CND212: Digital Testing and Verification

Packages

SystemVerilog - Packages

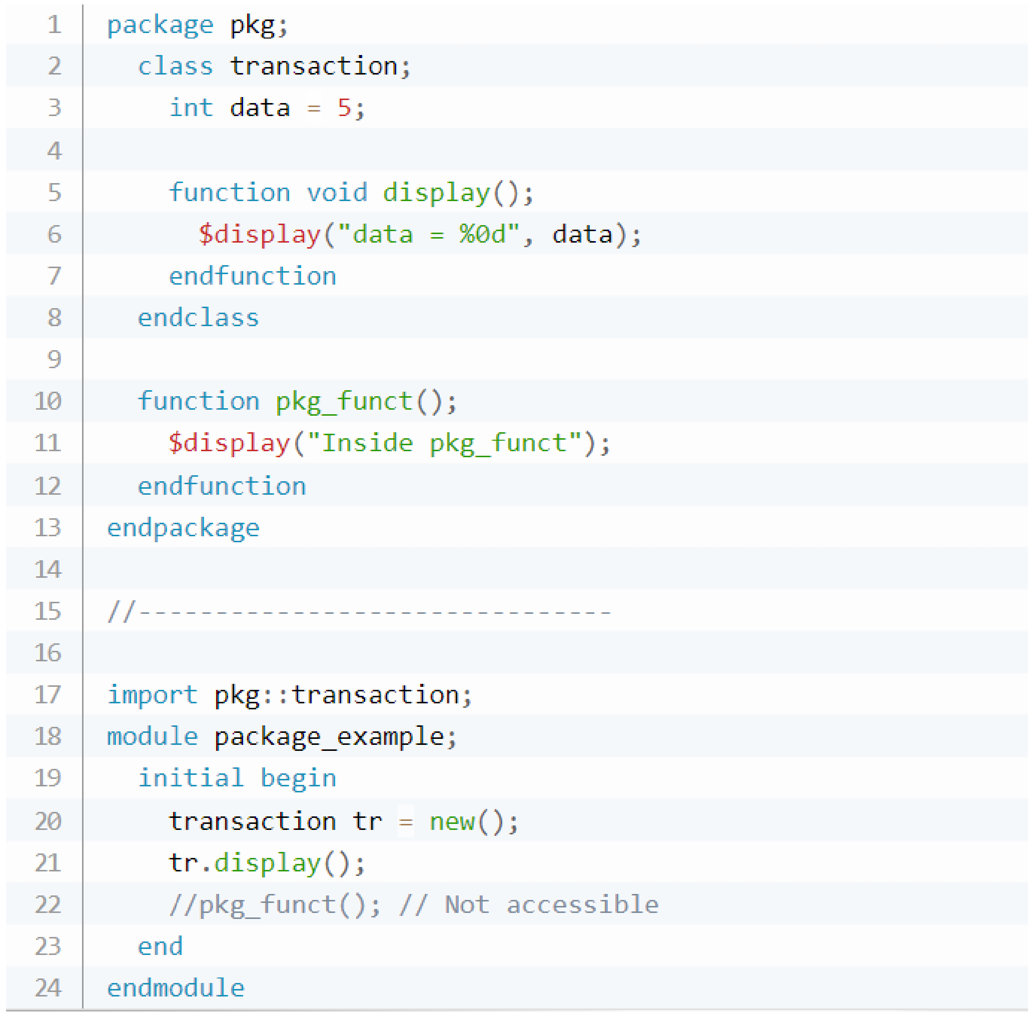
- Packages provide a systematic mechanism for sharing parameters, data, function, tasks, types, property to other interfaces, programs, or modules.
- A package can be made accessible within the interface, program, module, and other packages using an import keyword followed by scope resolution operator:: and what has to be imported. An import mechanism provides controlled access based on what is imported
- Items in the package cannot have hierarchical references.
- They are explicitly named scopes appearing at the same level as top-level modules or primitives.

SystemVerilog – Packages – Example1

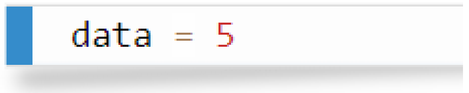
```
package pkg;
  class transaction;
    int data = 5;

    function void display();
      $display("data = %0d", data);
    endfunction
  endclass

  function pkg_funct();
    $display("Inside pkg_funct");
  endfunction
endpackage

//------------------------------

import pkg::*;
module package_example;
  initial begin
    transaction tr = new();
    tr.display();
    pkg_funct();
  end
endmodule
```

```
data = 5
Inside pkg_funct
```

