WIT NXP
Team 7

Report on Mini & Major Project

Topic

Design and Implementation of FSM-based adder synchronized with clock in Verilog, Verifying it's working through UVM Testbench

Part 1: Verilog implementation for Adder synchronized with the clock [Minor]
Part-2: UVM Testbench for Adder [Major]

Under the Mentorship
of
Mr Shashi Kant Sharma

Submitted by

- Sharanya, Snehalata, Sonakshi, Sanya, Somya, Sreenija

CONTENTS

LIST OF FIGURES

INTRODUCTION

Digital circuits and systems play a pivotal role in our modern world, powering everything from smartphones and computers to the intricate electronics within vehicles and industrial machinery. The foundation of these digital systems lies in the fundamental operation of arithmetic, particularly addition. The core of this operation often involves the design and implementation of adders, which are essential building blocks in digital circuitry.
Finite State Machines (FSMs) are a powerful and structured approach to controlling and sequencing digital operations, and they have a significant impact on the design and functionality of adders. In this report, we delve into the world of FSM-based adders synchronized with a clock—a critical aspect of digital design.

Significance of FSM-Based Adders:

An adder is a fundamental component used to perform arithmetic operations, primarily addition. While simple in concept, the efficient and controlled implementation of adders is crucial in the design of complex digital systems. FSM-based adders are a sophisticated approach to this implementation, and their significance can be summarized in several key points:

1. Controlled Sequencing: FSMs provide a structured means of controlling the sequence of operations within a digital circuit. When applied to adders, they enable precise control over when addition operations occur, ensuring that they happen in a well-defined sequence.

2. Resource Optimization: FSM-based adders allow for efficient utilization of hardware resources. By controlling when and how the adder operates, unnecessary resource consumption is minimized, which is essential for optimizing the performance and area of digital systems.

3. Error Handling: FSMs can be designed to include error detection and correction mechanisms, enhancing the reliability of adders in critical applications. Error detection can prevent incorrect results from propagating through a digital system.

Importance of Synchronization with a Clock

Synchronization with a clock is a fundamental concept in digital design and is of utmost importance in FSM-based adders. Clock signals act as the heartbeat of digital circuits, ensuring that operations occur at precise intervals and in a coordinated manner. Here's why synchronization with a clock is crucial in the context of FSM-based adders:

1. Deterministic Operation: Clocks provide determinism to the timing of operations, allowing digital designers to predict when events will occur. This predictability is essential for ensuring that data is processed correctly.

2. Data Stability: Clock signals help stabilize the data within the circuit. In FSM-based adders, this is vital to guarantee that operands are stable before addition and that results are valid when they are read.

3. Mitigating Glitches: Synchronisation with a clock helps prevent glitches and hazards that can arise in asynchronous designs. Glitches can lead to incorrect results and unpredictable behaviour, which are unacceptable in most digital systems.

PROJECT OBJECTIVES

The primary objectives of this project are to design and implement an FSM-based adder synchronised with a clock, incorporating control flags for enabling input and output. The project aims to achieve the following specific goals:

1. FSM-Based Adder Design: Develop a Verilog module for an FSM-based adder capable of performing binary addition. This module should include a Finite State Machine (FSM) to control the sequence of operations during addition.

2. Clock Synchronisation: Ensure that the FSM-based adder operates synchronously with a clock signal. All state transitions, data processing, and output generation should occur at well-defined clock edges to maintain determinism and predictability.

3. Control Flags Implementation: Integrate control flags into the FSM-based adder design to enable or disable input data and output results. Specifically, implement "Enable Input" and "Enable Output" control flags to control the flow of data through the adder.

4. Verification and Testing: Develop a comprehensive verification plan and a set of test cases to rigorously test the functionality of the FSM-based adder. Ensure that the adder operates correctly under various conditions and that control flags are effectively utilised.

5. Simulation and Waveform Analysis: Utilise simulation tools to validate the correct operation of the FSM-based adder. Generate waveform diagrams to visualise the behaviour of the adder, demonstrating how clock synchronisation and control flags influence its operation.

6. Converting the Verilog testbench (TB) to a Universal Verification Methodology (UVM) (TB): It involves restructuring and augmenting the Verilog testbench to adhere to the UVM methodology, which is a standardised and powerful approach for verifying complex digital designs

FINITE STATE MACHINE

Detailed explanation of the Finite State Machine (FSM) used in the design:

- Include a state diagram that illustrates the states and transitions.
- Explain the purpose of each state and how transitions occur based on input conditions.

In our project, we have made an FSM-based serial adder synchronized with a clock in Verilog

Serial Adder

A serial adder is a circuit that performs binary addition bit by bit (i.e., instead of presenting both operands at the inputs of an adder at the same time, the operands are fed into the serial adder bit by bit and it generates the answer on the fly). To design such a circuit, we use the state diagram as the mode of describing the behaviour of the circuit and then translate the state diagram into Verilog code.

Step 1: Describe the Serial Adder Using the State Diagram

Define Inputs and Outputs

In this case, we have two data inputs named A and B. Both of them are 1-bit wires. As with the adder we described in the arithmetic circuit, we have a data output F and another output called Cout (Carry Out). We also need a clock signal (named clock here) to provide the timing reference for both the inputs and the outputs and a reset signal (named rst) to bring the circuit into the initial state.

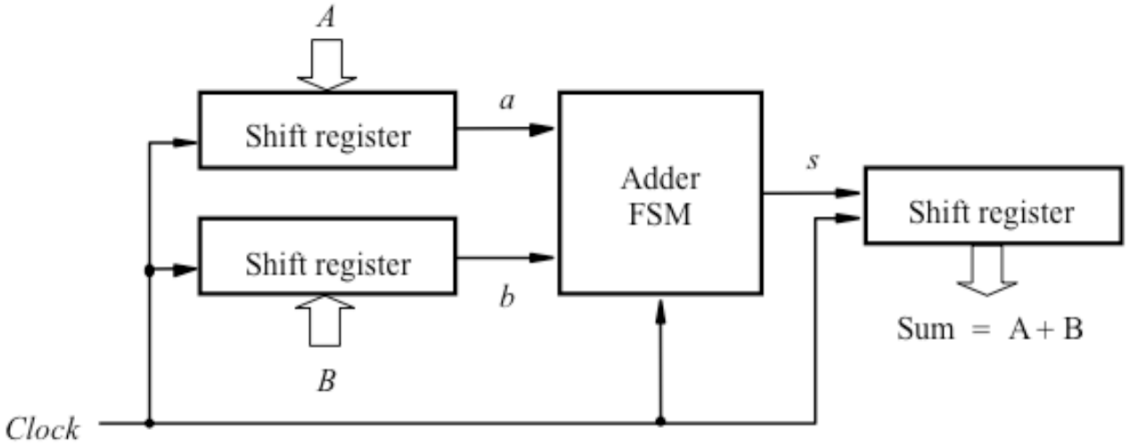


Fig - Design of Serial Adder

Design State Diagram

Since we are designing a Moore Machine thus it is always two possible output from one input given, thus we will need 4 states in Moore Machine accounting for the 4 possible states.

G0 and G1: carry is 0 sum is 0 or 1
H0 andH1: carry is 1 sum is 0 or 1

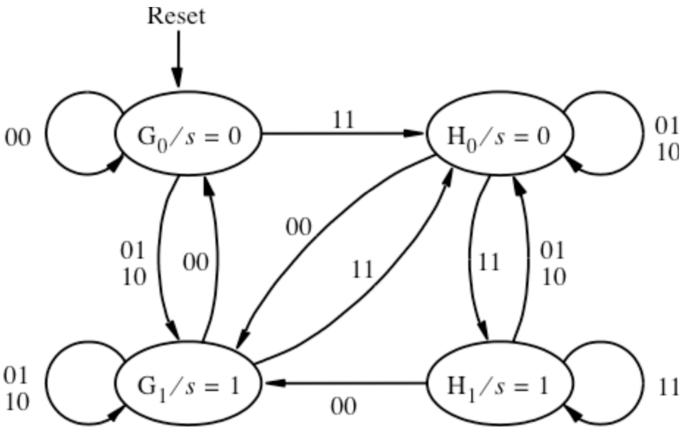
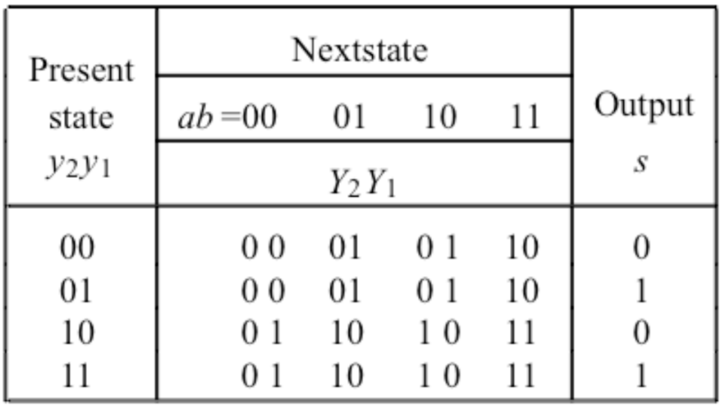


Fig -State Diagram for Moore Typed Serial Adder

State Table



Present	Nextstate				Output
state	*ab* =00	01	10	11	
y_2y_1	Y_2Y_1				*s*
00	0 0	01	0 1	10	0
01	0 0	01	0 1	10	1
10	0 1	10	1 0	11	0
11	0 1	10	1 0	11	1

