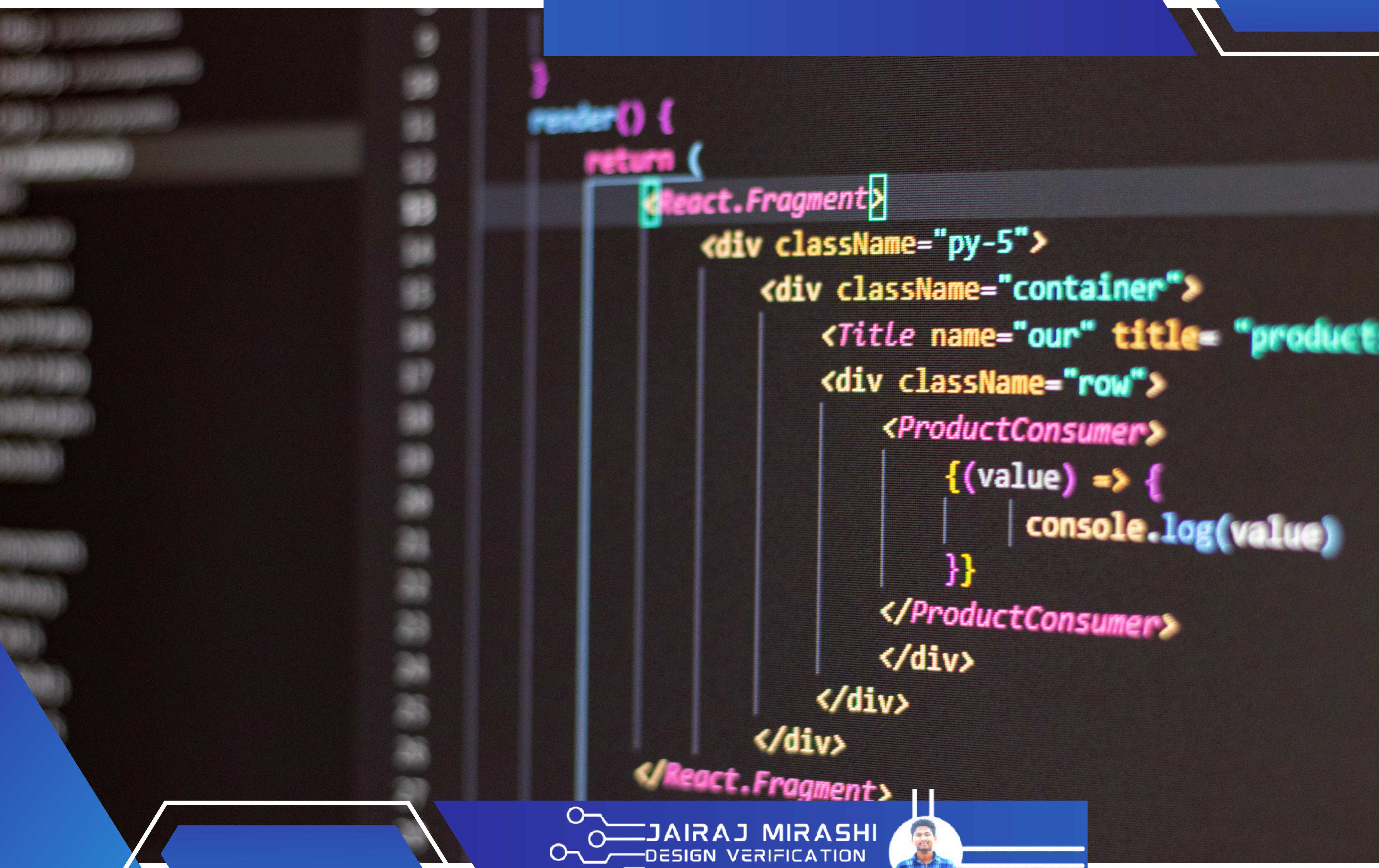
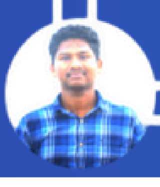
VERILOG INTERVIEW HANDBOOK



```
}
render() {
  return (
    <React.Fragment>
      <div className="py-5">
        <div className="container">
          <Title name="our" title= "product
          <div className="row">
            <ProductConsumer>
              {(value) => {
                console.log(value)
              }}
            </ProductConsumer>
          </div>
        </div>
      </div>
    </React.Fragment>
```

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER



Basic Level Questions

1. Difference between blocking and non-blocking assignments

Blocking Assignments: The blocking assignment statements are executed sequentially by evaluating the RHS operand and finishes the assignment to LHS operand without any interruption from another Verilog statement. Hence, it blocks other assignments until the current assignment completes and is named a "blocking assignment".

