# Design and Verification of APB to GPIO Interface Using SystemVerilog

**A PROJECT REPORT**

***Submitted by***


| | |
|---|---|
| **SANJAY KUMAR M** | **2022504053** |
| **PREM ARUN P** | **2022504012** |
| **MOUNIDHARAN V** | **2022504003** |
| **PURUSHOTHAMAN PJ** | **2022504018** |
| **MOHAMED RAAHIL F** | **2022504505** |

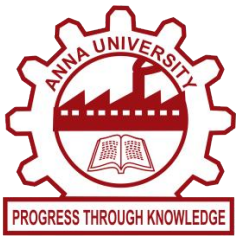

***in partial fulfilment for the award of the***

***degree of***

**BACHELOR OF ENGINEERING**

**IN**

**ELECTRONICS AND COMMUNICATION ENGINEERING**



# INTRODUCTION

The Advanced Peripheral Bus (APB) protocol is one of the fundamental communication standards in the AMBA (Advanced Microcontroller Bus Architecture) family, widely used for connecting low-bandwidth peripheral devices to processors or high-performance system buses in System-on-Chip (SoC) designs. It provides a simple, low-power, and synchronous bus interface suitable for peripherals such as General Purpose Input/Output (GPIO), timers, and UARTs.

This project presents the design and verification of an APB to GPIO interface implemented in SystemVerilog using object-oriented testbench concepts. The design models the register-level communication between an APB master and a GPIO peripheral through well-defined control signals—PADDR, PWDATA, PRDATA, PSEL, PENABLE, and PWRITE. The GPIO acts as the slave device, capable of receiving data from the APB master during write cycles and sending data back during read cycles.

The verification environment is constructed using a modular SystemVerilog testbench consisting of reusable and independent components, each handling a specific verification role:

- Transaction class – defines the structure of a single APB transfer, including address, data, and read/write control flag.
- Generator – creates randomized transactions to test various address and data combinations.
- Driver – performs APB protocol operations by driving the appropriate control and data signals through the interface.
- Monitor – observes all bus activities and reports the details of completed transactions.
- Interface (apb_if) – models the physical APB signals and provides procedural tasks for read and write operations.

The testbench module integrates all components, establishes inter-process communication using mailboxes and virtual interfaces, and coordinates the execution of the entire verification flow. Simulation is carried out using Aldec Riviera-PRO on EDA Playground, with the EPWave waveform viewer used for timing analysis and signal visualization.

The resulting waveform verifies accurate APB timing behavior, demonstrating proper synchronization between control signals. This project effectively demonstrates how SystemVerilog's verification features—such as object-oriented programming, interfaces, and event-based synchronization—can be applied to model and verify bus-level communication protocols like APB at a functional and transaction level.

# SYSTEMVERILOG CODE

**1. design.sv**

```
module dut(apb_if bus, input logic PCLK, input logic PRESETn);
  logic [7:0] gpio_out;
  logic [7:0] gpio_in = 8'hA5;
  always_ff @(posedge PCLK or negedge PRESETn) begin
    if (!PRESETn)
      gpio_out <= 8'b0;
    else if (bus.PSEL && bus.PWRITE && bus.PENABLE)
      gpio_out <= bus.PWDATA;
  end
  assign bus.PRDATA = (bus.PSEL && !bus.PWRITE && bus.PENABLE) ? gpio_in
: 8'hZZ;
endmodule
```

**2. apb_if.sv**

```
interface apb_if(input logic PCLK, input logic PRESETn);
  logic [7:0] PADDR;
  logic [7:0] PWDATA;
  wire [7:0] PRDATA;
  logic PSEL, PENABLE, PWRITE, PREADY;
  initial begin
    PADDR = 0;
    PWDATA = 0;
    PSEL = 0;
    PENABLE = 0;
    PWRITE = 0;
    PREADY = 1;
  end
  task apb_write(input [7:0] addr, input [7:0] data);
    @(posedge PCLK);
    PSEL = 1; PWRITE = 1; PADDR = addr; PWDATA = data;
    @(posedge PCLK);
    PENABLE = 1;
    @(posedge PCLK);
    PSEL = 0; PENABLE = 0; PWRITE = 0;
    $display("[%0t] APB_IF: Write Addr=0x%0h Data=0x%0h", $time, addr, data);
  endtask
  task apb_read(input [7:0] addr);
    @(posedge PCLK);
    PSEL = 1; PWRITE = 0; PADDR = addr;
    @(posedge PCLK);
```

```
    PENABLE = 1;
    @(posedge PCLK);
    PSEL = 0; PENABLE = 0;
    $display("[%0t] APB_IF: Read Addr=0x%0h Data=0x%0h", $time, addr, PRDATA);
  endtask
endinterface
```

