**CHAPTER 1**

**INTRODUCTION & THEORETICAL BACKGROUND**

* 1. **ARM**

Advanced RISC Machine, is a family of reduced instruction set computing (RISC) architectures for computer processors, configured for various environments. Arm Holdings develops the architecture and licenses it to other companies, who design their own products that implement one of those architectures including systems-on-chips (SoC) and systems-on-modules (SoM) that incorporate memory, interfaces, radios, etc. It also designs cores that implement this instruction set and licenses these designs to a number of companies that incorporate those core designs into their own products.

Processors that have a RISC architecture typically require fewer transistors than those with a complex instruction set computing (CISC) architecture (such as the x86 processors found in most personal computers), which improves cost, power consumption, and heat dissipation. These characteristics are desirable for light, portable, battery-powered devices—including smartphones, laptops and tablet computers, and other embedded systems but are also useful for servers and desktops to some degree. For supercomputers, which consume large amounts of electricity, Arm is also a power-efficient solution.

Arm Holdings periodically releases updates to the architecture. Architecture versions Armv3 to Armv7 support 32-bit address space (pre-Armv3 chips, made before Arm Holdings was formed, as used in the Acorn Archimedes, had 26-bit address space) and 32-bit arithmetic; most architectures have 32-bit fixed-length instructions. The Thumb version supports a variable-length instruction set that provides both 32- and 16-bit instructions for improved code density. Some older cores can also provide hardware execution of Java bytecodes; and newer ones have one instruction for JavaScript. Released in 2011, the Armv8-A architecture added support for a 64-bit address space and 64-bit arithmetic with its new 32-bit fixed-length instruction set. Some recent Arm CPUs have simultaneous multithreading (SMT) with e.g. Arm Neoverse E1 being able to execute two threads concurrently for improved aggregate throughput performance. Arm Cortex-A65AE for automotive applications is also a multithreaded processor, and has Dual Core Lock-Step for fault-tolerant designs (supporting Automotive Safety Integrity Level D, the highest level). The Neoverse N1 is designed for "as few as 8 cores" or "designs that scale from 64 to 128 N1 cores within a single coherent system.

* 1. **ARM AMBA**

****

**Fig. 1.1 AMBA Specifications**

AMBA (Advanced Microcontroller Bus Architecture) is a freely-available, open standard for the connection and management of functional blocks in a system-on-chip (SoC). It facilitates right-first-time development of multi-processor designs, with large numbers of controllers and peripherals.

AMBA specifications are royalty-free, platform-independent and can be used with any processor architecture. Due to its widespread adoption, AMBA has a robust ecosystem of partners that ensures compatibility and scalability between IP components from different design teams and vendors.

**1.3 APB (Advanced Peripheral Bus):**

The Advanced Peripheral Bus (APB) is part of the Advanced Microcontroller Bus Architecture (AMBA) protocol family. It defines a low-cost interface that is optimized for minimal power consumption and reduced interface complexity.

The APB protocol is not pipelined, use it to connect to low-bandwidth peripherals that do not require the high performance of the AXI protocol.

The APB protocol relates a signal transition to the rising edge of the clock, to simplify the integration of APB peripherals into any design flow. Every transfer takes at least two cycles.

The APB can interface with:

• AMBA Advanced High-performance Bus (AHB)

• AMBA Advanced High-performance Bus Lite (AHB-Lite)

• AMBA Advanced Extensible Interface (AXI)

• AMBA Advanced Extensible Interface Lite (AXI4-Lite)

You can use it to access the programmable control registers of peripheral devices.

The APB data bus is split into two separate directions:

• read (PRDATA), where data travels from the peripherals to the bridge

• write (PWDATA), where data travels from the bridge to the peripherals.

This simplifies driving the buses because turnaround time between the peripherals and bridge is avoided.

In the default system, because the bridge is the only master on the bus, PWDATA is driven continuously. PRDATA is a multiplexed connection of all peripheral PRDATA outputs on the bus, and is only driven when the slaves are selected by the bridge during APB read transfers.

It is possible to combine these two buses into a single bidirectional bus, but precautions must be taken to ensure that there is no bus clash between the bridge and the peripherals.

**1.4 AMBA Architecture**

1.AMBA Overview:

* AMBA is an open standard developed by ARM for connecting and managing functional blocks within a system-on-a-chip (SoC).
* It facilitates the development of multiprocessor designs with numerous controllers and peripherals, aiming for right-first-time development and reusability.

2. AMBA Buses:

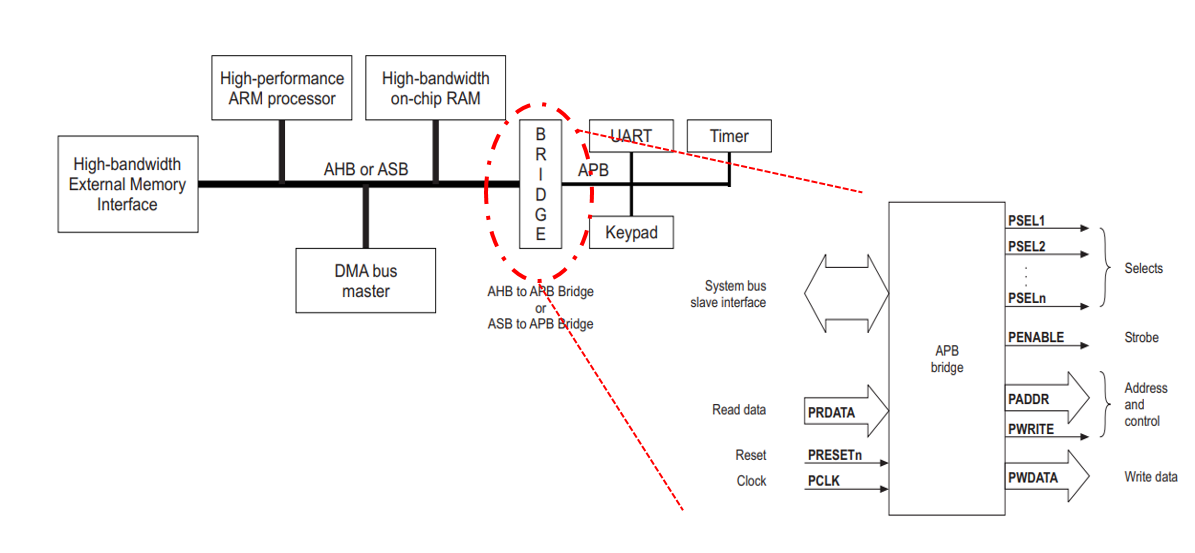
AMBA comprises three buses:

* Advanced High-Performance Bus (AHB): The backbone of the system, designed for high-performance, high-frequency components. It connects processors, on-chip memories, and memory interfaces.
* Advanced System Bus (ASB): An alternative to AHB, used when some high-performance features are unnecessary.
* Advanced Peripheral Bus (APB): A simplified interface for low-bandwidth peripherals that don’t require the performance of AHB or ASB. Examples include UARTs, low-frequency GPIO, and timers.

3. APB Characteristics:

* Compact and Low Power: APB is highly compact and consumes minimal power.
* Configuration and Low-Bandwidth Traffic: APB handles configuration registers and low-bandwidth data traffic in peripherals2.

In summary, the AMBA architecture provides a standardized way to connect and manage components in an SoC, ensuring efficient communication and reusability. The APB serves as a lightweight interface for low-bandwidth peripherals.

****

**Fig 1.2 AMBA Bus Architecture**

**1.5. Signal Description**

|  |  |  |
| --- | --- | --- |
| **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. |
| **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. |
| **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. |

**Chapter 2**

**Literature Survey**

**2.1 Introduction**

This chapter presents a comprehensive review of literature relevant to the design and verification of AMBA (Advanced Microcontroller Bus Architecture) protocols, focusing on the AHB (Advanced High-performance Bus) to APB (Advanced Peripheral Bus) bridge. The literature review encompasses research papers, journal articles, and conference proceedings that provide insights into different aspects of AMBA protocol design, implementation, and verification.