Ex: a = 5;

Non-Blocking Assignments: The non-blocking assignment statement starts its execution by evaluating the RHS operand at the beginning of a time slot and schedules its update to the LHS operand at the end of a time slot. Other Verilog statements can be executed between the evaluation of the RHS operand and the update of the LHS operand. As it does not block other Verilog statement assignments, it is called a non-blocking assignment.

Ex: a <= 5;

2. Difference between task and function

Function	Task
Can not contain simulation delay, so execute in the same time unit. It can not contain @, wait, negedge, and posedge time-controlled statements.	can or can not contain a simulation time delay(#), @, wait, negedge, and posedge time-controlled statements.
Can return a single value	Can return multiple values as output or inout argument. It can not return a value that a function can do.
Can not call another task	Can call another function or task

`module function_example;`
 	
`  function compare(input int a, b);`
`    if(a>b)`
`      $display("a is greater than b");`
`    else if(a<b)`
`      $display("a is less than b");`
`    else`
	`module task_example;`
	
	`  task compare(input int a, b, output`
	`done);`
	`    if(a>b)`
	`      $display("a is greater than b");`
	`    else if(a<b)`
	`      $display("a is less than b");`

```
      $display("a is equal to b");
    return  1;  //  Not  mandatory  to
write
  endfunction

  initial begin
    compare(10,10);
    compare(5, 9);
    compare(9, 5);
  end
endmodule
```

```
    else
      $display("a is equal to b");

    #10;
    done = 1;
  endtask

  initial begin
    bit done;
    compare(10,10, done);
    if(done)     $display("comparison
completed at time = %0t", $time);
    compare(5,9, done);
    if(done)     $display("comparison
completed at time = %0t", $time);
    compare(9,5, done);
    if(done)     $display("comparison
completed at time = %0t", $time);
  end
endmodule
```

3. Difference between wire and reg

Net types:

1) The net (wire, tri) is used for physical connection between structural elements.
2) Value is assigned by a continuous assignment or a gate output or port of a module.
3) It can not store any value. The values can be either read or assigned.
4) Default value – z

Register type:

1) The register (reg, integer, time, real, real-time) represents an abstract data storage element and they are not the physical registers.
2) Value is assigned only within an initial or an always statement.
3) It can store the value.
4) Default value – x

4. What is generate block in Verilog and its usage?

A generate block in Verilog is used to dynamically generate synthesizable code during the elaboration phase. It helps automate the replication of module instances or repeated code segments, enhancing design scalability and manageability. The primary uses include:

Code Repetition: It allows for the generation of multiple instances of modules or repetitive code structures, reducing manual coding.

Conditional Instantiation: It enables conditional instantiation of code based on parameters, though parameters themselves cannot be declared within the generate block.

Control Scope: The block controls variables, functions, tasks, and instantiation declarations.

Inside a generate block, you can use data types like integers, real numbers, nets, regs, time, and events. It also supports structures like modules, gate primitives, continuous assignments, initial and always blocks, and user-defined primitives. However, it does not allow port declarations, specify blocks, or parameter declarations.

There are two main types of generate constructs:

Generate Loop: Uses genvar for index variables in loop constructs, similar to a for-loop but specific to compile-time generation.

Generate Conditional: Uses if-else and case structures to conditionally generate code.

Here’s a quick example using a generate loop:

```
genvar k;
generate
  for (k = 0; k < 4; k++) begin
    always@(posedge clk) begin
      val[k] = a[k] & b[k];
    end
  end
endgenerate
```

5. Difference between while and do-while loop

In the while loop, a condition is checked first, and if it holds true statements will be executed else the loop terminates.

In do while loop, even if a condition is not true, a loop can execute at once.

Example of do while when a condition is not true:

```
module do_while_example;
  int count = 2;
  initial begin
    do begin
      $display("Value of count = %0d", count);
      count++;
    end
    while(count<1);
  end
endmodule
```

6. What is an automatic keyword in the task?

The automatic keyword specifies the variable's scope within a task i.e. memory is allocated for the variables in the task and deallocated once the task completes execution. All variables declared in an automatic task are automatic variables unless they are specifically mentioned as a static variable.