Fig - State Table for the Serial Adder

$$Y2 = ab+ay2+by2$$
$$Y1 = a \oplus b \oplus y2$$
$$s = y1$$

	y2’y1’	y2’y1	y2y1	y2y1’
a’b’	0	1	1	0
a’b	0	1	1	0
ab	0	1	1	0
ab’	0	1	1	0

S=y1

	y2’y1’	y2’y1	y3’y1	y2y1’
a’b’	0	0	0	0
a’b	0	0	1	1
ab	1	1	1	1
ab’	0	0	1	1

Y2= ab+ay2+by2

	y2’y1’	y2’y1	y2y1	y2y1’
a’b’	0	0	1	1
a’b	1	1	0	0
ab	0	0	1	1
ab’	1	1	0	0

Y1=a xor b xor y2

Circuit Diagram for the Moore-type Serial Adder

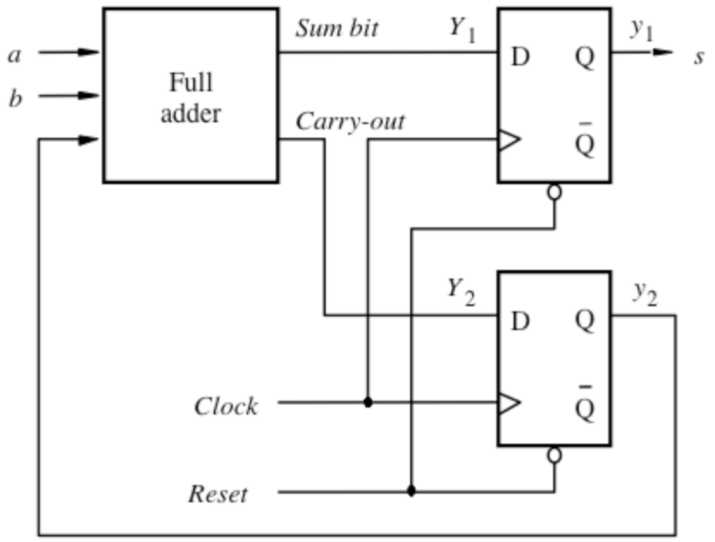


Fig - Circuit Diagram for the Serial Adder

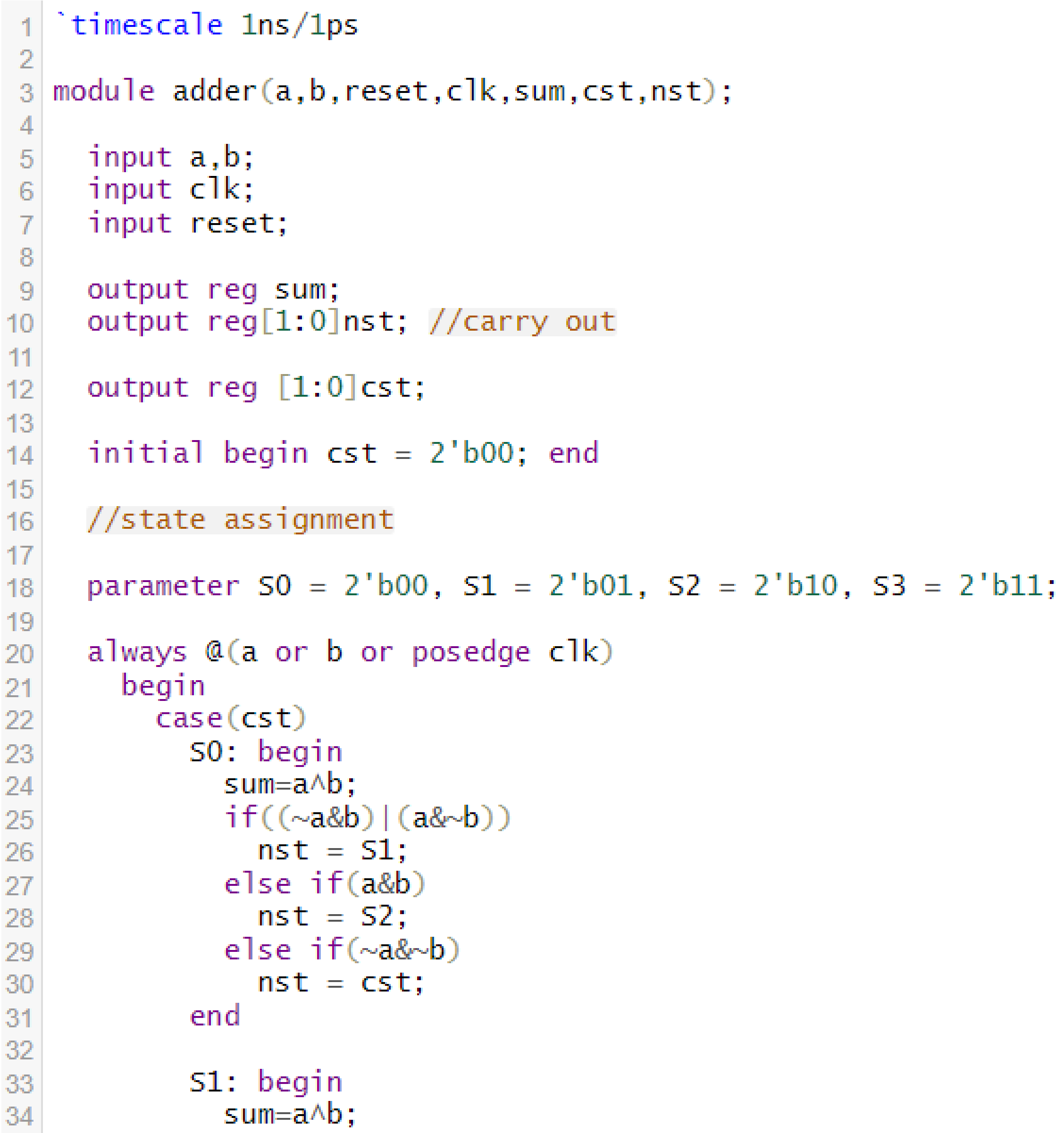
VERILOG CODE

DESIGN

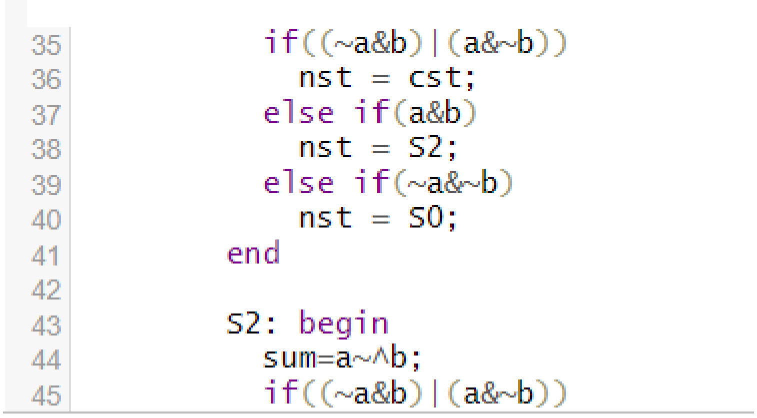
```
`timescale 1ns/1ps

module adder(a,b,reset,clk,sum,cst,nst);

  input a,b;
  input clk;
  input reset;

  output reg sum;
  output reg[1:0]nst; //carry out

  output reg [1:0]cst;

  initial begin cst = 2'b00; end

  //state assignment

  parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @(a or b or posedge clk)
    begin
  case(cst)
    S0: begin
      sum=a^b;
      if((~a&b)|(a&~b))
        nst = S1;
      else if(a&b)
        nst = S2;
      else if(~a&~b)
        nst = cst;
    end

    S1: begin
      sum=a^b;
      if((~a&b)|(a&~b))
        nst = cst;
      else if(a&b)
        nst = S2;
      else if(~a&~b)
        nst = S0;
    end

    S2: begin
      sum=a~^b;
      if((~a&b)|(a&~b))
        nst = cst;
      else if(a&b)
        nst = S3;
      else if(~a&~b)
        nst = S1;
    end

    S3: begin
      sum=a~^b;
      if((~a&b)|(a&~b))
        nst = S2;
      else if(a&b)
        nst = cst;
      else if(~a&~b)
        nst = S1;
    end


    default: nst = S0;
  endcase
end
```

```
  //reset facility

  always @ (posedge reset or posedge clk)
    begin
      if(reset)
        begin
          sum<=1'b0;
          cst<=S0;
        end
      else
        begin
          cst<=nst;
        end
    end
endmodule
```

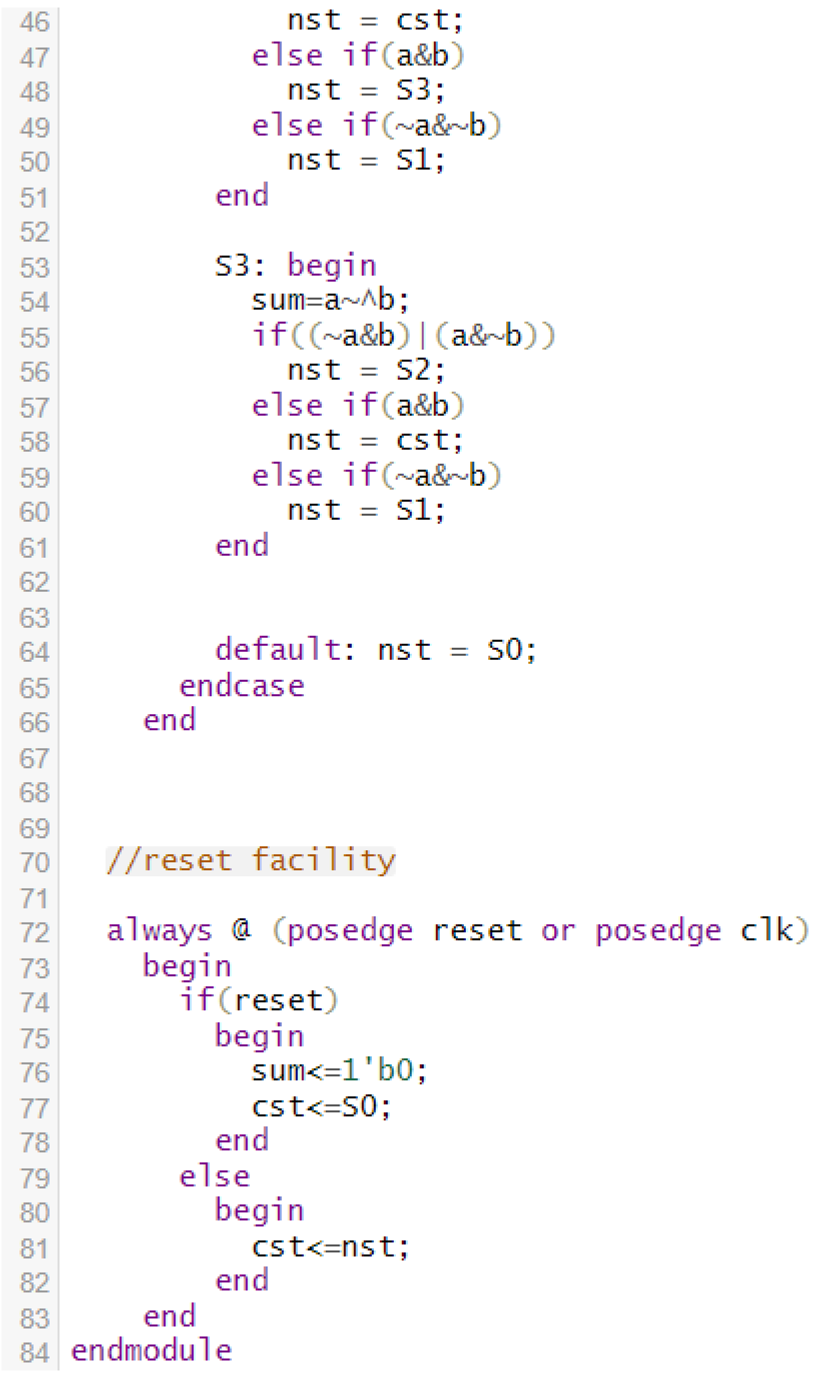
Code Snippets from EDA Playground



```
`timescale 1ns/1ps

module adder(a,b,reset,clk,sum,cst,nst);

  input a,b;
  input clk;
  input reset;

  output reg sum;
  output reg[1:0]nst; //carry out

  output reg [1:0]cst;

  initial begin cst = 2'b00; end

  //state assignment

  parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @(a or b or posedge clk)
    begin
      case(cst)
        S0: begin
          sum=a^b;
          if((~a&b)|(a&~b))
            nst = S1;
          else if(a&b)
            nst = S2;
          else if(~a&~b)
            nst = cst;
        end

        S1: begin
          sum=a^b;
```



```
          if((~a&b)|(a&~b))
            nst = cst;
          else if(a&b)
            nst = S2;
          else if(~a&~b)
            nst = S0;
        end

        S2: begin
          sum=a~^b;
          if((~a&b)|(a&~b))
```



