AMBA® APB4 Protocol

topics

1. AMBA APB4 specs
    - 1.1 signals
    - 1.2 transfers
    - 1.3 functionalities
2. Master APB interface
    - 2.1 Verilog code
    - 2.2 RTL elaboration and synthesis schematic
    - 2.3 static-time analysis reports and utilization report
    - 2.4 implementation schematic
    - 2.5 static-time analysis reports and utilization report
3. Slave APB interface
    - 3.1 Verilog code
    - 3.2 RTL elaboration and synthesis schematic
    - 3.3 static-time analysis reports and utilization report
    - 3.4 implementation schematic
    - 3.5 static-time analysis reports and utilization report
4. APB wrapper
    - 4.1 Verilog code
    - 4.2 RTL elaboration and synthesis schematic
    - 4.3 static-time analysis reports and utilization report
    - 4.4 implementation schematic
    - 4.5 static-time analysis reports and utilization report

1. AMBA APB4 specs

1.1 signals

Signal	Source	Description
PCLK	Clock source	Clock. The rising edge of **PCLK** times all transfers on the APB.
PRESETn	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.
PADDR	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
PPROT	APB bridge	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
PSELx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a **PSELx** signal for each slave.
PENABLE	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when **PWRITE** is HIGH. This bus can be up to 32 bits wide.
PSTRB	APB bridge	Write strobes. This signal indicates which byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, **PSTRB[n]** corresponds to **PWDATA[(8n + 7):(8n)]**. Write strobes must not be active during a read transfer.
PREADY	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
PRDATA	Slave interface	Read Data. The selected slave drives this bus during read cycles when **PWRITE** is LOW. This bus can be up to 32-bits wide.
PSLVERR	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the **PSLVERR** pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

Figure(1)

1.2 transfers

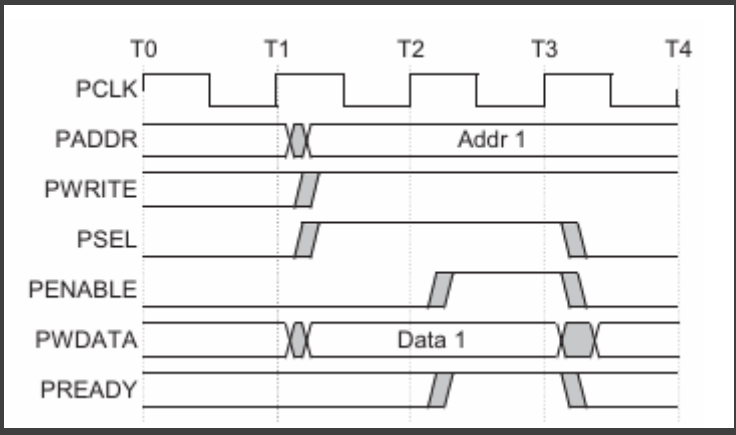
Write transfers (with no waiting)

•**T1 (Setup Phase):**
The transfer begins with the registration of address (PADDR), write data (PWDATA), write signal (PWRITE), and select signal (PSEL) at the rising edge of PCLK.

This phase continuous until the next clock.

•**T2 (Beginning of Access Phase):**
The enable signal (PENABLE) and ready signal (PREADY) are registered. PENABLE's assertion marks the start of the Access phase, and when PREADY is asserted, it indicates that the slave can complete the transfer at the next rising edge of PCLK.

•**T3 (End of Access Phase):**
The address, write data, and control signals remain valid until the transfer completes. At the end, PENABLE is deasserted, and PSEL is deasserted unless another transfer to the same peripheral is immediately scheduled and the slave sample the PWDATA signal

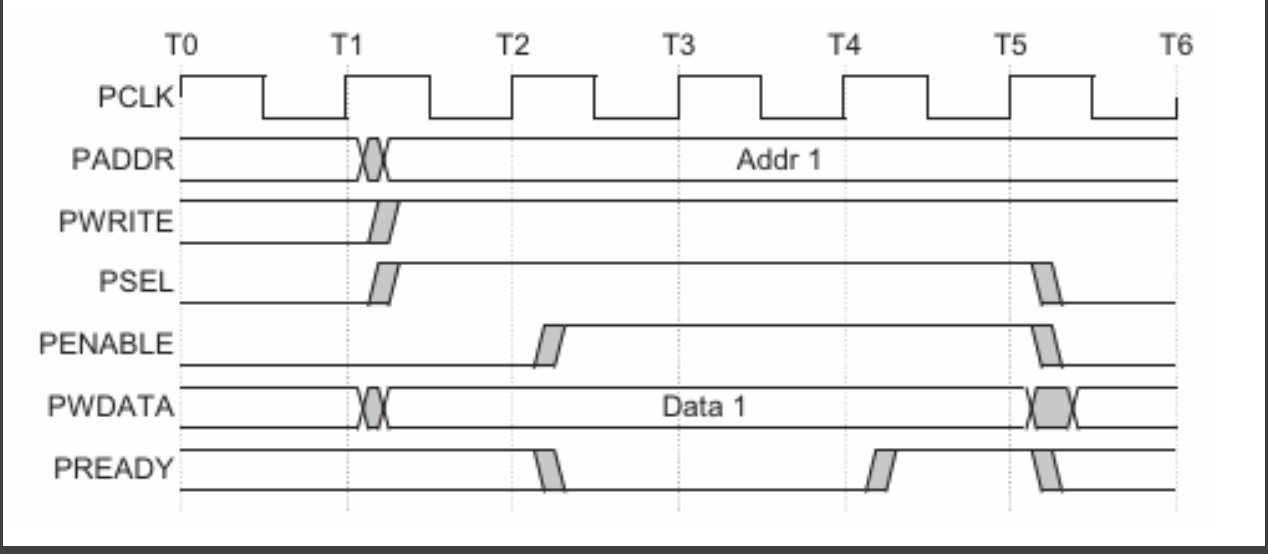


Figure(2)

Write transfers (with waiting)

In the normal case, the slave asserts PREADY at T2, signaling that it is ready to complete the transfer at the next clock edge (T3). In this scenario, the address, data, and control signals remain stable only until the transfer is completed.

In contrast, when the slave needs more time during the Access phase, it drives PREADY LOW even though PENABLE remains HIGH. This effectively extends the transfer by delaying its completion. During this extension, all the critical signals (PADDR, PWRITE, PSEL, PENABLE, PWDATA, PSTRB, and PPROT) are held constant until the slave finally asserts PREADY HIGH to complete the transfer.



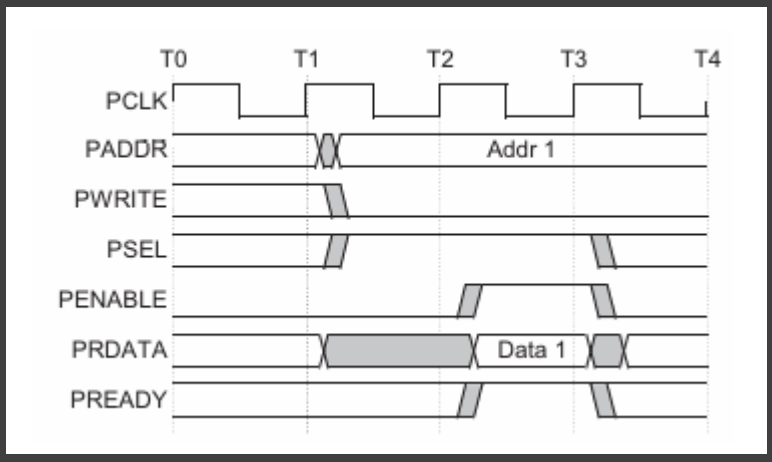
Figure(3)

Read transfers (with no waiting)

•**T1 (Setup Phase):**
Like the write transfers, the read transfer begins with the registration of the address (PADDR), along with select (PSEL) and enable (PENABLE) signals at the rising edge of PCLK.

•**T2 (Access Phase):**
During the Access phase, PENABLE is asserted. The slave drives the ready signal (PREADY) to indicate when it is ready to provide the requested read data.

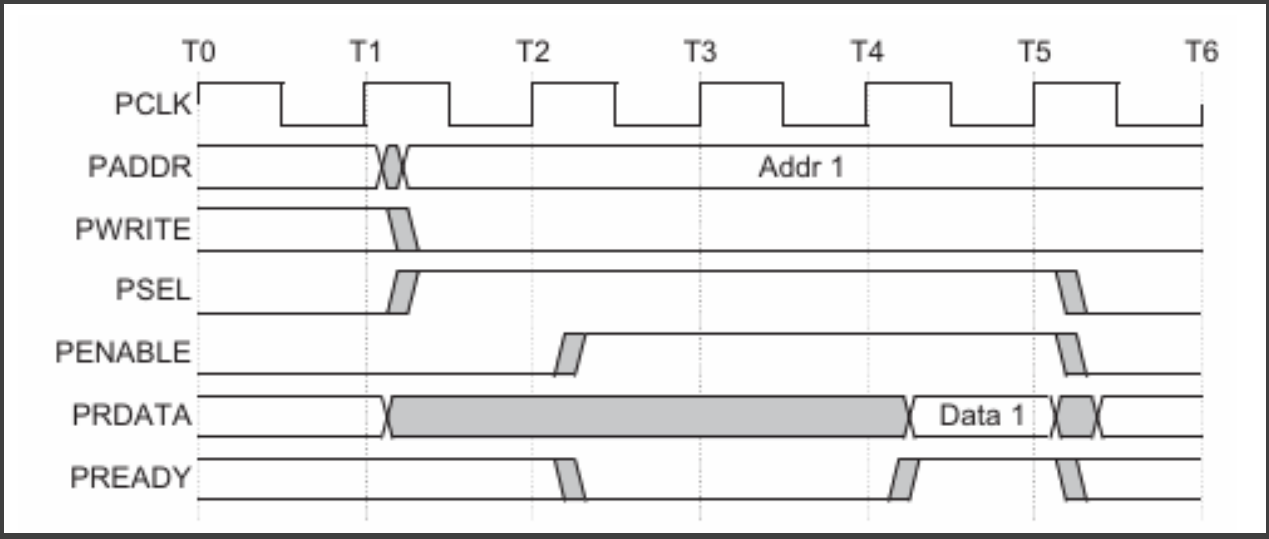
•**T3 (End of Access Phase):**
The transfer completes when PREADY is asserted at the appropriate time. Crucially, the slave must supply the requested read data before the end of the Access phase, ensuring data validity by the end of the transfer.



Figure(4)

Read transfers (with waiting)

The same happens as the write transfer



Figure(5)

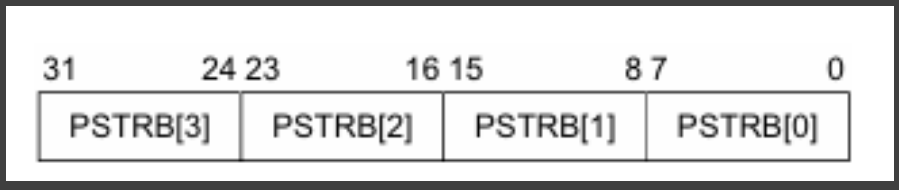
1.3 functionalities

Write strobe

The write strobe signals, PSTRB, enable sparse data transfer on the write data bus.

Each write strobe signal corresponds to one byte of the write data bus.

When asserted HIGH, a write strobe indicates that the corresponding byte lane of the write data bus contains valid information.



Figure(6)

Protection unit support

PPROT	Protection	Description	Comments
PPROT[0]	Normal or Privileged	**PPROT[0]** is used by Requesters to indicate processing mode. A privileged processing mode typically has a greater level of access within a system.	• LOW indicates normal access. • HIGH indicates privileged access.
PPROT[1]	Secure or Non-secure	**PPROT[1]** is used in systems where a greater degree of differentiation between processing modes is required.	• LOW indicates secure access. • HIGH indicates non-secure access.
PPROT[2]	Data or Instruction	**PPROT[2]** gives an indication if the transaction is a data or instruction access. The transaction indication is provided as a hint and might not be accurate in all cases.	• LOW indicates data access. • HIGH indicates instruction access.

Figure(7)

PPROT	Completer: signal not present	Completer: signal present
Requester: signal not present	Compatible.	Not compatible. If fixed protection attributes are functionally correct, then the interfaces are Compatible. See Table 3-2 on page 3-27.
Requester: signal present	Compatible. The Completer has no access protection so **PPROT** can be ignored.	Compatible.