SystemVerilog – Packages – Example 2

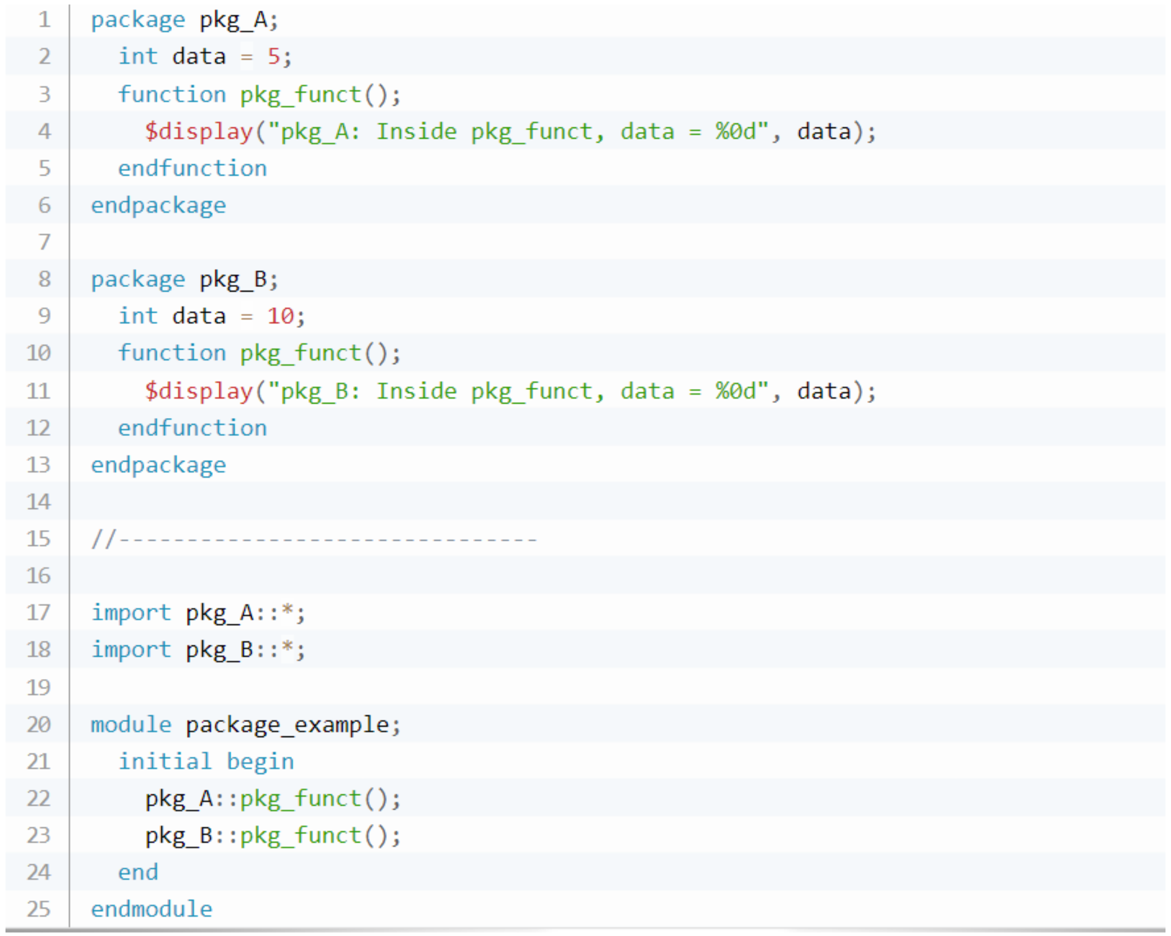


```systemverilog
package pkg;
  class transaction;
    int data = 5;

    function void display();
      $display("data = %0d", data);
    endfunction
  endclass

  function pkg_funct();
    $display("Inside pkg_funct");
  endfunction
endpackage

//------------------------------

import pkg::transaction;
module package_example;
  initial begin
    transaction tr = new();
    tr.display();
    //pkg_funct(); // Not accessible
  end
endmodule
```

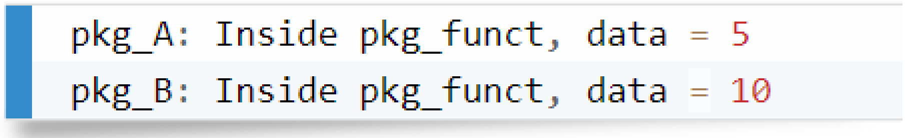


```
data = 5
```

SystemVerilog – Packages – Example 3



```
package pkg_A;
  int data = 5;
  function pkg_funct();
    $display("pkg_A: Inside pkg_funct, data = %0d", data);
  endfunction
endpackage

package pkg_B;
  int data = 10;
  function pkg_funct();
    $display("pkg_B: Inside pkg_funct, data = %0d", data);
  endfunction
endpackage

//-------------------------------

import pkg_A::*;
import pkg_B::*;

module package_example;
  initial begin
    pkg_A::pkg_funct();
    pkg_B::pkg_funct();
  end
endmodule
```

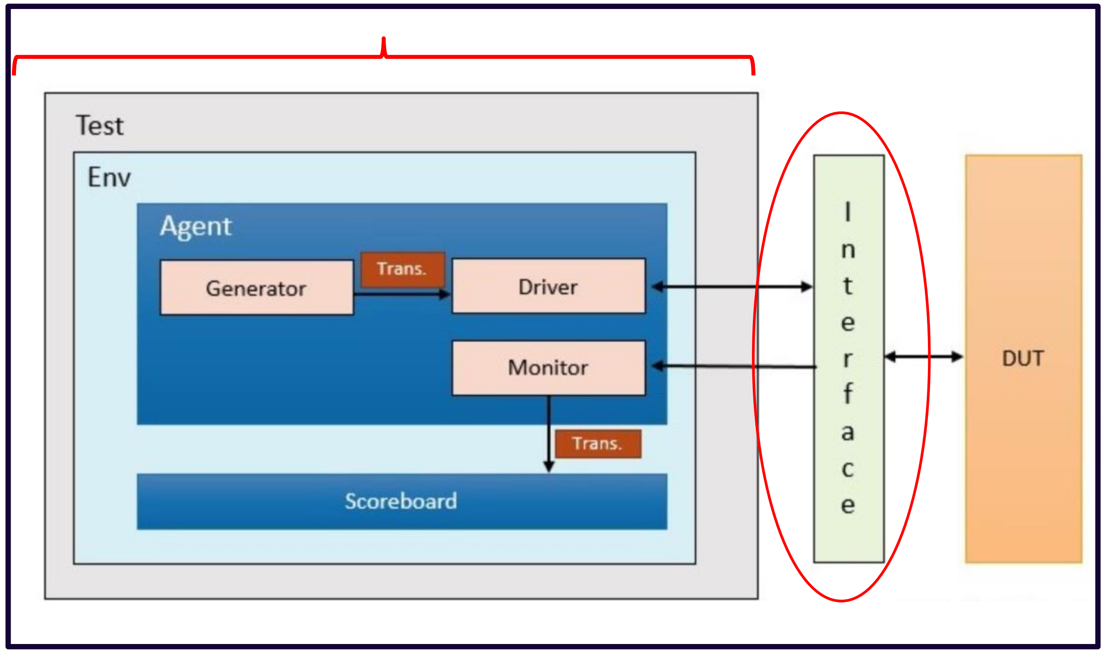


```
pkg_A: Inside pkg_funct, data = 5
pkg_B: Inside pkg_funct, data = 10
```