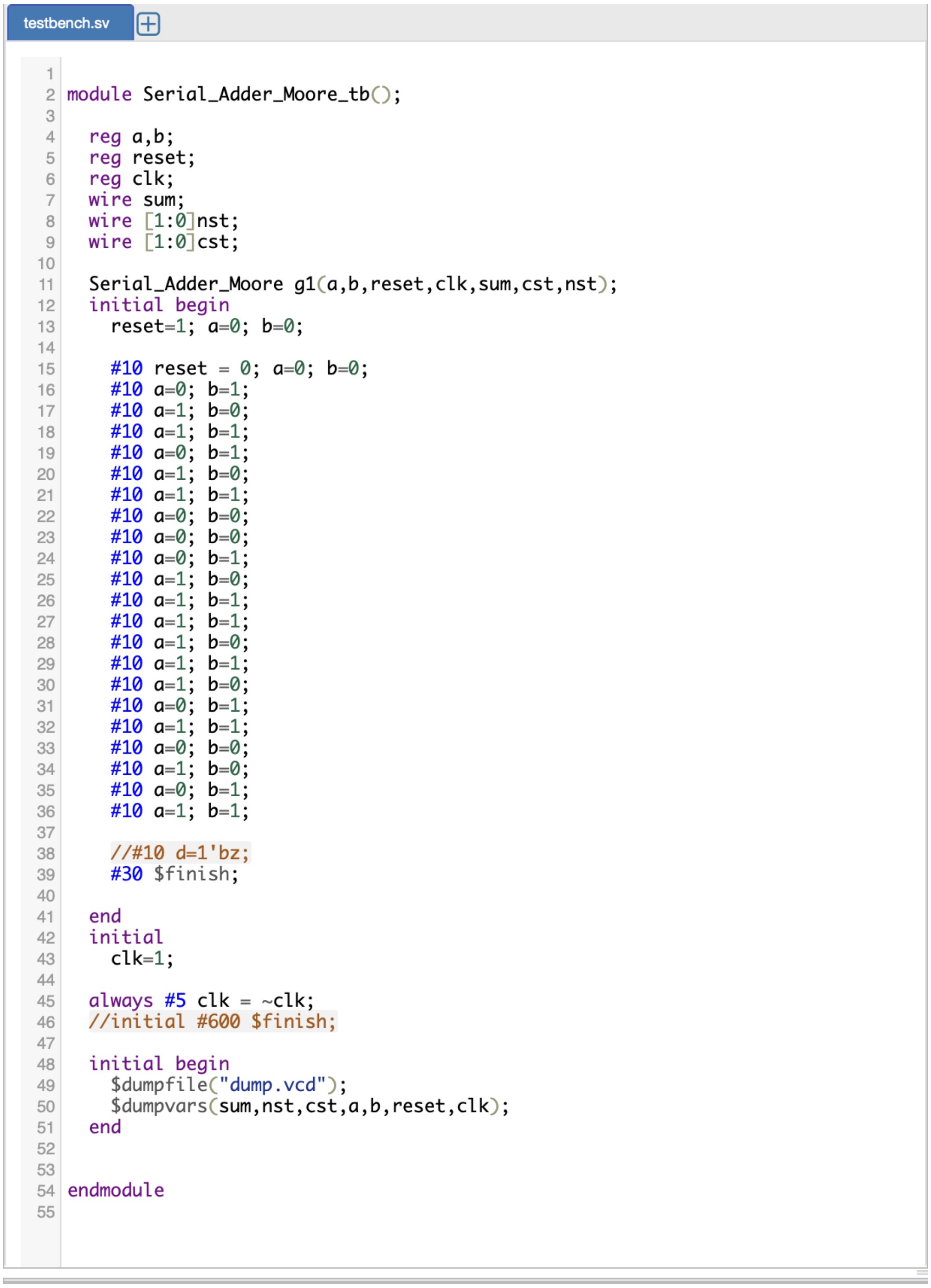
```
            nst = cst;
          else if(a&b)
            nst = S3;
          else if(~a&~b)
            nst = S1;
        end

        S3: begin
          sum=a~^b;
          if((~a&b)|(a&~b))
            nst = S2;
          else if(a&b)
            nst = cst;
          else if(~a&~b)
            nst = S1;
        end


        default: nst = S0;
      endcase
    end



  //reset facility

  always @ (posedge reset or posedge clk)
    begin
      if(reset)
        begin
          sum<=1'b0;
          cst<=S0;
        end
      else
        begin
          cst<=nst;
        end
    end
endmodule
```

TESTBENCH

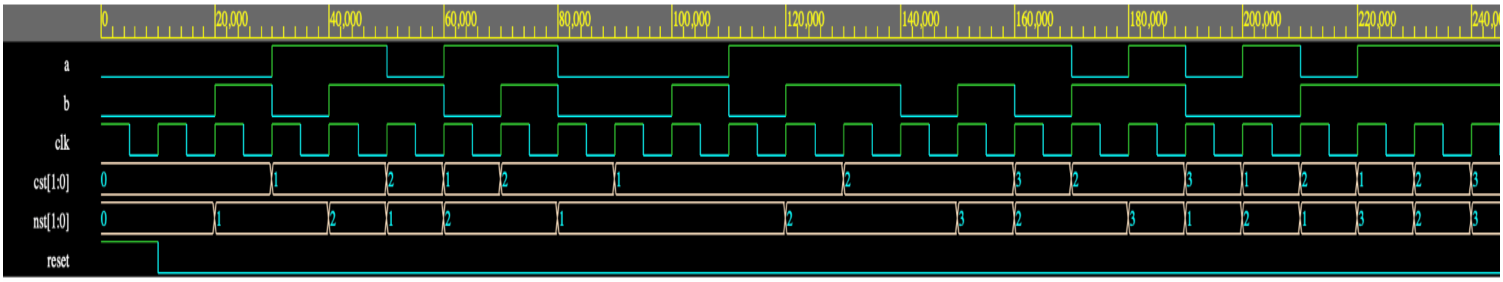
```
module Serial_Adder_Moore_tb();

 reg a,b;
 reg reset;
 reg clk;
 wire sum;
 wire [1:0]nst;
```

```
  wire [1:0]cst;

  Serial_Adder_Moore g1(a,b,reset,clk,sum,cst,nst);
  initial begin
    reset=1; a=0; b=0;

    #10 reset = 0; a=0; b=0;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #10 a=0; b=0;
    #10 a=0; b=0;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #10 a=1; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #10 a=1; b=0;
    #10 a=0; b=1;
    #10 a=1; b=1;
    #10 a=0; b=0;
    #10 a=1; b=0;
    #10 a=0; b=1;
    #10 a=1; b=1;

    //#10 d=1'bz;
    #30 $finish;

  end
  initial
    clk=1;

  always #5 clk = ~clk;
  //initial #600 $finish;

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(sum,nst,cst,a,b,reset,clk);
  end


endmodule
```

Code Snippet from EDA Playground



```

module Serial_Adder_Moore_tb();

  reg a,b;
  reg reset;
  reg clk;
  wire sum;
  wire [1:0]nst;
  wire [1:0]cst;

  Serial_Adder_Moore g1(a,b,reset,clk,sum,cst,nst);
  initial begin
    reset=1; a=0; b=0;

    #10 reset = 0; a=0; b=0;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #10 a=0; b=0;
    #10 a=0; b=0;
    #10 a=0; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #10 a=1; b=1;
    #10 a=1; b=0;
    #10 a=1; b=1;
    #10 a=1; b=0;
    #10 a=0; b=1;
    #10 a=1; b=1;
    #10 a=0; b=0;
    #10 a=1; b=0;
    #10 a=0; b=1;
    #10 a=1; b=1;

    //#10 d=1'bz;
    #30 $finish;

  end
  initial
    clk=1;

  always #5 clk = ~clk;
  //initial #600 $finish;

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(sum,nst,cst,a,b,reset,clk);
  end


endmodule

```



Output EPWave after Run

Part 2 - UVM Testbench for FSM-Based-Adder

The Universal Verification Methodology (UVM) emerged as a response to the growing demand for **automated verification processes**. UVM is essentially a comprehensive set of API (Application Programming Interface) and well-established verification principles designed for use with SystemVerilog. This methodology assists engineers in creating efficient and effective verification environments. Importantly, UVM is an open-source standard maintained by Accellera, making it readily accessible on their website.

By establishing a universal set of conventions and techniques in the realm of verification, UVM prompted the development of generic verification components that could be easily transported from one project to another. This emphasis on uniformity fostered collaboration and the sharing of best practices among the user community. Additionally, it motivated the creation of verification components that were sufficiently generic to allow for straightforward extensions and improvements without altering the original code.

These combined factors significantly reduced the effort required to develop new verification environments. Designers could now readily reuse testbenches and adapt components from previous projects to suit their specific requirements, streamlining the verification process.

The Universal Verification Methodology (UVM) testbench architecture:

It is a structured framework for verifying digital designs using UVM, which is a widely accepted methodology in the semiconductor industry. The UVM testbench is designed to facilitate efficient verification of a Design Under Test (DUT). Here's an overview of the UVM testbench architecture and how to proceed ideally:

UVM Testbench Architecture:

Testbench Top:

- The top-level module that coordinates the testbench and instantiates various components of the UVM architecture.

Test Sequences:

- These are high-level sequences that model specific test scenarios or use cases. Sequences determine the order in which different transactions are applied to the DUT.

Sequence Items:

- Sequence items represent individual transactions or data transfers to and from the DUT. Each sequence item captures the essential characteristics of a transaction.

Driver:

- The driver is responsible for taking sequence items from the sequences and driving them into the DUT's inputs. It coordinates the stimulus generation.

Monitor:

- The monitor observes the DUT's outputs, captures data, and checks for errors. It can be used to generate response sequence items based on the DUT's outputs.

