ELEC 5566M

FPGA Design System-on-Chip

Report on

2-Bit by 3-Bit Multiplier in Gate-Level Verilog HDL

Dagogo Godwin Orifama

201177661

MSc Electronics and Electrical Engineering

University of Leeds

February 2018

Abstract

This report outlines a comprehensive overview of the design of a digital logic circuit using the Quartus application software with C programming codes, it also focusses on the generation of test bench algorithm and a graphical representation of the code using the real time viewer. The generated test bench code was used to test the behaviour of the designed digital circuit using ModelSim – Altera Software. The ModelSim software generates a truth table showing the output of the designed circuit for various input combination. This was used as a validator to check for the correctness of the designed circuit following predetermined configurations. The multiplier circuit was implemented using a full adder circuit, after which the design was validated and tested in ModelSim through the generated truth table and output signal waveforms

Introduction

With the introduction of the binary system by Gottfried Leibniz in 1679, there has been an increase in the use of binary code. Binary code is a symbol system which is used for data representation and a for implementing computer instructions. The binary system is made up of just two digits (0 and 1) which can be manipulated to depict other meaning just b interchanging the position of the 0s or 1s present in the binary code. For example, a 5-bit binary code can represent up to 2^5 (32) possible values or computer instruction.

Today most computer systems use the binary number system for data representation and for issuing instructions. The evaluation of these data and instructions requires lots of logical and mathematical operations to be made with the computer processor. This brings the need to develop logical hardware circuit which can be used to execute such tasks with maximum efficiency. Earlier Microprocessors before the 1970's lacks a multiply instruction, which made it difficult for programmers to make use of the multiply routines. With the advance in technology, there was an increased possibility of combining up adder circuit together with other circuit on a single chip, which can be used to sum up all the partial product available in multiplier circuitry.

This project presents a comprehensive and systematic design and implementation of a hardware multiplier circuit which can multiply a 2-bit by 3-bit binary number. The report also outlines how Verilog Hardware Description Language can be used in modelling hardware designs. Quartus II 15.0 IDE was used as the Integrated Development Environment for the system modelling while verification and validation of the system design was done in Altera ModelSim 10.3d.

The design process was done using hierarchical design concept which is broken down from the top-level module down to other submodules. The process starts with the creation of a full adder which is then integrated into two submodules, these submodules are then cascaded into two rows and columns to form the 2-bit by 3-bit multiplier circuit design.

Design Methodology and Verification

Figure 1 shown the approach used in realising a functional 2-bits by 3-bits multiplier circuit, the following are the main components;

Full Adder

The multiplier circuit was design using full adder circuit, the Verilog HDL code for the full adder is in appendix . The full adder was design using primitive gate-level design techniques and it consists of two AND, and Exclusive OR gates. A 3-bit wire is used in carrying signal from the input to the output of the full adder circuit. Figure 2 (appendix) shows the RTL viewer output for the full adder circuit. A validation of figure 2 showing the expected output of the full adder is shown in table 1, and the mathematical simplification of both output of the RTL viewer and the source code (in appendix) is done subsequently.