7. Explain the difference between a static and automatic function with example.

Static and Automatic Functions

1) By default, functions declared are static except they are declared inside a class scope. If the function is declared within class scope, they behave as an automatic function by default unless they are specifically mentioned as static functions. We will discuss more on this concept in class (OOP) concepts.
2) All variables declared in a static function are static variables unless they are specifically mentioned as an automatic variable.
3) All variables declared in an automatic function are automatic variables unless they are specifically mentioned as a static variable.

To understand the scope of variables in functions, static and automatic variables are declared in each static, automatic, and normal function.

```
module task_example;

  task static increment_static();
    static int count_A;
    automatic int count_B;
    int count_C;

    count_A++;
    count_B++;
    count_C++;
    $display("Static: count_A = %0d, count_B = %0d, count_C = %0d", count_A,
count_B, count_C);
  endtask

  task automatic increment_automatic();
    static int count_A;
    automatic int count_B;
    int count_C;

    count_A++;
    count_B++;
    count_C++;
    $display("Automatic: count_A = %0d, count_B = %0d, count_C = %0d", count_A,
count_B, count_C);
  endtask

  task increment();
    static int count_A;
    automatic int count_B;
    int count_C;

    count_A++;
    count_B++;
    count_C++;
    $display("Normal: count_A = %0d, count_B = %0d, count_C = %0d", count_A,
count_B, count_C);
  endtask

  initial begin
    $display("Calling static tasks");
    increment_static();
    increment_static();
    increment_static();
    $display("\nCalling automatic tasks");
    increment_automatic();
    increment_automatic();
    increment_automatic();
    $display("\nCalling normal tasks: without static/automatic keyword");
    increment();
    increment();
    increment();

    //Accessing variables using task
    // count_A
    $display("\nStatic: count_A = %0d", increment_static.count_A);
    $display("Automatic: count_A = %0d", increment_automatic.count_A);
    $display("Normal: count_A = %0d", increment.count_A);

    // count_B: Hierarchical reference to automatic variable is not legal.
    /*
    $display("\nStatic: count_B = %0d", increment_static.count_B);
    $display("Automatic: count_B = %0d", increment_automatic.count_B);
    $display("Normal: count_B = %0d", increment.count_B);
    */
    // count_C
    $display("\nStatic: count_C = %0d", increment_static.count_C);
    //$display("Automatic:  count_C  =  %0d",  increment_automatic.count_C);  //
illegal reference to automatic variable
    $display("Normal: count_C = %0d", increment.count_C);
  end
endmodule
```

OUTPUT

```
Calling static functions
Static: count_A = 1, count_B = 1, count_C = 1
Static: count_A = 2, count_B = 1, count_C = 2
Static: count_A = 3, count_B = 1, count_C = 3

Calling automatic functions
Automatic: count_A = 1, count_B = 1, count_C = 1
Automatic: count_A = 2, count_B = 1, count_C = 1
Automatic: count_A = 3, count_B = 1, count_C = 1

Calling normal functions: without static/automatic keyword
Normal: count_A = 1, count_B = 1, count_C = 1
Normal: count_A = 2, count_B = 1, count_C = 2
Normal: count_A = 3, count_B = 1, count_C = 3

Static: count_A = 3
Automatic: count_A = 3
Normal: count_A = 3
Static: count_C = 3
Normal: count_C = 3
```

8. Difference between $stop and $finish.

$stop suspends the simulation and puts a simulator in an interactive mode.

$finish exits the simulation.

9. Difference between $random and $urandom

Both generate 32-bit pseudorandom numbers, but $random generates signed whereas $urandom generates unsigned numbers.

```
EXAMPLE:
        module Tb();
            integer address;
              integer data;

                initial
                    begin
                        repeat(5)
                            begin
                              address = $random()%10; // signed numbers
                               data = $urandom()%10; // unsigned numbers

                              $display("address = %0d;",address);
                              $display("data = %0d;",data);
                            end
                    end

            endmodule


RESULTS:

address = 8;
data = 2;
address = -9;
data = 0;
address = -9;
data = 2;
address = -9;
data = 2;
address = 7;
data = 6;
```

Intermediate level questions

1. What is the default value of wire and reg?

The default value of the wire or net is z

The default value of the reg is x

2. Explain Regular delay control, Intra-assignment delay control

Regular delay control:

The regular delay control delays the execution of the entire statement by a specified value. The non-zero delay is specified at the LHS of the procedural statement.