SystemVerilog – Key features

SystemVerilog – Key Features

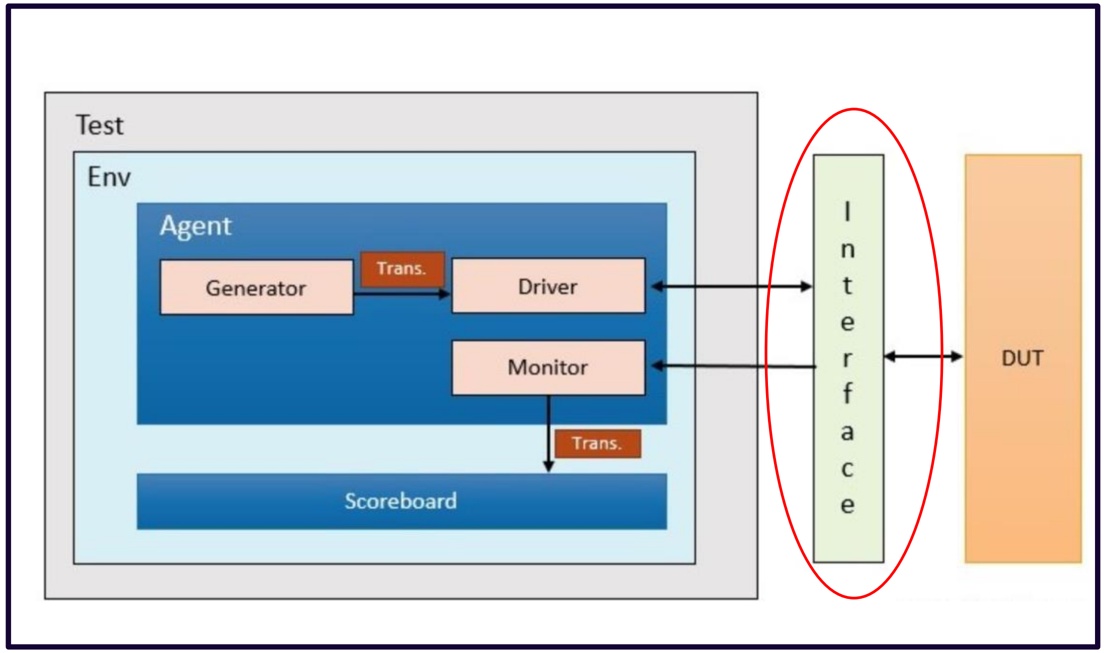
- System Verilog introduces two new design units
  - – The program block
  - – The interface construct



SystemVerilog – Key features
1) The Interface Construct

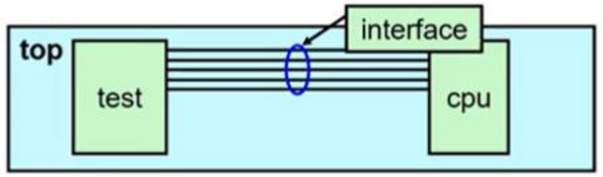
The interface Construct

- Designs have become complicated, and communication between blocks needs to be separated into entities
- An interface is a mechanism to connect the testbench to the DUT
- It is an intelligent bundle of wires that can be passed just like a port in a port list



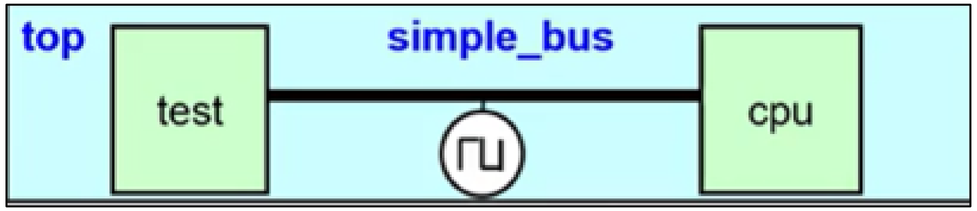
The interface Construct

- The interface encapsulates connectivity/communication between the DUT and the testbench:
  - Connectivity (signals)
  - Directional information (modports)
  - Timing (clocking blocks)
  - Optionally: Functionality (routines, assertions)



- Solves multiple problems with traditional connections
  - Port lists for the connections are compact
  - Easy to add new connections
  - Pass DUT connections throughout the testbench

The interface Construct – Example



```
program automatic test (simple_bus sb);

//....

endprogram
```

```
module cpu(simple_bus sb);

//...

endmodule
```

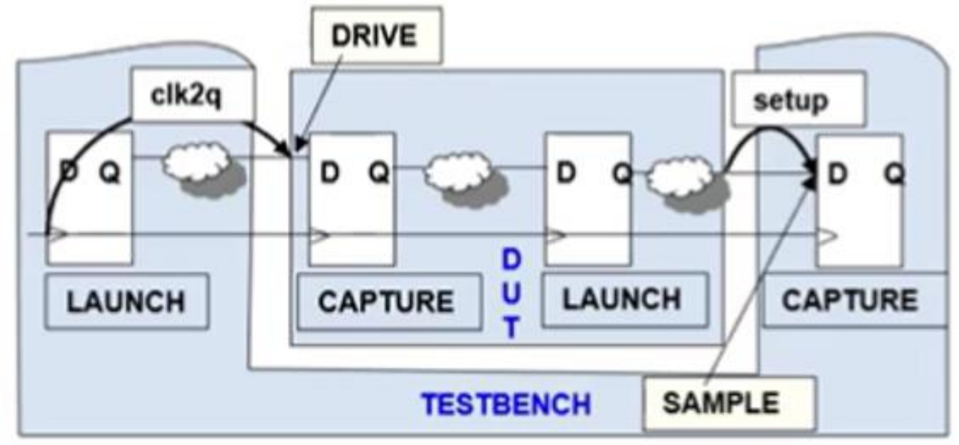
```
interface simple_bus(input bit clk);
    logic req, gnt;
    logic [7:0] addr;
    wire [7:0] data;
    logic [1:0] mode;
    logic start, rdy;
endinterface
```

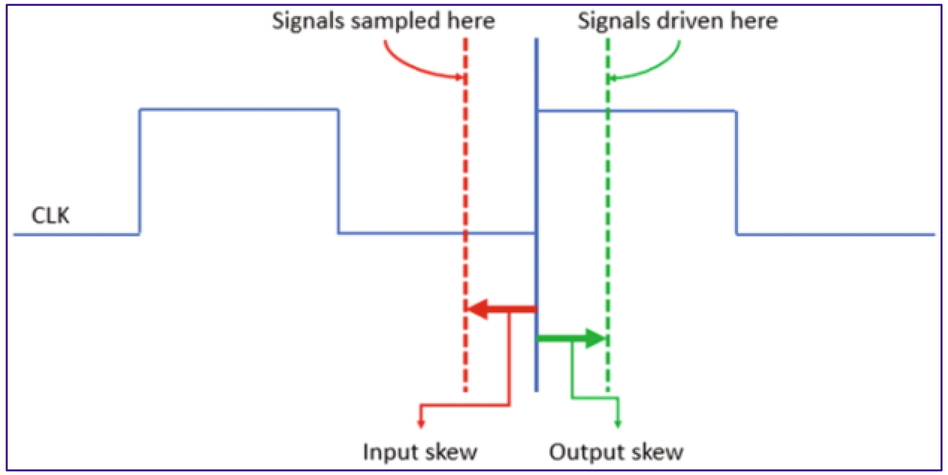
```
module top;
    logic clk = 0;
    always #10 clk = !clk;

    simple_bus sb (clk);
    test t1(sb);
    cpu c1(sb);
endmodule
```