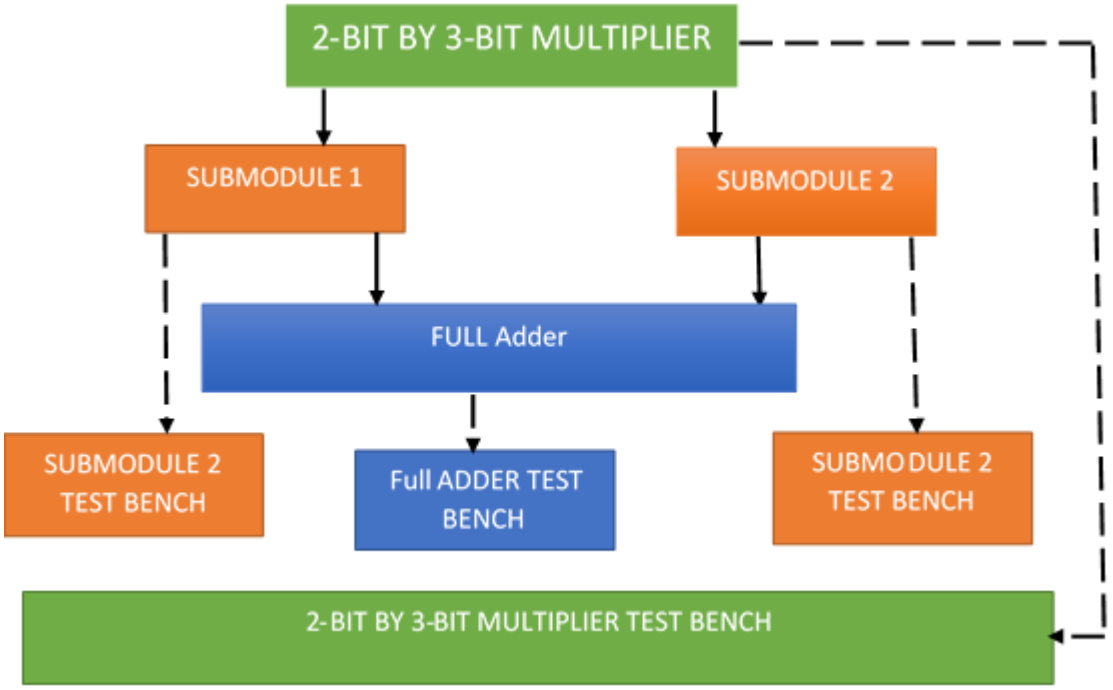

Figure 1: Design conceptualisation of 2-bits by 3-bits multiplier circuit

Number	Inputs (test_input)			Outputs	
	a	b	Cin	Cout	sum
1	0	0	0	0	0
2	0	0	1	0	1
3	0	1	0	0	1
4	0	1	1	1	0
5	1	0	0	0	1
6	1	0	1	1	0
7	1	1	0	1	0
8	1	1	1	1	1

Table 1: Truth table of full adder showing expected result

$$link[1] = ab$$

$$link[0] = a + b$$

$$link[2] = (a\bar{b} + \bar{a}b).Cin$$

$$Cout = link[1] + link[2] = ab + (a\bar{b} + \bar{a}b).Cin$$

$$sum = (Cin.\overline{link[0]}) + (\overline{Cin}.link[0]) = ((a\bar{b} + \bar{a}b).\overline{Cin}) + (\overline{(a\bar{b} + \bar{a}b)}.Cin)$$

The full adder design was validated in ModelSim as seen in figure 2 and figure 4 in the appendix.

Submodules

The multiplier design consists of two submodules, these submodules instantiates the full adder circuit design. The Verilog HDL code for the submodules is shown in the appendix.

Submodule 1: consists of 5 inputs signals, 2 output signals, two AND gates and two 1-bit wires. The wires are used to carry output signals from the AND gates to the full adder. Test bench signals was generated and used to validate submodule 1 Verilog design in ModelSim. The generated truth table and output waveform is shown in figure 4 and figure 5 respectively.

Submodule 2 consists of 4 inputs, 2 outputs, an AND gate and a wire which connects the full adder from the AND gate

2-Bit by 3-Bit Multiplier Design and Verification

The multiplier design integrates the full adder, submodule 1 and submodule 2 modules. The RTL viewer output of the multiplier is shown in figure 10 (in appendix). Verification of the multiplier was done in ModelSIm and th generated truth table and output waveform of the output is shown in figure 12 and figure 11 respectively in the appendix.

Conclusion

This report has actively investigated and presented the design of a 2-bit by 3-bit multiplier circuit which integrated the full adder circuit design, it is designed using Verilog HDL. It has been shown that there is a relationship between the number of input a circuit design has and the possible combination available for verifying the output. Similarly, the test bench code is a vital tool which is required in simulating the behaviour of the designed circuit by generating the input signal required for the verification of the design in ModelSim.

Appendix

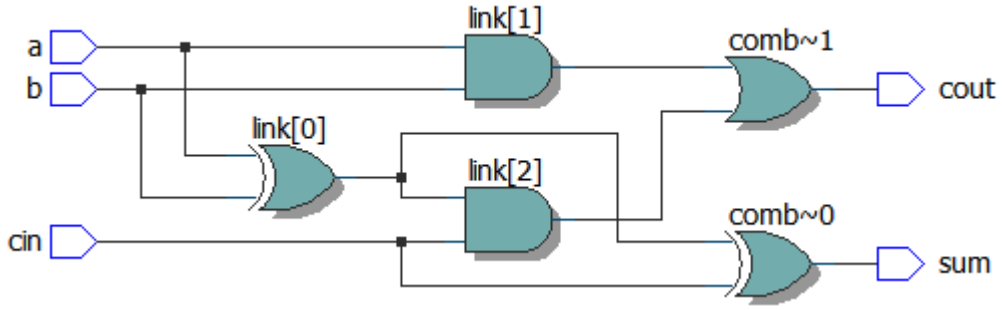

Figure 2: RTL View of the Full Adder

```
# ========== Simulation Starts Here ==
#
# run -all
# test_input Sum cout
# 000    sum=0    cout=0
# 001    sum=1    cout=0
# 010    sum=1    cout=0
# 011    sum=0    cout=1
# 100    sum=1    cout=0
# 101    sum=0    cout=1
# 110    sum=0    cout=1
# 111    sum=1    cout=1
```