Example: #5 data = i_value;

In this case, the result signal value will be updated after 5-time units for change happen in its input.

Intra-assignment delay:

Intra-assignment delay control delays computed value assignment by a specified value. The RHS operand expression is evaluated at the current simulation time and assigned to the LHS operand after a specified delay value.

Example: data = #5 i_value;

3. Difference between full and parallel case

Full Case:

In a full case statement, case statements cover every possible input value is explicitly specified and there are no unspecified or “don’t care” conditions.

Example:

```
case (input)
  3'b000: ………
  3'b001: ………
  3'b010: ………
  3'b011: ………
  3'b100: ………
  3'b101: ………
  3'b110: ………
  3'b110: ………
  default: // any other input case which is not covered
endcase
```

Parallel Case:

In a parallel case statement, multiple case items can match the input value simultaneously and the corresponding behaviors for that will be executed in parallel.

Example:

```
case (input)

  4'b0?: ………
  4'b1?: ………
  // other cases
  default: // any other input case which is not covered
endcase
```

4. Difference between casex and casez

casex and cazez statements

The case statement also has a total of three variations: case, casex and casez. Note the following differences.

1) case: considers x and z as it is (as shown in above example). If an exact match is not found, the default statement will be executed.
2) casex: considers all x and z values as don't care.
3) casez: considers all z values as don't cares. The z can also be specified as ?

casez statement example

```
module casez_example(
  input  [1:0] data,
  output reg [3:0] out);

  always @(*) begin
    casez(data)
      2'b0z: out = 1;
      2'bz0: out = 2;
      2'b1z: out = 3;
      2'bxz: out = 4;

      2'b0x: out = 5;
      2'bx0: out = 6;
      2'b1x: out = 7;
      2'bx1: out = 8;

      default: $display("Invalid sel
input");
    endcase
  end
endmodule
```

casex statement example

```
module casex_example(
  input  [1:0] data,
  output reg [3:0] out);

  always @(*) begin
    casex(data)
      2'b0z: out = 1;
      2'bz0: out = 2;
      2'b1z: out = 3;
      2'bxz: out = 4;

      2'b0x: out = 5;
      2'bx0: out = 6;
      2'b1x: out = 7;
      2'bx1: out = 8;

      default: $display("Invalid sel
input");
    endcase
  end
endmodule
```

```
Output:

data = x1 -> out = 4
data = 0x -> out = 1
data = x0 -> out = 2
data = z1 -> out = 1
data = 0z -> out = 1
data = z0 -> out = 1
data = 1z -> out = 2
```

```
Output:

data = x1 -> out = 1
data = 0x -> out = 1
data = x0 -> out = 1
data = z1 -> out = 1
data = 0z -> out = 1
data = z0 -> out = 1
data = 1z -> out = 2
```

Note:

1) In simple terms, casez ignores bit positions having z value alone and casex ignores bit positions having x or z values.
2) The 'casez' is more likely used as compared to 'casex' as the 'casez' does not ignore bit positions having x values and 'casex' is not synthesizable as well.

5. What is synchronous and asynchronous reset? Can you explain using DFF and write their Verilog code?

In asynchronous reset, a flip flop gets reset as soon as the 'reset' signal is asserted. Thus, in Verilog implementation, the 'reset' signal has to be written in the sensitivity list of always block.

In synchronous reset, a flip flop gets reset at the active 'clock' edge when the 'reset' signal is asserted.

