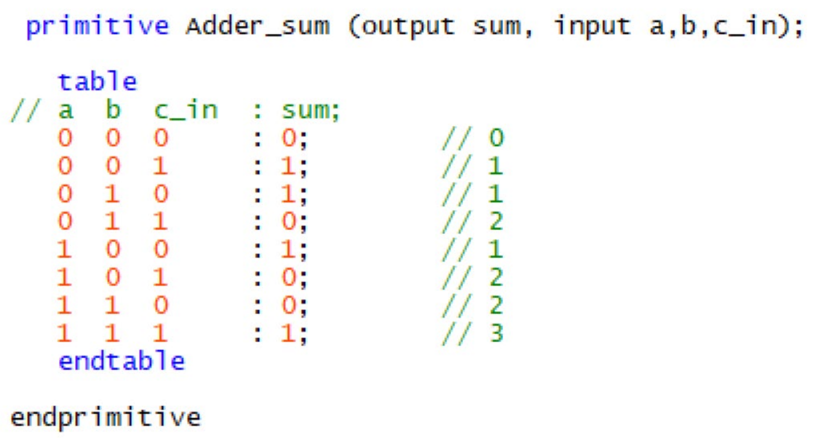
Lesson 2: Structural Design

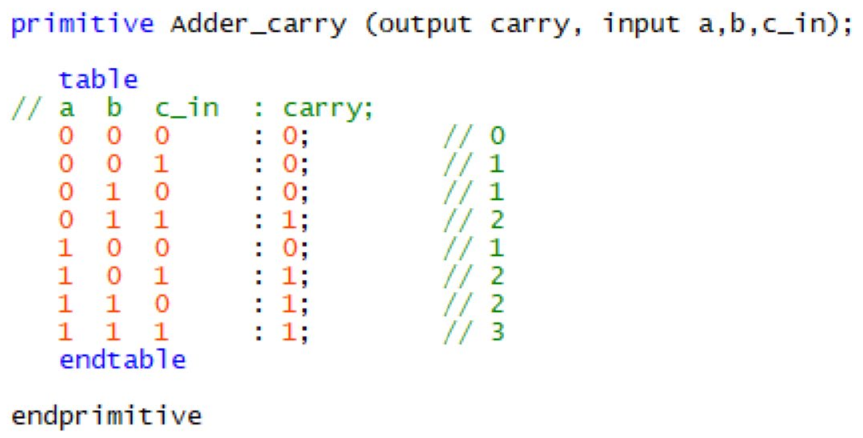
Assignment 1:
Design a Two-8bit-input adder using primitive full adders and structural hierarchical design.

(a) Design a primitive Adder_sum with 3 inputs for a, b, and c_in that finds the LSB of the sum of the three inputs.



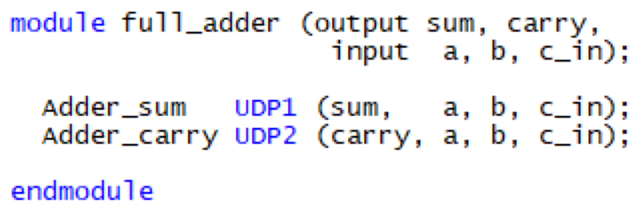
```
 primitive Adder_sum (output sum, input a,b,c_in);

   table
// a  b  c_in  : sum;
   0  0  0     : 0;       // 0
   0  0  1     : 1;       // 1
   0  1  0     : 1;       // 1
   0  1  1     : 0;       // 2
   1  0  0     : 1;       // 1
   1  0  1     : 0;       // 2
   1  1  0     : 0;       // 2
   1  1  1     : 1;       // 3
   endtable

endprimitive
```

(b) Design a primitive Adder_carry with 3 inputs for a, b, and c_in that finds the MSB or carry_out of the sum of the three inputs.



```
 primitive Adder_carry (output carry, input a,b,c_in);

   table
// a  b  c_in  : carry;
   0  0  0     : 0;       // 0
   0  0  1     : 0;       // 1
   0  1  0     : 0;       // 1
   0  1  1     : 1;       // 2
   1  0  0     : 0;       // 1
   1  0  1     : 1;       // 2
   1  1  0     : 1;       // 2
   1  1  1     : 1;       // 3
   endtable

 endprimitive
```

