USN | | | | | | | | | |

Internal Assessment Test 3 – Dec 2022

Sub:	Verilog HDL				Sub Code:	18EC56	Branch:	ECE	
Date:	27/1/22	Duration:	90 min’s	Max Marks:	50	Sem/Sec:	5th/A,B,C,D		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	Use dataflow description style of Verilog HDL to design 4-bit adder using Carry look ahead logic.	[10]	C03	L2
2	Describe the following statements with an example: initial and always	[10]	CO4	L3
3	What are blocking and non-blocking assignment statements? Explain with examples.	[10]	CO4	L2
4	With syntax explain conditional, branching and loop statements available in Verilog HDL behavioral description.	[10]	CO4	L2
5	Define a function to multiply two 4-bit numbers a and b. The output is an 8-bitvalue.		CO4	L3
6	Create a design that uses the 4-bit full adder. Use a conditional compilation (`ifdef). Compile the fulladd4 with defparam statements if the text macro DPARAM is defined by the `define statement; otherwise, compile the fulladd4 with module instance parameter values.	[10]	C05	L1
7	What is logic synthesis? Explain the basic computer-aided logic synthesis using flow chart also List the problems addressed by automated logic synthesis.	[10]	C06	L2

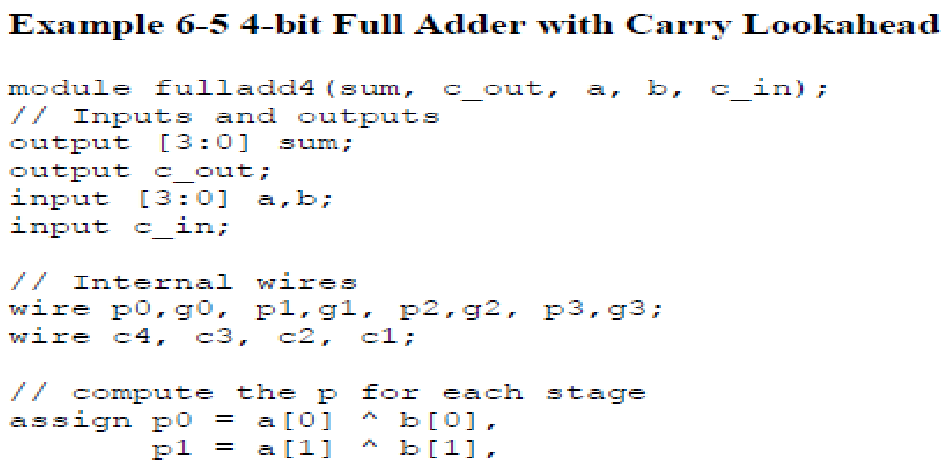
USN | | | | | | | | | |

Internal Assessment Test 3 – Dec 2022

Sub:	Verilog HDL				Sub Code:	18EC56	Branch:	ECE	
Date:	27/1/22	Duration:	90 min’s	Max Marks:	50	Sem/Sec:	5th/A,B,C,D		OBE

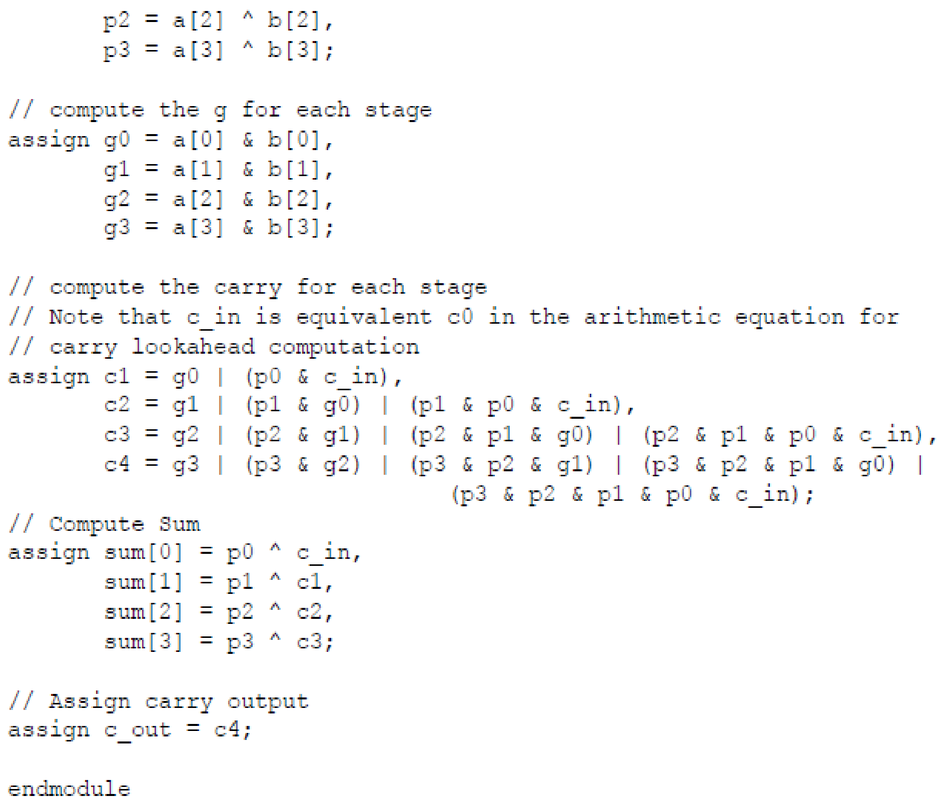
	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	Use dataflow description style of Verilog HDL to design 4-bit adder using Carry look ahead logic.	[10]	C03	L2
2	Describe the following statements with an example: initial and always	[10]	CO4	L3
3	What are blocking and non-blocking assignment statements? Explain with examples.	[10]	CO4	L2
4	With syntax explain conditional, branching and loop statements available in Verilog HDL behavioral description.	[10]	CO4	L2
5	Define a function to multiply two 4-bit numbers a and b. The output is an 8-bitvalue.		CO4	L3
6	Create a design that uses the 4-bit full adder. Use a conditional compilation (`ifdef). Compile the fulladd4 with defparam statements if the text macro DPARAM is defined by the `define statement; otherwise, compile the fulladd4 with module instance parameter values.	[10]	C05	L1
7	What is logic synthesis? Explain the basic computer-aided logic synthesis using flow chart also List the problems addressed by automated logic synthesis.	[10]	C06	L2

1.Use dataflow description style of Verilog HDL to design 4-bit adder using Carry look ahead logic.