Figure(8)

2. Master APB interface

2.1 Verilog code

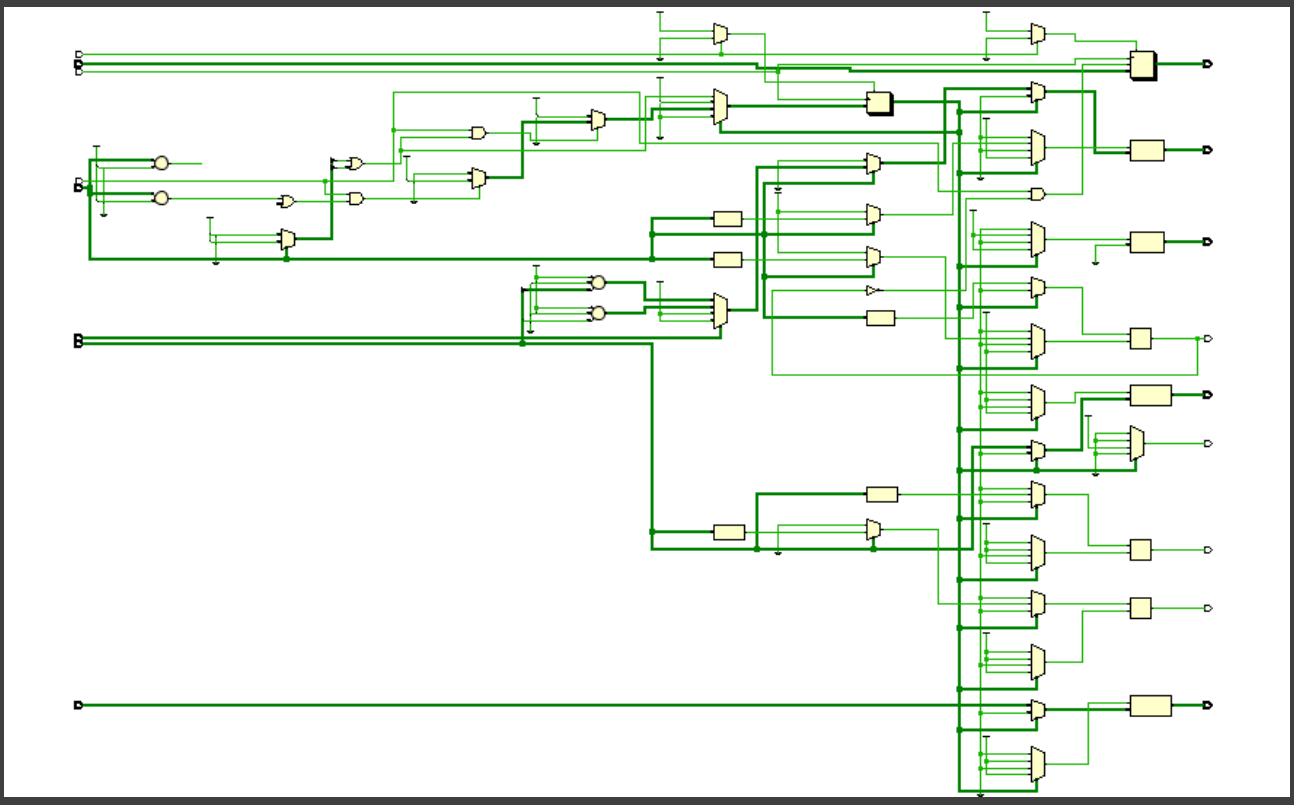
```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed Ragab
/// Module-Name: apb_master_interface
////////////////////////////////////////////////////////////////////////////////

module apb_master_interface (PCLK, PRESETn, PADDR, PPROT, PSEL0, PSEL1, PENABLE, PWRITE, PWDATA, PSTRB, PREADY, PRDATA, address, write_data, read_data, process, data_size);

    // Parameters
    parameter ADDR_WIDTH = 32;                          // Width of the address bus
    parameter DATA_WIDTH = 32;                          // Width of the data bus
    parameter STRB_WIDTH = DATA_WIDTH / 8;              // Byte strobe width
    parameter IDLE_PHASE = 2'b00;                       // Idle state
    parameter SETUP_PHASE = 2'b01;                      // Setup state
    parameter ACCESS_PHASE = 2'b10;                     // Access state

    // Inputs
    input [DATA_WIDTH-1:0] PRDATA;                      // Data read from slave
    input [DATA_WIDTH-1:0] write_data;                  // Data to be written to slave
    input [ADDR_WIDTH-1:0] address;                     // Address for the transaction
    input [6:0] process;                                // Process control signal
    input [1:0] data_size;                              // Data size signal
    input PCLK;                                         // Clock signal
    input PRESETn;                                      // Reset (active low)
    input PREADY;                                       // Ready signal from slave

    // Outputs
    output reg [ADDR_WIDTH-1:0] PADDR;                  // APB address output
    output reg [DATA_WIDTH-1:0] PWDATA;                 // Write data output for APB
    output reg [DATA_WIDTH-1:0] read_data;              // Read data captured from APB
    output reg [STRB_WIDTH-1:0] PSTRB;                  // Byte strobe output
    output reg [2:0] PPROT;                             // Protection bits output
    output reg PSEL1;                                   // Peripheral select 1
    output reg PSEL0;                                   // Peripheral select 0
    output reg PENABLE;                                 // APB enable signal
    output reg PWRITE;                                  // APB write signal

    // Internal signals
    reg [1:0] current_state;                            // Current state of FSM
    reg [1:0] next_state;                               // Next state of FSM

    // Always block triggered on the rising edge of the clock
    always @(posedge PCLK) begin
        if (!PRESETn) begin
            current_state <= IDLE_PHASE;                // Reset state to IDLE
        end else begin
            current_state <= next_state;                // Update state
        end
    end
```

```
// Always block triggered on the rising edge of the clock
always @(*) begin
        case (current_state)
            IDLE_PHASE: begin
                PSEL0 <= 0;                             // Deassert peripheral select 0
                PSEL1 <= 0;                             // Deassert peripheral select 1
                PENABLE <= 0;                           // Deassert enable signal
            end

            SETUP_PHASE: begin
                PADDR <= address;                       // Set address
                PENABLE <= 0;                           // Deassert enable signal
                PPROT <= 3'b000;                        // Set protection level to 0
                PWDATA <= write_data;                 // Set write data
                if (address == 4000) begin              // Select peripheral based on address
                    PSEL1 <= 0;
                    PSEL0 <= 1;
                end else if (address == 4001) begin
                    PSEL1 <= 1;
                    PSEL0 <= 0;
                end else begin
                    PSEL1 <= 0;
                    PSEL0 <= 0;
                end
                if (process == 7'b0000011) begin        // Read operation
                    PWRITE <= 0;                        // Set write signal to 0
                    PSTRB <= 4'b0000;                   // Set strobe to 0
                end else if (process == 7'b0100011) begin // Write operation
                    PWRITE <= 1;                        // Set write signal to 1
                    case (data_size)
                        2'b00:                          // Byte
                            PSTRB <= (4'b0001 << address[1:0]);
                        2'b01:                          // Half-word
                            PSTRB <= (4'b0011 << {address[1],1'b0});
                        2'b10:                          // Word
                            PSTRB <= 4'b1111;
                        default: PSTRB <= 4'b1111;
                    endcase
                end
            end

            ACCESS_PHASE: begin
                PENABLE <= 1;                           // Enable the transfer
            end

            default: begin
                PADDR <= 0;                             // Reset address
                PWDATA <= 0;                            // Reset write data
                PSTRB <= 0;                             // Reset strobe
                PPROT <= 0;                             // Reset protection bits
                PSEL1 <= 0;                             // Reset peripheral select 1
                PSEL0 <= 0;                             // Reset peripheral select 0
                PENABLE <= 0;                           // Reset enable signal
                PWRITE <= 0;                            // Reset write signal
            end
        endcase
    end
```

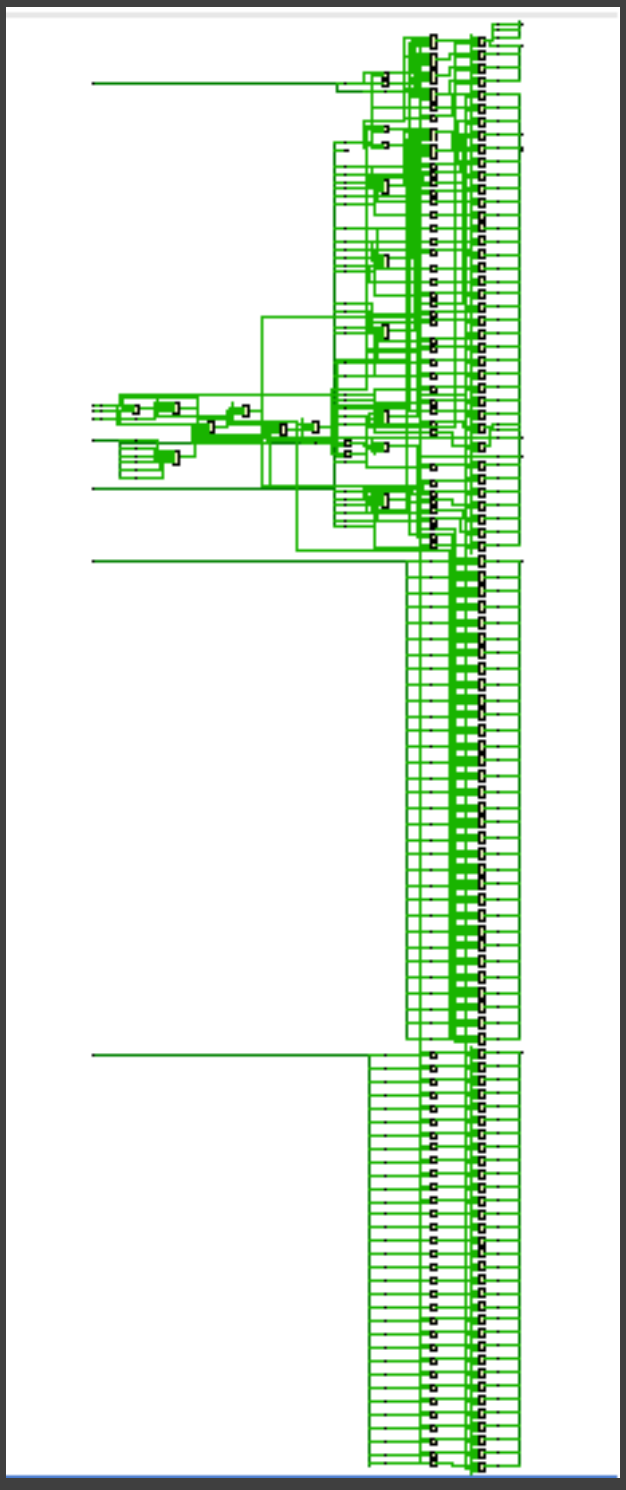
```
    // Always block for next state logic
    always @(*) begin
        case (current_state)
            IDLE_PHASE: begin
                if (process == (7'b0000011) || process == (7'b0100011)) begin
                    next_state <= SETUP_PHASE;              // Transition to SETUP
                end else begin
                    next_state <= IDLE_PHASE;               // Remain in IDLE
                end
            end

            SETUP_PHASE: begin
                next_state <= ACCESS_PHASE;                 // Transition to ACCESS
            end

            ACCESS_PHASE: begin
                if (PREADY && (process == (7'b0000011) || process == (7'b0100011))) begin
                    next_state <= SETUP_PHASE;              // Transition to SETUP
                end else if (PREADY && (process != (7'b0000011) || process != (7'b0100011))) begin
                    next_state <= IDLE_PHASE;               // Transition to IDLE
                end else begin
                    next_state <= ACCESS_PHASE;             // Remain in ACCESS
                end
            end
            default: next_state <= IDLE_PHASE;              // Default to IDLE
        endcase
    end

    // Always block to capture read data on the rising edge of the clock
    always @(posedge PCLK) begin
        if (!PRESETn) begin
            read_data <= 0;
        end else if (PREADY && !PWRITE) begin
            read_data <= PRDATA;                // Capture read data
        end
    end

endmodule
```

2.2 RTL and synthesis schematic

RTL ANALYSIS



Synthesis schematic



2.3 static-time analysis reports and utilization report

Static time analysis

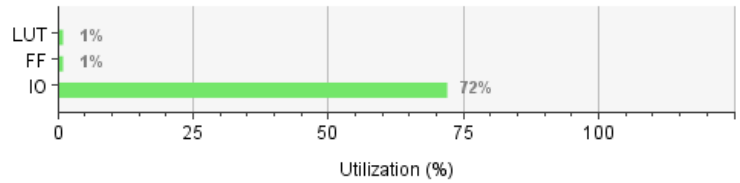
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	274	Total Number of Endpoints:	274	Total Number of Endpoints:	NA

There are no user specified timing constraints.