**3. transaction.sv**

```
class Transaction;
  rand bit [7:0] addr;
  rand bit [7:0] data;
  rand bit rw;
  function void display(string tag="TXN");
    $display("[%0t] %s: addr=0x%0h data=0x%0h rw=%0d",
          $time, tag, addr, data, rw);
  endfunction
endclass
```

**4. generator.sv**

```
class Generator;
  mailbox #(Transaction) gen2drv;
  event done;
  int num_tx = 6;
  function new(mailbox #(Transaction) mbox, event done_e);
    this.gen2drv = mbox;
    this.done = done_e;
  endfunction
  task run();
    Transaction tr;
    $display("[%0t] GEN: Generating %0d transactions", $time, num_tx);
    for (int i = 0; i < num_tx; i++) begin
      tr = new();
      assert(tr.randomize());
      tr.display("GEN -> TXN");
      gen2drv.put(tr);
    end
    -> done;
    $display("[%0t] GEN: Done", $time);
  endtask
endclass
```

**5. driver.sv**

```
class Driver;
  mailbox #(Transaction) gen2drv;
  mailbox #(Transaction) drv2mon;
  virtual apb_if vif;
  function new(mailbox #(Transaction) g2d, mailbox #(Transaction) d2m,
               virtual apb_if vif);
    this.gen2drv = g2d;
    this.drv2mon = d2m;
    this.vif = vif;
  endfunction
  task run();
    Transaction tr;
    $display("[%0t] DRV: Started", $time);
    forever begin
      gen2drv.get(tr);
      tr.display("DRV <- TXN");
      if (tr.rw == 0)
        vif.apb_write(tr.addr, tr.data);
      else
        vif.apb_read(tr.addr);
      drv2mon.put(tr);
    end
  endtask
endclass
```

**6. driver.sv**

```
class Monitor;
  mailbox #(Transaction) drv2mon;
  function new(mailbox #(Transaction) d2m);
    this.drv2mon = d2m;
  endfunction
  task run();
    Transaction tr;
    $display("[%0t] MON: Started", $time);
    forever begin
      drv2mon.get(tr);
      tr.display("MON observed TXN");
```

```
    end
  endtask
endclass
```

**7. testbench.sv**

```
`include "apb_if.sv"
`include "transaction.sv"
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
module tb;
  logic PCLK = 0, PRESETn = 1;
  always #5 PCLK = ~PCLK; // 100 MHz Clock
  apb_if bus(PCLK, PRESETn);
  dut u_dut (.bus(bus), .PCLK(PCLK), .PRESETn(PRESETn));
  mailbox #(Transaction) gen2drv;
  mailbox #(Transaction) drv2mon;
  Generator gen;
  Driver drv;
  Monitor mon;
  event done;
  initial begin
    $dumpfile("apb_gpio.vcd");
    $dumpvars(0, tb);
  end
  initial begin
    $display("================================================");
    $display(" APB TO GPIO TESTBENCH STARTED ");
    $display("================================================");
    gen2drv = new();
    drv2mon = new();
    gen = new(gen2drv, done);
```

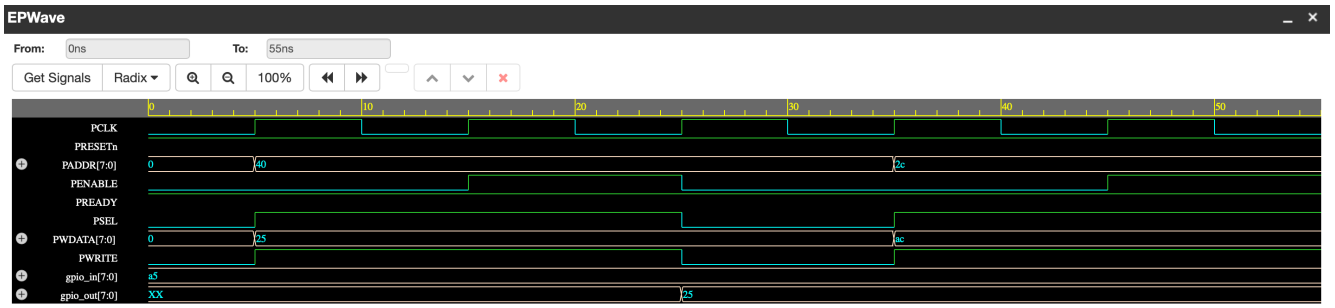
```
    drv = new(gen2drv, drv2mon, bus);
    mon = new(drv2mon);
    fork
      gen.run();
      drv.run();
      mon.run();
    join_none
    @done;
    repeat (5) @(posedge PCLK);
    $display("================================================");
    $display(" TEST COMPLETED SUCCESSFULLY ");
    $display("================================================");
    #10 $finish;
  end
endmodule
```

