The American University in Cairo

CENTER OF NANOELECTRONICS & DEVICES

CND212: Digital Testing and Verification

SystemVerilog Routines

Subroutines (task & function)

- Tasks and functions provide the ability to execute common procedures from several different places in a code.
- They also provide a means of breaking up large procedures into smaller ones to make them easier to read and debug.
- Tasks and functions are collectively referred to as subroutines

Subroutines (task & function)

- Functions
  - Cannot block
    - Can’t have time-controlling statements such as @(posedge clk)
    - The statements in the body of a function execute in one simulation time unit
  - A non-void function must return a single value
    - SystemVerilog improves this by adding (void function) which does not return a value
  - Cannot enable a task
- Tasks
  - Can block
    - May contain time-controlling statements
  - Does not return a value
  - Can enable other tasks and/or functions

Subroutines (task & function) - example

```
module non_void_func;

function [15:0] myfunc1 (input [7:0] x,y);
myfunc1 = x * y - 1; // return value assigned to function name
endfunction

function [15:0] myfunc2 (input [7:0] x,y);
return x * y - 1; //return value is specified using return statement
endfunction

initial begin
    int m = 10;
    int result1,result2;
  for (int n = 1; n <= 8; n++) begin
        result1 = myfunc1(m,n);
        result2 = myfunc1(m,n);
        $display("%0d result1 =%0d", n, result1);
        $display("%0d result2 =%0d", n, result2);
    end
end
endmodule
```

```
module void_function_example;
 int x;
 //void function to display current simulation time
 function void current_time;
   $display("\tCurrent simulation time is %0d",$time);
 endfunction

 initial begin
   #10;
   current_time();
   #20;
   current_time();
 end
endmodule
```

Subroutines (task & function) - example

```
module traffic_lights;
    logic clock, red, amber, green;
    parameter on = 1, off = 0, red_tics = 350,amber_tics = 30, green_tics = 200;
// initialize colors
    initial red = off;
    initial amber = off;
    initial green = off;
    always begin // sequence to control the lights
        red = on; // turn red light on
        light(red, red_tics); // and wait.
        green = on; // turn green light on
        light(green, green_tics); // and wait.
        amber = on; // turn amber light on
        light(amber, amber_tics); // and wait.
    end
// task to wait for 'tics' positive edge clocks
// before turning 'color' light off
    task light (output color, input [31:0] tics);
        repeat (tics) @ (posedge clock);
        color = off; // turn light off.
    endtask: light
    always begin // waveform for the clock
        #100 clock = 0;
        #100 clock = 1;
    end
endmodule: traffic_lights
```

Task/function Memory Usage

- Static
  - Tasks/functions are static by default except if they are declared inside a class scope
  - All variables of a static task/function are static by default
  - Static variables retain their value between invocations
  - There is a single static variable corresponding to each declared local variable in a module instance, regardless of the number of concurrent activations of the task/function.
  - Static tasks/functions in different instances of a module have separate storage from each other
- Automatic
  - Allocate unique, separate storage for each task/function call
  - All variables of an automatic task/function are automatic by default
  - Automatic variables do not retain their values between invocations
  - Automatic variables are replicated on each concurrent task/function invocation

Task/function Memory Usage - example

```
module static_automatic_function_example;

  function static increment_static();
    static int count_A;
    automatic int count_B;
    int count_C;

    count_A++;
    count_B++;
    count_C++;
    $display("Static: count_A = %0d, count_B = %0d, count_C = %0d", count_A, count_B, count_C);
  endfunction

  function automatic increment_automatic();
    static int count_A;
    automatic int count_B;
    int count_C;

    count_A++;
    count_B++;
    count_C++;
    $display("Automatic: count_A = %0d, count_B = %0d, count_C = %0d", count_A, count_B, count_C);
  endfunction

  initial begin
    $display("Calling static functions");
    increment_static();
    increment_static();
    increment_static();
    $display("\nCalling automatic functions");
    increment_automatic();
    increment_automatic();
    increment_automatic();
  end
endmodule
```

```
Calling static functions
Static: countA = 1 , countB =1 , countC =1
Static: countA = 2 , countB =1 , countC =2
Static: countA = 3 , countB =1 , countC =3
Calling automatic functions
Automatic: countA = 1 , countB =1 , countC =1
Automatic: countA = 2 , countB =1 , countC =1
Automatic: countA = 3 , countB =1 , countC =1
        V C S   S i m u l a t i o n   R e p o r t
```

