DAY 41 - 111 DAYS VERIFICATION CHALLENGE

Topic: Advanced Verilog Concepts

Skill: Verilog

DAY 41 CHALLENGE:

1. Differentiate between Task and Function.

Task

Tasks are subroutines that can be called anytime in the module they are defined, but it is possible to define them in a different file and include the file in the module. Some characteristics of tasks are:

- Task can include time delays, functions can't.
- Tasks can have any number of inputs and outputs, functions can have only one output.
- If a variable is declared within the task it is local to the task and can't be used outside the task.
- Tasks can drive global variables if no local variables are declared.
- Tasks are called with statements and cannot be used in a expression, functions can be used in a expression.

```
module  task_calling (adc_a, adc_b, adc_a_conv, adc_b_conv);

input [7:0] adc_a, adc_b;

output [7:0] adc_a_conv, adc_b_conv;

reg [7:0] adc_a_conv, adc_b_conv;

task convert;

input [7:0] adc_in;

output [7:0] out;

begin

    out = (9/5) *( adc_in + 32)
```

```
end

endtask

always @ (adc_a)

begin

  convert (adc_a, adc_a_conv);

end

always @ (adc_b)

begin

  convert (adc_b, adc_b_conv);

end

endmodule
```

Functions

Functions are like tasks, with some differences. Functions cannot drive more than one output and cannot have time delays. Some differences are:

- Functions cannot include timing delays, like posedge, negedge, simulation delay, which means that functions implement combitional logic.
- Functions can have any number of inputs but only one output.
- The variables declared within the function are local to that function.
- The order of declaration within the function are considered and have to be the same as the caller.
- Functions can use and modify global variables, when no local variables are used.
- Functions can call other functions, but cannot call tasks. Function example:

```
module  function_calling(a, b, c, d, e, f);

input a, b, c, d, e ;

output f;

wire f;

function  myfunction;
```

```
input a, b, c, d;

begin

    myfunction = ((a+b) + (c-d));

end

endfunction

assign f =  (myfunction (a,b,c,d)) ? e :0;

endmodule
```

2. What is PLI? What are its applications?

PLI (Programming Language Interface) is an interface used in hardware description languages like Verilog and VHDL to enable interaction between the hardware simulator and external programs. It allows the extension of the simulator's capabilities by integrating C/C++ code, providing custom functionalities that are difficult to implement directly in Verilog or VHDL.

Applications of PLI:

1. Custom Testbenches: Create advanced testbenches using external code.
2. File I/O Operations: Read and write data files for simulation purposes.
3. Interfacing with External Tools: Connect simulations with waveform viewers, analyzers, or custom interfaces.
4. Custom System Functions: Define new functions and tasks beyond the language's built-in capabilities.
5. Integration with Software Models: Combine software algorithms with hardware simulations.
6. Enhanced Debugging: Develop custom debugging tools.

3. What is Virtual and Pure virtual function in Verilog?

In Verilog, the concepts of virtual and pure virtual functions are associated with Object-Oriented Programming (OOP) principles, similar to those in languages like C++ and SystemVerilog. However, standard Verilog does not support these features natively. Instead, these concepts are relevant in SystemVerilog, an extension of Verilog that supports OOP.

- Virtual Function: A virtual function is a method in a base class that can be overridden in derived classes. It allows for dynamic binding, where the function to be executed is determined at runtime based on the object type.
- Pure Virtual Function: A pure virtual function is a method declared in a base class without an implementation, requiring derived classes to provide their implementations. In SystemVerilog, this is declared using the pure virtual keyword.

4. What is DPI? What are its applications?

DPI (Direct Programming Interface) is a standard in SystemVerilog that allows calling C/C++ functions directly from SystemVerilog and vice versa. DPI provides a more straightforward and efficient way to interface between SystemVerilog and C/C++ compared to PLI.

Applications of DPI:

1. Integration with C/C++ Code: Easily call C/C++ functions for complex computations or to reuse existing code.
2. Performance Enhancement: Implement performance-critical sections in C/C++ for faster simulation.
3. Interfacing with External Models: Connect with external software models or algorithms.
4. Access to System Resources: Use system calls and libraries that are not available in SystemVerilog.

5. Explain following system tasks in Verilog:

I. **$random :** $random generates pseudo-random numbers in Verilog. It's commonly used for creating random test vectors or initializing variables in simulation.

II. **$readmemb :** $readmemb reads binary data from a file and stores it in a memory array. It's useful for initializing memory arrays with binary data during simulation.

III. **$readmemh :** $readmemh reads hexadecimal data from a file into a memory array. Similar to $readmemb, but it interprets the data in hexadecimal format.

6. Explain following file support operations in Verilog:

I. **$fopen :** $fopen opens a file for reading or writing and returns a file descriptor used for subsequent file operations.
II. **$fscan :** $fscanf reads formatted data from a file into specified variables. It's similar to the scanf function in C.
III. **Sfdisplay :** $fdisplay writes formatted data to a file, similar to fprintf in C. It can also be used with the standard output.
IV. **$fwrite :** $fwrite writes binary data to a file. It's more low-level compared to $fdisplay and doesn't format the output.
V. **$fclose :** $fclose closes an open file, ensuring that all data is properly written and resources are released.

7. Design & execute the following Verilog Codes:

I. T Flip Flop

```
module T_flip_flop (
    input clk,
    input reset,
    input T,
    output reg Q
);
    always @(posedge clk or posedge reset) begin
        if (reset)
            Q <= 0;
        else if (T)
            Q <= ~Q;
    end
endmodule
```

II. 4-bit shift register

```
module shift_register (
    input clk,
    input reset,
    input shift,
    input [3:0] data_in,
    output reg [3:0] data_out
);
    always @(posedge clk or posedge reset) begin
        if (reset)
            data_out <= 0;
        else if (shift)
            data_out <= {data_out[2:0], data_in};
    end
endmodule
```

III. FSM to detect the overlapping sequence: 01010

```
module sequence_detector (
    input clk,
    input reset,
    input data_in,
    output reg detected
);
    typedef enum logic [2:0] {
        S0, S1, S2, S3, S4
    } state_t;

    state_t state, next_state;

    always @(posedge clk or posedge reset) begin
        if (reset)
            state <= S0;
        else
            state <= next_state;
    end

    always @(*) begin
        next_state = state;
        detected = 0;
        case (state)
            S0: if (data_in) next_state = S1;
            S1: if (!data_in) next_state = S2;
            S2: if (data_in) next_state = S3; else next_state =
S0;
            S3: if (!data_in) next_state = S4; else next_state
= S1;
            S4: begin
                detected = 1;
                if (data_in) next_state = S1; else next_state =
S2;
            end
        endcase
    end
endmodule
```