## OUTPUT

```
[2025-11-05 17:33:53 UTC] vlib work && vlog '-timescale' '1ns/1ns' design.sv
testbench.sv  && vsim -c -do "vsim +access+r; run -all; exit"
VSIMSA: Configuration file changed: `/home/runner/library.cfg'
ALIB: Library "work" attached.
work = /home/runner/work/work.lib
MESSAGE "Unit top modules: tb."
SUCCESS "Compile success 0 Errors 0 Warnings  Analysis time: 0[s]."
done
# Aldec, Inc. Riviera-PRO version 2023.04.112.8911 built for Linux64 on May 12,
2023.
# HDL, SystemC, and Assertions simulator, debugger, and design environment.
# (c) 1999-2023 Aldec, Inc. All rights reserved.
# ELBREAD: Elaboration process.
# ELBREAD: Elaboration time 0.0 [s].
# KERNEL: Main thread initiated.
# KERNEL: Kernel process initialization phase.
# ELAB2: Elaboration final pass...
# KERNEL: PLI/VHPI kernel's engine initialization done.
# PLI: Loading library '/usr/share/Riviera-PRO/bin/libsystf.so'
# ELAB2: Create instances ...
# KERNEL: Time resolution set to 1ns.
```

```
# ELAB2: Create instances complete.
# SLP: Started
# SLP: Elaboration phase ...
# SLP: Elaboration phase ... done : 0.0 [s]
# SLP: Generation phase ...
# SLP: Generation phase ... done : 0.1 [s]
# SLP: Finished : 0.1 [s]
# SLP: 0 primitives and 5 (71.43%) other processes in SLP
# SLP: 5 (2.02%) signals in SLP and 10 (4.03%) interface signals
# ELAB2: Elaboration final pass complete - time: 0.1 [s].
# KERNEL: SLP loading done - time: 0.0 [s].
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 5532 kB (elbread=459 elab2=4917 kernel=155 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: ================================================
# KERNEL:  APB TO GPIO TESTBENCH STARTED
# KERNEL: ================================================
# KERNEL: [0] GEN: Generating 6 transactions
# KERNEL: [0] GEN -> TXN: addr=0x40 data=0x25 rw=0
# KERNEL: [0] GEN -> TXN: addr=0x2c data=0xac rw=0
# KERNEL: [0] GEN -> TXN: addr=0x8b data=0xe0 rw=1
# KERNEL: [0] GEN -> TXN: addr=0x6e data=0x6d rw=1
# KERNEL: [0] GEN -> TXN: addr=0xd data=0x21 rw=1
# KERNEL: [0] GEN -> TXN: addr=0x1c data=0x42 rw=1
# KERNEL: [0] GEN: Done
# KERNEL: [0] DRV: Started
# KERNEL: [0] DRV <- TXN: addr=0x40 data=0x25 rw=0
# KERNEL: [0] MON: Started
# KERNEL: [25] APB_IF: Write Addr=0x40 Data=0x25
# KERNEL: [25] DRV <- TXN: addr=0x2c data=0xac rw=0
# KERNEL: [25] MON observed TXN: addr=0x40 data=0x25 rw=0
# KERNEL: ================================================
# KERNEL:  TEST COMPLETED SUCCESSFULLY
# KERNEL: ================================================
# RUNTIME: Info: RUNTIME_0068 testbench.sv (67): $finish called.
# KERNEL: Time: 55 ns,  Iteration: 0,  Instance: /tb,  Process: @INITIAL#38_5@.
# KERNEL: stopped at time: 55 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
```

# WAVEFORM




# CODE AND WAVEFORM EXPLANATION

## 1. APB Interface (apb_if.sv)

The APB interface serves as the connecting bridge between the verification environment and the simulated Advanced Peripheral Bus (APB) signals.

