DAY 38 -111 DAYS VERIFICATION CHALLENGE

Topic: Behavioural modelling

Skill: Verilog

DAY 38 CHALLENGE:

1. Explain difference between case, casex & casez.

case Statement:

- The case statement performs exact matching. Each case item must exactly match the case expression, including the bits that are x or z.
- Don't care (x) and high-impedance (z) values are treated as specific values and must match exactly for the corresponding case item to be selected.

```
module case_example();
  reg [3:0] a;
  reg [1:0] b;
  always @(*) begin
    case (a)
      4'b0001: b = 2'b01;
      4'b001x: b = 2'b10; // x is treated as a specific value
      4'bzzzz: b = 2'b11; // z is treated as a specific value
      default: b = 2'b00;
    endcase
  end
endmodule
```

casex Statement:

- The casex statement ignores x and z values in both the case expression and the case items. It treats these values as wildcards, which means any bit that is x or z in either the case expression or the case item can match any corresponding bit in the other.

- This can be useful when you want to implement a case structure that can match a range of values or when certain bits in the case expression are unknown or don't care.

```
module casex_example();
  reg [3:0] a;
  reg [1:0] b;
  always @(*) begin
    casex (a)
      4'b0001: b = 2'b01;
      4'b001x: b = 2'b10; // x is treated as a wildcard
      4'bzzzz: b = 2'b11; // z is treated as a wildcard
      default: b = 2'b00;
    endcasex
  end
endmodule
```

casez Statement:

- The casez statement ignores only z values in both the case expression and the case items. It treats z values as wildcards, which means any bit that is z in either the case expression or the case item can match any corresponding bit in the other.
- x values are treated as exact values and must match exactly.
- This can be useful when you want to ignore high-impedance states but still consider x values as specific states that need exact matching.

```
module casez_example();
  reg [3:0] a;
  reg [1:0] b;
  always @(*) begin
    casez (a)
      4'b0001: b = 2'b01;
      4'b001x: b = 2'b10; // x is treated as a specific value
      4'bzzzz: b = 2'b11; // z is treated as a wildcard
      default: b = 2'b00;
    endcasez
```

```
  end
endmodule
```

2. Explain sequential & parallel blocks with an example.

sequential and parallel blocks are used to control the execution order of statements within always blocks. These blocks are defined using begin...end (for sequential execution) and fork...join (for parallel execution).

Sequential Blocks (begin...end)

Sequential blocks execute the statements one after the other, in the order they are written. This means that each statement must complete before the next statement begins

```
module sequential_example();
  reg [3:0] a, b, c;
  initial begin
    a = 4'b0000; // Statement 1
    b = 4'b0001; // Statement 2
    c = a + b;   // Statement 3
  end
endmodule
```

Parallel Blocks (fork...join)

Parallel blocks execute the statements simultaneously. All statements within a fork...join block begin execution at the same time and run concurrently. The block completes when all statements have finished executing.

```
module parallel_example();
  reg [3:0] a, b, c;
  initial begin
    fork
      a = 4'b0000; // Statement 1
      b = 4'b0001; // Statement 2
      c = a + b;   // Statement 3
    join
  end
endmodule
```

here in parallel there are problem will occur because the all three statement are execute concurrently but c is depends on the a that's why at the end of execution the c is indeterminant.

```
module parallel_example_with_sync();
  reg [3:0] a, b, c;
  reg done_a, done_b;
  initial begin
    done_a = 0;
    done_b = 0;
    fork
      begin
        a = 4'b0000;
        done_a = 1;
      end
      begin
        b = 4'b0001;
        done_b = 1;
      end
      begin
        wait (done_a && done_b); // Wait for both a and b to be
assigned
        c = a + b;
      end
    join
  end
endmodule
```

this is solution of earlier problem.

3. Explain following event-based timing control mechanisms:

Event-based timing control mechanisms in Verilog are used to synchronize the execution of procedural statements based on specific events.

I. Regular Event Control

Regular Event Control is the most common method for specifying that a statement or a block of statements should execute when a particular event

occurs. This is typically done using the @ symbol followed by an event expression.

1. **Posedge and Negedge Events:**
   - Execute a block of code when there is a positive or negative edge on a signal.

