

Electronics II

Jordan Thomas - 1503513

Dr Gary Jones



Prifysgol Cymru

Y Drindod Dewi Sant

University of Wales

Trinity Saint David

**Assignment submitted as part of the requirements for the
award of BEng (Hons) Electrical and Electronic Engineering**

2020/2021

Contents

Table of Figures	3
Introduction	3
Chapter 1 - Research	4
1.1 Literature Review	4
1.2 Multiplier types	5
1.2.1 Serial Multipliers	5
1.2.2 Parallel Multipliers	5
1.2.3 Serial Parallel Multiplier	5
1.3 Booth Multiplier	5
1.4 Wallace Tree Multiplier	5
1.4 Braun Multiplier	6
Chapter 2 Design of 12 x 4 Bit Braun Multiplier	7
2.1 Design Overview	7
2.2 Module 1 - Half Adder	8
2.3 Module 2 - Full Adder	10
2.4 Module 3 - Line 0	12
2.5 Module 4 - Line X	15
Module 5	17
Chapter 3	18
3.1 Design, Simulate and Synthesise the FSM described by the state diagram below as both a VHDL and discrete gate solution	18
3.2 State & Next State Table	19
3.3 Schematic Diagram	20
3.4 FSM Using VHDL	21
Bibliography	23

Table of Figures

Figure 1 - Half Adder Truth Table.....	8
Figure 2 - Half Adder VHDL Code	8
Figure 3 - Half Adder Test Bench	9
Figure 4 - Example of Simulator & Truth Table.....	9
Figure 5 - Example of Simulator & Truth Table (A=1, B=1)	9
Figure 6 - Full Adder Truth Table	10
Figure 7 - Full Adder VHDL Code	10
Figure 8 - Full Adder Test Bench	11
Figure 9 - Full Adder Simulator & Truth Table	11
Figure 10 - Full Adder Simulator (A=0, B=1, C=1).....	12
Figure 11 - Line 0 VHDL Code	12
Figure 12 - Line 0 Test Bench Lines 35 - 90	13
Figure 13 - Line 0 Test Bench Lines 90 - 145	13
Figure 14 - Line 0 Test Bench Lines 146 - 200	14
Figure 15 - Line 0 Test Bench Lines 203 - 253	14
Figure 16 - Line 0 Simulator	15
Figure 17 - Line X, Lines 32 to 87	15
Figure 18 - Line X, Lines 88 to 143	16
Figure 19 - Line X, Lines 143 to 184	16
Figure 20 - Hierarchy.....	17
Figure 21 - State 'A' K-Map	19
Figure 22 - State 'B' K-Map.....	19
Figure 23 - State 'C' K-Map.....	20
Figure 24 - Schematic Diagram	20
Figure 25 - FSM in VHDL part 1	21
Figure 26 - FSM in VHDL part 2	21
Figure 27 - Design Summary 1	22
Figure 28 - Design Summary 2	22

Introduction

This report firstly contains a review of 3 popular digital multipliers that are used for digital signal processing and control applications. The Braun multiplier, the Wallace Tree multiplier and the Booth multiplier have been described. The report then looks at completing task 1 of the assignment and contains the basics of a Braun 12 x 4 bit multiplier that has been designed and simulated on Xilinx software using VHDL code. The report closes with an FSM design using the discrete gate solution.

Chapter 1 - Research

1.1 Literature Review

The binary multiplier is an electronic device/circuit capable of multiplying two binary numbers with the use of half adders and full adders. The result of the two binary numbers is known as the product. These multipliers are vitally important and play a fundamental role in digital signal processing (DSP) and are required for many electrical applications such as mobile phones, microprocessors and all computers. [1] [2]

It was only in the late 70's when minicomputers began to operate with multiply instructions, notably the Motorola 6809 was one of the first processors to utilise the multiply instruction. [3]

Microprocessors and DSP systems largely use the multiplication as the basic arithmetic for their design. At the beginning the multipliers were very slow due to the number of transistors per chip. [3] During design, area optimization and speed are key consideration due to the demand from applications and components that require high speed processing systems. Multiplier design firstly involved reusing a single adder to handle each partial product at a time. This changed when more transistors per chip became available allowing for enough adders on a single chip to sum all partial products at once. [3]

To achieve higher processing speeds, parallel multipliers are usually adopted at the expense of high area optimization.

The main considerations when designing a successful and efficient multiplier are accuracy, speed, area utilization and power requirements [3]

1.2 Multiplier types

1.2.1 Serial Multipliers involve sequential circuits with feedback. Serial multipliers are typically slower than parallel multipliers due to the fact that each bit of the multiplication is added sequentially. [4]

1.2.2 Parallel Multipliers can be considered as 2 parts, the partial product accumulation and partial product generation. It is the number of partial products to be added that determines the length of the delay caused. Parallel multipliers generally require less hardware and power compared to alternatives. [4]

1.2.3 Serial Parallel Multiplier commonly referred to as SPM. This multiplier uses the parallel adder for partial product accumulation but operates on each bit of multiplier serially. These multipliers will have the speed of the parallel and small area similar to serial multiplier. [4]

1.3 Booth Multiplier

The Booth's multiplier is a conventional multiplication algorithm that is highly powerful and effective however, it generally requires a larger space of silicon than alternative algorithms. The booth multiplier has a performance speed advantage over alternative multiplier designs as a result of reducing the amount of partial products by considering 2 bits of the multiplier concurrently. [5] It is possible for binary integers that are signed to be multiplied using two's complement representation. [6]

1.4 Wallace Tree Multiplier

The Wallace tree multiplier is a parallel multiplier that is very efficient and offers superior speed for large operands. The first process of the multiplier is to form a partial product array of N^2 bits. For the second step of the process 3 adjacent rows are collected in groups. Half adders and full adders are then used in different ways to reduce the rows. The full adders are used in columns with 3 bits and the half adders are used in columns with 2 bits. This process is repeated until the rows are reduced and only 2 remain [8] Due to the design the Wallace tree multiplier has speed advantages however, the higher gate count can cause increased gate delays. The multiplier also has a large area and an irregular layout. [7]

1.4 Braun Multiplier

The Braun Multiplier is a parallel array multiplier. There are 3 main components which make up the multiplier including carry save adders, AND gates & 1 ripple carry adder. This is commonly referred to as non-additive multipliers. [9] [10] This multiplier is easier to design than alternatives due to the fact there is no requirement for logic registers. Due to the design and function, the amount of building blocks required increases quadratically for every additional bit. This will result in an increased power consumption, area and time delay. The multiplier will be unsuitable for any application greater than 16 bit. It is also known that there is a glitch potential due to the Ripple Carry Adder [9]

Chapter 2 Design of 12 x 4 Bit Braun Multiplier

2.1 Design Overview

As per the assignment brief a 12 x 4 bit Braun Multiplier will need to be designed using 5 different VHDL Modules. The first 4 smaller sub modules are very important as they provide the base for the design. The fifth module will be the biggest and will contain the all codes as one 12x4 bit multiplier.