Synchronous Timing: clocking Blocks

- Are just for testbench
  - Emulate the launch and capture of IO of DUT
- Create explicit synchronous timing domains
  - All signals driven or sampled at a clocking event (synchronous)
  - Interaction between testbench and DUT ideally happens only at clock edges
- Specify synchronous signal direction

```
clocking cb @(posedge clk);
    default: input #10ns output #2ns;
    output read, enable, addr;
    input data;
endclocking
```



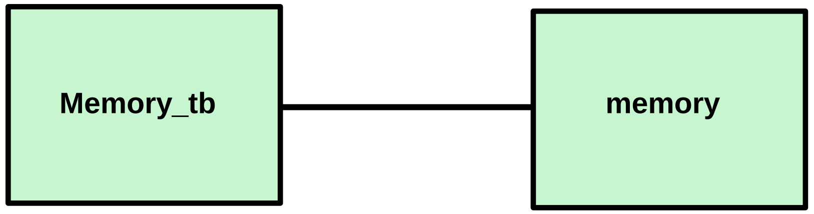


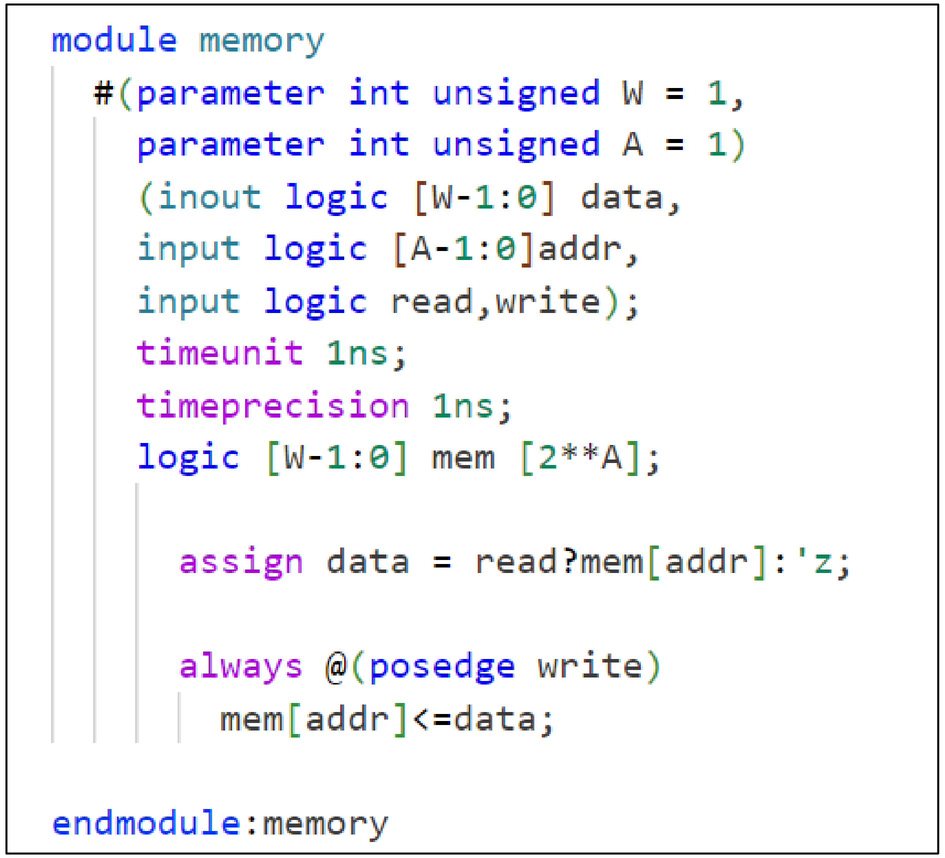
Signal Access Direction: modport

- Enforce signal access and direction using modport
- Modports are used for connecting the interface to the test program
- In the argument list there should be:
  - References to the clocking block (if any)
  - All other asynchronous signals

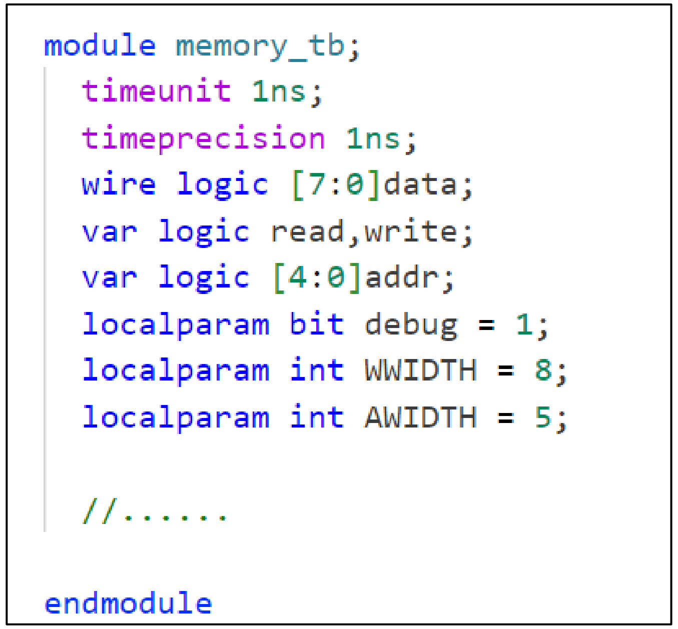
interface + modport – **Example**

Without using interface





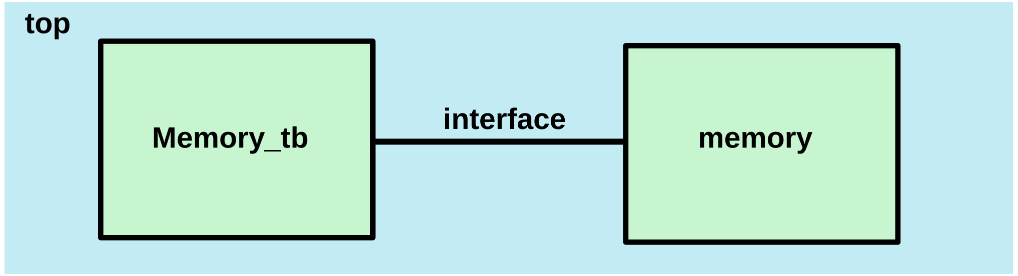
```
module memory
  #(parameter int unsigned W = 1,
    parameter int unsigned A = 1)
    (inout logic [W-1:0] data,
    input logic [A-1:0]addr,
    input logic read,write);
    timeunit 1ns;
    timeprecision 1ns;
    logic [W-1:0] mem [2**A];

      assign data = read?mem[addr]:'z;

      always @(posedge write)
        mem[addr]<=data;

endmodule:memory
```



```
module memory_tb;
  timeunit 1ns;
  timeprecision 1ns;
  wire logic [7:0]data;
  var logic read,write;
  var logic [4:0]addr;
  localparam bit debug = 1;
  localparam int WWIDTH = 8;
  localparam int AWIDTH = 5;

  //......

endmodule
```

interface + modport – **Example**

With interface



```
module memory
  #(parameter int unsigned W = 1,
  parameter int unsigned A = 1)(memory_intf.DUT m1);

   timeunit 1ns;
   timeprecision 1ns;
   logic [W-1:0] mem [2**A];
   assign m1.data = m1.read?mem[m1.addr]:'z;
   always @(posedge m1.write)
      mem[m1.addr]<=m1.data;

endmodule:memory
```