   Ex. `always @(posedge clock),always @(negedge reset)`

2. **Any Change Event:**
   Ex. `always @(a or b or c)`

3. **Multiple Events:**
   Ex. `always @(posedge clock or negedge reset)`

II. named event Control

Named Event Control allows you to define custom events using the event keyword and trigger or wait for these events using the -> and @ operators, respectively. This mechanism is useful for more complex synchronization that is not directly tied to a specific signal change.

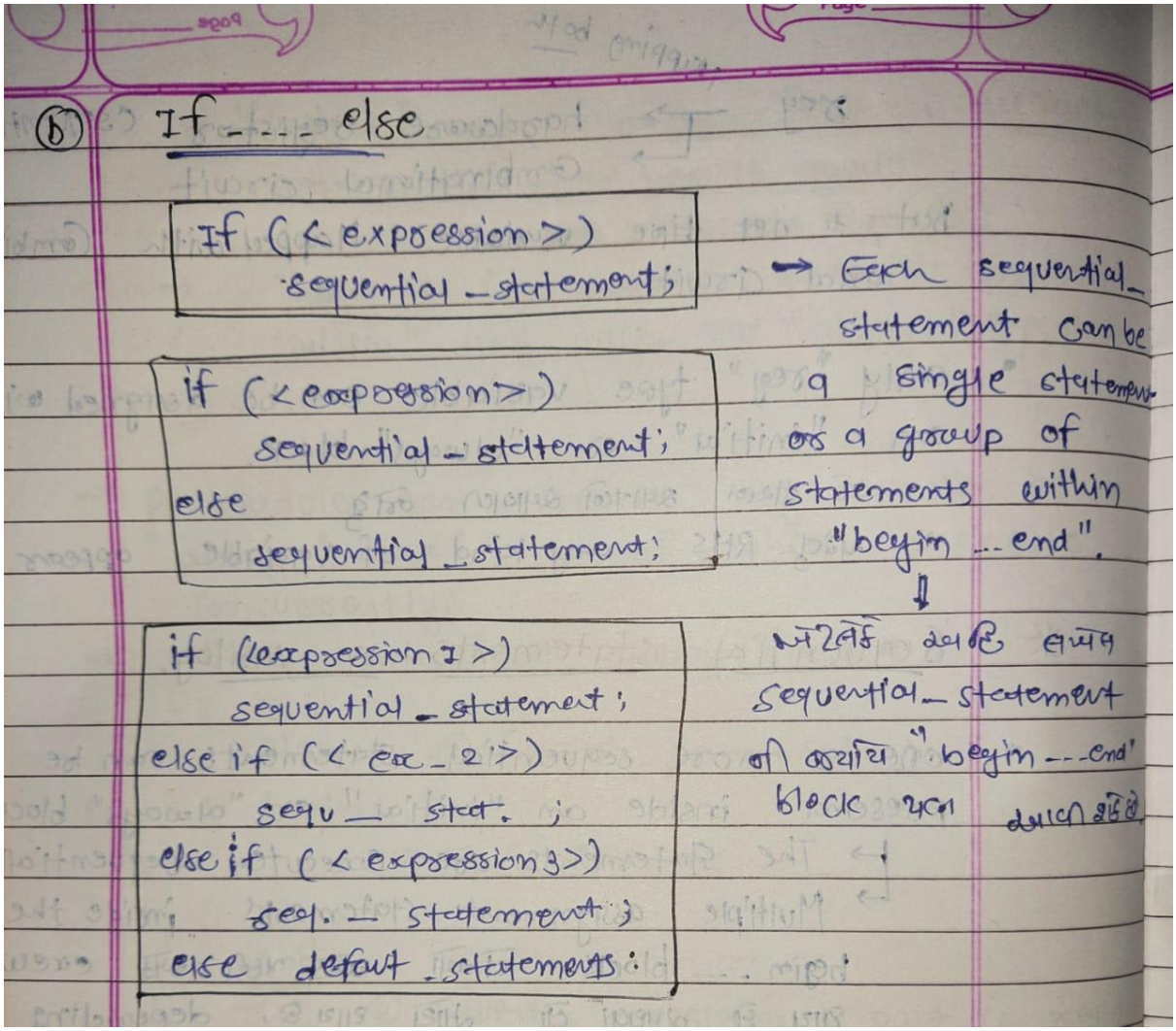
1. **Defining and Triggering an Event:**

```
module named_event_example();
  event my_event; // Define an event

  initial begin
    // Some initial code
    #10 -> my_event; // Trigger the event after 10 time units
  end
endmodule
```