**2.2 APB Protocol Design and Verification**

Vaishnavi R.K et al. [2] presented a study on the design and verification of the APB protocol using SystemVerilog and Universal Verification Methodology (UVM). Their work emphasizes the importance of rigorous verification techniques to ensure protocol compliance and functional correctness.

Kommirisetti Bheema Raju and Bala Krishna Konda [3] also addressed the design and verification of the AMBA APB protocol, highlighting the significance of systematic verification methodologies in ensuring the reliability and robustness of communication protocols.

**2.3 FPGA Implementation and Performance Optimization**

M. Kiran Kumar et al. [4] focused on the FPGA implementation of an AMBA APB bridge with clock skew minimization techniques to enhance system performance. Their work demonstrates practical strategies for optimizing hardware designs to meet performance requirements.

**2.4 RTL Implementation and System-Level Integration**

Kiran Rawat et al. [5] presented an RTL implementation for the AMBA ASB APB protocol at the system-on-chip level. Their work addresses the challenges of integrating different bus protocols within complex system architectures.

Roopa.M et al. [6] proposed the design of low bandwidth peripherals using high-performance bus architecture, highlighting the importance of efficient peripheral design in maximizing system performance.

**2.5 Summary**

In this chapter, we have reviewed a range of literature sources related to the design, verification, and implementation of AMBA protocols, with a specific focus on the AHB to APB bridge. The literature survey encompasses research papers and journal articles that provide valuable insights into various aspects of AMBA protocol design, FPGA implementation, performance optimization, and system-level integration. These studies serve as a foundation for the development of our approach and methodology in the subsequent chapters.

**CHAPTER 3**

**DESIGN**

**3.1 APB INTERFACE WITH SLAVES**

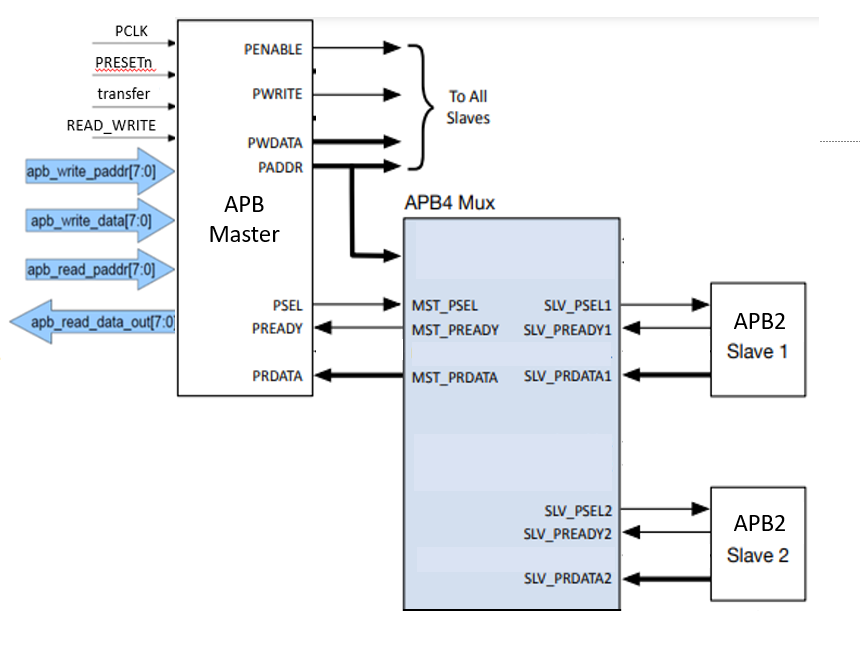
The APB data bus is split into two separate directions:

• read (PRDATA), where data travels from the peripherals to the bridge

• write (PWDATA), where data travels from the bridge to the peripherals.

This simplifies driving the buses because turnaround time between the peripherals and bridge is avoided.

In the default system, because the bridge is the only master on the bus, PWDATA is driven continuously. PRDATA is a multiplexed connection of all peripheral PRDATA outputs on the bus, and is only driven when the slaves are selected by the bridge during APB read transfers.



**Fig 3.1 APB Interface with Slaves**

**3.2 TRANSFERS**

This chapter describes typical AMBA 3 APB write and read transfers, and the error response. It contains the following sections:

• Write transfers

• Read transfers

**3.2.1 Write Transfers**

Two types of write transfer are described in this section:

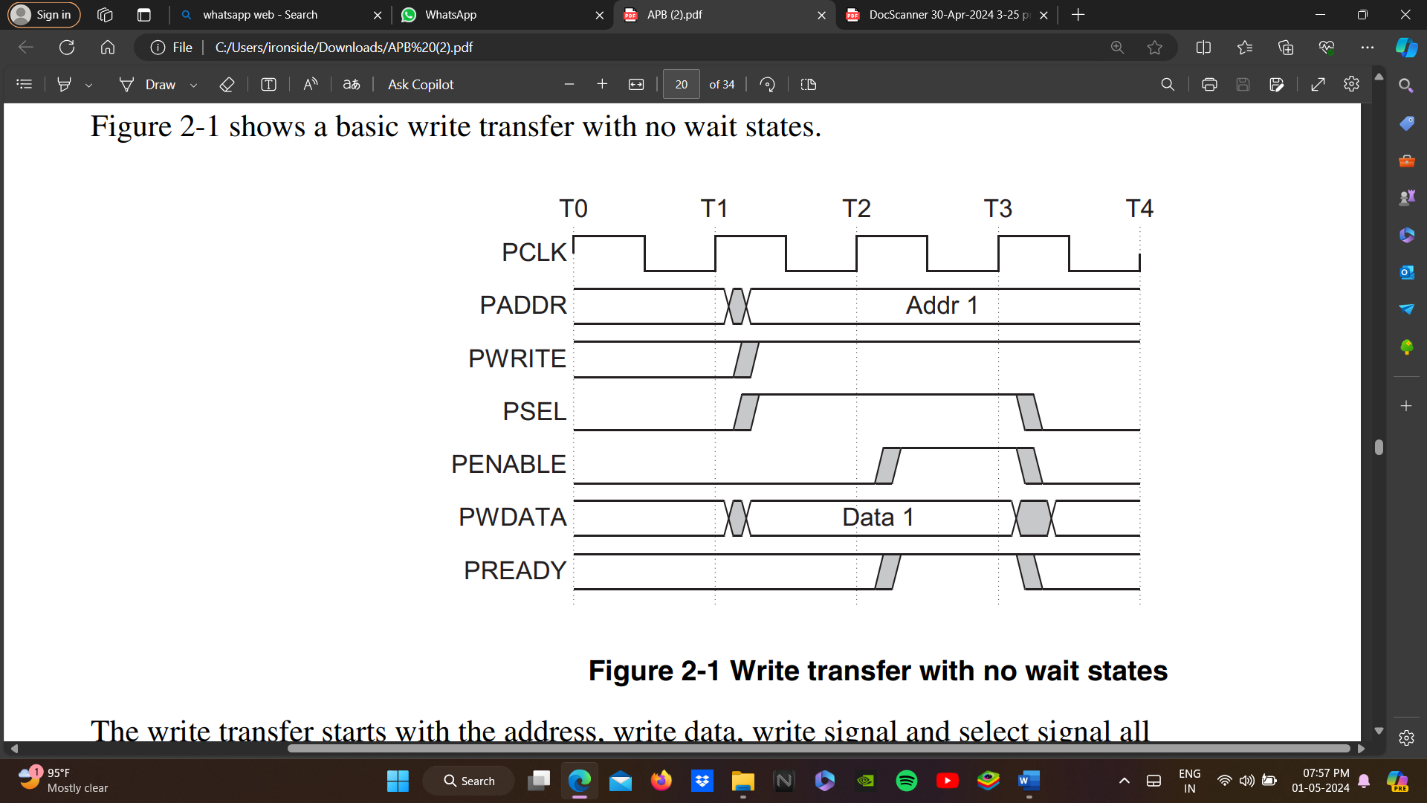
• With no wait states

• With wait states.