Module 1 & 2 - The half adder and full adder are two circuits which will provide a solid base for the multiplier and once designed correctly can be used multiple times. The half adder takes 2 inputs and produces two outputs in the form of the sum and a carry bit. The full adder takes 3 inputs and produces 2 outputs in the form of the sum and carry out.

Module 3 - Module 3 is line 0. This is the top line of the multiplier and contains AND gates. signals between the adders can be controlled with the Part Map.

Module 4 - Module 4 is line x which contains 12 adders. This line also includes port mapping and can be reused.

Module 5 - The 12x4 bit Braun multiplier can be constructed only on completion of module 1, 2, 3 & 4. Those modules can be copied into the main multiplier module and connected via port mapping. Final bench tests can be carried out to prove the functionality of the design

When all VHDL modules have been created the code will need to be synthesised. this process takes place in order to check the functionality and design. The Xilinx software has a synthesise program built in. If synthesise is completed successfully then the next stage will be to complete a bench test which will simulate via DUT (Design under test). Each module will require unique test benches to be written.

2.2 Module 1 - Half Adder

The half adder circuit is an important component for the 12x4 multiplier as it will be used multiple times. The half adder circuit will add input 'A' & 'B' and produce 2 outputs of 'SUM' and 'CARRY BIT'. This cannot be used for applications which require a 'carry' input.

A	B	SUM	CARRY BIT
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 1 - Half Adder Truth Table

Figure 2 below shows the VHDL code that has been designed on Xilinx. Lines 33 - 36 state the inputs and outputs. Lines 43 & 44 show the SUM and CARRY outputs.

```
31
32 entity half_adder is
33     Port ( A : in  STD_LOGIC;
34           B : in  STD_LOGIC;
35           SUM : out STD_LOGIC;
36           CARRY : out STD_LOGIC);
37 end half_adder;
38
39 architecture Behavioral of half_adder is
40
41 begin
42
43     SUM <= A xor B;
44     CARRY <= A and B;
45
46 end Behavioral;
47
```

Figure 2 - Half Adder VHDL Code

On completion of writing the code it is a requirement to synthesize. This highlights any errors within the code before moving forward. When the code has been synthesized a test bench will need to be created that is capable of simulating the design.

Figure 3 shows the test bench created. Ports A and B are stimulated via the test bench with the aim of generating the outputs as required. stim_proc0 is used to switch the inputs from 0 to 1. Input A is switched every 10ns and input B is switched every 20ns.

```

62 BEGIN
63
64     -- Instantiate the Unit Under Test (UUT)
65     uut: half_adder PORT MAP (
66         A => A,
67         B => B,
68         SUM => SUM,
69         CARRY => CARRY
70     );
71
72     -- Stimulus process
73     stim_proc0: process
74
75     begin
76         -- hold reset state for 10 ns.
77         wait for 10 ns;
78
79         A <= not A;
80
81     end process;
82
83     stim_proc: process
84     begin
85         -- hold reset state for 20 ns.
86         wait for 20 ns;
87
88         B <= not B;
89
90     end process;
91
92 END;

```

Figure 3 - Half Adder Test Bench

The simulator mode on Xilinx is used to simulate the code. As the value of A and B changes so does the output value. This can be compared with the appropriate truth table to check for correct function.

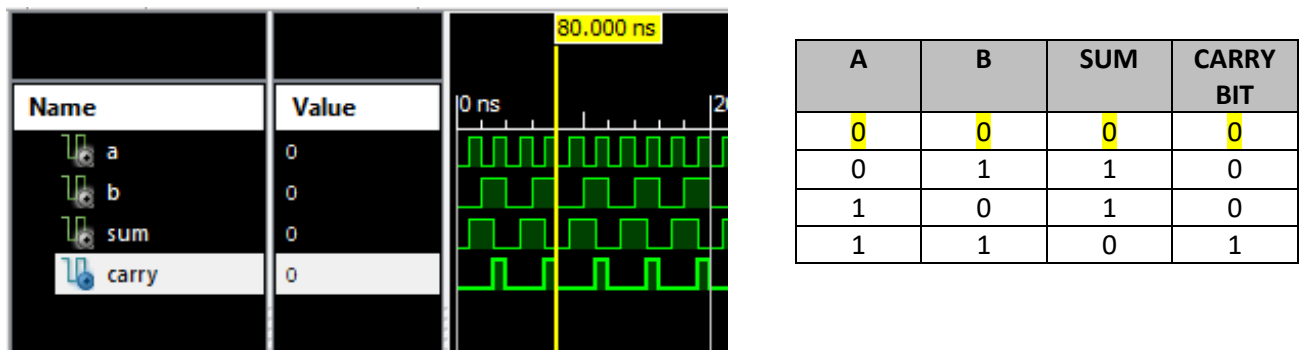


Figure 4 - Example of Simulator & Truth Table

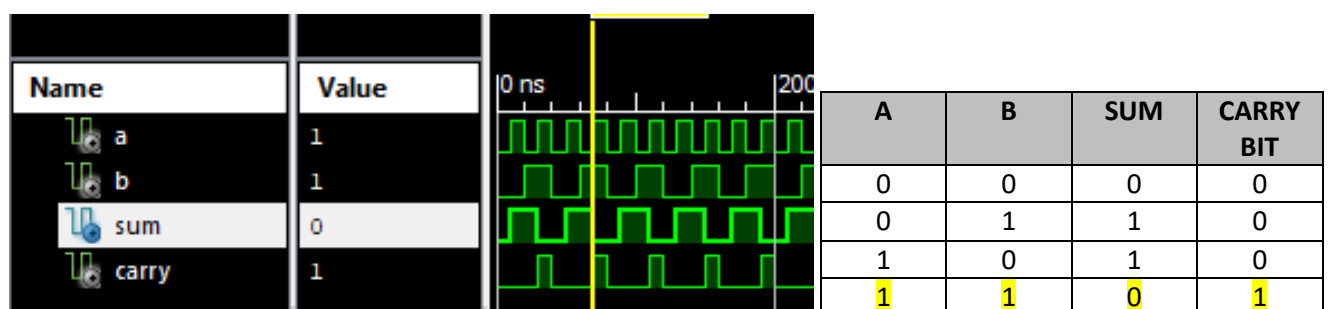


Figure 5 - Example of Simulator & Truth Table (A=1, B=1)

2.3 Module 2 - Full Adder

The main difference with the full adder is it contains a Carry input (Port C), as a result there are 3 inputs, 2 outputs and 3 more logic gates within its construction. Due to the extra input there are 8 possible combinations within the truth table.