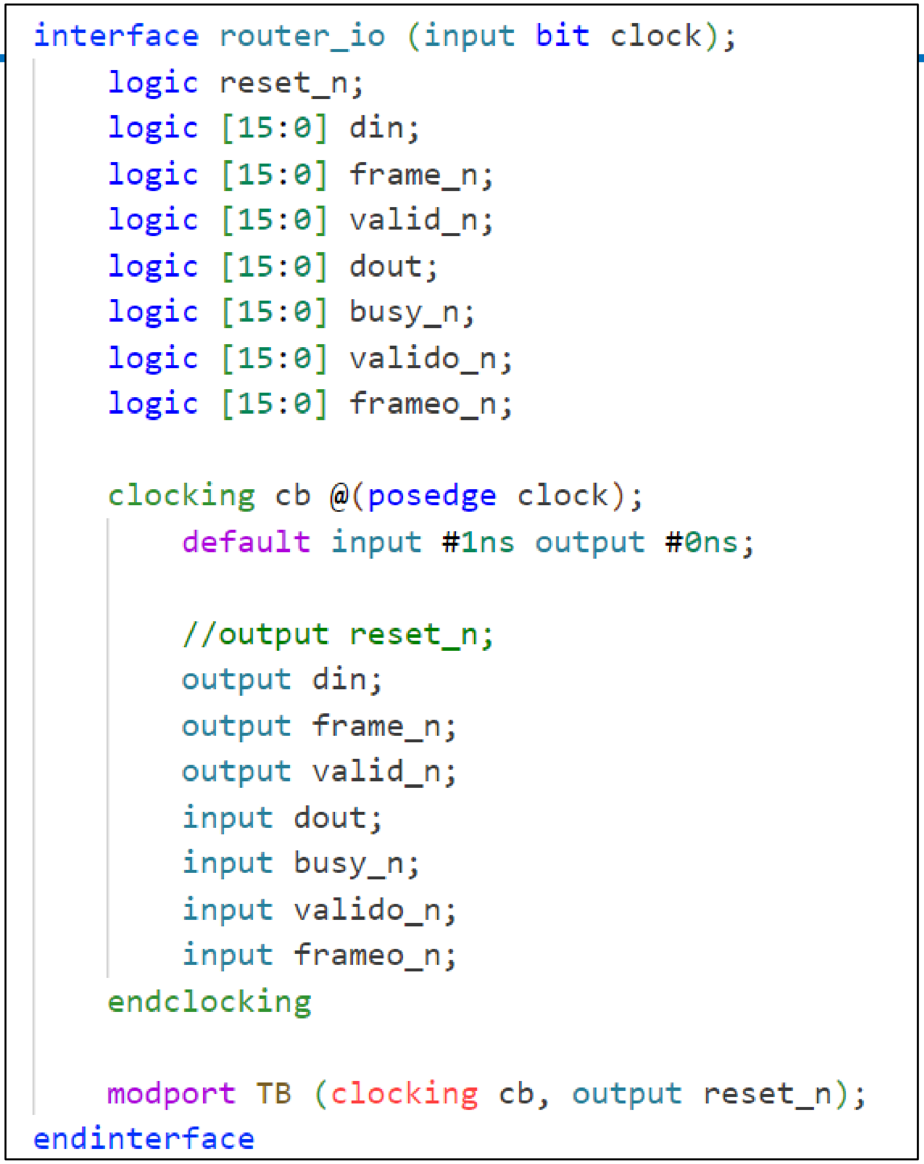
```
program memory_tb(memory_intf.TB m1);
  timeunit 1ns;
  timeprecision 1ns;
  localparam bit debug = 1;
  localparam int WWIDTH = 8;
  localparam int AWIDTH = 5;

  //....

endprogram
```

```
interface memory_intf#(parameter int unsigned W = 1,
  parameter int unsigned A = 1);
  timeunit 1ns;
  timeprecision 1ns;
  wire logic [W-1:0] data;
  logic [A-1:0]addr;
  logic read,write;
  modport DUT(inout data,input addr,input read, input write);
  modport TB(inout data,output addr, output read, output write);
endinterface
```

```
module top;
  timeunit 1ns;
  timeprecision 1ns;
  localparam int unsigned W = 8;
  localparam int unsigned A = 5;
  memory_intf #(W,A)m1 (.*);
  memory #(W,A) m2(m1);
  memory_tb m3(m1);
endmodule
```

A Complete interface

- Signals are declared in the interface body with no direction
- Synchronous signals are defined inside the clocking block with their specified direction
- A Modport is declared for connectivity and has the arguments for the clocking block and all other asynchronous signals