(c) Design a full_adder top module that instantiates the two primitives created in parts (a) and (b) as shown in the given netlist viewer.



```
module full_adder (output sum, carry,
                   input  a, b, c_in);

  Adder_sum   UDP1 (sum,   a, b, c_in);
  Adder_carry UDP2 (carry, a, b, c_in);

endmodule
```

(d) Create a higher level two-4bit_input adder by instantiating the single-bit full adder 4 times as follows as indicated in the book and the PowerPoint document.

```
module four_bit_adder (sum_4, c_out, a_4, b_4, c_input);

     output [3:0] sum_4;
     output       c_out;
     input  [3:0] a_4;
     input  [3:0] b_4;
     input        c_input;

     wire c_0, c_1, c_2;

     full_adder A0  (sum_4[0], c_0,   a_4[0], b_4[0], c_input);
     full_adder A1  (sum_4[1], c_1,   a_4[1], b_4[1], c_0);
     full_adder A2  (sum_4[2], c_2,   a_4[2], b_4[2], c_1);
     full_adder A3  (sum_4[3], c_out, a_4[3], b_4[3], c_2);

  endmodule
```

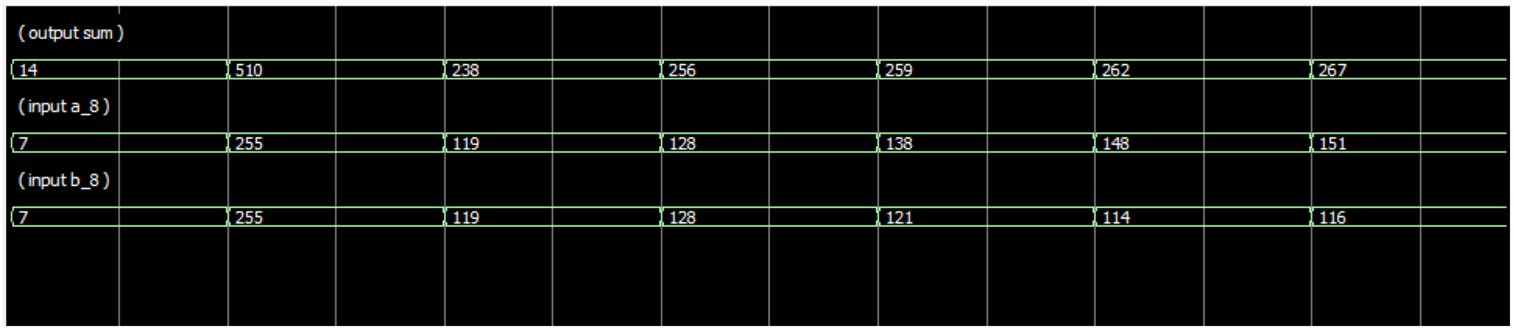
(e) Design a top module Two-8bit-input (sum_out, a_in, b_in) that adds two 8-bit inputs. Make sure, that the size of the output sum can accommodate the maximum numbers that can be reached by the inputs a_in and b_in without any overflow.

```
module Eight_bit_adder (sum_9, a_8, b_8);

     output [8:0] sum_9;
     input  [7:0] a_8;
     input  [7:0] b_8;

     wire c_LSB;

     four_bit_adder M0  (sum_9[3:0], c_LSB,    a_8[3:0], b_8[3:0], 1'b0);
     four_bit_adder M1  (sum_9[7:4], sum_9[8], a_8[7:4], b_8[7:4], c_LSB);

endmodule
```

