1. Are tasks and functions re-entrant, and how are they different from static task and function calls?
   In Verilog, functions are always re-entrant, but tasks are not re-entrant by default unless specified.
   Re-entrant tasks allow multiple calls to the task at the same time.
   Static task/function calls use static variables, meaning they retain their values between calls and across processes, which can lead to unexpected behaviour when multiple processes access the same task.
2. Can a ‘task’ be called within a ‘function’?
   In Verilog, tasks cannot be called within functions. Functions are purely combinational, have no delays, and must return a value. Tasks, however, can have delays and don't need to return a value. Therefore, tasks are typically called from modules or other tasks.
3. What are the restrictions of using tasks? How can I override variables in an automatic task?
   Restrictions
   - Tasks can contain delays (# or @), events, and multiple outputs.
   - Tasks can modify the variables by which they are passed.
   - Tasks cannot return a value like functions.
   - Tasks must be used when you need non-combinational logic (e.g., delays, event control).
     To override variables, you can declare a task as automatic, allowing it to allocate its variables dynamically. You can override variables by passing different arguments to the task.
4. When should you use a task instead of a function in Verilog?
   - You need to model behaviour involving delays or timing.
   - You have multiple outputs.
   - You want to avoid returning a single value like in a function.
   - Use functions for pure combinational logic where you need to return a single value and cannot include delays.
5. How can i call a function like a task, that is, not have a return value assigned to a variable.
   Use functions for pure combinational logic where you need to return a single value and cannot include delays.
6. Define a function to design an 8-function ALU that takes 4-bit inputs a & b & computes a 5-bit output out based on 3-bit input sel as shown below:

sel	out	sel	out
3’b000	a	3’b100	a%1
3’b001	a+b	3’b101	a<<1
3’b010	a-b	3’b110	a>>1
3’b011	a/b	3’b111	a>b

```
function [4:0] alu_function(input [3:0] a, input [3:0] b, input [2:0] sel);
  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
endfunction
```

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.

```
task factorial;
  input [3:0] num;        // 4-bit input number
  output reg [31:0] fact; // 32-bit output factorial
  integer i;
  begin
    fact = 1;
    for (i = 1; i <= num; i = i + 1) begin
      fact = fact * i;
    end
    #11; // Delay of 11-time units
  end
endtask
```