Module: SV for Verification
Section: Modules & Classes **Task:** Passing Data to Methods

Task 1

Passing Data to Methods

➢ **What is the best approach to implement a subprogram (task/function) that does not consume time and why should you use that technique?**
The best approach to implement a routine is to make use of System verilog functions because functions executes immediately and don't consume any simulation time. They are super useful because they can't contain time consuming constructs such as delays, posedge macros or wait statements.

It ensures efficiency, simplicity, and predictable behavior in your design, making it ideal for synchronous operations and combinational logic.

➢ **Differentiate between Verilog and SystemVerilog ways to handle arguments.**
Simplicity: System Verilog allows us to declare arguments with less verbose and in c-style format for ease. For example, following verilog task requires you to declare some arguments twice: once for direction and once for the type.

```
task example;
output [31:0] x;
reg [31:0] x;
Input y;
endtask
```

While System Verilog can achieve the same objective using less verbose as follows:

```
task example;
Output logic [31:0] x;
Input logic y;
endtask
```

You can also take even more shortcuts with declaring routine arguments taking advantage of default types and directions in system verilog. For instance, if we don't specify the type or direction of a variable, it will automatically default to "input logic".

Passing By Reference: System Verilog allows us to pass values by reference. Meaning that no memories can be passed into a verilog routine. When we pass arguments to a routine in verilog, it copies those values to a local variable at start of the routine and output was copied while exiting the routine.

On the other hand, we can specify that an argument is passed by reference using **ref** keyword. It also allows us to pass an array by reference.

Default Value: Verilog does not support default argument values, so every argument must be explicitly provided when calling a task or function. SystemVerilog allows default values for arguments, providing more flexibility and reducing the need for overloaded functions or tasks.

➢ **Write a SystemVerilog code for passing arrays using “ref” and “const”. Submit a code example.**
Please refer to the attached code_example.docx file for this.

➢ **Why should you always make program blocks use automatic storage? What happens if we don’t use automatic storage?**

We can use the SystemVerilog automatic keyword to declare a function as reentrant, as shown in the code snippet below.

```
function automatic int sum (input <arguments>);
  int a = 10;
  int b = 5;
  int sum = a + b;
endfunction
```

When we declare a function as reentrant, the variables and arguments which we declare in the function will be dynamically allocated.

In contrast, normal functions use **static allocation** for internal variables and arguments. This means that all of the memory which is required to perform the processing of the function is allocated only once at the start of the simulation.

As a result of this, our simulation software must execute the function in it's entirety before it can use the function again.

This also means that any memory which is allocated to the function will never be deallocated. As a result of this, any values stored in this memory will maintain their value between calls to the function.

In contrast, our simulator **allocates** memory to **automatic functions** whenever we call the function. Once our function has finished executing, this memory will be deallocated.

As a result of this, our simulation software can execute multiple instances of an automatic function.

We can use the automatic keyword to write recursive functions in SystemVerilog. This means we can create functions which call themselves to perform a calculation.