Figure 3: Verification of the Full adder by ModelSim

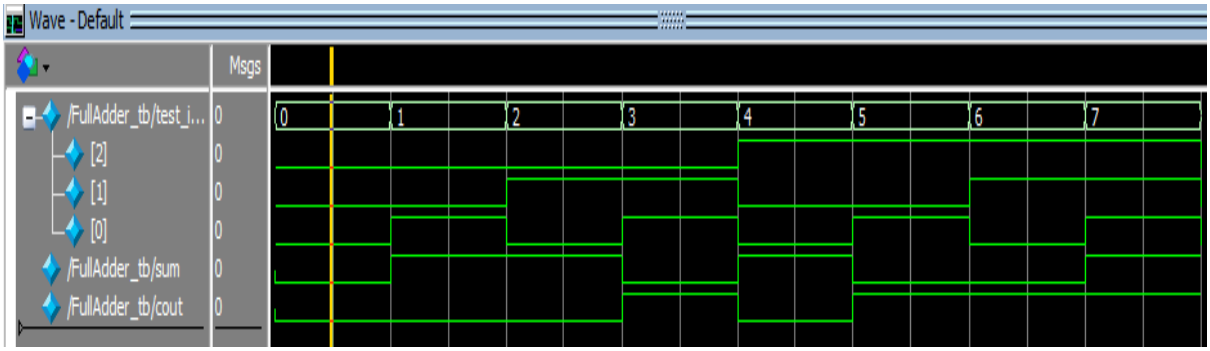

Figure 4: Verification of the full adder module showing output waveforms

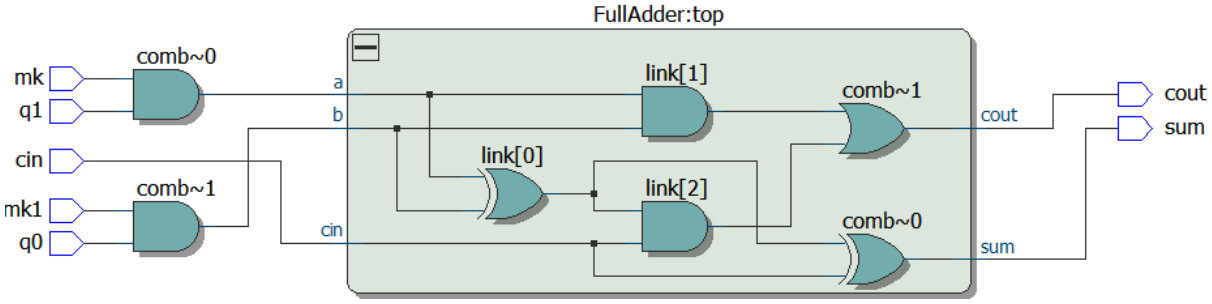

Figure 6: RTL Viewer output of SubModule 1

```
  0 ns simulation started
  0 ns mk = 0 mk1 = 0 q0 = 0 q1 = 0 cin = 0 sum = 0 cout = 0
 10 ns mk = 1 mk1 = 0 q0 = 0 q1 = 0 cin = 0 sum = 0 cout = 0
 20 ns mk = 0 mk1 = 1 q0 = 0 q1 = 0 cin = 0 sum = 0 cout = 0
 30 ns mk = 1 mk1 = 1 q0 = 0 q1 = 0 cin = 0 sum = 0 cout = 0
 40 ns mk = 0 mk1 = 0 q0 = 1 q1 = 0 cin = 0 sum = 0 cout = 0
 50 ns mk = 1 mk1 = 0 q0 = 1 q1 = 0 cin = 0 sum = 0 cout = 0
 60 ns mk = 0 mk1 = 1 q0 = 1 q1 = 0 cin = 0 sum = 1 cout = 0
 70 ns mk = 1 mk1 = 1 q0 = 1 q1 = 0 cin = 0 sum = 1 cout = 0
 80 ns mk = 0 mk1 = 0 q0 = 0 q1 = 1 cin = 0 sum = 0 cout = 0
 90 ns mk = 1 mk1 = 0 q0 = 0 q1 = 1 cin = 0 sum = 1 cout = 0
100 ns mk = 0 mk1 = 1 q0 = 0 q1 = 1 cin = 0 sum = 0 cout = 0
110 ns mk = 1 mk1 = 1 q0 = 0 q1 = 1 cin = 0 sum = 1 cout = 0
120 ns mk = 0 mk1 = 0 q0 = 1 q1 = 1 cin = 0 sum = 0 cout = 0
130 ns mk = 1 mk1 = 0 q0 = 1 q1 = 1 cin = 0 sum = 1 cout = 0
140 ns mk = 0 mk1 = 1 q0 = 1 q1 = 1 cin = 0 sum = 1 cout = 0
150 ns mk = 1 mk1 = 1 q0 = 1 q1 = 1 cin = 0 sum = 0 cout = 1
160 ns mk = 0 mk1 = 0 q0 = 0 q1 = 0 cin = 1 sum = 1 cout = 0
170 ns mk = 1 mk1 = 0 q0 = 0 q1 = 0 cin = 1 sum = 1 cout = 0
180 ns mk = 0 mk1 = 1 q0 = 0 q1 = 0 cin = 1 sum = 1 cout = 0
190 ns mk = 1 mk1 = 1 q0 = 0 q1 = 0 cin = 1 sum = 1 cout = 0
200 ns mk = 0 mk1 = 0 q0 = 1 q1 = 0 cin = 1 sum = 1 cout = 0
210 ns mk = 1 mk1 = 0 q0 = 1 q1 = 0 cin = 1 sum = 1 cout = 0
220 ns mk = 0 mk1 = 1 q0 = 1 q1 = 0 cin = 1 sum = 0 cout = 1
230 ns mk = 1 mk1 = 1 q0 = 1 q1 = 0 cin = 1 sum = 0 cout = 1
240 ns mk = 0 mk1 = 0 q0 = 0 q1 = 1 cin = 1 sum = 1 cout = 0
250 ns mk = 1 mk1 = 0 q0 = 0 q1 = 1 cin = 1 sum = 0 cout = 1
260 ns mk = 0 mk1 = 1 q0 = 0 q1 = 1 cin = 1 sum = 1 cout = 0
270 ns mk = 1 mk1 = 1 q0 = 0 q1 = 1 cin = 1 sum = 0 cout = 1
280 ns mk = 0 mk1 = 0 q0 = 1 q1 = 1 cin = 1 sum = 1 cout = 0
290 ns mk = 1 mk1 = 0 q0 = 1 q1 = 1 cin = 1 sum = 0 cout = 1
300 ns mk = 0 mk1 = 1 q0 = 1 q1 = 1 cin = 1 sum = 0 cout = 1
310 ns mk = 1 mk1 = 1 q0 = 1 q1 = 1 cin = 1 sum = 1 cout = 1
```

