DAY 39 - 111 DAYS VERIFICATION CHALLENGE

Topic: Tasks & functions

Skill: Verilog

DAY 39 CHALLENGE:

1. Are tasks and functions re-entrant, and how are they different from static task and function calls?

Re-entrancy refers to the ability of a task or function to be paused in the middle of its execution and then safely called again ("re-entered") before its previous executions are complete.

- Tasks and functions in Verilog can be made re-entrant by declaring their variables as automatic. This ensures that each invocation gets its own copy of variables, avoiding conflicts.
- Static task and function calls use shared variables for all invocations, which can lead to conflicts and unintended interactions if re-entered.

2. Can a 'task' be called within a 'function"?

No, a task cannot be called within a function in Verilog. Functions are required to execute in zero simulation time and cannot include any time-consuming operations, while tasks can include delays and other timing controls.

3. What are the restrictions of using automatic tasks? How can I override variables in an automatic task?

Restrictions of using automatic tasks:

- Automatic tasks cannot have static variables; all variables are local to each invocation.
- Automatic tasks can be re-entrant and recursive, which is not possible with static tasks.
- They consume more memory due to the creation of new instances of variables for each call.

Overriding variables in an automatic task:

- Automatic tasks allow parameter passing by value, which can be used to override the default variable values. Here's an example:

  ```
  task automatic my_task(input int a, input int b);
    int sum;
    begin
      sum = a + b;
      $display("Sum: %d", sum);
    end
  endtask
  ```

Each call to my_task will have its own sum variable, and you can override a and b by passing different values.

4. When should you use task instead of a function in Verilog?

Use a task instead of a function in Verilog when:

- You need to perform operations that require delays (#, @, wait statements).
- You need to perform file I/O operations.
- The operation spans multiple simulation time steps.
- You need to pass more than one output or inout arguments.

5. How can I call a function like a task, that is, not have a return value assigned to a variable.

In Verilog, functions are intended to return values and be used in expressions. However, if you want to call a function without assigning its return value, you can call it and simply ignore the result. Here's an example:

function int add(int a, int b);

```
  add = a + b;
endfunction


module test;
  initial begin
    add(3, 4); // Calling function without using the return value
  end
endmodule
```

While this works syntactically, it's unconventional since the purpose of a function is to return a value. If the function has side effects (e.g., printing or logging), those effects will still occur even if the return value is not used.

6. Define a function to design 8-function ALU that takes two 4-bit inputs a & b & computes a 5-bit output

out based on 3-bit input sel as below:

select	out	select	out
000	a	100	a%1
001	a + b	101	a << 1
010	a – b	110	a >> 1
011	a/b	111	a > b

```
module ALU (
    input  [3:0] a,
    input  [3:0] b,
    input  [2:0] sel,
    output [4:0] out
);
    assign out = alu_function(a, b, sel);

    function [4:0] alu_function;
        input [3:0] a;
        input [3:0] b;
        input [2:0] sel;
        begin
            case (sel)
                3'b000: alu_function = a;           // a
                3'b001: alu_function = a + b;       // a + b
                3'b010: alu_function = a - b;       // a - b
                3'b011: alu_function = a / b;       // a / b
                3'b100: alu_function = a % 1;       // a % 1
                3'b101: alu_function = a << 1;      // a << 1
                3'b110: alu_function = a >> 1;      // a >> 1
                3'b111: alu_function = a > b;       // a > b
                default: alu_function = 5'b00000;  // default case
            endcase
        end
    endfunction
endmodule
```

7. Define a task to compute the factorial of a 4-bit number. The output is a 32-bit value. The result is assigned to output after a delay of 11-time units.

```
module FactorialCalculator (
    input [3:0] num,
    output reg [31:0] result
);
    task factorial;
        input [3:0] n;
        output reg [31:0] fact;
        integer i;
        begin
            fact = 1;
            for (i = 1; i <= n; i = i + 1) begin
                fact = fact * i;
            end
        end
    endtask
    initial begin
        #11;
        factorial(num, result);
    end
endmodule
```