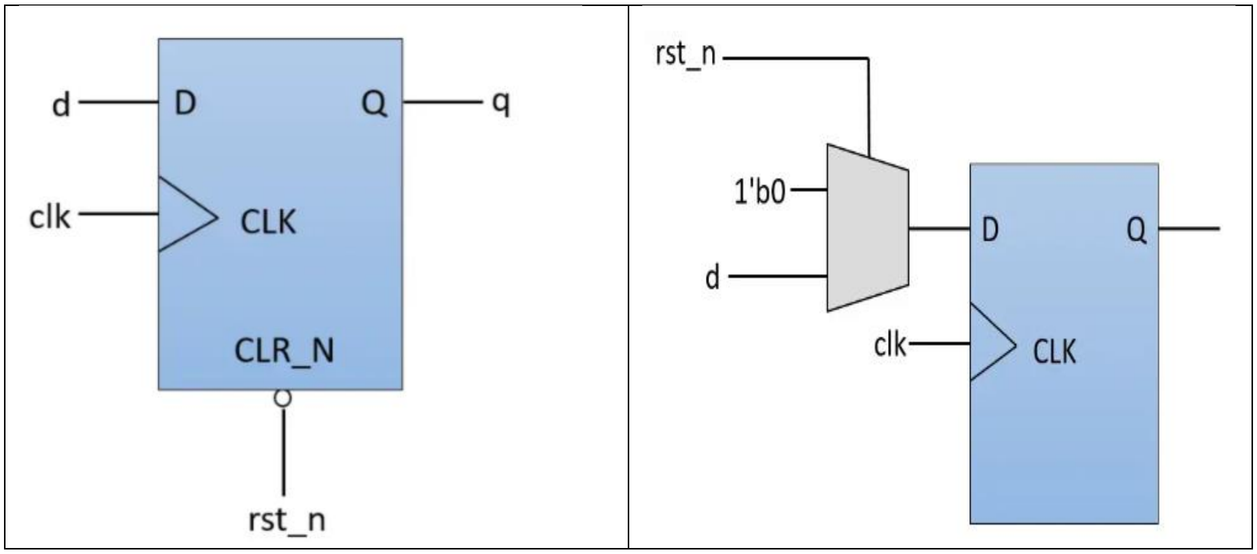
Thus, in Verilog implementation, the 'reset' signal must not be written in the sensitivity list of always block.

Asynchronous Reset

```
module D_flipflop (
  input clk, rst_n,
  input d,
  output reg q
  );

  always@(posedge clk or negedge
rst_n) begin
    if(!rst_n) q <= 0;
    else       q <= d;
  end

endmodule
```

Synchronous Reset

```
module D_flipflop (
  input clk, rst_n,
  input d,
  output reg q
  );

  always@(posedge clk) begin
    if(!rst_n) q <= 0;
    else       q <= d;
  end

endmodule
```



6. What is #0 in Verilog and its usage?

Zero delay control is used to control execution order when multiple procedural blocks try to update values of the same variable. Both always and initial blocks execution order is non-deterministic as they start evaluation at the same simulation time. The statement having zero control delay executes last, thus it avoids race conditions.

Example:

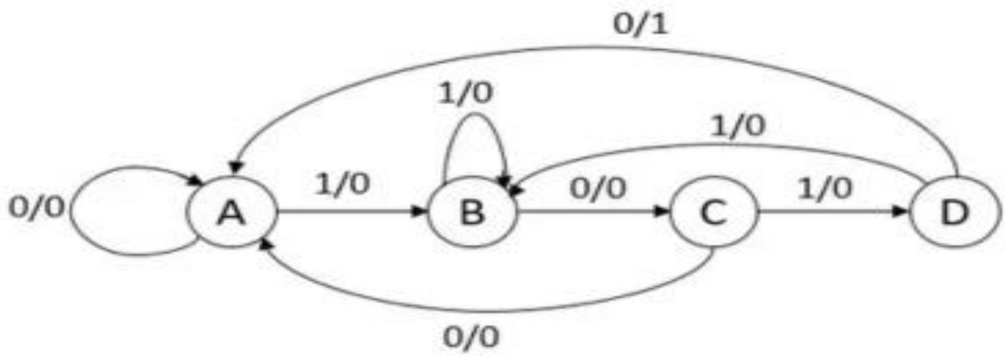
```
reg [2:0] data;
initial begin
  data = 2;
end
initial begin
  #0 data = 3;
End
```

Without zero delay control, the 'data' variable may have a value of either 2 or 3 due to race conditions. Having zero delay statement as specified in the above code guarantees the outcome to be 3. However, it is not recommended to assign value to the variable at the same simulation time.

7. How to generate two different clocks in testbench?

```
module tb;
  bit clk1, clk2;
  initial forever #5ns clk1 = ~clk1;
  initial forever #4ns clk2 = ~clk2;
endmodule
```