Functions vs Tasks - Summary

Function	Task
It cannot contain simulation delay, so it executes in the same time unit.	Can contain a simulation time delay and include time-controlling statements
Can return a single value unless it is a void function	Does not return a value but can achieve the same effect using output arguments
Cannot call another task	Can call another function or task

SystemVerilog Threading

fork – join (Parallel blocks)

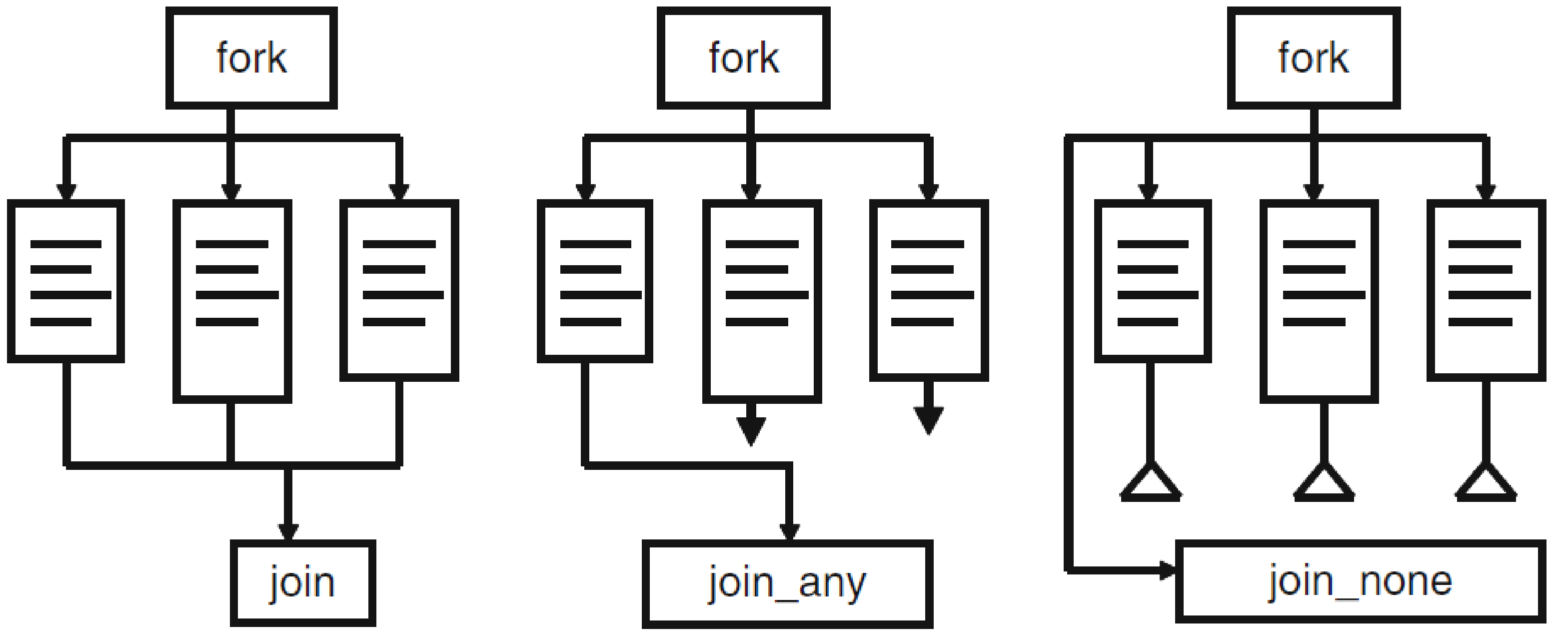
- There are two types of blocks in SystemVerilog
  - Sequential blocks
    - begin – end block
  - Parallel blocks
    - fork – join block
- The parallel block is delimited by the keywords fork and join, join_any, or join_none.
- The procedural statements in a parallel block is executed concurrently (in parallel)

fork – join (Parallel blocks)