A	B	Carry In (C)	SUM	CARRY BIT
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	1	0	0	0
1	1	1	1	1

Figure 6 - Full Adder Truth Table

Figure 7 below shows the full adder VHDL code. Lines 33 - 37 state the inputs and outputs. The relationship between the ports and interconnections between logic gates are expressed between lines 40 and 55.

```
31
32 entity Full_Adder 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 Full_Adder;
39
40 architecture Behavioral of Full_Adder is
41     signal AxB : STD_LOGIC;
42     signal AxBnC: STD_LOGIC;
43     signal AnB : STD_LOGIC;
44
45 begin
46
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 7 - Full Adder VHDL Code

As per the half adder, a test bench is required to simulate the design of the full adder. Figure 8 shows the code of the test bench created. To create the correct cross over and to ensure all states are covered input A is switched every 10ns, input B every 20 ns and input C every 40 ns.

```

63 BEGIN
64     -- Instantiate the Unit Under Test (UUT)
65     uut: Full_Adder PORT MAP (
66         A => A,
67         B => B,
68         C => C,
69         SUM => SUM,
70         CARRY => CARRY
71     );
72     -- Stimulus process
73     stim_proc: process
74     begin
75         -- hold reset state for 10 ns.
76         wait for 10 ns;
77         A <= not A;
78     end process;
79     stim_proc1: process
80     begin
81         -- hold reset state for 20 ns.
82         wait for 20 ns;
83         B <= not B;
84     end process;
85     stim_proc2: process
86     begin
87         -- hold reset state for 40 ns.
88         wait for 40 ns;
89         C <= not C;
90     end process;
91 END;

```

Figure 8 - Full Adder Test Bench

Figure 9 & 1 show 2 examples of the simulator in operation with the full adder wave form. The truth table can be used to check accuracy off results.

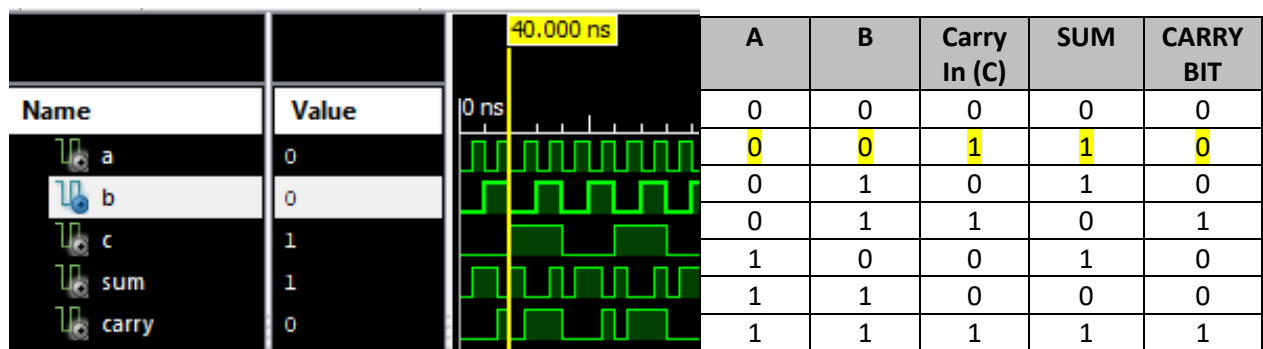


Figure 9 - Full Adder Simulator & Truth Table

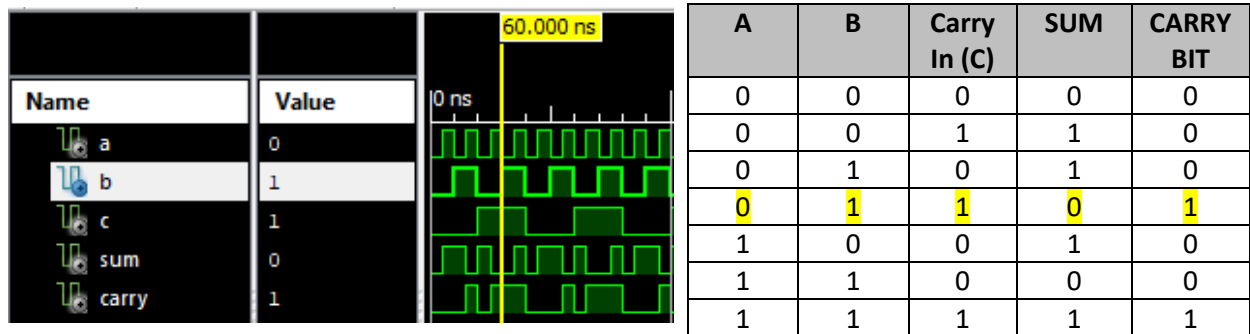


Figure 10 - Full Adder Simulator (A=0, B=1, C=1)

2.4 Module 3 - Line 0

The next step is to create the line 0 behavioural will be the top line of the multiplier and consist of 12 AND gates. Figure 11 shows VHDL code created which states the inputs, outputs, architecture body and relationship between ports.

```

32 entity line_0 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 end line_0;
59
60 architecture Behavioral of line_0 is
61 begin
62     Q1 <= A1 and B;
63     Q2 <= A2 and B;
64     Q3 <= A3 and B;
65     Q4 <= A4 and B;
66     Q5 <= A5 and B;
67     Q6 <= A6 and B;
68     Q7 <= A7 and B;
69     Q8 <= A8 and B;
70     Q9 <= A9 and B;
71     Q10 <= A10 and B;
72     Q11 <= A11 and B;
73     Q12 <= A12 and B;
74
75
76
77 end Behavioral;

```

Figure 11 - Line 0 VHDL Code

A VHDL test bench has been designed to test the code written. The test bench will create an input stimuli which will make the circuit react and generate the required outputs. the inputs are first set to 0 and the stim_proc command is used to switch from 0 to 1. The first input is switched every 10ns, the time delay for every input after that is doubled. The last process (stim_proc12) is 81920ns

```

35 ENTITY Line_0_Test IS
36 END Line_0_Test;
37
38 ARCHITECTURE behavior OF Line_0_Test IS
39
40     -- Component Declaration for the Unit Under Test (UUT)
41
42     COMPONENT line_0
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 --Inputs
73 signal A1 : std_logic := '0';
74 signal A2 : std_logic := '0';
75 signal A3 : std_logic := '0';
76 signal A4 : std_logic := '0';
77 signal A5 : std_logic := '0';
78 signal A6 : std_logic := '0';
79 signal A7 : std_logic := '0';
80 signal A8 : std_logic := '0';
81 signal A9 : std_logic := '0';
82 signal A10 : std_logic := '0';
83 signal A11 : std_logic := '0';
84 signal A12 : std_logic := '0';
85 signal B : std_logic := '0';
86
87 --Outputs
88 signal Q1 : std_logic;
89 signal Q2 : std_logic;
90 signal Q3 : std_logic;

```

Figure 12 - Line 0 Test Bench Lines 35 - 90