Figure 7: Truth table verification of submodule 1 from ModelSim

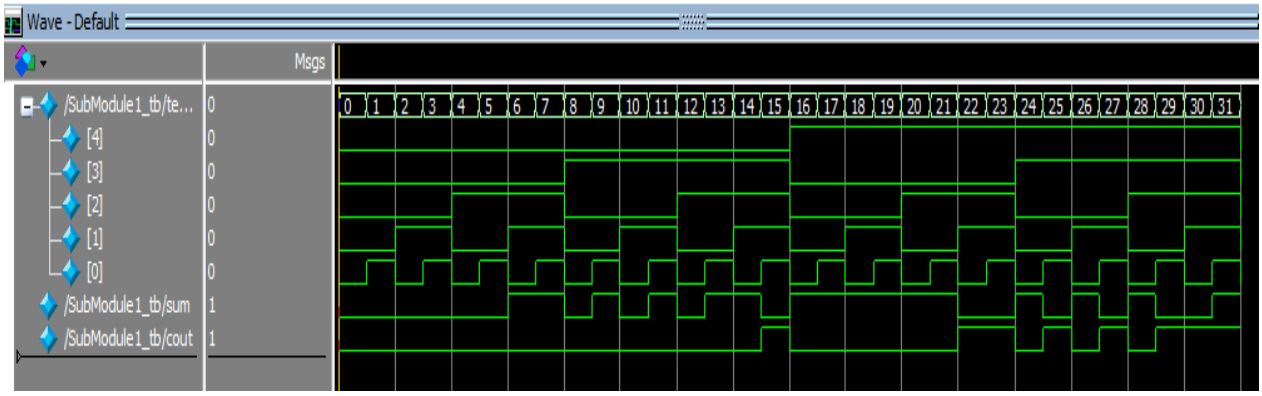

Figure 5: Verification of SubModule 1 in ModelSim

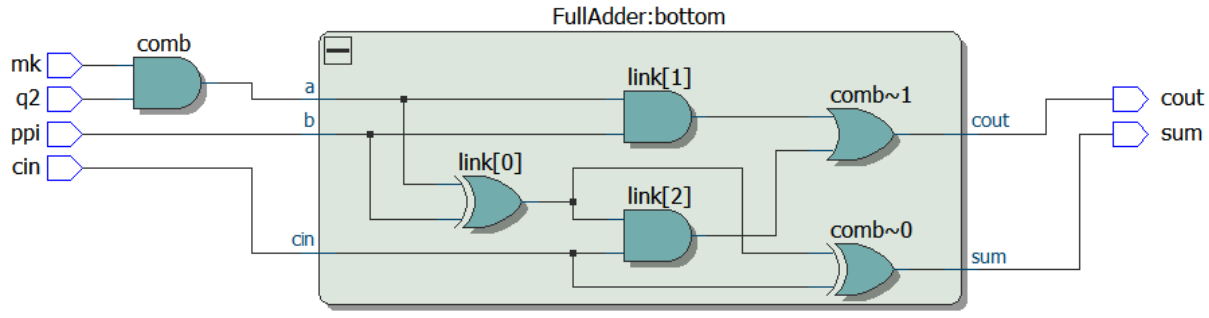

Figure 7: RTL viewer output of submodule 2

```
# ========== Simulation Starts Here ===
#
# run -all
# testInput sum cout
# 0000   sum=0    cout=0
# 0001   sum=0    cout=0
# 0010   sum=0    cout=0
# 0011   sum=1    cout=0
# 0100   sum=1    cout=0
# 0101   sum=1    cout=0
# 0110   sum=1    cout=0
# 0111   sum=0    cout=1
# 1000   sum=1    cout=0
# 1001   sum=1    cout=0
# 1010   sum=1    cout=0
# 1011   sum=0    cout=1
# 1100   sum=0    cout=1
# 1101   sum=0    cout=1
# 1110   sum=0    cout=1
# 1111   sum=1    cout=1
```

