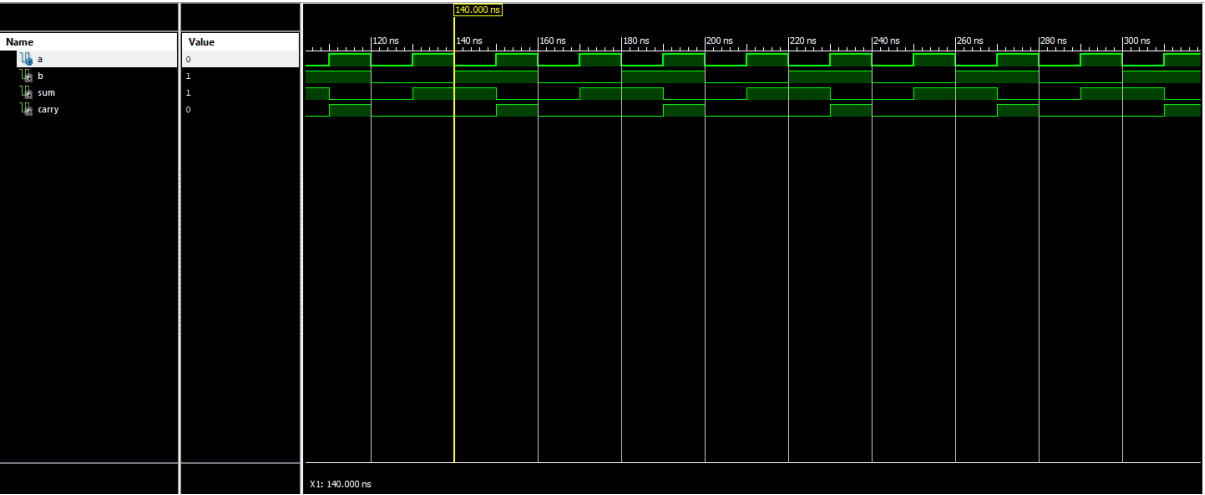
Half adder behavioural model

The images below show the waveforms generated when running the behavioural model simulator and the truth tables that correspond with each state are also displayed.



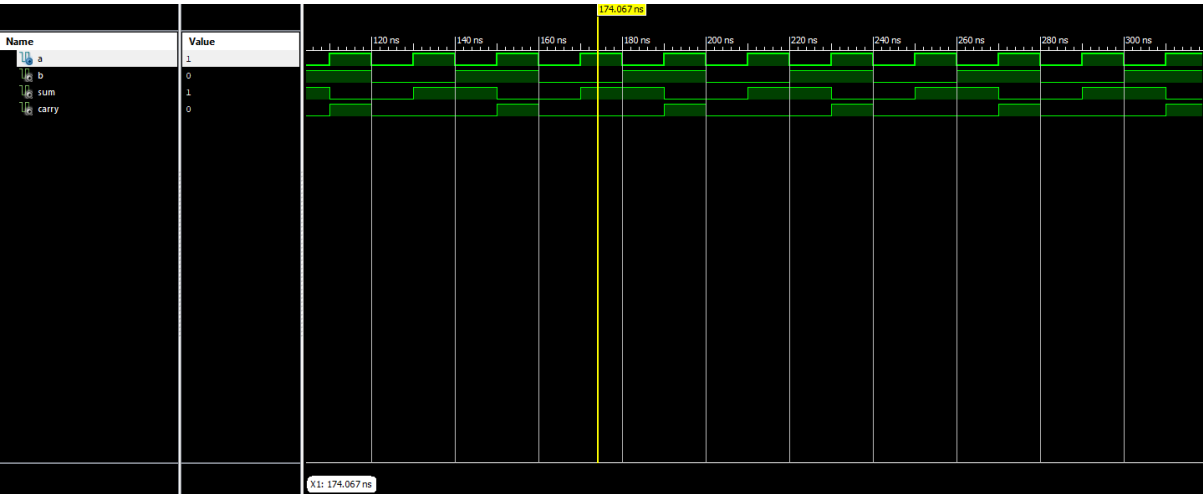
A	B	Sum	Carry out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 5 - Half adder, A=0, B=0, Sum=0, Carry=0



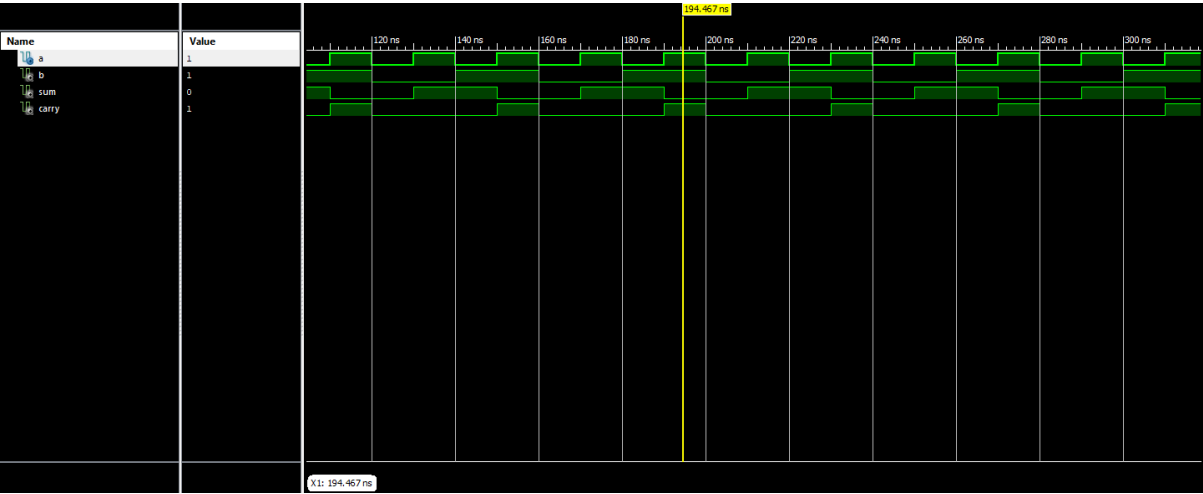
A	B	Sum	Carry out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 6 - Half adder, A=0, B=1 Sum=1, Carry=0



A	B	Sum	Carry out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 7 - Half adder, A=1, B=0, Sum=1, Carry=0



A	B	Sum	Carry out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 8 - Half adder, A=1, B=1, Sum=0, Carry=1

Full adder

The full adder has three inputs and produces two outputs. There are two inputs in the form of A and B, the third is the carry input (C_{in}). The outputs are an output carry (C_{out}) and the sum (S) output.

The full adder uses additional logic gates in the form of XOR, AND and OR gates.

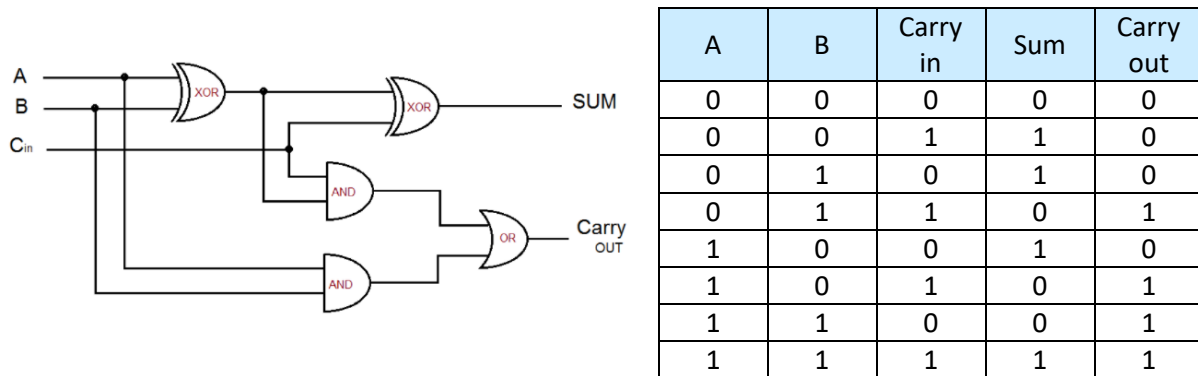


Figure 9 - Full adder circuit diagram and full adder truth table

The full adder VHDL code is detailed below. Line 32 details the name of the entity. Lines 33-37 designates the inputs and outputs. Lines 40-55 starts the architecture body, this describes the relationship between the ports and the interconnections between the logic gates.

```

32  entity fulladder is
33      Port ( A : in  STD_LOGIC;
34            B : in  STD_LOGIC;
35            C : in  STD_LOGIC;
36            Sum : out STD_LOGIC;
37            Carry : out STD_LOGIC);
38  end fulladder;
39
40  architecture Behavioral of fulladder is
41
42      signal AxB : STD_LOGIC;
43      signal AxBnC : STD_LOGIC;
44      signal AnB : STD_LOGIC;
45
46  begin
47      AxB <= A xor B;
48      AxBnC <= AxB and C;
49      AnB <= A and B;
50
51      Sum <= AxB xor C;
52      Carry <= AxBnC or AnB;
53
54
55  end Behavioral;
56

```

Figure 10 - Full adder VHDL code

When the full adder VHDL logic has been synthesized a VHDL test bench was created. The purpose of the test bench is to simulate the design, you need both the design under test (DUT) or unit under test (UUT) and the stimulus provided by the test bench. A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. [6]

The code generates input stimuli to the input ports, the circuit will then react and generate outputs as required. Lines 38-60 displays the architecture of the full adder and details the component inputs and outputs. Lines In lines 65-104 of the test bench, the input stimuli is generated on inputs A, B and C using the stimproc*: process begin command, this command switches the input state from 0 to 1. In this case, A is switched every 10ns, B is switched every 20ns and C every 40ns.