Utilization report

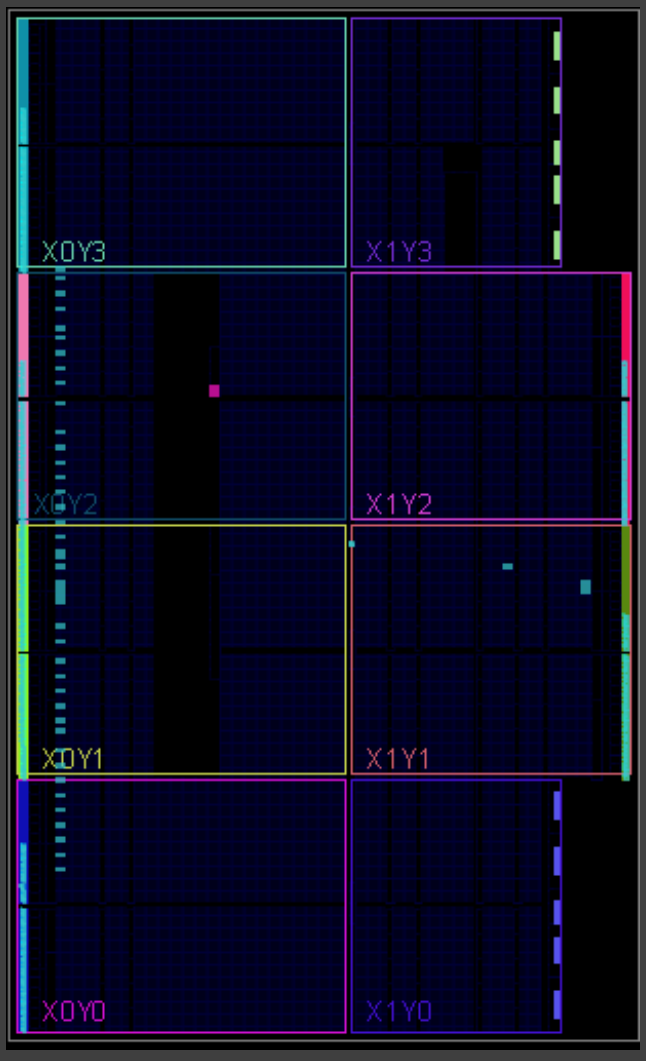
Name	Slice LUTs (47200)	Slice Registers (94400)	Bonded IOB (300)	BUFGCTRL (32)
N apb_master_interface	87	106	215	1

Resource	Utilization	Available	Utilization %
LUT	87	47200	0.18
FF	106	94400	0.11
IO	215	300	71.67



2.4 implementation schematic

implementation schematic



2.5 static-time analysis reports and utilization report

Static time analysis

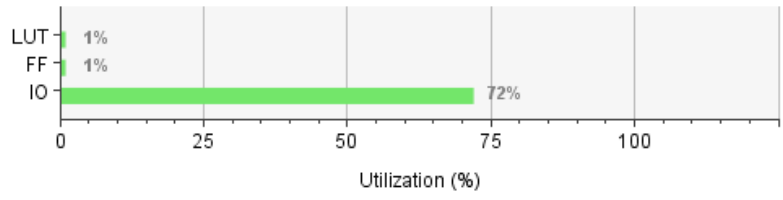
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	274	Total Number of Endpoints:	274	Total Number of Endpoints:	NA

There are no user specified timing constraints.

Utilization report

Name	Slice LUTs (47200)	Slice Registers (94400)	Slice (15850)	LUT as Logic (47200)	LUT Flip Flop Pairs (47200)	Bonded IOB (300)	BUFGCTRL (32)
N apb_master_interface	86	106	59	86	5	215	2

Resource	Utilization	Available	Utilization %
LUT	86	47200	0.18
FF	106	94400	0.11
IO	215	300	71.67



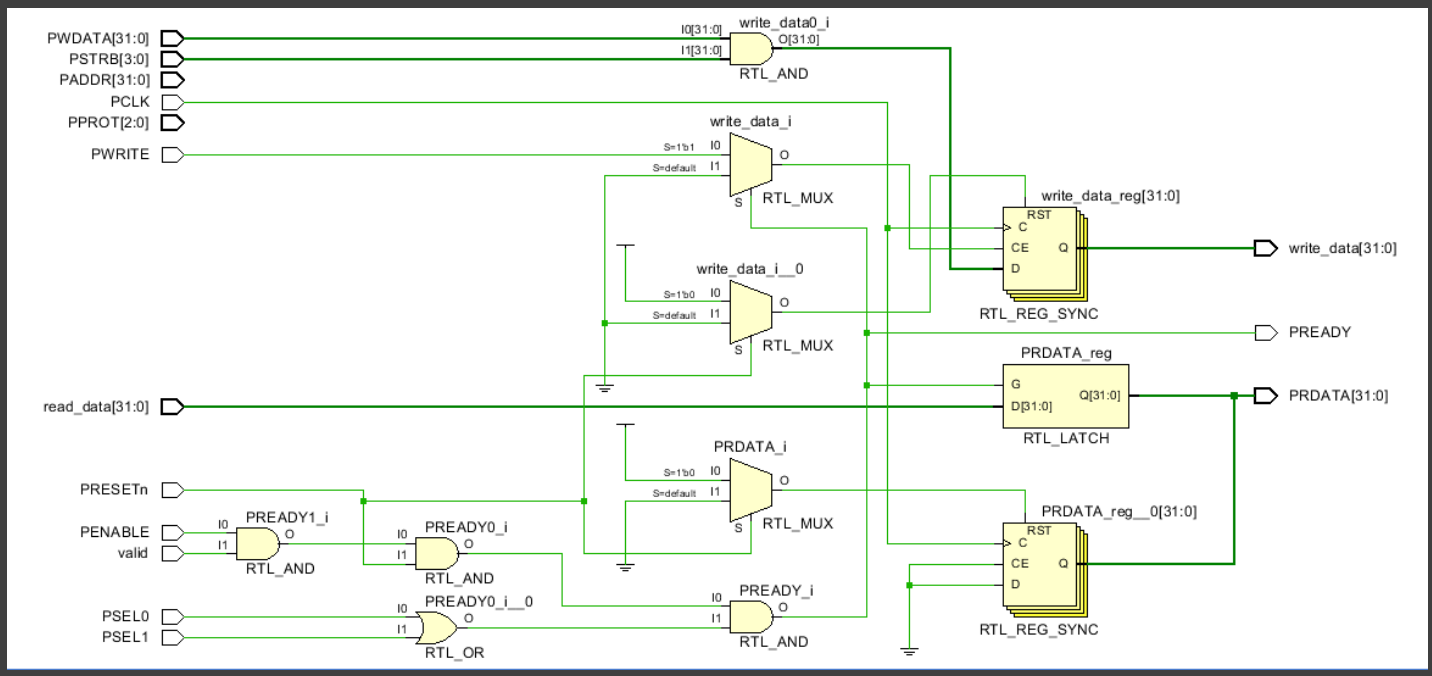
3. Slave APB interface

3.1 Verilog code

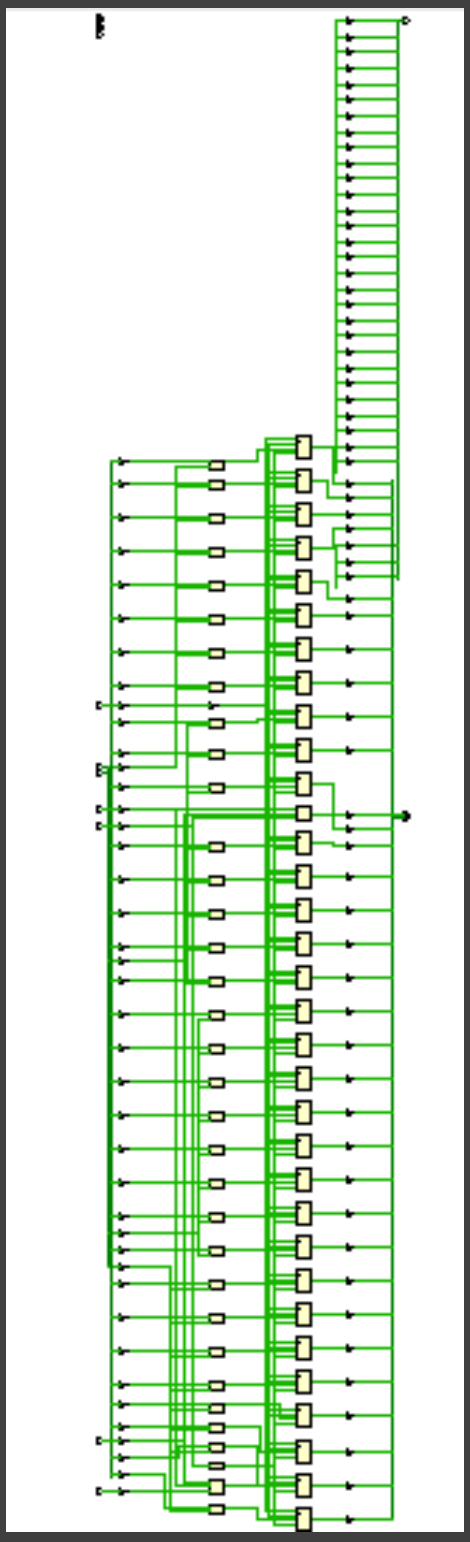
```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed Ragab
/// Module-Name: slave_interface
////////////////////////////////////////////////////////////////////////////////

module apb_slave_interface (PCLK, PRESETn, PADDR, PPROT, PSEL0, PSEL1, PENABLE, PWRITE, PWDATA, PSTRB, PREADY, PRDATA, write_data, read_data, valid);

    // Parameters
    parameter ADDR_WIDTH = 32;                          // Width of the address bus
    parameter DATA_WIDTH = 32;                          // Width of the data bus
    parameter STRB_WIDTH = DATA_WIDTH / 8;              // Byte strobe width
    parameter IDLE_PHASE = 2'b00;                       // Idle state
    parameter SETUP_PHASE = 2'b01;                      // Setup state
    parameter ACCESS_PHASE = 2'b10;                     // Access state

    // Inputs
    input [DATA_WIDTH-1:0] read_data;                   // Data read from master
    input [DATA_WIDTH-1:0] PWDATA;                      // Write data from master
    input [ADDR_WIDTH-1:0] PADDR;                       // Address from master
    input [STRB_WIDTH-1:0] PSTRB;                       // Byte strobe from master
    input [2:0] PPROT;                                  // Protection bits from master
    input PCLK;                                         // Clock signal
    input PRESETn;                                      // Active low reset signal
    input PSEL1;                                        // Slave select signal for the first slave device
    input PSEL0;                                        // Slave select signal for the second slave device
    input PENABLE;                                      // Enable signal for the slave device
    input PWRITE;                                       // Write signal for the slave device
    input valid;                                        // Valid signal for the slave device

    // Outputs
    output reg [DATA_WIDTH-1:0] write_data;             // Data to be written to the master device
    output reg [DATA_WIDTH-1:0] PRDATA;                 // Data to be written to the master device
    output PREADY;                                    // Ready signal to the master

    // Always block triggered on the rising edge of the clock
    always @(posedge PCLK) begin
        if (!PRESETn) begin
            write_data <= 0;                            // Reset write_data
            PRDATA <= 0;                                // Reset PRDATA
        end else if (PREADY) begin
            if (PWRITE) begin
                write_data <= PWDATA & {{8{PSTRB[3]}},{8{PSTRB[2]}},{8{PSTRB[1]}},{8{PSTRB[0]}}};             // Write data to the master device
            end
        end
    end

    assign PREADY = (PENABLE && valid && PRESETn && (PSEL0||PSEL1)); // Generate ready signal based on select and enable signals

    always @(*) begin
        if (PREADY) begin
            PRDATA <= read_data;
        end
    end

endmodule
```

3.2 RTL and synthesis schematic

RTL ANALYSIS



Synthesis schematic



3.3 static-time analysis reports and utilization report

Static time analysis

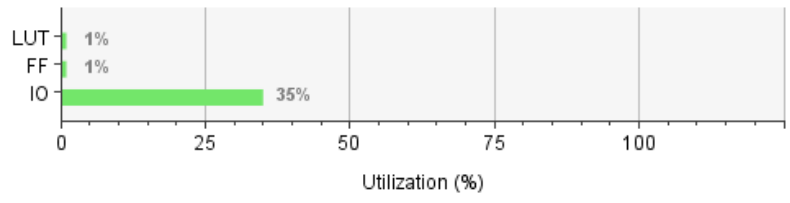
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	129	Total Number of Endpoints:	129	Total Number of Endpoints:	NA

There are no user specified timing constraints.

Utilization report

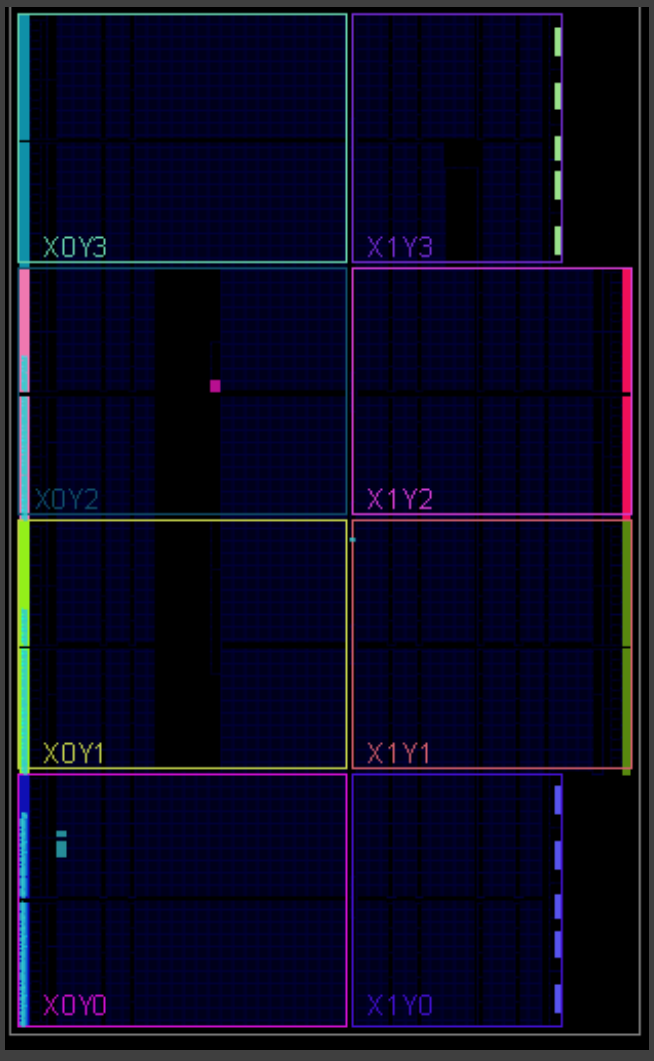
Name 1	Slice LUTs (47200)	Slice Registers (94400)	Bonded IOB (300)	BUFGCTRL (32)
N apb_slave_interface	19	32	106	1

Resource	Utilization	Available	Utilization %
LUT	19	47200	0.04
FF	32	94400	0.03
IO	106	300	35.33



3.4 implementation schematic

implementation schematic



3.5 static-time analysis reports and utilization report

Static time analysis

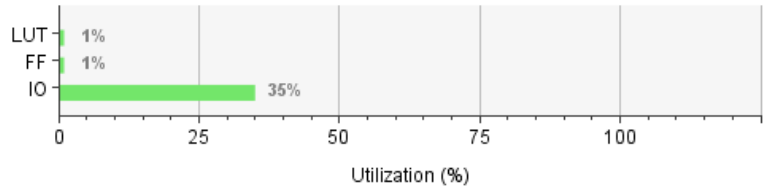
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	129	Total Number of Endpoints:	129	Total Number of Endpoints:	NA

There are no user specified timing constraints.

