DESIGN & VERIFICATION OF APB PROTOCOL

Aakash Gupta
Linkedin | Github

APB

- **A**dvanced **P**eripheral **B**us protocol
- One of the protocols of AMBA1
- Simple non-pipelined protocol
- Used for connecting peripherals with low-bandwidth requirements
- Optimized for minimal power consumption
- Single Master and multiple Slaves
- Buses can be upto 32 bits wide
- Separate buses for read and write but no separate handshake signals. So, data transfer cannot occur on both buses at the same time

Evolution of APB

- **AMBA 2 APB specification**
  - Referred to as APB2
  - Defines 32 bit read and write transfers
- **AMBA 3 APB Protocol specification 1.0**
  - Referred to as APB3
  - Defines Wait States(PREADY) and Error Reporting(PSLVERR)
- **AMBA 4 APB Protocol specification 2.0**
  - Referred to as APB4
  - Defines Transaction Protection(PPROT) and Sparse Data Transfer(PSTRB)

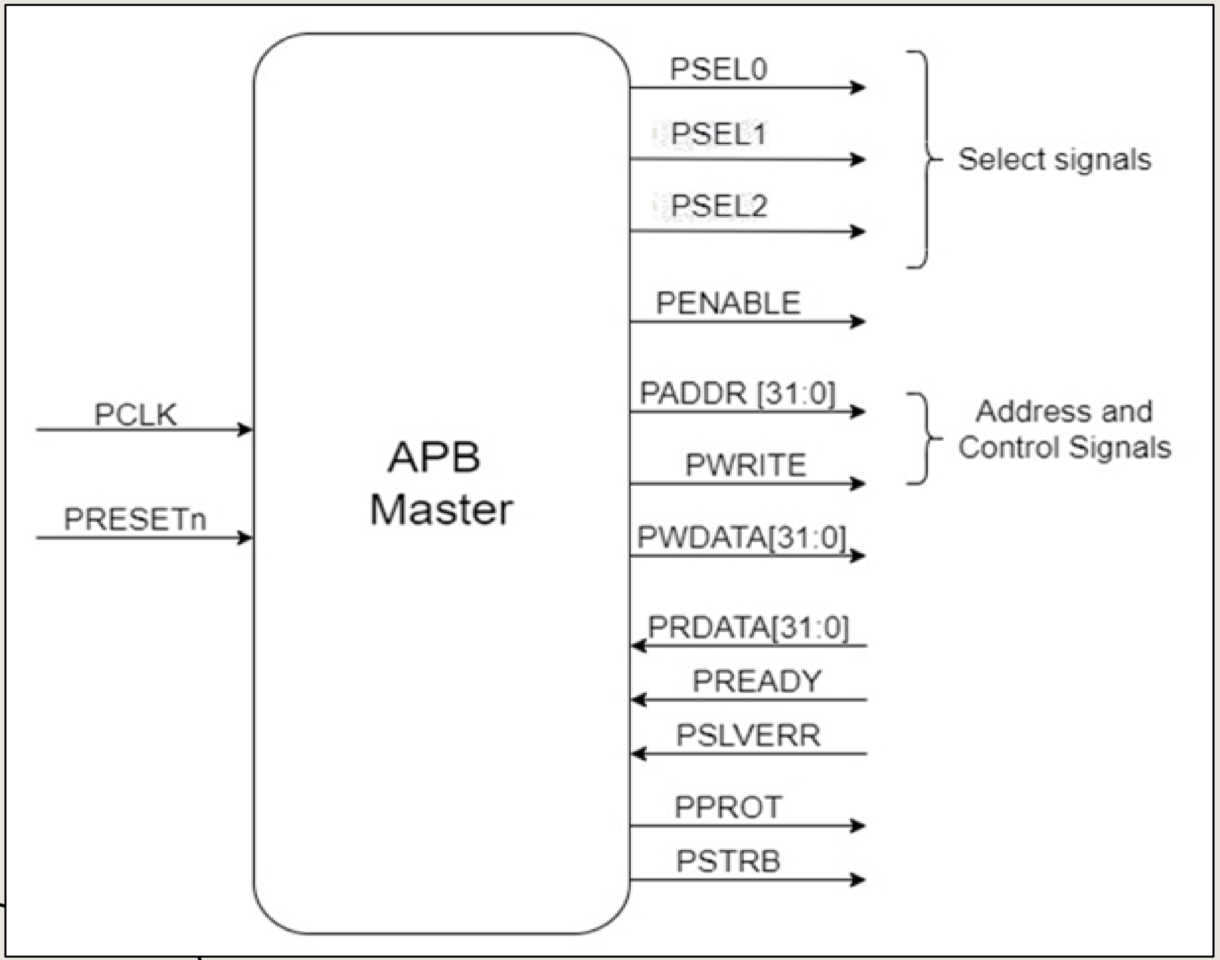
APB:Signals (1)

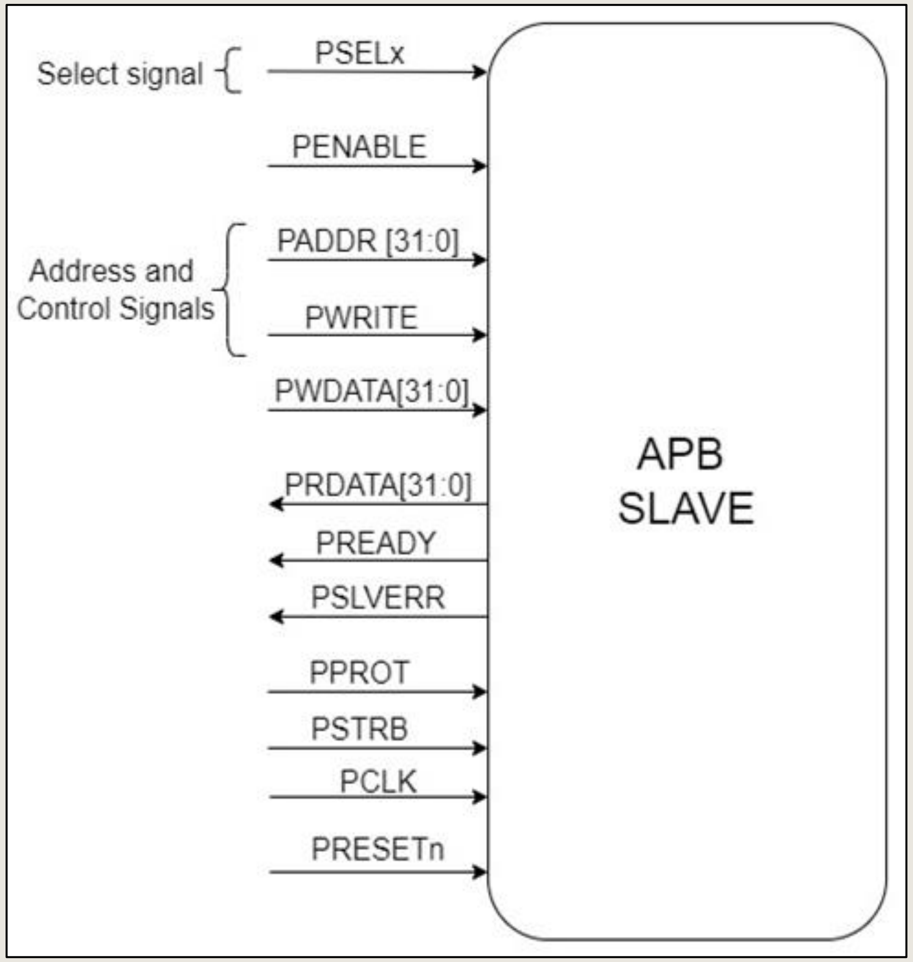
SIGNAL	DIRECTION	DESCRIPTION
PCLK	Clock source → All APB Blocks	Common clock which times all the transfers on the APB at its positive edge
PRESETn	Reset source → All APB Blocks	Common active LOW reset signal that resets the system bus
PSELx	Master → Slave	Each slave has a select signal. Indicates which slave is selected
PADDR	Master → Slave	32 bits wide Address Bus
PENABLE	Master → Slave	Enable signal that indicates 2nd cycle of a transfer
PWRITE	Master → Slave	HIGH ⇒ Write transfer LOW ⇒ Read transfer

APB:Signals (2)

SIGNAL	DIRECTION	DESCRIPTION
PWDATA	Master → Slave	32 bits wide write data bus which is driven during write cycles when PWRITE is HIGH
PRDATA	Slave → Master	32 bits wide read data bus which is driven during read cycles when PWRITE is LOW
PREADY	Slave → Master	Indicates if the slave is ready for the completion of a transfer. Used from APB3 and onwards
PSLVERR	Slave → Master	Indicates failure of a transfer when asserted HIGH. Used from APB3 and onwards
PPROT	Master → Slave	3 bit bus which indicates protection level of the transaction. Used in APB4
PSTRB	Master → Slave	Byte strobe for write operations. Used in APB4