Scoreboard:

- The scoreboard compares the DUT's outputs with expected results or reference models. It detects errors and generates reports if discrepancies are found.

Functional Coverage:

- Functional coverage is used to track the completeness of testing by defining coverage bins that represent different aspects of the DUT's behaviour. The coverage data helps in determining how well the DUT has been tested.

UVM Testbench Environment:

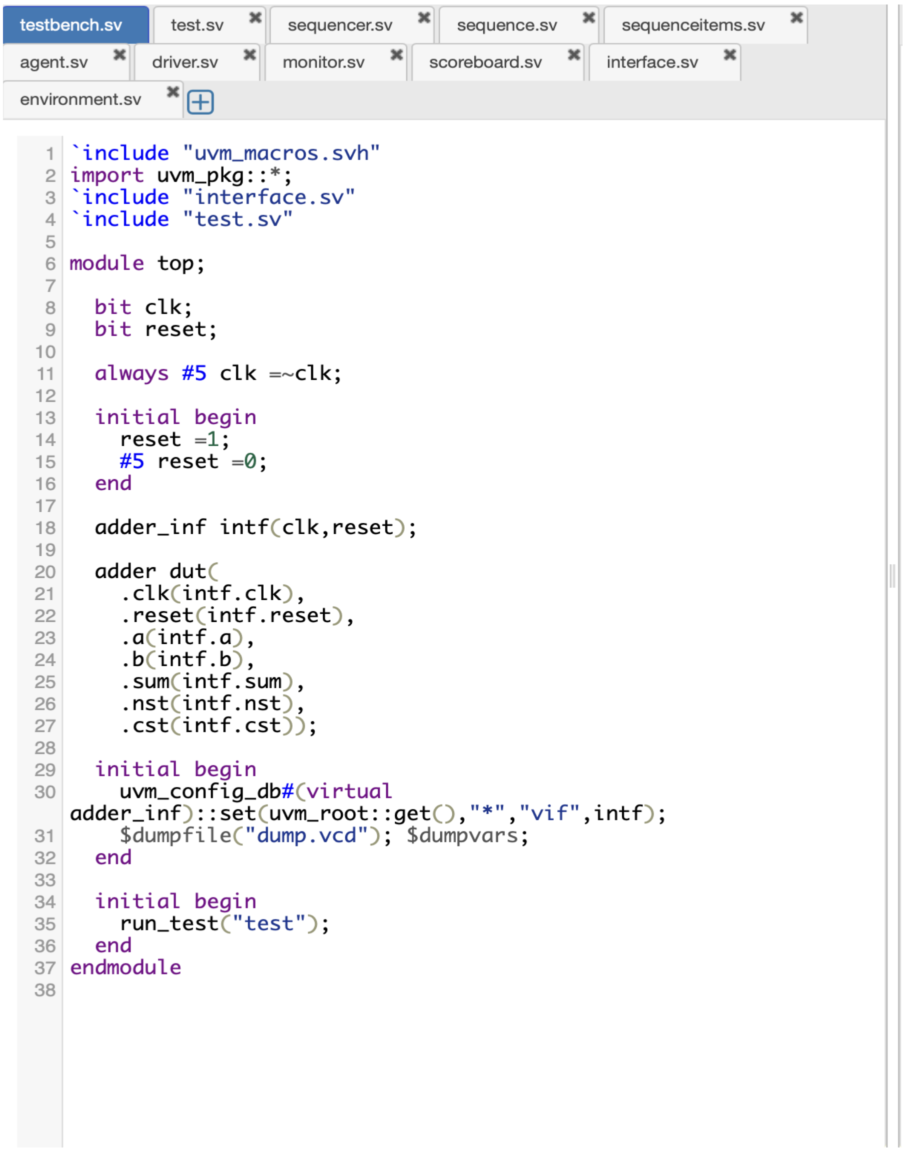
- This environment provides the necessary setup and configuration for the DUT and the test. It may include:
  - Clock and reset generators
  - Bus functional models (BFMs) for interfacing with the DUT
  - Test-specific configuration and settings
  - Randomization and constraint mechanisms for generating stimulus

Being subjective to our project, we initially started with writing DUT and Tb for Verilog code. Later translated the tb into UVM tb, followed by writing all the essential components involved and observing the waveform obtained.

In a Universal Verification Methodology (UVM) testbench, each of the components below mentioned plays a specific role in the verification process. Let's discuss the significance of each of these SystemVerilog (SV) files in the context of a UVM testbench:

1. testbench.sv:

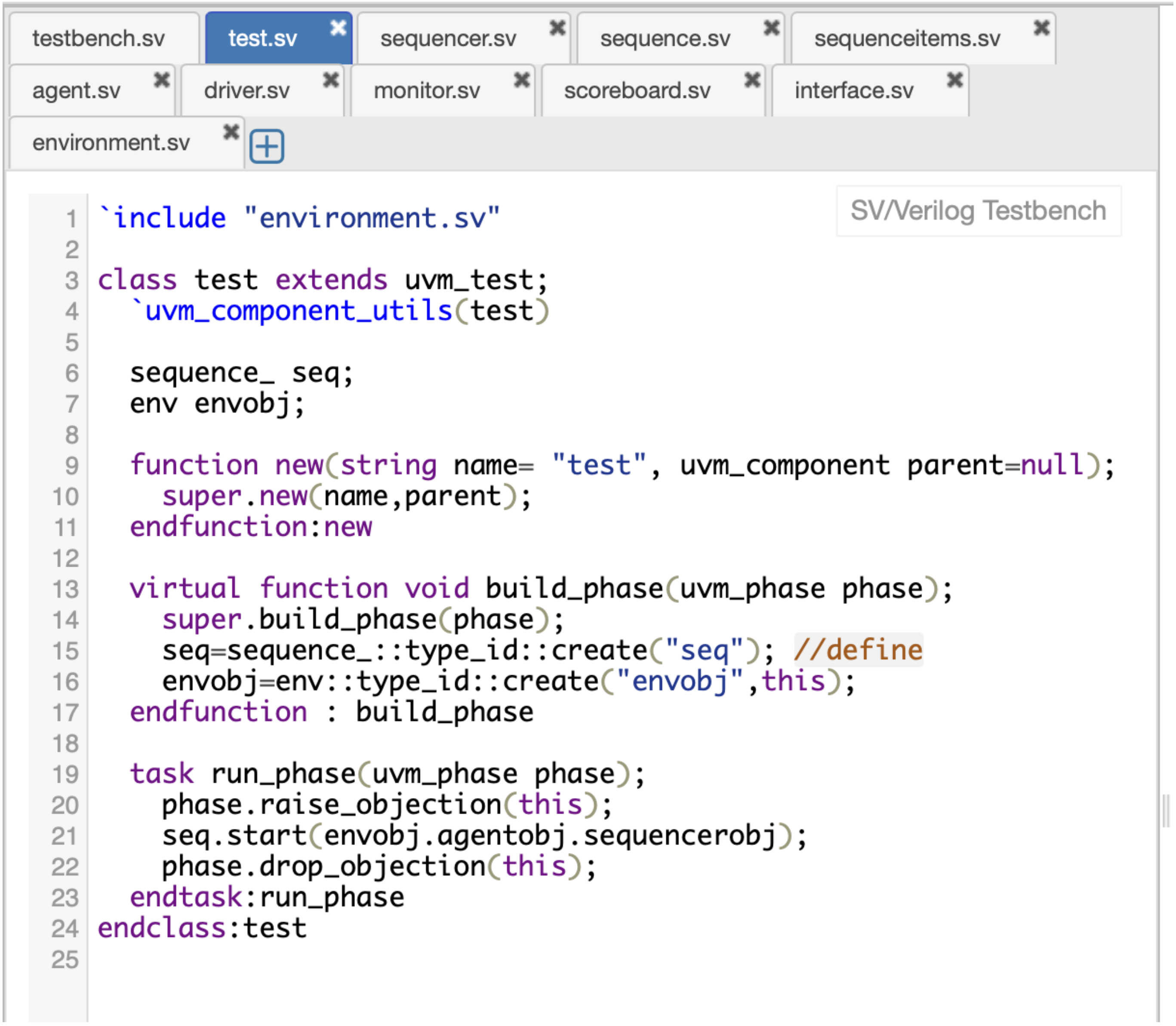
Significance: The testbench.sv file typically serves as the top-level module for your UVM testbench. It is the central point of coordination for all testbench components, including test scenarios, environment setup, and simulation configuration.



```
`include "uvm_macros.svh"
import uvm_pkg::*;
`include "interface.sv"
`include "test.sv"

module top;

  bit clk;
  bit reset;

  always #5 clk =~clk;

  initial begin
    reset =1;
    #5 reset =0;
  end

  adder_inf intf(clk,reset);

  adder dut(
    .clk(intf.clk),
    .reset(intf.reset),
    .a(intf.a),
    .b(intf.b),
    .sum(intf.sum),
    .nst(intf.nst),
    .cst(intf.cst));

  initial begin
    uvm_config_db#(virtual
adder_inf)::set(uvm_root::get(),"*","vif",intf);
    $dumpfile("dump.vcd"); $dumpvars;
  end

  initial begin
    run_test("test");
  end
endmodule

