

8 Bit x 8 Bit Multiplier & Finite State Machine

Leon Scammell
2017986

Laboratory Report

8 x 8 Multiplier

&

Finite State Machine

11/05/2022
—
Electronics II
—
Module Code - ACDA6001

Contents

1. Figures	3
2. Tables	4
3. 8 x 8 Multiplier	5
3.1 Abstract	5
3.2 Introduction	6
3.2 Procedure	7
3.2.1 How Does a Braun Multiplier Work?	7
3.2.2 Circuit Diagram	9
3.2.3 Coding the VHDL 8 x 8 Multiplier	12
3.3 Discussion – 8 x 8 Multiplier	23
3.4 Conclusion	23
4. Finite State Machine	25
4.1 Discrete Gate Solution	25
4.1.2 Introduction	25
4.1.3 Procedure	26
4.1.4. Discussion	33
4.1.5 Conclusion	33
4.2 VHDL Code	34
4.2.1 Abstract	34
4.2.2 Introduction	34
4.2.3 Procedure	35
4.2.4 Discussion	40
4.2.5 Conclusion	40
5. References	41

1. Figures

[16] Figure 1 - VHDL.....	6
[15] Figure 2 - ISE Design Suite Logo.....	6
[13] Figure 3 – Example of Binary Multiplication.....	7
Figure 4 - Full Adder Simplified Logic Diagram in the Multiplier.....	8
[14] Figure 5 - Half Adder Logical Circuit.....	8
[3.a] Figure 6 - Full Adder Logical Circuit.....	8
[3.b] Figure 7 - Half Adder Truth Table.....	8
[3.c] Figure 8 - Full Adder Truth Table	8
Figure 9 – Line 1 of The 8x8 Multiplier.....	9
Figure 10 - Line 1 and Line x Logic Circuit Diagram	10
Figure 11 - Full Logic Circuit Diagram of the 8x8 Multiplier	11
Figure 12 - Half Adder VHDL Code	12
Figure 13 - Half Adder Simulation.....	12
Figure 14 - Full Adder VHDL Code.....	13
Figure 15 - Full Adder Simulation	13
Figure 16 - 8 x 8 Multiplier Line 1 VHDL Code	14
Figure 17 - 8 x 8 Multiplier Line 1 RTL Schematic	15
Figure 18 - 8 x 8 Multiplier Line x Architecture Code	16
Figure 19 - 8 x 8 Multiplier Line x Entity Code	16
Figure 20 - 8 x 8 Multiplier, Line x Internal Signals	16
Figure 21 - 8 x 8 Multiplier, Line x AND Gate Assignments and Port Mapping	17
Figure 22 - 8 x 8 Multiplier, Line x Component Hierarchy	17
Figure 23 - 8 x 8 Multiplier, Line x RTL Model.....	18
Figure 24 - 8 x 8 Multiplier, Full Multi Entity Identification Section	19
Figure 25 - 8 x 8 Multiplier, Full Multi Architecture Section.....	20
Figure 26 - 8 x 8 Multiplier, Full Multiplier Signals	20
Figure 27 - 8 x 8 Multiplier, Full Multiplier Line Hierarchy	21
Figure 28 - 8 x 8 Multiplier, Full Multiplier Port Mapping.....	21
Figure 29 - 8 x 8 Multiplier, Full Multiplier Test Bench Stimulus Processes	22
Figure 30 - 8 x 8 Multiplier Behavioral Model Simulation (3 Examples).....	22
[17] Figure 31 - Multiplier Comparisons for Power and Area.....	23
Figure 32 - Finite State Machine, State Diagram	25
Figure 33 - Finite State Machine, Proteus Logic Circuit	29
Figure 34 - Simulation, State A (000) Transition to State D (011).....	30
Figure 35 - Simulation, State E (100) Transition to State F (101)	31
Figure 36 - Simulation, State H (111) Transition to State A (000)	32
Figure 37 – Finite State Machine (FSM), State Diagram	34
Figure 38 - VHDL FSM Entity Statements.....	35
Figure 39 - VHDL FSM Architecture.....	35
Figure 40 - VHDL FSM Process Transition Statements.....	36
Figure 41 - VHDL FSM Clock Description.....	36
Figure 42 - VHDL FSM Test Bench Inputs & Outputs	37
Figure 43 - VHDL FSM Port Mapping.....	37
Figure 44 - VHDL FSM Test Bench Simulation Process Statements	38
Figure 45 - FSM VHDL Simulation, Transition A/000 to B/001	38
Figure 46 - FSM VHDL, Simulation, Transition D/011 to E/100	39
Figure 47 - FSM VHDL, Simulation, Transition H/111 to A/000	39

2. Tables

Table 1 - Basic Transition/State Table.....	26
Table 2 - State Table	27
Table 3 - Karnaugh Map A.....	27
Table 4 - Karnaugh Map B.....	28
Table 5 - Karnaugh Map C.....	28
Table 6 – State Table, State A (000) Transition to State D (011).....	30
Table 7 – State Table, State E (100) Transition to State F (101)	31
Table 8 - State Table, State H (111) Transition to State A (000)	32

3. 8 x 8 Multiplier

The purpose producing this laboratory report, was to produce, test, and evaluate an 8 bit x 8 bit Braun multiplier and a finite state machine.

3.1 Abstract

The 8 bit x 8 bit multiplier that is being designed and coded involves the use of 64 AND gates, 49 full adders, and 7 half adders. The multiplier is split up into 8 lines, each linked together by 8 signals. There are 16 inputs labelled as 'a0 to a7', 'b0 to b7', and 16 outputs labeled as 'q0 to q15'.

This circuit will be first drawn up with each of the components having their own unique identification numbers to aid with coding. Then the multiplier will be coded in VHDL format on the coding software called Xilinx Design Suite.

The code for the circuit was tested by producing a 'test bench' code in Xilinx and running a behavioral simulation model. The output being the product of the two inputs. The inputs and outputs when arranged in reverse order form a binary number. This binary number is converted to unsigned decimal to make the multiplier easier to read for people who do not understand binary numbers.

When tested, the eight x eight multiplier functioned exactly as required. The output q provided the product of a and b and the VHDL code replicated the circuit in the software

3.2 Introduction

The 8 x 8 multiplier is a circuit that outputs the product of two inputs. The output being a 16 bit binary signal from two 8 bit inputs. This circuit will be produced in the coding software Xilinx ISE Design Suite, and must operate exactly as required purely from the code that is inputted to the software.

A circuit diagram must first be produced. This must clearly show how the circuit will be coded, with all inputs, outputs, intermediate signals, carry signals, AND gates, and the identifications of each component.

The circuit drawing and VHDL code, meaning very high-speed integrated circuit hardware description language [1], will be explained in detail throughout this report with the simulations results being shown.



[15] Figure 2 - ISE Design Suite Logo



[16] Figure 1 - VHDL

3.2 Procedure

3.2.1 How Does a Braun Multiplier Work?

At the input of the multiplier there are two 8 bit numbers and 16 bit output from the circuit to form the numbers in binary. In figure 3, two 8 bit numbers 00011101/29 and 00100011/89 multiply with each other to output the product 0000001111110111/1015. This can be seen in figure 3, in which two binary numbers are multiplied in the same manner as how decimal numbers are multiplied, using the standard binary multiplication method. This is the multiplication that the circuit must carry out to be a functioning Braun multiplier. In the calculations shown in figure 3, the single bits were multiplied 64 times (8 times for each line), so in the circuit there must be 64 calculations also. This is why there must be 64 adders in the circuit to perform these calculations.

$$\begin{array}{r}
 00011101 \\
 \times 00100011 \\
 \hline
 00011101 \\
 00000000 \\
 00000000 \\
 00000000 \\
 00000000 \\
 00000000 \\
 00011101 \\
 00000000 \\
 00000000 \\
 \hline
 0000001111110111
 \end{array}$$

[13] Figure 3 – Example of Binary Multiplication

Each full and half adder is the building block of the multiplier. The a and b inputs will be first multiplied by the AND gate, and then summed by the full adder circuit. Full adders have 3 inputs and 2 outputs. Inputs being a, b, and Cin, and the outputs being sum and carry output. The a input is identical on each line so this goes down to the next line. The b input is multiplied by every other a-input in that line. The other input is the sum from the previous line's full adders. The carry input is the carry from the previous adder. This can be seen in figure 4. Each of these full adders forms a single bit of the partial binary product number. There will be 8 of these per line, with each line forming the 8 bit partial binary product for the binary multiplication to be completed, exactly the same as the manual calculation method shown in figure 3. The only difference of a half adder is that there is no carry input, which is not required here.

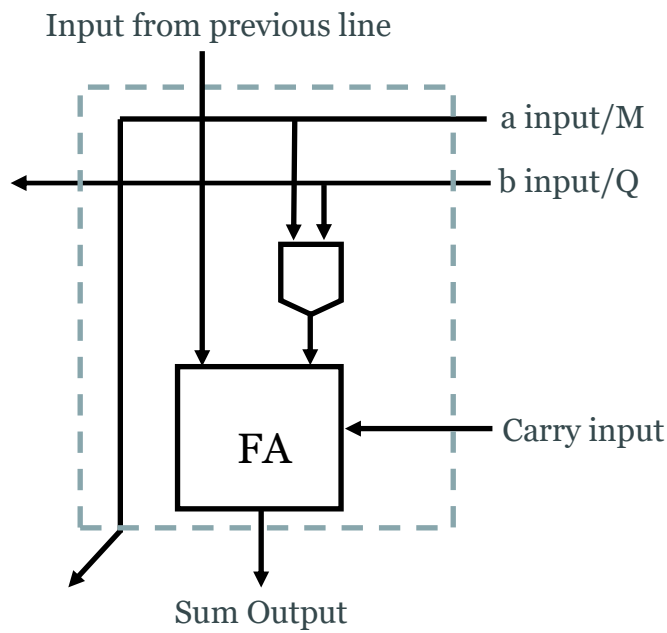
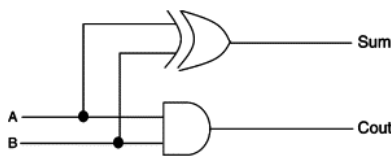


Figure 4 - Full Adder Simplified Logic Diagram in the Multiplier

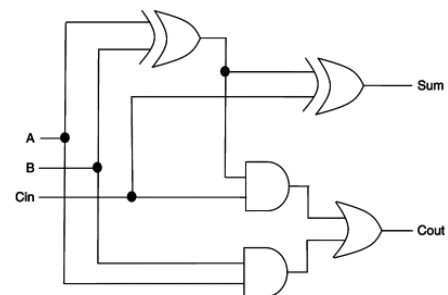
The half adder can be seen in figure 4 and works by using a xor gate to output the sum when a and b are different. The carry output uses an and gate for when a and b are 1, which produces a 2 which can't be shown as one binary digit, so a carry output is produced by the and gate [2]. This replicates the method for manually multiplying binary as when two ones are being multiplied in the method, a one gets carried over for the next calculation or full adder whilst providing an external binary output of 0. This can be seen in the truth table in figures 7 & 8 and logic circuits in figures 5 & 6. The full adder has an additional xor gate, or gate, and another and gate.