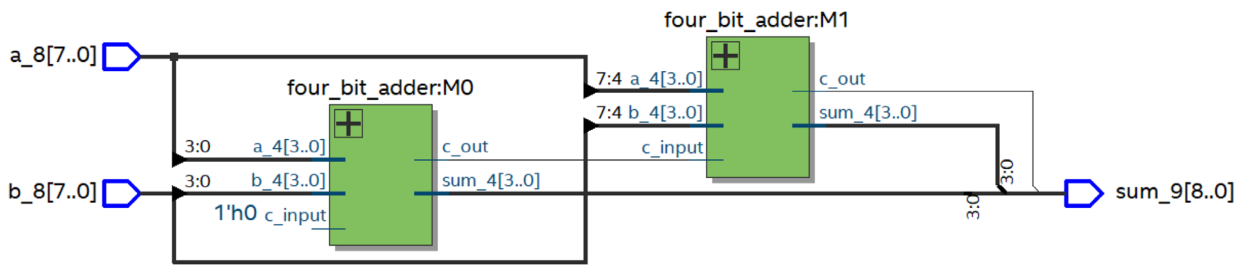
(f) Create a testbench that tests the functionality of the single building blocks of the design as well as the interconnections.

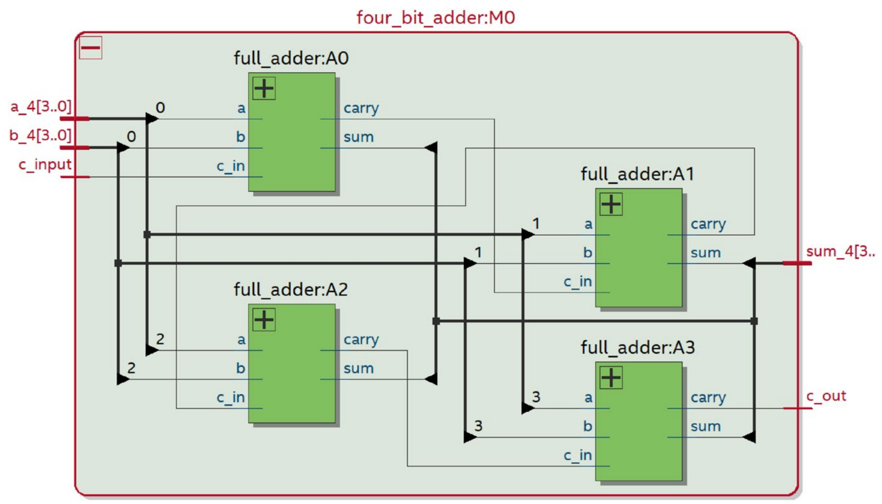
```
  module Eight_bit_adder_tb ();

     wire [8:0] sum_9;
     reg  [7:0] a_8;
     reg  [7:0] b_8;

     Eight_bit_adder UUT (sum_9, a_8, b_8);

     initial
     begin
           a_8 = 8'b0000_0111;         b_8 = 8'b0000_0111;
     #10   a_8 = 8'b1111_1111;         b_8 = 8'b1111_1111;
     #10   a_8 = 8'b0111_0111;         b_8 = 8'b0111_0111;
     #10   a_8 = 8'b1000_0000;         b_8 = 8'b1000_0000;
     repeat (2)  begin
     #10   a_8 = a_8 + 10;             b_8 = b_8 - 7; end
     repeat (2)  begin
     #10   a_8 = a_8 + 3;              b_8 = b_8 + 2; end
     end

     endmodule
```

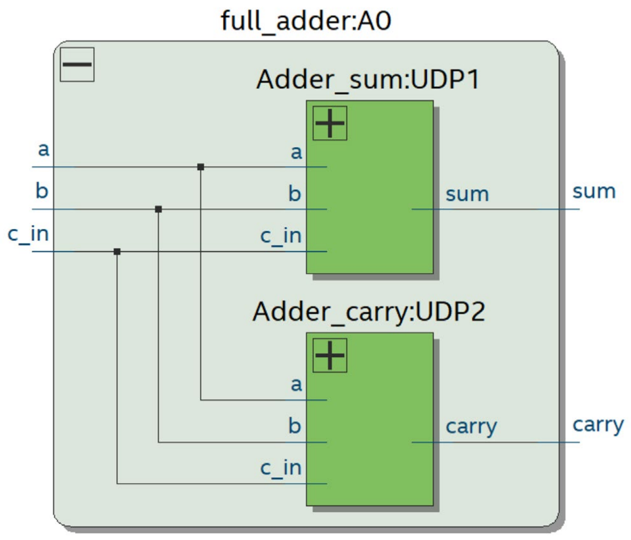
Simulation Results:


```
( output sum )
14        510       238       256       259       262       267
( input a_8 )
7         255       119       128       138       148       151
( input b_8 )
7         255       119       128       121       114       116
```

The sum is verified and correct for each pair of input values tested.