```

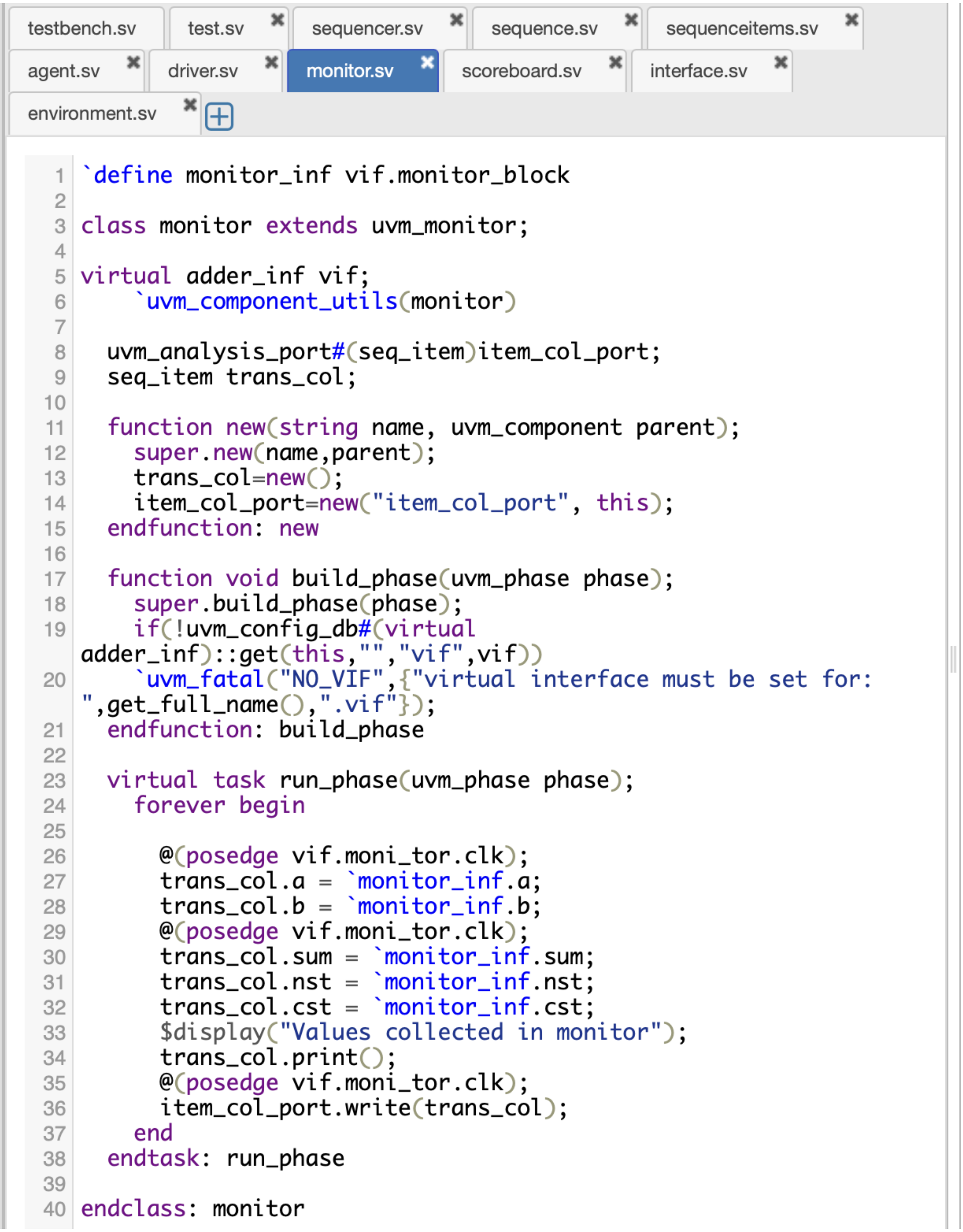
2. test.sv:

- Significance: The test.sv file specifies the test scenario or use case to be executed. It controls the execution of sequences and serves as a link between the testbench and the sequences that apply stimulus to the DUT.



```
`include "environment.sv"

class test extends uvm_test;
  `uvm_component_utils(test)

  sequence_ seq;
  env envobj;

  function new(string name= "test", uvm_component parent=null);
    super.new(name,parent);
  endfunction:new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    seq=sequence_::type_id::create("seq"); //define
    envobj=env::type_id::create("envobj",this);
  endfunction : build_phase

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    seq.start(envobj.agentobj.sequencerobj);
    phase.drop_objection(this);
  endtask:run_phase
endclass:test
```

3. monitor.sv:

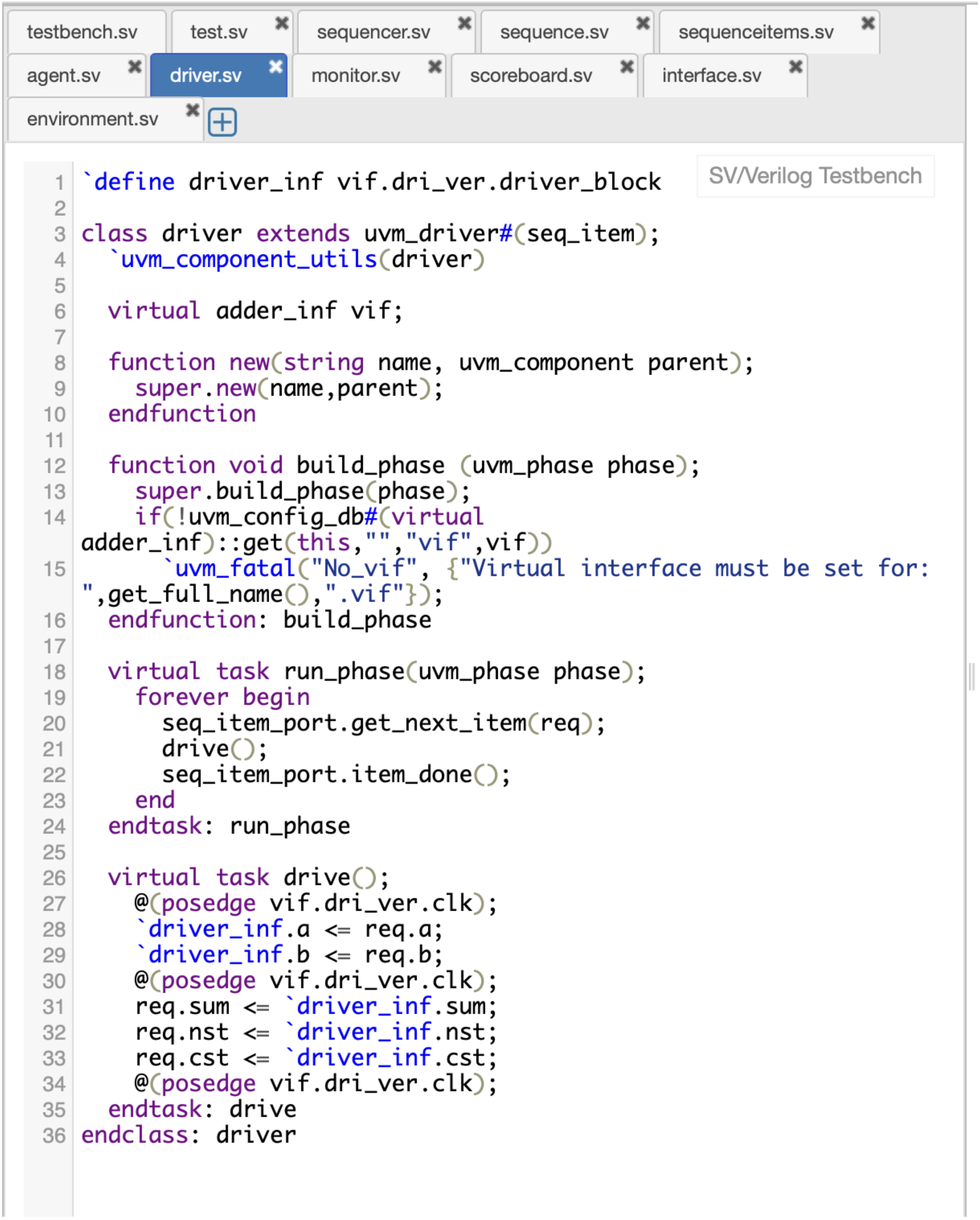


```
`define monitor_inf vif.monitor_block

class monitor extends uvm_monitor;

virtual adder_inf vif;
    `uvm_component_utils(monitor)

  uvm_analysis_port#(seq_item)item_col_port;
  seq_item trans_col;

  function new(string name, uvm_component parent);
    super.new(name,parent);
    trans_col=new();
    item_col_port=new("item_col_port", this);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual adder_inf)::get(this,"","vif",vif))
    `uvm_fatal("NO_VIF",{"virtual interface must be set for: ",get_full_name(),".vif"});
  endfunction: build_phase

  virtual task run_phase(uvm_phase phase);
    forever begin

      @(posedge vif.moni_tor.clk);
      trans_col.a = `monitor_inf.a;
      trans_col.b = `monitor_inf.b;
      @(posedge vif.moni_tor.clk);
      trans_col.sum = `monitor_inf.sum;
      trans_col.nst = `monitor_inf.nst;
      trans_col.cst = `monitor_inf.cst;
      $display("Values collected in monitor");
      trans_col.print();
      @(posedge vif.moni_tor.clk);
      item_col_port.write(trans_col);
    end
  endtask: run_phase

endclass: monitor
```

- Significance: The monitor.sv file is responsible for observing the outputs of the Design Under Test (DUT) and capturing relevant data. It is a crucial component for checking the behavior of the DUT and may also generate response sequence items.

4. driver.sv:

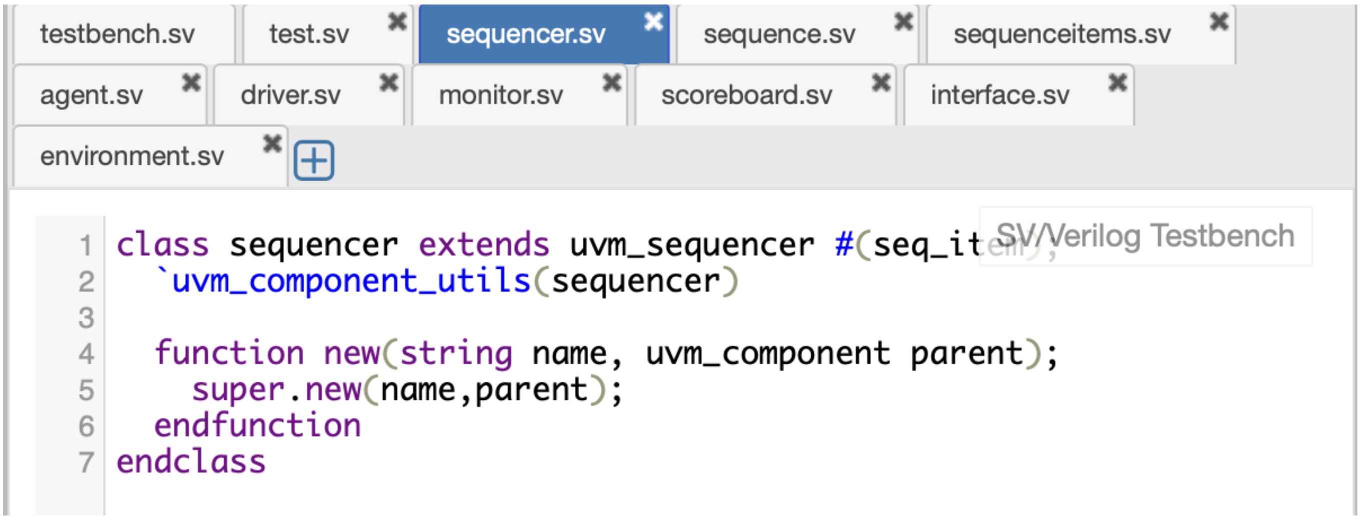
- Significance: The driver.sv file is responsible for driving sequence items into the DUT's inputs. It takes the stimulus generated by sequences and applies it to the DUT.



```
`define driver_inf vif.dri_ver.driver_block

class driver extends uvm_driver#(seq_item);
  `uvm_component_utils(driver)

  virtual adder_inf vif;

  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction

  function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual adder_inf)::get(this,"","vif",vif))
      `uvm_fatal("No_vif", {"Virtual interface must be set for: ",get_full_name(),".vif"});
  endfunction: build_phase

  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      drive();
      seq_item_port.item_done();
    end
  endtask: run_phase

  virtual task drive();
    @(posedge vif.dri_ver.clk);
    `driver_inf.a <= req.a;
    `driver_inf.b <= req.b;
    @(posedge vif.dri_ver.clk);
    req.sum <= `driver_inf.sum;
    req.nst <= `driver_inf.nst;
    req.cst <= `driver_inf.cst;
    @(posedge vif.dri_ver.clk);
  endtask: drive
endclass: driver
```