```

90 signal Q3 : std_logic;
91 signal Q4 : std_logic;
92 signal Q5 : std_logic;
93 signal Q6 : std_logic;
94 signal Q7 : std_logic;
95 signal Q8 : std_logic;
96 signal Q9 : std_logic;
97 signal Q10 : std_logic;
98 signal Q11 : std_logic;
99 signal Q12 : std_logic;
100 -- No clocks detected in port list. Replace <clock> below with
101 -- appropriate port name
102
103 BEGIN
104
105     -- Instantiate the Unit Under Test (UUT)
106     uut: line_0 PORT MAP (
107         A1 => A1,
108         A2 => A2,
109         A3 => A3,
110         A4 => A4,
111         A5 => A5,
112         A6 => A6,
113         A7 => A7,
114         A8 => A8,
115         A9 => A9,
116         A10 => A10,
117         A11 => A11,
118         A12 => A12,
119         B => B,
120         Q1 => Q1,
121         Q2 => Q2,
122         Q3 => Q3,
123         Q4 => Q4,
124         Q5 => Q5,
125         Q6 => Q6,
126         Q7 => Q7,
127         Q8 => Q8,
128         Q9 => Q9,
129         Q10 => Q10,
130         Q11 => Q11,
131         Q12 => Q12
132     );
133
134     -- Stimulus process
135     stim_proc: process
136     begin
137         -- hold reset state for 20 ns.
138         wait for 20 ns;
139
140         A1 <= not A1;
141
142     end process;
143     stim_procl: process
144     begin
145         -- hold reset state for 100 ns.

```

Figure 13 - Line 0 Test Bench Lines 90 - 145

<pre> 146 wait for 40 ns; 147 148 A2 <= not A2; 149 150 end process; 151 152 stim_proc2: process 153 begin 154 -- hold reset state for 100 ns. 155 wait for 80 ns; 156 157 A3 <= not A3; 158 159 end process; 160 161 stim_proc3: process 162 begin 163 -- hold reset state for 100 ns. 164 wait for 160 ns; 165 166 A4 <= not A4; 167 168 end process; 169 170 stim_proc4: process 171 begin 172 -- hold reset state for 100 ns. 173 wait for 320 ns; </pre>	<pre> 174 175 A5 <= not A5; 176 177 end process; 178 179 stim_proc5: process 180 begin 181 -- hold reset state for 100 ns. 182 wait for 640 ns; 183 184 A6 <= not A6; 185 186 end process; 187 188 stim_proc6: process 189 begin 190 -- hold reset state for 100 ns. 191 wait for 1280 ns; 192 193 A7 <= not A7; 194 195 end process; 196 197 stim_proc7: process 198 begin 199 -- hold reset state for 100 ns. 200 wait for 2560 ns; </pre>
--	---

Figure 14 - Line 0 Test Bench Lines 146 - 200

<pre> 203 204 end process; 205 206 stim_proc8: process 207 begin 208 -- hold reset state for 100 ns. 209 wait for 5120 ns; 210 211 A9 <= not A9; 212 213 end process; 214 215 stim_proc9: process 216 begin 217 -- hold reset state for 100 ns. 218 wait for 10240 ns; 219 220 A10 <= not A10; 221 222 end process; 223 224 stim_proc10: process 225 begin 226 -- hold reset state for 100 ns. 227 wait for 20480 ns; 228 229 A11 <= not A11; 230 231 end process; </pre>	<pre> 226 -- hold reset state for 100 ns. 227 wait for 20480 ns; 228 229 A11 <= not A11; 230 231 end process; 232 233 stim_proc11: process 234 begin 235 -- hold reset state for 100 ns. 236 wait for 40960 ns; 237 238 A12 <= not A12; 239 240 end process; 241 242 stim_proc12: process 243 begin 244 -- hold reset state for 100 ns. 245 wait for 81920 ns; 246 247 B <= not B; 248 249 end process; 250 251 252 END; 253 </pre>
--	--

Figure 15 - Line 0 Test Bench Lines 203 - 253

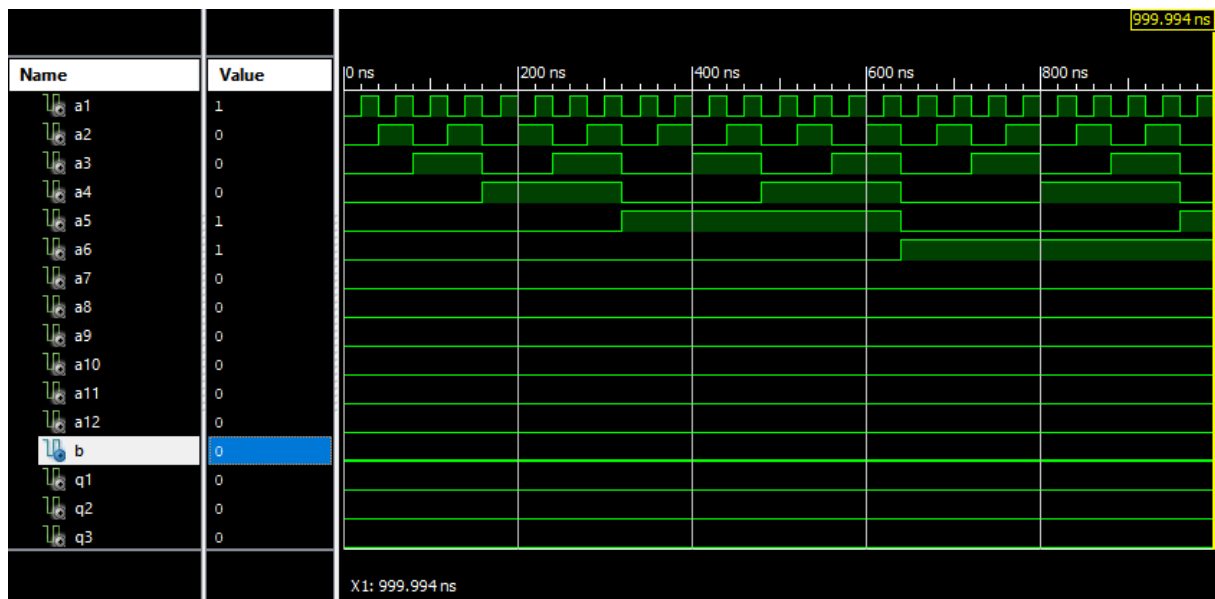


Figure 16 - Line 0 Simulator

2.5 Module 4 - Line X

Line x consists of 1 half adder and 11 full adders from the previous modules using internal and external connections. The lines below show the input and output signals, the half adder and full adder references, the gate sequence and the port mapping.