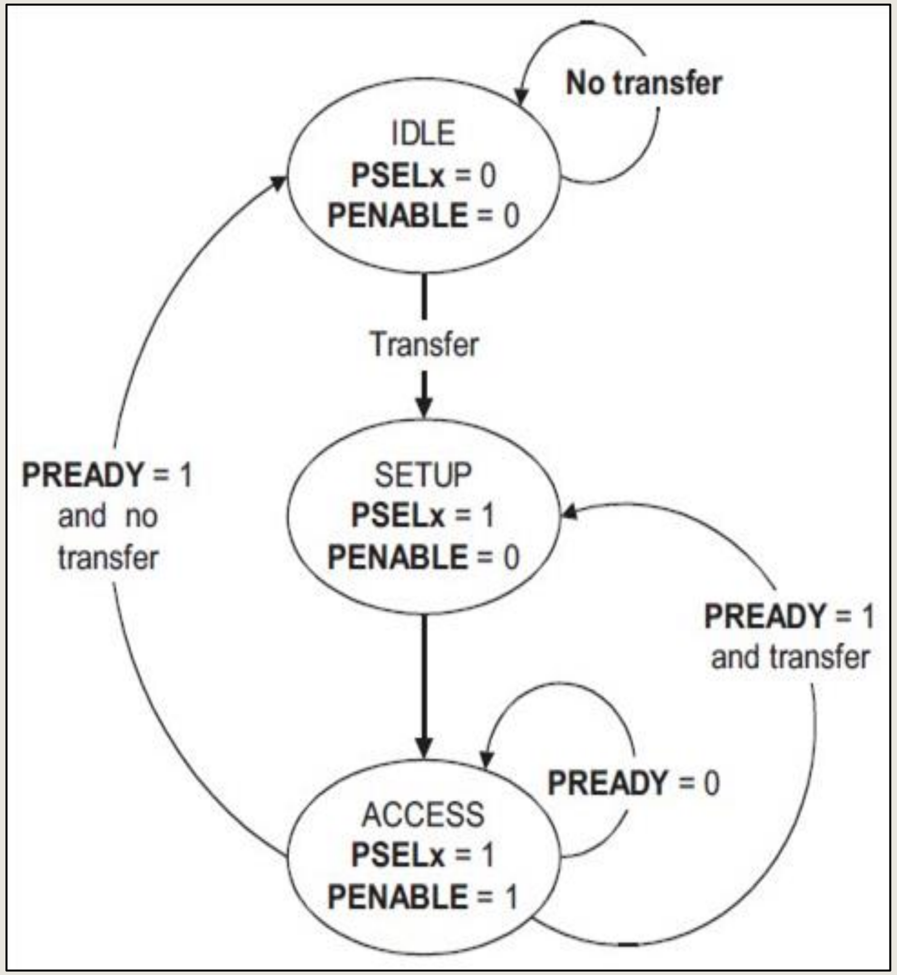
APB:Master & Slave





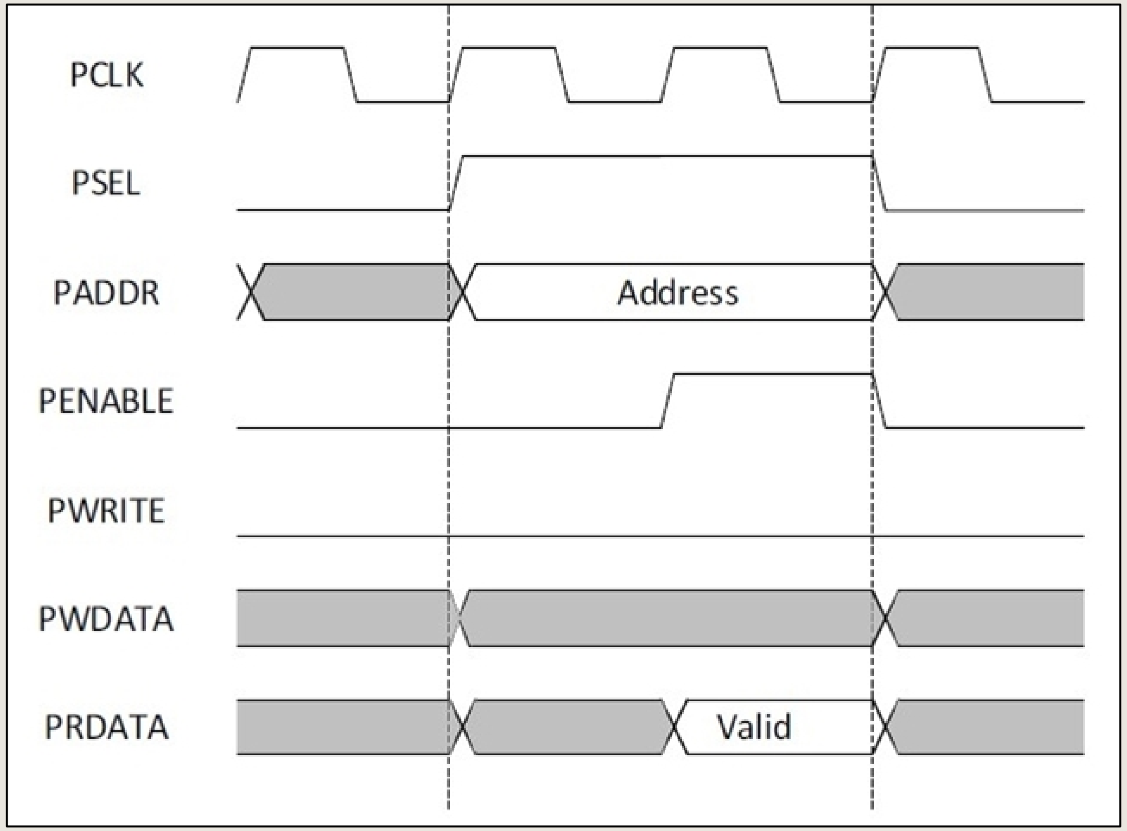
APB:Operating states

- **IDLE**
  - Default state of APB
- **SETUP**
  - When a transfer is necessary, bus moves to SETUP state and selects the appropriate slave by asserting PSELx as HIGH.
  - Bus remains in this state for 1 clock cycle and moves to ACCESS state on the next rising edge of the clock.
  - During this transition, PADDR, PSELx, PWRITE and PWDATA must be stable.
- **ACCESS**
  - PENABLE is asserted HIGH. Depending on the value of PREADY, the next state is decided.

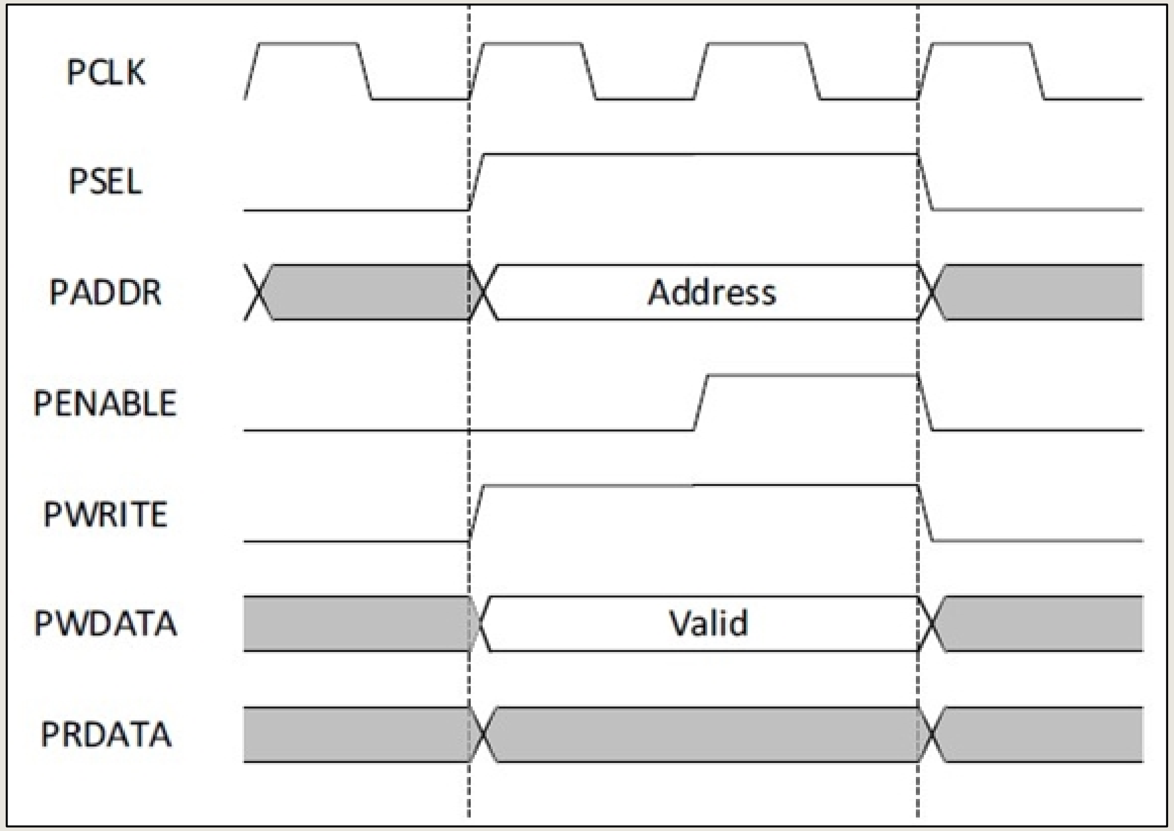


Ref : AMBA APB Protocol Specification Version 2.0

APB:Transfers (1)

