1. What is fork-join, and what are its types?

The fork-join construct in Verilog/SystemVerilog is used to create parallel execution of multiple blocks of code (processes). It forks (starts) multiple processes concurrently and then waits for all of them to complete before continuing.

Types

* fork-join:
  + All processes run in parallel, and the join waits for all of them to complete before proceeding.

Example:

fork

process1;

process2;

join

* fork-join\_any:
  + Waits for any one of the forked processes to complete. As soon as one process finishes, the join\_any block allows the code to continue.

Example:

fork

process1;

process2;

join\_any // Execution will proceed after the first process finishes.

* fork-join\_none:
  + This waits for none of the forked processes to complete. The block does not wait for any process to finish and continues immediately.

Example:

fork

process1;

process2;

join\_none // Execution proceeds immediately without waiting.

1. Difference between fork-join, fork-join\_any and fork\_join\_none.

* fork-join:Waits for all processes to complete.
* fork-join\_any: Waits for any one of the forked processes to complete.
* fork-join\_none: Does not wait for any process to complete.

1. What is the main limitation of fork-join in Verilog and how is this overcome in System Verilog?

* In Verilog, tasks and functions run in parallel when forked, but fork-join doesn't allow for proper handling of time-based events or synchronization. Specifically, Verilog lacks the ability to control the scheduling of the forked processes, and there can be issues with processes being prematurely terminated or with race conditions.
* SystemVerilog introduces the join\_any and join\_none options, which allow more flexible synchronization, as well as better control over parallel process termination and timing.
* Improved synchronization: SystemVerilog allows better management of process synchronization and avoids issues with race conditions and premature termination.

1. What are the feature added in System Verilog for function and task?

Function:

* Functions can be synthesized and executed in a more optimized way.
* Return values: Functions can return a single value (which is also possible in Verilog).
* Time delay: Functions in SystemVerilog can no longer contain delays (#), unlike in Verilog where delays could be used.
* Inline functions: SystemVerilog supports inline functions, making the code more efficient.
* Variable number of arguments: Functions can accept variable numbers of arguments (using ...).
* Automatic variables: Functions can have local variables that are automatically allocated during execution.

Task:

* Tasks can have multiple input and output arguments.
* Tasks can execute delays, event controls, and fork-join constructs.
* Tasks are more flexible and can be non-blocking, whereas functions are always blocking

1. How to kill a process in fork/join?

In SystemVerilog, you can kill a process in a fork-join block using the $stop or $finish system tasks, or by controlling the event inside the process. However, SystemVerilog does not provide a direct function to "kill" a running process, but you can use the disable command for named blocks.

1. Difference between while and do while.

while:

* The condition is checked before the loop is executed.
* If the condition is false initially, the loop does not execute at all.

do-while:

* The condition is checked after the loop executes, meaning the loop executes at least once.

1. Why function has 0 simulation time?

A function in SystemVerilog is considered to have 0 simulation time because it is expected to be purely combinational. Functions execute without causing any simulation time delay, meaning they don't invoke events, delays, or time-consuming operations like tasks do.

1. Illustrate how the errors of passing arguments to a function in incorrect order is eliminated in System Verilog.

In SystemVerilog, to eliminate errors from passing arguments in the wrong order, you can use named arguments when calling functions. This helps in ensuring that the arguments are correctly assigned, even if their order is incorrect.

Example:

function int add\_numbers(input int a, input int b);

return a + b;

endfunction

initial begin

int result;

result = add\_numbers(b=5, a=3); // Named arguments prevent order issues

$display("Result: %0d", result);

end

1. How does program block avoid the race condition?

The program block in SystemVerilog is used primarily for testbenches and synchronizes with simulation time. It avoids race conditions by ensuring that:

* All tasks and functions inside the program block are executed sequentially.
* No simulation-time events (like #, @) or parallel execution are allowed within a program block, which prevents the possibility of race conditions.
* Additionally, program blocks are executed in a separate simulation context and don't interact with the design logic, thus preventing conflicts.

1. Difference between always\_comb and always@(\*)?

The key difference between always\_comb and always @(\*) is that always\_comb is safer for combinational logic because it prevents latch inference, whereas always@(\*) may infer latches if not carefully written.

1. How we can have #delay which is independent of time scale in System Verilog?

To specify a delay independent of time scale, you can use the timeunit directive or scale the delay using time units explicitly in your code (like ns, ps, etc.), while making sure the delay doesn't depend on the simulation's global time scale.