**With no wait states**

The write transfer starts with the address, write data, write signal and select signal all changing after the rising edge of the clock. The first clock cycle of the transfer is called the Setup phase. After the following clock edge the enable signal is asserted, PENABLE, and this indicates that the Access phase is taking place. The address, data and control signals all remain valid throughout the Access phase. The transfer completes at the end of this cycle.

The enable signal, PENABLE, is deasserted at the end of the transfer. The select signal, PSELx, also goes LOW unless the transfer is to be followed immediately by another transfer to the same peripheral

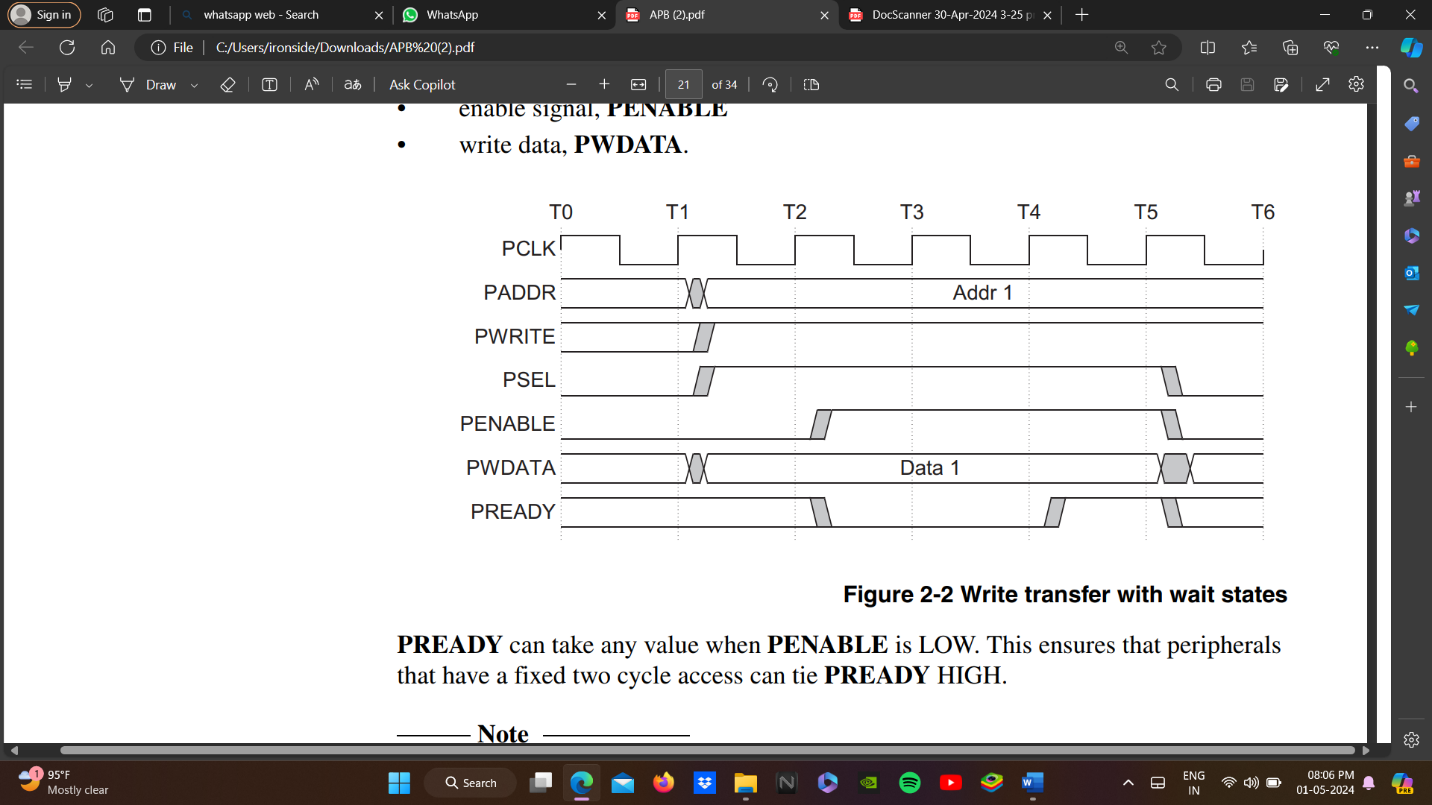


**Fig 3.2 Write transfer with no wait states**

**With wait states**

Figure 2.3 shows how the PREADY signal from the slave can extend the transfer. During an Access phase, when PENABLE is HIGH, the transfer can be extended by driving PREADY LOW. The following signals remain unchanged for the additional cycles:

* address, PADDR
* write signal, PWRITE
* select signal, PSEL
* enable signal, PENABLE
* write data, PWDATA.

****

**Fig 3.3 Write transfer with wait states**

PREADY can take any value when PENABLE is LOW. This ensures that peripherals that have a fixed two cycle access can tie PREADY HIGH.

**Note:** It is recommended that the address and write signals are not changed immediately after a transfer but remain stable until another access occurs. This reduces power consumption.

**3.2.2 Read transfers**

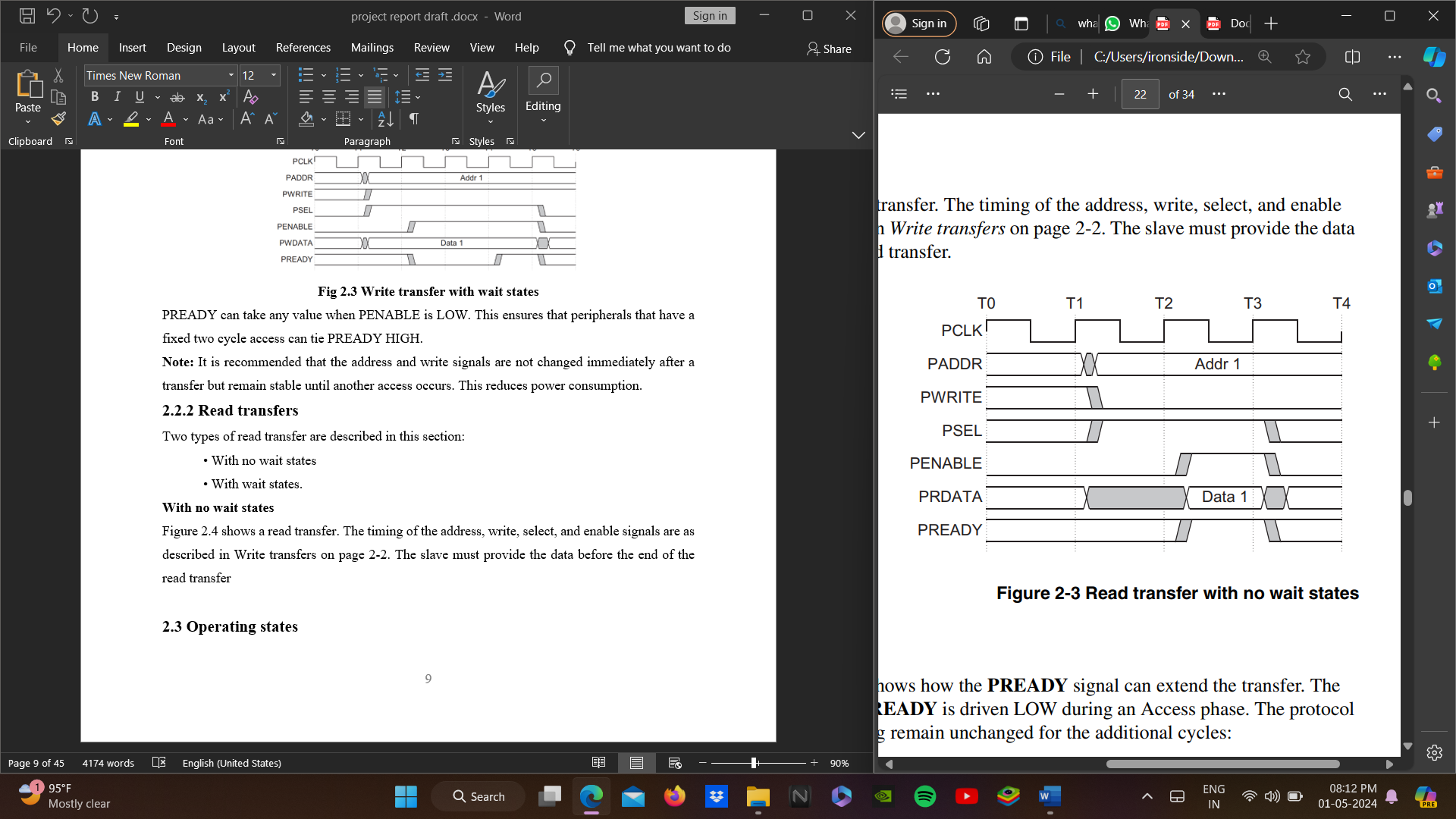
Two types of read transfer are described in this section:

• With no wait states

• With wait states.

**With no wait states**

Figure 2.4 shows a read transfer. The timing of the address, write, select, and enable signals are as described in Write transfers on page 2-2. The slave must provide the data before the end of the read transfer

****

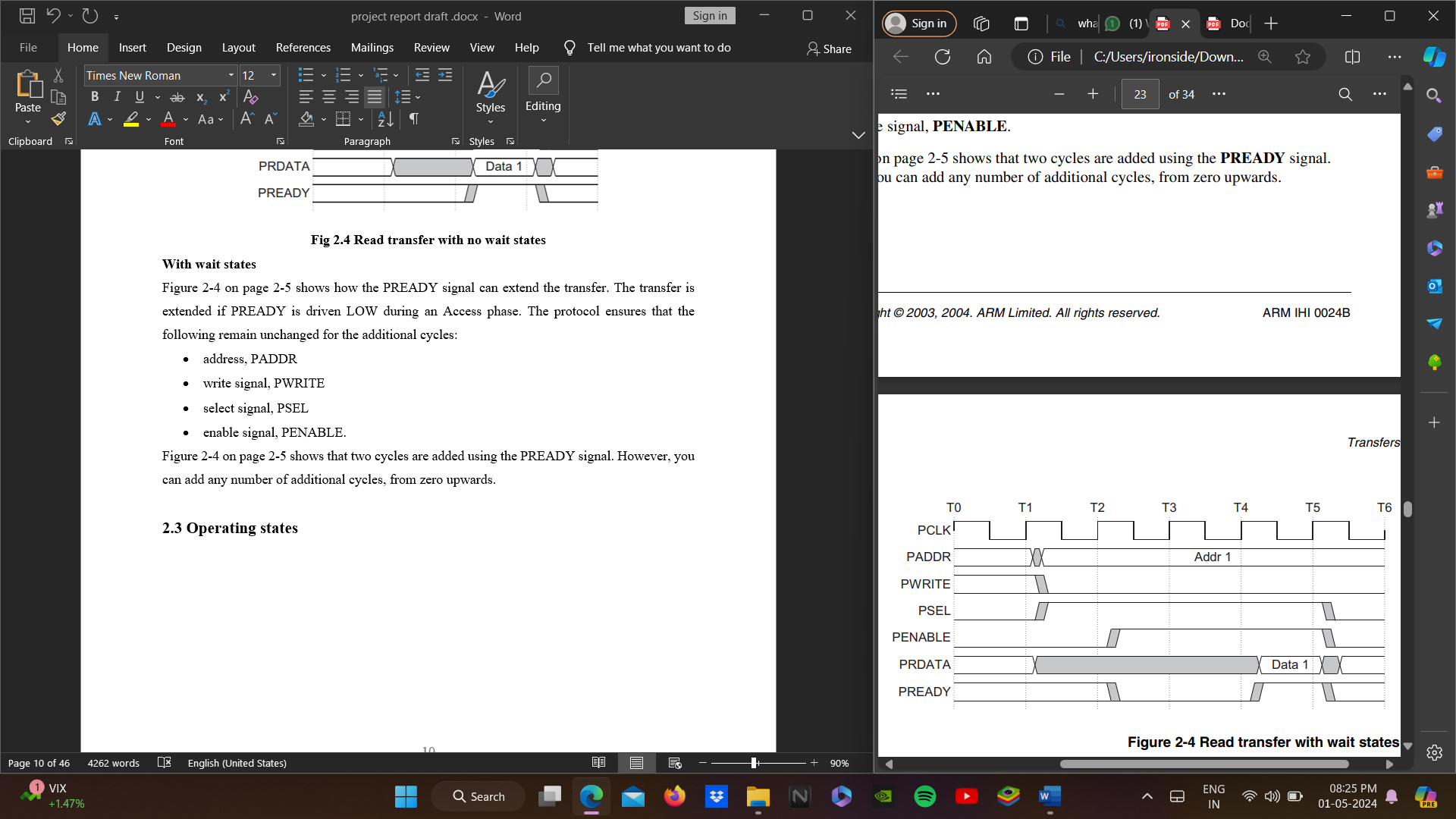
**Fig 3.4 Read transfer with no wait states**

**With wait states**

Figure 2.5 shows how the PREADY signal can extend the transfer. The transfer is extended if PREADY is driven LOW during an Access phase. The protocol ensures that the following remain unchanged for the additional cycles:

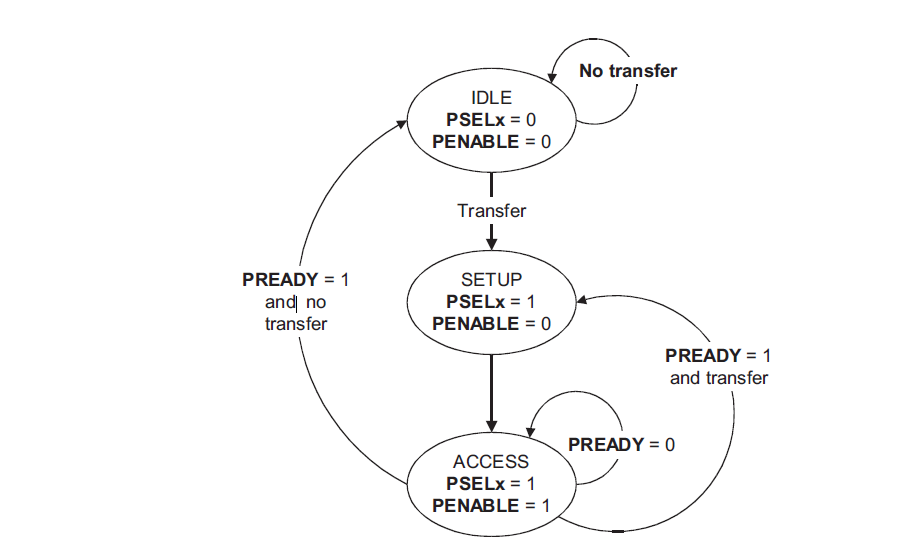
* address, PADDR
* write signal, PWRITE
* select signal, PSEL
* enable signal, PENABLE.