5. sequencer.sv:

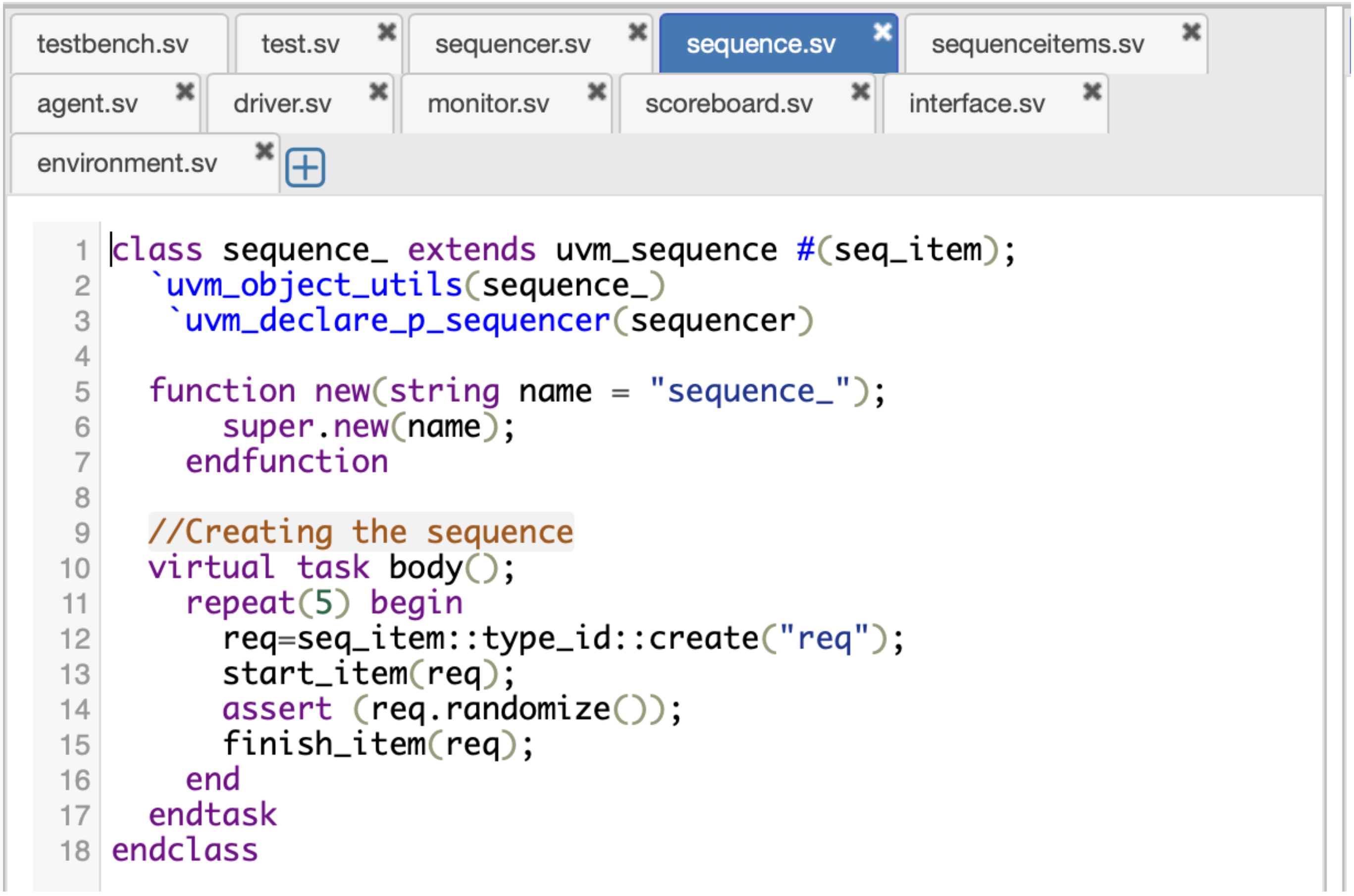
- Significance: The sequencer.sv file manages the scheduling and execution of sequences. It ensures that sequences are executed in the desired order and can control the sequencing of transactions.



```
class sequencer extends uvm_sequencer #(seq_item);
  `uvm_component_utils(sequencer)

  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction
endclass
```

6. sequence.sv:

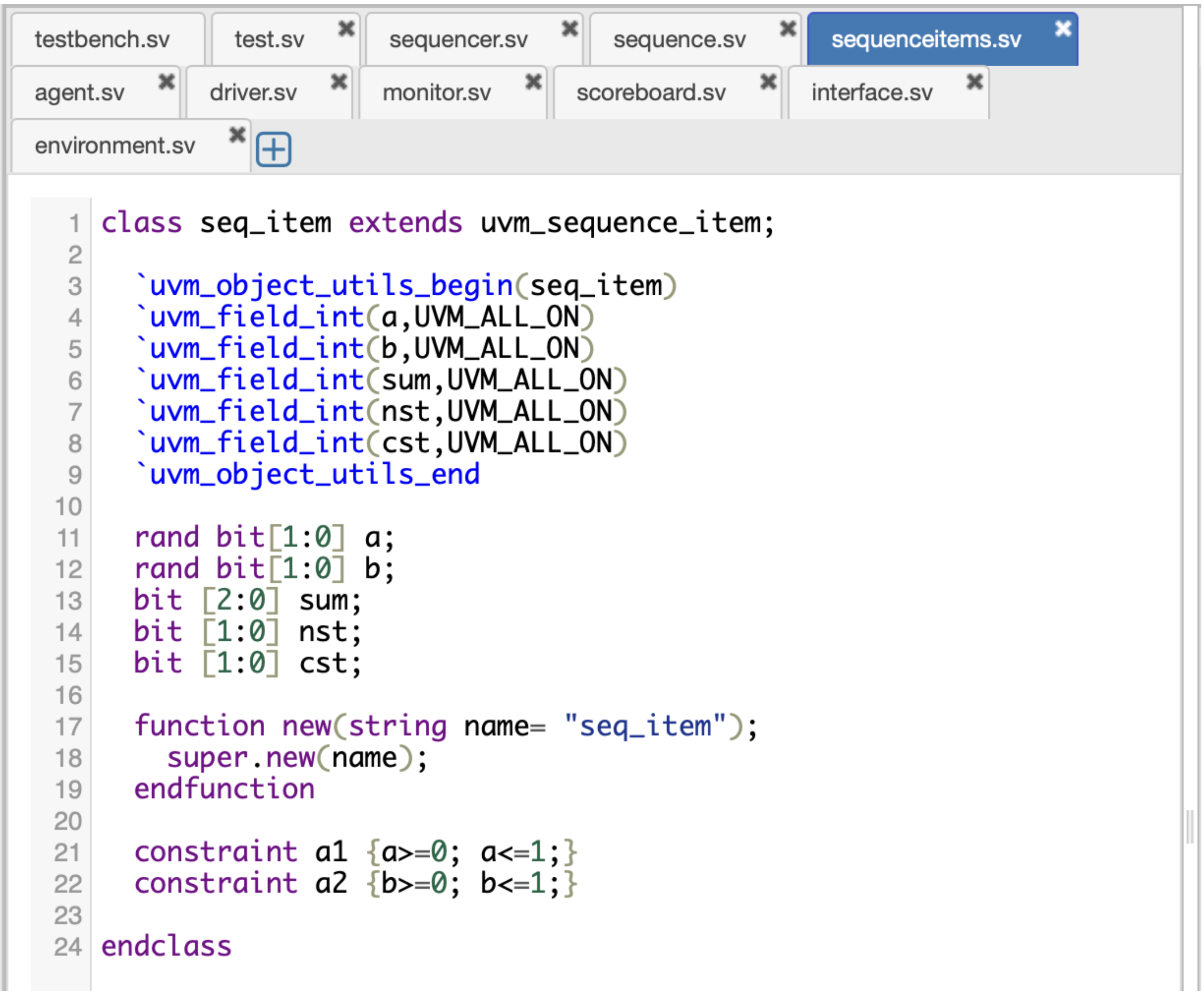
- Significance: The sequence.sv file represents a specific test scenario or transaction sequence. It contains the logic to generate and control the application of sequence items to the DUT. Sequences are essential for stimulus generation.



```
class sequence_ extends uvm_sequence #(seq_item);
  `uvm_object_utils(sequence_)
   `uvm_declare_p_sequencer(sequencer)

  function new(string name = "sequence_");
      super.new(name);
    endfunction

  //Creating the sequence
  virtual task body();
    repeat(5) begin
      req=seq_item::type_id::create("req");
      start_item(req);
      assert (req.randomize());
      finish_item(req);
    end
  endtask
endclass
```