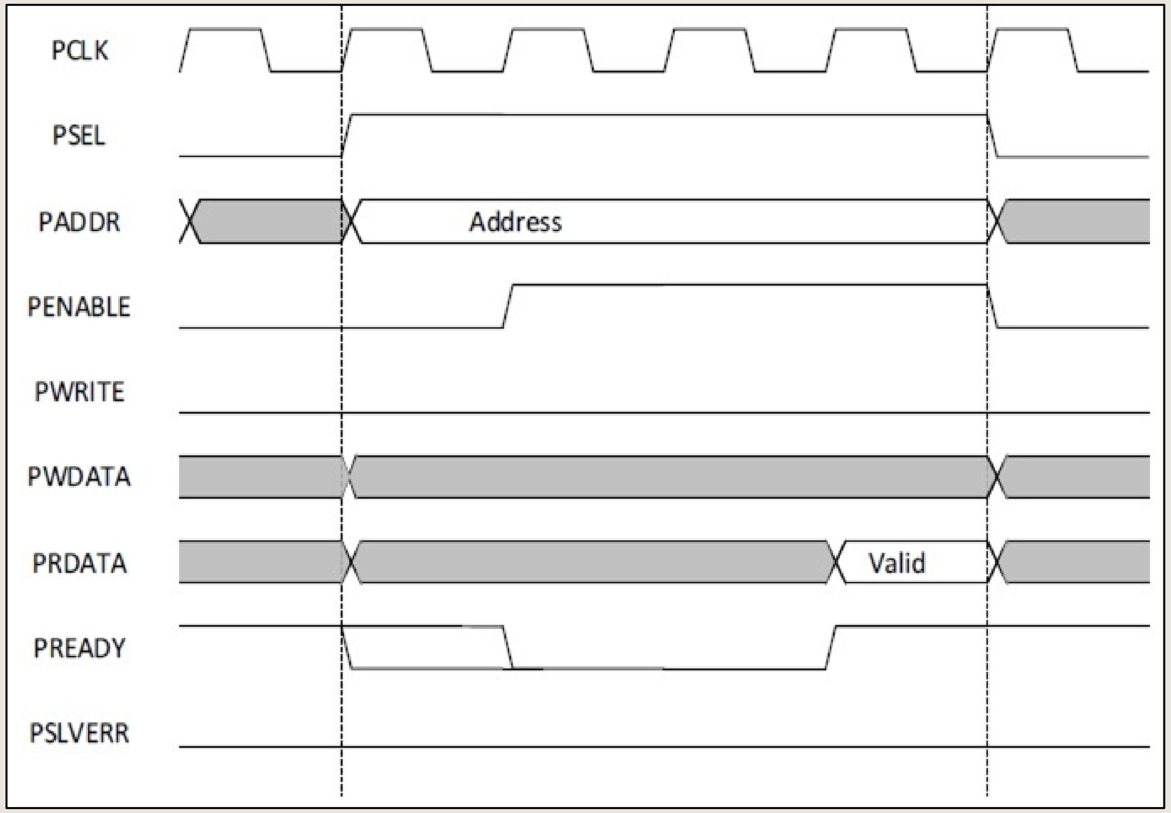


APB2 Read transfer

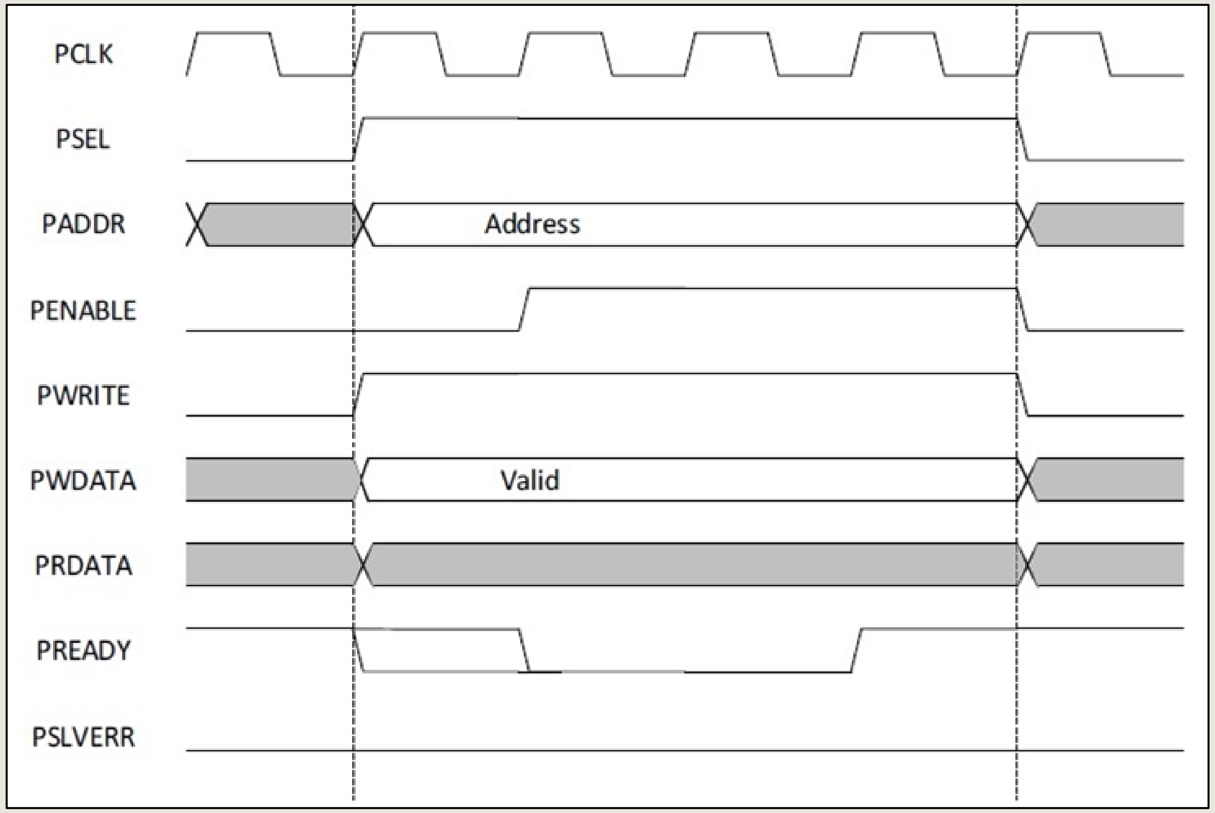


APB2 Write transfer

APB:Transfers (2)

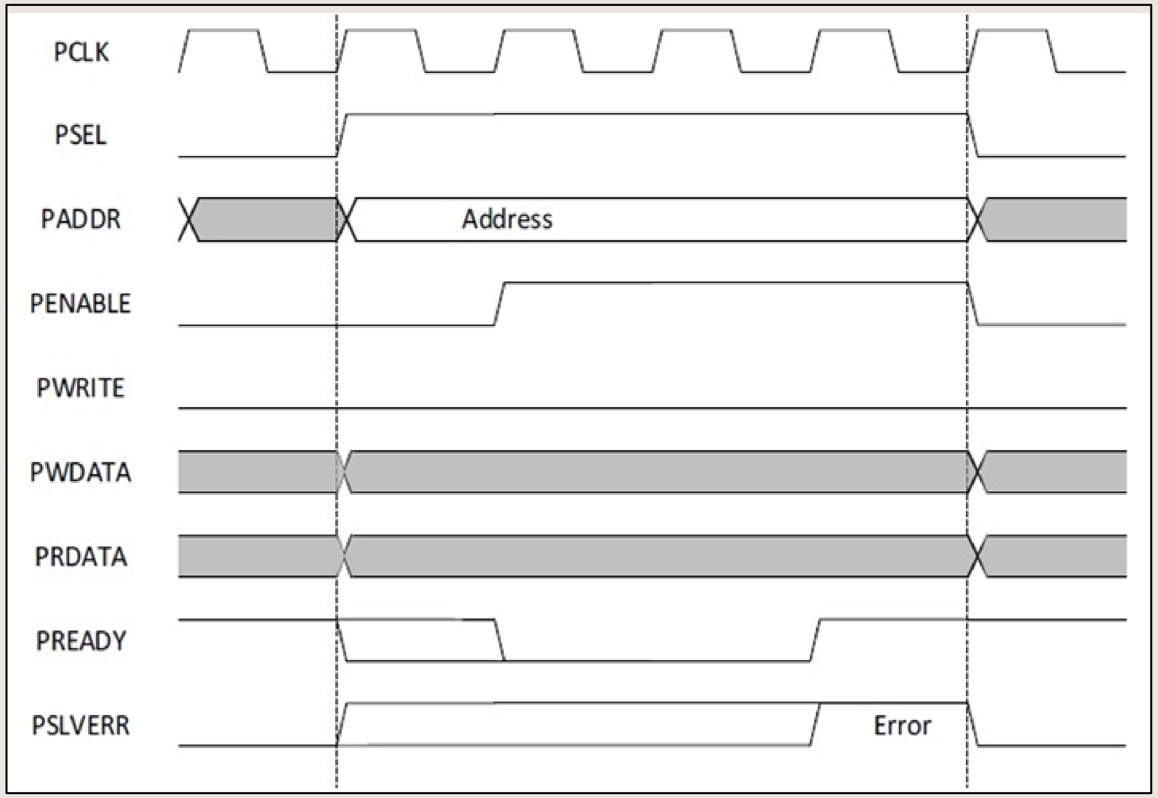


APB Read transfer with wait states

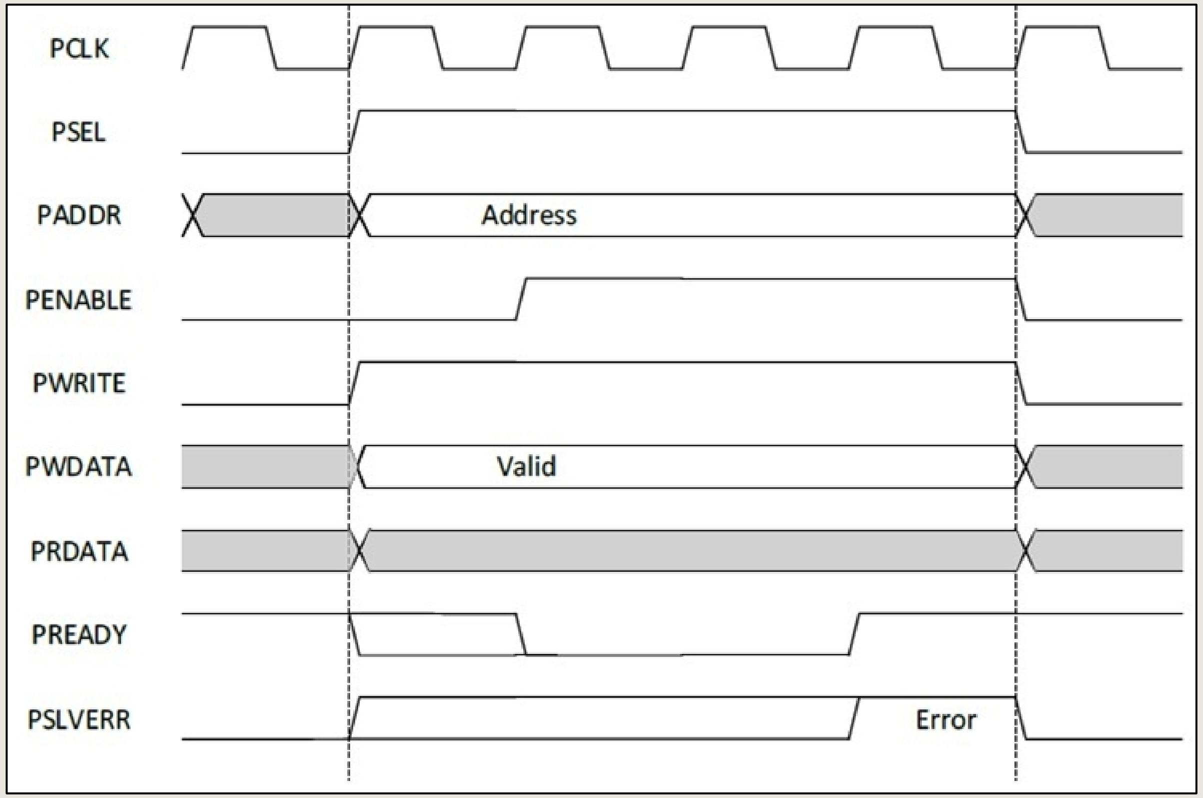


APB Write transfer with wait states

APB:Transfers (3)



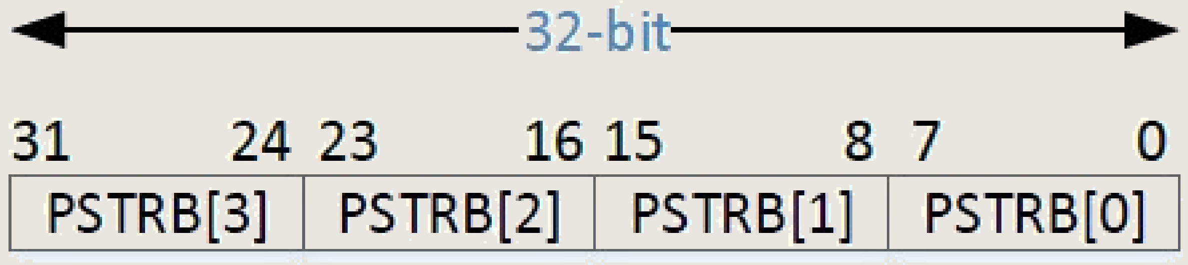
APB Read transfer with wait states and error response



APB Write transfer with wait states and error response

APB Write Strobes

- Enables sparse data transfer on write data bus
- 4-bit signal PSTRB[3:0] - one bit per byte of PWDATA
- Used only for write operations and ignored for read operations