Utilization report

Name	Slice LUTs (47200)	Slice Registers (94400)	Slice (15850)	LUT as Logic (47200)	LUT Flip Flop Pairs (47200)	Bonded IOB (300)	BUFGCTRL (32)
N apb_slave_interface	19	32	5	19	16	106	1

Resource	Utilization	Available	Utilization %
LUT	19	47200	0.04
FF	32	94400	0.03
IO	106	300	35.33



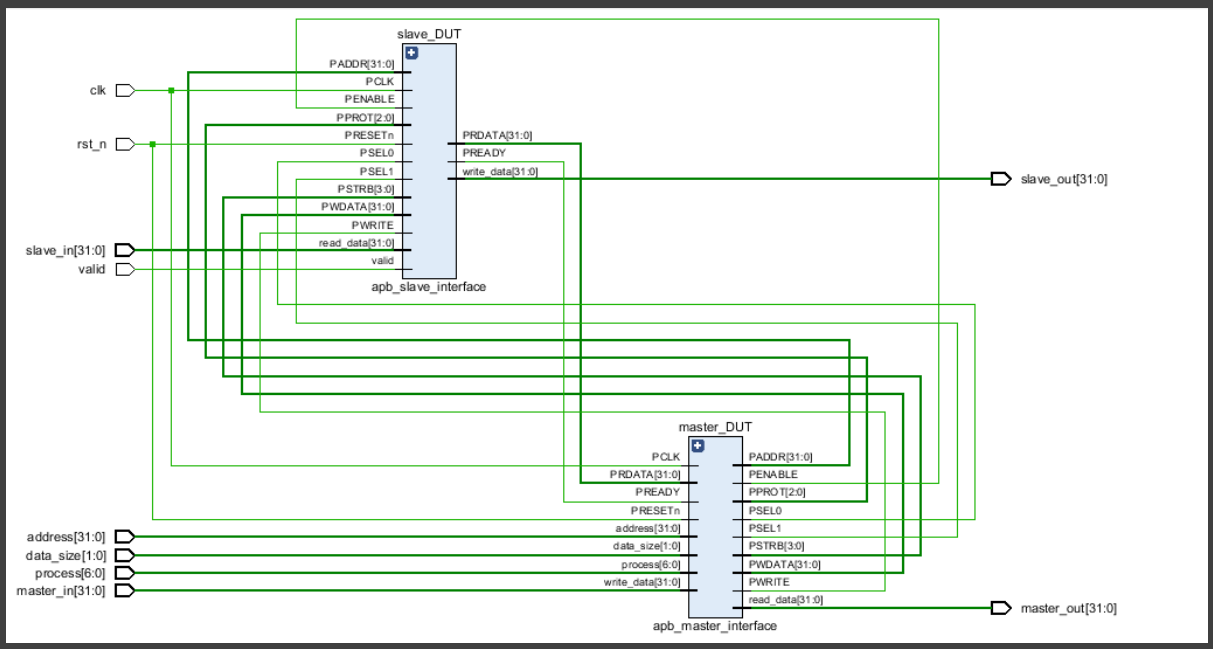
4. APB wrapper

4.1 Verilog code

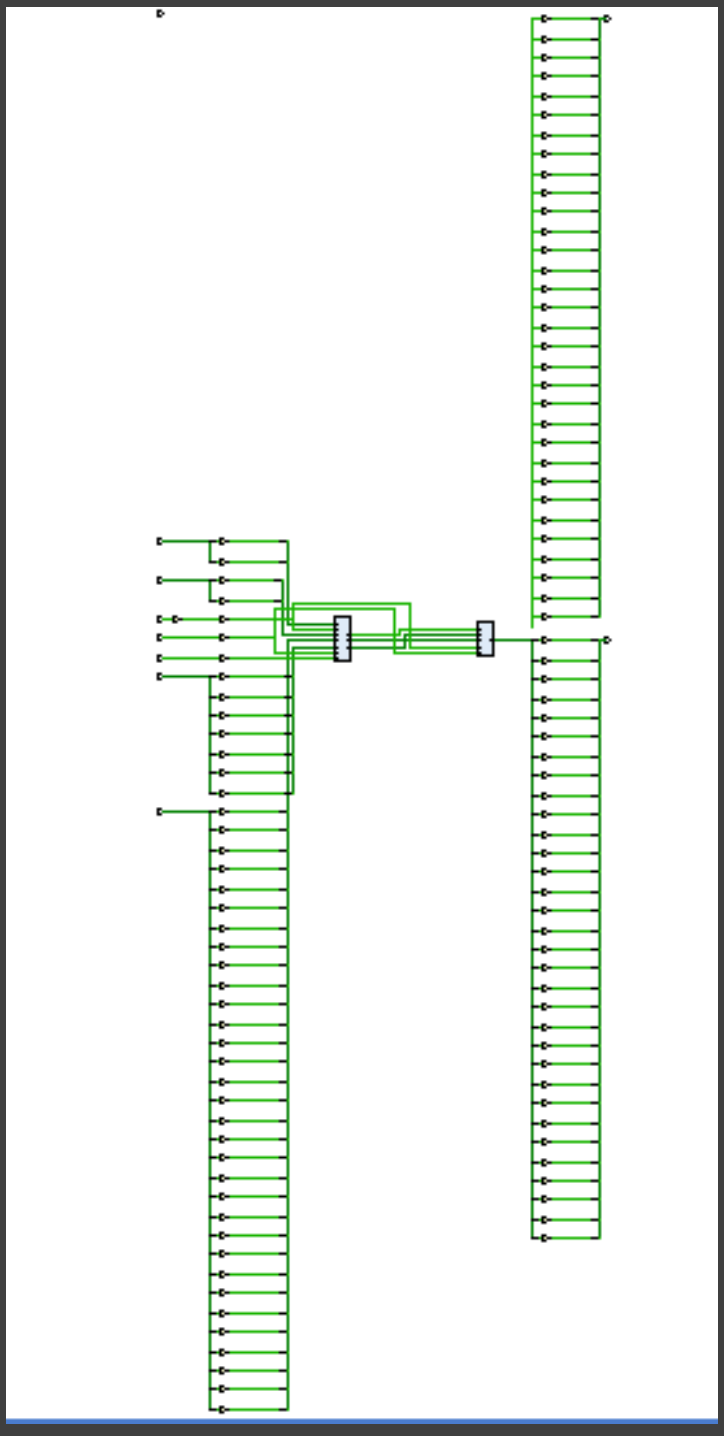
```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed Ragab
/// Module-Name: apb_top_design
////////////////////////////////////////////////////////////////////////////////

module apb_top_design (apb_int apbint);

    // Instantiate the master interface
    apb_master_interface master_DUT (
        .PADDR(apbint.PADDR),
        .PWDATA(apbint.PWDATA),
        .PSTRB(apbint.PSTRB),
        .PPROT(apbint.PPROT),
        .PSEL1(apbint.PSEL1),
        .PSEL0(apbint.PSEL0),
        .PENABLE(apbint.PENABLE),
        .PWRITE(apbint.PWRITE),
        .PREADY(apbint.PREADY),
        .PRDATA(apbint.PRDATA),
        .PCLK(apbint.clk),
        .PRESETn(apbint.rst_n),
        .address(apbint.address),
        .process(apbint.process),
        .data_size(apbint.data_size),
        .read_data(apbint.master_out),
        .write_data(apbint.master_in)
    );

    // Instantiate the slave interface
    apb_slave_interface slave_DUT (
        .PADDR(apbint.PADDR),
        .PWDATA(apbint.PWDATA),
        .PSTRB(apbint.PSTRB),
        .PPROT(apbint.PPROT),
        .PSEL1(apbint.PSEL1),
        .PSEL0(apbint.PSEL0),
        .PENABLE(apbint.PENABLE),
        .PWRITE(apbint.PWRITE),
        .PREADY(apbint.PREADY),
        .PRDATA(apbint.PRDATA),
        .PCLK(apbint.clk),
        .PRESETn(apbint.rst_n),
        .read_data(apbint.slave_in),
        .write_data(apbint.slave_out),
        .valid(apbint.valid)
    );

endmodule
```

4.2 RTL and synthesis schematic

RTL ANALYSIS



Synthesis schematic



4.3 static-time analysis reports and utilization report

Static time analysis

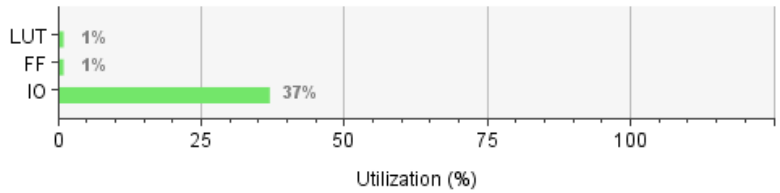
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	13.306 ns	Worst Hold Slack (WHS):	0.164 ns	Worst Pulse Width Slack (WPWS):	7.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	35	Total Number of Endpoints:	35	Total Number of Endpoints:	36

All user specified timing constraints are met.

Utilization report

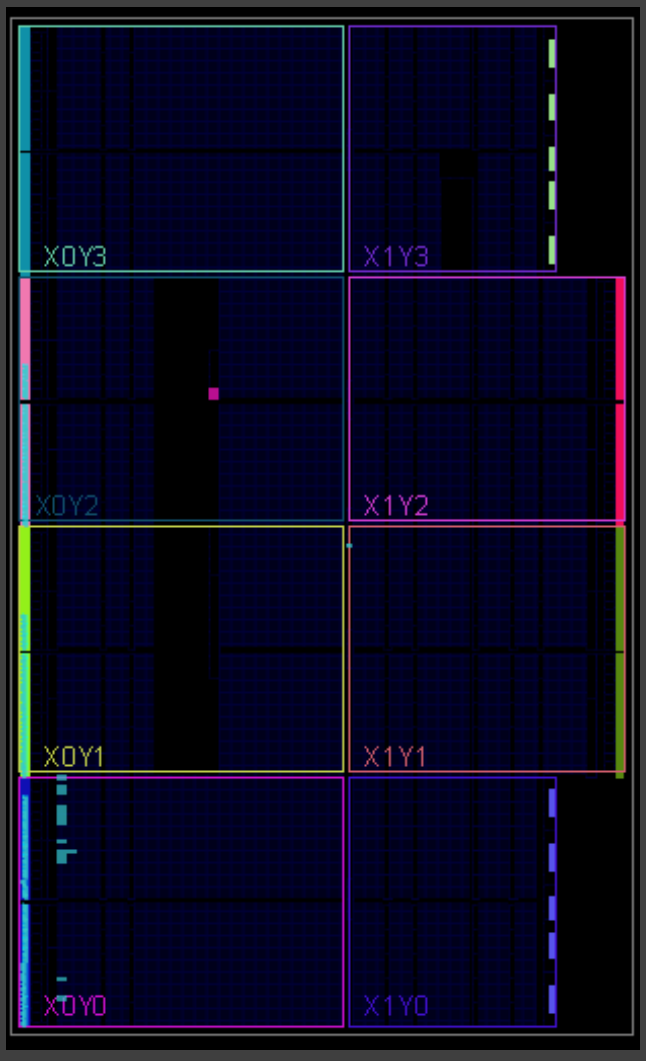
Name	Slice LUTs (47200)	Slice Registers (94400)	Bonded IOB (300)	BUFGCTRL (32)
N apb_top_design	61	72	110	1
master_DUT (apb_ma...	44	40	0	0
slave_DUT (apb_slave...	17	32	0	0

Resource	Utilization	Available	Utilization %
LUT	61	47200	0.13
FF	72	94400	0.08
IO	110	300	36.67



4.4 implementation schematic

implementation schematic



4.5 static-time analysis reports and utilization report

Static time analysis

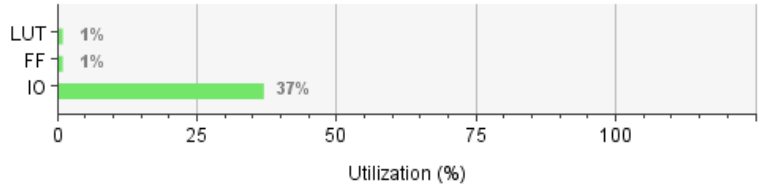
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	13.473 ns	Worst Hold Slack (WHS):	0.217 ns	Worst Pulse Width Slack (WPWS):	7.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	35	Total Number of Endpoints:	35	Total Number of Endpoints:	36

All user specified timing constraints are met.