7. Sequenceitems.sv :

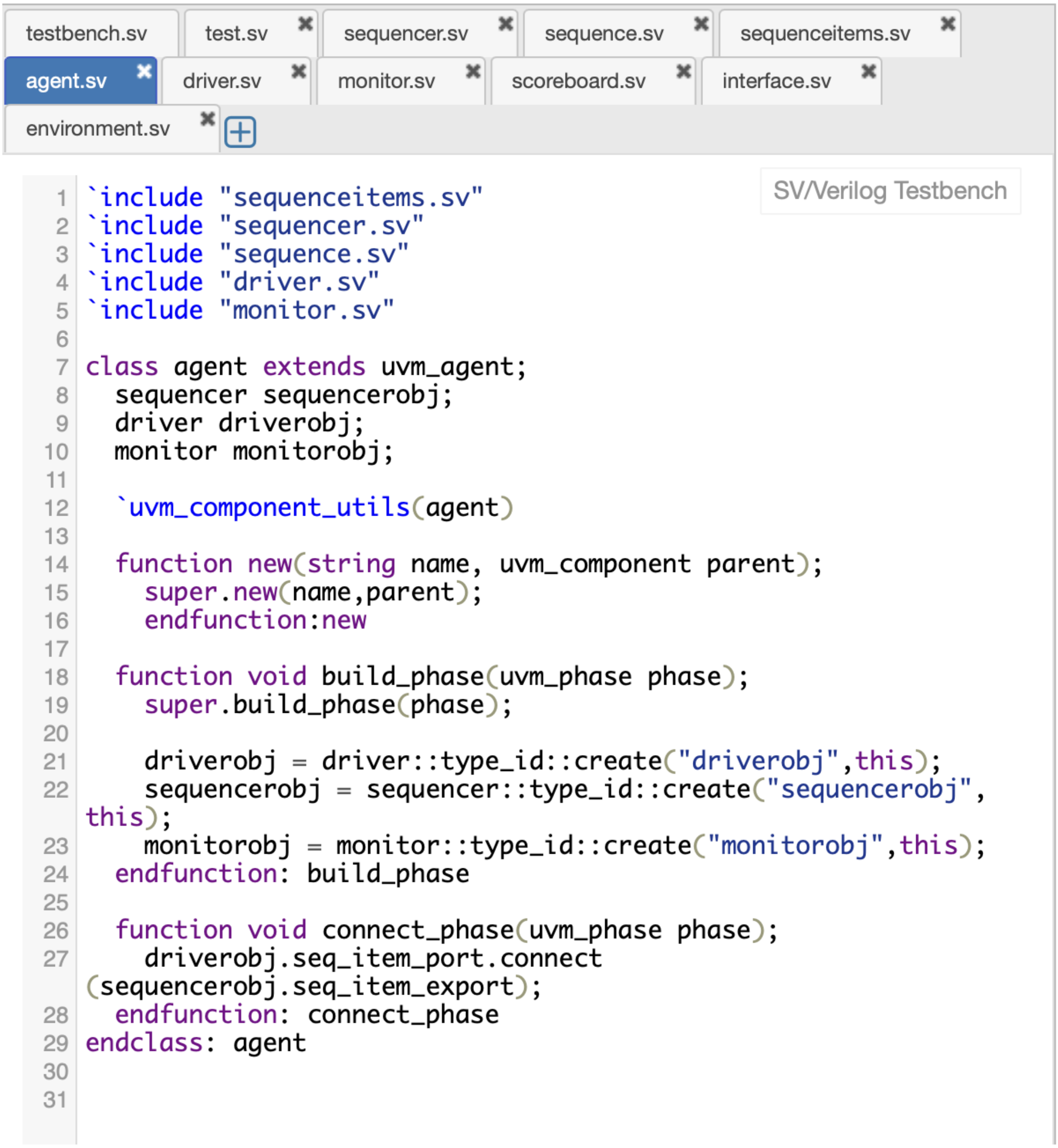
Significance: Contains the definitions of sequence items, which are individual transactions or data transfers applied to the DUT by sequences. Sequence items encapsulate specific data and properties to be transferred, and their definition is crucial for test scenario development.



```
class seq_item extends uvm_sequence_item;

  `uvm_object_utils_begin(seq_item)
  `uvm_field_int(a,UVM_ALL_ON)
  `uvm_field_int(b,UVM_ALL_ON)
  `uvm_field_int(sum,UVM_ALL_ON)
  `uvm_field_int(nst,UVM_ALL_ON)
  `uvm_field_int(cst,UVM_ALL_ON)
  `uvm_object_utils_end

  rand bit[1:0] a;
  rand bit[1:0] b;
  bit [2:0] sum;
  bit [1:0] nst;
  bit [1:0] cst;

  function new(string name= "seq_item");
    super.new(name);
  endfunction

  constraint a1 {a>=0; a<=1;}
  constraint a2 {b>=0; b<=1;}

endclass
```

8. agent.sv:

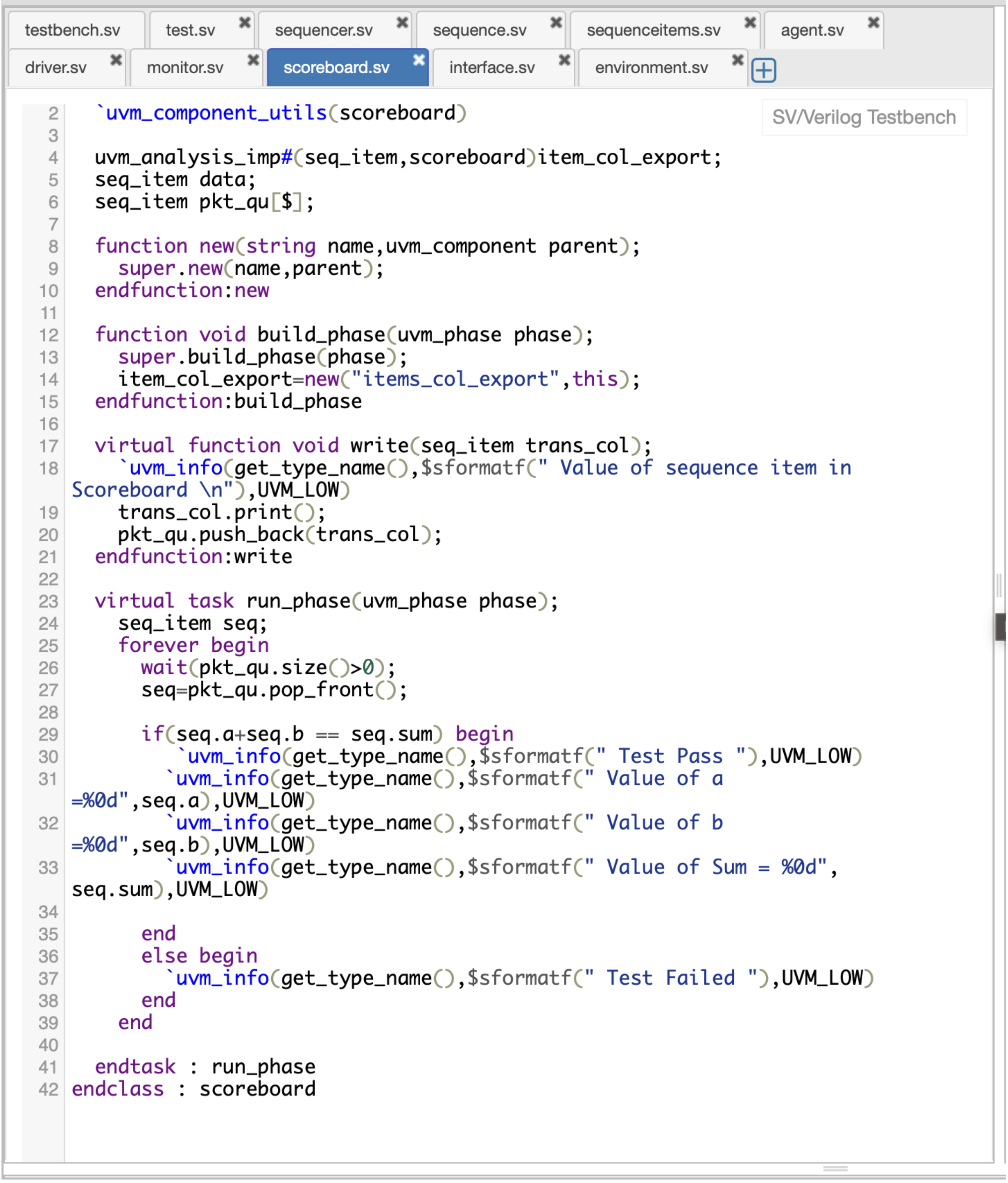
- Significance: The agent.sv file typically combines the sequencer, driver, and monitor into a cohesive unit. It encapsulates the components responsible for interacting with the DUT and observing its behavior.



```
`include "sequenceitems.sv"
`include "sequencer.sv"
`include "sequence.sv"
`include "driver.sv"
`include "monitor.sv"

class agent extends uvm_agent;
  sequencer sequencerobj;
  driver driverobj;
  monitor monitorobj;

  `uvm_component_utils(agent)

  function new(string name, uvm_component parent);
    super.new(name,parent);
    endfunction:new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    driverobj = driver::type_id::create("driverobj",this);
    sequencerobj = sequencer::type_id::create("sequencerobj",this);
    monitorobj = monitor::type_id::create("monitorobj",this);
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    driverobj.seq_item_port.connect(sequencerobj.seq_item_export);
  endfunction: connect_phase
endclass: agent
```

9. scoreboard.sv::

- Significance: The scoreboard.sv file is responsible for comparing the DUT's outputs to expected results or reference models. It detects errors or discrepancies and generates reports to highlight any issues.