[14] Figure 5 - Half Adder Logical Circuit

A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

[3.b] Figure 7 - Half Adder Truth Table



[3.a] Figure 6 - Full Adder Logical Circuit

A	B	Cin	Sum	Cout
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

[3.c] Figure 8 - Full Adder Truth Table

3.2.2 Circuit Diagram

The circuit diagram of the whole 8 x 8 multiplier was produced in a flow chart producing software called Yed. Each of the lines was clearly identified with their line name, and every component and signal was labeled.

Shown in figure 9 is the first line of the 8 x 8 multiplier. The AND gates are in place to begin the computation of the product of a and b for each gate. In line 1, each a0 to a7 input and b0 external input connects to the AND gates. The a inputs all represent the same input. This is outputted to line x via the internal signals q1_1 and q2_1 etc. The first (right) AND gate's output becomes the multipliers q0 output. The outputs of line 1 connect to the 'Cin' inputs of the full and half adders in line x.

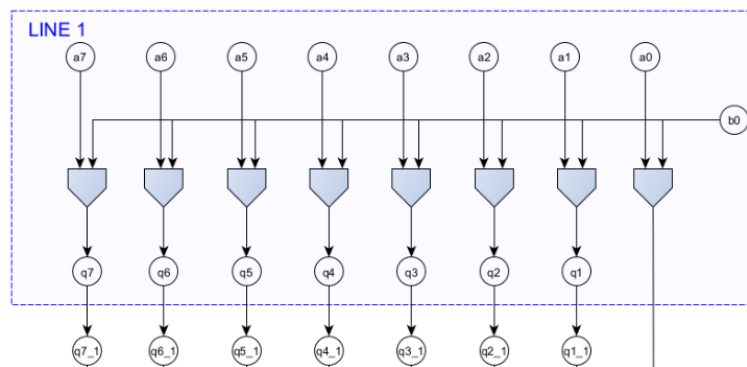


Figure 9 – Line 1 of The 8x8 Multiplier

In line x which is shown in figure 10, the output of each AND gate is then connected to the input of the half adder and full adders, identical as line 1. The half and full adders sum the product which is produced by the AND gates of line x and line 1. The half adder provides a sum output to the next line, and outputs a carry signal to the next full adder. The full adders receive the output from the output from the previous line, whilst also receiving the carry signal from the half adder or previous full adder. The full adders then output a sum signal to the next line. The last full adder (full7) does not have a 'Cin' from line 1, so this input is fixed at 0, being called xc.

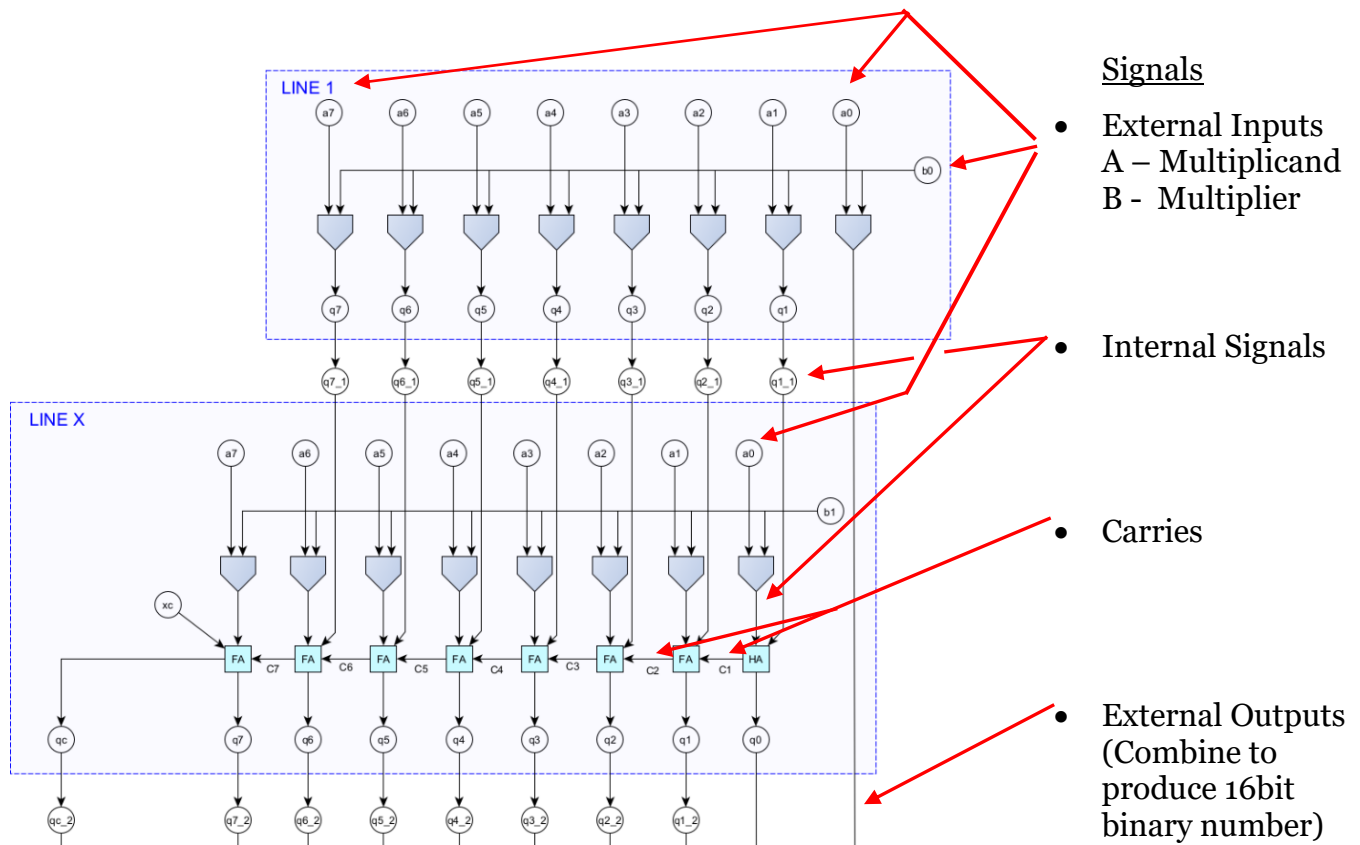
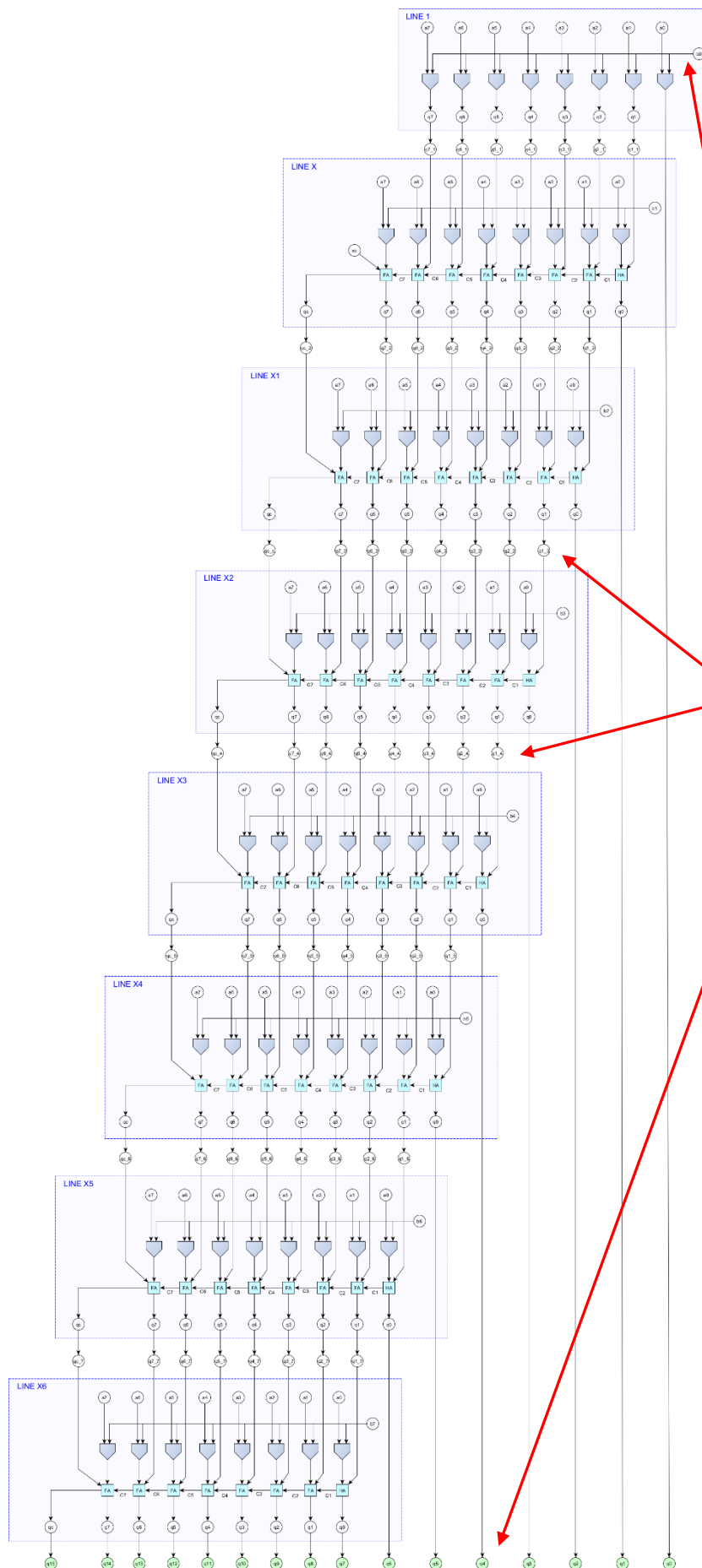


Figure 10 - Line 1 and Line x Logic Circuit Diagram

Figure 11 shows the full circuit diagram with the multiplier being split up into 8 lines. The components in each line are identical, but the signals change to $q1_2$ before line $x1$ and $q1_3$ before line $x2$ for example. The half adder of each line outputs an external output, however in $linex6$ every output is external for the multiplier. These external outputs all combine to produce the 16bit binary output. Qc is the carry output from the 7th full adder to the Cin of the final full adder of the next line.