Figure 2.5 shows that two cycles are added using the PREADY signal. However, you can add any number of additional cycles, from zero upwards.



**Fig 3.5 Read transfer with wait states**

**3.3 OPERATING STATES**



**Fig 3.6 State diagram**

The state machine operates through the following states:

**1. IDLE** This is the default state of the APB.

**2. SETUP** When a transfer is required the bus moves into the SETUP state, where the appropriate select signal, **PSELx**, is asserted. The bus only remains in the SETUP state for one clock cycle and always moves to the ACCESS state on the next rising edge of the clock.

**3. ACCESS** The enable signal, **PENABLE**, is asserted in the ACCESS state. The address, write, select, and write data signals must remain stable during the transition from the SETUP to ACCESS state. Exit from the ACCESS state is controlled by the **PREADY** signal from the slave:

• If **PREADY** is held LOW by the slave then the peripheral bus remains in the ACCESS state.

**3.4 SYSTEM VERILOG**

**SystemVerilog**is an extension of **Verilog**. Gateway Design Automation introduced **Verilog**as an evolutionary HDL in **1985**.

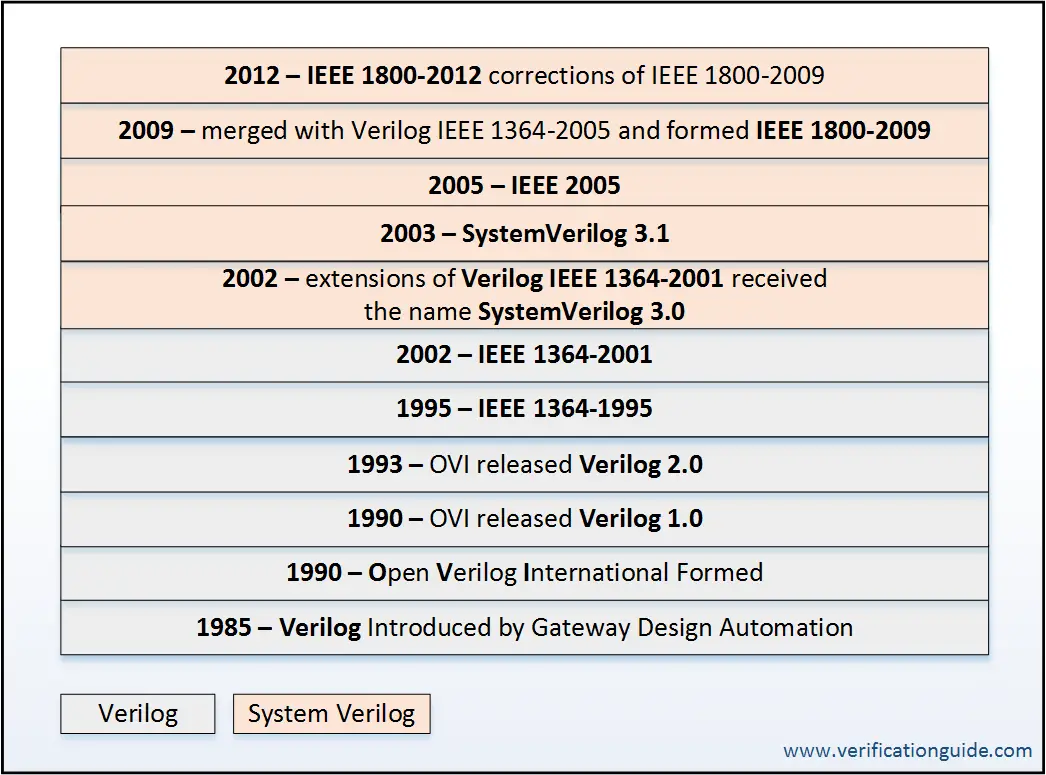
Verilog language stemmed primarily from two earlier languages,

* HILO-2
* Occam parallel-processing language

In **1990**, Cadence placed the Verilog language in the public domain, and **O**pen **V**erilog **I**nternational (OVI) formed to manage the language. Cadence released the **Verilog-XL** user manual as the basis for the ﬁrst **L**anguage **R**eference **M**anual. This manual became known as **OVI Verilog 1.0**. In **1993**, OVI released **Verilog 2.0** to the IEEE, and in **1995**this became **IEEE Std. 1364**. The IEEE working group released a **revised**standard in March of 2002, known as **IEEE 1364-2001**. A revised version was released in 2003, known as **IEEE 1364-2001 Revision C**. In **1998**, the original developers of Verilog and HILO-2 formed Co-Design Automation and created **Superlog**Language. **Superlog’s** goal is to integrate verification features into the Verilog language and create the ﬁrst hardware **design** and **veriﬁcation** language. The new LRM for **extensions**to **Verilog**received the name **SystemVerilog 3.0**, which Accellera approved as a standard in June **2002**. Concurrently, Synopsys announced that it was donating several verification technologies to the **SystemVerilog** effort. The donations included testbench constructs based on Vera, OpenVera assertions, Synopsys’ VCS DirectC simulation interface to C and C++, and a coverage application programming interface that provides links to coverage metrics. On 2 June **2003**, Accellera announced that its board and technical committee members had approved the **SystemVerilog 3.1** In **2005**, SystemVerilog was adopted as **IEEE**Standard **1800-2005** In **2009**, the standard was merged with the base Verilog (IEEE 1364-2005) standard, creating **IEEE**Standard **1800-2009**.Error corrections and clarification of a few aspects of IEEE Std 1800-2009 lead to the release of **IEEE**Standard **1800-20012**.The current version is the IEEE standard **1800-2017**.

SystemVerilog can be divided into two distinct based on its roles,

* SystemVerilog for **design**is an extension of Verilog-2005
* SystemVerilog for **verification**



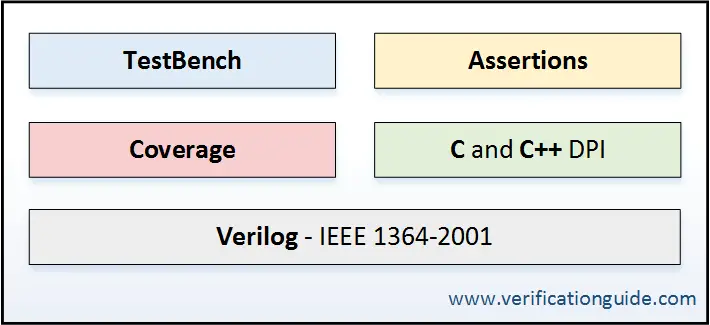
**Fig 3.7 Evolution of System Verilog**

**SystemVerilog Components**

SystemVerilog language is a combination of concepts of multiple languages.

SystemVerilog language components are,

* Concepts of Verilog HDL
* Testbench constructs based on Vera
* OpenVera assertions
* Synopsys’ VCS DirectC simulation interface to C and C++
* A coverage application programming interface that provides links to coverage metrics



**Fig 3.8 SystemVerilog Components**

**CHAPTER 4**

**VERIFICATION**

Verification is the most important part of the VLSI design flow. It aims to find out the bugs in the RTL (Register Transfer Level) design at an early stage so that it does not prove out destructive at the later stage in the design process. Around 70% of the time is consumed in the verification process. So, it is the most time-consuming process. Due to the increase in number of transistors in the integrated circuit (IC), reducing feature size and improved design tools, the complexity of the IC has increased. This raises the probability of occurrence of bugs in the design. Hence, the need for the verification of the IC became necessary.