Example 6-5 4-bit Full Adder with Carry Lookahead

```
module fulladd4(sum, c_out, a, b, c_in);
// Inputs and outputs
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;

// Internal wires
wire p0,g0, p1,g1, p2,g2, p3,g3;
wire c4, c3, c2, c1;

// compute the p for each stage
assign p0 = a[0] ^ b[0],
       p1 = a[1] ^ b[1],
```



```
       p2 = a[2] ^ b[2],
       p3 = a[3] ^ b[3];

// compute the g for each stage
assign g0 = a[0] & b[0],
       g1 = a[1] & b[1],
       g2 = a[2] & b[2],
       g3 = a[3] & b[3];

// compute the carry for each stage
// Note that c_in is equivalent c0 in the arithmetic equation for
// carry lookahead computation
assign c1 = g0 | (p0 & c_in),
       c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
       c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
                           (p3 & p2 & p1 & p0 & c_in);
// Compute Sum
assign sum[0] = p0 ^ c_in,
       sum[1] = p1 ^ c1,
       sum[2] = p2 ^ c2,
       sum[3] = p3 ^ c3;

// Assign carry output
assign c_out = c4;

endmodule
```

2. Describe the following statements with an example: initial and always

- Two basic structured procedure statements

always
initial

  - All behavioral statements can appear only inside these blocks
  - Each always or initial block has a separate activity flow (concurrency)

Start from **simulation time 0**

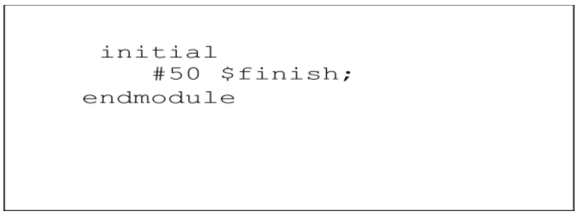
Structured Procedures:
initial statement

- Starts at time 0
- Executes only once during a simulation
- Multiple initial blocks, execute in parallel
  - All start at time 0
  - Each finishes independently
- Syntax:

```
initial
begin
            // behavioral statements
end
```

- Example:

```
module stimulus;
  reg x, y, a, b, m;
  initial
    m= 1'b0;
  initial
  begin
    #5 a=1'b1;
    #25 b=1'b0;
  end
 initial
  begin
    #10 x=1'b0;
    #25 y=1'b1;
  end
```



```
  initial
     #50 $finish;
endmodule
```

always statement

- Start at time 0
- Execute the statements in a looping fashion
- Example

```
module clock_gen(output reg clock);
  // Initialize clock at time zero
  initial
    clock = 1'b0;
  // Toggle clock every half-cycle (time period =20)
  always
    #10 clock = ~clock;
  initial
    #1000 $finish;
endmodule
```

3.What are blocking and non-blocking assignment statements? Explain with examples.

Procedural Assignment

- It updates the value of reg, integer , real or time variables.
- Types of Procedural Assignment :
- Blocking Statement
- Non blocking Statement

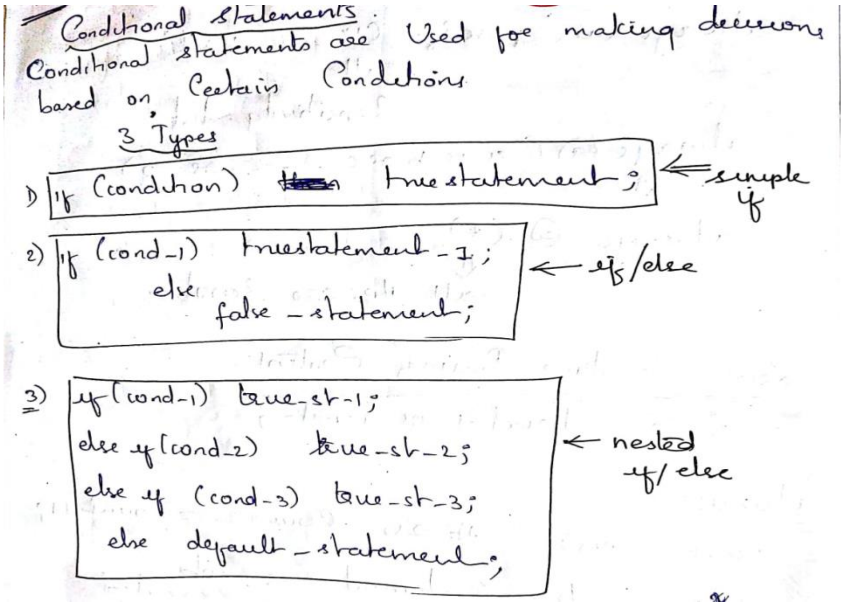
- The two types of procedural assignments
    - Blocking assignments
    - Non-blocking assignments
- Blocking assignments
    - are executed in order (sequentially)
    - They use = operator
    - Example:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
  x=0; y=1; z=1;
  count=0;
reg_a= 16'b0; reg_b = reg_a;
  #15 reg_a[2] = 1'b1;
  #10 reg_b[15:13] = {x, y, z};
  count = count + 1;
end
```

- Non-blocking assignments
    - All statements are executed parallely , except the statements with delays specified.
    - They use <= *operator*
    - Example:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
  x=0; y=1; z=1;
  count=0;
reg_a= 16’b0; reg_b = reg_a;
 #15 reg_a[2] <= 1’b1;
 #10 reg_b[15:13] <= {x, y, z};
  count <= count + 1;
end
```