Description

- External Inputs
- Signals between lines
- External outputs

Figure 11 - Full Logic Circuit Diagram of the 8x8 Multiplier

3.2.3 Coding the VHDL 8 x 8 Multiplier

3.2.3.1 Half Adder

The half and full adders were coded in VHDL format. The logic diagrams were followed to produce the code, with the xor and and gate being written in for the sum and carry outputs as shown in figure 12. Figure 13 shows the simulation that matches the half adder truth table, shown in figure 7.

```
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
48
```

Figure 12 - Half Adder VHDL Code

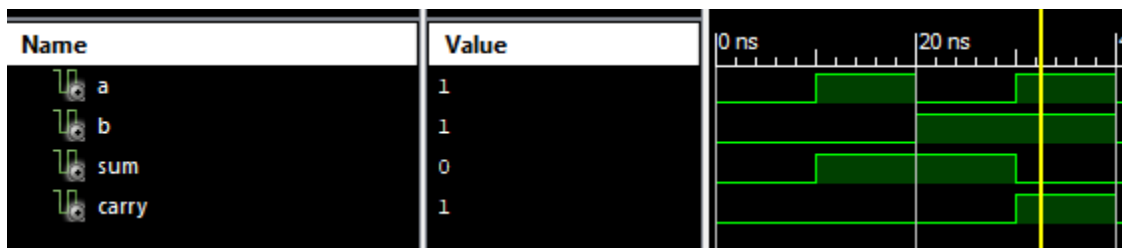


Figure 13 - Half Adder Simulation

3.2.3.2 Full Adder

The same process was followed to produce the full adder, shown in figure 14. With the additional carry input and and xor gate, as well as internal signals between them as shown in the logic diagram in figure 6. The simulation shown in figure 15 verified the code works as it matches the full adder truth table, shown in figure 8.

```

31
32 entity Full_Adder is
33     Port ( A : in  STD_LOGIC;
34           B : in  STD_LOGIC;
35           Cin : 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
42     signal AxB : STD_LOGIC;
43     signal AxBandC : STD_LOGIC;
44     signal AandB : STD_LOGIC;
45
46 begin
47
48     AxB <= A xor B;
49     Sum <= AxB xor Cin;
50
51     AxBandC <= AxB and Cin;
52     AandB <= A and B;
53     Carry <= AxBandC or AandB;
54

```

Figure 14 - Full Adder VHDL Code

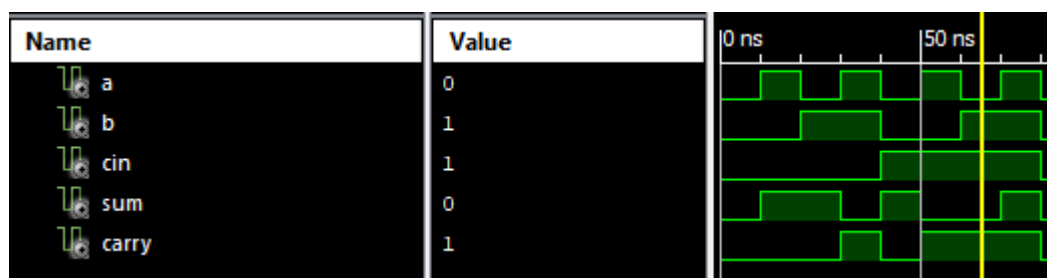


Figure 15 - Full Adder Simulation

3.2.3.3 8 x 8 Multiplier Line 1

Line 1 shown in figure 16 was coded by entering all external inputs and outputs into the entity section. The and gates were included with their inputs and outputs in the architecture section. A test bench was then produced to tell the code to operate each input in turn to display the calculations for the full circuit later on.

```

32 entity eight_x_eight_line_1 is
33     Port ( a0 : in  STD_LOGIC;
34            a1 : in  STD_LOGIC;
35            a2 : in  STD_LOGIC;
36            a3 : in  STD_LOGIC;
37            a4 : in  STD_LOGIC;
38            a5 : in  STD_LOGIC;
39            a6 : in  STD_LOGIC;
40            a7 : in  STD_LOGIC;
41            b0 : in  STD_LOGIC;
42            q0 : out STD_LOGIC;
43            q1 : out STD_LOGIC;
44            q2 : out STD_LOGIC;
45            q3 : out STD_LOGIC;
46            q4 : out STD_LOGIC;
47            q5 : out STD_LOGIC;
48            q6 : out STD_LOGIC;
49            q7 : out STD_LOGIC);
50 end eight_x_eight_line_1;
51
52 architecture Behavioral of eight_x_eight_line_1 is
53
54 begin
55
56     q0 <= a0 and b0;  -- AND gates
57     q1 <= a1 and b0;
58     q2 <= a2 and b0;
59     q3 <= a3 and b0;
60     q4 <= a4 and b0;
61     q5 <= a5 and b0;
62     q6 <= a6 and b0;
63     q7 <= a7 and b0;

```

Figure 16 - 8 x 8 Multiplier Line 1 VHDL Code

The register transfer language model of the circuit was produced by the software from the code to show the logic of the circuit, as seen in figure 17. This matches the circuit diagram shown in figure 9 and proves the code is correct.

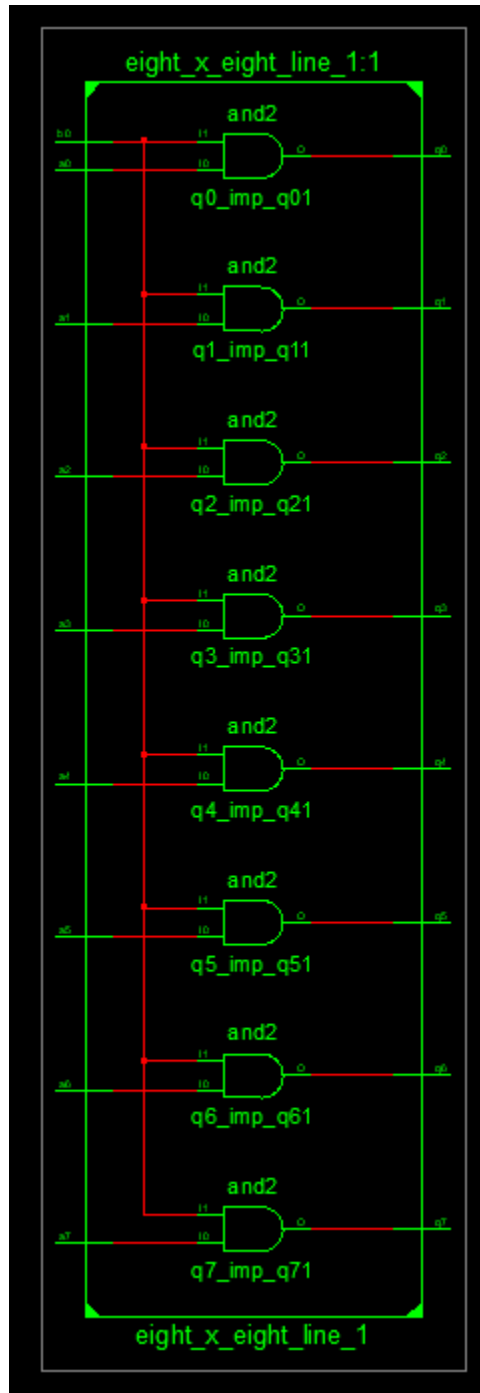


Figure 17 - 8 x 8 Multiplier Line 1 RTL Schematic

3.2.3.4 8 x 8 Multiplier Line x

To produce line x, the half and full adders previously produced needed to be copied in to the code. First the external inputs and outputs were included in the entity section. Then the adders were included in the architecture section as components of the code. Their inputs and outputs were written in, as shown in figures 18,19, and 20.

```

32 entity eight_x_eight_line_x is
33     Port ( a0 : in  STD_LOGIC;
34           a1 : in  STD_LOGIC;
35           a2 : in  STD_LOGIC;
36           a3 : in  STD_LOGIC;
37           a4 : in  STD_LOGIC;
38           a5 : in  STD_LOGIC;
39           a6 : in  STD_LOGIC;
40           a7 : in  STD_LOGIC;
41           b0 : in  STD_LOGIC;
42           x1 : in  STD_LOGIC;
43           x2 : in  STD_LOGIC;
44           x3 : in  STD_LOGIC;
45           x4 : in  STD_LOGIC;
46           x5 : in  STD_LOGIC;
47           x6 : in  STD_LOGIC;
48           x7 : in  STD_LOGIC;
49           xc : in  STD_LOGIC;
50           q0 : out STD_LOGIC;
51           q1 : out STD_LOGIC;
52           q2 : out STD_LOGIC;
53           q3 : out STD_LOGIC;
54           q4 : out STD_LOGIC;
55           q5 : out STD_LOGIC;
56           q6 : out STD_LOGIC;
57           q7 : out STD_LOGIC;
58           qc : out STD_LOGIC);
59 end eight_x_eight_line_x;

```

Figure 19 - 8 x 8 Multiplier Line x Entity Code

```

63 COMPONENT half_adder is
64     PORT(
65         A : IN  std_logic;
66         B : IN  std_logic;
67         Sum : OUT std_logic;
68         Carry : OUT std_logic
69     );
70 END COMPONENT;
71
72 COMPONENT full_Adder is
73     PORT(
74         A : IN  std_logic;
75         B : IN  std_logic;
76         Cin : IN std_logic;
77         Sum : OUT std_logic;
78         Carry : OUT std_logic
79     );
80 END COMPONENT;

```

Figure 18 - 8 x 8 Multiplier Line x Architecture Code

```

83 signal a0andb0 :STD_LOGIC;  -- Internal AND Gate Signals
84 signal a1andb0 :STD_LOGIC;
85 signal a2andb0 :STD_LOGIC;
86 signal a3andb0 :STD_LOGIC;
87 signal a4andb0 :STD_LOGIC;
88 signal a5andb0 :STD_LOGIC;
89 signal a6andb0 :STD_LOGIC;
90 signal a7andb0 :STD_LOGIC;
91
92 signal c1  :STD_LOGIC;      -- Internal Carry Signals
93 signal c2  :STD_LOGIC;
94 signal c3  :STD_LOGIC;
95 signal c4  :STD_LOGIC;
96 signal c5  :STD_LOGIC;
97 signal c6  :STD_LOGIC;
98 signal c7  :STD_LOGIC;

```

Figure 20 - 8 x 8 Multiplier, Line x Internal Signals

The internal carries and and gates were written in as intermediate signals, shown in figure 20 above. The and gates were then each assigned an a input and a b input, and then named to identify each one. The inputs and outputs for each half and full adder were then port mapped to tell the code what inputs and outputs are involved and where, as shown in figure 21. The port mapping also including all internal carry signals.

```

103      a0andb0 <= a0 and b0;          -- Assignments for AND Gates
104      a1andb0 <= a1 and b0;
105      a2andb0 <= a2 and b0;
106      a3andb0 <= a3 and b0;
107      a4andb0 <= a4 and b0;
108      a5andb0 <= a5 and b0;
109      a6andb0 <= a6 and b0;
110      a7andb0 <= a7 and b0;
111
112
113      half1 : half_adder              -- Port map the half adder
114      port map(a0andb0,x1,q0,c1);
115      full1 : full_adder              -- Port map the 1st full adder, looking right to left
116      port map(a1andb0,x2,c1,q1,c2);
117      full12 : full_adder             -- Port map the 2nd full adder
118      port map(a2andb0,x3,c2,q2,c3);
119      full13 : full_adder             -- Port map the 3rd full adder
120      port map(a3andb0,x4,c3,q3,c4);
121      full14 : full_adder             -- Port map the 4th full adder
122      port map(a4andb0,x5,c4,q4,c5);
123      full15 : full_adder             -- Port map the 5th full adder
124      port map(a5andb0,x6,c5,q5,c6);
125      full16 : full_adder             -- Port map the 6th full adder
126      port map(a6andb0,x7,c6,q6,c7);
127      full17 : full_adder             -- Port map the 7th full adder
128      port map(a7andb0,xc,c7,q7,qc);

```

Figure 21 - 8 x 8 Multiplier, Line x AND Gate Assignments and Port Mapping

The files for the half and full adders were then copied in to the code by selecting 'add source' so that the test bench and code is inserted to this code for the circuit to function as required, shown in figure 22.