Figure 8: Truth table verification of submodule 2 from ModelSim

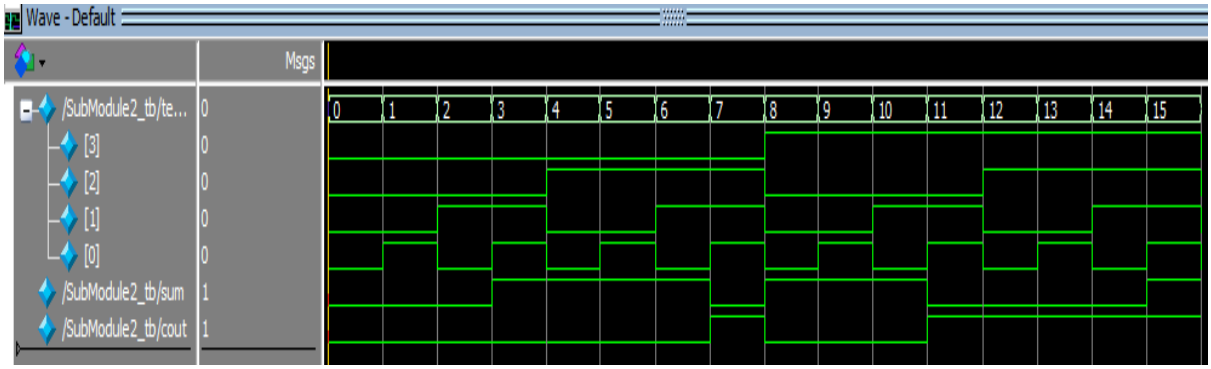

Figure 9: Verification of SubModule 2 in ModelSim

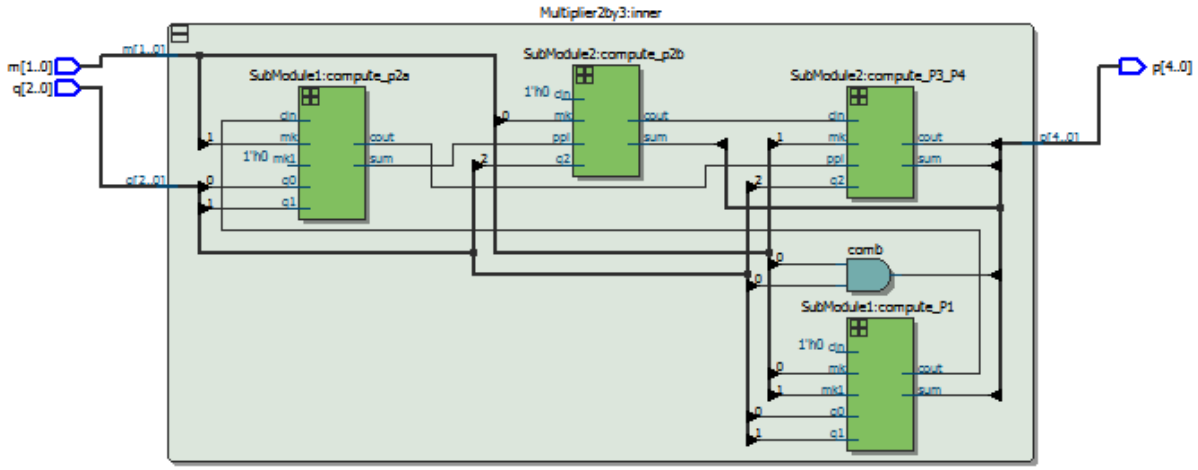

Figure 10: RTL viewer output of 2-bit by 3-bit multiplier design

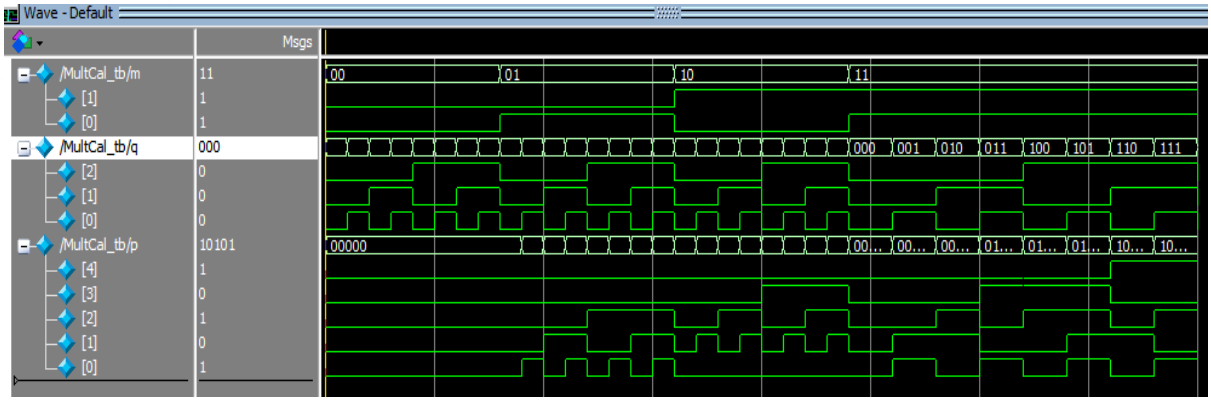

Figure 11: Verification of 2-bit by 3-bit Multiplier design in ModelSim

```
  0 ns Simulation Started
  0 ns m=00 q=000 p=00000
 10 ns m=00 q=001 p=00000
 20 ns m=00 q=010 p=00000
 30 ns m=00 q=011 p=00000
 40 ns m=00 q=100 p=00000
 50 ns m=00 q=101 p=00000
 60 ns m=00 q=110 p=00000
 70 ns m=00 q=111 p=00000
 80 ns m=01 q=000 p=00000
 90 ns m=01 q=001 p=00001
100 ns m=01 q=010 p=00010
110 ns m=01 q=011 p=00011
120 ns m=01 q=100 p=00100
130 ns m=01 q=101 p=00101
140 ns m=01 q=110 p=00110
150 ns m=01 q=111 p=00111
160 ns m=10 q=000 p=00000
170 ns m=10 q=001 p=00010
180 ns m=10 q=010 p=00100
190 ns m=10 q=011 p=00110
200 ns m=10 q=100 p=01000
210 ns m=10 q=101 p=01010
220 ns m=10 q=110 p=01100
230 ns m=10 q=111 p=01110
240 ns m=11 q=000 p=00000
260 ns m=11 q=001 p=00011
280 ns m=11 q=010 p=00110
300 ns m=11 q=011 p=01001
320 ns m=11 q=100 p=01100
340 ns m=11 q=101 p=01111
360 ns m=11 q=110 p=10010
380 ns m=11 q=111 p=10101
```

Figure 12: Verification of 2-bit by 3-bit Multiplier design using generated truth table

Source Codes

Complete working code for the module and and submodule

Full Adder Module code

```
1.  //full adder Module Code
2.  module FullAdder (
3.      // Declare input and output po
    rts
4.      input  a,
5.      input  b,
6.      input  cin,
7.      output cout,
8.      output sum
9.  );
10.
11. //wires used to connect the gates

12.     wire [2:0] link;
13.     // Instantiate gates to calcul
    ate sum output
14.     xor(link[0],a,b);
15.     xor(sum,link[0],cin);
16.     // Instantiate gates to calcul
    ate carry (cout) output
17.     and(link[1],a,b);
18.     and(link[2],cin,link[0]);
19.     or (cout,link[1],link[2]);
20. endmodule
```

Full Adder Test Bench Code

```
1.  //Test Bench of full adder
2.  `timescale 1 ns/100 ps
3.
4.  module FullAdder_tb;