With syntax explain conditional, branching and loop statements available in Verilog HDL behavioral description



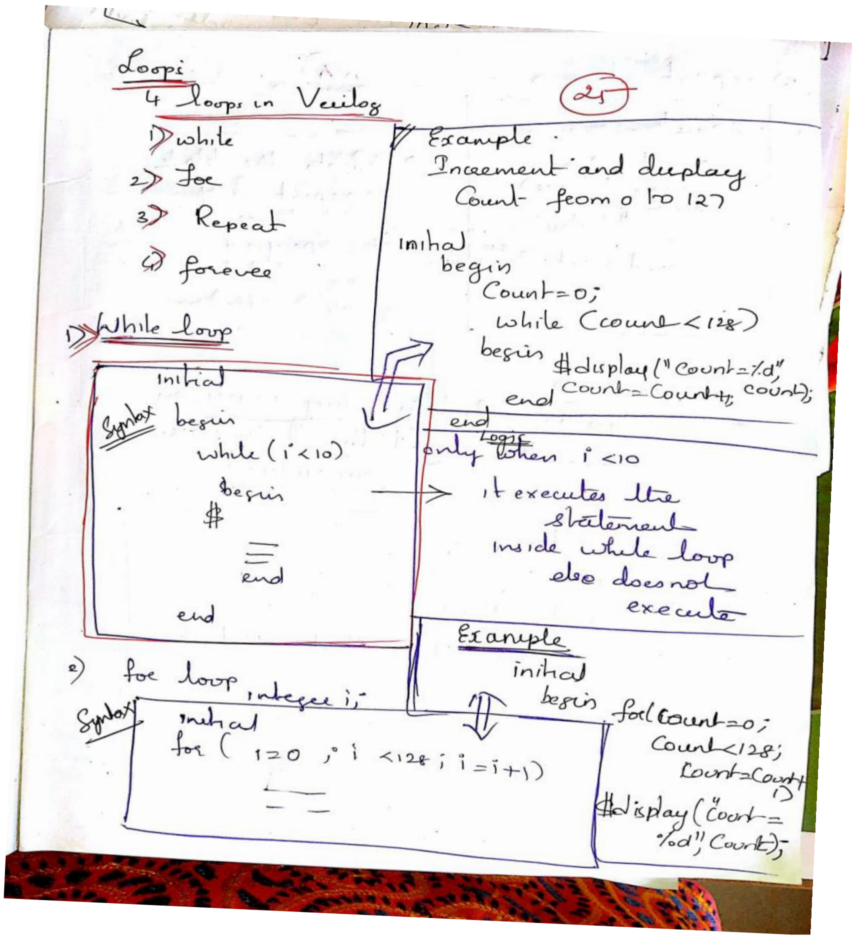
Conditional statements:
Conditional statements are used for making decisions based on certain conditions

3 Types

1) `if (condition) ~~then~~ true statement;` ⟸ simple if

2) ⟵ if/else
```
if (cond_1)  truestatement_1;
    else
        false_statement;
```

3) ⟵ nested if/else
```
if(cond_1)  true_st_1;
else if(cond_2)  true_st_2;
else if (cond_3)  true_st_3;
  else  default_statement;
```



Loops:
4 loops in Verilog
1) while
2) for
3) Repeat
4) forever

1) While loop

Syntax
```
initial
begin
   while (i<10)
     begin
      $
        ≡
      end
end
```
Logic
only when i <10 → it executes the statement inside while loop else does not execute

Example
Increment and display Count from 0 to 127
```
initial
   begin
     Count=0;
      while (count<128)
     begin  $display ("Count=%d", count);
            Count=Count+1;
      end
   end
```

2) for loop

Syntax
```
integer i;
initial
for (  i=0  ; i <128 ; i=i+1)
        ——
        — —
```

Example
```
initial
  begin  for(Count=0;
              Count<128;
               Count=Count+1)
$display ("Count=%d", Count);
```

3) repeat

```
initial
  count = 0;
   repeat( 128)
begin
  $display("Count = %0d", count);
end Count = Count +1;
```

⟶ Repeats the block repeat no of times Specified

4) forever loop

// does not Contains any expression

// Executes Until the $finish is encountered

```
reg clock
initial
    Clock = 1'b0;
     forever  #10 Clock = ~clock;
```

Example ALU

Opcode	
00	a+b
01	a-b
10	a*b
11	a&b

I

case (op) → Treats 0,1,x,z as i/ps for bit by bit

```
2'b00: y=a+b;
2'b01: y=a-b;
2'b10: y=a*b;
2'b11: y=a&b;
endcase.
```