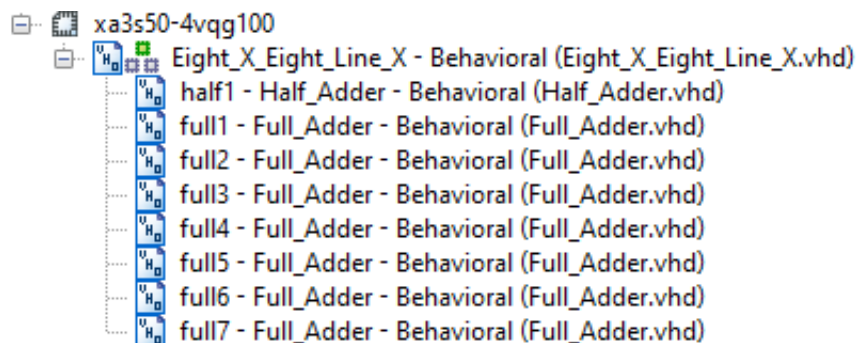


Figure 22 - 8 x 8 Multiplier, Line x Component Hierarchy

The RTL model of line x was then produced by the software with the code, shown in figure 21. This matches the logic circuit diagram shown in figure 10, so the code is correct and ready to be copied into the multiplier.

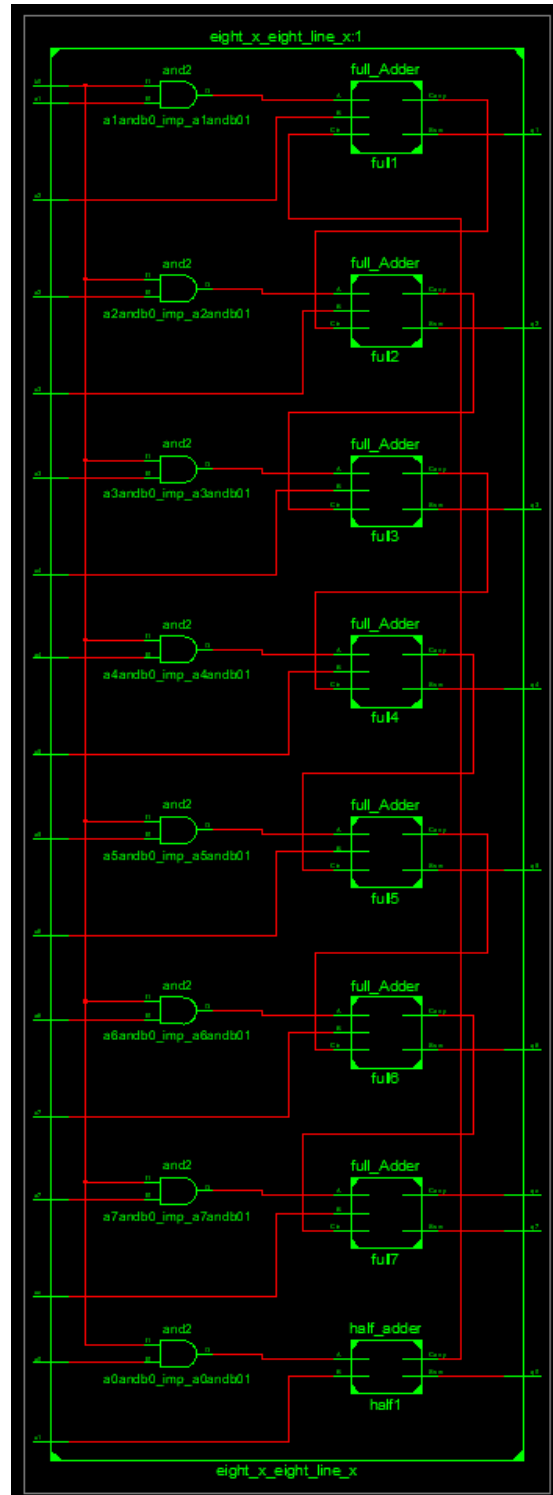


Figure 23 - 8 x 8 Multiplier, Line x RTL Model

3.2.3.5 8 x 8 Multiplier, Full Multiplier

First step was to identify all of the external inputs and outputs to the multiplier in the entity section of the code, including all as, bs and qs as shown in figure 24.

```

32 entity Eight_x_eight_multi is
33     Port ( a0 : in  STD_LOGIC;
34           a1 : in  STD_LOGIC;
35           a2 : in  STD_LOGIC;
36           a3 : in  STD_LOGIC;
37           a4 : in  STD_LOGIC;
38           a5 : in  STD_LOGIC;
39           a6 : in  STD_LOGIC;
40           a7 : in  STD_LOGIC;
41           b0 : in  STD_LOGIC;
42           b1 : in  STD_LOGIC;
43           b2 : in  STD_LOGIC;
44           b3 : in  STD_LOGIC;
45           b4 : in  STD_LOGIC;
46           b5 : in  STD_LOGIC;
47           b6 : in  STD_LOGIC;
48           b7 : 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 Eight_x_eight_multi;

```

Figure 24 - 8 x 8 Multiplier, Full Multi Entity Identification Section

Line1 and linex were then included in the architecture section with all external inputs and outputs, shown in figure 25.

```

69     COMPONENT eight_x_eight_line_1 is
70     PORT(a0 : in  std_logic;
71          a1 : in  std_logic;
72          a2 : in  std_logic;
73          a3 : in  std_logic;
74          a4 : in  std_logic;
75          a5 : in  std_logic;
76          a6 : in  std_logic;
77          a7 : in  std_logic;
78          b0 : in  std_logic;
79          q0 : out std_logic;
80          q1 : out std_logic;
81          q2 : out std_logic;
82          q3 : out std_logic;
83          q4 : out std_logic;
84          q5 : out std_logic;
85          q6 : out std_logic;
86          q7 : out std_logic
87     );
88     END COMPONENT;
89
90     COMPONENT eight_x_eight_line_x is
91     PORT(a0 : in  std_logic;
92          a1 : in  std_logic;
93          a2 : in  std_logic;
94          a3 : in  std_logic;
95          a4 : in  std_logic;
96          a5 : in  std_logic;
97          a6 : in  std_logic;
98          a7 : in  std_logic;
99          b0 : in  std_logic;
100         x1 : in  std_logic;
101         x2 : in  std_logic;
102         x3 : in  std_logic;
103         x4 : in  std_logic;
104         x5 : in  std_logic;
105         x6 : in  std_logic;
106         x7 : in  std_logic;
107         xc : in  std_logic;
108         q0 : out std_logic;
109         q1 : out std_logic;
110         q2 : out std_logic;
111         q3 : out std_logic;
112         q4 : out std_logic;
113         q5 : out std_logic;
114         q6 : out std_logic;
115         q7 : out std_logic;
116         qc : out std_logic);
117

```

Figure 25 - 8 x 8 Multiplier, Full Multi Architecture Section

Next the internal signals between each line were included, telling the code that they are signals and their individual identifications. Shown in figure 26.

```

120     signal q1_1 : std_logic;      -- Line1 to linex signals
121     signal q2_1 : std_logic;
122     signal q3_1 : std_logic;
123     signal q4_1 : std_logic;
124     signal q5_1 : std_logic;
125     signal q6_1 : std_logic;
126     signal q7_1 : std_logic;
127
128     signal q1_2 : std_logic;      -- Linex to linex1 signals
129     signal q2_2 : std_logic;
130     signal q3_2 : std_logic;
131     signal q4_2 : std_logic;
132     signal q5_2 : std_logic;
133     signal q6_2 : std_logic;
134     signal q7_2 : std_logic;
135     signal qc_2 : std_logic;

```

Figure 26 - 8 x 8 Multiplier, Full Multiplier Signals

A test bench module was produced to tell the program to operate each input in turn to show all possible combinations for the simulation to work, as shown in figure 29.

```

156  -- Stimulus process
157  stim_proc1: process
158  begin
159      -- hold reset state for 100 ns.
160      wait for 10 ns;
161      a0 <= not a0;
162  end process;
163
164  -- Stimulus process
165  stim_proc2: process
166  begin
167      -- hold reset state for 100 ns.
168      wait for 20 ns;
169      a1 <= not a1;
170  end process;
171
172  -- Stimulus process
173  stim_proc3: process
174  begin
175      -- hold reset state for 100 ns.
176      wait for 40 ns;
177      a2 <= not a2;
178  end process;
179
180  -- Stimulus process
181  stim_proc4: process
182  begin
183      -- hold reset state for 100 ns.
184      wait for 80 ns;
185      a3 <= not a3;
186  end process;
187

```

Figure 29 - 8 x 8 Multiplier, Full Multiplier Test Bench Stimulus Processes

3.2.3.6. 8 x 8 Multiplier Test Results

Finally the multiplier was tested by running a behavioral model simulation as shown in figure 30. With the inputs and outputs positioned in reverse order (a7-a0, b7-b0, q15-q0), this provides the 8bit and 16bit binary numbers that are being multiplied. These were converted to unsigned decimal. The circuits functions exactly as required, with the 16 bit binary output number being the product of the two 8 bits inputs, a and b. Numbers have been converted to unsigned decimal.

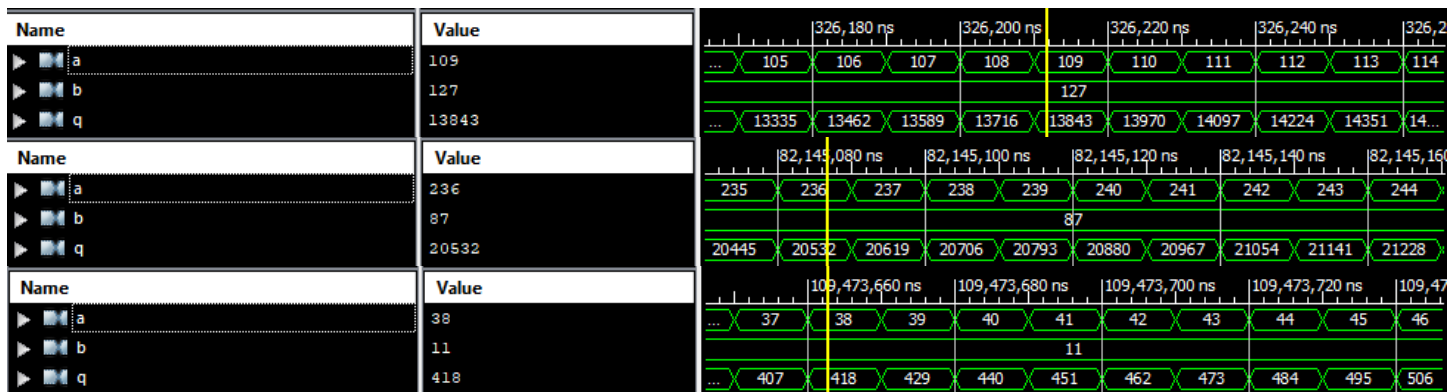


Figure 30 - 8 x 8 Multiplier Behavioral Model Simulation (3 Examples)

3.3 Discussion – 8 x 8 Multiplier

The multiplier used in this experiment was a 8 x 8 Braun Multiplier, which is relatively simple to design, as it follows the manual binary multiplication method. The logic circuit diagram was easy to follow to produce the code, and the VHDL code functioned exactly as required once tested. The code accurately replicated the logic diagram, meaning the code was written correctly and in a manner that the software to understand. The experiment was a success therefor as the test results were correct and matched the logic.