5.
6.  //Test Bench generated signal
7.  // a - reg[0], b - reg[1], cin - r
    eg[2]
8.  reg [2:0] test_input;
9.
10. // DUT output signal
11. wire sum;
12. wire cout;
13.
14. //device under test
15. FullAdder FullAdder_dut(
16.     .a(test_input[0]         ),
17.     .b(test_input[1]         ),
18.     .cin(test_input[2]       ),
19.     .sum(sum                 ),
20.     .cout(cout               )
21. );
22.
23. //Test Bench genrated signal
24. initial begin
25.     //initialise the inputs
26.     test_input = 1'b0;
27.     // print to console that simul
    ation has started and output the c
    urrent time
28.     $display("test_input\tSum\tcou
    t");
29.     // monitor the input an output
     signals
30.     $monitor("%b\tsum=%b\tcout=%b"
    ,test_input,sum,cout);
31. end
32.
33. always begin
34.
35.      //10 nanosecond for each alte
    ration
36.     repeat(8) #10 test_input = tes
    t_input + 1'b1;
37.     $stop;
38.     $display("%d nd\tSimulation co
    mpleted", $time);
39. end
40. endmodule
```

SubModule 1 Module code

```
1.  ///SubModule 1 module code
2.  module SubModule1(
3.      input mk,
4.      input mk1,
5.      input q0,
6.      input q1,
7.      input cin,
8.      output sum,
9.      output cout
10. );
11.     wire linka, linkb;
12.     and(linka,q1,mk);
13.     and(linkb,q0,mk1);
14. // Attach full adder to the module
     component
15.     FullAdder top(
16.         .a(linka    ),
17.         .b(linkb    ),
18.         .cin(cin    ),
19.         .sum(sum    ),
20.         .cout(cout  )
21. );
22. endmodule
23.
24. module FullAdder (
25.     // Declare input and output po
    rts
26.     input  a,
27.     input  b,
28.     input  cin,
29.     output cout,
30.     output sum
31. );
32.
```

```
33. //wires used to connect the gates

34.     wire [2:0] link;
35.     // Instantiate gates to calcul
    ate sum output
36.     xor(link[0],a,b);
37.     xor(sum,link[0],cin);
38.     // Instantiate gates to calcul
    ate carry (cout) output
39.     and(link[1],a,b);
40.     and(link[2],cin,link[0]);
41.     or (cout,link[1],link[2]);
42. endmodule
```