(1) opcode

	y
00	a+b
01	a-b
10	a*b
11	a&b

(2) 0x, x0, 1x, x1 } o/p z if not predefined else as assigned

(3) 0z, z0, 1z, z1 } o/p z if not predefined else as assigned

II

II

z1 } predefined else as assigned

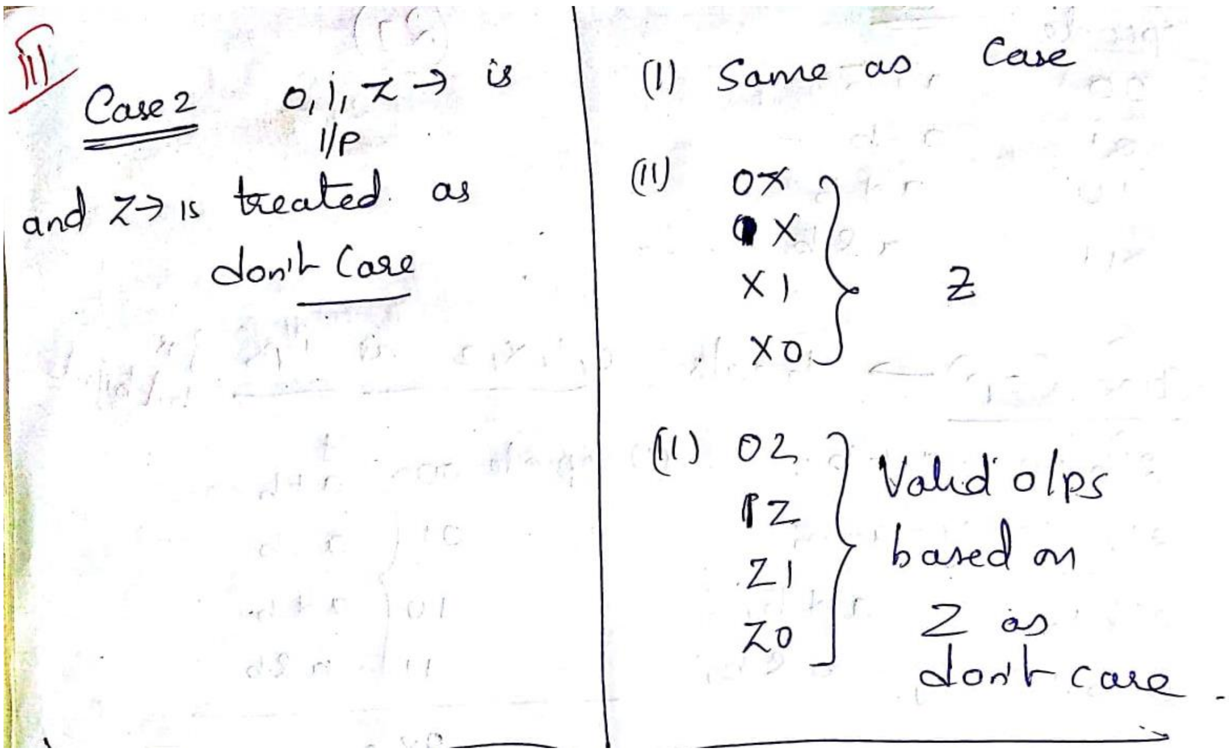
casex (op) → 0,1,x as i/p either 0 or 1

```
2'b00: y=a+b;
2'b01: y=a-b;
2'b10: y=a*b;
2'b11: y=a&b;
endcase
```

(1) same as case combination

(ii) 0x → checks $\boxed{00}$, 01 } y=a+b Valid o/p based on x as don't cares

(3) 0z, z0, 1z, z1 } o/p z



(ii) Case 2 0, 1, z → is i/p

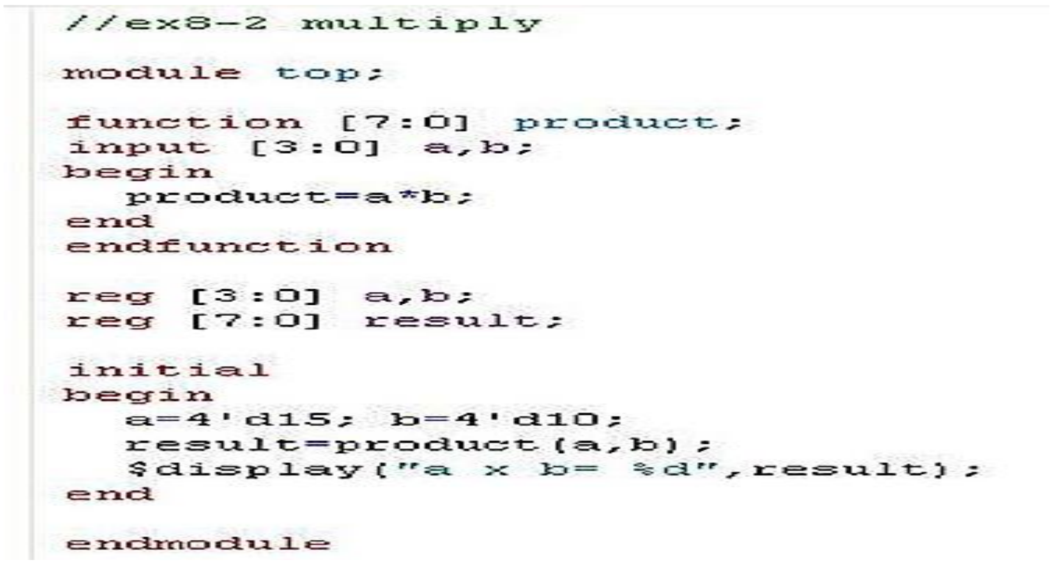
and z → is treated as don't care

(I) Same as Case

(II) 0x, 1x, x1, x0 } z

(II) 0z, 1z, z1, z0 } Valid o/ps based on z as don't care.