8. Design overlapping and non-overlapping FSM for sequence detector 1010.



1010 Non-Overlapping Mealy Sequence Detector

```
module seq_detector_1010(input bit clk, rst_n, x, output z);
  parameter A = 4'h1;
  parameter B = 4'h2;
  parameter C = 4'h3;
  parameter D = 4'h4;

  bit [3:0] state, next_state;
  always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
      state <= A;
    end
    else state <= next_state;
  end

  always @(state or x) begin
    case(state)
      A: begin
           if(x == 0) next_state = A;
           else       next_state = B;
         end
      B: begin
           if(x == 0) next_state = C;
           else       next_state = B;
         end
      C: begin
           if(x == 0) next_state = A;
           else       next_state = D;
         end
      D: begin
        if(x == 0) next_state = A; //This state only differs when compared 
with Mealy Overlaping Machine
           else       next_state = B;
         end
      default: next_state = A;
    endcase
  end
  assign z = (state == D) && (x == 0)? 1:0;
endmodule


//Testbench Code

module TB;
  reg clk, rst_n, x;
  wire z;

  seq_detector_1010 sd(clk, rst_n, x, z);
  initial clk = 0;
  always #2 clk = ~clk;

  initial begin
    x = 0;
    #1 rst_n = 0;
    #2 rst_n = 1;

    #3 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 1;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #4 x = 1;
    #4 x = 0;
    #10;
    $finish;
  end

  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end
endmodule
```

9. Write a Verilog code for D-Latch.

The latch has two inputs 'data (D)' and 'clock (clk)'

One output data (Q)

If clk = 1, then data passes to the output Q

If clk = 0, then data is not passed to the output Q

```
module d_latch (input d, en, rst_n, output reg q);
  always @(en or rst_n or d) begin
    if(!rst_n) begin
      q <= 0;
    end
    else begin
      if(en) q <= d;
    end
  end
  endmodule
```

10. How can you override the existing parameter value?

Verilog parameter is used to pass a constant to the module when it is instantiated. It is not considered under net or reg data types. The parameter value can not be changed at run time. Verilog allows changing parameter values during compilation time using the 'defparam' keyword. The 'defparam' is used as overriding the parameter value using a hierarchical name of the module instance.

The parameter value can be updated in two ways

1. Pass constant or define based value
2. Use the 'defparam' keyword

Syntax:

```
Module <module_name> #(<parameter list>) <port_list>;


Example:

module param_example #(parameter DATA_WIDTH = 8, ID_WIDTH = 32) (data, id);
  input bit [DATA_WIDTH-1: 0] data;
  input bit [ID_WIDTH-1: 0] id;

  initial begin
    $display("DATA_WIDTH = %0d, ID_WIDTH = %0d", DATA_WIDTH, ID_WIDTH);

    $display("data = %0d,  id = %0d", data, id);
  end
endmodule
```

```
Testbench:

`define D_WIDTH 32
`define I_WIDTH 8

module tb_top;

  param_example p1(.data(2), .id(1)); // without passing parameter
  param_example #(4, 16) p2(.data(3), .id(2)); // constant parameter passing
  param_example #(`D_WIDTH, `I_WIDTH) p3(.data(6), .id(3));  // macro define
based parameter passing

  param_example p4(.data(9), .id(4));
  // Change parameter value using defparam
  defparam p4.DATA_WIDTH = 10;
  defparam p4.ID_WIDTH = 16;
endmodule
```

11. What is Synthesis?

The process of converting hardware description language like Verilog code into the equivalent netlist design that has flip-flops, logic gates, and required digital circuit components.

12. Write an RTL code to generate 60% duty cycle clock.

```
`define CLK_PERIOD 10ns

module clk_gen;
  realtime on_t = `CLK_PERIOD * 0.6;
  realtime off_t = `CLK_PERIOD * 0.4;
  bit clk;

  always begin
    #on_t clk = 0;
    #off_t clk = 1;
  end

  initial begin
    clk = 1;
    #50 $finish;
  end

  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(0);
  end
endmodule
```

13. Write an RTL code to generate 100MHz clock.

To generate clock frequency, time period needs to be calculated.

Time Period = 1/frequency = 1/100MHz = 10ns

With a 50% duty cycle, clock has to flip a bit after every 5ns.

```
module clk_gen;
  reg clk;
  always #5 clk = ~clk;
endmodule
```

14. Difference between `define and `include.

`define is a compiler directive that substitutes itself in the code with a defined context. In simple words, wherever macro is used, it is replaced with macro context and gives compilation error in case of misuse.

The `include is also a compiler directive is used to include another filename. The double quote "<file_name>" is used in the `include directive. It is widely used to include library files, and common code instead of pasting the same code repeatedly.

15. What will be output of the following code?

```
always@(clock) begin
  a = 0;
  a <= 1;
  $display(a);
end
```

In brief, A single time cycle or slot is divided into various regions and that helps to schedule the events. The scheduling an event terminology describes keeping all such events in an event queue and processing them in the correct order. In this case, there are 3 events scheduled in an event queue.