The other multiplier options that could have been used include Braun multipliers, Baugh-Wooley Multiplier, Modified Booth Multiplier, and Wallace Tree multipliers [3]. For low power applications, the modified Booth multiplier and Baugh-Wooley multipliers are ideal as they consume low power. Braun and Wallace Tree multipliers consume more power but can process operands greater than 16 bit, whereas the Baugh Wooley has a max operand of 16bit. Braun and Wallace Tree are the fastest multipliers. Braun and Wallace Tree are the largest however in physical scale, as show nin figure 31.

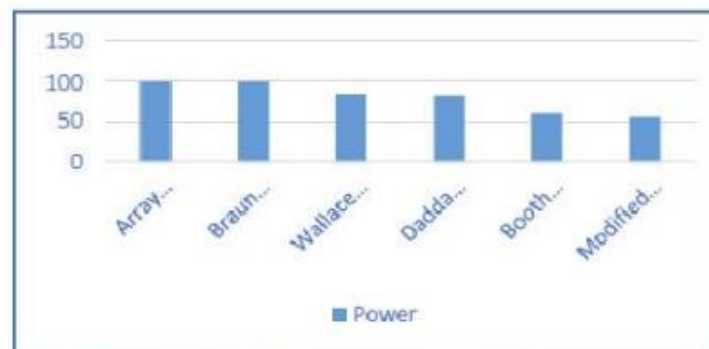


Fig.7 Comparison of different types of Multipliers with respect to power

LUTs

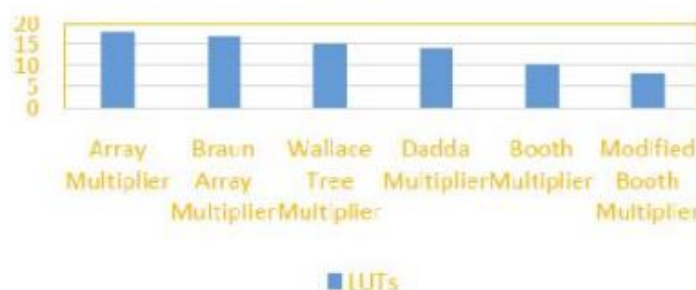


Fig.8 Comparison of different types of Multipliers with respect to area

[17] Figure 31 - Multiplier Comparisons for Power and Area

3.4 Conclusion

To conclude, once the logic diagram was produced, the overall system could be easily visualized and all of the various internal signals and operations could be identified. This made writing the VHDL code very efficient, as the identification numbers for each component and signal were clearly shown in

the flow chart for the logic. The software allowed each line of the multiplier to be coded separately, combined in the same file, and then mapped to connect them to each other. The software then produced the logic from the inputted code, allowing a visual representation of the logic circuit to be seen. Then a simulation could be carried out to test that the logic circuit functioned exactly as shown in the logic diagram, which the simulated circuit did successfully. The logic circuits produced by the software, and the test results were saved and shown in this report to prove that the code was written correctly. The code was written with some descriptions next to the various stages of the code to make the code more user friendly. The code was also generally written in a neat and organized manner, with spaces between each section to make reading the code and potential fault finding in future much easier. Overall, when combining the planning of the code via the utilization of the logic circuit diagram, and the implementation of the code with the aid of the diagram, the experiment was successful and the logic circuit was replicated with the VHDL modules.

4. Finite State Machine

4.1 Discrete Gate Solution

4.1.2 Introduction

Figure 31 shows the state diagram of the finite state machine that is being replicated via a discrete gate solution. This state diagram includes 8 different states and 16 transitions between them. In this FSM the states are each identified by their own unique 3 bit identification number. It also starts with an initial state, being state A or 000, and a final state which is H or 111.

A state diagram is a method of graphically representing the various decision making and configuration states that a circuit or system undertakes during its whole operation. The different configurations of the system is displayed as states and arrows show the transitions between these states, similarly to a flow chart. [4] The state diagram shows all of the potential internal operating states of a circuit or system, from the initial state, to the final state.

The purpose of this experiment is to replicate this state diagram with logic circuitry, being simulated in software called Proteus. Logic circuits involve a combination of logic gates, such as AND and OR gates, and allow an electrical circuit to perform logic operations [5]. This allows a circuit to receive a number of inputs, perform some form of calculation depending on the design of the logic, and provide an output that is applicable to the intended application [6]. The logic circuitry will then be simulated to prove that it functions accordingly to the state diagram.

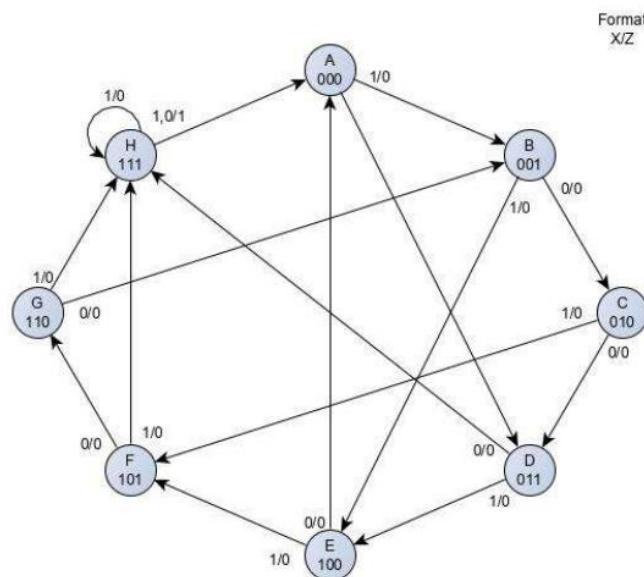


Figure 32 - Finite State Machine, State Diagram

4.1.3 Procedure

4.1.3.1 State Table

The first step of producing the logic circuit design, is to analyze all of the various states and transitions of the state diagram, and produce a state table. The state diagram has an initial and a final state, so the table will be completed by working through every state, with all transitions being included to show exactly how the logic will transition between each state depending on the inputs being received. A state table is a tabular method of presenting the behavioral model of a finite state machine, involving information on the systems response to all potential external inputs and outputs. [7]

First a basic transition table was produced, shown in table 1, which simply shows which state the logic will transition to depending on the input condition. [8] The present state is shown with the 3 bit binary identification number, the input being received, and the next state that the logic transitions to.

State Table		
Present State	Input	Next State
A (000)	0	D
	1	B
B (001)	0	C
	1	E
C (010)	0	D
	1	F
D (011)	0	H
	1	E
E (100)	0	A
	1	F
F (101)	0	G
	1	H
G (110)	0	B
	1	H
H (111)	0	A
	1	H

Table 1 - Basic Transition/State Table

A more detailed state table was then produced from the state diagram to display all potential states that the finite state machine will transition between, as well as all concurrent inputs and outputs. This table is shown in table 2. This includes the present state (showing the 3 bit identification as A, B, and C), input (X), next state, and the output (Z), following the standard state table design. [9]

State Table								
	Present State			Input	Next State			Output
	A	B	C	X	A	B	C	Z
A	0	0	0	0	0	1	1	0
	0	0	0	1	0	0	1	0
B	0	0	1	0	0	1	0	0
	0	0	1	1	1	0	0	0
C	0	1	0	0	0	1	1	0
	0	1	0	1	1	0	1	0
D	0	1	1	0	1	1	1	0
	0	1	1	1	1	0	0	0
E	1	0	0	0	0	0	0	0
	1	0	0	1	1	0	1	0
F	1	0	1	0	1	1	0	0
	1	0	1	1	1	1	1	0
G	1	1	0	0	0	0	1	0
	1	1	0	1	1	1	1	0
H	1	1	1	0	0	0	0	1
	1	1	1	1	1	1	1	0

Table 2 - State Table

From the state table, three Karnaugh maps were produced to allow the components of the logic circuit to be selected. These can be seen in tables 3, 4, and 5. Four-variable Karnaugh maps were produced with a matrix of 4x4 cells, with the four state variables a, b, c, and x being included. This is completed by looking at each state number and input, then inserting each bit of the next state that is being addressed by the Karnaugh map into the associated cell in the map. [10]

4.1.3.2. Karnaugh Maps

Karnaugh Map A					
DA	AB CX	00	01	11	10
	00	0	0	0	0
	01	0	1	1	1
	11	1	1	1	1
	10	0	1	0	1

Table 3 - Karnaugh Map A

Karnaugh maps allows for the selection of the components in the logic circuit. This is done by grouping the (1) variables in the map in vertical or horizontal groups of 2 or 4. Then the state variables are selected from the grouped/looped cells with their logic function being chosen depending

on whether the letter is a 0 (not) or a 1. The cells must be grouped as largely as possible to reduce the logic complexity. [10] From the map shown in table 3, the AND-OR network, $DA = \overline{A.C.X} + B.X + \overline{A.B.C} + A./B.C.X + A./B.X$ was produced. These are the inputs that will be connected to AND and OR gates as indicated by the Karnaugh maps. [11]

Karnaugh Map B					
DB	AB CX	00	01	11	10
	00	1	1	0	0
	01	0	0	1	0
	11	0	0	1	1
	10	1	1	0	1

Table 4 - Karnaugh Map B

Karnaugh map B shown in table 4 produced the logic functions, $DB = \overline{A./C./X} + \overline{A.C./X} + A.B.X + A./B.C.$

Karnaugh Map C					
DC	AB CX	00	01	11	10
	00	1	1	1	0
	01	1	1	1	1
	11	0	0	1	1
	10	0	1	0	0

Table 5 - Karnaugh Map C

Karnaugh map C shown in table 5 produced the logic functions, $DC = \overline{A./C} + \overline{A.B.C./X} + A.B./C + A.X.$

4.1.3.3 Proteus Logic Circuit

The next step of this experiment was to produce the logic circuit in software called Proteus. First the inputs a, b, c, and x were entered into the circuit. Then inverters connected next to them for the 'not' signals for each input. AND gates were then inserted to the circuit by following the AND-OR networks identified by the Karnaugh maps. The AND-OR networks showed how many inputs each AND gate required and what signals they were to connect to. The inputs (a, /a, b, /b, c, /c, x and /x) were then connected to their respective AND gates by following the AND-OR networks. The AND gates for each Karnaugh map group were then connected to OR gates. Each OR gate provided each bit for the next state identification. The top OR gate (A) was placed as a NOR gate as the software did not have a 5 input OR gate available, so a 5 input NOR gate was used with an inverter.