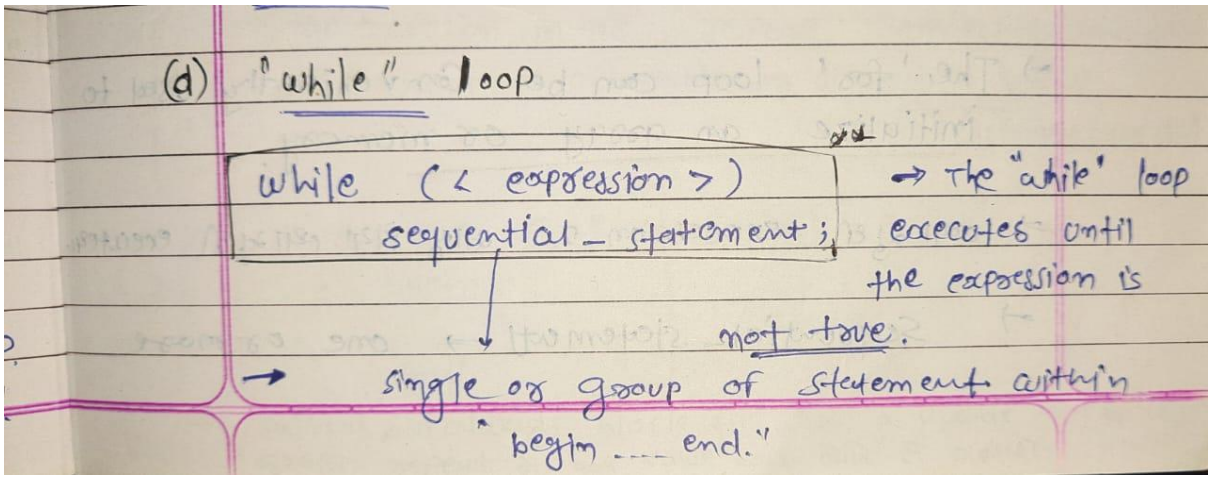
2. **Waiting for an Event:**

```
module named_event_example();
  event my_event; // Define an event

  initial begin
    // Some initial code
    @my_event; // Wait for the event to be triggered
    // Code to execute when the event is triggered
  end
endmodule
```

4. Explain the conditional statement if and else with an example.



5. Explain following looping statements: while, for



Example

```
integer mycount;
initial
  begin
    mycount = 0;
    while (mycount <= 255)
      begin
        $ display ("my count: %d", mycount);
        mycount = mycount + 1;
      end
  end
```

(c) "for" loop

```
for (expr 1 ; expr 2 ; expr 3)
    sequential_statements
```

→ for loop consist of three parts:
  a) An initial condition (expr 1)
  b) A check to see if the terminating condition is true (expr 2).
  c) A procedural assignment to change the value of control variable (expr 3)

→ The 'for' loop can be conveniently used to initialize an array or memory.

→ ક્યાં સુધી expression 2 સાચું રહ્યા ત્યાં સુધી execute

→ Sequential_statement → one or more

Example

```
integer mycount;
reg [7:0] data;
integer i;
initial
    for (mycount = 0; mycount <= 255;
                      mycount = mycount + 1)
        $ display ("My count: %d", mycount);
initial
    for (i=1; i<=100; i=i+1)
        data[i] = 8'b0;
```

6. How can we disable naming of blocks?

we can disable a named block using the disable statement. Named blocks allow for more structured and readable code, especially when dealing with complex conditional or iterative logic.

```
module example();
  reg [3:0] a;
  reg [3:0] b;
  reg en;
  initial begin : my_block
    a = 4'b0000;
    b = 4'b0001;
    if (en) begin
      a = 4'b0010;
    end else begin
      disable my_block; // Disable the named block
    end
    b = 4'b0100; // This line will not execute if the block is disabled
  end
endmodule
```

7. What is the output?

```
initial begin
     m = 1'b0 ;
end
initial begin
     #5 a = 1'b1 ;
     #25 b = 1'b0 ;
end
initial
     #50 $finish ;
```

Output : m = 1'b0, a = 1'b1, b = 1'b0