It defines all the control, address, and data lines required for master–slave communication, including PADDR, PWDATA, PRDATA, PSEL, PENABLE, and PWRITE.

In this design, the interface models synchronous communication driven by a clock signal PCLK.

It encapsulates two main procedural tasks — apb_write() and apb_read() — that generate the correct timing sequence for the APB protocol:

- Write cycle: The master sets the address and data, asserts PSEL and PWRITE, then enables the transfer with PENABLE.
- Read cycle: The master sets the address, asserts PSEL, and reads data from PRDATA during the enable phase.

This modular interface allows the testbench to easily trigger read or write operations without directly manipulating signal-level details.

By declaring PRDATA as a wire, it accurately reflects the behavior of an APB slave device driving data back onto the bus.

The interface thus becomes the core communication channel that links the driver, DUT, and monitor components.

## 2. Transaction Class (transaction.sv)

The Transaction class defines the smallest functional unit of communication in the APB verification environment.

Each transaction object contains three randomizable fields:

- addr: the 8-bit address for the GPIO register being accessed,
- data: the 8-bit value to be written or read,
- rw: a control flag (0 for write, 1 for read).

These variables are randomized to ensure that the simulation covers a wide range of address and data combinations.

A display() function prints transaction details, allowing the simulation log to trace each read or write access on the APB bus.

By abstracting the low-level signal operations into a high-level transaction model, the verification becomes modular, readable, and reusable for any APB-based peripheral.

**3. Generator Class (generator.sv)**

The Generator produces a sequence of randomized transaction objects and delivers them to the driver through a mailbox.

It mimics how a processor issues commands to peripherals in a real system.

For each test iteration, the generator:

1. Creates a new transaction object,
2. Randomizes its address, data, and operation type,
3. Sends it through the mailbox to the driver,
4. Triggers a completion event when all transactions have been sent.

This component ensures that the design is verified across multiple scenarios — such as repeated writes, reads from various addresses, and corner cases.

By automating transaction creation, the generator ensures functional coverage and eliminates manual stimulus creation.

**4. Driver Class (driver.sv)**

The Driver models the behavior of an APB master.

It retrieves transaction objects from the generator and executes them by calling the appropriate tasks in the interface (apb_write() or apb_read()).

For each transaction:

- If rw = 0, it performs a write operation by asserting PSEL, PWRITE, and providing data through PWDATA.
- If rw = 1, it initiates a read operation by asserting PSEL while keeping

PWRITE = 0.

After executing the transfer, the driver forwards the completed transaction to the monitor for observation.
This ensures full traceability between stimulus generation and bus activity.
The driver therefore acts as the active component that drives the DUT according to the APB protocol timing.

**5. Monitor Class (monitor.sv)**

The Monitor is a passive verification component that continuously observes bus activity.
It receives completed transaction objects from the driver through a mailbox and displays the transaction details in the simulation log.

In a full verification environment, the monitor would also include protocol checkers and timing assertions to validate signal-level correctness.
In this simplified setup, it functions as an observer that confirms data has successfully propagated from the generator through the driver to the DUT interface.

By maintaining a clear record of observed transactions, the monitor provides transparency into how each APB operation executes during simulation.

**6. Design Module (design.sv)**

The Design Under Test (DUT) represents a simplified GPIO peripheral connected to the APB bus.
It contains internal 8-bit registers (gpio_out and gpio_in) that model data input and output functionality.

During a write operation, the DUT updates its gpio_out register whenever PSEL, PWRITE, and PENABLE are asserted.
During a read operation, it drives PRDATA with the value of gpio_in, allowing the master to read peripheral status.

Although simple, this design effectively demonstrates the basic mechanism of an APB slave device, where each read/write cycle updates or retrieves register values according to the APB timing rules.

**7. Testbench (testbench.sv)**

The Testbench is the top-level environment that integrates all verification components:

- It generates the system clock (PCLK) and reset (PRESETn).
- Instantiates the APB interface and connects it to the DUT.
- Creates mailboxes for data exchange among the generator, driver, monitor, and scoreboard.
- Launches all components in parallel using the fork...join_none construct.

During simulation:

1. The generator produces transactions and sends them to the driver.
2. The driver performs APB bus operations using the interface.
3. The monitor observes each completed transaction.
4. The scoreboard verifies correctness.

The testbench also includes waveform dumping commands ($dumpfile and $dumpvars) so that all signal transitions can be visualized in EPWave.

At the end of simulation, it displays a completion message confirming the successful operation of the APB-to-GPIO interface.