5. Define a function to multiply two 4-bit numbers a and b. The output is an 8-bitvalue

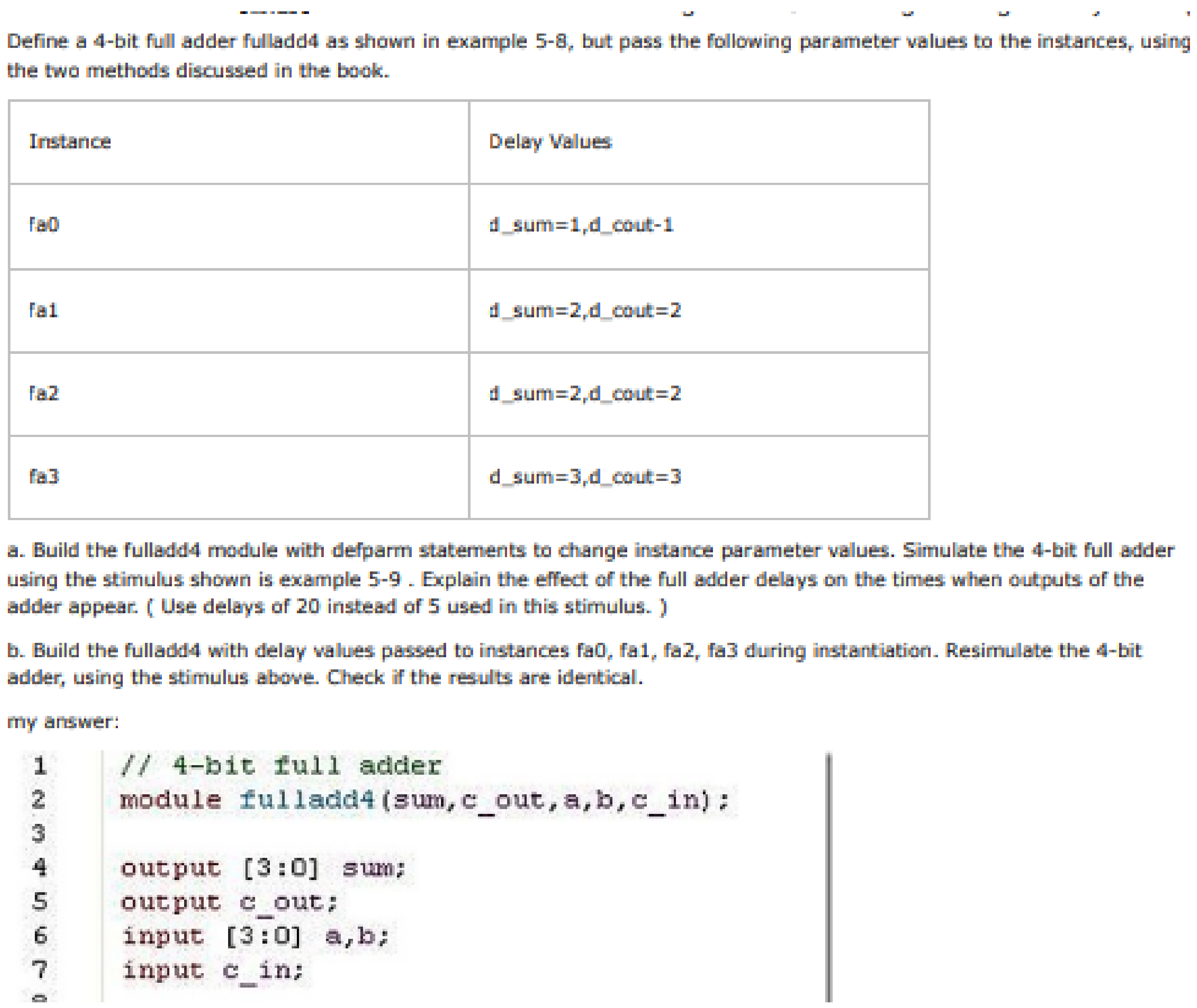


```
//ex8-2 multiply

module top;

function [7:0] product;
input [3:0] a,b;
begin
   product=a*b;
end
endfunction

reg [3:0] a,b;
reg [7:0] result;

initial
begin
   a=4'd15; b=4'd10;
   result=product(a,b);
   $display("a x b= %d",result);
end

endmodule
```

6. Create a design that uses the 4-bit full adder. Use a conditional compilation (`ifdef). Compile the fulladd4 with defparam statements if the text macro DPARAM is defined by the `define statement; otherwise, compile the fulladd4 with module instance parameter values.

A 1-bit full adder FA is defined with gates and with delay parameters as shown below. // Define a 1-bit full adder module fulladd(sum,c_out,a,b,c_in);

```
parameter d_sum=0,d_cout=0; //I/O port declarations
output sum,c_out;
 input a,b,c_in; //Internal nets
 wire s1,c1,c2;
//Instantiate logic gate primitives
xor(s1,a,b);
and(c1,a,b);
 xor #(d_sum) (sum,s1,c_in); //delay on output sum is d_sum
and (c2,s1,c_in);
or #(d_out) (c_out,c2,c1); //delay on output c_out is d_cout
endmodule
```



Define a 4-bit full adder fulladd4 as shown in example 5-8, but pass the following parameter values to the instances, using the two methods discussed in the book.