SubModule 1 Test Bench

```
1.  //Test Bench for SubModule1
2.  `timescale 1 ns/100 ps
3.
4.  module SubModule1_tb;
5.
6.  /* Test Bench generated signals
7.      mk - testInput[0]
8.      mk1- testInput[1]
9.      q0- testInput[2]
10.     q1 - testInput[3]
11.     cin - testInput[4]
12. */
13.     reg [4:0] testInput;
14.
15. //DUT output signals
16.     wire sum;
17.     wire cout;
18.
19. // Device under test
20. SubModule1 SubModule_dut(
21.     .mk(testInput[0]    ),
22.     .mk1(testInput[1]   ),
23.     .q0(testInput[2]    ),
24.     .q1(testInput[3]    ),
25.     .cin(testInput[4]   ),
26.     .sum(sum            ),
27.     .cout(cout          )
28.
29. );
30.
31. //Test Bench generated signals
32. initial begin
33.     testInput = 0;
34. // Print to the console that simul
    ation has started and also thesimu
    lation time
35.     $display("%d ns\tsimulation st
    arted ", $time);
36. // monitor te input an output sign
    als
37.     $monitor("%d ns\ mk = %b\ mk1
    = %b\ q0 = %b\ q1 = %b\ cin = %b\
    sum = %b\ cout = %b",$time,testInp
    ut[0],testInput[1],testInput[2]
38.                     ,testInput[3],t
    estInput[4],sum,cout);
39.
40. end
41. always begin
42. // Testing for all 32 possible inp
    uts
43.     repeat(32) #10 testInput = tes
    tInput + 1'b1;
44.     $stop;
45.     $display("%d ns\tSimulation co
    mpleted",$time);
46. end
47. endmodule
```

SubModule 2 Module code

```
1.  // SubMoule 2 verilog code
2.  module SubModule2(
3.      input mk,
4.      input q2,
5.      input cin,
6.      input ppi,
7.      output sum,
8.      output cout
9.  );
10.     wire linka;
11.     and(linka,q2,mk);
12. // Attach full adder to module com
    ponents
13.     FullAdder bottom(
14.         .a(linka),
15.         .b(ppi),
16.         .cin(cin),
17.         .sum(sum),
18.         .cout(cout)
19.     );
20. endmodule
21.
22. module FullAdder (
23.     // Declare input and output po
    rts
24.     input  a,
25.     input  b,
26.     input  cin,
27.     output cout,
28.     output sum
29. );
30. //wires used to connect the gates

31.     wire [2:0] link;
32. // Instantiate gates to calculate
    sum output
33.     xor(link[0],a,b);
34.     xor(sum,link[0],cin);
35. // Instantiate gates to calculate
    carry (cout) output
36.     and(link[1],a,b);
37.     and(link[2],cin,link[0]);
38.     or (cout,link[1],link[2]);
39. endmodule
```

SubModule 2 Test Bench code

```
1.  //Test Bench for SubModule1
2.  `timescale 1 ns/100 ps
3.
4.  module SubModule2_tb;
5.  /* Test Bench generated signals
6.      mk - testInput[0]
7.      q2 - testInput[1]
8.      cin - testInput[2]
9.      ppi - testInput[3]
10. */
11.     reg [3:0] testInput;
12. //DUT output signals
13.     wire sum;
14.     wire cout;
15.
16. // Device under test
17. SubModule2 SubModule_dut(
18.     .mk(testInput[0]    ),
19.     .q2(testInput[1]    ),
20.     .cin(testInput[2]   ),
21.     .ppi(testInput[3]   ),
22.     .sum(sum            ),
23.     .cout(cout          )
24.
25. );
26.
27. //Test Bench generated signals
28. initial begin
29.
30.     testInput = 0;
31. // Print to the console that simul
    ation has started and also thesimu
    lation time
32.     $display("%d ns\tsimulation st
    arted ", $time);
33. // monitor te input an output sign
    als
34.     $monitor("%d ns\ mk = %b\ q2 =
     %b\ ppi = %b\ cin = %b\ sum=%b\ c
    out=%b",$time,
35.             testInput[0], testInpu
    t[1], testInput[2],testInput[3],su
    m,cout);
36.
37. end
38. always begin
39. // Testing for all 32 possible inp
    uts
40.     repeat(16) #10 testInput = tes
    tInput + 1'b1;
41.     $stop;
42.     $display("%d ns\tSimulation co
    mpleted",$time);
43. end
44. endmodule
```