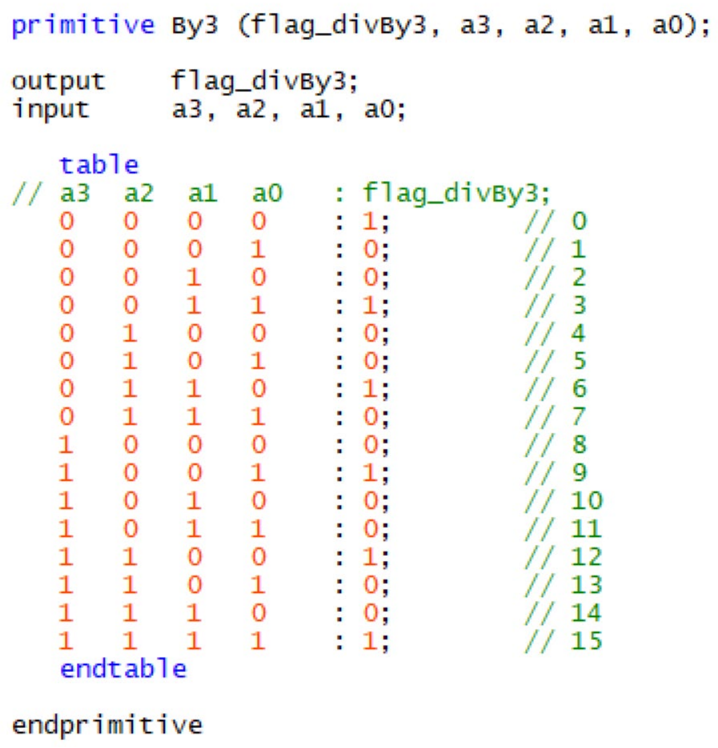




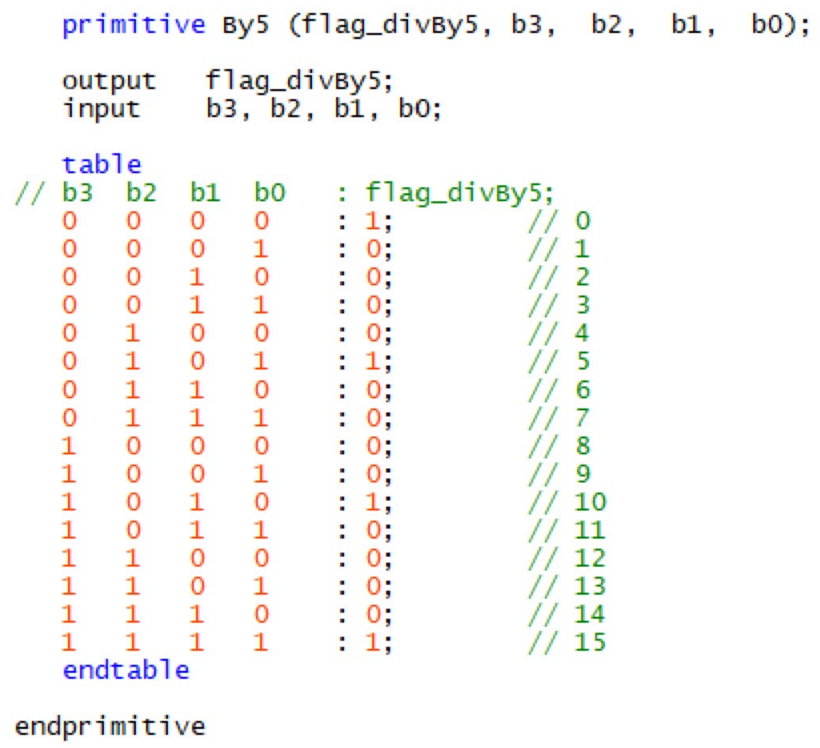


Assignment 2:

Design a module using the structural design of simple logic gates that takes as an input a 4bit word and has two single-bit outputs divBy3 and divBy5:
divBy3 is raised to a logic high if the 4-bit input is divisible by 3 including 0.
divBy5 is raised to a logic high if the 4-bit input is divisible by 5 including 0.



```
primitive By3 (flag_divBy3, a3, a2, a1, a0);

output    flag_divBy3;
input     a3, a2, a1, a0;

   table
// a3  a2  a1  a0   : flag_divBy3;
   0   0   0   0    : 1;        // 0
   0   0   0   1    : 0;        // 1
   0   0   1   0    : 0;        // 2
   0   0   1   1    : 1;        // 3
   0   1   0   0    : 0;        // 4
   0   1   0   1    : 0;        // 5
   0   1   1   0    : 1;        // 6
   0   1   1   1    : 0;        // 7
   1   0   0   0    : 0;        // 8
   1   0   0   1    : 1;        // 9
   1   0   1   0    : 0;        // 10
   1   0   1   1    : 0;        // 11
   1   1   0   0    : 1;        // 12
   1   1   0   1    : 0;        // 13
   1   1   1   0    : 0;        // 14
   1   1   1   1    : 1;        // 15
   endtable

endprimitive
```



```
   primitive By5 (flag_divBy5, b3,  b2,  b1,  b0);

   output   flag_divBy5;
   input    b3, b2, b1, b0;

   table
// b3  b2  b1  b0   : flag_divBy5;
   0   0   0   0    : 1;        // 0
   0   0   0   1    : 0;        // 1
   0   0   1   0    : 0;        // 2
   0   0   1   1    : 0;        // 3
   0   1   0   0    : 0;        // 4
   0   1   0   1    : 1;        // 5
   0   1   1   0    : 0;        // 6
   0   1   1   1    : 0;        // 7
   1   0   0   0    : 0;        // 8
   1   0   0   1    : 0;        // 9
   1   0   1   0    : 1;        // 10
   1   0   1   1    : 0;        // 11
   1   1   0   0    : 0;        // 12
   1   1   0   1    : 0;        // 13
   1   1   1   0    : 0;        // 14
   1   1   1   1    : 1;        // 15
   endtable

endprimitive
```

- Use Karnaugh map to derive the switching function for the two output bits. Use simple logic gates and internal wires to build the structural design for the module.