Utilization report

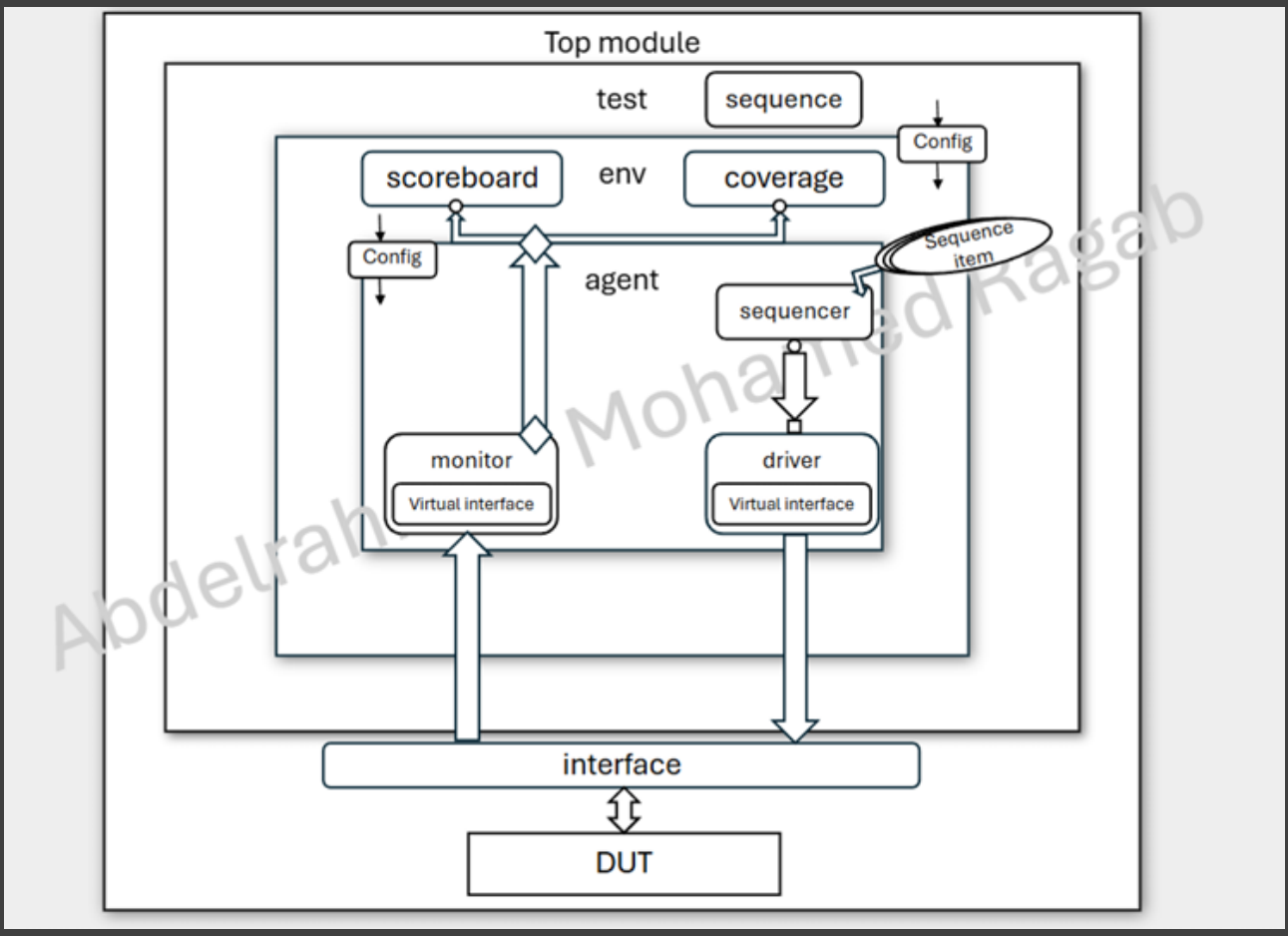
Name	Slice LUTs (47200)	Slice Registers (94400)	Slice (15850)	LUT as Logic (47200)	LUT Flip Flop Pairs (47200)	Bonded IOB (300)	BUFGCTRL (32)
N apb_top_design	61	72	24	61	23	110	1
master_DUT (apb_ma...	44	40	20	44	7	0	0
slave_DUT (apb_slave...	17	32	5	17	16	0	0

Resource	Utilization	Available	Utilization %
LUT	61	47200	0.13
FF	72	94400	0.08
IO	110	300	36.67



5. APB wrapper verification

5.1 UVM plan



5.2 verification plan

Label	Description	input stimulus
1) reset sequence	verify the functionality of the reset signal and that the reset is: 1) the reset is sync with the clock 2) the reset is active low 3) the reset do reset function on the states and the outputs	directed (on for 1 clock cycle)
2) main sequence	check for the behavour on all other data signals in the design as: 1)APB data signals which is between the master and slave interfaces are combinational and can be edited alomg the setup phase 2) the states are sequential. 3) next state generation is combinational and the next state calculation is right. 4) the master interface generate the APB signals according to its input and behave according to the states in a right way . 5) the slave interface generate the ready signal in the right time when it recieve the valid signal and read and write successfully.	random 1m times according to the following constraints: 1)rst_n: high 90%, low 10% 2)valid: high 90%, low 10% 3)address: "4000" 45%, "4001" 45%, other 10% 4)process: "load" 45%, "store" 45%, other 10% 5) data_size: "0" 30%, "1" 30%, "2" 30%

5.3 UVM sequence_item

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_sequence_item
////////////////////////////////////////////////////////////////////////////////

package apb_sequence_item_pkg;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_sequence_item extends uvm_sequence_item;

        // Factory registration
        `uvm_object_utils(apb_sequence_item);

        // Parameters
        parameter ADDR_WIDTH = 32;                      // Width of the address bus
        parameter DATA_WIDTH = 32;                      // Width of the data bus
        parameter STRB_WIDTH = DATA_WIDTH / 8;          // Byte strobe width

        // Internal signals
            rand logic [DATA_WIDTH-1:0] master_in;              // Data read from slave
            rand logic [DATA_WIDTH-1:0] slave_in;               // Data to be written to slave
            rand logic [ADDR_WIDTH-1:0] address;                // Address for the transaction
            logic [ADDR_WIDTH-1:0] master_out;              // APB address output
            logic [DATA_WIDTH-1:0] slave_out;               // Write data output for APB
            rand logic [6:0] process;                           // Process control signal
            rand logic [1:0] data_size;                         // Data size signal
            rand logic rst_n;                                   // Reset (active low)
            rand logic valid;                                   // Valid signal
            logic [DATA_WIDTH-1:0] PWDATA;                  // Write data for slave
            logic [STRB_WIDTH-1:0] PSTRB;                   // Byte strobe signals
            logic [DATA_WIDTH-1:0] PRDATA;                  // Data read from slave
            logic [ADDR_WIDTH-1:0] PADDR;                   // Address for the transaction
            logic [2:0] PPROT;                              // Protection bits
            logic PSEL1;                                    // Peripheral select 1
            logic PSEL0;                                    // Peripheral select 0
            logic PENABLE;                                  // Enable signal
            logic PWRITE;                                   // Write signal
            logic PREADY;                                   // Ready signal from slave
            bit clk;                                        // Clock signal
```

```
    // Functions

    // Convert the sequence item to a string for debugging
    function string convert2string();
        return $sformatf("%s master_in = 0h%0h, slave_in = 0h%0h, address = 0d%0d, process = 0b%0b, data_size = 0b%0b, valid = 0b%0b, rst_n = 0b%0b, master_out = 0h%0h, slave_out = 0h%0h,
        PWDATA = 0h%0h, PSTRB = 0b%0b, PRDATA = 0h%0h, PADDR = 0d%0d, PPROT = 0b%0b, PSEL1 = 0b%0b, PSEL0 = 0b%0b, PENABLE = 0b%0b, PWRITE = 0b%0b, PREADY = 0b%0b",
                        super.convert2string(), master_in, slave_in, address, process, data_size, valid, rst_n, master_out, slave_out, PWDATA, PSTRB, PRDATA, PADDR, PPROT, PSEL1, PSEL0, PENABLE, PWRITE, PREADY);
    endfunction // convert2string function

    // Convert the stimulus to a string for debugging
    function string convert2string_stimulus();
        return $sformatf("master_in = 0h%0h, slave_in = 0h%0h, address = 0d%0d, process = 0b%0b, data_size = 0b%0b, valid = 0b%0b, rst_n = 0b%0b",
                        master_in, slave_in, address, process, data_size, valid, rst_n);
    endfunction // convert2string_stimulus function

    // Constraints
    constraint apb_constraints {
        rst_n dist {0:/10, 1:/90};                      // Reset is off 10% and on 90%
        valid dist {0:/10, 1:/90};                      // Valid is off 10% and on 90%
        address dist {4000:/45, 4001:/45, [0:32'hFFFFFFFF]:/10};            // Address distribution
        process dist {7'b0000011:/45, 7'b0100011:/45, [0:7'b1111111]:/10};  // Process distribution
        data_size dist {0:/30, 1:/30, 2:/30};           // Data size distribution
    } // apb_constraints

endclass // apb_sequence_item class

endpackage // apb_sequence_item_pkg
```

5.4 UVM sequencer

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_sequencer
////////////////////////////////////////////////////////////////////////////////

package apb_sequencer_pkg;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    import apb_sequence_item_pkg::*;                    // Import APB sequence item package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_sequencer extends uvm_sequencer #(apb_sequence_item);

        // Factory registration
        `uvm_component_utils(apb_sequencer);

        // Constructor
        function new(string name = "apb_sequencer", uvm_component parent = null);
            super.new(name, parent);                    // Call parent constructor
        endfunction // new function

    endclass // apb_sequencer class

endpackage // apb_sequencer_pkg
```

5.5 UVM reset_sequence

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_rst_sequence
////////////////////////////////////////////////////////////////////////////////

package apb_rst_sequence_pkg;

    // Package imports
    import uvm_pkg::*;                                      // Import UVM package
    import apb_sequence_item_pkg::*;                        // Import APB sequence item package
    `include "uvm_macros.svh";                              // Include UVM macros

    // Class definition
    class apb_rst_sequence extends uvm_sequence #(apb_sequence_item);

        // Factory registration
        `uvm_object_utils(apb_rst_sequence);

        // Object declaration
        apb_sequence_item seq_item;                         // Sequence item object

        // Constructor
        function new(string name = "apb_rst_sequence");
            super.new(name);                                // Call parent constructor
        endfunction // new function

        // Body task
        task body();
            // Sequence randomization
            seq_item = apb_sequence_item::type_id::create("seq_item"); // Create sequence item
            start_item(seq_item);                           // Start the sequence item
            seq_item.rst_n = 0;                             // Set reset signal to 0
            finish_item(seq_item);                          // Finish the sequence item
        endtask // body task

    endclass // apb_rst_sequence class

endpackage // apb_rst_sequence_pkg
```

5.6 UVM main_sequence

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_main_sequence
////////////////////////////////////////////////////////////////////////////////

package apb_main_sequence_pkg;

    // Package imports
    import uvm_pkg::*;                                      // Import UVM package
    import apb_sequence_item_pkg::*;                        // Import APB sequence item package
    `include "uvm_macros.svh";                              // Include UVM macros

    // Class definition
    class apb_main_sequence extends uvm_sequence #(apb_sequence_item);

        // Factory registration
        `uvm_object_utils(apb_main_sequence);

        // Object declaration
        apb_sequence_item seq_item;                         // Sequence item object

        // Constructor
        function new(string name = "apb_main_sequence");
            super.new(name);                                // Call parent constructor
        endfunction // new function

        // Body task
        task body();
            // Sequence randomization
            repeat(1000000) begin
                seq_item = apb_sequence_item::type_id::create("seq_item"); // Create sequence item
                start_item(seq_item);                       // Start the sequence item
                assert (seq_item.randomize());              // Randomize the sequence item
                finish_item(seq_item);                      // Finish the sequence item
            end
        endtask // body task

    endclass // apb_main_sequence class

endpackage // apb_main_sequence_pkg
```

5.7 UVM coverage

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_coverge
////////////////////////////////////////////////////////////////////////////////

package apb_coverage_pkg;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    import apb_sequence_item_pkg::*;                    // Import APB sequence item package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_coverge extends uvm_component;

        // Factory registration
        `uvm_component_utils(apb_coverge);

        // Object declaration
        apb_sequence_item seq_item;                     // Sequence item object

        // TLM analysis export declaration
        uvm_analysis_export #(apb_sequence_item) cov_exp;   // Analysis export for coverage
        uvm_tlm_analysis_fifo #(apb_sequence_item) cov_apb;  // TLM analysis for APB