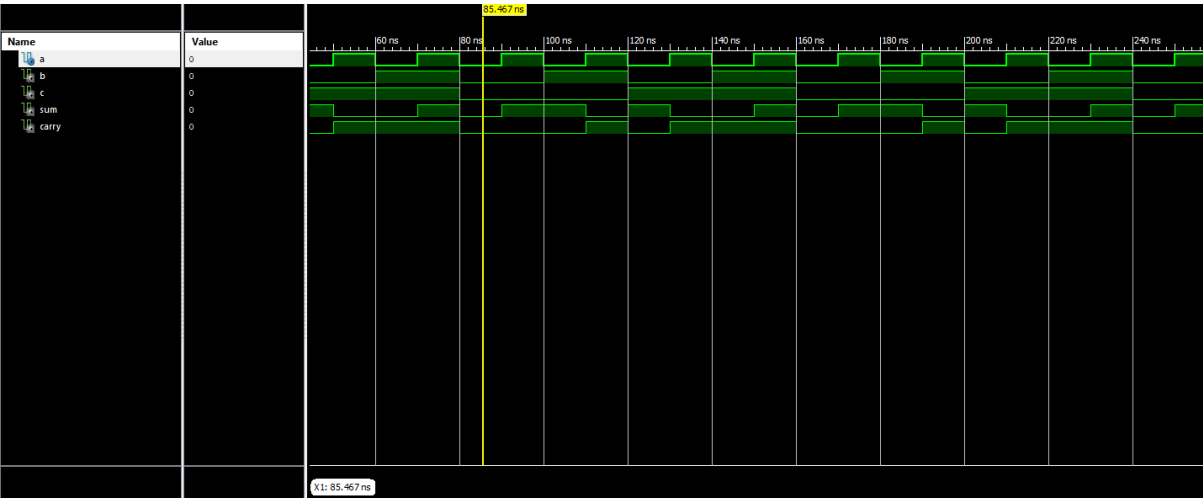
```
38 ARCHITECTURE behavior OF fulladder_TB IS
39
40     -- Component Declaration for the Unit Under Test (UUT)
41
42     COMPONENT fulladder
43     PORT(
44         A : IN  std_logic;
45         B : IN  std_logic;
46         C : IN  std_logic;
47         Sum : OUT std_logic;
48         Carry : OUT std_logic
49     );
50     END COMPONENT;
51
52
53     --Inputs
54     signal A : std_logic := '0';
55     signal B : std_logic := '0';
56     signal C : std_logic := '0';
57
58     --Outputs
59     signal Sum : std_logic;
60     signal Carry : std_logic;
61     -- No clocks detected in port list. Replace <clock> below with
62     -- appropriate port name
63
```

```
63
64
65 BEGIN
66
67     -- Instantiate the Unit Under Test (UUT)
68     uut: fulladder PORT MAP (
69         A => A,
70         B => B,
71         C => C,
72         Sum => Sum,
73         Carry => Carry
74     );
75
76
77
78     -- Stimulus process
79     stim_proc: process
80     begin
81         -- hold reset state for 10 ns.
82         wait for 10 ns;
83         A <= not A;
84
85     end process;
86
87     stim_proc1: process
88     begin
89         -- hold reset state for 20 ns.
90         wait for 20 ns;
91         B <= not B;
92
93     end process;
94
95     stim_proc2: process
96     begin
97         -- hold reset state for 40 ns.
98         wait for 40 ns;
99         C <= not C;
100
101     end process;
102
103
104 END;
105
```

Figure 11 - Full adder test bench

Full adder behavioural model

The images below show the waveforms generated when running the behavioural model simulator and the truth tables that correspond with each state are also displayed.



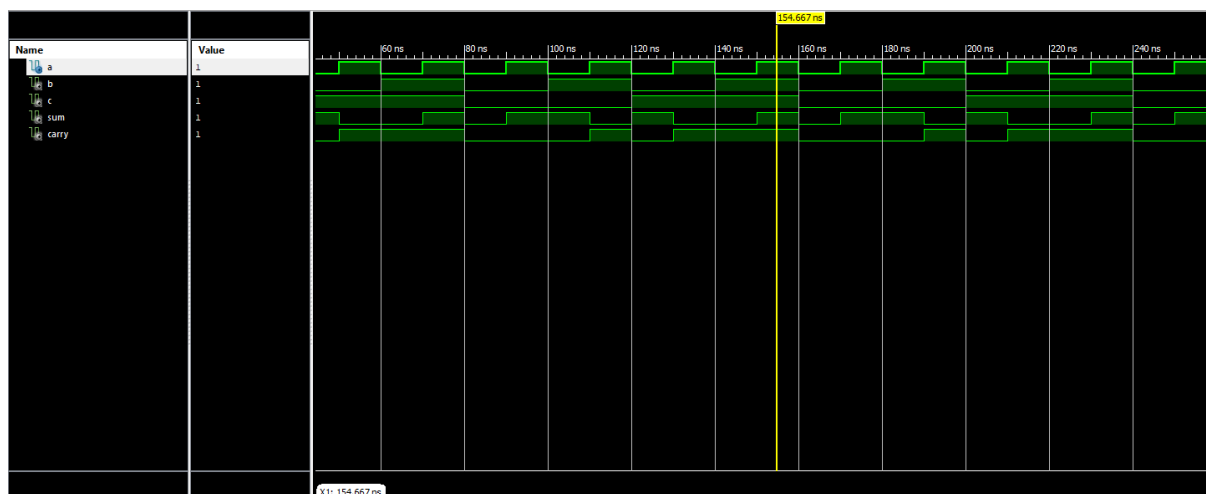
A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 12 - Full adder, A=0, B=0, Cin=0, Sum=0, Cout=0



A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 13 - Full adder, A=0, B=1, Cin=1, Sum=0, Cout=1



A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 14 - Full adder, A=1, B=1, Cin=1, Sum=1, Cout=1

The images show the correct functionality of the full adder. Although the images show only three of the states, all states have been tested for correct functionality as per the truth table.

Line 0

Line 0 is the top line of the multiplier. In this multiplier it consists of 12 AND gates. The image below shows on lines 32 – 61 detail the inputs (A1 to A12) and the outputs (Q1 to Q12). Lines 63 - 80 details the architecture body, this shows what outputs have been assigned to what inputs and the type of gate.

```
32 entity multil2x4_line0 is
33     Port ( A1 : in  STD_LOGIC;
34            A2 : in  STD_LOGIC;
35            A3 : in  STD_LOGIC;
36            A4 : in  STD_LOGIC;
37            A5 : in  STD_LOGIC;
38            A6 : in  STD_LOGIC;
39            A7 : in  STD_LOGIC;
40            A8 : in  STD_LOGIC;
41            A9 : in  STD_LOGIC;
42            A10 : in  STD_LOGIC;
43            A11 : in  STD_LOGIC;
44            A12 : in  STD_LOGIC;
45            B : in  STD_LOGIC;
46            Q1 : out STD_LOGIC;
47            Q2 : out STD_LOGIC;
48            Q3 : out STD_LOGIC;
49            Q4 : out STD_LOGIC;
50            Q5 : out STD_LOGIC;
51            Q6 : out STD_LOGIC;
52            Q7 : out STD_LOGIC;
53            Q8 : out STD_LOGIC;
54            Q9 : out STD_LOGIC;
55            Q10 : out STD_LOGIC;
56            Q11 : out STD_LOGIC;
57            Q12 : out STD_LOGIC);
58
59
60
61 end multil2x4_line0;
```

```
63 architecture Behavioral of multil2x4_line0 is
64
65 begin
66
67     Q1 <= A1 and B;
68     Q2 <= A2 and B;
69     Q3 <= A3 and B;
70     Q4 <= A4 and B;
71     Q5 <= A5 and B;
72     Q6 <= A6 and B;
73     Q7 <= A7 and B;
74     Q8 <= A8 and B;
75     Q9 <= A9 and B;
76     Q10 <= A10 and B;
77     Q11 <= A11 and B;
78     Q12 <= A12 and B;
79
80 end Behavioral;
81
```

Figure 15 - Line 0 VHDL code

When the line 0 VHDL logic has been synthesized a VHDL test bench was created. The purpose of the test bench is to simulate the design, you need both the design under test (DUT) or unit under test (UUT) and the stimulus provided by the test bench. A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. [6]