1. Active event – blocking statement "a = 0".
2. A non-blocking event – non-blocking statement "a <= 1"
3. Monitor event – "$display(a)".

$display acknowledges and displays what value is being calculated as an active region. Also, a non-blocking event computes (not assigned to RHS 'a' variable) LHS during the active region. Thus, it will pick up the value calculated by the statement "a = 0". But in the next clock cycle, the value of a = 1.

Difficult level questions

1. Why always block is not used inside a program block?

The program block is generally used to develop a test case that initiates a stimulus and then it should end. But the ‘always’ block does not have any provision to end by itself. Thus, we can not have a program block. Even if you try to do so, a compilation error is expected.

For a need basic, we can use the ‘forever’ loop as a work-around with a ‘break’ statement to terminate the loop as per requirement.

2. What is FIFO? What are underflow and overflow conditions in FIFO? Write Verilog code for the design.

FIFO stands for first in first out which means that the first element enters into the buffer and comes out first.

Underflow: When an attempt is made to read data from an empty FIFO, it is called an underflow. The design should have an ‘empty’ flag to avoid getting invalid values

Overflow: When an attempt is made to write data into the already full FIFO, it is called an overflow. The design should have a ‘full’ flag to avoid losing the data sent from the previous module.

3. What will happen if there is no else part in if-else?

In such a case, the missing ‘else’ (i.e. valid = 0 in the below case) infers to latch in synthesis.

Example:

```
always@(*) begin
  if(en) begin
    data <= 8'hFF;
  end
end
```

4. Swap register content with and without using an extra register.

Without using an extra register:

```
always @(posedge clk) begin
  m <= n;
  n <= m;
end
```

Using an extra register (in case the interviewer asks):

Here, temp is an extra register used.

```
always @(posedge clk) begin
  temp = n;
  n = m;
  m = temp;
end
```

5. What is infer latch means? How can you avoid it?

Infer latch means creating a feedback loop from the output back to the input due to missing if-else condition or missing 'default' in a 'case' statement.

Infer latch indicates that the design might not be implemented as intended and can result in race conditions and timing issues.

How to avoid it?

1. Always use all branches in the 'if' and 'case' statements.
2. Use default in the 'case' statement.
3. Have a proper code review.
4. Use lint tools, and logical-equivalence-check tools

6. What is parameter overriding in Verilog?

Verilog parameter is used to pass a constant to the module when it is instantiated. The parameter value can not be changed at run time.

There are two ways to override the parameters in Verilog

[1]. During module instantiation

```
module param_example #(parameter DATA_WIDTH = 8, ID_WIDTH = 32) (data, id);
param_example #(4, 16) p2(.data(3), .id(2));
```

[2].Using defparam

```
defparam p4.DATA_WIDTH = 10;
defparam p4.ID_WIDTH = 16;
```

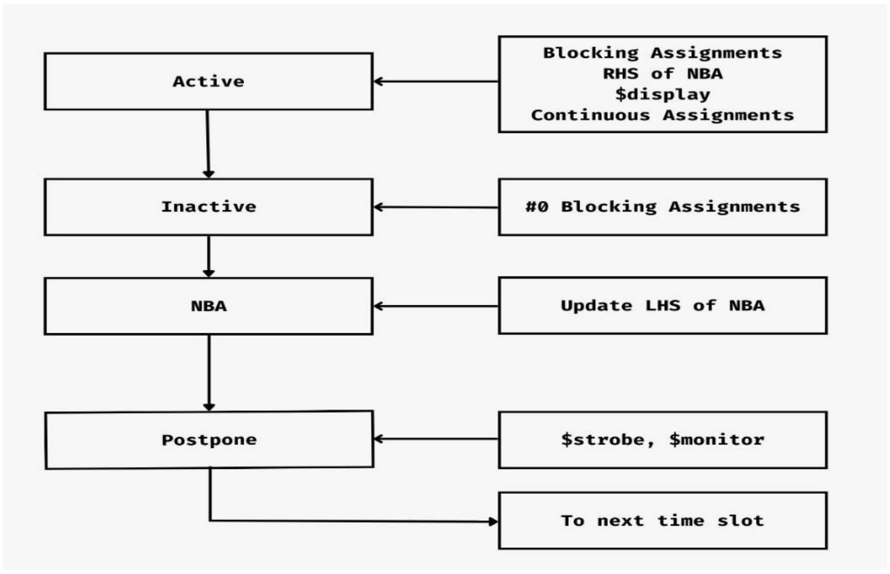
7. Write a Verilog code for 5:1 MUX

5:1 MUX selects one out of 5 signals based on 3-bit select input and forwards it to single-bit output.

```
module mux_5_1 (input [4:0] i_data, [2:0] sel, output reg out);
  always@(*) begin
    case(sel)
      5'h0: out = i_data[0];
      5'h1: out = i_data[1];
      5'h2: out = i_data[2];
      5'h3: out = i_data[3];
      default: out = i_data[4];
    endcase
  end
endmodule
```