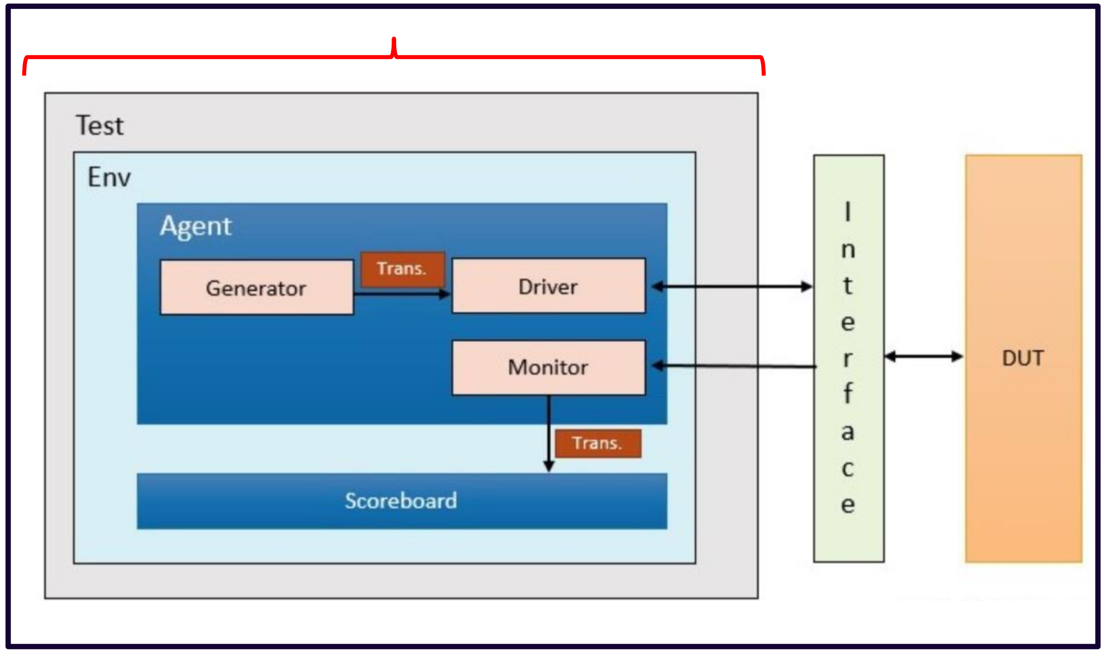
```
interface router_io (input bit clock);
    logic reset_n;
    logic [15:0] din;
    logic [15:0] frame_n;
    logic [15:0] valid_n;
    logic [15:0] dout;
    logic [15:0] busy_n;
    logic [15:0] valido_n;
    logic [15:0] frameo_n;

    clocking cb @(posedge clock);
        default input #1ns output #0ns;

        //output reset_n;
        output din;
        output frame_n;
        output valid_n;
        input dout;
        input busy_n;
        input valido_n;
        input frameo_n;
    endclocking

    modport TB (clocking cb, output reset_n);
endinterface
```

SystemVerilog – Key features
2) Program Block

The program Block

- The program block is where the testbench code is developed
- It is an entry point to test execution
- It facilitates a race-free interaction between the design and the testbench

```
program automatic test (simple_bus sb);

//testbench code in initial block
    initial
        run();
    task run();
    //..
    endtask: run

endprogram
```