```

32  entity Line_x 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 Line_x;
72
73  architecture Behavioral of Line_x is
74
75
76      COMPONENT half_adder is
77          Port ( A : in  STD_LOGIC;
78                B : in  STD_LOGIC;
79                SUM : out  STD_LOGIC;
80                CARRY : out  STD_LOGIC);
81      END COMPONENT;
82
83      COMPONENT full_adder is
84          Port ( A : in  STD_LOGIC;
85                B : in  STD_LOGIC;
86                C : in  STD_LOGIC;
87                SUM : out  STD_LOGIC;

```

Figure 17 - Line X, Lines 32 to 87

<pre> 88 CARRY : out STD_LOGIC); 89 END COMPONENT; 90 91 signal s0 : STD_LOGIC; 92 signal s1 : STD_LOGIC; 93 signal s2 : STD_LOGIC; 94 signal s3 : STD_LOGIC; 95 signal s4 : STD_LOGIC; 96 signal s5 : STD_LOGIC; 97 signal s6 : STD_LOGIC; 98 signal s7 : STD_LOGIC; 99 signal s8 : STD_LOGIC; 100 signal s9 : STD_LOGIC; 101 signal s10 : STD_LOGIC; 102 signal s11 : STD_LOGIC; 103 104 signal c1 : STD_LOGIC; 105 signal c2 : STD_LOGIC; 106 signal c3 : STD_LOGIC; 107 signal c4 : STD_LOGIC; 108 signal c5 : STD_LOGIC; 109 signal c6 : STD_LOGIC; 110 signal c7 : STD_LOGIC; 111 signal c8 : STD_LOGIC; 112 signal c9 : STD_LOGIC; 113 signal c10 : STD_LOGIC; 114 signal c11 : STD_LOGIC; 115 </pre>	<pre> 116 begin 117 118 -- and gate sequence. 119 s0 <= A1 and B; 120 s1 <= A2 and B; 121 s2 <= A3 and B; 122 s3 <= A4 and B; 123 s4 <= A5 and B; 124 s5 <= A6 and B; 125 s6 <= A7 and B; 126 s7 <= A8 and B; 127 s8 <= A9 and B; 128 s9 <= A10 and B; 129 s10 <= A11 and B; 130 s11 <= A12 and B; 131 132 HA1 : half_adder 133 Port Map 134 (s0 , P1 , Q1 , c1); 135 136 FA1 : full_adder 137 Port Map 138 (s1 , P2 , c1 , Q2 , c2); 139 140 FA2 : full_adder 141 Port Map 142 (s2 , P3 , c2 , Q3 , c3); 143 </pre>
---	--

Figure 18 - Line X, Lines 88 to 143

<pre> 143 144 FA3 : full_adder 145 Port Map 146 (s3 , P4 , c3 , Q4 , c4); 147 148 FA4 : full_adder 149 Port Map 150 (s4 , P5 , c4 , Q5 , c5); 151 152 FA5 : full_adder 153 Port Map 154 (s5 , P6 , c5 , Q6 , c6); 155 156 FA6 : full_adder 157 Port Map 158 (s6 , P7 , c6 , Q7 , c7); 159 160 FA7 : full_adder 161 Port Map 162 (s7 , P8 , c7 , Q8 , c8); 163 164 FA8 : full_adder 165 Port Map 166 (s8 , P9 , c8 , Q9 , c9); 167 168 FA9 : full_adder 169 Port Map 170 (s9 , P10 , c9 , Q10 , c10); </pre>	<pre> 157 Port Map 158 (s6 , P7 , c6 , Q7 , c7); 159 160 FA7 : full_adder 161 Port Map 162 (s7 , P8 , c7 , Q8 , c8); 163 164 FA8 : full_adder 165 Port Map 166 (s8 , P9 , c8 , Q9 , c9); 167 168 FA9 : full_adder 169 Port Map 170 (s9 , P10 , c9 , Q10 , c10); 171 172 FA10 : full_adder 173 Port Map 174 (s10 , P11 , c10 , Q11 , c11); 175 176 FA11 : full_adder 177 Port Map 178 (s11 , P12 , c11 , Q12 , Q13); 179 180 181 182 end Behavioral; 183 184 </pre>
---	--

Figure 19 - Line X, Lines 143 to 184

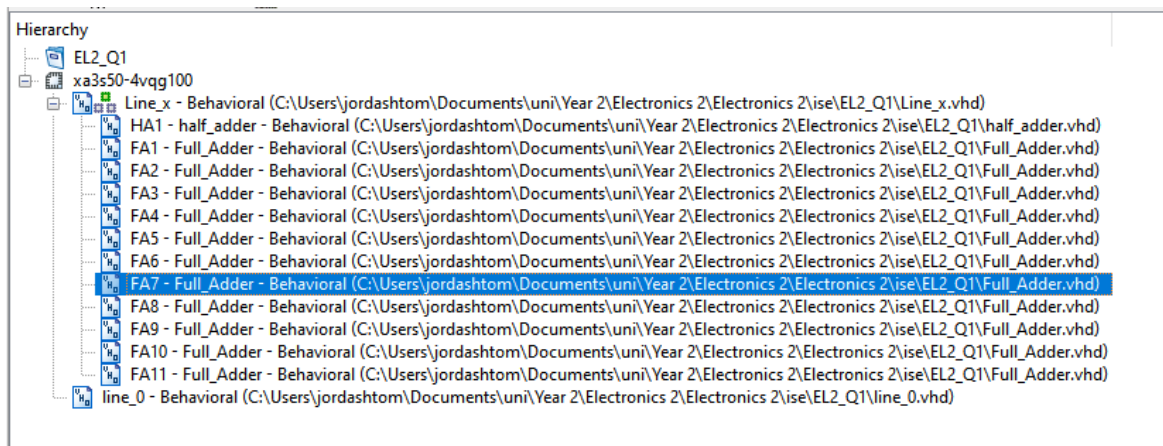


Figure 20 - Hierarchy

Test bench

Module 5

Chapter 3

3.1 Design, Simulate and Synthesise the FSM described by the state diagram below as both a VHDL and discrete gate solution.

This part of the report looks at how a Finite State Machine can be designed using both VHDL code and the discrete gate solution. A Finite State Machine, or FSM, is a form of computation that is used to model Logic. An FSM found in a specific state from a finite set of possible states. [11] Only by accepting an input can it transition to the next state.

Figure 21 shows the state diagram. An FSM will need to be designed to describe it. The circuit has only 1 input which is either 0 or 1.

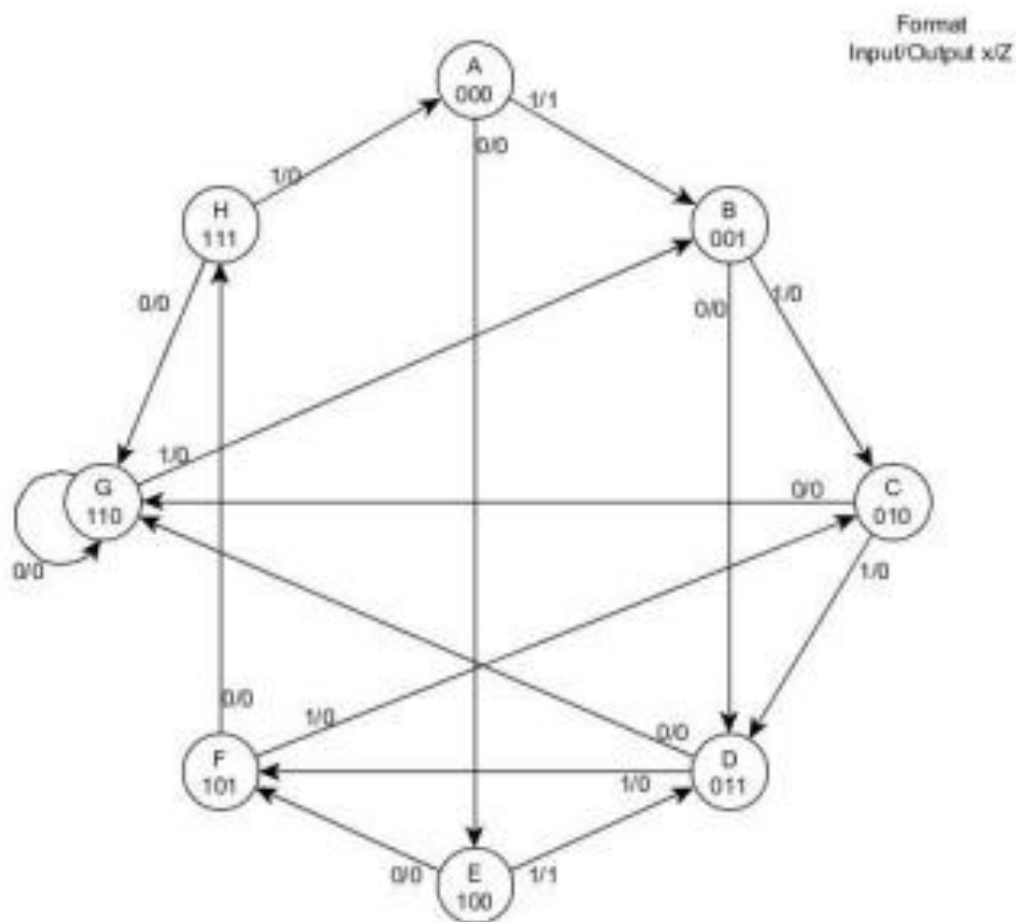


Figure 21 - State Diagram

The next step is to design a state and next state table based on the information in figure 21

3.2 State & Next State Table

Below shows the State & Next State tables that have been designed from the state diagram.

	State			Input	Next State			Output
	A	B	C	X	A	B	C	Z
0	0	0	0	0	1	0	0	1
	0	0	0	1	0	0	1	
1	0	0	1	0	0	1	1	
	0	0	1	1	0	1	0	
2	0	1	0	0	1	1	0	
	0	1	0	1	0	1	1	
3	0	1	1	0	1	1	0	
	0	1	1	1	1	0	1	
4	1	0	0	0	1	0	1	1
	1	0	0	1	0	1	1	
5	1	0	1	0	1	1	1	
	1	0	1	1	0	1	0	
6	1	1	0	0	1	1	0	
	1	1	0	1	0	0	1	
7	1	1	1	0	1	1	0	
	1	1	1	1	0	0	0	