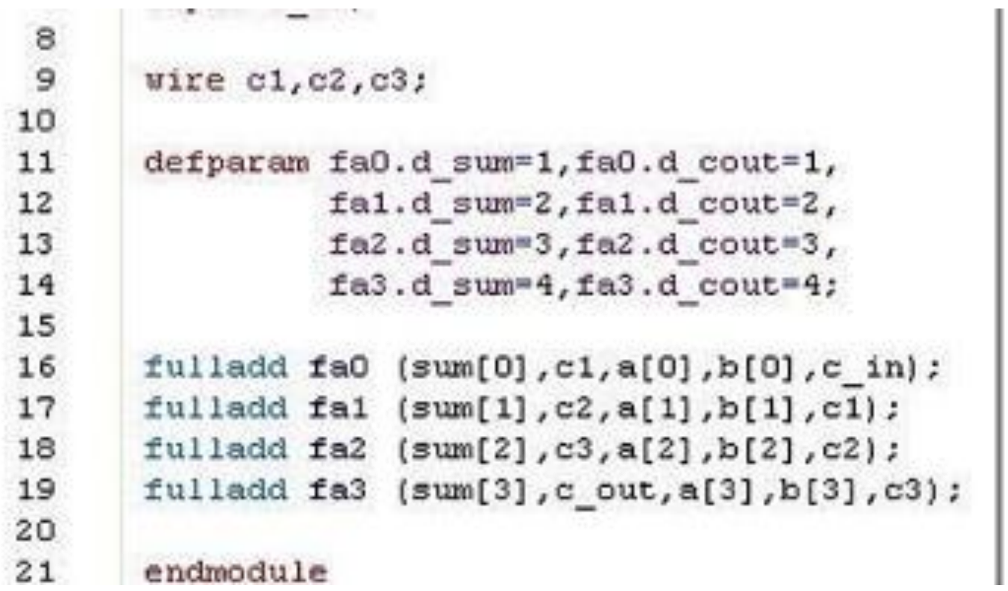
Instance	Delay Values
fa0	d_sum=1,d_cout-1
fa1	d_sum=2,d_cout=2
fa2	d_sum=2,d_cout=2
fa3	d_sum=3,d_cout=3

a. Build the fulladd4 module with defparm statements to change instance parameter values. Simulate the 4-bit full adder using the stimulus shown is example 5-9 . Explain the effect of the full adder delays on the times when outputs of the adder appear. (Use delays of 20 instead of 5 used in this stimulus.)

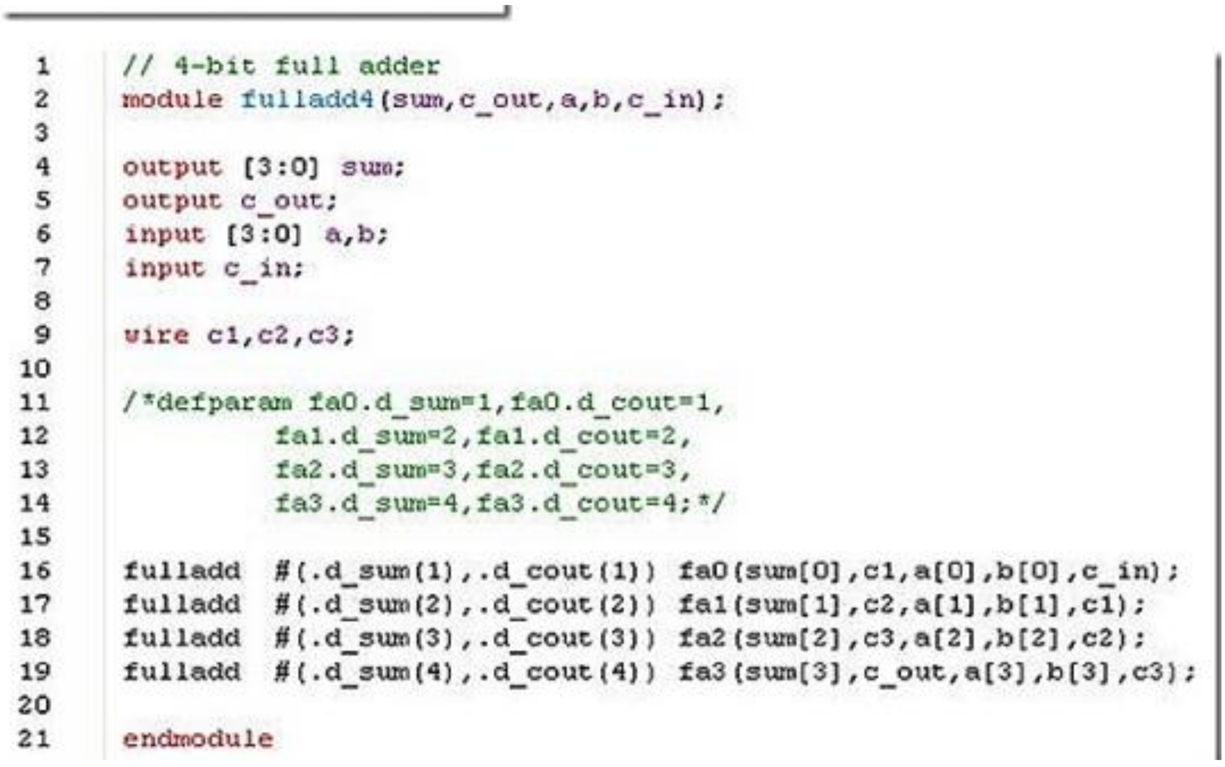
b. Build the fulladd4 with delay values passed to instances fa0, fa1, fa2, fa3 during instantiation. Resimulate the 4-bit adder, using the stimulus above. Check if the results are identical.

my answer:

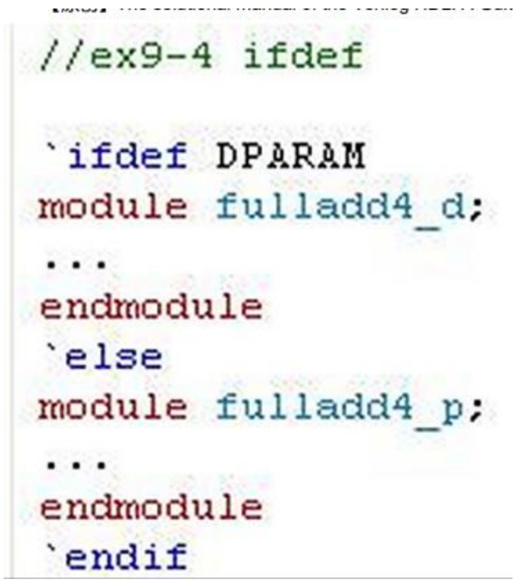
```
// 4-bit full adder
module fulladd4(sum,c_out,a,b,c_in);

output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;
```



```
wire c1,c2,c3;

defparam fa0.d_sum=1,fa0.d_cout=1,
         fa1.d_sum=2,fa1.d_cout=2,
         fa2.d_sum=3,fa2.d_cout=3,
         fa3.d_sum=4,fa3.d_cout=4;

fulladd fa0  (sum[0],c1,a[0],b[0],c_in);
fulladd fa1  (sum[1],c2,a[1],b[1],c1);
fulladd fa2  (sum[2],c3,a[2],b[2],c2);
fulladd fa3  (sum[3],c_out,a[3],b[3],c3);

endmodule
```

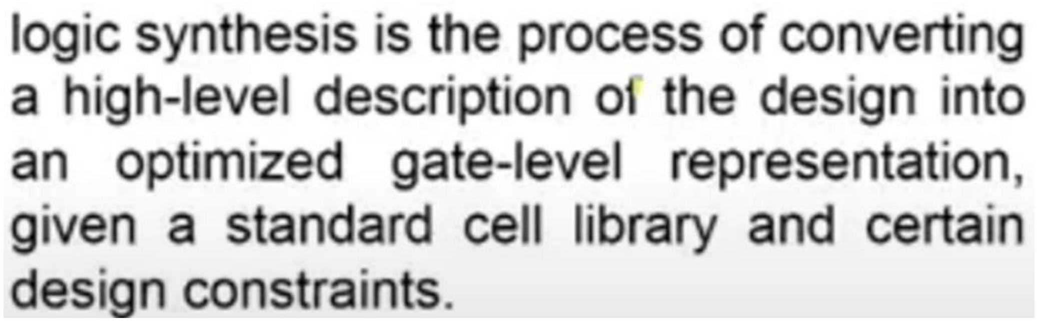


```
// 4-bit full adder
module fulladd4(sum,c_out,a,b,c_in);

output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;

wire c1,c2,c3;

/*defparam fa0.d_sum=1,fa0.d_cout=1,
         fa1.d_sum=2,fa1.d_cout=2,
         fa2.d_sum=3,fa2.d_cout=3,
         fa3.d_sum=4,fa3.d_cout=4;*/

fulladd  #(.d_sum(1),.d_cout(1)) fa0(sum[0],c1,a[0],b[0],c_in);
fulladd  #(.d_sum(2),.d_cout(2)) fa1(sum[1],c2,a[1],b[1],c1);
fulladd  #(.d_sum(3),.d_cout(3)) fa2(sum[2],c3,a[2],b[2],c2);
fulladd  #(.d_sum(4),.d_cout(4)) fa3(sum[3],c_out,a[3],b[3],c3);

endmodule
```



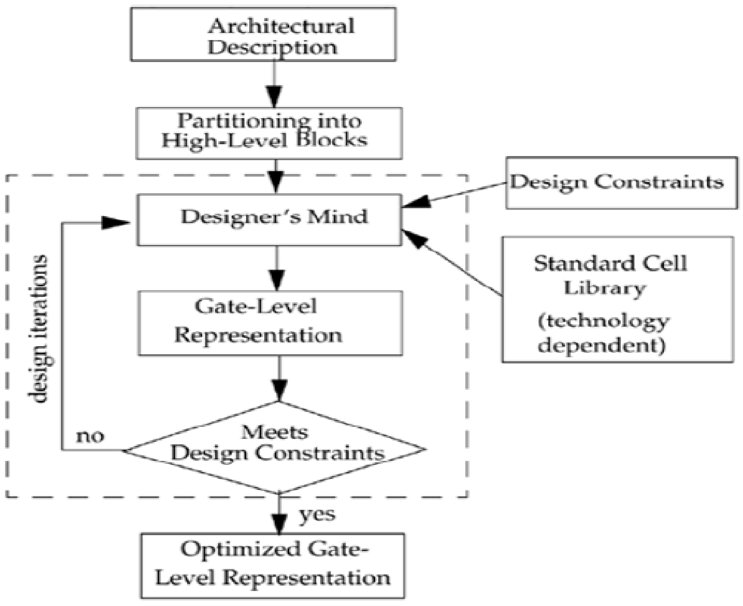
```
//ex9-4 ifdef

`ifdef DPARAM
module fulladd4_d;
...
endmodule
`else
module fulladd4_p;
...
endmodule
`endif
```

7. What is logic synthesis? Explain the basic computer-aided logic synthesis using flow chart also List the problems addressed by automated logic synthesis.



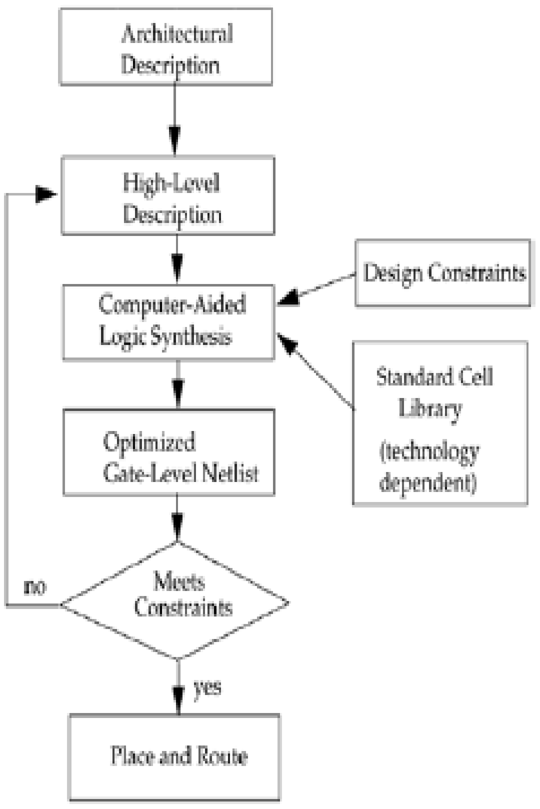
logic synthesis is the process of converting a high-level description of the design into an optimized gate-level representation, given a standard cell library and certain design constraints.