The code generates input stimuli to the input ports, the circuit will then react and generate outputs as required. Lines 42-70 displays the architecture of line 0 and details the component inputs and outputs. In lines 73-100 of the test bench, the input stimuli is generated on inputs A1 to A12 and B to 0 state, then, using the stimproc*: process begin command from line 137 onwards, this command switches the input state from 0 to 1 at the set times. Lines 109 – 134 assigns the inputs to an output through port mapping.

```
35 ENTITY line0_TB IS
36 END line0_TB;
37
38 ARCHITECTURE behavior OF line0_TB IS
39
40     -- Component Declaration for the Unit Under Test (UUT)
41
42     COMPONENT multil2x4_line0
43     PORT (
44         A1 : IN  std_logic;
45         A2 : IN  std_logic;
46         A3 : IN  std_logic;
47         A4 : IN  std_logic;
48         A5 : IN  std_logic;
49         A6 : IN  std_logic;
50         A7 : IN  std_logic;
51         A8 : IN  std_logic;
52         A9 : IN  std_logic;
53         A10 : IN std_logic;
54         A11 : IN std_logic;
55         A12 : IN std_logic;
56         B : IN  std_logic;
57         Q1 : OUT std_logic;
58         Q2 : OUT std_logic;
59         Q3 : OUT std_logic;
60         Q4 : OUT std_logic;
61         Q5 : OUT std_logic;
62         Q6 : OUT std_logic;
63         Q7 : OUT std_logic;
64         Q8 : OUT std_logic;
65         Q9 : OUT std_logic;
66         Q10 : OUT std_logic;
67         Q11 : OUT std_logic;
68         Q12 : OUT std_logic;
69     );
70     END COMPONENT;
```

```
71
72
73  --Inputs
74  signal A1 : std_logic := '0';
75  signal A2 : std_logic := '0';
76  signal A3 : std_logic := '0';
77  signal A4 : std_logic := '0';
78  signal A5 : std_logic := '0';
79  signal A6 : std_logic := '0';
80  signal A7 : std_logic := '0';
81  signal A8 : std_logic := '0';
82  signal A9 : std_logic := '0';
83  signal A10 : std_logic := '0';
84  signal A11 : std_logic := '0';
85  signal A12 : std_logic := '0';
86  signal B : std_logic := '0';
87
88  --Outputs
89  signal Q1 : std_logic;
90  signal Q2 : std_logic;
91  signal Q3 : std_logic;
92  signal Q4 : std_logic;
93  signal Q5 : std_logic;
94  signal Q6 : std_logic;
95  signal Q7 : std_logic;
96  signal Q8 : std_logic;
97  signal Q9 : std_logic;
98  signal Q10 : std_logic;
99  signal Q11 : std_logic;
100 signal Q12 : std_logic;
101 -- No clocks detected in port list. Replace <clock> below with
102 -- appropriate port name
103
104
105
```

```
106 BEGIN
107
108 -- Instantiate the Unit Under Test (UUT)
109 uut: multil2x4_line0 PORT MAP (
110     A1 => A1,
111     A2 => A2,
112     A3 => A3,
113     A4 => A4,
114     A5 => A5,
115     A6 => A6,
116     A7 => A7,
117     A8 => A8,
118     A9 => A9,
119     A10 => A10,
120     A11 => A11,
121     A12 => A12,
122     B => B,
123     Q1 => Q1,
124     Q2 => Q2,
125     Q3 => Q3,
126     Q4 => Q4,
127     Q5 => Q5,
128     Q6 => Q6,
129     Q7 => Q7,
130     Q8 => Q8,
131     Q9 => Q9,
132     Q10 => Q10,
133     Q11 => Q11,
134     Q12 => Q12
135 );
136
137 -- Stimulus process
138 stim_proc: process
139 begin
140     -- hold reset state for 100 ns.
141     wait for 20 ns;
142
143     A1 <= not A1;
144
145 end process;
146
```

```
147 stim_proc1: process
148 begin
149     -- hold reset state for 100 ns.
150     wait for 40 ns;
151
152     A2 <= not A2;
153
154 end process;
155
156 stim_proc2: process
157 begin
158     -- hold reset state for 100 ns.
159     wait for 80 ns;
160
161     A3 <= not A3;
162
163 end process;
164
165 stim_proc3: process
166 begin
167     -- hold reset state for 100 ns.
168     wait for 160 ns;
169
170     A4 <= not A4;
171
172 end process;
173
174 stim_proc4: process
175 begin
176     -- hold reset state for 100 ns.
177     wait for 320 ns;
178
179     A5 <= not A5;
180
181 end process;
182
183 stim_proc5: process
184 begin
185     -- hold reset state for 100 ns.
186     wait for 640 ns;
187
188     A6 <= not A6;
189
190 end process;
```

```
191
192 stim_proc6: process
193 begin
194     -- hold reset state for 100 ns.
195     wait for 1280 ns;
196
197     A7 <= not A7;
198
199 end process;
200
201 stim_proc7: process
202 begin
203     -- hold reset state for 100 ns.
204     wait for 2560 ns;
205
206     A8 <= not A8;
207
208 end process;
209
210 stim_proc8: process
211 begin
212     -- hold reset state for 100 ns.
213     wait for 5120 ns;
214
215     A9 <= not A9;
216
217 end process;
218
219 stim_proc9: process
220 begin
221     -- hold reset state for 100 ns.
222     wait for 10240 ns;
223
224     A10 <= not A10;
225
226 end process;
227
228 stim_procl0: process
229 begin
230     -- hold reset state for 100 ns.
231     wait for 20480 ns;
232
233     A11 <= not A11;
234
235 end process;
236
237 stim_procl1: process
238 begin
239     -- hold reset state for 100 ns.
240     wait for 40960 ns;
241
242     A12 <= not A12;
243
244 end process;
245
246 stim_procl2: process
247 begin
248     -- hold reset state for 100 ns.
249     wait for 81920 ns;
250
251     B <= not B;
252
253 end process;
254
255
256 END;
```

Figure 16 - Line 0 test bench

Line 0 behavioural model

The images below show the waveforms generated when running the behavioural model simulator. When using the simulation, when B is 0, the output is 0, regardless of the input value. When B is 1, the output value is equal to the input value. The inputs and outputs have been made into virtual busses and the radix has been changed to unsigned decimal for ease of interoperating results.

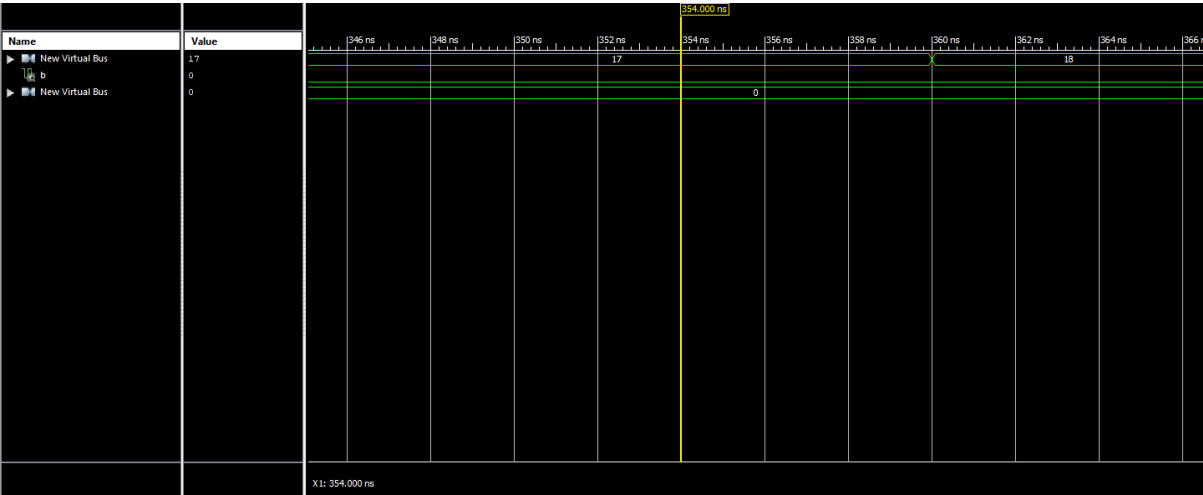


Figure 17 - Line 0 when B is 0

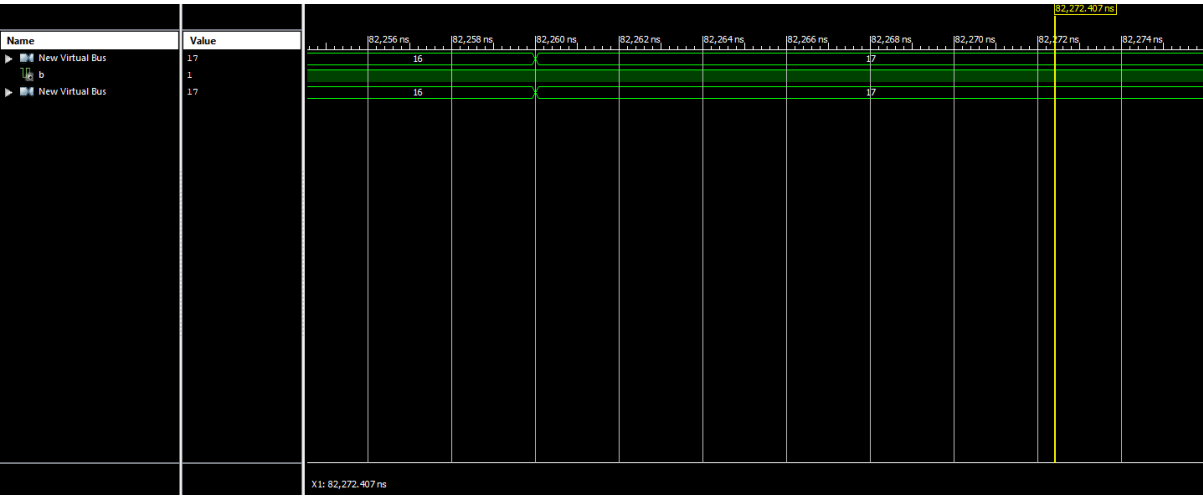


Figure 18 - Line 0 when B is 1

Line x

The line x function combines a half adder and the full adders. This is achieved by copying the adder modules into the line x module as components. This module uses 1 half adder and 11 full adders.