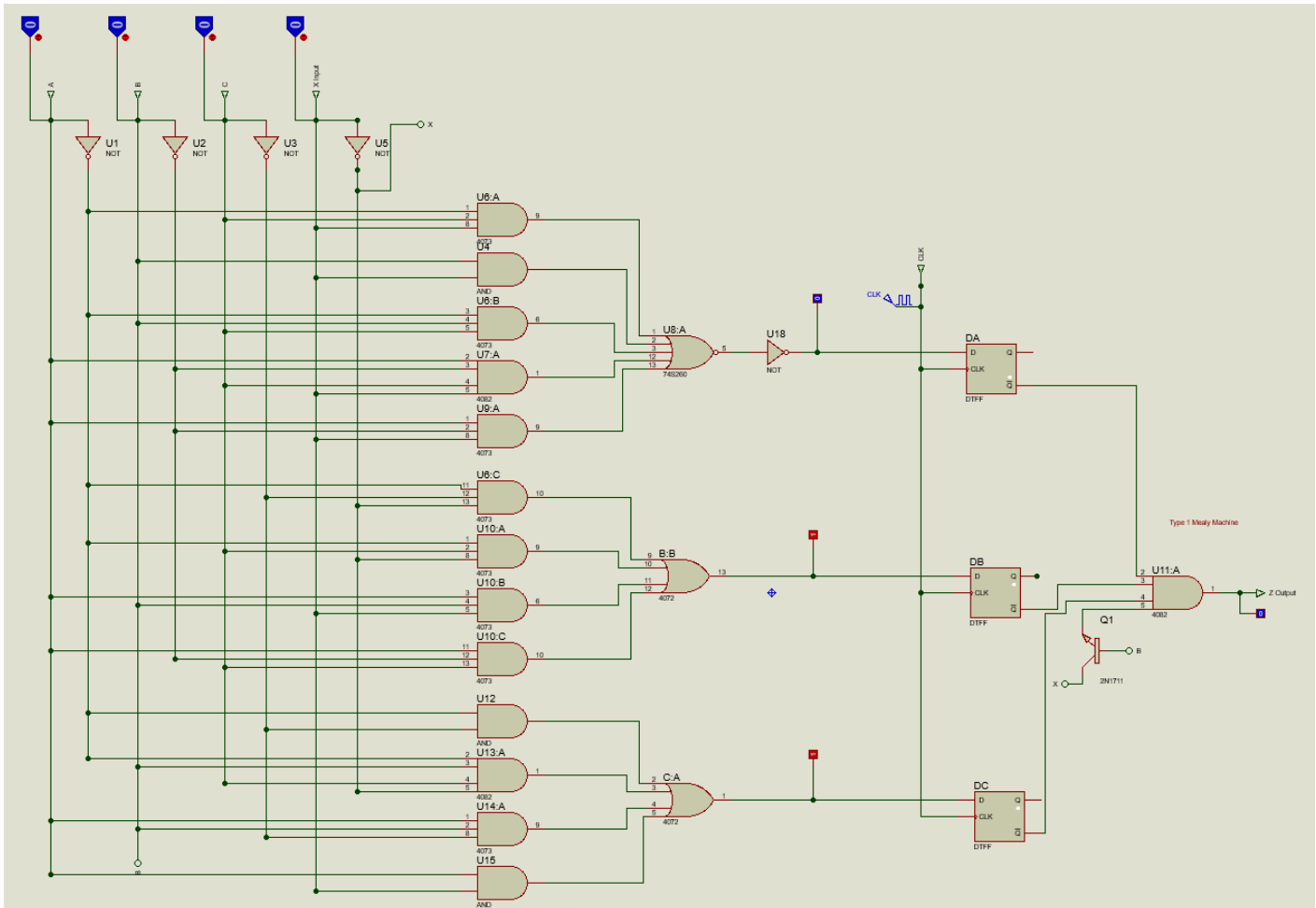


Figure 33 - Finite State Machine, Proteus Logic Circuit

In the state table (table 2), the transition from state H (111) to state A (000) with an input of 0, the output is 1. This is the only transition that has an output of 1. For the circuit to produce this output, three D-type flip flops were connected to the OR gates of each AND-OR network. The /Q outputs of the flip flops were then connected to a final AND gate. This AND gate will switch on when all three outputs are 0 (000) for the final transition in the state diagram (figure 31), and with /X. A transistor

was added to operate the final AND gate when B is on during state H (111), ensuring this output will not operate in state E (100) with /X.

4.1.3.4. Testing the Logic Circuit in Proteus & Results

To test the logic circuit, a clock input was connected to the clk input on the D-type flip flops, this will reset them once a second to test the outputs. Logic toggles were connected to each input and to the inverters for the NOT inputs. Logic probes were connected to each output to check the circuit's accuracy to the state table and state diagram. All of the states were tested and the logic circuit transitioned to the correct next states. 3 of these tests are shown below in table 6 & figure 33, table 7 & .

	Present State			Input	Next State			Output
	A	B	C		A	B	C	
A	0	0	0	0	0	1	1	0

Table 6 – State Table, State A (000) Transition to State D (011)

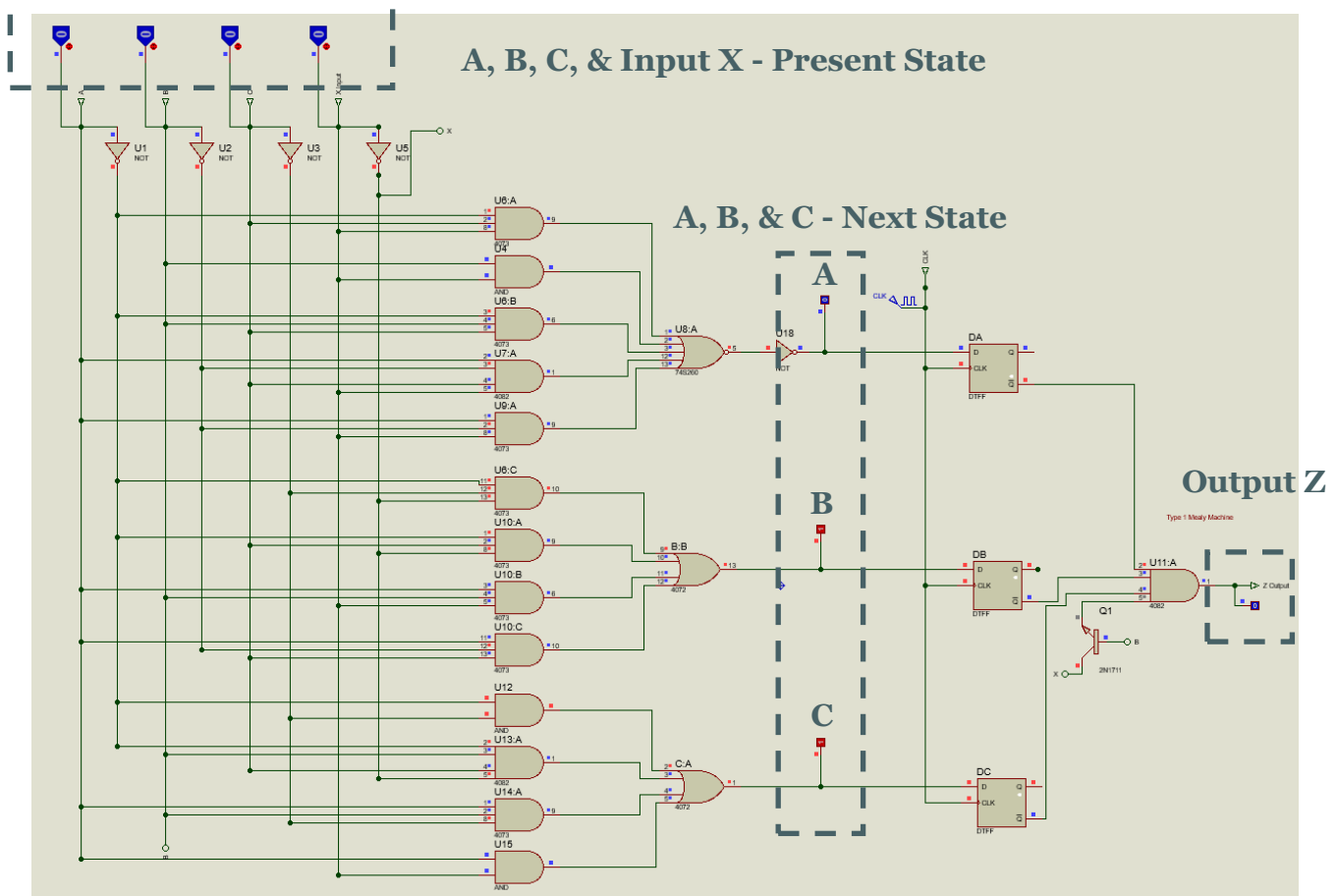


Figure 34 - Simulation, State A (000) Transition to State D (011)

	Present State			Input	Next State			Output
	A	B	C	X	A	B	C	Z
E	1	0	0	1	1	0	1	0

Table 7 – State Table, State E (100) Transition to State F (101)

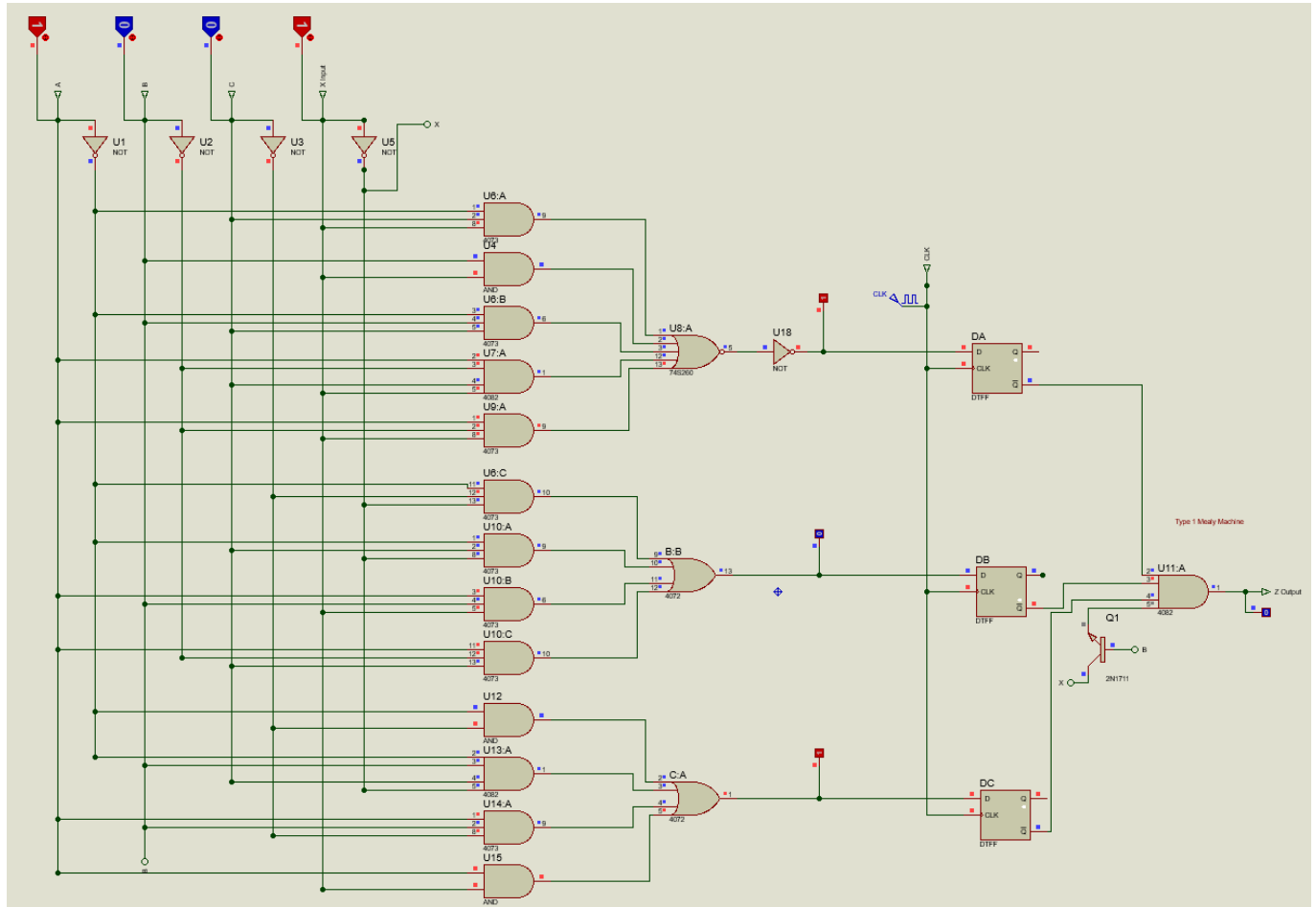


Figure 35 - Simulation, State E (100) Transition to State F (101)