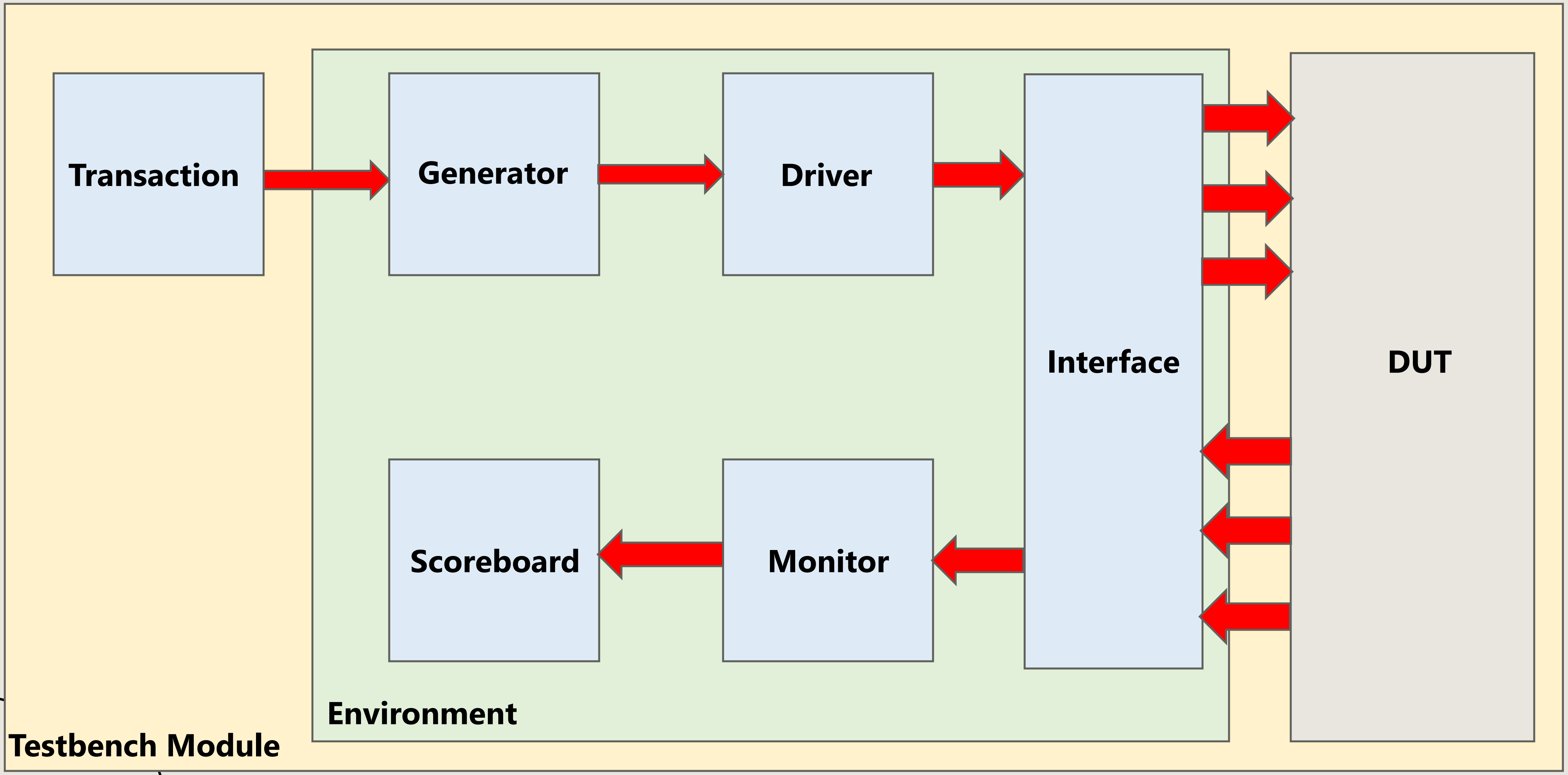
Byte lane mapping

APB Protection Unit

❑ 3-bit signal PPROT[2:0] to indicate protection level of a transaction.

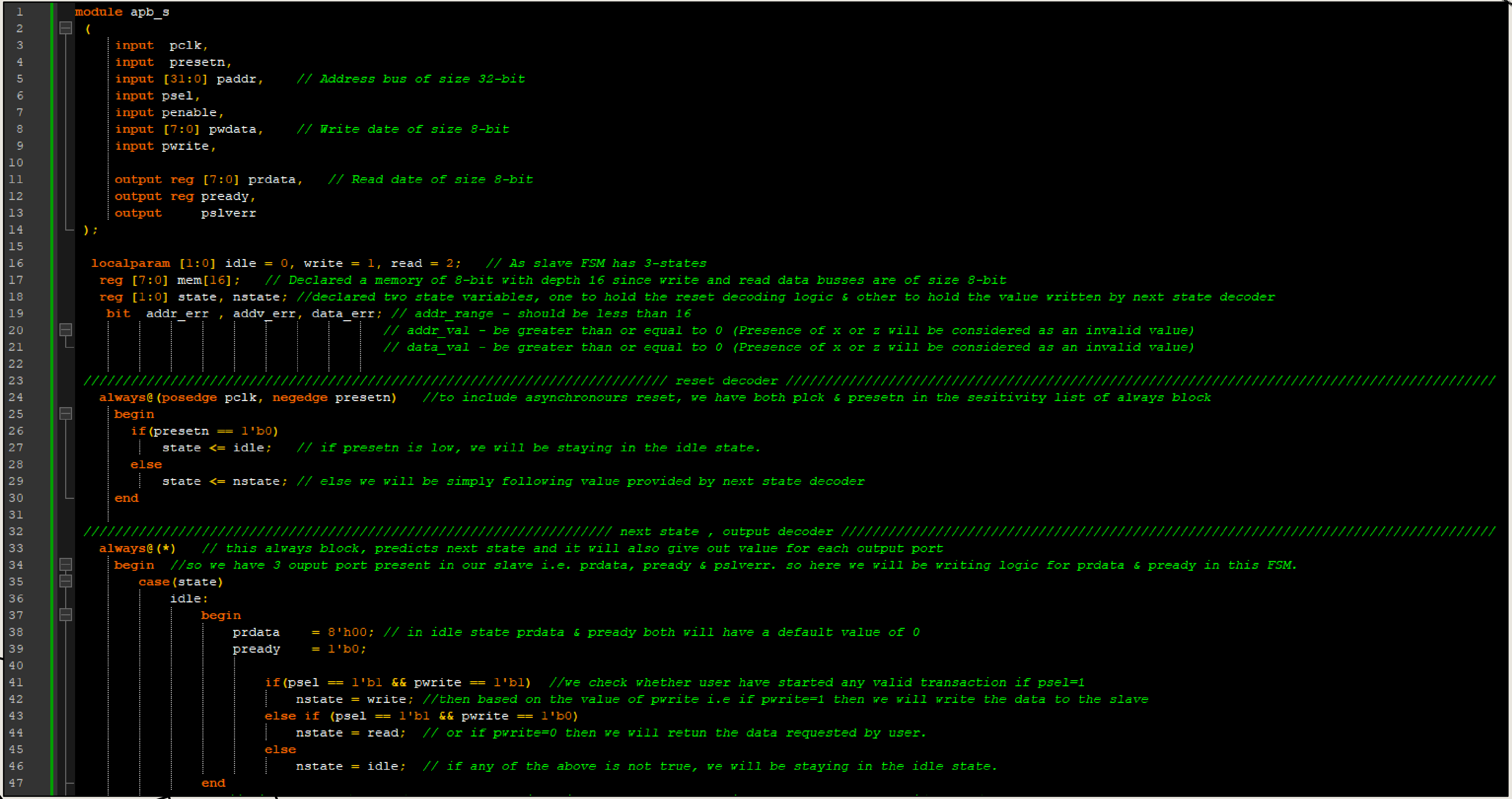
Signal	**When LOW**	**When HIGH**
PPROT[0]	Non-privileged access	Privileged access
PPROT[1]	Secure access	Non-secure access
PPROT[2]	Data access	Instruction access

Design & Verification Enviornment



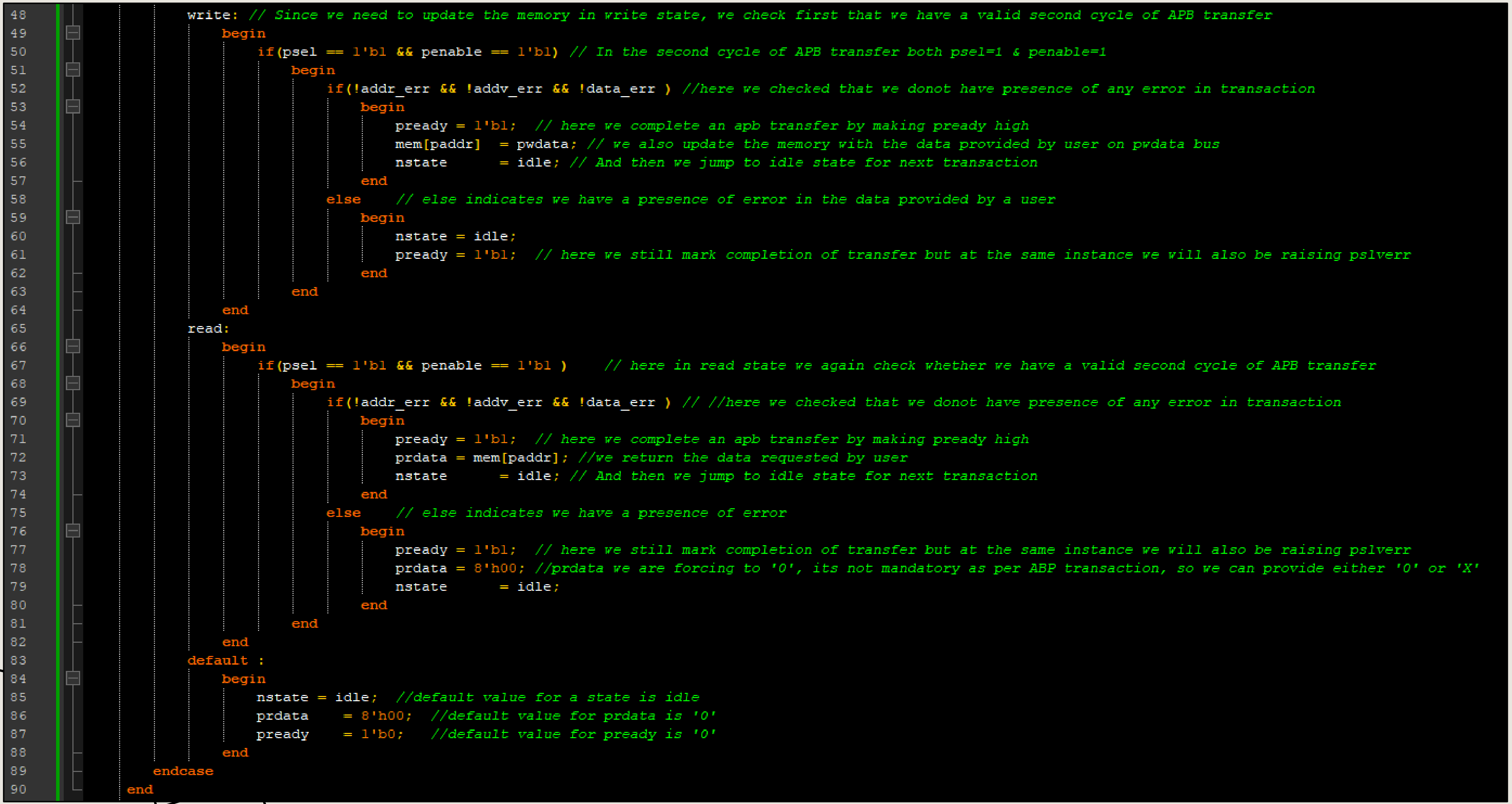
In the upcoming slides, we will see all the blocks of design & verification environment of APB Protocol