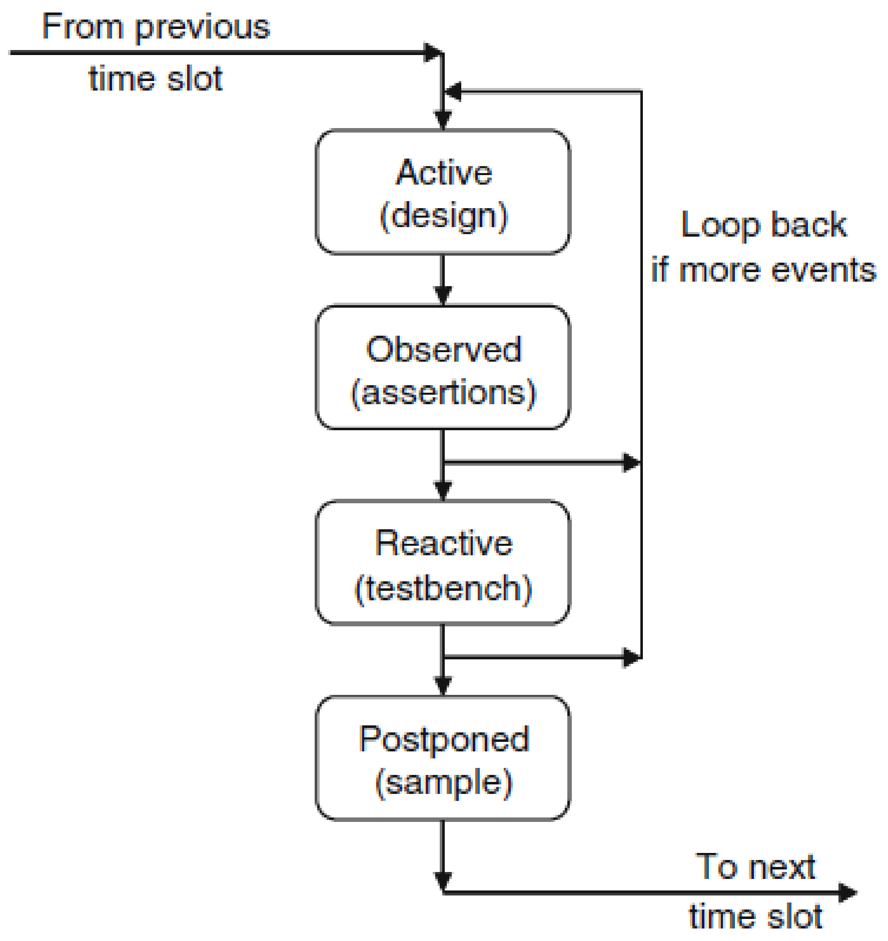


The program Block – timing problems

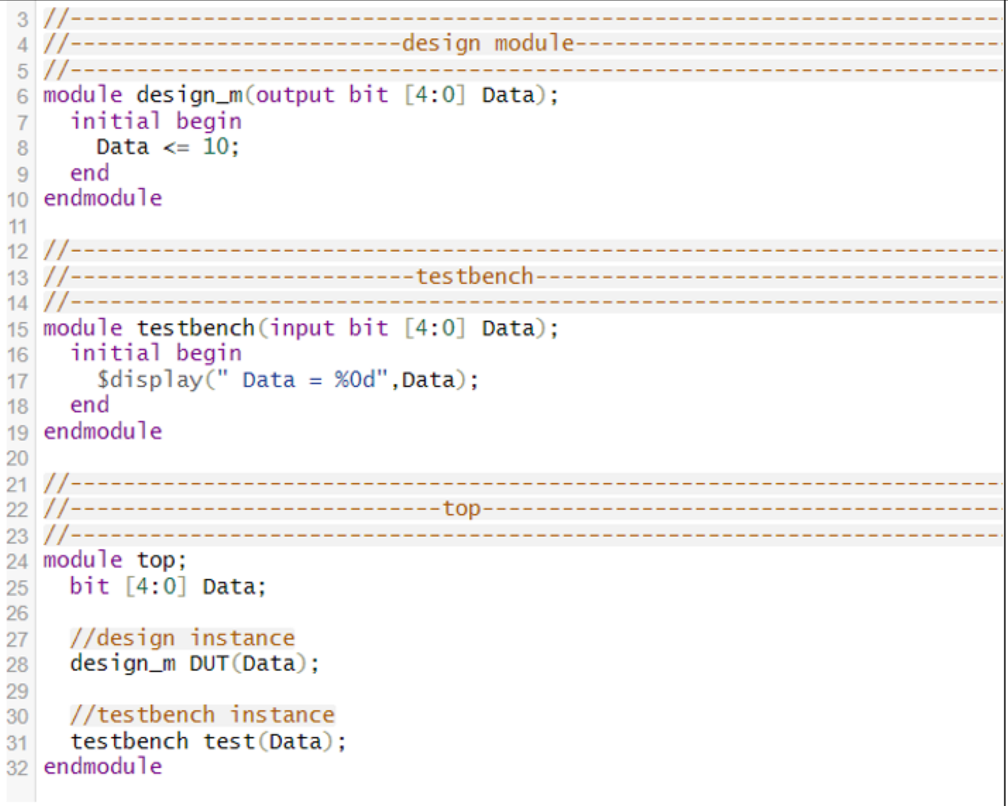
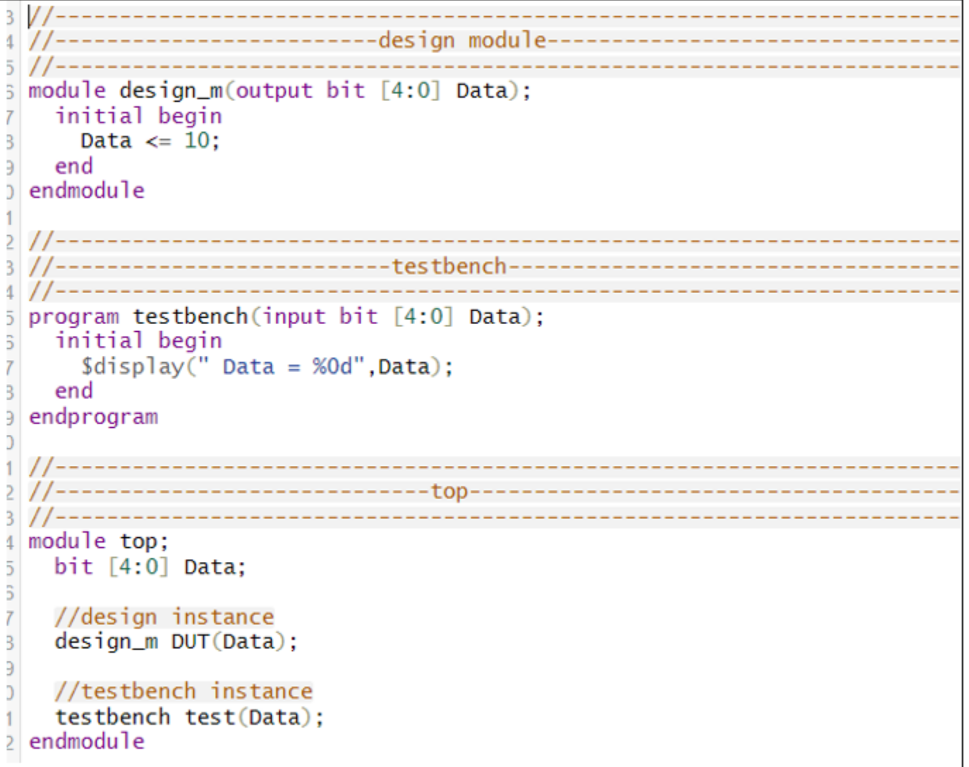
- Problems arise when design and testbench events happen in the same time slot
- SystemVerilog introduces a new time slot for testbench execution
- Program blocks allow the testbench to execute in a separate time slot (reactive region)
- It offers a structured approach to designing and verifying complex systems by effectively managing the execution flow of different elements within a program block.

Timing Regions

- The execution flow during a time slot consists of four regions:
  - **<u>Preponed region:</u>** Samples signals before any changes
  - **<u>Active region:</u>** It is responsible for running design events, including RTL and gate code, as well as the clock generator.
  - **<u>Observed region:</u>** This region allows for the verification of specific properties and conditions within the design. (Assertions)
  - **<u>Reactive region:</u>** The testbench interacts with the design (program).
  - **<u>Postponed region:</u>** Read-only phase ($monitor)



Program vs Module

	Testbench using module	Testbench using program
Code		
output	Data = 0	Data = 10

```
//--------------------------------------------------------------------------
//------------------------design module-------------------------------------
//--------------------------------------------------------------------------
module design_m(output bit [4:0] Data);
  initial begin
    Data <= 10;
  end
endmodule

//--------------------------------------------------------------------------
//-------------------------testbench----------------------------------------
//--------------------------------------------------------------------------
module testbench(input bit [4:0] Data);
  initial begin
    $display(" Data = %0d",Data);
  end
endmodule

//--------------------------------------------------------------------------
//----------------------------top-------------------------------------------
//--------------------------------------------------------------------------
module top;
  bit [4:0] Data;

  //design instance
  design_m DUT(Data);

  //testbench instance
  testbench test(Data);
endmodule
```

```
//--------------------------------------------------------------------------
//------------------------design module-------------------------------------
//--------------------------------------------------------------------------
module design_m(output bit [4:0] Data);
  initial begin
    Data <= 10;
  end
endmodule

//--------------------------------------------------------------------------
//-------------------------testbench----------------------------------------
//--------------------------------------------------------------------------
program testbench(input bit [4:0] Data);
  initial begin
    $display(" Data = %0d",Data);
  end
endprogram

//--------------------------------------------------------------------------
//----------------------------top-------------------------------------------
//--------------------------------------------------------------------------
module top;
  bit [4:0] Data;

  //design instance
  design_m DUT(Data);

  //testbench instance
  testbench test(Data);
endmodule
```

A Complete Example
(Interface + Test Program + Top Level Harness)

A Complete Example

```
interface router_io (input bit clock);
    logic reset_n;
    logic [15:0] din;
    logic [15:0] frame_n;
    logic [15:0] valid_n;
    logic [15:0] dout;
    logic [15:0] busy_n;
    logic [15:0] valido_n;
    logic [15:0] frameo_n;

    clocking cb @(posedge clock);
        default input #1ns output #0ns;

        //output reset_n;
        output din;
        output frame_n;
        output valid_n;
        input dout;
        input busy_n;
        input valido_n;
        input frameo_n;
    endclocking

    modport TB (clocking cb, output reset_n);
endinterface
```

```
program automatic test (router_io.TB rtr_io);
    //testbench code in initial block
    inital begin
        reset();
    end

    task reset();
        //reset conditions
        rtr_io.reset_n = 1'b0;
        rtr_io.cb.frame_n <= 16'hffff;
        rtr_io.cb.valid_n <= ~('b0);
        //advance 2 clock cycles
        repeat(2) @(rtr_io.cb);
        rtr_io.cb.reset_n <= 1'b1;
        //advance 15 clock cycles
        repeat(15) @(rtr_io.cb);
    endtask: reset
endprogram
```