2-bit by 3-bit Multiplier Module Verilog Code

```
1.  /*
2.   * 2-bit by 3-
    bit Multiplier in Gate-
    Level Verilog HDL
3.   * ----------------------------
4.   * By:   Dagogo Orifama
5.   * SID: 201177661
6.   * Date: 05/02/2017
7.   *
8.   * Description
9.   * ------------
10.  * The module is a 2-bit by 3-
    bit multiplier which has been
11.  * built in Verilog using Gate-
    Level 1-bit full adders.
12.  *
13.  */
14. module MultCal(
15. //Declearing input and output port
    s
16.     input [1:0] m,
17.     input[2:0] q,
18.     output[4:0] p
19. );
20. Multiplier2by3 inner(
21.     .m(m  ),
22.     .q(q  ),
23.     .p(p  )
24. );
25. endmodule
26.
27. /*
28.  * 2-bits by 3-bits multiplier
29.  * ------------------------------
    ---
30.  * The module is a 2-bit by 3-
    bit multiplier which has been
31.  * built in Verilog using Gate-
    Level 1-bit full adders.
32.  *
33.  */
34.
35. module Multiplier2by3(
36. //Declearing input and output port
    s
37.     input [1:0] m,  // 2-
    bit input port, made up of m1,m0
38.     input[2:0] q,   // 3-
    bit input port, made up of q2,q1,q
    0
39.     output[4:0] p   // 5-
    bits output port, made up of p4,p3
    ,p,p1,p0
40. );
41. and(p[0],q[0],m[0]);    // evaluat
    ing the value of p0
42.  wire [2:0] carry;      //carry si
    gnal between each module
43.  wire ppSum;
44.
45.  // computing the value of p1
46.  SubModule1 compute_P1(
47.     .mk(m[0]),
48.     .mk1(m[1]),
49.     .q0(q[0]),
50.     .q1(q[1]),
51.     .cin (0),
```

```
    .sum(p[1]),
    .cout(carry[0])
 );
 // working towards computing the value of p2
 SubModule1 compute_p2a(
    .mk(m[1]),
    .mk1(0),
    .q0(q[0]),
    .q1(q[1]),
    .cin (carry[0]),
    .sum(ppSum),
    .cout(carry[1])

 );
 // computing the value of p2
 SubModule2 compute_p2b(
    .mk(m[0]),
    .q2(q[2]),
    .cin(0),
    .ppi(ppSum),
    .sum(p[2]),
    .cout(carry[2])
);
// computing the values of p3 and p4
 SubModule2 compute_P3_P4(
    .mk(m[1]),
    .q2(q[2]),
    .cin(carry[2]),
    .ppi(carry[1]),
    .sum(p[3]),
    .cout(p[4])
);
 endmodule

module SubModule1(
    input mk,
    input mk1,
    input q0,
    input q1,
    input cin,
    output sum,
    output cout
);
    wire linka, linkb;
    and(linka,q1,mk);
    and(linkb,q0,mk1);

    FullAdder top(
                .a(linka),
                .b(linkb),
                .cin(cin),
                .sum(sum),
                .cout(cout)

            );

       endmodule

       module SubModule2(
           input mk,
           input q2,
           input cin,
           input ppi,
           output sum,
           output cout
       );
           wire linka;
           and(linka,q2,mk);

           FullAdder bottom(
               .a(linka),
               .b(ppi),
               .cin(cin),
               .sum(sum),
               .cout(cout)
           );
       endmodule

       module FullAdder (
           // Declare input and output ports
           input  a,
           input  b,
           input  cin,
           output cout,
           output sum
       );

       //wires used to connect the gates
           wire [2:0] link;

           // Instantiate gates to calculate sum output
           xor(link[0],a,b);
           xor(sum,link[0],cin);

           // Instantiate gates to calculate carry (cout) output
           and(link[1],a,b);
           and(link[2],cin,link[0]);
           or (cout,link[1],link[2]);
       endmodule
```

2-bit by 3-bit Multiplier Test Bench

```
/*
 * 2-bit by 3-bit Multiplier Test Bench
 * ----------------------------
 * By:   Dagogo Orifama
 * SID: 201177661
 * Date: 05/02/2017
 *
 * Description
 * ------------
 * This is a test bench module to test the 2-bit by 3-but multiplier.
 *
 */

/* Timescale indicates unit of delays.
```

```
15.   `timescale  unit / precision
16.    Where delays are given as:
17.    #unit.precision
18. */
19.
20. `timescale 1 ns/100 ps
21. // Test bench module declaration
22. //  There is no port list for a te
    st bench
23. module MultCal_tb;
24. //Test bench generated signal
25. reg [1:0] m;    // 2-bit signal
26. reg [2:0] q;    // 3-bit signal
27.
28. //DUT Output Signal
29. wire [4:0] p;   // 5-bit signal
30.
31. // device Under test
32. MultCal MultCal_dut (
33.     .m(m        ),
34.     .q(q        ),
35.     .p(p        )
36. );
37.
38. // Test bench logic
39. initial begin
40.    //Print to console that the sim
    ulation has started. $time is the
    current simulation time.
41.     $display("%d ns\tSimulation St
    arted",$time);
42.     //Monitor any changes to any v
    alues listed, and print to the con
    sole when they change.
43.     //There can only be one $monit
    or per simulation.
44.     $monitor("%d ns\tm=%b\tq=%b\tp
    =%b",$time,m,q,p);
45.
66.
```

```
46. // Generate the testbench signals
47. q = 2'b0;  // initialise q=0
48. //increase the value of m from 0 t
    o 3
49. for(m = 2'b0; m < 2'b11; m = m + 2
    'b1)
50.
51. case(m)
52. // generate 7 more values of of q
    based on current value of m
53.     2'd00: repeat(8) #10 q = q + 2
    'b01; //when m =0, increase q from
     0 to 7
54.     2'b01: repeat(8) #10 q = q + 2
    'b01; //when m =1, increase q from
     0 to 7
55.     2'b10: repeat(8) #10 q = q + 2
    'b01; //when m =2, increase q from
     0 to 7
56. endcase
57.     if(m==2'b11)
58.     repeat(8) #10 q = q + 2'b01;
      //when m =3, increase q from 0 t
    o 7
59.     $stop;
60.  $display("%d ns\tSimulation Finis
    hed",$time);
61.     //There are no other processes
     running in this testbench, so "ru
    n -all" in ModelSim
62.     //will finish the simulation a
    utomatically now.
63.     //We can also use $stop(); to
    finish the simulation whenever we
    want.
64. end
65. endmodule
```