Design/DUT (1)



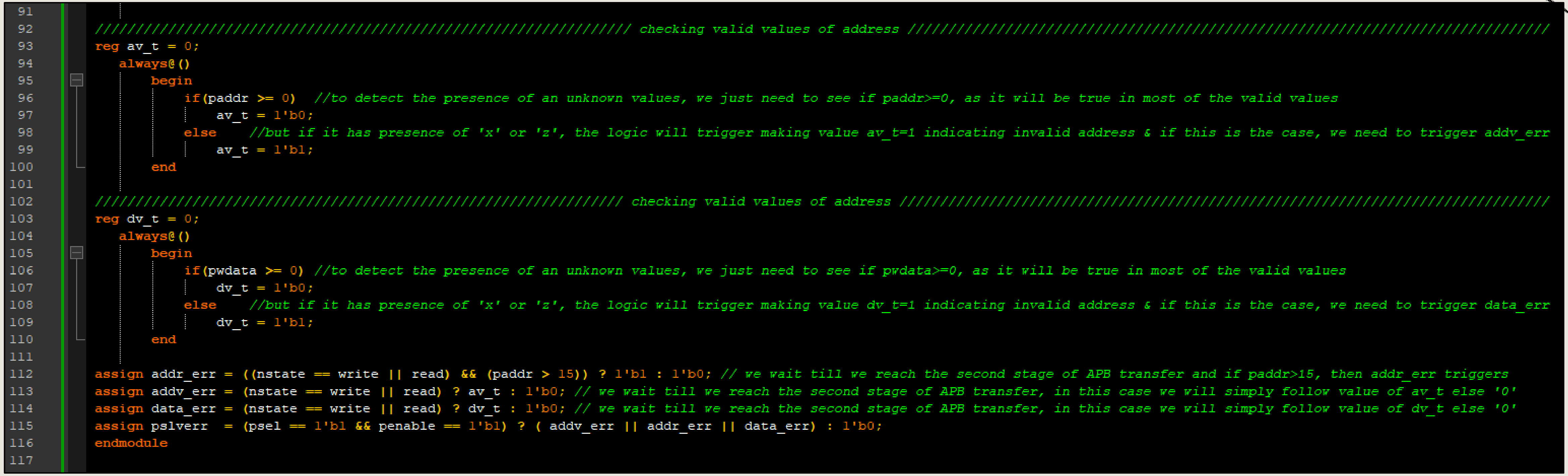
```verilog
module apb_s
 (
    input  pclk,
    input  presetn,
    input [31:0] paddr,    // Address bus of size 32-bit
    input psel,
    input penable,
    input [7:0] pwdata,    // Write date of size 8-bit
    input pwrite,

    output reg [7:0] prdata,   // Read date of size 8-bit
    output reg pready,
    output     pslverr
 );

  localparam [1:0] idle = 0, write = 1, read = 2;   // As slave FSM has 3-states
   reg [7:0] mem[16];   // Declared a memory of 8-bit with depth 16 since write and read data busses are of size 8-bit
   reg [1:0] state, nstate; //declared two state variables, one to hold the reset decoding logic & other to hold the value written by next state decoder
    bit  addr_err , addv_err, data_err; // addr_range - should be less than 16
                                        // addr_val - be greater than or equal to 0 (Presence of x or z will be considered as an invalid value)
                                        // data_val - be greater than or equal to 0 (Presence of x or z will be considered as an invalid value)

 ////////////////////////////////////////////////////////////////////////// reset decoder //////////////////////////////////////////////////////////////////////////////////////////
   always@(posedge pclk, negedge presetn)   //to include asynchronours reset, we have both plck & presetn in the sesitivity list of always block
     begin
       if(presetn == 1'b0)
           state <= idle;   // if presetn is low, we will be staying in the idle state.
       else
           state <= nstate; // else we will be simply following value provided by next state decoder
     end

 ///////////////////////////////////////////////////////////////////// next state , output decoder ////////////////////////////////////////////////////////////////////////////////
   always@(*)   // this always block, predicts next state and it will also give out value for each output port
     begin  //so we have 3 ouput port present in our slave i.e. prdata, pready & pslverr. so here we will be writing logic for prdata & pready in this FSM.
        case(state)
            idle:
                begin
                    prdata    = 8'h00; // in idle state prdata & pready both will have a default value of 0
                    pready    = 1'b0;

                        if(psel == 1'b1 && pwrite == 1'b1)  //we check whether user have started any valid transaction if psel=1
                            nstate = write; //then based on the value of pwrite i.e if pwrite=1 then we will write the data to the slave
                        else if (psel == 1'b1 && pwrite == 1'b0)
                            nstate = read;  // or if pwrite=0 then we will retun the data requested by user.
                        else
                            nstate = idle;  // if any of the above is not true, we will be staying in the idle state.
                end
```

Design/DUT (2)



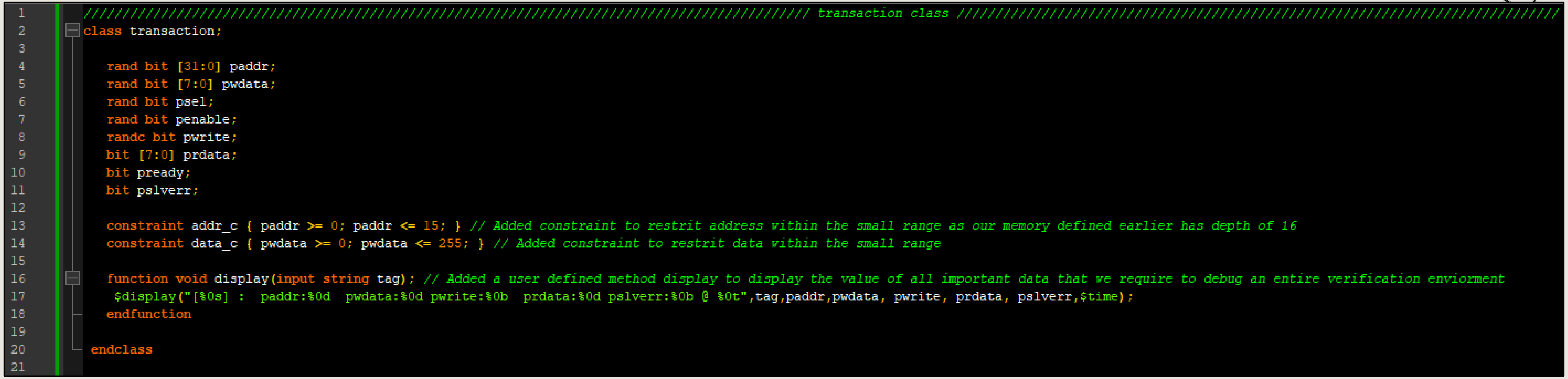
```
       write: // Since we need to update the memory in write state, we check first that we have a valid second cycle of APB transfer
           begin
               if(psel == 1'b1 && penable == 1'b1) // In the second cycle of APB transfer both psel=1 & penable=1
                   begin
                       if(!addr_err && !addv_err && !data_err ) //here we checked that we donot have presence of any error in transaction
                           begin
                               pready = 1'b1;  // here we complete an apb transfer by making pready high
                               mem[paddr]  = pwdata; // we also update the memory with the data provided by user on pwdata bus
                               nstate      = idle; // And then we jump to idle state for next transaction
                           end
                       else    // else indicates we have a presence of error in the data provided by a user
                           begin
                               nstate = idle;
                               pready = 1'b1;  // here we still mark completion of transfer but at the same instance we will also be raising pslverr
                           end
                   end
           end
       read:
           begin
               if(psel == 1'b1 && penable == 1'b1 )    // here in read state we again check whether we have a valid second cycle of APB transfer
                   begin
                       if(!addr_err && !addv_err && !data_err ) // //here we checked that we donot have presence of any error in transaction
                           begin
                               pready = 1'b1;  // here we complete an apb transfer by making pready high
                               prdata = mem[paddr]; //we return the data requested by user
                               nstate      = idle; // And then we jump to idle state for next transaction
                           end
                       else    // else indicates we have a presence of error
                           begin
                               pready = 1'b1;  // here we still mark completion of transfer but at the same instance we will also be raising pslverr
                               prdata = 8'h00; //prdata we are forcing to '0', its not mandatory as per ABP transaction, so we can provide either '0' or 'X'
                               nstate      = idle;
                           end
                   end
           end
       default :
           begin
               nstate = idle;  //default value for a state is idle
               prdata    = 8'h00;  //default value for prdata is '0'
               pready    = 1'b0;   //default value for pready is '0'
           end
   endcase
end
```

Design/DUT (3)