A Complete Example

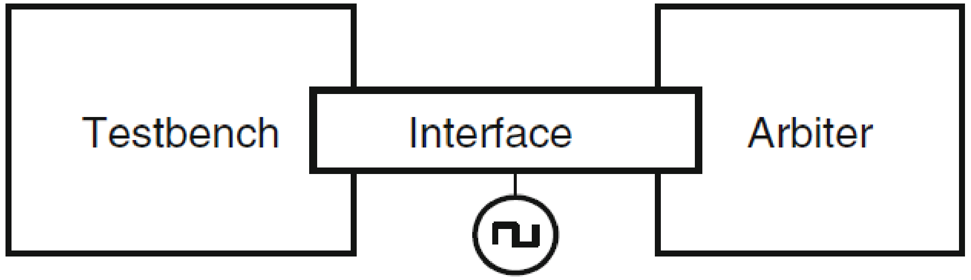
```
module router(
    reset_n;
     din;
     frame_n;
     valid_n;
     dout;
     busy_n;
     valido_n;
     frameo_n;)

     //....
endmodule: router
```

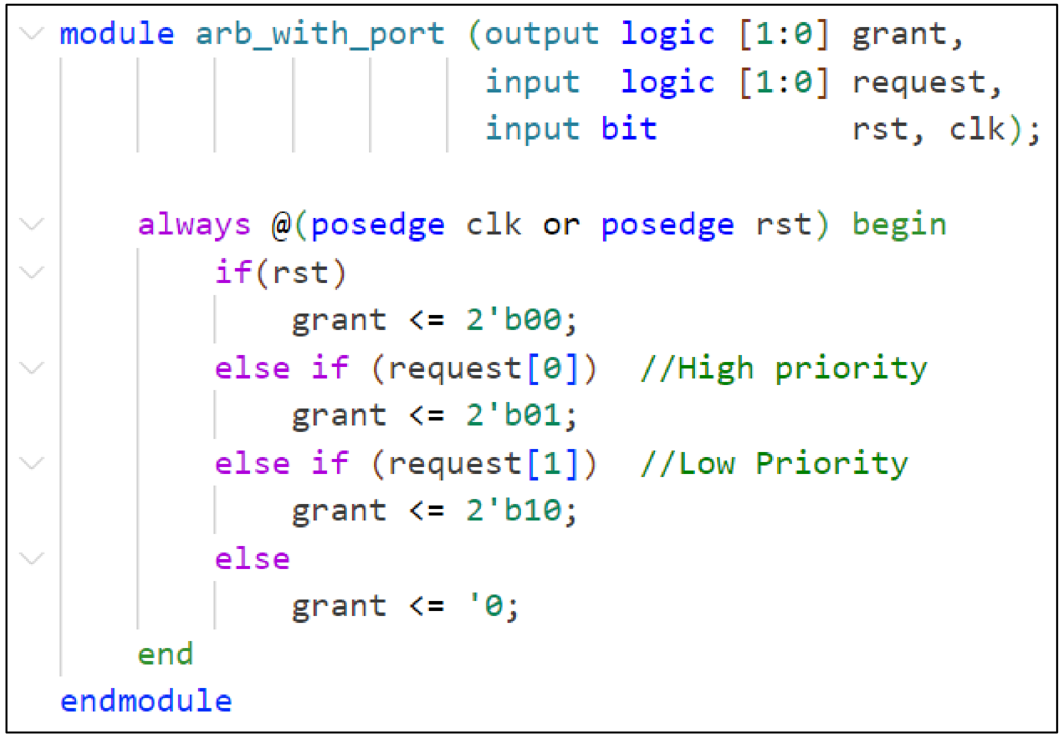
```
module router_test_top;
    bit          clk;
    always #50 clk = ~clk;

    router_io top_io(clk);
    test router_test(top_io);
    router dut(.reset_n  (top_io.reset_n),
               .clock    (top_io.clock),
               .frame_n (top_io.frame_n),
               .valid_n (top_io.valid_n),
               .din      (top_io.din),
               .dout     (top_io.dout),
               .busy_n   (top_io.busy_n),
               .valido_n(top_io.valido_n),
               .frameo_n(top_io.frameo_n));
        //...
endmodule: router_test_top
```

Lab Task

Lab Task

- Develop a SystemVerilog test program to test the DUT (arbiter.v).
- 1) Develop SystemVerilog testbench files
- 2) Compile and simulate with VCS
- 3) Verify Simulation results using DVE





```
module arb_with_port (output logic [1:0] grant,
                      input  logic [1:0] request,
                      input bit          rst, clk);

    always @(posedge clk or posedge rst) begin
        if(rst)
            grant <= 2'b00;
        else if (request[0])  //High priority
            grant <= 2'b01;
        else if (request[1])  //Low Priority
            grant <= 2'b10;
        else
            grant <= '0;
    end
endmodule
```

Lab Task: Steps

1. For the given DUT create an interface to connect the test program to the router (**arbiter_io.sv**)
   a. Declare a **clocking** block and **modport** as needed.
   b. Optional: add specifications for input and output skews.
2. Write the test program (**test.sv**)
   a. Include arguments that connect to the modport in the interface block
   b. Print a message to the screen inside the initial block, ex: “Hello World!”
3. Connect everything in a harness file and include a simple clock generator (**arbiter_test_top.sv**)
4. Compile all files with the following command (using VCS):
   - vcs -sverilog arbiter_test_top.sv test.sv arbiter_io.sv arbiter.v
   - VCS will compile the files and create an executable file called simv
5. Simulate by executing the binary created by VCS
   - ./simv
6. Check to see if the $display message appears.

Lab Assignment

Lab Assignment: Steps

1. Edit your test program (**test.sv**) to include the following:
   a. reset the DUT
   b. drive the arbiter
2. Run the simulation using DVE

- dve &
- File -> open database -> test.vcd

3. To generate the vcd file add the following in (**arbiter_test_top.sv**):

```
initial begin
    $dumpfile(test.vcd);
    $dumpvars;
  end
```