The next stage is to use the information from the next state table and use it to design Karnaugh Maps. A telecommunications engineer, Maurice Karnaugh, developed the Karnaugh map whilst designing digital logic based telephone switching circuits in 1953 [12] Karnaugh maps are an effective way of reducing logic functions by having a lower number of gates.

The Karnaugh map allows us to combine terms that are adjacent to each other within the table. When the grouping happens it allows for 1 or 2 terms to drop out of the final function. This reduces the total function length and simplifies the circuit which results in less components.

Da		AB				
		Da	_00	_01	11	10
CX	_00	1	1	1	1	
	_01	0	0	0	0	
	11	0	1	0	0	
	10	0	1	1	1	

$$da = \overline{C}/X + \overline{A}.B.C + A.C/\overline{X}$$

Figure 22 - State 'A' K-Map

Figure22 shows the Karnaugh map and simplified expression for state 'A'.

		AB				
		Da	_00	_01	11	10
CX	Da	_00	0	1	1	0
	_00	0	1	0	1	
	_01	0	1	0	1	
	11	1	0	0	1	
	10	1	1	1	1	

Db=C.X + A./B.X + /A.B./C + B./C./X + /A./B.C

Figure 23 - State 'B' K-Map

Figure23 shows the Karnaugh map and simplified expression for state 'B'.

Dc		AB			
CX	Da	_00	_01	11	10
	_00	0	0	0	1
	_01	1	1	1	1
	11	0	1	0	0
	10	1	0	0	1

$$Dc = C.X + A./B./C + /A.B.X + /B.C./X$$

Figure 24 - State 'C' K-Map

Figure24 shows the Karnaugh map and simplified expression for state 'C'.

3.3 Schematic Diagram

Figure 25 is a complete schematic diagram that has been derived from the Karnaugh Maps above.

The diagram which has been created on Proteus contains logic gates and D-Type flip flops. This shows the FSM as a discrete gate solution. This known as a class A machine as an X input goes straight to an output.

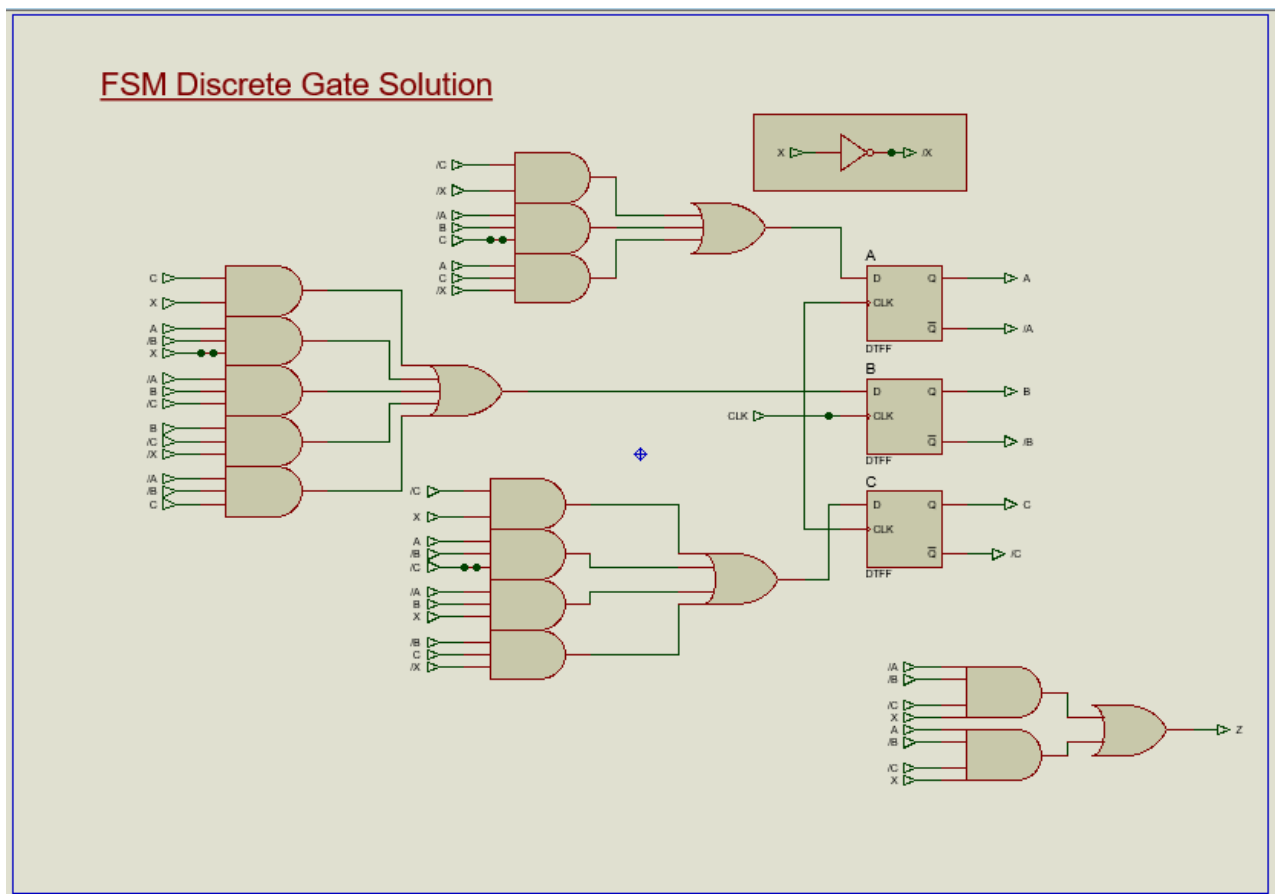


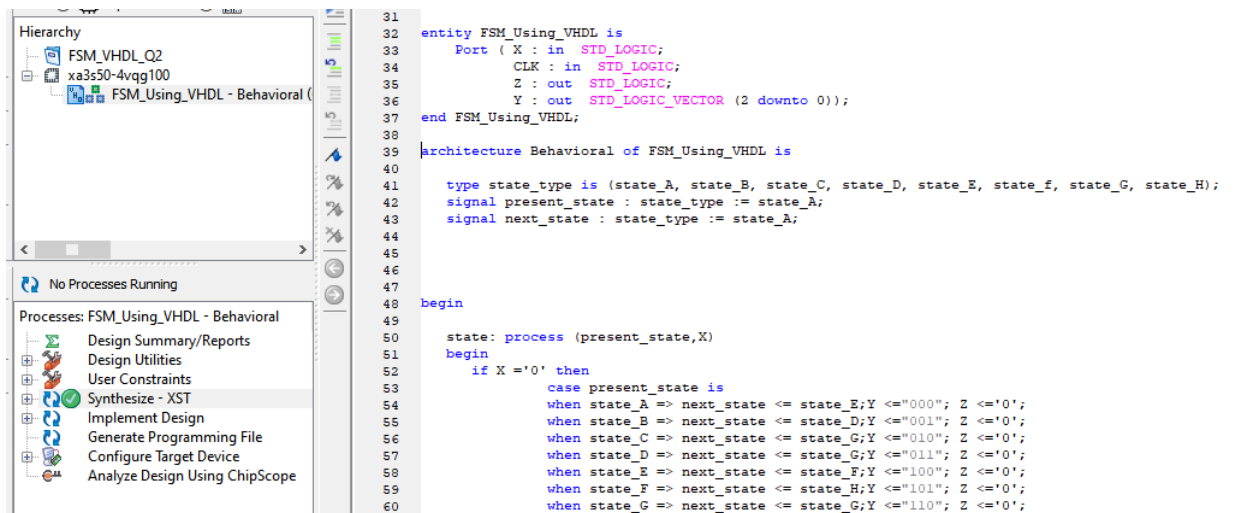
Figure 25 - Schematic Diagram

3.4 FSM Using VHDL

Xilinx is the software that has been used to design the FSM using VHDL. A VHDL module was created and all inputs/outputs listed. For this task the inputs were 'X' & 'CLK' and the outputs were 'Z' & 'Y'.

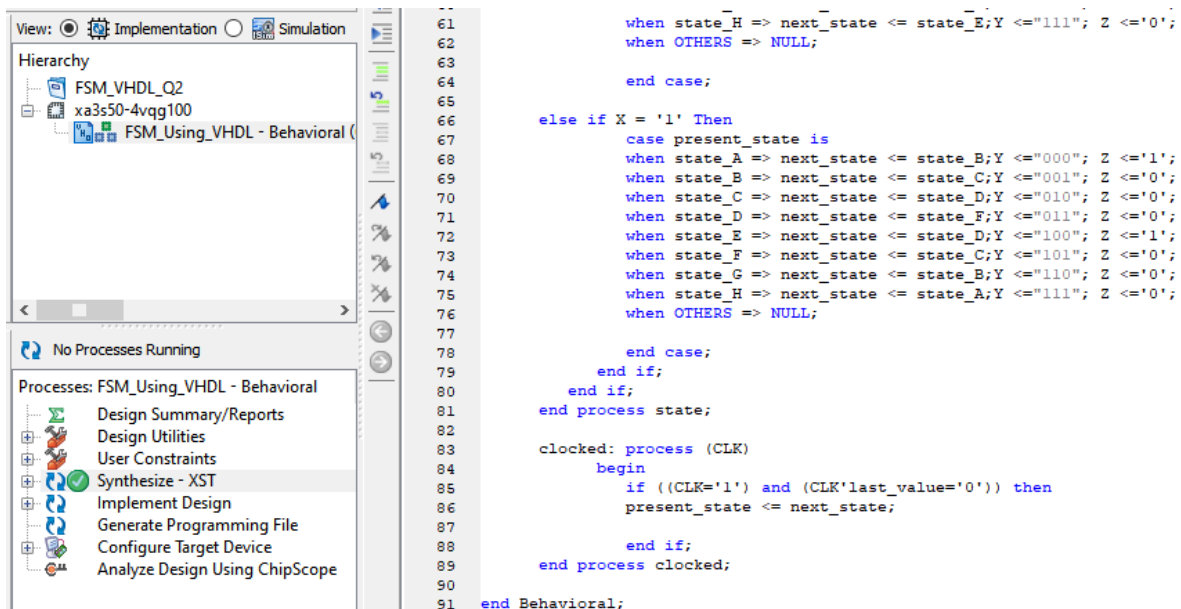
The value for input x is either 0 or 1.

Figure 26 & 27 show the VHDL code that has been written. Both have been successfully synthesized which proves the code to be error free.