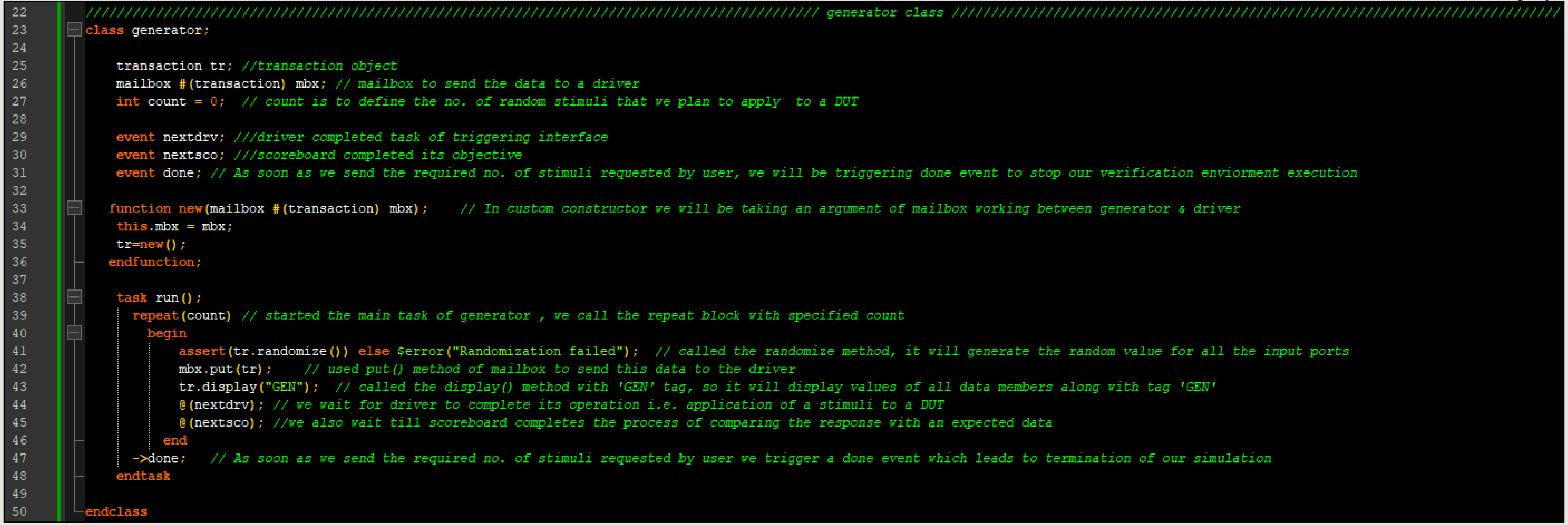
```

//////////////////////////////////////////////////////////////////// checking valid values of address ////////////////////////////////////////////////////////////////////////////////
reg av_t = 0;
   always@()
       begin
           if(paddr >= 0)  //to detect the presence of an unknown values, we just need to see if paddr>=0, as it will be true in most of the valid values
               av_t = 1'b0;
           else    //but if it has presence of 'x' or 'z', the logic will trigger making value av_t=1 indicating invalid address & if this is the case, we need to trigger addv_err
               av_t = 1'b1;
       end

/////////////////////////////////////////////////////////////////// checking valid values of address /////////////////////////////////////////////////////////////////////////////////
reg dv_t = 0;
   always@()
       begin
           if(pwdata >= 0) //to detect the presence of an unknown values, we just need to see if pwdata>=0, as it will be true in most of the valid values
               dv_t = 1'b0;
           else    //but if it has presence of 'x' or 'z', the logic will trigger making value dv_t=1 indicating invalid address & if this is the case, we need to trigger data_err
               dv_t = 1'b1;
       end

assign addr_err = ((nstate == write || read) && (paddr > 15)) ? 1'b1 : 1'b0; // we wait till we reach the second stage of APB transfer and if paddr>15, then addr_err triggers
assign addv_err = (nstate == write || read) ? av_t : 1'b0; // we wait till we reach the second stage of APB transfer, in this case we will simply follow value of av_t else '0'
assign data_err = (nstate == write || read) ? dv_t : 1'b0; // we wait till we reach the second stage of APB transfer, in this case we will simply follow value of dv_t else '0'
assign pslverr  = (psel == 1'b1 && penable == 1'b1) ? ( addv_err || addr_err || data_err) : 1'b0;
endmodule

```

Transaction



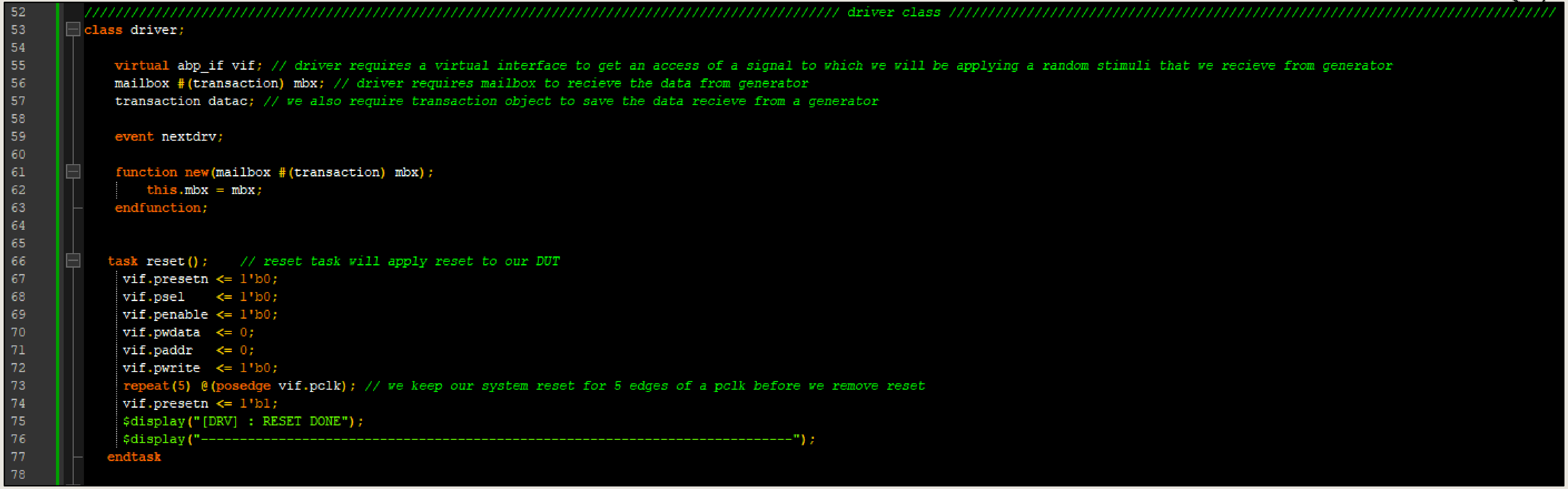
```
//////////////////////////////////////////////////////////////////////////////////////////////// transaction class //////////////////////////////////////////////////////////////////////////////
class transaction;

   rand bit [31:0] paddr;
   rand bit [7:0] pwdata;
   rand bit psel;
   rand bit penable;
   randc bit pwrite;
   bit [7:0] prdata;
   bit pready;
   bit pslverr;

   constraint addr_c { paddr >= 0; paddr <= 15; } // Added constraint to restrit address within the small range as our memory defined earlier has depth of 16
   constraint data_c { pwdata >= 0; pwdata <= 255; } // Added constraint to restrit data within the small range

   function void display(input string tag); // Added a user defined method display to display the value of all important data that we require to debug an entire verification enviorment
    $display("[%0s] :  paddr:%0d  pwdata:%0d pwrite:%0b  prdata:%0d pslverr:%0b @ %0t",tag,paddr,pwdata, pwrite, prdata, pslverr,$time);
   endfunction

 endclass
```

Generator



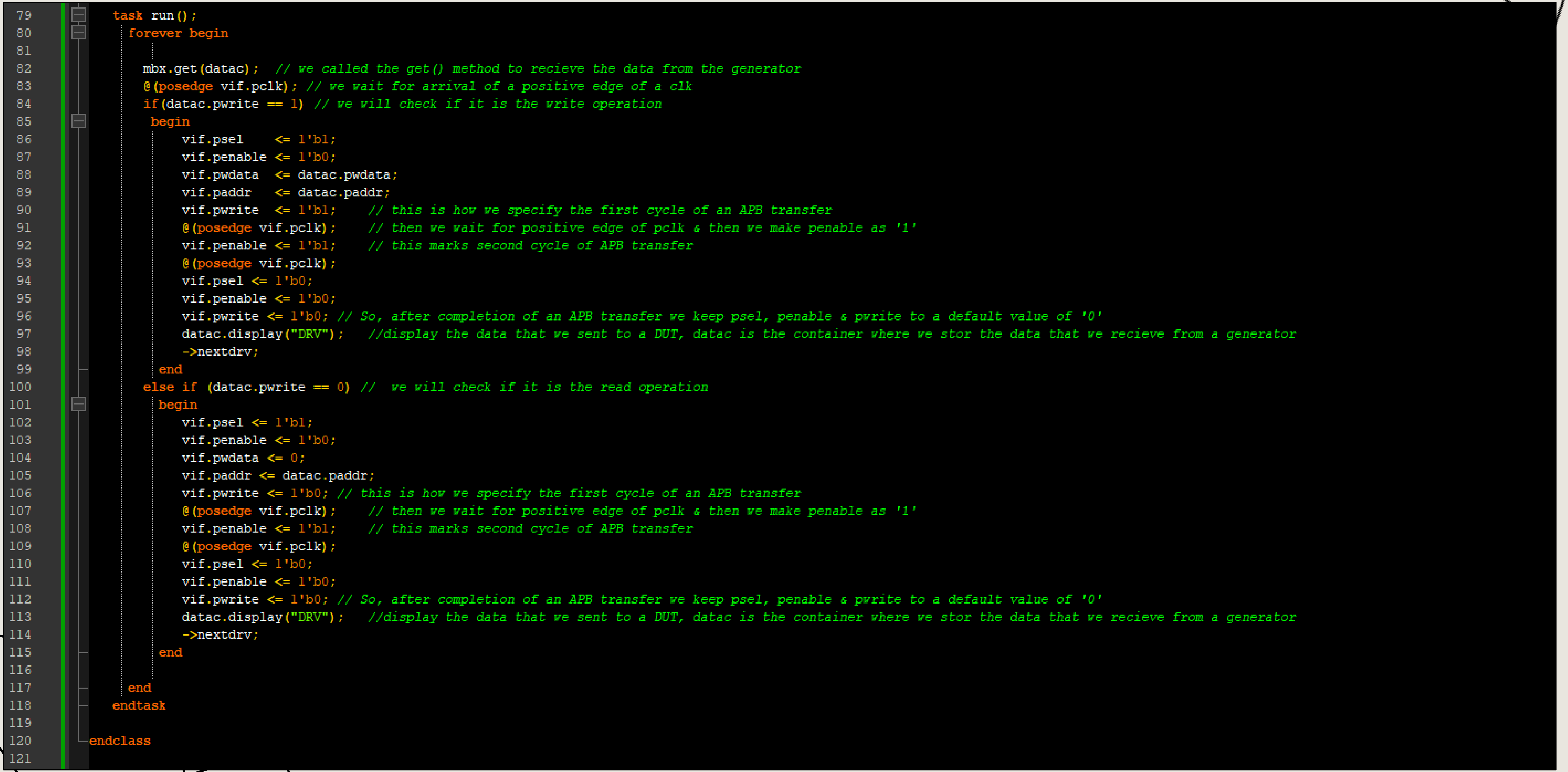
```
/////////////////////////////////////////////////////////////////////////////////////////////// generator class //////////////////////////////////////////////////////////////////////////////
class generator;

    transaction tr; //transaction object
    mailbox #(transaction) mbx; // mailbox to send the data to a driver
    int count = 0;  // count is to define the no. of random stimuli that we plan to apply  to a DUT

    event nextdrv; ///driver completed task of triggering interface
    event nextsco; ///scoreboard completed its objective
    event done; // As soon as we send the required no. of stimuli requested by user, we will be triggering done event to stop our verification enviorment execution

   function new(mailbox #(transaction) mbx);    // In custom constructor we will be taking an argument of mailbox working between generator & driver
    this.mbx = mbx;
    tr=new();
   endfunction;

    task run();
      repeat(count) // started the main task of generator , we call the repeat block with specified count
        begin
            assert(tr.randomize()) else $error("Randomization failed");  // called the randomize method, it will generate the random value for all the input ports
            mbx.put(tr);    // used put() method of mailbox to send this data to the driver
            tr.display("GEN");  // called the display() method with 'GEN' tag, so it will display values of all data members along with tag 'GEN'
            @(nextdrv); // we wait for driver to complete its operation i.e. application of a stimuli to a DUT
            @(nextsco); //we also wait till scoreboard completes the process of comparing the response with an expected data
          end
      ->done;   // As soon as we send the required no. of stimuli requested by user we trigger a done event which leads to termination of our simulation
    endtask

endclass
```