There are internal and external connections made in the line x module. Lines 32 – 71 designates the input and output signals. Lines 75 – 88 show the half adder and full adder references. Lines 90 – 113 show signal connections. Lines 117 – 179 shows the gate sequences and then the required port mapping.

```

32 entity multil2x4_linex is
33     Port ( A1 : in  STD_LOGIC;
34           A2 : in  STD_LOGIC;
35           A3 : in  STD_LOGIC;
36           A4 : in  STD_LOGIC;
37           A5 : in  STD_LOGIC;
38           A6 : in  STD_LOGIC;
39           A7 : in  STD_LOGIC;
40           A8 : in  STD_LOGIC;
41           A9 : in  STD_LOGIC;
42           A10 : in  STD_LOGIC;
43           A11 : in  STD_LOGIC;
44           A12 : in  STD_LOGIC;
45           B : in  STD_LOGIC;
46           P1 : in  STD_LOGIC;
47           P2 : in  STD_LOGIC;
48           P3 : in  STD_LOGIC;
49           P4 : in  STD_LOGIC;
50           P5 : in  STD_LOGIC;
51           P6 : in  STD_LOGIC;
52           P7 : in  STD_LOGIC;
53           P8 : in  STD_LOGIC;
54           P9 : in  STD_LOGIC;
55           P10 : in  STD_LOGIC;
56           P11 : in  STD_LOGIC;
57           P12 : in  STD_LOGIC;
58           Q1 : out  STD_LOGIC;
59           Q2 : out  STD_LOGIC;
60           Q3 : out  STD_LOGIC;
61           Q4 : out  STD_LOGIC;
62           Q5 : out  STD_LOGIC;
63           Q6 : out  STD_LOGIC;
64           Q7 : out  STD_LOGIC;
65           Q8 : out  STD_LOGIC;
66           Q9 : out  STD_LOGIC;
67           Q10 : out  STD_LOGIC;
68           Q11 : out  STD_LOGIC;
69           Q12 : out  STD_LOGIC;
70           Q13 : out  STD_LOGIC);
71 end multil2x4_linex;

```

```
72
73 architecture Behavioral of multil2x4_lindex is
74
75     COMPONENT halfadder is
76         Port ( A : in  STD_LOGIC;
77               B : in  STD_LOGIC;
78               Sum : out STD_LOGIC;
79               Carry : out STD_LOGIC);
80     END COMPONENT;
81
82     COMPONENT fulladder is
83         Port ( A : in  STD_LOGIC;
84               B : in  STD_LOGIC;
85               C : in  STD_LOGIC;
86               Sum : out STD_LOGIC;
87               Carry : out STD_LOGIC);
88     END COMPONENT;
89
90     signal s0 : STD_LOGIC;
91     signal s1 : STD_LOGIC;
92     signal s2 : STD_LOGIC;
93     signal s3 : STD_LOGIC;
94     signal s4 : STD_LOGIC;
95     signal s5 : STD_LOGIC;
96     signal s6 : STD_LOGIC;
97     signal s7 : STD_LOGIC;
98     signal s8 : STD_LOGIC;
99     signal s9 : STD_LOGIC;
100    signal s10 : STD_LOGIC;
101    signal s11 : STD_LOGIC;
102
103    signal c1 : STD_LOGIC;
104    signal c2 : STD_LOGIC;
105    signal c3 : STD_LOGIC;
106    signal c4 : STD_LOGIC;
107    signal c5 : STD_LOGIC;
108    signal c6 : STD_LOGIC;
109    signal c7 : STD_LOGIC;
110    signal c8 : STD_LOGIC;
111    signal c9 : STD_LOGIC;
112    signal c10 : STD_LOGIC;
113    signal c11 : STD_LOGIC;
114
```

```

115 begin
116
117 -- and gate sequence.
118 s0 <= A1 and B;
119 s1 <= A2 and B;
120 s2 <= A3 and B;
121 s3 <= A4 and B;
122 s4 <= A5 and B;
123 s5 <= A6 and B;
124 s6 <= A7 and B;
125 s7 <= A8 and B;
126 s8 <= A9 and B;
127 s9 <= A10 and B;
128 s10 <= A11 and B;
129 s11 <= A12 and B;
130
131 HA1 : halfadder
132 Port Map
133 ( s0 , P1 , Q1 , c1 );
134
135 FA1 : fulladder
136 Port Map
137 ( s1 , P2 , c1 , Q2 , c2 );
138
139 FA2 : fulladder
140 Port Map
141 ( s2 , P3 , c2 , Q3 , c3 );
142
143 FA3 : fulladder
144 Port Map
145 ( s3 , P4 , c3 , Q4 , c4 );
146
147 FA4 : fulladder
148 Port Map
149 ( s4 , P5 , c4 , Q5 , c5 );
150
151 FA5 : fulladder
152 Port Map
153 ( s5 , P6 , c5 , Q6 , c6 );
154
155 FA6 : fulladder
156 Port Map
157 ( s6 , P7 , c6 , Q7 , c7 );
158
159 FA7 : fulladder
160 Port Map
161 ( s7 , P8 , c7 , Q8 , c8 );
162
163 FA8 : fulladder
164 Port Map
165 ( s8 , P9 , c8 , Q9 , c9 );
166
167 FA9 : fulladder
168 Port Map
169 ( s9 , P10 , c9 , Q10 , c10 );
170
171 FA10 : fulladder
172 Port Map
173 ( s10 , P11 , c10 , Q11 , c11 );
174
175 FA11 : fulladder
176 Port Map
177 ( s11 , P12 , c11 , Q12 , Q13 );
178
179 end Behavioral;

```

Figure 19 - Line x VHDL code

The image below shows the linex hierarchy. This details the components required below the main module.

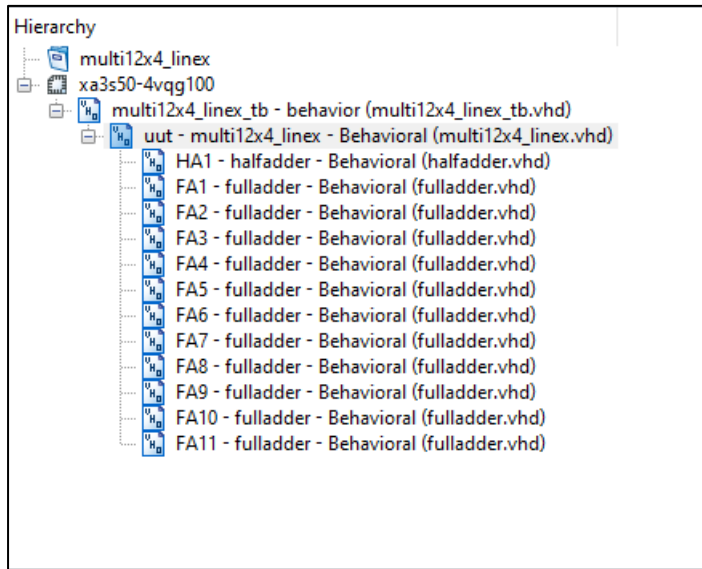


Figure 20 – Line x hierarchy

When the line x VHDL logic has been synthesized a VHDL test bench was created. The purpose of the test bench is to simulate the design, you need both the design under test (DUT) or unit under test (UUT) and the stimulus provided by the test bench. A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. [6]

The code generates input stimuli to the input ports, the circuit will then react and generate outputs as required. Lines 38 - 81 displays the architecture of linex and details the component inputs and outputs. In lines 86 - 126 of the test bench designate the input and output logic type. Lines 130 – 172 assign the inputs and outputs that are under test. Using the stimproc*: process begin command from line 174 onwards, this command switches the input state from 0 to 1 at the times set at each stimproc.

```
35 ENTITY multil2x4_linex_tb IS
36 END multil2x4_linex_tb;
37
38 ARCHITECTURE behavior OF multil2x4_linex_tb IS
39
40     -- Component Declaration for the Unit Under Test (UUT)
41
42     COMPONENT multil2x4_linex
43     PORT(
44         A1 : IN  std_logic;
45         A2 : IN  std_logic;
46         A3 : IN  std_logic;
47         A4 : IN  std_logic;
48         A5 : IN  std_logic;
49         A6 : IN  std_logic;
50         A7 : IN  std_logic;
51         A8 : IN  std_logic;
52         A9 : IN  std_logic;
53         A10 : IN std_logic;
54         A11 : IN std_logic;
55         A12 : IN std_logic;
56         B : IN  std_logic;
57         P1 : IN  std_logic;
58         P2 : IN  std_logic;
59         P3 : IN  std_logic;
60         P4 : IN  std_logic;
61         P5 : IN  std_logic;
62         P6 : IN  std_logic;
63         P7 : IN  std_logic;
64         P8 : IN  std_logic;
65         P9 : IN  std_logic;
66         P10 : IN std_logic;
67         P11 : IN std_logic;
68         P12 : IN std_logic;
69         Q1 : OUT std_logic;
70         Q2 : OUT std_logic;
71         Q3 : OUT std_logic;
72         Q4 : OUT std_logic;
73         Q5 : OUT std_logic;
74         Q6 : OUT std_logic;
75         Q7 : OUT std_logic;
76         Q8 : OUT std_logic;
77         Q9 : OUT std_logic;
78         Q10 : OUT std_logic;
79         Q11 : OUT std_logic;
80         Q12 : OUT std_logic;
81         Q13 : OUT std_logic;
```

```
82     );
83     END COMPONENT;
84
85
86     --Inputs
87     signal A1 : std_logic := '0';
88     signal A2 : std_logic := '0';
89     signal A3 : std_logic := '0';
90     signal A4 : std_logic := '0';
91     signal A5 : std_logic := '0';
92     signal A6 : std_logic := '0';
93     signal A7 : std_logic := '0';
94     signal A8 : std_logic := '0';
95     signal A9 : std_logic := '0';
96     signal A10 : std_logic := '0';
97     signal A11 : std_logic := '0';
98     signal A12 : std_logic := '0';
99     signal B : std_logic := '0';
100    signal P1 : std_logic := '0';
101    signal P2 : std_logic := '0';
102    signal P3 : std_logic := '0';
103    signal P4 : std_logic := '0';
104    signal P5 : std_logic := '0';
105    signal P6 : std_logic := '0';
106    signal P7 : std_logic := '0';
107    signal P8 : std_logic := '0';
108    signal P9 : std_logic := '0';
109    signal P10 : std_logic := '0';
110    signal P11 : std_logic := '0';
111    signal P12 : std_logic := '0';
112
113    --Outputs
114    signal Q1 : std_logic;
115    signal Q2 : std_logic;
116    signal Q3 : std_logic;
117    signal Q4 : std_logic;
118    signal Q5 : std_logic;
119    signal Q6 : std_logic;
120    signal Q7 : std_logic;
121    signal Q8 : std_logic;
122    signal Q9 : std_logic;
123    signal Q10 : std_logic;
124    signal Q11 : std_logic;
125    signal Q12 : std_logic;
126    signal Q13 : std_logic;
127    -- No clocks detected in port list. Replace <clock> below with
128    -- appropriate port name
```