```
31 entity FSM_Using_VHDL is
32   Port ( X : in  STD_LOGIC;
33         CLK : in  STD_LOGIC;
34         Z : out STD_LOGIC;
35         Y : out STD_LOGIC_VECTOR (2 downto 0));
36 end FSM_Using_VHDL;
37
38
39 architecture Behavioral of FSM_Using_VHDL 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
46
47
48 begin
49
50   state: process (present_state,X)
51   begin
52     if X = '0' then
53       case present_state is
54         when state_A => next_state <= state_E; Y <= "000"; Z <= '0';
55         when state_B => next_state <= state_D; Y <= "001"; Z <= '0';
56         when state_C => next_state <= state_G; Y <= "010"; Z <= '0';
57         when state_D => next_state <= state_G; Y <= "011"; Z <= '0';
58         when state_E => next_state <= state_F; Y <= "100"; Z <= '0';
59         when state_F => next_state <= state_H; Y <= "101"; Z <= '0';
60         when state_G => next_state <= state_G; Y <= "110"; Z <= '0';
```

Figure 26 - FSM in VHDL part 1



```
61   when state_H => next_state <= state_E; Y <= "111"; Z <= '0';
62   when OTHERS => NULL;
63
64   end case;
65
66   else if X = '1' Then
67     case present_state is
68       when state_A => next_state <= state_B; Y <= "000"; Z <= '1';
69       when state_B => next_state <= state_C; Y <= "001"; Z <= '0';
70       when state_C => next_state <= state_D; Y <= "010"; Z <= '0';
71       when state_D => next_state <= state_F; Y <= "011"; Z <= '0';
72       when state_E => next_state <= state_D; Y <= "100"; Z <= '1';
73       when state_F => next_state <= state_C; Y <= "101"; Z <= '0';
74       when state_G => next_state <= state_B; Y <= "110"; Z <= '0';
75       when state_H => next_state <= state_A; Y <= "111"; Z <= '0';
76       when OTHERS => NULL;
77
78     end case;
79   end if;
80 end if;
81 end process state;
82
83 clocked: process (CLK)
84 begin
85   if ((CLK='1') and (CLK'last_value='0')) then
86     present_state <= next_state;
87   end if;
88 end process clocked;
89
90
91 end Behavioral;
```

Figure 27 - FSM in VHDL part 2

Figure 28 & 29 below show the design summary. These show the logic utilization and that the code is error free.

FSM_Using_VHDL Project Status			
Project File:	FSM_VHDL_Q2.xise	Parser Errors:	No Errors
Module Name:	FSM_Using_VHDL	Implementation State:	Synthesized
Target Device:	xa3s50-4vqg100	• Errors:	No Errors
Product Version:	ISE 14.7	• Warnings:	No Warnings
Design Goal:	Balanced	• Routing Results:	
Design Strategy:	Xilinx Default (unlocked)	• Timing Constraints:	
Environment:	System Settings	• Final Timing Score:	

Figure 28 - Design Summary 1

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	2	768	0%
Number of Slice Flip Flops	3	1536	0%
Number of 4 input LUTs	4	1536	0%
Number of bonded IOBs	6	63	9%
Number of GCLKs	1	8	12%

Detailed Reports					
Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	Mon 10. May 15:19:53 2021	0	0	0
Translation Report					
Map Report					
Place and Route Report					
Power Report					
Post-PAR Static Timing Report					
Bitgen Report					

Figure 29 - Design Summary 2

Bibliography

- [1] "Electrical Technology," [Online]. Available: <https://www.electricaltechnology.org/2018/05/binary-multiplier-types-binary-multiplication-calculator.html>. [Accessed 20 February 2021].
- [2] "Multiplier Designing," Technobyte, [Online]. Available: <https://technobyte.org/multiplier-2-bit-3-bit-digital/>. [Accessed 20 February 20].
- [3] "Binary Multiplier," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Binary_multiplier. [Accessed 26 February 2021].
- [4] "Performance evaluation of parallel multipliers," [Online]. Available: <http://docsdrive.com/pdfs/medwelljournals/jeasci/2017/5186-5189.pdf>. [Accessed 02 04 2021].
- [5] A. Grunturu, "Analysis of Booths Multiplier," [Online]. Available: https://etd.ohiolink.edu/apexprod/rws_etd/send_file/send?accession=ysu1578311566143241&disposition=inline. [Accessed 06 May 2021].
- [6] "Booth's Algorithm," GeeksForGeeks, [Online]. Available: <https://www.geeksforgeeks.org/computer-organization-booths-algorithm/>. [Accessed 09 may 2021].
- [7] [Online]. Available: https://www.researchgate.net/figure/fig-2-Flow-Chat-of-Booth-Multiplier-Booths-algorithm-can-be-implemented-by-repeatedly_fig2_324819984.
- [8] "Wallace Tree Multiplier Designs," 2014. [Online]. Available: <https://core.ac.uk/download/pdf/234643291.pdf>. [Accessed 08 May 2021].
- [9] V. K. a. H. Saini, "A review on comparative performance Analysis of Different Digital Multipliers," [Online]. Available: https://www.ripublication.com/acst17/acstv10n5_54.pdf. [Accessed 7 May 2021].
- [10] "Design of Braun Multiplier," Oct 2012. [Online]. Available: <https://www.ijser.org/paper/Design-of-Braun-Multiplier-with-Kogge-Stone-Adder-Its-Implementation-on-FPGA.html#:~:text=Braun%20multiplier%20is%20a%20type,and%20one%20Ripple%20Carry%20Adder..> [Accessed 01 May 2021].
- [11] "Finite State Machines," Isaac Computer Science, [Online]. Available: https://isaacomputerscience.org/concepts/dsa_toc_fsm. [Accessed 1 May 2021].
- [12] "Karnaugh Maps," All About Circuits, [Online]. Available: <https://www.allaboutcircuits.com/textbook/digital/chpt-8/karnaugh-maps-truth-tables-boolean-expressions/>. [Accessed 05 April 2021].