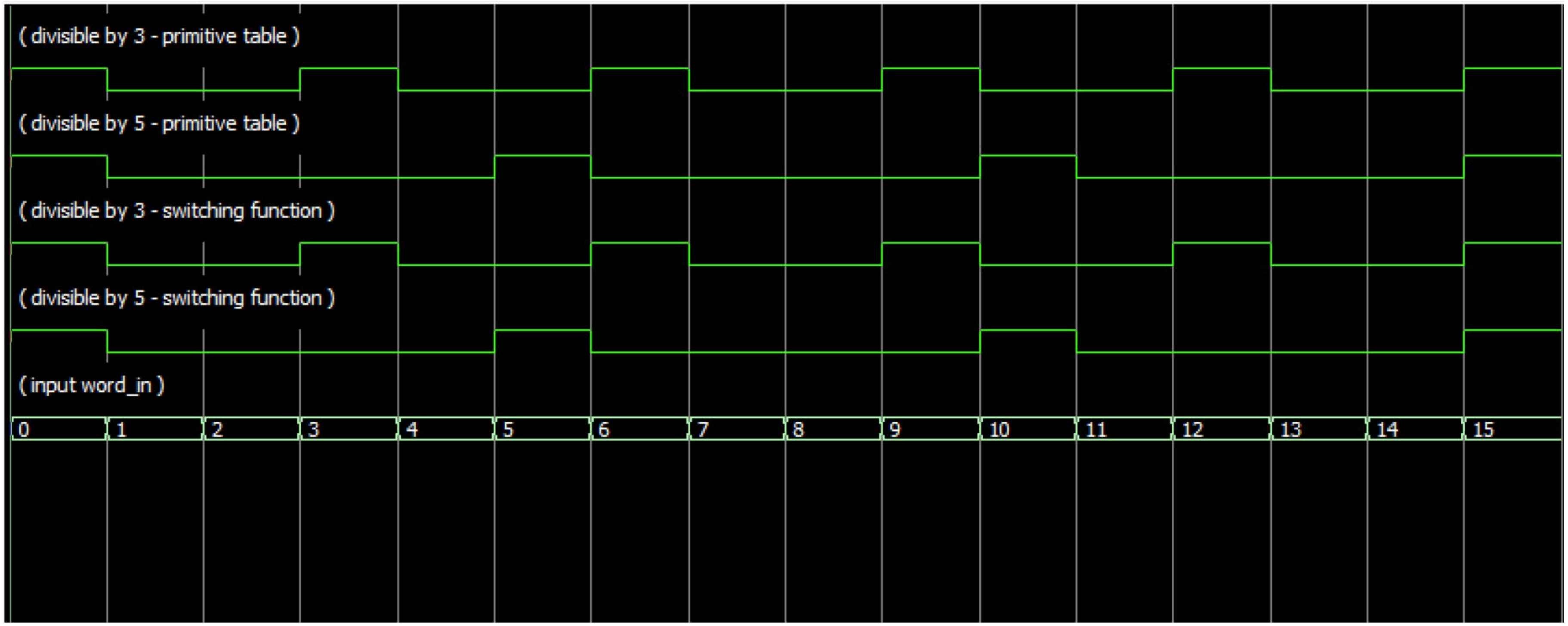
b3b2 b1b0	00	01	11	10
00	1	0	1	0
01	0	0	0	1
11	1	0	1	0
10	0	1	0	0

```
module DIVBY3   (divBy3, a3,  a2,  a1,  a0);

output    divBy3;
input     a3, a2, a1, a0;

wire out_0, out_3, out_6, out_9, out_12, out_15;

assign out_0  = ~a3 & ~a2 & ~a1 & ~a0 ;
assign out_3  = ~a3 & ~a2 &  a1 &  a0 ;
assign out_6  = ~a3 &  a2 &  a1 & ~a0 ;
assign out_9  =  a3 & ~a2 & ~a1 &  a0 ;
assign out_12 =  a3 &  a2 & ~a1 & ~a0 ;
assign out_15 =  a3 &  a2 &  a1 &  a0 ;

assign divBy3 = out_0 | out_3 | out_6 | out_9 | out_12 | out_15;

endmodule
```

b3b2 b1b0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

```
module DIVBY5   (output divBy5, input a3,  a2,  a1,  a0);

wire out_0, out_5, out_10, out_15;

assign out_0  = ~a3 & ~a2 & ~a1 & ~a0 ;
assign out_5  = ~a3 &  a2 & ~a1 &  a0 ;
assign out_10 =  a3 & ~a2 &  a1 & ~a0 ;
assign out_15 =  a3 &  a2 &  a1 &  a0 ;

assign divBy5 = out_0 | out_5 | out_10 | out_15;

endmodule
```

```
module Divisible_tb ();

wire                table_divBy3, table_divBy5;
wire                Kmap_divBy3,  Kmap_divBy5;
reg     [3:0]       word_in;

Divisible UUT       (table_divBy3, table_divBy5, Kmap_divBy3,  Kmap_divBy5, word_in);

initial
begin
      word_in    = 0;
forever
 #10  word_in    = word_in + 1;
 end

 initial
 #160   $stop;

 endmodule
```


(divisible by 3 - primitive table)
(divisible by 5 - primitive table)
(divisible by 3 - switching function)
(divisible by 5 - switching function)
(input word_in)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

The output of the primitives and the modules with simple logic gates are consistent with the correct math.