8. Can you talk about the Verilog event scheduler?

The Verilog scheduling semantics is used to describe the Verilog language element's behavior and their interaction with each other. This interaction is described for event execution and its scheduling. Verilog is like a parallel programming language in terms of blocks or process executions. Hence, the user should know the guaranteed or indeterminate execution order while using it.



9. Difference between dual port ram and FIFO.

	Dual Port RAM	FIFO
Functionality	Concurrent access to different memory access for read and write operations without causing interference.	FIFO stands for first in first out which means that the first element enters into the buffer and comes out first.
Access	It has two separate ports for read/write simultaneous access.	It has one end for writing into the FIFO and another end for reading.
Control signals	The operation takes care using signals like Read/Write enable signals.	The operation takes care using read/write pointers.
Applications	Useful where simultaneous access to the memory is required like shared memory among processors, GPU, etc	Useful as a buffer to exchange data between two systems that work on different clock frequencies.

10. What is `timescale? What does `timescale 1 ns/ 1 ps in a Verilog code?

It is a 'compile directive' and is used for the measurement of simulation time and delay.

`timescale

Syntax:

```
`timescale <time_unit>/<time_precision>
```

time_unit: Measurement for simulation time and delay.

time_precision: Rounding the simulation time values means the simulator can at least advance by a specified value.

Examples

`timescale 1ns/1ns

```
`timescale 1ns/1ns
module tb;
  initial begin
    $display ("At time T=%0t",
$realtime);
    #0.45;
    $display ("At time T=%0t",
$realtime);
    #0.50;
    $display ("At time T=%0t",
$realtime);
    #0.55;
    $display ("At time T=%0t",
$realtime);
  end
endmodule
```

```
OUTPUT:
      At time T=0
      At time T=0
      At time T=1
      At time T=2
```

`timescale 1ns/1ps

```
`timescale 1ns/1ps
module tb;
  initial begin
    $display ("At time T=%0t",
$realtime);
    #0.45;
    $display ("At time T=%0t",
$realtime);
    #0.50;
    $display ("At time T=%0t",
$realtime);
    #0.55;
    $display ("At time T=%0t",
$realtime);
  end
endmodule
```

```
OUTPUT:
      At time T=0
      At time T=450
      At time T=950
      At time T=1500
```

`timescale 10ns/1ns

```
`timescale 10ns/1ns
module tb;
  initial begin
    $display ("At time T=%0t", $realtime);
    #0.45;
    $display ("At time T=%0t", $realtime);
    #0.50;
    $display ("At time T=%0t", $realtime);
    #0.55;
    $display ("At time T=%0t", $realtime);
  end
endmodule


Output:
      At time T=0
      At time T=5
      At time T=10
      At time T=16
```

Explaination:

`timescale 1ns/1ns: Since precision = 1ns, the simulator will advance its time if the delay value is greater or equal to 0.5ns. Thus, time advancement does not happen for 0.45ns delay.

`timescale 1ns/1ps: Since precision = 1ps, the simulator will advance for all the cases.

`timescale 10ns/1ns: Since precision = 1ns, the simulator will advance for all the cases. Here, the delay involved is multiplied with mentioned time units and then it rounds off based on precision to calculate actual simulation advancement.

Actual simulation time

1. (0.45*10) = 4.5ns — Rounded off to 5ns
2. (0.50*10) = 5ns
3. (0.55*10) = 5.5ns — Rounded off to 6ns

Note: if the timescale is not mentioned then the simulator takes default timescale values.

11. What will be the output of m, n, o if c is the clock?

```
logic m = c;

reg n = c;

wire o = c;
```

Since clock c needs to be generated, it has to be of reg type. If we do direct assignment as above, then m and n will have default values as x and wire o will be the same as how clock is driven.

But if we do declaration and assignment in the 'always' block then m and n can be driven same as a clock, but 'o = c' assignment can not be possible inside 'always' block as the 'o' variable is a wire type.