        // Covergroup declaration
        covergroup apb_coverage;

            // Valid signal coverage
            valid: coverpoint seq_item.valid {
                bins valid = {1};                       // Valid signal is high
                bins invalid = {0};                     // Valid signal is low
            }

            // Reset signal coverage
            rst_n: coverpoint seq_item.rst_n {
                bins reset = {0};                       // Reset signal is active
                bins no_reset = {1};                    // Reset signal is inactive
            }

            // Address signal coverage
            address: coverpoint seq_item.address {
                bins slave_0 = {4000};                    // Address for slave 0
                bins slave_1 = {4001};                    // Address for slave 1
                bins others = default;                    // Other addresses
            }

            // Process signal coverage
            process: coverpoint seq_item.process {
                bins load = {7'b0000011};                 // Load operation
                bins store = {7'b0100011};                // Store operation
                bins others = default;                  // Other instructions
            }

            // Data size signal coverage
            data_size: coverpoint seq_item.data_size {
                bins byte_ = {0};                       // Byte data size
                bins half_word = {1};                   // Half-word data size
                bins word = {2};                        // Word data size
            }

            // Cross coverage
            cross valid, rst_n;                         // Cross coverage for valid and reset signals
            cross address, process;                     // Cross coverage for address and process signals

        endgroup // apb_coverage covergroup
```

```
        // Constructor
        function new(string name = "apb_coverge", uvm_component parent = null);
            super.new(name, parent);                        // Call parent constructor
            apb_coverage = new();                           // Initialize covergroup
        endfunction // new function

        // Build phase
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);                       // Call parent build phase
            cov_exp = new("cov_exp", this);                 // Create analysis export
            cov_apb = new("cov_apb", this);                 // Create TLM analysis
        endfunction // build_phase function

        // Connect phase
        function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);                     // Call parent connect phase
            cov_exp.connect(cov_apb.analysis_export);       // Connect analysis export to TLM analysis
        endfunction // connect_phase function

        // Run phase
        task run_phase(uvm_phase phase);
            super.run_phase(phase);                         // Call parent run phase
            forever begin
                cov_apb.get(seq_item);                      // Get sequence item from TLM analysis
                apb_coverage.sample();                      // Sample the covergroup
            end
        endtask // run_phase task

    endclass // apb_coverge class

endpackage // apb_coverage_pkg
```

5.8 UVM scoreboard

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_scoreboard
////////////////////////////////////////////////////////////////////////////////

package apb_scoreboard_pkg;

    // Import required packages
    import uvm_pkg::*;                                  // UVM library
    import apb_sequence_item_pkg::*;                    // APB sequence item definitions
    `include "uvm_macros.svh";                          // UVM macro definitions

    // Scoreboard class definition
    class apb_scoreboare extends uvm_scoreboard;

        // Register the scoreboard with the UVM factory
        `uvm_component_utils(apb_scoreboare);

        // Parameter definitions for APB interface properties
        parameter ADDR_WIDTH = 32;                      // Address bus width
        parameter DATA_WIDTH = 32;                      // Data bus width
        parameter STRB_WIDTH = DATA_WIDTH / 8;          // Width of byte strobe signals
        parameter IDLE_PHASE = 2'b00;                   // Identifier for the Idle state
        parameter SETUP_PHASE = 2'b01;                  // Identifier for the Setup state
        parameter ACCESS_PHASE = 2'b10;                 // Identifier for the Access state

        // Internal signal declarations representing APB interface signals
        logic [ADDR_WIDTH-1:0] master_out_ref;          // Reference for master output
        logic [DATA_WIDTH-1:0] slave_out_ref;           // Reference for slave write data
        logic [DATA_WIDTH-1:0] PRDATA_ref;              // Reference for slave read data
        logic [ADDR_WIDTH-1:0] PADDR_ref;               // Reference for transaction address
        logic [DATA_WIDTH-1:0] PWDATA_ref;              // Reference for data to be written to slave
        logic [STRB_WIDTH-1:0] PSTRB_ref;               // Reference for byte strobe signals
        logic [2:0] PPROT_ref;                          // Reference for protection bits
        logic PSEL1_ref;                                // Reference for peripheral select signal 1
        logic PSEL0_ref;                                // Reference for peripheral select signal 0
        logic PENABLE_ref;                              // Reference for the enable signal
        logic PWRITE_ref;                               // Reference for the write control signal
        logic PREADY_ref;                               // Reference for slave ready signal
        logic end_access;                               // Signal to indicate end of access phase
        logic setup_rst_ideal;                          // Signal to indicate reset during setup phase

        // State machine variables for transaction phases
        logic [1:0] current_state;                      // Current FSM state
        logic [1:0] next_state;                         // Next FSM state

        // Counters for tracking transaction matches
        int right_count = 0;                            // Count of matching transactions (golden reference)
        int wrong_count = 0;                            // Count of mismatches detected

        // Object declaration for APB sequence item transactions
        apb_sequence_item seq_item;

        // TLM export declarations for analysis transactions
        uvm_analysis_export #(apb_sequence_item) sb_exp;
        uvm_tlm_analysis_fifo #(apb_sequence_item) sb_apb;

        // Constructor
        function new(string name = "apb_scoreboare", uvm_component parent = null);
            super.new(name, parent);
        endfunction // new

        // Build phase: Create analysis exports and FIFOs
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            sb_exp = new("sb_exp", this);               // Instantiate analysis export for APB transactions
            sb_apb = new("sb_apb", this);               // Instantiate TLM FIFO for APB transactions
        endfunction // build_phase
```

```
// Connect phase: Connect analysis export to TLM FIFO
function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    sb_exp.connect(sb_apb.analysis_export);       // Connect export to the analysis FIFO's export
endfunction // connect_phase

// Run phase: Process incoming transactions and compare with golden reference
task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
        sb_apb.get(seq_item);                   // Receive sequence item from the monitor
        golden_ref(seq_item);                   // Update golden reference based on received item

        // Compare DUT output with the golden reference signals
        if ((master_out_ref === seq_item.master_out) &&
            (slave_out_ref === seq_item.slave_out) &&
            (PRDATA_ref === seq_item.PRDATA) &&
            (PADDR_ref === seq_item.PADDR) &&
            (PWDATA_ref === seq_item.PWDATA) &&
            (PSTRB_ref === seq_item.PSTRB) &&
            (PPROT_ref === seq_item.PPROT) &&
            (PSEL1_ref === seq_item.PSEL1) &&
            (PSEL0_ref === seq_item.PSEL0) &&
            (PENABLE_ref === seq_item.PENABLE) &&
            (PWRITE_ref === seq_item.PWRITE) &&
            (PREADY_ref === seq_item.PREADY)) begin
                right_count++;                  // Increment match counter
            `uvm_info("run_phase", $sformatf("Golden reference matches the DUT output, right_count = %0d, wrong_count = %0d, time = %0t", right_count,wrong_count,$time), UVM_MEDIUM);
        end else begin
            wrong_count++;                      // Increment mismatch counter
            `uvm_error("run_phase", $sformatf("Mismatch detected: master_out_ref = %0h, slave_out_ref = %0h, PRDATA_ref = %0h, PADDR_ref = %0d, PWDATA_ref = %0h,
            PSTRB_ref = %0b, PPROT_ref = %0b, PSEL0_ref = %0b, PSEL1_ref = %0b, PENABLE_ref = %0b, PWRITE_ref = %0b, PREADY_ref = %0b, right_count = %0d, wrong_count = %0d, time = %0t",
                                    master_out_ref, slave_out_ref, PRDATA_ref, PADDR_ref, PWDATA_ref, PSTRB_ref, PPROT_ref, PSEL0_ref, PSEL1_ref, PENABLE_ref, PWRITE_ref, PREADY_ref, right_count, wrong_count, $time));
        end
    end
endtask // run_phase
```

```
// Golden reference function: Update expected signal values based on transaction data
function void golden_ref(apb_sequence_item seq_item);

    // If reset is asserted, clear outputs and return to IDLE state
    if (!seq_item.rst_n) begin
        master_out_ref = 0;                         // Clear master output reference
        slave_out_ref = 0;                          // Clear slave output reference
        if (current_state == SETUP_PHASE) begin
            setup_rst_ideal = 1;                    // Indicate reset during setup
        end
        current_state = IDLE_PHASE;                 // Reset FSM to IDLE state
    end
    else begin
        // State machine transitions for APB transaction phases
        case (current_state)
            IDLE_PHASE: begin
                ideal_phase();                      // Process idle phase behavior
                if (seq_item.process == (7'b0000011) || seq_item.process == (7'b0100011)) begin
                    current_state = SETUP_PHASE;  // Transition to SETUP phase for valid operations
                end else begin
                    current_state = IDLE_PHASE;   // Remain in IDLE state
                end
            end
            SETUP_PHASE: begin
                setup_phase();                      // Process setup phase behavior
                current_state = ACCESS_PHASE;  // Transition to ACCESS phase
            end
            ACCESS_PHASE: begin
                access_phase();                     // Process access phase behavior
                if (PREADY_ref && (seq_item.process == (7'b0000011) || seq_item.process == (7'b0100011))) begin
                    current_state = SETUP_PHASE;  // Transition to SETUP phase when ready
                    end_access = 1;                 // Signal end of access
                end else if (PREADY_ref && (seq_item.process != (7'b0000011) || seq_item.process != (7'b0100011))) begin
                    current_state = IDLE_PHASE;   // Transition back to IDLE state when ready
                    end_access = 1;                 // Signal end of access
                end else begin
                    current_state = ACCESS_PHASE; // Remain in ACCESS phase
                end
            end
            default: current_state = IDLE_PHASE;    // Default to IDLE state
        endcase
    end
```

```
    // Ensure correct phase functions are called based on current state
    case (current_state)
        IDLE_PHASE: begin
            ideal_phase();                      // Call idle phase function
        end
        SETUP_PHASE: begin
           setup_phase();                       // Call setup phase function
        end
        ACCESS_PHASE: begin
            access_phase();                      // Call access phase function
        end
        default: begin
            PADDR_ref = 0;                      // Clear address reference
            PWDATA_ref = 0;                     // Clear write data reference
            PSTRB_ref = 0;                      // Clear strobe reference
            PPROT_ref = 0;                      // Clear protection bits reference
            PSEL1_ref = 0;                      // Deassert peripheral select 1
            PSEL0_ref = 0;                      // Deassert peripheral select 0
            PENABLE_ref = 0;                    // Clear enable signal
            PWRITE_ref = 0;                     // Clear write control signal
        end
    endcase

endfunction // golden_ref
```

```
// Ideal phase function: Manage idle state activities and initial signal setup
function ideal_phase;
    if (setup_rst_ideal) begin
        // Update outputs based on end-of-access conditions
        if (end_access && PWRITE_ref) begin
            slave_out_ref = PWDATA_ref & {{8{PSTRB_ref[3]}},
                                          {8{PSTRB_ref[2]}},
                                          {8{PSTRB_ref[1]}},
                                          {8{PSTRB_ref[0]}}}; // Update slave output for write transactions
        end

        if (end_access && !PWRITE_ref) begin
            master_out_ref = seq_item.PRDATA;   // Capture read data for read transactions
        end

        end_access = 0;                         // Clear end access signal

        // Setup initial signal values based on sequence item
        PADDR_ref = seq_item.address;           // Assign transaction address
        PENABLE_ref = 0;                        // Deassert enable signal
        PPROT_ref = 3'b000;                     // Set protection bits to default level
        PWDATA_ref = seq_item.master_in;        // Assign write data from sequence item
        PREADY_ref = 0;                         // Clear ready signal
        // Select peripheral based on address value
        if (seq_item.address == 4000) begin
            PSEL1_ref = 0;
            PSEL0_ref = 1;
        end else if (seq_item.address == 4001) begin
            PSEL1_ref = 1;
            PSEL0_ref = 0;
        end else begin
            PSEL1_ref = 0;
            PSEL0_ref = 0;
        end
        // Configure operation type (read or write) and strobe signals
        if (seq_item.process == 7'b0000011) begin        // For read operation
            PWRITE_ref = 0;                         // Set as read
            PSTRB_ref = 4'b0000;                    // No strobe for reads
        end else if (seq_item.process == 7'b0100011) begin // For write operation
            PWRITE_ref = 1;                         // Set as write
            case (seq_item.data_size)
                2'b00:                          // Byte access
                    PSTRB_ref = (4'b0001 << seq_item.address[1:0]);
                2'b01:                          // Half-word access
                    PSTRB_ref = (4'b0011 << {seq_item.address[1],1'b0});
                2'b10:                          // Word access
                    PSTRB_ref = 4'b1111;
                default: PSTRB_ref = 4'b1111;
            endcase
        end

        setup_rst_ideal = 0;                    // Clear the setup reset flag

    end

    // Finalize idle phase signals
    end_access = 0;                         // Clear end access signal
    PSEL0_ref = 0;                          // Deassert peripheral select 0
    PSEL1_ref = 0;                          // Deassert peripheral select 1
    PENABLE_ref = 0;                        // Deassert enable signal
    PREADY_ref = 0;                         // Clear ready signal
    PRDATA_ref = 0;                         // Clear read data reference

endfunction
```

```
// Setup phase function: Prepare transaction signals before data access
function setup_phase;
    // Update outputs if an access has just finished
    if (end_access && PWRITE_ref) begin
        slave_out_ref = PWDATA_ref & {{8{PSTRB_ref[3]}},
                                      {8{PSTRB_ref[2]}},
                                      {8{PSTRB_ref[1]}},
                                      {8{PSTRB_ref[0]}}}; // Process write data update
    end

    if (end_access && !PWRITE_ref) begin
        master_out_ref = seq_item.PRDATA;   // Capture read data from sequence item
    end

    end_access = 0;                         // Reset end access signal

    // Set up transaction signals based on the current sequence item
    PADDR_ref = seq_item.address;           // Set the target address
    PENABLE_ref = 0;                        // Ensure enable is deasserted during setup
    PPROT_ref = 3'b000;                     // Default protection level
    PWDATA_ref = seq_item.master_in;        // Capture write data
    PREADY_ref = 0;                         // Clear the ready signal
    // Choose the appropriate peripheral based on the address
    if (seq_item.address == 4000) begin
        PSEL1_ref = 0;
        PSEL0_ref = 1;
    end else if (seq_item.address == 4001) begin
        PSEL1_ref = 1;
        PSEL0_ref = 0;
    end else begin
        PSEL1_ref = 0;
        PSEL0_ref = 0;
    end
    // Define operation mode and strobe configuration based on process type
    if (seq_item.process == 7'b0000011) begin        // Read operation
        PWRITE_ref = 0;
        PSTRB_ref = 4'b0000;
    end else if (seq_item.process == 7'b0100011) begin // Write operation
        PWRITE_ref = 1;
        case (seq_item.data_size)
            2'b00:                          // Byte access
                PSTRB_ref = (4'b0001 << seq_item.address[1:0]);
            2'b01:                          // Half-word access
                PSTRB_ref = (4'b0011 << {seq_item.address[1],1'b0});
            2'b10:                          // Word access
                PSTRB_ref = 4'b1111;
            default: PSTRB_ref = 4'b1111;
        endcase
    end
endfunction
```

```
    // Access phase function: Activate transaction transfer signals
    function access_phase;
        PENABLE_ref = 1;                        // Assert the enable signal to initiate transfer
        PREADY_ref = (seq_item.valid && PENABLE_ref && seq_item.rst_n); // Generate ready signal based on transaction validity

        if (seq_item.valid && PENABLE_ref) begin
            PRDATA_ref = seq_item.slave_in;     // Capture data from slave during read operations
        end
    endfunction

    // Report phase: Output final scoreboard results
    function void report_phase(uvm_phase phase);
        super.report_phase(phase);
        `uvm_info("report_phase", "Scoreboard report phase completed", UVM_MEDIUM);
    endfunction // report_phase

endclass // apb_scoreboare

endpackage // apb_scoreboard_pkg
```