129		174	-- Stimulus process
130	BEGIN	175	stim_proc: process
131		176	begin
132	-- Instantiate the Unit Under Test (UUT)	177	-- hold reset state for 100 ns.
133	uut: multil2x4_linex PORT MAP (	178	wait for 10 ns;
134	A1 => A1,	179	A1 <= not A1;
135	A2 => A2,	180	end process;
136	A3 => A3,	181	
137	A4 => A4,	182	-- Stimulus process
138	A5 => A5,	183	stim_proc2: process
139	A6 => A6,	184	begin
140	A7 => A7,	185	-- hold reset state for 100 ns.
141	A8 => A8,	186	wait for 20 ns;
142	A9 => A9,	187	A2 <= not A2;
143	A10 => A10,	188	end process;
144	A11 => A11,	189	
145	A12 => A12,	190	-- Stimulus process
146	B => B,	191	stim_proc3: process
147	P1 => P1,	192	begin
148	P2 => P2,	193	-- hold reset state for 100 ns.
149	P3 => P3,	194	wait for 40 ns;
150	P4 => P4,	195	A3 <= not A3;
151	P5 => P5,	196	end process;
152	P6 => P6,	197	
153	P7 => P7,	198	-- Stimulus process
154	P8 => P8,	199	stim_pro4: process
155	P9 => P9,	200	begin
156	P10 => P10,	201	-- hold reset state for 100 ns.
157	P11 => P11,	202	wait for 80 ns;
158	P12 => P12,	203	A4 <= not A4;
159	Q1 => Q1,	204	end process;
160	Q2 => Q2,	205	
161	Q3 => Q3,	206	-- Stimulus process
162	Q4 => Q4,	207	stim_proc5: process
163	Q5 => Q5,	208	begin
164	Q6 => Q6,	209	-- hold reset state for 100 ns.
165	Q7 => Q7,	210	wait for 160 ns;
166	Q8 => Q8,	211	A5 <= not A5;
167	Q9 => Q9,	212	end process;
168	Q10 => Q10,		
169	Q11 => Q11,		
170	Q12 => Q12,		
171	Q13 => Q13		
172	);		
173			

213		253	
214	-- Stimulus process	254	-- Stimulus process
215	stim_proc6: process	255	stim_procl1: process
216	begin	256	begin
217	-- hold reset state for 100 ns.	257	-- hold reset state for 100 ns.
218	wait for 320 ns;	258	wait for 10240 ns;
219	A6 <= not A6;	259	All <= not All;
220	end process;	260	end process;
221		261	
222	-- Stimulus process	262	-- Stimulus process
223	stim_proc7: process	263	stim_procl2: process
224	begin	264	begin
225	-- hold reset state for 100 ns.	265	-- hold reset state for 100 ns.
226	wait for 640 ns;	266	wait for 20480 ns;
227	A7 <= not A7;	267	Al2 <= not Al2;
228	end process;	268	end process;
229		269	
230	-- Stimulus process	270	-- Stimulus process
231	stim_proc8: process	271	stim_procl3: process
232	begin	272	begin
233	-- hold reset state for 100 ns.	273	-- hold reset state for 100 ns.
234	wait for 1280 ns;	274	wait for 40960 ns;
235	A8 <= not A8;	275	P1 <= not P1;
236	end process;	276	end process;
237		277	
238	-- Stimulus process	278	-- Stimulus process
239	stim_proc9: process	279	stim_procl4: process
240	begin	280	begin
241	-- hold reset state for 100 ns.	281	-- hold reset state for 100 ns.
242	wait for 2560 ns;	282	wait for 81920 ns;
243	A9 <= not A9;	283	P2 <= not P2;
244	end process;	284	end process;
245		285	
246	-- Stimulus process	286	-- Stimulus process
247	stim_procl0: process	287	stim_procl5: process
248	begin	288	begin
249	-- hold reset state for 100 ns.	289	-- hold reset state for 100 ns.
250	wait for 5120 ns;	290	wait for 163840 ns;
251	Al0 <= not Al0;	291	P3 <= not P3;
252	end process;	292	end process;

293		333	
294	-- Stimulus process	334	-- Stimulus process
295	stim_procl6: process	335	stim_proc21: process
296	begin	336	begin
297	-- hold reset state for 100 ns.	337	-- hold reset state for 100 ns.
298	wait for 327680 ns;	338	wait for 10485760 ns;
299	P4 <= not P4;	339	P9 <= not P9;
300	end process;	340	end process;
301		341	
302	-- Stimulus process	342	-- Stimulus process
303	stim_procl7: process	343	stim_proc22: process
304	begin	344	begin
305	-- hold reset state for 100 ns.	345	-- hold reset state for 100 ns.
306	wait for 655360 ns;	346	wait for 20971520 ns;
307	P5 <= not P5;	347	P10 <= not P10;
308	end process;	348	end process;
309		349	
310	-- Stimulus process	350	-- Stimulus process
311	stim_procl8: process	351	stim_proc23: process
312	begin	352	begin
313	-- hold reset state for 100 ns.	353	-- hold reset state for 100 ns.
314	wait for 1310720 ns;	354	wait for 41943040 ns;
315	P6 <= not P6;	355	P11 <= not P11;
316	end process;	356	end process;
317		357	
318	-- Stimulus process	358	-- Stimulus process
319	stim_procl9: process	359	stim_proc24: process
320	begin	360	begin
321	-- hold reset state for 100 ns.	361	-- hold reset state for 100 ns.
322	wait for 2621440 ns;	362	wait for 83886080 ns;
323	P7 <= not P7;	363	P12 <= not P12;
324	end process;	364	end process;
325		365	
326	-- Stimulus process	366	-- Stimulus process
327	stim_proc20: process	367	stim_proc25: process
328	begin	368	begin
329	-- hold reset state for 100 ns.	369	-- hold reset state for 100 ns.
330	wait for 5242880 ns;	370	wait for 167772160 ns;
331	P8 <= not P8;	371	B <= not B;
332	end process;	372	end process;
		373	
		374	END;

Figure 21 – Line x test bench

Line x behavioural model

To check the operation of linex is correct, the behavioural model was run. This proved ok as when B is 0, the input is not added to the carry signal and so not added to the output. When B is 1, the input is added to the carry signal and is included in the output.

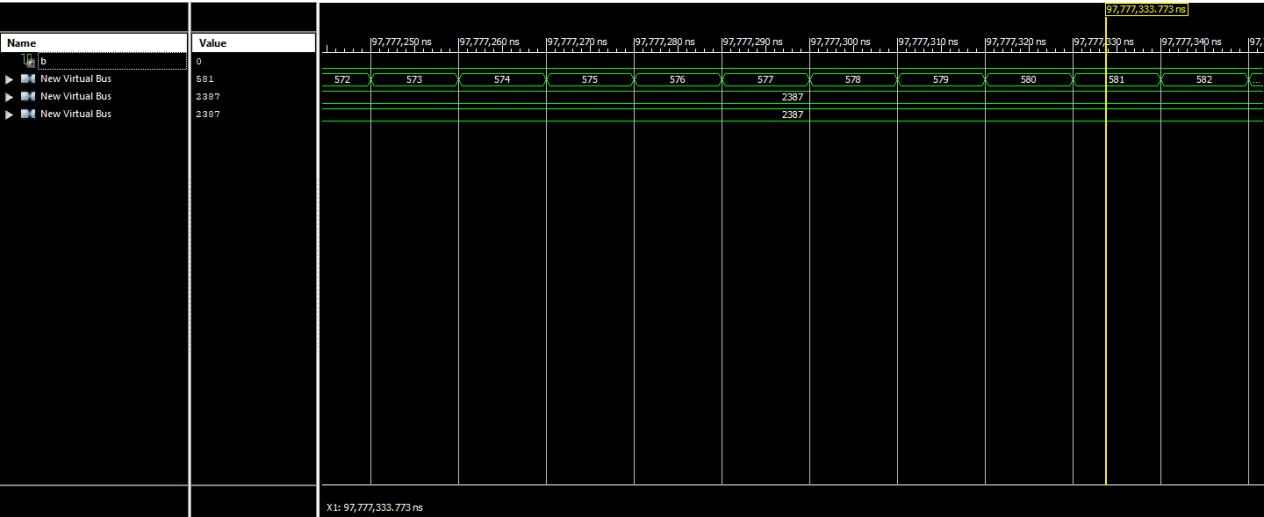


Figure 22 - Line x when *b* = 0

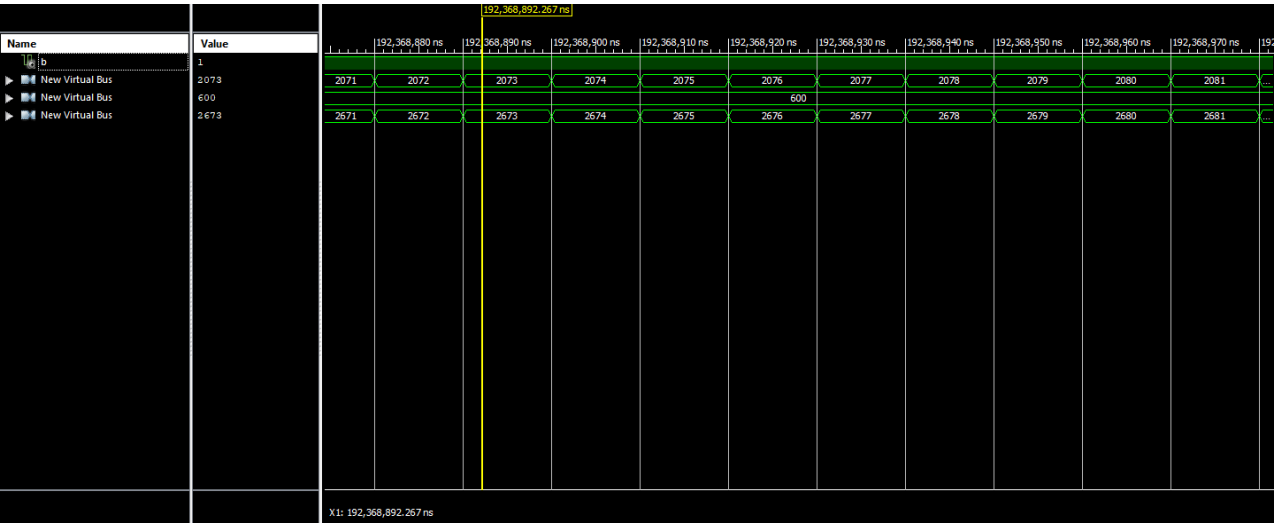


Figure 23 - Line x when *b* = 1

This proves that linex is functioning as required.