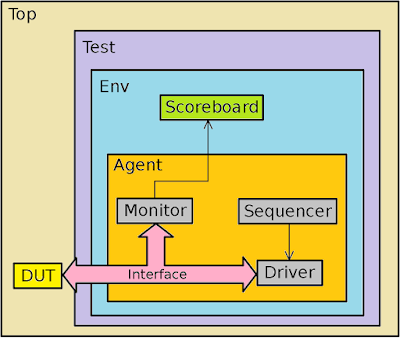


**Fig 4.1 Position of RTL Verification in the VLSI Design Flow**

**4.1 UVM**

Universal Verification Methodology (UVM) is a standard verification methodology used to verify the RTL (Register Transfer Level) design. It consists of base class library coded in SystemVerilog[8]. The verification engineer can create different verification components by extending these classes. Moreover, UVM provides many other useful verification features such as use of macros for implementing complex function, factory for object creation [8].

Figure 8 shows the various UVM verification components created to verify APB design.

****

**Fig 4.2 UVM Architecture**

**4.1.1 Sequence item**

The transactions are extended from the uvm\_sequence\_item. This component randomizes the address and data. The field automation macros are applied to the data members of this class.

**4.1.2 Sequences**

A sequence is a series of transaction. In the sequence class, the users can create complex stimulus. These sequences can be randomized, extended to create another sequence and can be combined.

**4.1.3 Sequencer**

UVM sequencer coordinates between the driver and sequence. It passes the transaction to the driver for execution and obtains the response from the driver. It also acts as an arbitrator for multiple sequences running in parallel.

**4.1.4 Driver**

Driver initiates the request for the next transaction and drives it to the lower-level components. It is created by extending the uvm\_driver.

**4.1.5 Collector and Monitor**

The collector extracts the signal information from the bus and converts it into the transactions and passes it through the analysis port to the monitor for further comparing.

**4.1.6 Agent**

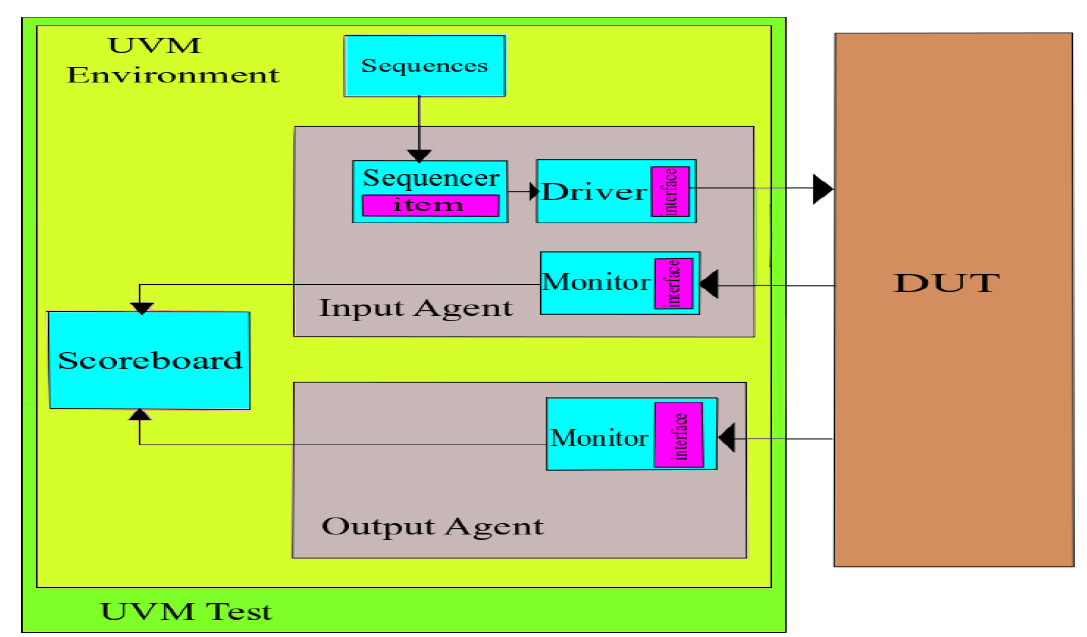
The agent instantiates the verification components driver, monitor, collector and sequencer. It also connects these components using TLM connections. The agent can have one of the operating modes active or passive. In the active mode of operation, the agent instantiates driver, sequencer collector and monitor whereas in the passive mode of operation only monitor and collector are instantiated and configured.

**4.1.7 Environment**

The Environment class instantiates all the sub components such as agents, driver, monitor etc. and configures them.

**4.1.8 Test**

The uvm\_test is extended from the uvm\_component. Different testcases can be generated for the given verification environment.



**Fig 4.3 UVM Verification components**

* a user-defined agent is extended from uvm\_agent, uvm\_agent is inherited by uvm\_component
* An agent typically contains a driver, a sequencer, and a monitor
* Agents can be configured either active or passive

## **Active agent**

* Active agents generate stimulus and drive to DUT
* An active agent shall consist of all the three components driver, sequencer, and monitor

## **Passive agent**

* Passive agents’ sample DUT signals but do not drive them  
  A passive agent consists of only the monitor

**4.2 UVM PHASES**

UVM Phases are a synchronizing mechanism for the environment **Phases** are represented by callback **methods**, A set of predefined phases and corresponding callbacks are provided in uvm\_component. The Method can be either a function or task.

Any class deriving from uvm\_component may implement any or all of these callbacks, which are executed in a particular order

The UVM Phases are,

1. build
2. connect
3. end of elaboration
4. start of simulation
5. run
6. extract
7. check
8. report

**Phases Description**

|  |  |  |
| --- | --- | --- |
| **Phase** | **Description** | **Execution Order** |
| build | Used to construct the testbenchcomponents | top-down |
| connect | Used to connect TLM ports of components | bottom-up |
| end\_of\_elaboration | Used to make any final adjustments to the structure, configuration or connectivity of the testbench before simulation starts | bottom-up |
| start\_of\_simulation | used for printing testbench topology or configuration information | bottom-up |
| run | Used for stimulus generation, driving, monitoring, and checking | parallel |
| extract | Used to retrieve and process information from scoreboards and functional coverage monitors |  |
| check | Used to check that the DUT behaved correctly and to identify any errors that may have occurred during the execution of the test bench |  |
| report | Used to display the results of the simulation or to write the results to file |  |
| final | Used to complete any other outstanding actions that the test bench has not already completed |  |

**4.3 UVM CODING**

1) Factory Registration: Object and Component

* Factory Registration for the object
  + Factory Registration for the object w/o field ->Sequence
  + Factory Registration for the object with field ->Sequence\_item
* Factory Registration for the Component
* Factory Registration for the Component w/o field

2) Constructor: Object and Component

* Constructor for the object (have 1 field(argument) i.e Name)
* Constructor for the Component (have 2 field(argument) i.e Name and Parent)

Object: Object is one which we create and we can destroy whenever we want it will not be their till end of the simulation

Component: Component is one which we create and we can't destroy whenever we want it will be their till end of the simulation

**4.3.1 Factory Registration for Component:**

Test:

class apb\_test extends uvm\_test

`uvm\_component\_utils(apb\_test)

Env:

class apb\_env extends uvm\_env

`uvm\_component\_utils(apb\_env)

Scoreboard:

class apb\_scoreboard extends uvm\_scoreboard

`uvm\_component\_utils(apb\_scoreboard)

Agent:

class apb\_agent extends uvm\_agent

`uvm\_component\_utils(apb\_agent)

Sequencer:

class apb\_sequencer extends uvm\_sequencer

`uvm\_component\_utils(apb\_sequencer)

Driver:

class apb\_driver extends uvm\_driver

`uvm\_component\_utils(apb\_driver)

Monitor:

class apb\_monitor extends uvm\_monitor

`uvm\_component\_utils(apb\_monitor)

**4.3.2 Factory Registration for Object:**

Sequence\_item:

class apb\_sequence\_item extends uvm\_sequence\_item

`uvm\_object\_utils(apb\_sequence\_item)