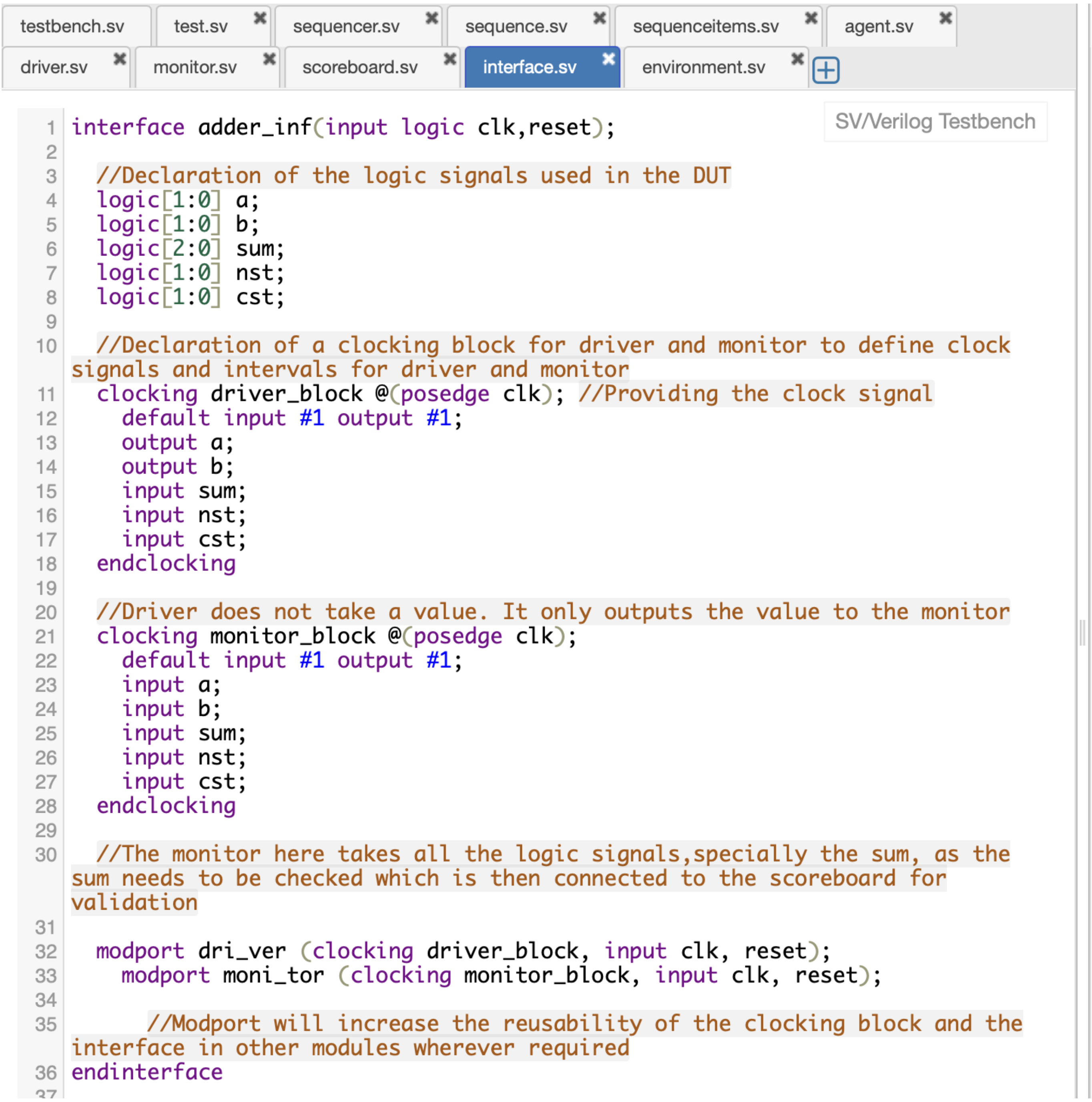


```
  `uvm_component_utils(scoreboard)

  uvm_analysis_imp#(seq_item,scoreboard)item_col_export;
  seq_item data;
  seq_item pkt_qu[$];

  function new(string name,uvm_component parent);
    super.new(name,parent);
  endfunction:new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    item_col_export=new("items_col_export",this);
  endfunction:build_phase

  virtual function void write(seq_item trans_col);
    `uvm_info(get_type_name(),$sformatf(" Value of sequence item in Scoreboard \n"),UVM_LOW)
    trans_col.print();
    pkt_qu.push_back(trans_col);
  endfunction:write

  virtual task run_phase(uvm_phase phase);
    seq_item seq;
    forever begin
      wait(pkt_qu.size()>0);
      seq=pkt_qu.pop_front();

      if(seq.a+seq.b == seq.sum) begin
         `uvm_info(get_type_name(),$sformatf(" Test Pass "),UVM_LOW)
        `uvm_info(get_type_name(),$sformatf(" Value of a =%0d",seq.a),UVM_LOW)
        `uvm_info(get_type_name(),$sformatf(" Value of b =%0d",seq.b),UVM_LOW)
        `uvm_info(get_type_name(),$sformatf(" Value of Sum = %0d", seq.sum),UVM_LOW)

      end
      else begin
        `uvm_info(get_type_name(),$sformatf(" Test Failed "),UVM_LOW)
      end
    end

  endtask : run_phase
endclass : scoreboard
```

1

10. Interface.sv:

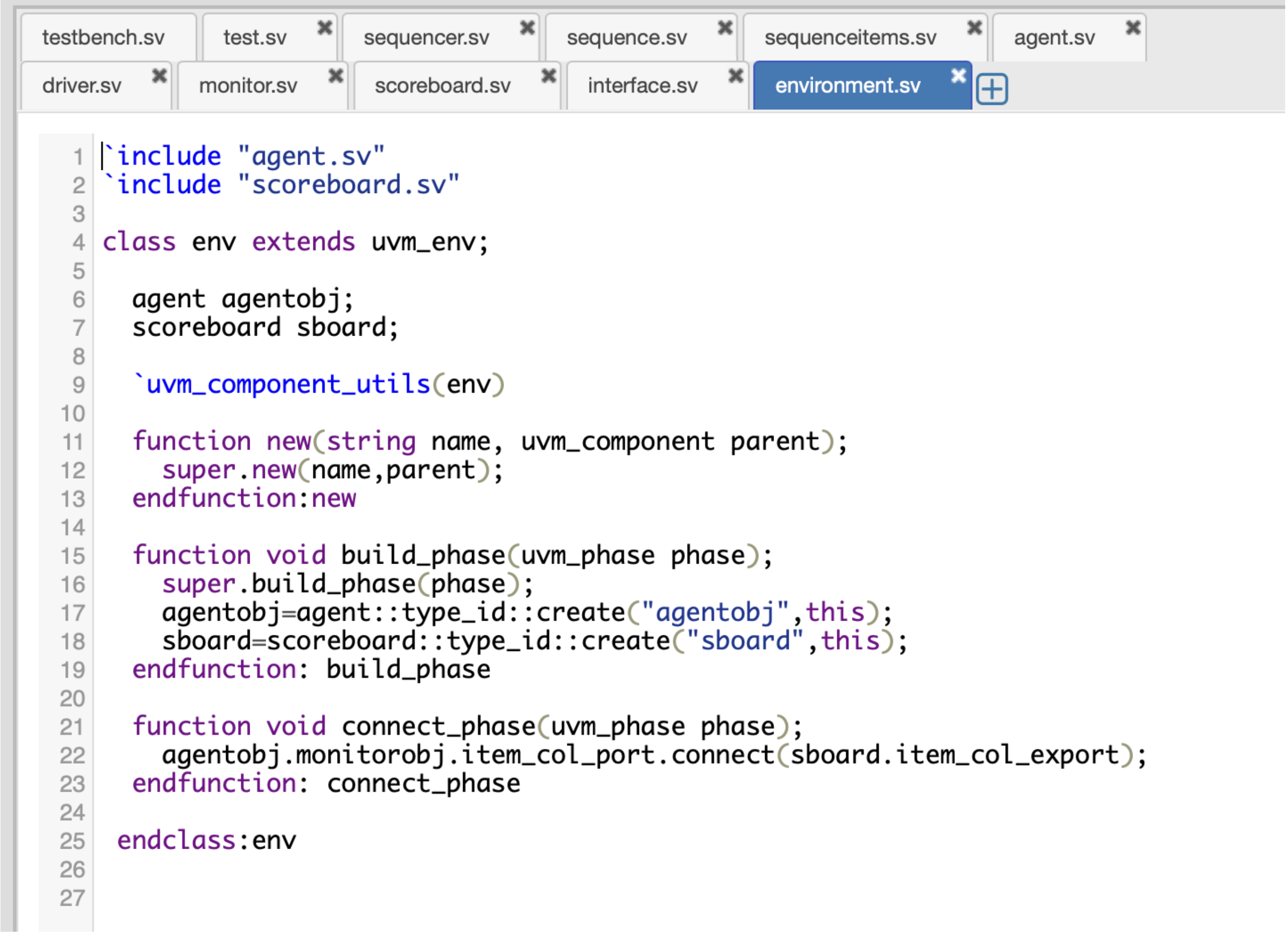
- Significance: The interface.sv file represents the communication protocol or interface used to connect the testbench to the DUT. It defines the pins, signals, and protocols used for communication.



```
interface adder_inf(input logic clk,reset);

  //Declaration of the logic signals used in the DUT
  logic[1:0] a;
  logic[1:0] b;
  logic[2:0] sum;
  logic[1:0] nst;
  logic[1:0] cst;

  //Declaration of a clocking block for driver and monitor to define clock signals and intervals for driver and monitor
  clocking driver_block @(posedge clk); //Providing the clock signal
    default input #1 output #1;
    output a;
    output b;
    input sum;
    input nst;
    input cst;
  endclocking

  //Driver does not take a value. It only outputs the value to the monitor
  clocking monitor_block @(posedge clk);
    default input #1 output #1;
    input a;
    input b;
    input sum;
    input nst;
    input cst;
  endclocking

  //The monitor here takes all the logic signals,specially the sum, as the sum needs to be checked which is then connected to the scoreboard for validation

  modport dri_ver (clocking driver_block, input clk, reset);
    modport moni_tor (clocking monitor_block, input clk, reset);

      //Modport will increase the reusability of the clocking block and the interface in other modules wherever required
endinterface
```

11. environment.sv:

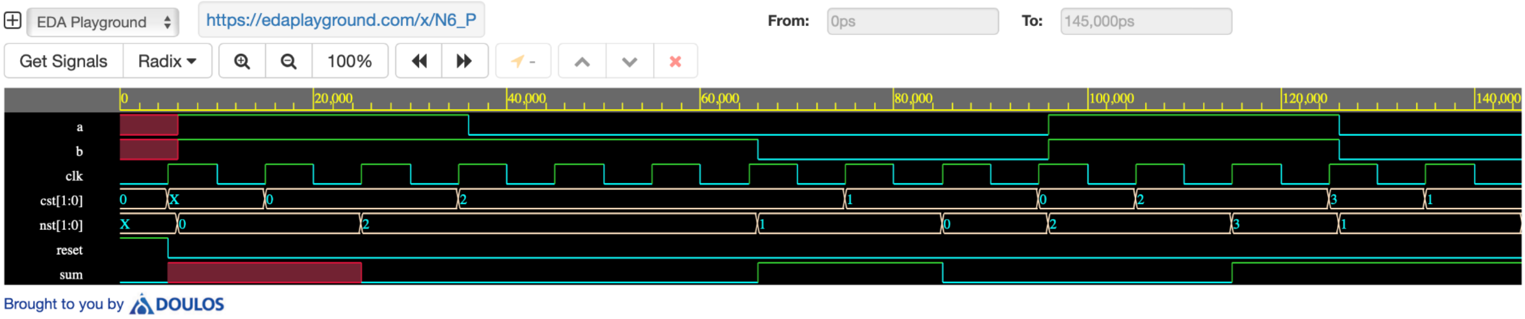
- Significance: The environment.sv file is where the overall testbench environment is set up. It includes the configuration of clocks, resets, and the integration of various agents and components. It creates the context for executing tests.



```
`include "agent.sv"
`include "scoreboard.sv"

class env extends uvm_env;

  agent agentobj;
  scoreboard sboard;

  `uvm_component_utils(env)

  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction:new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agentobj=agent::type_id::create("agentobj",this);
    sboard=scoreboard::type_id::create("sboard",this);
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    agentobj.monitorobj.item_col_port.connect(sboard.item_col_export);
  endfunction: connect_phase

 endclass:env


```

The significance of these components lies in their collective role in verifying a Design Under Test (DUT). Together, they enable stimulus generation, DUT observation, error checking, and coverage tracking, ultimately ensuring that the DUT functions correctly according to the specified requirements. By dividing these responsibilities into separate components, UVM promotes modular and scalable testbench development, making it easier to reuse, extend, and maintain verification environments.

Output Waveform:


EDA Playground
https://edaplayground.com/x/N6_P
From: 0ps
To: 145,000ps
Get Signals
Radix
a
b
clk
cst[1:0]
nst[1:0]
reset
sum
Brought to you by DOULOS