	Present State			Input	Next State			Output
	A	B	C		A	B	C	
H	1	1	1	0	0	0	0	1

Table 8 - State Table, State H (111) Transition to State A (000)

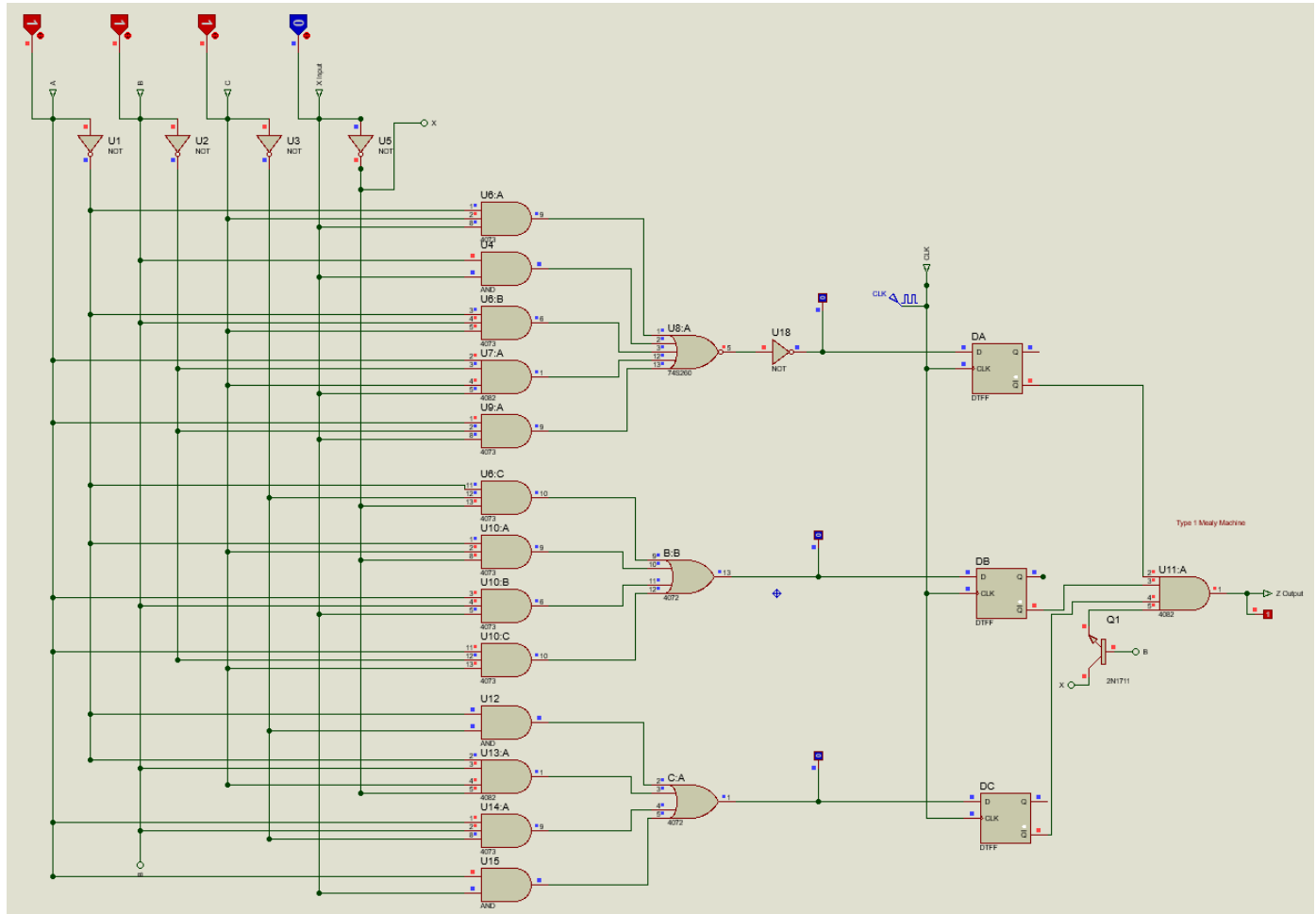


Figure 36 - Simulation, State H (111) Transition to State A (000)

4.1.4. Discussion

The finite state machine state diagram was a highly useful tool for visualizing the behavior of the system and for aiding with producing the circuit in Proteus. This was easy to follow to allow the state table and Karnaugh maps to be produced, simplify the logic circuit, and thus the logic circuit be designed and tested in Proteus. The Proteus circuit was designed exactly as the Karnaugh maps had stated, and the logic functioned perfectly, proving that design method works. The greatest feature of a logic circuit is its simplicity [12], and Karnaugh maps allow the greatest simplicity to be achieved very successfully.

4.1.5 Conclusion

To conclude, the circuit that was produced in Proteus perfectly replicated the required state transitions and showed the correct state identifications, as well as showing the correct inputs and output signals for each state transition. The state diagram and state tables were simple to follow, and allowed the logic circuit design to be made as simplistic as possible.

4.2 VHDL Code

4.2.1 Abstract

This laboratory report includes the whole process of utilizing a state diagram to produce a logical circuit in the simulation software called Xilinx ISE Design Suite. The state diagram shown in figure 36 below, shows the identification 3 bit number for each state with the identification letter. The arrows show the transitions and directions between states depending on the state of the input and output, shown as 1(input)/0(output) as an example. This diagram is followed to produce the VHDL code, which will tell the software how the system will behave depending on the different potential input and output states.

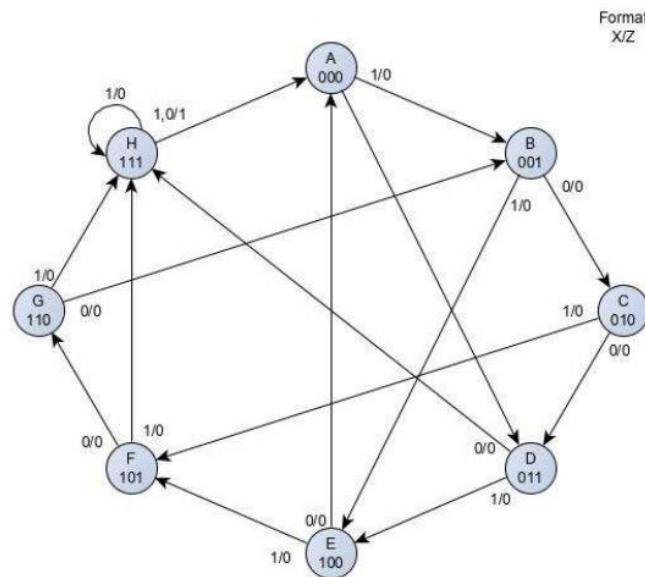


Figure 37 – Finite State Machine (FSM), State Diagram

4.2.2 Introduction

The purpose of this experiment was to use the state diagram to produce a circuit in a programming software called Xilinx ISE Design Suite. In this software, the circuit will be coded in VHDL computer code, meaning Very High-Speed Integrated Circuit Hardware Description Language [1], and then the software will generate the logic circuit and display the various states and the transitions between them.

Once the code and generated logic circuit is produced in Xilinx, this report will explain the entire design process and analyze the results to determine the effectiveness of the code. The simulation should replicate the state diagram shown in figure 31 exactly, which all transitions leading to the correct states, and showing the 3 bit identification numbers, as well as the input and output state for each transition.

4.2.3 Procedure

4.2.3.1 VHDL Module Coding

First, a VHDL Module was produced in ISE Project Navigator set to automotive spartan3. The first step was to input all external inputs and outputs to the circuit. The entity of the FSM was described with the input ports clk and x, and outputs a and z. The 'clk' will allow the machine to change state for each clock pulse. This is shown in figure 37.

```

32 entity eight_state_fsm is
33     Port ( clk : in  std_logic;
34           x  : in  std_logic;
35           a  : out std_logic_vector (2 downto 0);
36           z  : out  std_logic);
37 end eight_state_fsm;

```

Figure 38 - VHDL FSM Entity Statements

The next step was to input the 'architecture' by writing all of the states in the FSM, being the eight states from state_1 to state_8. Then the present state was set to state 1 as a signal, as this is the first state in the state diagram. Next the transitions or arrows were written into the code as signals, being 'next_z' and 'next_a'. This section is shown in figure 38 below.

```

39 architecture Behavioral of eight_state_fsm is
40
41     type state_type is (state_1, state_2, state_3, state_4, state_5, state_6, state_7, state_8);
42
43     signal present_state : state_type := state_1;
44     signal next_state   : state_type := state_1;
45     signal next_z       : std_logic ;
46     signal next_a       : std_logic_vector (2 downto 0) := "000";
47

```

Figure 39 - VHDL FSM Architecture

Next, the whole process is described in the code by first stating the 'process' for the present state as it changes around the FSM, depending on the state of the input x. This was coded by stating the current state and then where the machine will transfer to when x is 0. This is shown in figure 39. The 3 bit identification of the next state is shown at the end of each line. The state of the output z is also shown during each transition. All other states will be 'null' as the FSM is not concerned with anything other than the current and next state. This process is repeated for when the input x is 1, showing all transitions around the FSM when the input is in this state.

```

48 begin
49     state : process (present_state, x)
50     begin
51         if x='0' then
52             case present_state is
53                 when state_1 => next_state <=state_4; next_z <= '0'; next_a <="011";
54                 when state_2 => next_state <=state_3; next_z <= '0'; next_a <="010";
55                 when state_3 => next_state <=state_4; next_z <= '0'; next_a <="011";
56                 when state_4 => next_state <=state_8; next_z <= '0'; next_a <="111";
57                 when state_5 => next_state <=state_1; next_z <= '0'; next_a <="000";
58                 when state_6 => next_state <=state_7; next_z <= '0'; next_a <="110";
59                 when state_7 => next_state <=state_2; next_z <= '0'; next_a <="001";
60                 when state_8 => next_state <=state_1; next_z <= '1'; next_a <="000";
61                 when others => null;
62             end case;
63         else if x='1' then
64             case present_state is
65                 when state_1 => next_state <=state_2; next_z <= '0'; next_a <="001";
66                 when state_2 => next_state <=state_5; next_z <= '0'; next_a <="100";
67                 when state_3 => next_state <=state_6; next_z <= '0'; next_a <="101";
68                 when state_4 => next_state <=state_5; next_z <= '0'; next_a <="100";
69                 when state_5 => next_state <=state_6; next_z <= '0'; next_a <="101";
70                 when state_6 => next_state <=state_8; next_z <= '0'; next_a <="111";
71                 when state_7 => next_state <=state_8; next_z <= '0'; next_a <="111";
72                 when state_8 => next_state <=state_8; next_z <= '0'; next_a <="111";
73                 when others => null;
74             end case;
75         end if; -- end else if second if
76     end if; -- end 1st if
77 end process state;