Driver (1)



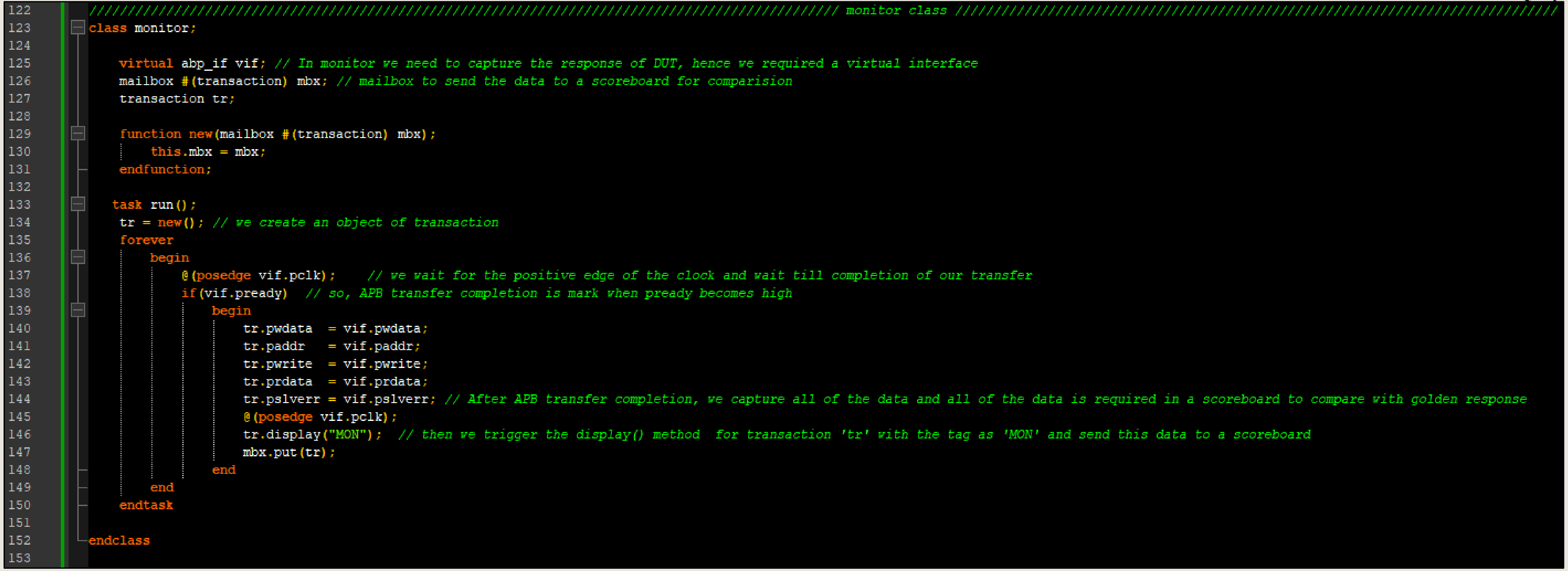
```
///////////////////////////////////////////////////////////////////////////////////////////////// driver class //////////////////////////////////////////////////////////////////////////////
class driver;

    virtual abp_if vif; // driver requires a virtual interface to get an access of a signal to which we will be applying a random stimuli that we recieve from generator
    mailbox #(transaction) mbx; // driver requires mailbox to recieve the data from generator
    transaction datac; // we also require transaction object to save the data recieve from a generator

    event nextdrv;

    function new(mailbox #(transaction) mbx);
        this.mbx = mbx;
    endfunction;


   task reset();    // reset task will apply reset to our DUT
     vif.presetn <= 1'b0;
     vif.psel    <= 1'b0;
     vif.penable <= 1'b0;
     vif.pwdata  <= 0;
     vif.paddr   <= 0;
     vif.pwrite  <= 1'b0;
     repeat(5) @(posedge vif.pclk); // we keep our system reset for 5 edges of a pclk before we remove reset
     vif.presetn <= 1'b1;
     $display("[DRV] : RESET DONE");
     $display("----------------------------------------------------------------------------");
   endtask
```

Driver (2)



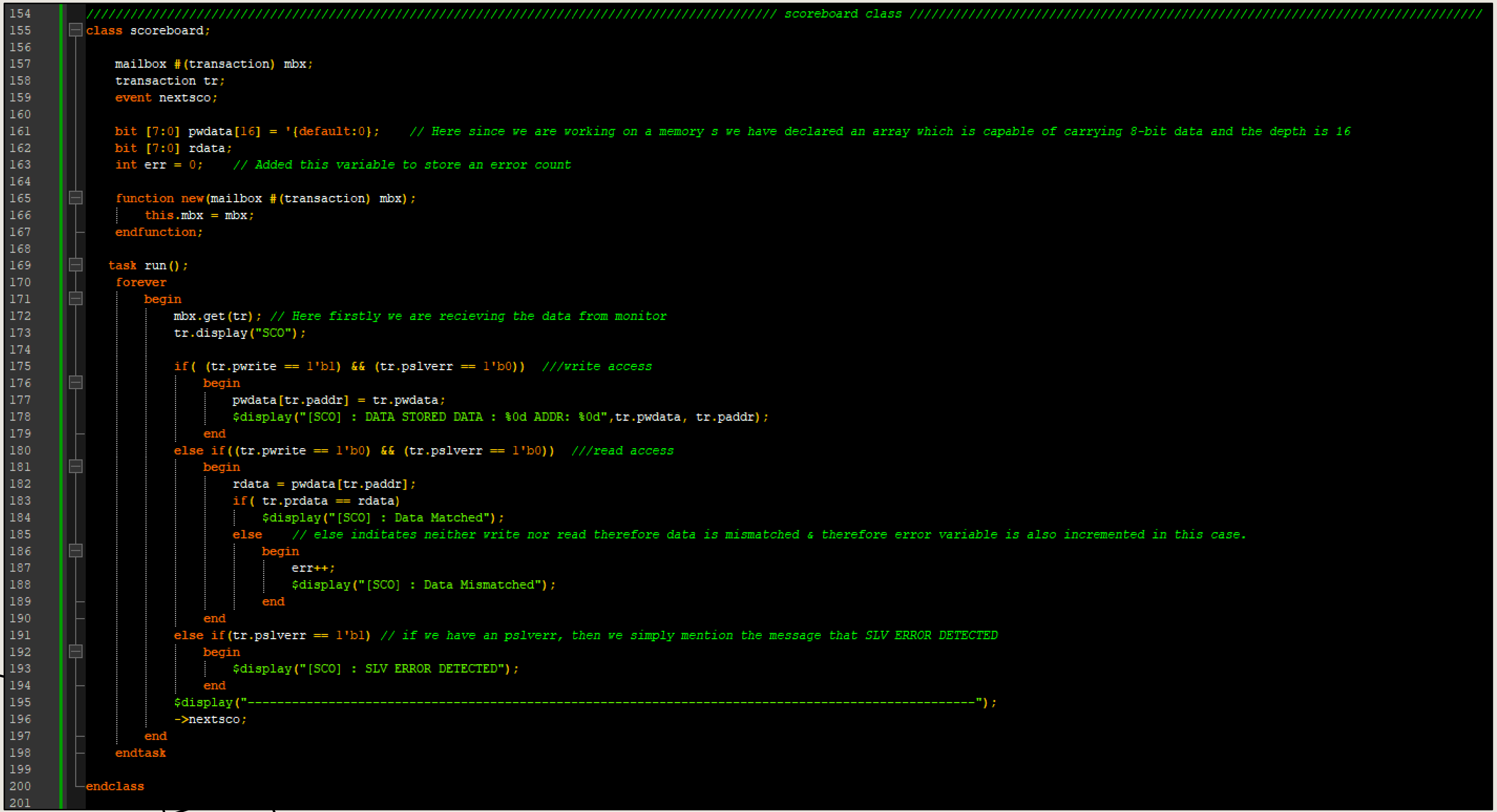
```
  task run();
   forever begin
   
     mbx.get(datac);  // we called the get() method to recieve the data from the generator
     @(posedge vif.pclk); // we wait for arrival of a positive edge of a clk
     if(datac.pwrite == 1) // we will check if it is the write operation
      begin
          vif.psel    <= 1'b1;
          vif.penable <= 1'b0;
          vif.pwdata  <= datac.pwdata;
          vif.paddr   <= datac.paddr;
          vif.pwrite  <= 1'b1;    // this is how we specify the first cycle of an APB transfer
          @(posedge vif.pclk);    // then we wait for positive edge of pclk & then we make penable as '1'
          vif.penable <= 1'b1;    // this marks second cycle of APB transfer
          @(posedge vif.pclk);
          vif.psel <= 1'b0;
          vif.penable <= 1'b0;
          vif.pwrite <= 1'b0; // So, after completion of an APB transfer we keep psel, penable & pwrite to a default value of '0'
          datac.display("DRV");   //display the data that we sent to a DUT, datac is the container where we stor the data that we recieve from a generator
          ->nextdrv;
       end
     else if (datac.pwrite == 0) //  we will check if it is the read operation
       begin
          vif.psel <= 1'b1;
          vif.penable <= 1'b0;
          vif.pwdata <= 0;
          vif.paddr <= datac.paddr;
          vif.pwrite <= 1'b0; // this is how we specify the first cycle of an APB transfer
          @(posedge vif.pclk);    // then we wait for positive edge of pclk & then we make penable as '1'
          vif.penable <= 1'b1;    // this marks second cycle of APB transfer
          @(posedge vif.pclk);
          vif.psel <= 1'b0;
          vif.penable <= 1'b0;
          vif.pwrite <= 1'b0; // So, after completion of an APB transfer we keep psel, penable & pwrite to a default value of '0'
          datac.display("DRV");   //display the data that we sent to a DUT, datac is the container where we stor the data that we recieve from a generator
          ->nextdrv;
       end
       
   end
  endtask

endclass