- SystemVerilog provides three choices for specifying when the parent (forking) process resumes execution

Option	Description
`join`	The parent process blocks until all the processes spawned by this fork complete.
`join_any`	The parent process blocks until any one of the processes spawned by this fork completes.
`join_none`	The parent process continues to execute concurrently with all the processes spawned by the fork. The spawned processes do not start executing until the parent thread executes a blocking statement or terminates.

fork – join (Parallel blocks)



fork – join (example1)

Example1: fork - join

- fork block will be blocked until the completion of process-1 and Process-2.
- Both process-1 and Process-2 will start at the same simulation time
  - Process-1 will finish at 5ns
  - Process-2 will finish at 20ns.
- fork-join will be unblocked at 20ns.

```
module fork_join_example;
  initial begin
    fork
      //--------------------
      //Process-1
      //--------------------
      begin
        $display($time,"\tProcess-1 Started");
        #5;
        $display($time,"\tProcess-1 Finished");
      end

      //--------------------
      //Process-2
      //--------------------
      begin
        $display($time,"\tProcess-2 Started");
        #20;
        $display($time,"\tProcess-2 Finished");
      end
    join
    $display($time,"\tOutside Fork-Join");
    $finish;
  end
endmodule
```

Sequential blocks

fork – join_any (example2)

Example2: fork – join_any

- The fork block will be blocked until the completion of any of the processes, Process-1 or Process-2.
- Both Process-1 and Process-2 will sta at the same simulation time,
  - Process-1 will finish at 5ns
  - Process-2 will finish at 20ns
- fork-join_any will be unblocked at 5ns.

```
module fork_join_any_example;
  initial begin
    fork
      //Process-1
      begin
        $display($time,"\tProcess-1 Started");
        #5;
        $display($time,"\tProcess-1 Finished");
      end

      //Process-2
      begin
        $display($time,"\tProcess-2 Started");
        #20;
        $display($time,"\tProcess-2 Finished");
      end
    join_any

    $display($time,"\tOutside Fork-Join");
  end
endmodule
```

```
         0     Process 1 Started
         0     Process 2 Started
         5     Process 1 Finished
         5     Outside fork join-none
        20     Process 2 Finished
V C S   S i m u l a t i o n   R e p o r t
```

fork – join_none (example3)

Example3: fork – join_none

- The fork will start Process-1 and Process-2 at the same time, and it will come out of the block.
- Process-1 and Process-2 will be executed until they are completed.

```
module fork_join_none;
  initial begin
    fork
      //Process-1
      begin
        $display($time,"\tProcess-1 Started");
        #5;
        $display($time,"\tProcess-1 Finished");
      end
      //Process-2
      begin
        $display($time,"\tProcess-2 Startedt");
        #20;
        $display($time,"\tProcess-2 Finished");
      end
    join_none

    $display($time,"\tOutside Fork-Join_none");
  end
endmodule
```

```
             0    Outside fork join-none
             0    Process 1 Started
             0    Process 2 Started
             5    Process 1 Finished
            20    Process 2 Finished
     V C S   S i m u l a t i o n   R e p o r t
```

fork – join (Parallel blocks)

Summary

- For sequential blocks, the start time is when the first statement is executed, and the finish time is when the last statement has been executed.
- For parallel blocks, the start time is the same for all the statements, and the finish time is controlled by the join construct used
- Sequential and parallel blocks can be embedded within each other, allowing complex control structures to be expressed easily and with a high degree of structure.

SystemVerilog Interprocess Synchronization & Communication

Overview

- Verilog provides basic synchronization mechanisms that are adequate at the hardware level (i.e. @)
- This type of control is adequate for static objects but falls short in a dynamic and highly reactive testbench.
- Hence, SystemVerilog adds the following communication mechanisms:
  - Semaphores
    - built-in class, which can be used for synchronization and mutual exclusion to shared resources
  - Mailboxes
    - built-in class, which can be used as a communication channel between processes

Semaphore

```
semaphore identifier_name;
```

- Conceptually, a semaphore is a bucket.
- When a semaphore is allocated, a bucket containing a fixed number of keys is created
- Processes using semaphores must first get a key from the bucket before executing.
- All other processes must wait until a sufficient number of keys is returned to the bucket.