12x4 multiplier

The last of the VHDL modules combines all of the previous modules to create the full multiplier. The half adder, full adder, line0 and linex will all be added to the final multiplier as components. Using port mapping, the components within this module can be used multiple times.

```
32 entity multil2x4_full is
33     Port ( A1 : in  STD_LOGIC;
34           A2 : in  STD_LOGIC;
35           A3 : in  STD_LOGIC;
36           A4 : in  STD_LOGIC;
37           A5 : in  STD_LOGIC;
38           A6 : in  STD_LOGIC;
39           A7 : in  STD_LOGIC;
40           A8 : in  STD_LOGIC;
41           A9 : in  STD_LOGIC;
42           A10 : in  STD_LOGIC;
43           A11 : in  STD_LOGIC;
44           A12 : in  STD_LOGIC;
45           B1 : in  STD_LOGIC;
46           B2 : in  STD_LOGIC;
47           B3 : in  STD_LOGIC;
48           B4 : in  STD_LOGIC;
49           Q0 : out  STD_LOGIC;
50           Q1 : out  STD_LOGIC;
51           Q2 : out  STD_LOGIC;
52           Q3 : out  STD_LOGIC;
53           Q4 : out  STD_LOGIC;
54           Q5 : out  STD_LOGIC;
55           Q6 : out  STD_LOGIC;
56           Q7 : out  STD_LOGIC;
57           Q8 : out  STD_LOGIC;
58           Q9 : out  STD_LOGIC;
59           Q10 : out  STD_LOGIC;
60           Q11 : out  STD_LOGIC;
61           Q12 : out  STD_LOGIC;
62           Q13 : out  STD_LOGIC;
63           Q14 : out  STD_LOGIC;
64           Q15 : out  STD_LOGIC);
65 end multil2x4_full;
```

Figure 24 - Multi12x4_full multiplier entity

```
69  COMPONENT multil2x4_line0 is
70  Port ( A1 : in  STD_LOGIC;
71         A2 : in  STD_LOGIC;
72         A3 : in  STD_LOGIC;
73         A4 : in  STD_LOGIC;
74         A5 : in  STD_LOGIC;
75         A6 : in  STD_LOGIC;
76         A7 : in  STD_LOGIC;
77         A8 : in  STD_LOGIC;
78         A9 : in  STD_LOGIC;
79         A10 : in  STD_LOGIC;
80         A11 : in  STD_LOGIC;
81         A12 : in  STD_LOGIC;
82         B : in  STD_LOGIC;
83         Q1 : out  STD_LOGIC;
84         Q2 : out  STD_LOGIC;
85         Q3 : out  STD_LOGIC;
86         Q4 : out  STD_LOGIC;
87         Q5 : out  STD_LOGIC;
88         Q6 : out  STD_LOGIC;
89         Q7 : out  STD_LOGIC;
90         Q8 : out  STD_LOGIC;
91         Q9 : out  STD_LOGIC;
92         Q10 : out  STD_LOGIC;
93         Q11 : out  STD_LOGIC;
94         Q12 : out  STD_LOGIC);
95
96
97  END COMPONENT;
```

Figure 25 - Line 0 component

```
99      COMPONENT multil2x4_lineX is
100      Port ( A1 : in  STD_LOGIC;
101              A2 : in  STD_LOGIC;
102              A3 : in  STD_LOGIC;
103              A4 : in  STD_LOGIC;
104              A5 : in  STD_LOGIC;
105              A6 : in  STD_LOGIC;
106              A7 : in  STD_LOGIC;
107              A8 : in  STD_LOGIC;
108              A9 : in  STD_LOGIC;
109              A10 : in  STD_LOGIC;
110              A11 : in  STD_LOGIC;
111              A12 : in  STD_LOGIC;
112              B : in  STD_LOGIC;
113              P1 : in  STD_LOGIC;
114              P2 : in  STD_LOGIC;
115              P3 : in  STD_LOGIC;
116              P4 : in  STD_LOGIC;
117              P5 : in  STD_LOGIC;
118              P6 : in  STD_LOGIC;
119              P7 : in  STD_LOGIC;
120              P8 : in  STD_LOGIC;
121              P9 : in  STD_LOGIC;
122              P10 : in  STD_LOGIC;
123              P11 : in  STD_LOGIC;
124              P12 : in  STD_LOGIC;
125              Q1 : out STD_LOGIC;
126              Q2 : out STD_LOGIC;
127              Q3 : out STD_LOGIC;
128              Q4 : out STD_LOGIC;
129              Q5 : out STD_LOGIC;
130              Q6 : out STD_LOGIC;
131              Q7 : out STD_LOGIC;
132              Q8 : out STD_LOGIC;
133              Q9 : out STD_LOGIC;
134              Q10 : out STD_LOGIC;
135              Q11 : out STD_LOGIC;
136              Q12 : out STD_LOGIC;
137              Q13 : out STD_LOGIC);
138      END COMPONENT;
```

Figure 26 - Line x component

```

140     signal p1_1 : STD_LOGIC;
141     signal p1_2 : STD_LOGIC;
142     signal p1_3 : STD_LOGIC;
143     signal p1_4 : STD_LOGIC;
144     signal p1_5 : STD_LOGIC;
145     signal p1_6 : STD_LOGIC;
146     signal p1_7 : STD_LOGIC;
147     signal p1_8 : STD_LOGIC;
148     signal p1_9 : STD_LOGIC;
149     signal p1_10 : STD_LOGIC;
150     signal p1_11 : STD_LOGIC;
151
152     signal p2_1 : STD_LOGIC;
153     signal p2_2 : STD_LOGIC;
154     signal p2_3 : STD_LOGIC;
155     signal p2_4 : STD_LOGIC;
156     signal p2_5 : STD_LOGIC;
157     signal p2_6 : STD_LOGIC;
158     signal p2_7 : STD_LOGIC;
159     signal p2_8 : STD_LOGIC;
160     signal p2_9 : STD_LOGIC;
161     signal p2_10 : STD_LOGIC;
162     signal p2_11 : STD_LOGIC;
163     signal p2_12 : STD_LOGIC;
164
165     signal p3_1 : STD_LOGIC;
166     signal p3_2 : STD_LOGIC;
167     signal p3_3 : STD_LOGIC;
168     signal p3_4 : STD_LOGIC;
169     signal p3_5 : STD_LOGIC;
170     signal p3_6 : STD_LOGIC;
171     signal p3_7 : STD_LOGIC;
172     signal p3_8 : STD_LOGIC;
173     signal p3_9 : STD_LOGIC;
174     signal p3_10 : STD_LOGIC;
175     signal p3_11 : STD_LOGIC;
176     signal p3_12 : STD_LOGIC;
177

```

Figure 27 - Multiplier assigned signals

```

179 begin
180     line1 : multilDet_line0
181     Port Map
182     ( A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, B1, Q0, p1_1, p1_2, p1_3, p1_4, p1_5, p1_6, p1_7, p1_8, p1_9, p1_10, p1_11 );
183
184     line2 : multilDet_line1
185     Port Map
186     ( A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, B2, p1_1, p1_2, p1_3, p1_4, p1_5, p1_6, p1_7, p1_8, p1_9, p1_10, p1_11, '0', Q1, p2_1, p2_2, p2_3, p2_4, p2_5, p2_6, p2_7, p2_8, p2_9, p2_10, p2_11, p2_12 );
187
188     line3 : multilDet_line2
189     Port Map
190     ( A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, B3, p2_1, p2_2, p2_3, p2_4, p2_5, p2_6, p2_7, p2_8, p2_9, p2_10, p2_11, p2_12, Q2, p3_1, p3_2, p3_3, p3_4, p3_5, p3_6, p3_7, p3_8, p3_9, p3_10, p3_11, p3_12 );
191
192     line4 : multilDet_line3
193     Port Map
194     ( A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, B4, p3_1, p3_2, p3_3, p3_4, p3_5, p3_6, p3_7, p3_8, p3_9, p3_10, p3_11, p3_12, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q10, Q11, Q12, Q13, Q14, Q15 );
195
196 end Behavioral;
197

```

Figure 28 - Multiplier port maps

When the full 12x4 multiplier VHDL logic has been synthesized a VHDL test bench was created. The purpose of the test bench is to simulate the design, you need both the design under test (DUT) or unit under test (UUT) and the stimulus provided by the test bench. A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. [6]

12 bit x 4 bit multiplier behavioural model

The check the multiplier is functioning as required a behavioural model was created. This allowed to check the multiplier at various stages to ensure correct operation. For ease of interoperating the results, the inputs and outputs were made into virtual busses, A1 – A12 in one bus, B1 – B4 in another and Q0 – Q15 in another. The radix was also changed from binary to unsigned decimal to allow easier interpretation of values.