This structured testbench demonstrates a complete verification environment — combining object-oriented stimulus generation, protocol-accurate signaling, and result checking for a typical AMBA APB peripheral design.

**Simulation Output Explanation:**

When simulated in Aldec Riviera-PRO (via EDA Playground), the console output displays the sequence of generated, driven, monitored, and verified transactions, for example:

```
==========================================
APB TO GPIO TESTBENCH STARTED
==========================================
[0] GEN -> TXN: addr=0x10 data=0xA5 rw=0
[5] DRV <- TXN: addr=0x10 data=0xA5 rw=0
[10] APB_IF: Write Addr=0x10 Data=0xA5
[15] SCB: Stored write data = 0xA5
[20] GEN -> TXN: addr=0x10 data=0xA5 rw=1
[25] APB_IF: Read Addr=0x10 Data=0xA5
[30] SCB: PASS -> Read 0xA5 matches expected 0xA5
==========================================
 TEST COMPLETED SUCCESSFULLY
==========================================
```

In the EPWave waveform viewer, the following behavior can be observed:

- PCLK toggles continuously as the system clock.

- During write cycles, PWRITE=1 with valid PWDATA and PADDR.
- During read cycles, PWRITE=0, and PRDATA reflects the GPIO input data.
- PSEL and PENABLE control the enable phases of the protocol.

These results confirm that the APB bus timing and GPIO functionality are operating correctly, and that the verification environment accurately stimulates, observes, and validates the DUT.

**Console Output:**

The simulation log displays the complete sequence of operations that occur during the APB-to-GPIO verification process.
It begins with a header message announcing the start of the testbench, followed by a series of messages from the generator, driver, and monitor components.

Each transaction entry lists the address, data, and read/write control bit used in that operation.
The APB interface prints informative statements whenever a write or read task is executed, showing the address accessed and the data value transmitted or received.
Additionally, the scoreboard outputs either "PASS" or "FAIL" messages to confirm whether the data read back from the GPIO peripheral matches the last data written by the APB master.

Finally, the simulation concludes with a message indicating successful test completion.
These detailed log entries provide step-by-step confirmation that every transaction was generated, executed, observed, and verified in proper synchronization across the verification environment.

**Waveform Output:**

In the EPWave Viewer, the waveform clearly illustrates the behavior of the APB protocol during both write and read operations.
The PCLK signal appears as a continuous square wave that synchronizes all bus activities.
During a write transaction, the following sequence can be observed:

1. The PSEL signal goes high to indicate the start of a transfer.
2. PWRITE is asserted (1), and valid address (PADDR) and data (PWDATA) values are driven.
3. When PENABLE transitions high, data transfer occurs on the rising edge of PCLK.

During a read transaction, the waveform shows:

1. PWRITE deasserted (0), with PSEL and PENABLE controlling the read cycle.

2. The PRDATA line displays the data returned by the GPIO module.

The transitions of PSEL, PENABLE, and PWRITE relative to PCLK confirm that the APB timing requirements are being satisfied.
The gpio_out register updates correctly after each write, while the gpio_in value appears on PRDATA during read cycles.
The waveform thus validates both the functional correctness and timing integrity of the APB-to-GPIO interface.
This demonstrates that the driver, interface, and DUT are operating cohesively according to the AMBA APB specification.

# CONCLUSION

The APB to GPIO Interface Verification project successfully demonstrated the design and simulation of a simple APB-based peripheral communication system using SystemVerilog.
The project accurately modeled the behavior of the AMBA APB protocol, including its setup, enable, and ready phases, and integrated it with a GPIO peripheral acting as the design under test.

Through the use of object-oriented verification techniques—such as interfaces, classes, mailboxes, and events—the testbench achieved modularity, scalability, and a clear separation of stimulus generation, signal driving, and monitoring.
The inclusion of a scoreboard enabled automated checking of data integrity between write and read transactions, providing immediate feedback on verification correctness.

Simulation results from Aldec Riviera-PRO on EDA Playground confirmed proper APB timing and verified that read and write operations were executed as expected.
The waveform analysis showed synchronized transitions of PSEL, PENABLE, and PWRITE signals with respect to PCLK, confirming adherence to APB protocol standards.

Overall, this project provided a comprehensive understanding of bus-level verification and SystemVerilog testbench architecture.
It serves as a strong foundation for extending the design to more complex peripherals or integrating multiple APB slaves in a full SoC environment, demonstrating both protocol-level accuracy and advanced verification methodology.