5.9 UVM driver

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_driver
////////////////////////////////////////////////////////////////////////////////

package apb_driver;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    import apb_sequence_item_pkg::*;                    // Import APB sequence item package
    import apb_config_obj::*;                           // Import APB configuration object package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_driver extends uvm_driver #(apb_sequence_item);

        // Factory registration
        `uvm_component_utils(apb_driver);

        // Virtual interface declaration
        virtual apb_int apb_driver_vif;                 // Virtual interface for APB signals

        // Object declaration
        apb_sequence_item seq_item;                     // Sequence item object

        // Constructor
        function new(string name = "apb_driver", uvm_component parent = null);
            super.new(name, parent);                    // Call parent constructor
        endfunction // new function

        // Build phase
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);                   // Call parent build phase
        endfunction // build_phase function

        // Run phase
        task run_phase(uvm_phase phase);
            super.run_phase(phase);                     // Call parent run phase

            forever begin
                // Object creation
                seq_item = apb_sequence_item::type_id::create("seq_item", this);

                // Start requesting items
                seq_item_port.get_next_item(seq_item);      // Request the next item

                // Assign sequence item values to the virtual interface
                apb_driver_vif.master_in = seq_item.master_in; // Assign master input data
                apb_driver_vif.slave_in = seq_item.slave_in;   // Assign slave input data
                apb_driver_vif.address = seq_item.address;     // Assign address
                apb_driver_vif.process = seq_item.process;     // Assign process control signal
                apb_driver_vif.data_size = seq_item.data_size; // Assign data size
                apb_driver_vif.rst_n = seq_item.rst_n;         // Assign reset signal
                apb_driver_vif.valid = seq_item.valid;         // Assign valid signal

                // Wait for the negative edge of the clock
                @(negedge apb_driver_vif.clk);

                // End the current item
                seq_item_port.item_done();                  // Mark the item as done

                // Log the stimulus details
                `uvm_info("run_phase", seq_item.convert2string_stimulus(), UVM_MEDIUM);

                #0; // Small delay
            end
        endtask // run_phase function

    endclass // apb_driver class

endpackage // apb_driver package
```

5.10 UVM monitor

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_monitor
////////////////////////////////////////////////////////////////////////////////

package apb_monitor_pkg;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    import apb_sequence_item_pkg::*;                    // Import APB sequence item package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_monitor extends uvm_monitor;

        // Factory registration
        `uvm_component_utils(apb_monitor)

        // Virtual interface declaration
        virtual apb_int apb_monitor_vif;                // Virtual interface for APB signals

        // Object declaration
        apb_sequence_item seq_item;                     // Sequence item object

        // TLM analysis port declaration
        uvm_analysis_port #(apb_sequence_item) mon_p;   // Analysis port for broadcasting sequence items

        // Constructor
        function new(string name = "apb_monitor", uvm_component parent = null);
            super.new(name, parent);                    // Call parent constructor
        endfunction // new function

        // Build phase
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);                   // Call parent build phase
            mon_p = new("mon_p", this);                 // Create analysis port
        endfunction // build_phase function

        // Run phase
        task run_phase(uvm_phase phase);
            super.run_phase(phase);                     // Call parent run phase
```

```
        forever begin
            // Object creation
            seq_item = apb_sequence_item::type_id::create("seq_item", this);

            // Assigning the value of the interface to the sequence item object
            @(negedge apb_monitor_vif.clk);
            seq_item.master_in = apb_monitor_vif.master_in;   // Capture master input data
            seq_item.slave_in = apb_monitor_vif.slave_in;     // Capture slave input data
            seq_item.address = apb_monitor_vif.address;       // Capture address
            seq_item.master_out = apb_monitor_vif.master_out; // Capture master output data
            seq_item.slave_out = apb_monitor_vif.slave_out;   // Capture slave output data
            seq_item.process = apb_monitor_vif.process;       // Capture process control signal
            seq_item.data_size = apb_monitor_vif.data_size;   // Capture data size
            seq_item.rst_n = apb_monitor_vif.rst_n;           // Capture reset signal
            seq_item.valid = apb_monitor_vif.valid;           // Capture valid signal
            seq_item.PWDATA = apb_monitor_vif.PWDATA;         // Capture write data for slave
            seq_item.PSTRB = apb_monitor_vif.PSTRB;           // Capture byte strobe signals
            seq_item.PRDATA = apb_monitor_vif.PRDATA;         // Capture data read from slave
            seq_item.PADDR = apb_monitor_vif.PADDR;           // Capture address for the transaction
            seq_item.PPROT = apb_monitor_vif.PPROT;           // Capture protection bits
            seq_item.PSEL1 = apb_monitor_vif.PSEL1;           // Capture peripheral select 1
            seq_item.PSEL0 = apb_monitor_vif.PSEL0;           // Capture peripheral select 0
            seq_item.PENABLE = apb_monitor_vif.PENABLE;       // Capture enable signal
            seq_item.PWRITE = apb_monitor_vif.PWRITE;         // Capture write signal
            seq_item.PREADY = apb_monitor_vif.PREADY;         // Capture ready signal from slave
            seq_item.clk = apb_monitor_vif.clk;               // Capture clock signal

            // Broadcasting the sequence item object
            mon_p.write(seq_item);                            // Write sequence item to analysis port
            `uvm_info("run_phase", seq_item.convert2string(), UVM_MEDIUM); // Log sequence item details
        end
    endtask // run_phase function

endclass // apb_monitor class

endpackage // apb_monitor_pkg
```

5.11 UVM agent

```
////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_agent
////////////////////////////////////////////////////////////////////////////

package apb_agent_pkg;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    import apb_sequencer_pkg::*;                        // Import APB sequencer package
    import apb_monitor_pkg::*;                          // Import APB monitor package
    import apb_driver::*;                               // Import APB driver package
    import apb_config_obj::*;                           // Import APB configuration object package
    import apb_sequence_item_pkg::*;                    // Import APB sequence item package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_agent extends uvm_agent;

        // Factory registration
        `uvm_component_utils(apb_agent);

        // Object declarations
        apb_driver driver;                              // APB driver
        apb_sequencer sequencer;                        // APB sequencer
        apb_monitor monitor;                            // APB monitor
        apb_config_obj apb_config_obj_agent;            // APB configuration object

        // Analysis port declaration
        uvm_analysis_port #(apb_sequence_item) agent_p;     // Analysis port for broadcasting sequence items

        // Constructor
        function new(string name = "apb_agent", uvm_component parent = null);
            super.new(name, parent);                    // Call parent constructor
        endfunction // new function

        // Build phase
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);                   // Call parent build phase

            // Agent port creation
            agent_p = new("agent_p", this);             // Create analysis port

            // Create objects
            driver = apb_driver::type_id::create("driver", this); // Create APB driver
            monitor = apb_monitor::type_id::create("monitor", this); // Create APB monitor
            sequencer = apb_sequencer::type_id::create("sequencer", this); // Create APB sequencer
            apb_config_obj_agent = apb_config_obj::type_id::create("apb_config_obj_agent", this); // Create configuration object

            // Get the configuration object (interface)
            uvm_config_db#(apb_config_obj)::get(this, "", "interface_test", apb_config_obj_agent);
        endfunction // build_phase function

        // Connect phase
        function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);                 // Call parent connect phase

            // TLM port-export connection
            driver.seq_item_port.connect(sequencer.seq_item_export); // Connect driver to sequencer

            // Interface connection
            driver.apb_driver_vif = apb_config_obj_agent.apb_config_vif; // Connect driver to virtual interface
            monitor.apb_monitor_vif = apb_config_obj_agent.apb_config_vif; // Connect monitor to virtual interface

            // TLM analysis port-port connection
            monitor.mon_p.connect(agent_p);             // Connect monitor analysis port to agent analysis port
        endfunction // connect_phase function

    endclass // apb_agent class

endpackage // apb_agent package
```

5.12 UVM environment

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_env
////////////////////////////////////////////////////////////////////////////////

package apb_env;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    import apb_agent_pkg::*;                            // Import APB agent package
    import apb_scoreboard_pkg::*;                       // Import APB scoreboard package
    import apb_coverage_pkg::*;                         // Import APB coverage package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_env extends uvm_env;

        // Factory registration
        `uvm_component_utils(apb_env);

        // Object declarations
        apb_agent agent;                                // APB agent
        apb_scoreboare sb;                              // APB scoreboard
        apb_coverge cov;                                // APB coverage

        // Constructor
        function new(string name = "apb_env", uvm_component parent = null);
            super.new(name, parent);                    // Call parent constructor
        endfunction // new function

        // Build phase
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);                   // Call parent build phase

            // Create objects
            agent = apb_agent::type_id::create("agent", this); // Create APB agent
            sb = apb_scoreboare::type_id::create("sb", this); // Create APB scoreboard
            cov = apb_coverge::type_id::create("cov", this);  // Create APB coverage
        endfunction // build_phase function

        // Connect phase
        function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);                 // Call parent connect phase

            // TLM analysis port-export connection
            agent.agent_p.connect(sb.sb_exp);           // Connect agent analysis port to scoreboard export
            agent.agent_p.connect(cov.cov_exp);         // Connect agent analysis port to coverage export
        endfunction // connect_phase function

    endclass // apb_env class

endpackage // apb_env package
```

5.13 UVM test

```
////////////////////////////////////////////////////////////////////////////////
// Name: Abdelrahman Mohamed
// Module-Name: apb_test
////////////////////////////////////////////////////////////////////////////////

package apb_test_pkg;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    import apb_env::*;                                  // Import APB environment package
    import apb_config_obj::*;                           // Import APB configuration object package
    import apb_main_sequence_pkg::*;                    // Import APB main sequence package
    import apb_rst_sequence_pkg::*;                     // Import APB reset sequence package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_test extends uvm_test;

        // Factory registration
        `uvm_component_utils(apb_test);

        // Object declarations
        apb_env env;                                    // APB environment
        apb_config_obj apb_config_obj_test;             // APB configuration object
        apb_main_sequence main_sequence;                // APB main sequence
        apb_rst_sequence reset_sequence;                // APB reset sequence

        // Constructor
        function new(string name = "apb_test", uvm_component parent = null);
            super.new(name, parent);                    // Call parent constructor
        endfunction // new function

        // Build phase
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);                   // Call parent build phase

            // Object creation
            env = apb_env::type_id::create("env", this);    // Create APB environment
            main_sequence = apb_main_sequence::type_id::create("main_sequence"); // Create main sequence
            reset_sequence = apb_rst_sequence::type_id::create("reset_sequence"); // Create reset sequence
            apb_config_obj_test = apb_config_obj::type_id::create("apb_config_obj_test", this); // Create configuration object