Figure 29 - Multiplier (1220*4 = 4880)

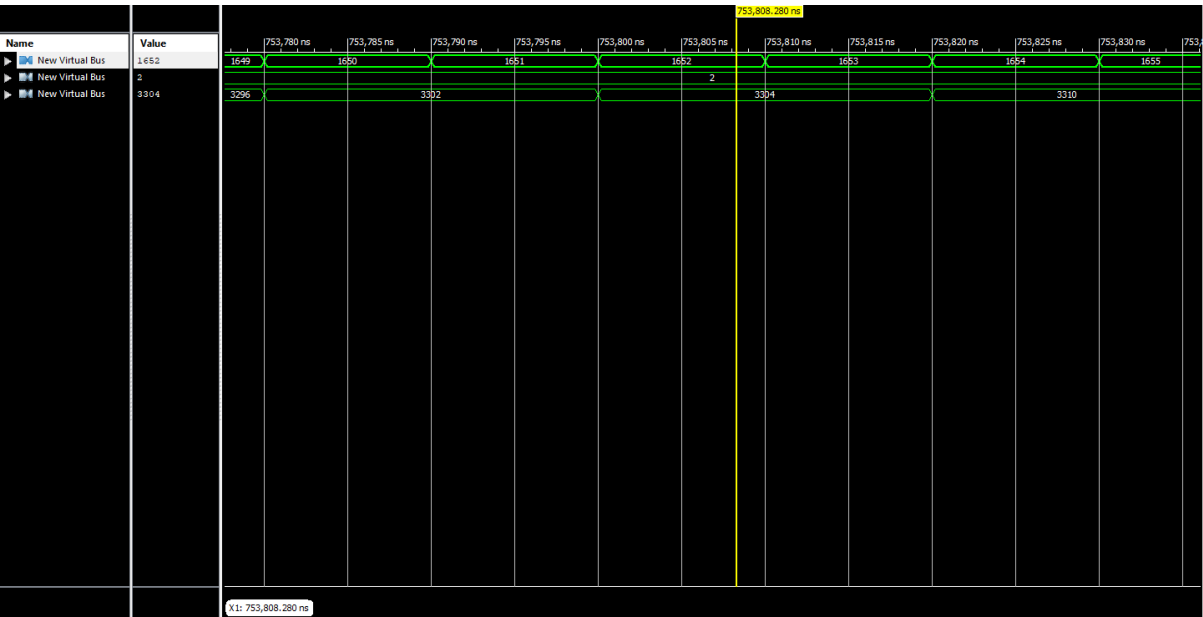


Figure 30 - Multiplier (1652*2 = 3304)

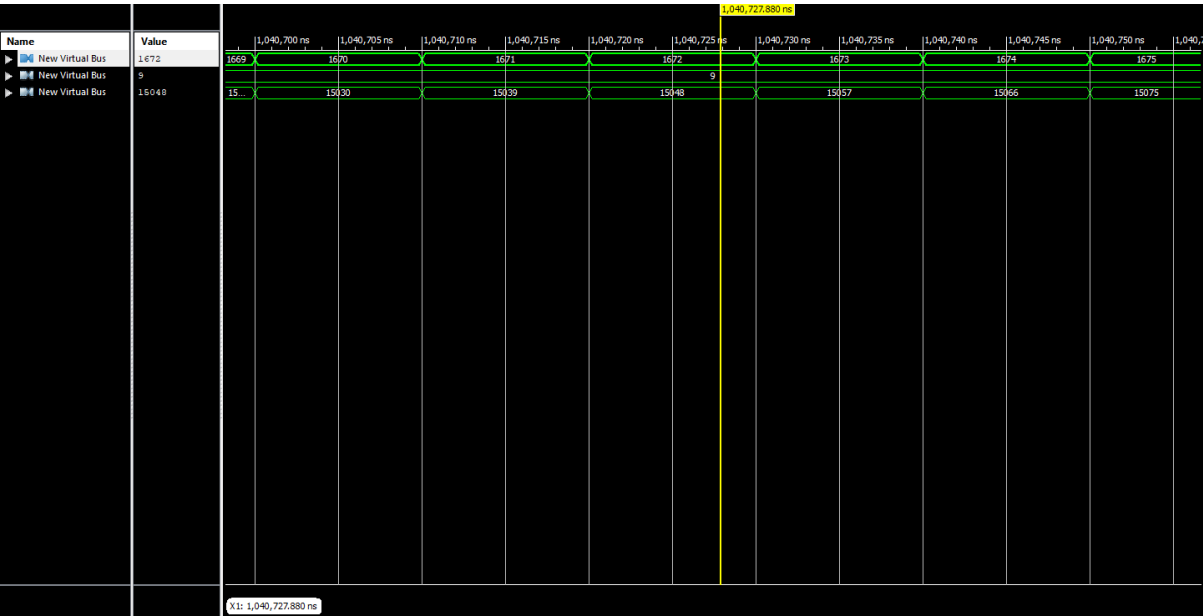


Figure 31 - Multiplier (1672*9 = 15048)

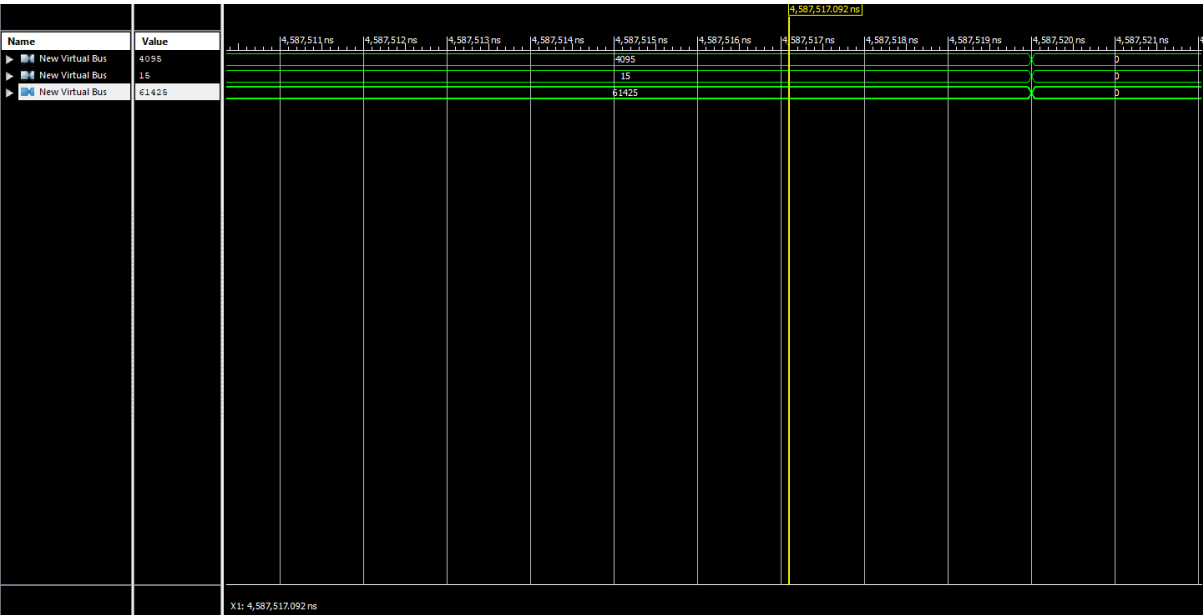


Figure 32 - Largest multiplication (4095*15=61425)

The largest combination for this multiplier is $(2^{12} - 1) * (2^4 - 1) = 61425$.

Part 2 – Design, simulate and synthesise the FSM as VHDL and discrete gate solution

This part of the report requires the design, simulation and synthetisation of a finite state machine in VHDL format using the Xilinx software but also as a discrete gate solution.

A state machine is a behaviour model. It consists of a finite number of states and is therefore also called finite-state machine (FSM). Based on the current state and a given input the machine performs state transitions and produces outputs. There are basic types like Mealy and Moore machines and more complex types like Harel and UML statecharts. [7]

Discrete gate solution

The FSM that is required to be designed is described by the state diagram pictured below.

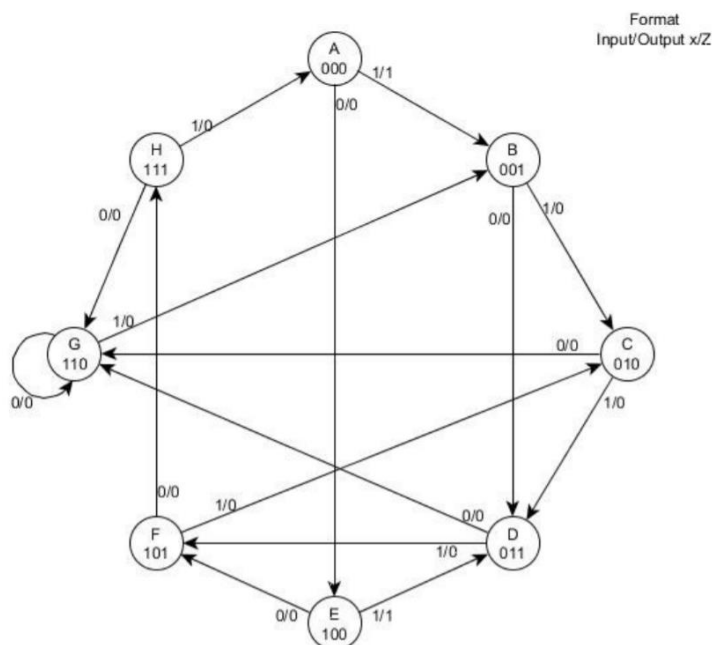


Figure 33 - State diagram

From the information provided by the state diagram, a state table and next state table can be generated. The state table is detailed below.