**4.3.3 Constructor for Component:**

**HIDDEN Constructor**

**function new();**

1)create a blank memory

2)Extract the variable of the class

3)Assigned the defualt value

4)Assign the handle to the memory

**endfunction**

Test,Env,Scoreboard,Agent,Sequencer,Driver,Monitor:

function new(string name,uvm\_component parent="Null")

super.new(name,parent)

endfunction

**4.3.4 Constructor for Object:**

Sequence\_item,Seqeuence:

function new(string name)

super.new(name)

endfunction

**4.3.5 Build\_Phase:**

Test:

//instance

mem\_env env\_h;

mem\_sequence sequence\_h;

//Build\_Phase

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

env = mem\_env::type\_id::create("env", this);

seq = mem\_sequence::type\_id::create("seq");

endfunction : build\_phase

Env:

mem\_agent agent\_h;

mem\_scoreboard scoreboard\_h;

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

agent\_h = mem\_agent::type\_id::create("agent\_h", this);

scoreboard\_h = mem\_scoreboard::type\_id::create("scoreboard\_h",this);

endfunction : build\_phase

Agent:

mem\_sequencer sequencer\_h;

mem\_driver driver\_h;

mem\_monitor monitor\_h;

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

seqeuncer\_h = mem\_sequencer::type\_id::create("sequencer\_h", this);

driver\_h = mem\_driver::type\_id::create("driver\_h",this);

monitor\_h = mem\_monitor::type\_id::create("monitor\_h",this);

endfunction : build\_phase

----------------------------------

Build Phase :

test (env,seq)

env (sb,agent)

agent (sqr,drv,mon)

before build we need to institate.

test.sv

//instatiate

mem\_env env\_h;

mem\_sequence sequence\_h;

//build

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

env = mem\_model\_env::type\_id::create("env", this);

sequence\_h = mem\_sequence::type\_id::create("sequence\_h");

endfunction : build\_phase

env.sv (agnt,scb)

//instatiate

mem\_scb scb\_h;

mem\_agent agent\_h;

//build

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

scb\_h = mem\_scb::type\_id::create("scb\_h", this);

agent\_h = mem\_agent::type\_id::create("agent\_h",this);

endfunction : build\_phase

agent.sv

//instatiate

mem\_sqr sqr\_h;

mem\_drv drv\_h;

mem\_mon mon\_h;

//build

virtual function void build\_phase(uvm\_phase phase);

super.build\_phase(phase);

sqr\_h = mem\_sqr::type\_id::create("sqr\_h", this);

drv\_h = mem\_drv::type\_id::create("drv\_h",this);

mon\_h = mem\_mon::type\_id::create("mon\_h",this);

endfunction : build\_phase

**4.3.6 connect.sv**

Sequencer-driver uvm rules followed (connect)

driver-interface no uvm rules followed coz of interface (no connect)

interface-dut no uvm rules followed coz of interface and dut (no connect)

dut-interface no uvm rules followed coz of interface and dut (no connect)

interface-Monitor no uvm rules folloed coz of interface (no connect)

monitor-scb uvm rules followed (connect)

sequncer-driver connect in agent.sv

monitor-scb connect in env.sv

**CHAPTER 5**

**IMPLEMENTATION**

**5.1 SOFTWARE TOOLS USED**

1. GVIM (Text Editor)

2. Questa Sim (System Verilog Compilation, Simulation and Wavefom Visualizer)

**5.2 APB DESIGN CODE**

module APB\_Protocol(

input PCLK,PRESETn,transfer,READ\_WRITE,

input [8:0] apb\_write\_paddr,

input [7:0]apb\_write\_data,

input [8:0] apb\_read\_paddr,

output PSLVERR,

output [7:0] apb\_read\_data\_out

);

wire [7:0]PWDATA,PRDATA,PRDATA1,PRDATA2;

wire [8:0]PADDR;

wire PREADY,PREADY1,PREADY2,PENABLE,PSEL1,PSEL2,PWRITE;

// assign PREADY = READ\_WRITE ? (apb\_read\_paddr[8] ? PREADY2 : PREADY1) : (apb\_write\_paddr[8] ? PREADY2 : PREADY1);

assign PREADY = PADDR[8] ? PREADY2 : PREADY1 ;

assign PRDATA = READ\_WRITE ? (PADDR[8] ? PRDATA2 : PRDATA1) : 8'dx ;

// assign PRDATA = READ\_WRITE ? (apb\_read\_paddr[8] ? PRDATA2 : PRDATA1) : 16'dx;

master\_bridge dut\_mas(

apb\_write\_paddr,

apb\_read\_paddr,

apb\_write\_data,

PRDATA,

PRESETn,

PCLK,

READ\_WRITE,

transfer,

PREADY,

PSEL1,

PSEL2,

PENABLE,

PADDR,

PWRITE,

PWDATA,

apb\_read\_data\_out,

PSLVERR

);

slave1 dut1( PCLK,PRESETn, PSEL1,PENABLE,PWRITE, PADDR[7:0],PWDATA, PRDATA1, PREADY1 );

slave2 dut2( PCLK,PRESETn, PSEL2,PENABLE,PWRITE, PADDR[7:0],PWDATA, PRDATA2, PREADY2 );

endmodule

**5.3 MASTER CODE**

module master\_bridge(

input [8:0]apb\_write\_paddr,apb\_read\_paddr,

input [7:0] apb\_write\_data,PRDATA,

input PRESETn,PCLK,READ\_WRITE,transfer,PREADY,

output PSEL1,PSEL2,

output reg PENABLE,

output reg [8:0]PADDR,

output reg PWRITE,

output reg [7:0]PWDATA,apb\_read\_data\_out,

output PSLVERR );

// integer i,count;

reg [2:0] state, next\_state;

reg invalid\_setup\_error,

setup\_error,

invalid\_read\_paddr,

invalid\_write\_paddr,

invalid\_write\_data ;

localparam IDLE = 3'b001, SETUP = 3'b010, ENABLE = 3'b100 ;

always @(posedge PCLK)

begin

if(!PRESETn)

state <= IDLE;

else

state <= next\_state;

end

always @(state,transfer,PREADY)

begin

if(!PRESETn)

next\_state = IDLE;

else

begin

PWRITE = ~READ\_WRITE;

case(state)

IDLE: begin

PENABLE =0;

if(!transfer)

next\_state = IDLE ;

else

next\_state = SETUP;

end

SETUP: begin

PENABLE =0;

if(READ\_WRITE)

// @(posedge PCLK)

begin PADDR = apb\_read\_paddr; end

else

begin

//@(posedge PCLK)

PADDR = apb\_write\_paddr;

PWDATA = apb\_write\_data; end

if(transfer && !PSLVERR)

next\_state = ENABLE;

else

next\_state = IDLE;

end

ENABLE:

begin if(PSEL1 || PSEL2)

PENABLE =1;

if(transfer & !PSLVERR)

begin

if(PREADY)

begin

if(!READ\_WRITE)

begin

next\_state = SETUP; end

else

begin

next\_state = SETUP;

apb\_read\_data\_out = PRDATA;

end

end

else next\_state = ENABLE;

end

else next\_state = IDLE;

end

/\* if(transfer && !PREADY && READ\_WRITE)

begin

// repeat(3) @(posedge PCLK)

// begin

// if(!transfer)

// next\_state = IDLE;

//else

// begin

//if(PREADY) begin

// next\_state = SETUP;

//apb\_read\_data\_out = PRDATA; end

//else

next\_state = ENABLE;

//end

end

end

else if(transfer && PREADY && READ\_WRITE )

begin

apb\_read\_data\_out = PRDATA;

next\_state = SETUP;

end

else if(transfer && !PREADY && !READ\_WRITE)

begin

// repeat(3) @(posedge PCLK)

// begin

// if(!transfer)

// next\_state = IDLE;

// else begin

// if(PREADY)

// next\_state = SETUP;