```

Figure 40 - VHDL FSM Process Transition Statements

Next the clk was coded into the program so that it changes the present state to the next state, with z changing also. This can be seen in figure 40. This code was then synthesized and the syntax was checked by the software, in which the code was successful and the software can understand the code.

```

79     clocked: process (clk)
80     begin
81         if((clk='1') and (clk'last_value='0')) then
82             present_state <= next_state;
83             z <= next_z;
84             a <= next_a;
85         end if;
86
87
88     end process clocked;

```

Figure 41 - VHDL FSM Clock Description

4.2.3.2 VHDL Test Bench Coding

Next step of the VHDL FSM was to produce the test bench file so that the finite state machine can be simulated. Once again the components or 'ports' of the FSM were inputted into the test bench file, the same as previously, as shown in figure 37. The inputs and outputs were inputted as signals, with the clock and x being the inputs, and a and z being the outputs as shown in the state diagram. This is shown in figure 41.

```

52  --Inputs
53  signal clk : std_logic := '0';
54  signal x : std_logic := '0';
55
56  --Outputs
57  signal a : std_logic_vector(2 downto 0);
58  signal z : std_logic;
```

Figure 42 - VHDL FSM Test Bench Inputs & Outputs

Next the system was port mapped in the test bench. This means all external inputs and outputs were declared for the software to understand what the previous declared input and output signals are set for. These including clk, x, a, and z. This is shown in figure 42.

```

63  -- Instantiate the Unit Under Test (UUT)
64  uut: eight_state_fsm PORT MAP (
65      clk => clk,
66      x => x,
67      a => a,
68      z => z
69  );
```

Figure 43 - VHDL FSM Port Mapping

To state to the software the order in which the states are expected to cycle through during the simulation is then inputted in the stimulus process section. For each clock pulse, the input (x) will change state, which will in turn change the present state as previously stated in the process statements shown in figure 39. This was written so that the software will cycle around every state in the finite state machine until it reaches state 8, and then switches the output (z) on to test the function of the code and FSM. The 20ns (nano second) wait between each is to add a delay to make the simulation easier to understand with slower transition speeds. To make the code more user friendly, a simple explanation of each step in the process is explained with the green statements to the right side of each statement. This process can be seen in figure 43.

```

80      -- Stimulus process
81      stim_procl: process
82      begin
83          -- hold reset state for 20 ns.
84
85          x <= '1';          -- move from a to b
86          wait for 20 ns;
87          x <= '0';          -- move to c
88          wait for 20 ns;
89          x <= '0';          -- move to d
90          wait for 20 ns;
91          x <= '1';          -- move to e
92          wait for 20 ns;
93          x <= '1';          -- move to f
94          wait for 20 ns;
95          x <= '0';          -- move to g
96          wait for 20 ns;
97          x <= '1';          -- move to h
98          wait for 20 ns;
99          x <= '0';          -- move to A and set Z
100         wait for 20 ns;
101
102     end process;

```

Figure 44 - VHDL FSM Test Bench Simulation Process Statements

4.2.3.3 VHDL Simulation/Results

To test the produced VHDL code, the whole code was checked by selecting 'behavioral check syntax', for the software to check that it can read and understand the code for it to simulate it correctly. Once this was successful, the behavioral model of the finite state machine was produced by the software. For each clock pulse, the machine transitioned between each state exactly as requested in the stimulus process code shown in figure 43 above. In figure 44, the transition between state A/000 to state B/001 is shown, with the input (x) being 1, which is correct.

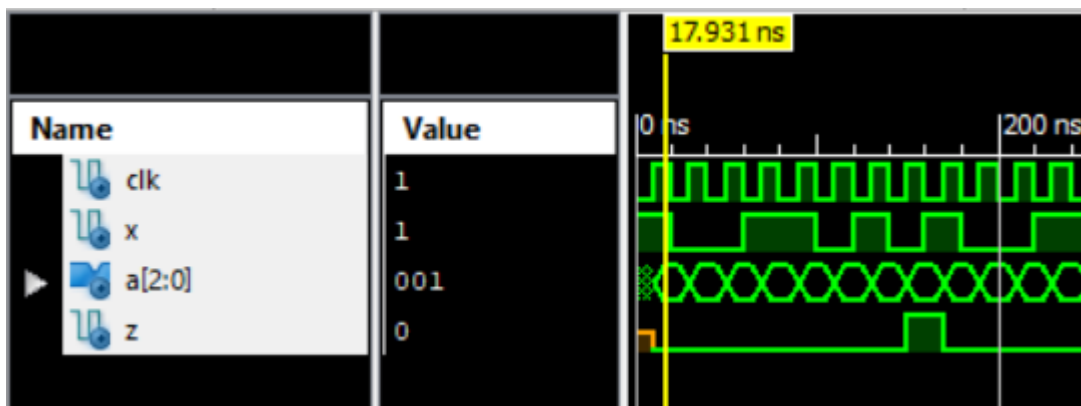


Figure 45 - FSM VHDL Simulation, Transition A/000 to B/001

Next figure 45 shows the transition between state D/011 and state E/100 with the input (x) being 1. The output (z) remains as 0, which is correct.



Figure 46 - FSM VHDL, Simulation, Transition D/011 to E/100

The transition between the final state H/111 to A/000 was then shown in the simulation. The transition input (x) is shown as 0 which is correct to the state diagram, and the output (z) is 1, which is also correct. The finite state machine has now completed a full cycle and the output switches on to 1 which proves that the code replicates the state diagram as required. This is shown in figure 46.

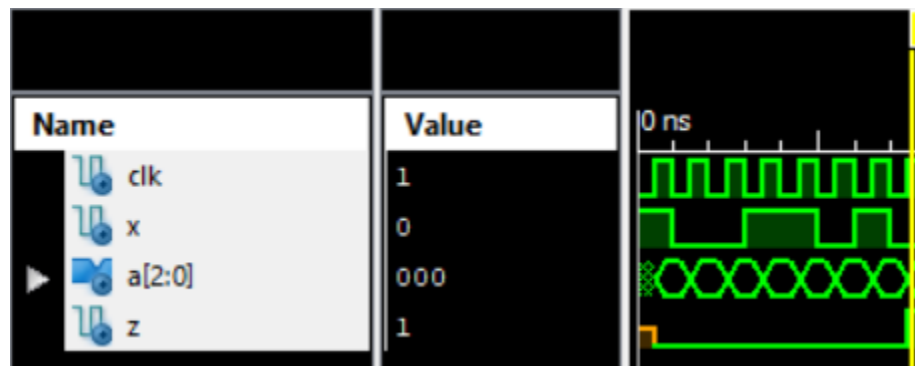


Figure 47 - FSM VHDL, Simulation, Transition H/111 to A/000

4.2.4 Discussion

The state diagram was very useful for visualizing the behavior of the finite state machine in a user friendly manner. This allowed the VHDL code to be easily written so that the software can reproduce the machine and simulate it within the software. The stimulus process at the bottom of the test bench can be easily modified to test the finite state machine in the other potential transitions. The code that was written was made easy to understand and follow, with some descriptive text in the code to make it more user friendly for the code writer. The simulation perfectly replicated the finite state machine's state diagram so this experiment was highly successful, and the code was written correctly.

4.2.5 Conclusion

From the finite state machine state diagram, the transitions between each state as well as the input and output states for each was noted. This was then written into VHDL code using software called Xilinx ISE Design Suite to virtually simulate the finite state machine. This was completed and the FSM was tested in the software, to which it functioned exactly as shown in the state diagram. This proves that finite state machines can be replicated and operated within software such as Xilinx.

5. References

- [1] G. Kaur, "Evolution of HDLs," in *VHDL Basic to Programming*, New Delhi 110017, India, Pearson, 2011, p. 6.
- [2] D. H. Sarah L. Harris, "Digital Building Blocks," in *Digital Design and Computer Architecture RISC-V Edition*, Morgan Kaufmann Publishers, 2021.
- [3] M. I. Abdellatif Bellaouar, "Multiplier's Comparison," in *Low-Power Digital Design, Circuits and Systems*, University of Waterloo, Springer Science+Business Media, LLC, 1995, p. 450.
- [4] P. P. Jan Pukite, "State Diagram, 6.7.1 State Definition," in *Modelling for Reliability Analysis*, The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, IEEE Press, 1998, p. 59.
- [5] F. E. American Heritage® Dictionary of the English Language, "Logic Circuit," Houghton Mifflin Harcourt Publishing, 2016. [Online]. Available: <https://www.thefreedictionary.com/logic+circuit#:~:text=logic%20circuit-,n,two%20or%20more%20input%20signals..> [Accessed 20 05 2022].
- [6] 6. E. Columbia Electronic Encyclopedia, "Logic Circuit," Columbia University Press, 2012. [Online]. Available: <https://www.infoplease.com/encyclopedia/science/engineering/computer/logic-circuit>. [Accessed 20 05 2022].
- [7] M. A. T. Robert B. Reese, "Synchronous Sequential Circuit Design," in *Introduction to Logic Synthesis Using Verilog HDL*, Southern Methodist University, United States of America, Morgan & Claypool Publishers, 2006, p. 59.
- [8] J. Wang, "1.2 Deterministic Finite Automata," in *Handbook of Finite State Based Models and Applications*, 6000 Broken Sound Parkway NW, Suite 300, Taylor & Francis Group, LLC, 2013, p. 3.
- [9] J. Seiffertt, "Table 17.3 State table for the state diagram of fig.17.6," in *Digital Logic For Computing*, Truman University Kirksville, Missouri, USA, Springer, 2017, p. 254.
- [10] S. P. Bali, "2000 Solved Problems in Digital Electronics," in *2000 Solved Problems in Digital Electronics*, Secundrabad, India, Tata McGraw-Hill Publishing Company Limited, 2005, p. 108.
- [11] S. P. Bali, "Product of Sums Method," in *2000 Solved Problems in Digital Electronics*, Secundrabad, India, Tata McGraw-Hill Publishing Company Limited, 2006, p. 114.
- [12] M. R. Bera, "Unit 6, Simplification using Karnaugh maps," in *Introductory Digital Systems for Engineering*, South Africa, Juta & CO, Ltd, 1999, p. 100.
- [13] D. Harres, "Running Assembly Language Programs," in *MSP430-Based Robot Applications*, Newnes, 2013.

- [14] I. Grout, "Introduction to Digital Logic Design," in *Digital Systems Design with FPGAs and CPLDs*, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, Newnes, Elsevier, 2008.
- [15] pradeepchakraborty, "Xilinx rolls out ISE Design Suite 11 for targeted design platforms!," 27 April 2009. [Online]. Available: <https://pradeepchakraborty.wordpress.com/2009/04/27/xilinx-rolls-out-ise-design-suite-11-for-targeted-design-platforms/>. [Accessed 20 05 2022].
- [16] M.-C. YANG, "Lecture 10: VHDL Versus Verilog," The Chinese University of Hong Kong, [Online]. Available: <http://www.cse.cuhk.edu.hk/~mcyang/ceng3430/2019S/Lec10%20VHDL%20versus%20Verilog.pdf>. [Accessed 20 05 2022].
- [17] 2. A. Madhu Gudhimetla, "Results," *COMPARISON OF DIFFERENT TYPES OF MULTIPLIERS WITH RESPECT TO SPEED, AREA AND POWER*, vol. 5, no. 12, p. 87, 2017.