Semaphore Methods

Method name	Description
new()	To create a semaphore with a specified number of keys
get()	To obtain or get a specified number of keys
put()	To put or return the number of keys
try_get()	Try to obtain or get a specified number of keys without blocking the execution

Semaphore – Example 1

```
module with_semaphore_example();
  semaphore sem = new(1);

  task write_mem();
    sem.get();
    $display("Before writing into memory");
    #5ns  // Assume 5ns is required to write into mem
    $display("Write completed into memory");
    sem.put();
  endtask

  task read_mem();
    sem.get();
    $display("Before reading from memory");
    #4ns  // Assume 4ns is required to read from mem
    $display("Read completed from memory");
    sem.put();
  endtask

  initial begin
    fork
      write_mem();
      read_mem();
    join
  end
endmodule
```

```
Before writing into memory
Write completed into memory
Before reading from memory
Read completed from memory
          V C S   S i m u l a t i o n   R e p o r t
```

```
module without_semaphore_example();
  task write_mem();
    $display("Before writing into memory");
    #5ns;  // Assume 5ns is required to write into mem
    $display("Write completed into memory");
  endtask

  task read_mem();
    $display("Before reading from memory");
    #4ns;  // Assume 4ns is required to read from mem
    $display("Read completed from memory");
  endtask

  initial begin
    fork
      write_mem();
      read_mem();
    join
  end
endmodule
```

```
Before writing into memory
Before reading from memory
Read completed from memory
Write completed into memory
          V C S   S i m u l a t i o n   R e p o r t
```

Semaphore – Example 2

Get single key per process

```
module semaphoreexp();
  semaphore key = new(2);
  //no.of keys identify no.of users that can use the semaphore at the same time
    initial begin
        fork
                begin
                    $display("Process A is trying to get the key at %0t", $time);
                    key.get();
                    $display("Process A got the key at %0t", $time);
                    #10 //work
                    key.put();
                    $display("Process A returned back the key at %0t", $time);
                end

                begin
                    $display("Process B is trying to get the key at %0t", $time);
                    key.get();
                    $display("Process B got the key at %0t", $time);
                    #10 //work
                    key.put();
                    $display("Process B returned back the key at %0t", $time);
                end
        join
    end
endmodule
```

```
Process A is trying to get the key at 0
Process A got the key at 0
Process B is trying to get the key at 0
Process B got the key at 0
Process A returned back the key at 10
Process B returned back the key at 10
           V C S   S i m u l a t i o n   R e p o r t
```

Semaphore – Example 3

Get multiple keys per process

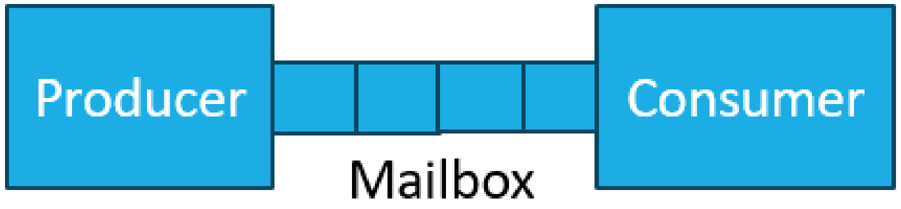
```
module semaphoreexp();
  semaphore key = new(2);
    initial begin
        fork
            begin
                $display("Process A is trying to get the key at %0t", $time);
                key.get(2);
                $display("Process A got the key at %0t", $time);
                #10 //work
                key.put();
                $display("Process A returned back the key at %0t", $time);
            end

            begin
                $display("Process B is trying to get the key at %0t", $time);
                key.get();
                $display("Process B got the key at %0t", $time);
                #10 //work
                key.put();
                $display("Process B returned back the key at %0t", $time);
            end
        join
    end
endmodule
```

```
Process A is trying to get the key at 0
Process A got the key at 0
Process B is trying to get the key at 0
Process A returned back the key at 10
Process B got the key at 10
Process B returned back the key at 20
            V C S   S i m u l a t i o n   R e p o r t
```