Figure 14-1. Designer's Mind as the Logic Synthesis Tool



Architectural Description
Partitioning into High-Level Blocks
Design Constraints
Designer's Mind
Standard Cell Library (technology dependent)
design iterations
Gate-Level Representation
no
Meets Design Constraints
yes
Optimized Gate-Level Representation

Figure 14-2. Basic Computer-Aided Logic Synthesis Process

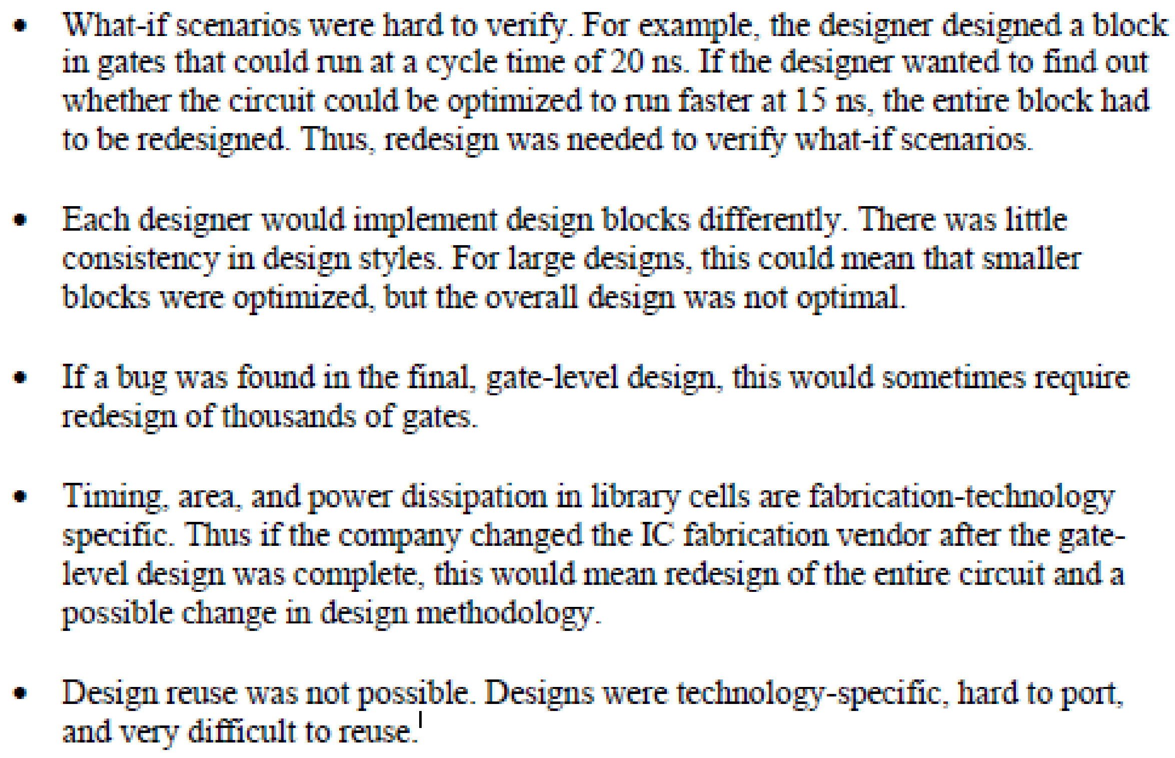


Architectural Description
High-Level Description
Design Constraints
Computer-Aided Logic Synthesis
Standard Cell Library (technology dependent)
Optimized Gate-Level Netlist
no
Meets Constraints
yes
Place and Route

- **Impact of Logic Synthesis**

- Logic synthesis has revolutionized the digital design industry by significantly improvingproductivity and by reducing design cycle time. Before the days of automated logicsynthesis, when designs were converted to gates manually, the design process had thefollowing limitations:

For large designs, manual conversion was prone to human error. A small gate missed somewhere could mean redesign of entire blocks.
• The designer could never be sure that the design constraints were going to be met until the gate-level implementation was completed and tested.
• A significant portion of the design cycle was dominated by the time taken to convert a high-level design into gates.
• If the gate-level design did not meet requirements, the turnaround time for redesign of blocks was very high.



- What-if scenarios were hard to verify. For example, the designer designed a block in gates that could run at a cycle time of 20 ns. If the designer wanted to find out whether the circuit could be optimized to run faster at 15 ns, the entire block had to be redesigned. Thus, redesign was needed to verify what-if scenarios.

- Each designer would implement design blocks differently. There was little consistency in design styles. For large designs, this could mean that smaller blocks were optimized, but the overall design was not optimal.

- If a bug was found in the final, gate-level design, this would sometimes require redesign of thousands of gates.

- Timing, area, and power dissipation in library cells are fabrication-technology specific. Thus if the company changed the IC fabrication vendor after the gate-level design was complete, this would mean redesign of the entire circuit and a possible change in design methodology.

- Design reuse was not possible. Designs were technology-specific, hard to port, and very difficult to reuse.[1]