	Present state			Input	Next state			Output
	a	b	c	x	a	b	c	Z
State A	0	0	0	0	1	0	0	0
	0	0	0	1	0	0	1	1
State B	0	0	1	0	0	1	1	0
	0	0	1	1	0	1	0	0
State C	0	1	0	0	1	1	0	0
	0	1	0	1	0	1	1	0
State D	0	1	1	0	1	1	0	0
	0	1	1	1	1	0	1	0
State E	1	0	0	0	1	0	1	0
	1	0	0	1	0	1	1	1
State F	1	0	1	0	1	1	1	0
	1	0	1	1	0	1	0	0
State G	1	1	0	0	1	1	0	0
	1	1	0	1	0	0	1	0
State H	1	1	1	0	1	1	0	0
	1	1	1	1	0	0	0	0

Figure 34 - State table

From the state table we can generate Karnaugh maps. A Karnaugh map gives a simplified, graphical expression of the state table. Using a Karnaugh map allows us to group together adjacent terms which reduces the function length and reduces the required circuitry so the final discrete gate solution will use minimal components.

Da

	ab			
	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	1	0	0
10	0	1	1	1

$$D_a = \overline{C} \cdot \overline{X} + \overline{A} \cdot B \cdot C + A \cdot C \cdot \overline{X}$$

Db

	ab			
	00	01	11	10
00	0	1	1	0
01	0	1	0	1
11	1	0	0	1
10	1	1	1	1

$$D_b = C \cdot \overline{X} + \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot \overline{C} + B \cdot \overline{C} \cdot \overline{X} + A \cdot \overline{B} \cdot X$$

Dc

	ab			
	00	01	11	10
00	0	0	0	1
01	1	1	1	1
11	0	1	0	0
10	1	0	0	1

$$D_c = \overline{C} \cdot X + \overline{B} \cdot C \cdot \overline{X} + \overline{A} \cdot B \cdot X + A \cdot \overline{B} \cdot \overline{C}$$

Schematic of discrete gate solution

Using the simplified functions generated from the Karnaugh maps, we can produce a schematic diagram using the Proteus software. The quantity of gates is determined by the SOPs derived from the Karnaugh maps.

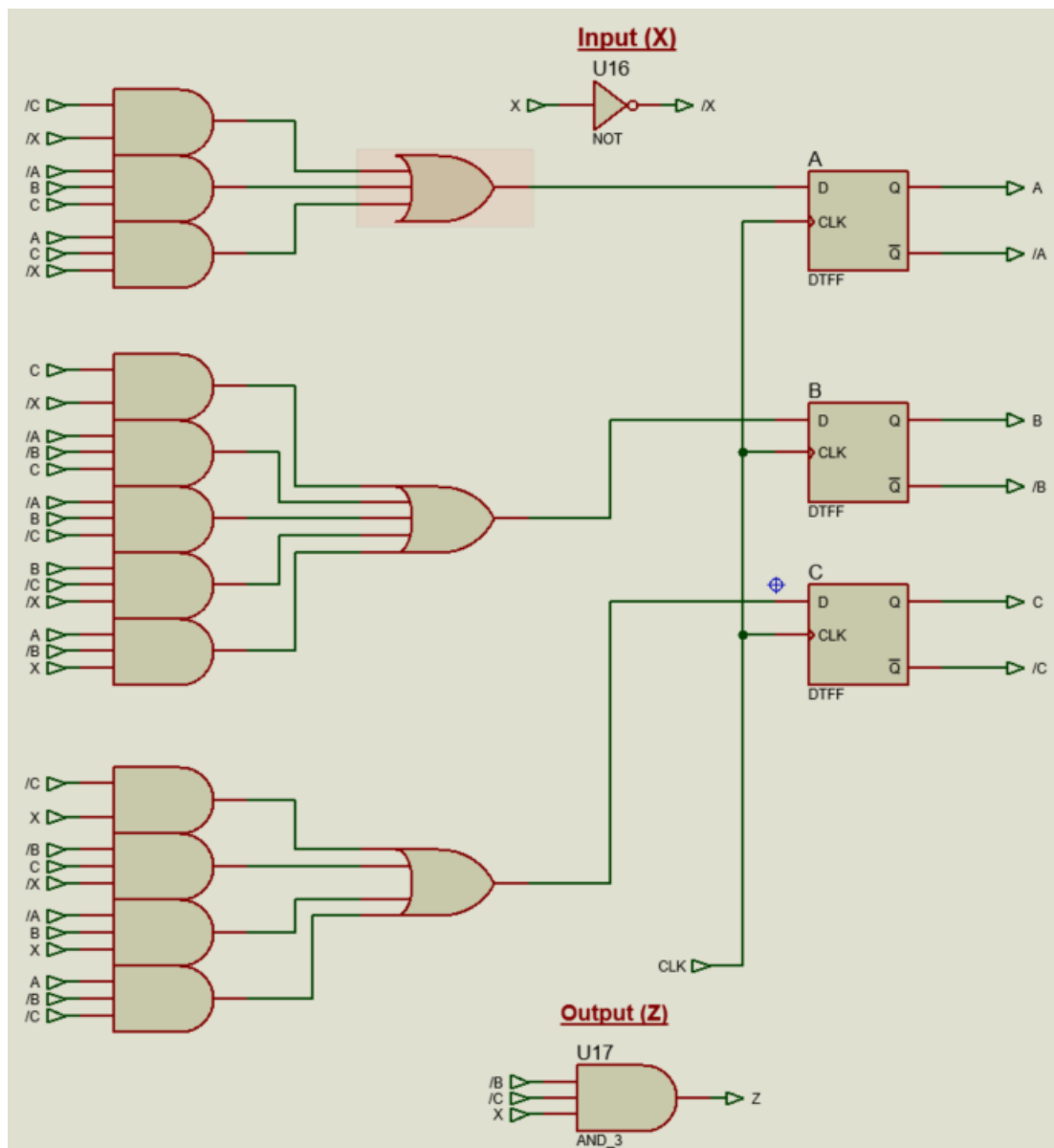


Figure 35 - FSM displayed as a discrete gate solution

The FSM has been designed using the Proteus software using the information gained from the state table and Karnaugh maps. This is a class A machine

FSM design using VHDL

Xilinx has been utilised to design the FSM using VHDL code. A VHDL module is created to be able to write the code. Firstly the inputs and outputs are created. In this instance, x is the input which can be assigned 0 or 1. The clock reference is also entered as an input. Z and y are assigned as outputs.

```

32 entity fsm is
33     Port ( x : in  STD_LOGIC;
34           clk : in  STD_LOGIC;
35           z : out  STD_LOGIC;
36           y : out  STD_LOGIC_VECTOR (2 downto 0));
37 end fsm;
38
39 architecture Behavioral of fsm is
40
41     type state_type is (state_A, state_B, state_C, state_D, state_E, state_F, state_G, state_H);
42     signal present_state : state_type := state_A;
43     signal next_state : state_type := state_A;
44
45 begin
46
47     State: process (present_state,x)
48     begin
49         if x='0' then
50             case present_state is
51                 When state_A => next_state <= state_E;Y <="000"; z <='0';
52                 When state_B => next_state <= state_D;Y <="001"; z <='0';
53                 When state_C => next_state <= state_G;Y <="010"; z <='0';
54                 When state_D => next_state <= state_G;Y <="011"; z <='0';
55                 When state_E => next_state <= state_F;Y <="100"; z <='0';
56                 When state_F => next_state <= state_H;Y <="101"; z <='0';
57                 When state_G => next_state <= state_G;Y <="110"; z <='0';
58                 When state_H => next_state <= state_G;Y <="111"; z <='0';
59                 When OTHERS => NULL;
60
61             end case;
62
63             else if x = '1' Then
64                 case present_state is
65                     When state_A => next_state <= state_B;Y <="000"; z <='1';
66                     When state_B => next_state <= state_C;Y <="001"; z <='0';
67                     When state_C => next_state <= state_D;Y <="010"; z <='0';
68                     When state_D => next_state <= state_F;Y <="011"; z <='0';
69                     When state_E => next_state <= state_D;Y <="100"; z <='1';
70                     When state_F => next_state <= state_C;Y <="101"; z <='0';
71                     When state_G => next_state <= state_B;Y <="110"; z <='0';
72                     When state_H => next_state <= state_A;Y <="111"; z <='0';
73                     When OTHERS => NULL;
74
75                 end case;
76             end if;
77         end if;
78     end Process State;
79
80     clocked: Process (clk)
81     begin
82         if((clk='1') and (clk'last_value='0')) Then
83             present_state <= next_state;
84
85         end if;
86     end Process clocked;
87 end Behavioral;

```

Figure 36 - FSM in VHDL

References

- [1] "Electrical technology," May 2018. [Online]. Available:
] <https://www.electricaltechnology.org/2018/05/binary-multiplier-types-binary-multiplication-calculator.html> . [Accessed 1 May 2021].
- [2] S. K. Soniya1, "A Review of Different Type of Multipliers and Multiplier-Accumulator Unit.,"
] *International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)*, vol. 2, no. 4, pp. 364-368, 2013.
- [3] 2. A. 1MADHU GUDHIMETLA, "COMPARISON OF DIFFERENT TYPES OF MULTIPLIERS WITH
] RESPECT TO SPEED, AREA AND POWER," *International Journal of Industrial Electronics and Electrical Engineering*, vol. 5, no. 12, pp. 83-87, 2017.
- [4] A.P.Paplinski, "classes/wi06/cse140/LN07.pdf," 18 August 2003. [Online]. Available:
] <https://cseweb.ucsd.edu/classes/wi06/cse140/LN07.pdf> . [Accessed 1 May 2021].
- [5] "support/documentation/sw_manuals/xilinx11/pp_p_process_synthesize.htm," Xilinx, [Online].
] Available:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/pp_p_process_synthesize.htm. [Accessed 1 May 2021].
- [6] "support/documentation/sw_manuals/xilinx11/ise_c_simulation_test_bench.htm," Xilinx,
] [Online]. Available:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_simulation_test_bench.htm. [Accessed 1 May 2021].
- [7] "en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines,"
] Itemis, [Online]. Available: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines. [Accessed 2 May 2021].