```

Monitor



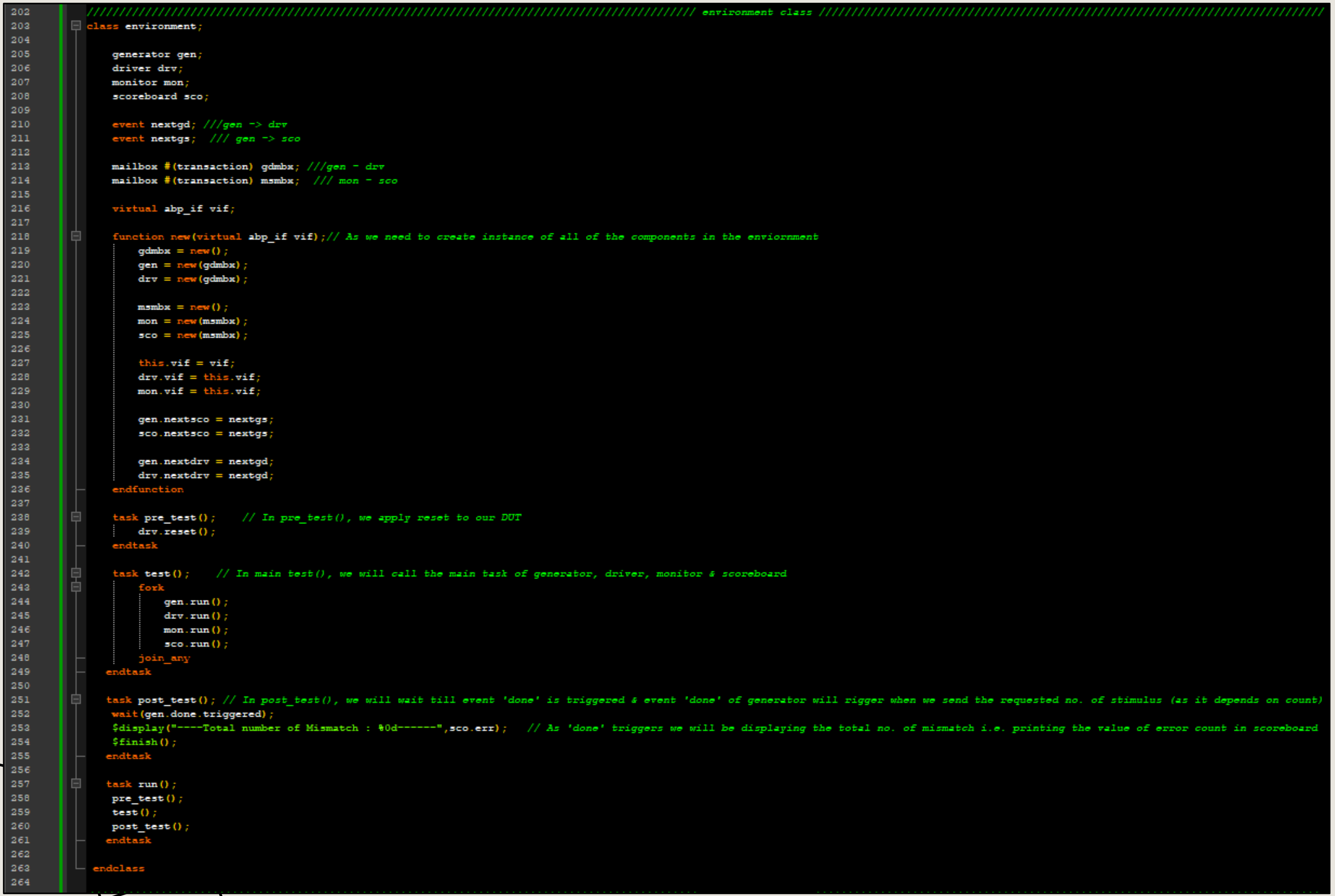
```
/////////////////////////////////////////////////////////////////////////////////////////////////// monitor class //////////////////////////////////////////////////////////////////////////////
class monitor;

    virtual abp_if vif; // In monitor we need to capture the response of DUT, hence we required a virtual interface
    mailbox #(transaction) mbx; // mailbox to send the data to a scoreboard for comparision
    transaction tr;

    function new(mailbox #(transaction) mbx);
        this.mbx = mbx;
    endfunction;

   task run();
    tr = new(); // we create an object of transaction
    forever
        begin
            @(posedge vif.pclk);    // we wait for the positive edge of the clock and wait till completion of our transfer
            if(vif.pready)  // so, APB transfer completion is mark when pready becomes high
                begin
                    tr.pwdata  = vif.pwdata;
                    tr.paddr   = vif.paddr;
                    tr.pwrite  = vif.pwrite;
                    tr.prdata  = vif.prdata;
                    tr.pslverr = vif.pslverr; // After APB transfer completion, we capture all of the data and all of the data is required in a scoreboard to compare with golden response
                    @(posedge vif.pclk);
                    tr.display("MON");  // then we trigger the display() method  for transaction 'tr' with the tag as 'MON' and send this data to a scoreboard
                    mbx.put(tr);
                end
        end
    endtask

endclass
```

Scoreboard



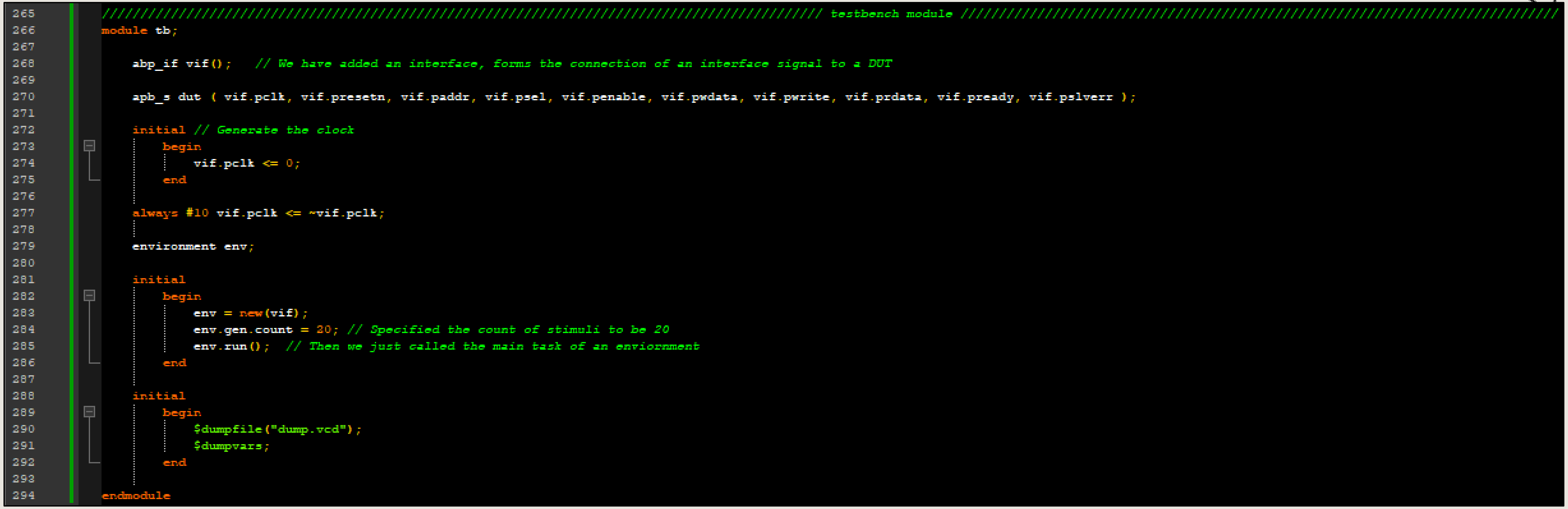
```
/////////////////////////////////////////////////////////////////////////////// scoreboard class ///////////////////////////////////////////////////////////////////////////////////
class scoreboard;

    mailbox #(transaction) mbx;
    transaction tr;
    event nextsco;

    bit [7:0] pwdata[16] = '{default:0};    // Here since we are working on a memory s we have declared an array which is capable of carrying 8-bit data and the depth is 16
    bit [7:0] rdata;
    int err = 0;    // Added this variable to store an error count

    function new(mailbox #(transaction) mbx);
        this.mbx = mbx;
    endfunction;

   task run();
    forever
        begin
            mbx.get(tr); // Here firstly we are recieving the data from monitor
            tr.display("SCO");

            if( (tr.pwrite == 1'b1) && (tr.pslverr == 1'b0))  ///write access
                begin
                    pwdata[tr.paddr] = tr.pwdata;
                    $display("[SCO] : DATA STORED DATA : %0d ADDR: %0d",tr.pwdata, tr.paddr);
                end
            else if((tr.pwrite == 1'b0) && (tr.pslverr == 1'b0))  ///read access
                begin
                    rdata = pwdata[tr.paddr];
                    if( tr.prdata == rdata)
                        $display("[SCO] : Data Matched");
                    else    // else inditates neither write nor read therefore data is mismatched & therefore error variable is also incremented in this case.
                        begin
                            err++;
                            $display("[SCO] : Data Mismatched");
                        end
                end
            else if(tr.pslverr == 1'b1) // if we have an pslverr, then we simply mention the message that SLV ERROR DETECTED
                begin
                    $display("[SCO] : SLV ERROR DETECTED");
                end
            $display("----------------------------------------------------------------------------------------------------");
            ->nextsco;
        end
    endtask

endclass
```

Enviornment



```
//////////////////////////////////////////////////////////////////////////////////////////////// environment class ////////////////////////////////////////////////////////////////////////////////////////////////
class environment;

    generator gen;
    driver drv;
    monitor mon;
    scoreboard sco;

    event nextgd; ///gen -> drv
    event nextgs;  /// gen -> sco

    mailbox #(transaction) gdmbx; ///gen - drv
    mailbox #(transaction) msmbx;  /// mon - sco

    virtual abp_if vif;

    function new(virtual abp_if vif);// As we need to create instance of all of the components in the enviornment
        gdmbx = new();
        gen = new(gdmbx);
        drv = new(gdmbx);

        msmbx = new();
        mon = new(msmbx);
        sco = new(msmbx);

        this.vif = vif;
        drv.vif = this.vif;
        mon.vif = this.vif;

        gen.nextsco = nextgs;
        sco.nextsco = nextgs;

        gen.nextdrv = nextgd;
        drv.nextdrv = nextgd;
    endfunction

    task pre_test();    // In pre_test(), we apply reset to our DUT
        drv.reset();
    endtask

    task test();    // In main test(), we will call the main task of generator, driver, monitor & scoreboard
        fork
            gen.run();
            drv.run();
            mon.run();
            sco.run();
        join_any
   endtask

   task post_test(); // In post_test(), we will wait till event 'done' is triggered & event 'done' of generator will rigger when we send the requested no. of stimulus (as it depends on count)
    wait(gen.done.triggered);
    $display("----Total number of Mismatch : %0d------",sco.err);   // As 'done' triggers we will be displaying the total no. of mismatch i.e. printing the value of error count in scoreboard
    $finish();
   endtask

   task run();
    pre_test();
    test();
    post_test();
   endtask

 endclass
```

Testbench Module



```
//////////////////////////////////////////////////////////////////////////////// testbench module ////////////////////////////////////////////////////////////////////////////////
module tb;

    abp_if vif();   // We have added an interface, forms the connection of an interface signal to a DUT

    apb_s dut ( vif.pclk, vif.presetn, vif.paddr, vif.psel, vif.penable, vif.pwdata, vif.pwrite, vif.prdata, vif.pready, vif.pslverr );

    initial // Generate the clock
        begin
            vif.pclk <= 0;
        end

    always #10 vif.pclk <= ~vif.pclk;

    environment env;

    initial
        begin
            env = new(vif);
            env.gen.count = 20; // Specified the count of stimuli to be 20
            env.run();  // Then we just called the main task of an enviornment
        end

    initial
        begin
            $dumpfile("dump.vcd");
            $dumpvars;
        end

endmodule
```

Design & Verification of APB Protocol Code: Github

THANK YOU