Mailbox

mailbox #(type) identifier_name;

- A mailbox is a communication mechanism that allows messages to be exchanged between processes.
- Data can be sent to a mailbox by one process and retrieved by another.
- Mailbox behaves as first-in-first-out (FIFO), with a source and sink



Mailbox Methods

Method name	Description
function new(int bound = 0)	Returns mailbox handle. An argument represents bounded mailbox size otherwise, it is an unbounded mailbox
task put(<data>)	Blocking method that stores data in the mailbox.
function int try_put(<data>)	The non-blocking method that stores data in the mailbox if it is not full and returns 1 else 0.
task get(ref <data>)	Blocking method to retrieve data from the mailbox
function int try_get(ref <data>)	The non-blocking method which returns data if a mailbox is non-empty else returns 0.
task peek(ref <data>)	Copies data from the mailbox without removing it from a mailbox
function int try_peek(ref <data>)	Tries to copy data from the mailbox without removing it from a mailbox
function int num()	Returns number of entries in the mailbox

Mailbox – bounded/unbounded

- Mailbox is unbounded by default.
- It can be bounded by passing the required size to its new function.
- A bounded mailbox becomes full when it contains the bounded number of messages.
  - When the source thread tries to put a value into a sized mailbox that is full, that thread blocks until the value is removed.
  - Likewise, if a sink threads tries to remove a value from an empty mailbox, that thread blocks until a value is put into the mailbox
- Unbounded mailboxes never suspend a thread in a send operation.

Mailbox – example

```
module mailbox_example();
  mailbox mb = new();

  task process_A();
    int value;
    repeat(10) begin
      value = $urandom_range(1, 50);
      mb.put(value);
      $display("Put data = %0d", value);
    end
    $display("----------------------");
  endtask

  task process_B();
    int value;
    repeat(10) begin
      mb.get(value);
      $display("Retrieved data = %0d", value);
    end
  endtask

  initial begin
    fork
      process_A();
      process_B();
    join
  end
endmodule
```

```
Put data = 40
Put data = 49
Put data = 20
Put data = 48
Put data = 7
Put data = 3
Put data = 20
Put data = 26
Put data = 19
Put data = 17
-------------------
Retrieved data = 40
Retrieved data = 49
Retrieved data = 20
Retrieved data = 48
Retrieved data = 7
Retrieved data = 3
Retrieved data = 20
Retrieved data = 26
Retrieved data = 19
Retrieved data = 17
           V C S   S i m u l a t i o n   R e p o r t
```

Lab 3: (~ 60 min)

Lab Task: Producer-Consumer Model

- In this lab, a producer and consumer model can be designed using the pre-defined packet data type, SystemVerilog mailbox, and tasks as follows:
  - Create a packet that has:
    - ID
    - Sent_time
    - packet_type
    - data
  - Types of packets are:
    - Message
    - Command
    - Control
  - Create a task for a producer that writes in the mailbox every 10ns.
  - Create a task for a consumer that reads from the mailbox every 5ns.
  - Use fork-join to synchronize between them.
  - Randomize multiple packets and send them from the producer in a fixed time interval.

Assignment: (~ 60 min)

Assignment: Producer-Consumer Model with priorities

- For the lab assignment, a producer and consumer model can be designed using the pre-defined packet data type, SystemVerilog mailbox, and tasks as follows:
  - ✓ Create a packet that has: ID, Sent_time, packet_type, priority and data.
  - ✓ Types of packets are Message, Command, and Control.
  - – There are three levels of priority:
    - High
    - Medium
    - Low
  - – Create a task for a producer that writes in the mailbox every 5ns
  - – Create a task for a consumer that reads from the mailbox every 20ns **but reads the packets with higher priority first.**
  - – Use fork-join to synchronize between them.
  - – Randomize multiple packets and send them from the producer in a fixed time interval.