            // Get the interface from the top
            uvm_config_db#(virtual apb_int)::get(this, "", "interface", apb_config_obj_test.apb_config_vif);

            // Set the configuration object
            uvm_config_db#(apb_config_obj)::set(this, "*", "interface_test", apb_config_obj_test);
        endfunction // build_phase function

        // Run phase
        task run_phase(uvm_phase phase);
            super.run_phase(phase);                     // Call parent run phase
            phase.raise_objection(this);                // Raise objection to keep the simulation running

            // Start the reset sequence
            reset_sequence.start(env.agent.sequencer);      // Start reset sequence driving
            `uvm_info("run_phase", "Finish first test", UVM_MEDIUM);

            // Start the main sequence
            main_sequence.start(env.agent.sequencer);       // Start main sequence driving
            `uvm_info("run_phase", "Finish second test", UVM_MEDIUM);

            phase.drop_objection(this);                 // Drop objection to end the simulation
        endtask // run_phase task

    endclass // apb_test class

endpackage // apb_test_pkg
```

5.14 UVM config object

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_config_obj
////////////////////////////////////////////////////////////////////////////////

package apb_config_obj;

    // Package imports
    import uvm_pkg::*;                                  // Import UVM package
    `include "uvm_macros.svh";                          // Include UVM macros

    // Class definition
    class apb_config_obj extends uvm_object;

        // Factory registration
        `uvm_object_utils(apb_config_obj);

        // Virtual interface declaration
        virtual apb_int apb_config_vif;                 // Virtual interface for APB signals

        // Constructor
        function new(string name = "apb_config_obj");
            super.new(name);                            // Call parent constructor
        endfunction // new function

    endclass // apb_config_obj class

endpackage // apb_config_obj package
```

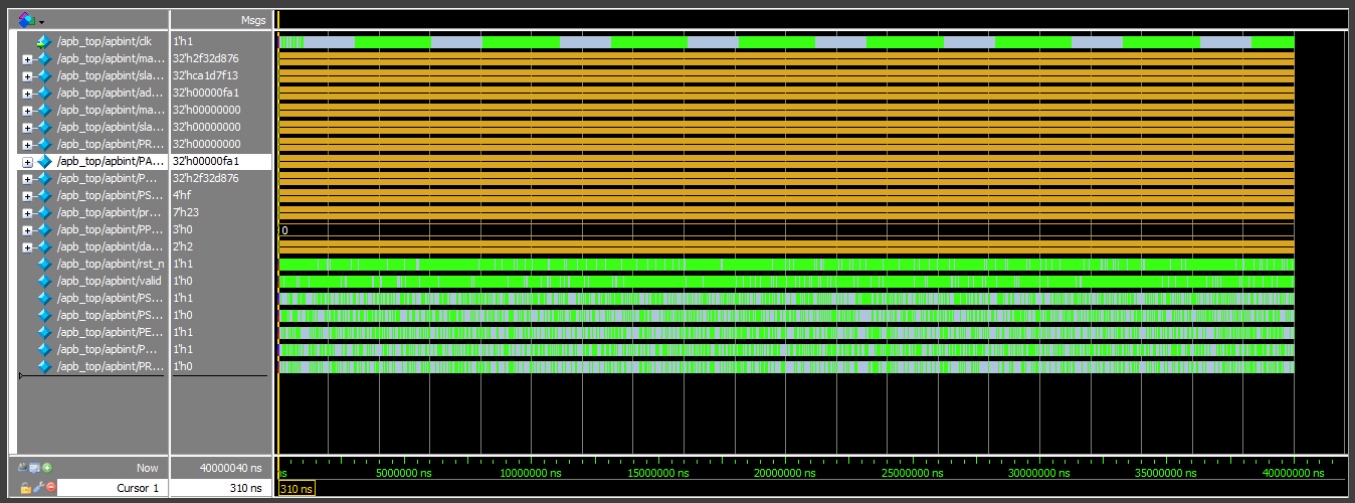
5.15 APB interface

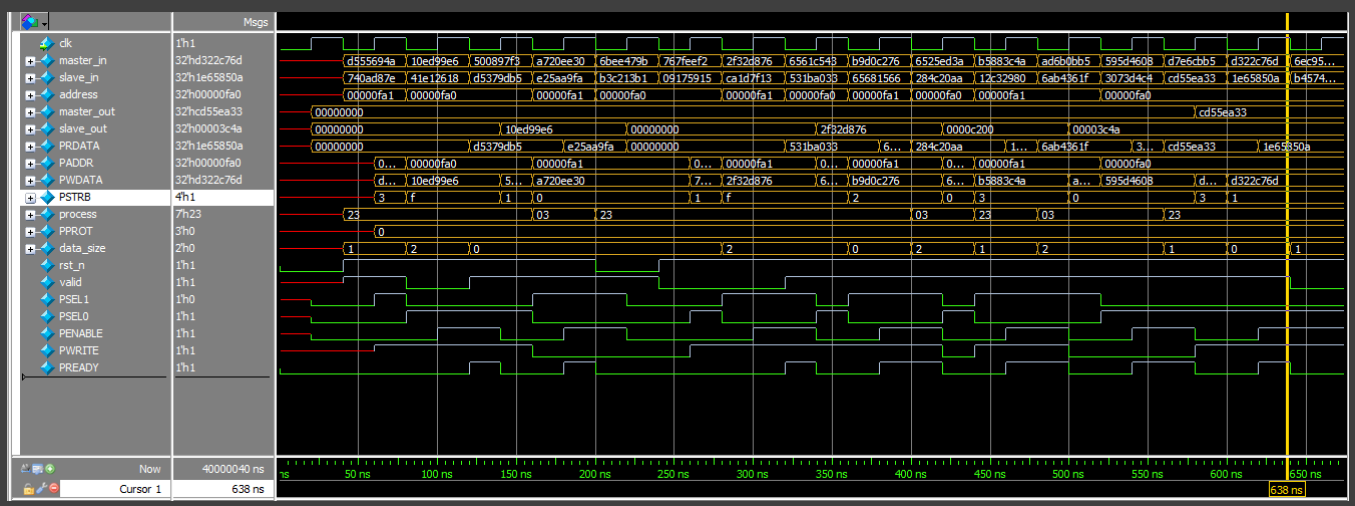
```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_int
////////////////////////////////////////////////////////////////////////////////

interface apb_int (clk);

    // Parameters
    parameter ADDR_WIDTH = 32;                          // Width of the address bus
    parameter DATA_WIDTH = 32;                          // Width of the data bus
    parameter STRB_WIDTH = DATA_WIDTH / 8;              // Byte strobe width

    // Input signals
    input bit clk;                                      // Clock signal

    // Internal signals
    logic [DATA_WIDTH-1:0] master_in;                   // Data read from slave
    logic [DATA_WIDTH-1:0] slave_in;                    // Data to be written to slave
    logic [ADDR_WIDTH-1:0] address;                     // Address for the transaction
    logic [ADDR_WIDTH-1:0] master_out;                  // APB address output
    logic [DATA_WIDTH-1:0] slave_out;                   // Write data output for APB
    logic [DATA_WIDTH-1:0] PRDATA;                      // Data read from slave
    logic [ADDR_WIDTH-1:0] PADDR;                       // Address for the transaction
    logic [DATA_WIDTH-1:0] PWDATA;                      // Write data for slave
    logic [STRB_WIDTH-1:0] PSTRB;                       // Byte strobe signals
    logic [6:0] process;                                // Process control signal
    logic [2:0] PPROT;                                  // Protection bits
    logic [1:0] data_size;                              // Data size signal
    logic rst_n;                                        // Reset (active low)
    logic valid;                                        // Valid signal
    logic PSEL1;                                        // Peripheral select 1
    logic PSEL0;                                        // Peripheral select 0
    logic PENABLE;                                      // Enable signal
    logic PWRITE;                                       // Write signal
    logic PREADY;                                       // Ready signal from slave

endinterface // apb_int
```

5.16 APB top

```
////////////////////////////////////////////////////////////////////////////////
/// Name: Abdelrahman Mohamed
/// Module-Name: apb_top
////////////////////////////////////////////////////////////////////////////////

import uvm_pkg::*;
import apb_test_pkg::*;
`include "uvm_macros.svh";

module apb_top ();

    // Internal signals
    bit clk;                                            // Clock signal

    // Clock generation
    initial begin
        clk = 0;                                        // Initialize clock to 0
        forever begin
            #20;                                        // Toggle clock every 20 time units
            clk = ~clk;
        end
    end

    // Modules instantiation
    apb_int apbint(clk);                                // Interface module
    apb_top_design apb_top_design(apbint);              // Design module

    // UVM instantiation
    initial begin
        uvm_config_db#(virtual apb_int)::set(null, "*", "interface", apbint); // Set virtual interface
        run_test("apb_test");                           // Run the UVM test
    end

endmodule
```

5.17 simulation result





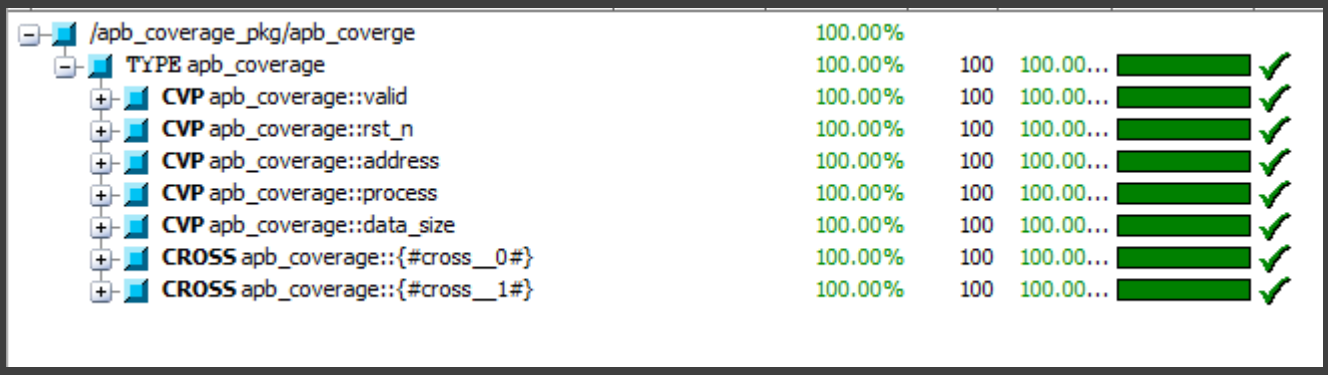

```
# UVM_INFO apb_monitor.sv(72) @ 40000040: uvm_test_top.env.agent.monitor [run_phase]  master_in = 0h996528c8, slave_in = 0h89298c14, address = 0d4001, process = 0b11, data_size = 0b10, valid = 0b1, rst_n = 0b1, master_out = 0h89298c14, slave_out = 0hfc58, PWDATA = 0h998528c8, PSTRB = 0b0, PRDATA = 0h89298c14, PADDR = 0d4001, PPROT = 0b0, PSEL1 = 0b1, PSEL0 = 0b0, PENABLE = 0b0, PWRITE = 0b0, PREADY = 0b0
# UVM_INFO apb_scoreboard.sv(97) @ 40000040: uvm_test_top.env.sb [run_phase] Golden reference matches the DUT output, right_count = 1000001, wrong_count = 0, time = 40000040
# UVM_INFO apb_test.sv(61) @ 40000040: uvm_test_top [run_phase] Finish second test
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 40000040: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO apb_scoreboard.sv(303) @ 40000040: uvm_test_top.env.sb [report_phase] Scoreboard report phase completed
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :3000010
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [report_phase]     1
# [run_phase] 3000005
# ** Note: $finish    : D:/programs/questasim64_2021.1/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 40000040 ns  Iteration: 61  Instance: /apb_top
```

```
Golden reference matches the DUT output, right_count = 1000001, wrong_count = 0, time = 40000040
```

```
# ** Report counts by severity
# UVM_INFO :3000010
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [report_phase]     1
# [run_phase] 3000005
```

As we can see all cases were right cases

5.18 coverage result



/apb_coverage_pkg/apb_coverge	100.00%			
TYPE apb_coverage	100.00%	100	100.00...	✓
CVP apb_coverage::valid	100.00%	100	100.00...	✓
CVP apb_coverage::rst_n	100.00%	100	100.00...	✓
CVP apb_coverage::address	100.00%	100	100.00...	✓
CVP apb_coverage::process	100.00%	100	100.00...	✓
CVP apb_coverage::data_size	100.00%	100	100.00...	✓
CROSS apb_coverage::{#cross__0#}	100.00%	100	100.00...	✓
CROSS apb_coverage::{#cross__1#}	100.00%	100	100.00...	✓

As we can see also 100% functional coverage

```
================================================================================
Toggle Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                        554       554         0    100%

--------------------------------------------------------------------------------
Branch Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Branches                         30        30         0    100%

Condition Coverage:
    Enabled Coverage              Bins   Covered    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Conditions                       13        13         0  100.00%

FSM Coverage:
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    FSM States                        3         3         0  100.00%
    FSM Transitions                   5         5         0  100.00%
```

As we can see also 100% code coverage

6. References

- AMBA® APB Protocol Version: 2.0 From ARM