//else

next\_state = ENABLE;

end

end

end

else if(transfer && PREADY && !READ\_WRITE )

begin

next\_state = SETUP;

$strobe($time," Enable write operation"); end \*/

default: next\_state = IDLE;

endcase

end

end

/\* always @(posedge PCLK)

begin

PWRITE = ~READ\_WRITE;

case(state)

IDLE: begin

PENABLE = 0;

end

SETUP: begin

PENABLE =0;

if(READ\_WRITE)

// @(posedge PCLK)

PADDR = apb\_read\_paddr;

else

//@(posedge PCLK)

PADDR = apb\_write\_paddr;

PWDATA = apb\_write\_data;

end

ENABLE: begin

PENABLE =1;

end

endcase

end \*/

assign {PSEL1,PSEL2} = ((state != IDLE) ? (PADDR[8] ? {1'b0,1'b1} : {1'b1,1'b0}) : 2'd0);

// PSLVERR LOGIC

always @(\*)

begin

if(!PRESETn)

begin

setup\_error =0;

invalid\_read\_paddr = 0;

invalid\_write\_paddr = 0;

invalid\_write\_paddr =0 ;

end

else

begin

begin

if(state == IDLE && next\_state == ENABLE)

setup\_error = 1;

else setup\_error = 0;

end

begin

if((apb\_write\_data===8'dx) && (!READ\_WRITE) && (state==SETUP || state==ENABLE))

invalid\_write\_data =1;

else invalid\_write\_data = 0;

end

begin

if((apb\_read\_paddr===9'dx) && READ\_WRITE && (state==SETUP || state==ENABLE))

invalid\_read\_paddr = 1;

else invalid\_read\_paddr = 0;

end

begin

if((apb\_write\_paddr===9'dx) && (!READ\_WRITE) && (state==SETUP || state==ENABLE))

invalid\_write\_paddr =1;

else invalid\_write\_paddr =0;

end

begin

if(state == SETUP)

begin

if(PWRITE)

begin

if(PADDR==apb\_write\_paddr && PWDATA==apb\_write\_data)

setup\_error=1'b0;

else

setup\_error=1'b1;

end

else

begin

if (PADDR==apb\_read\_paddr)

setup\_error=1'b0;

else

setup\_error=1'b1;

end

end

else setup\_error=1'b0;

end

end

invalid\_setup\_error = setup\_error || invalid\_read\_paddr || invalid\_write\_data || invalid\_write\_paddr ;

end

assign PSLVERR = invalid\_setup\_error ;

endmodule

**5.4 SLAVE-1 CODE**

module slave1(

input PCLK,PRESETn,

input PSEL,PENABLE,PWRITE,

input [7:0]PADDR,PWDATA,

output [7:0]PRDATA1,

output reg PREADY );

reg [7:0]reg\_addr;

reg [7:0] mem [0:63];

assign PRDATA1 = mem[reg\_addr];

always @(\*)

begin

if(!PRESETn)

PREADY = 0;

else

if(PSEL && !PENABLE && !PWRITE)

begin PREADY = 0; end

else if(PSEL && PENABLE && !PWRITE)

begin PREADY = 1;

reg\_addr = PADDR;

end

else if(PSEL && !PENABLE && PWRITE)

begin PREADY = 0; end

else if(PSEL && PENABLE && PWRITE)

begin PREADY = 1;

mem[PADDR] = PWDATA; end

else PREADY = 0;

end

endmodule

**5.5 SLAVE-2 CODE**

module slave2(

input PCLK,PRESETn,

input PSEL,PENABLE,PWRITE,

input [7:0]PADDR,PWDATA,

output [7:0]PRDATA2,

output reg PREADY );

reg [7:0]reg\_addr;

reg [7:0] mem2 [0:63];

assign PRDATA2 = mem2[reg\_addr];

always @(\*)

begin

if(!PRESETn)

PREADY = 0;

else

if(PSEL && !PENABLE && !PWRITE)

begin PREADY = 0; end

else if(PSEL && PENABLE && !PWRITE)

begin PREADY = 1;

reg\_addr = PADDR;

end

else if(PSEL && !PENABLE && PWRITE)

begin PREADY = 0; end

else if(PSEL && PENABLE && PWRITE)

begin PREADY = 1;

mem2[PADDR] = PWDATA; end

else PREADY = 0;

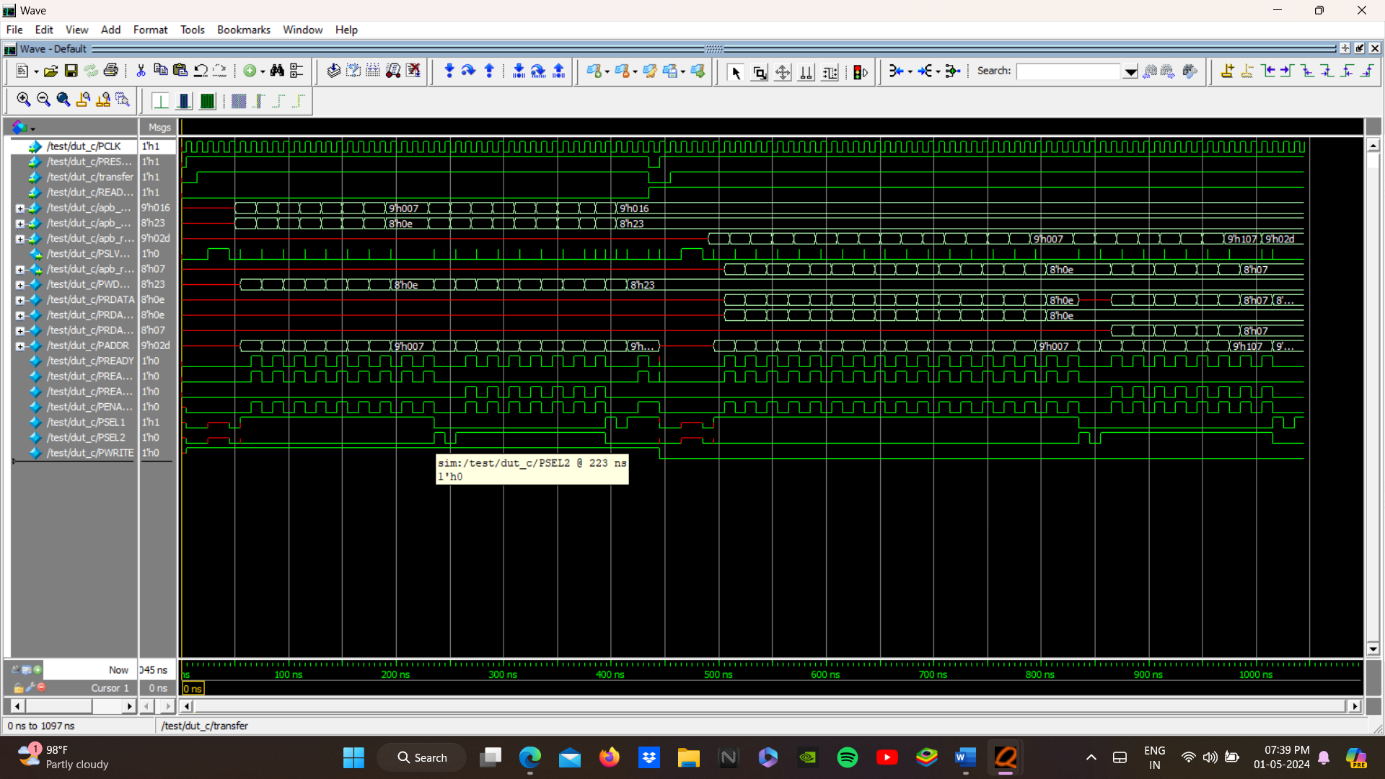
end

endmodule

**Chapter 6**

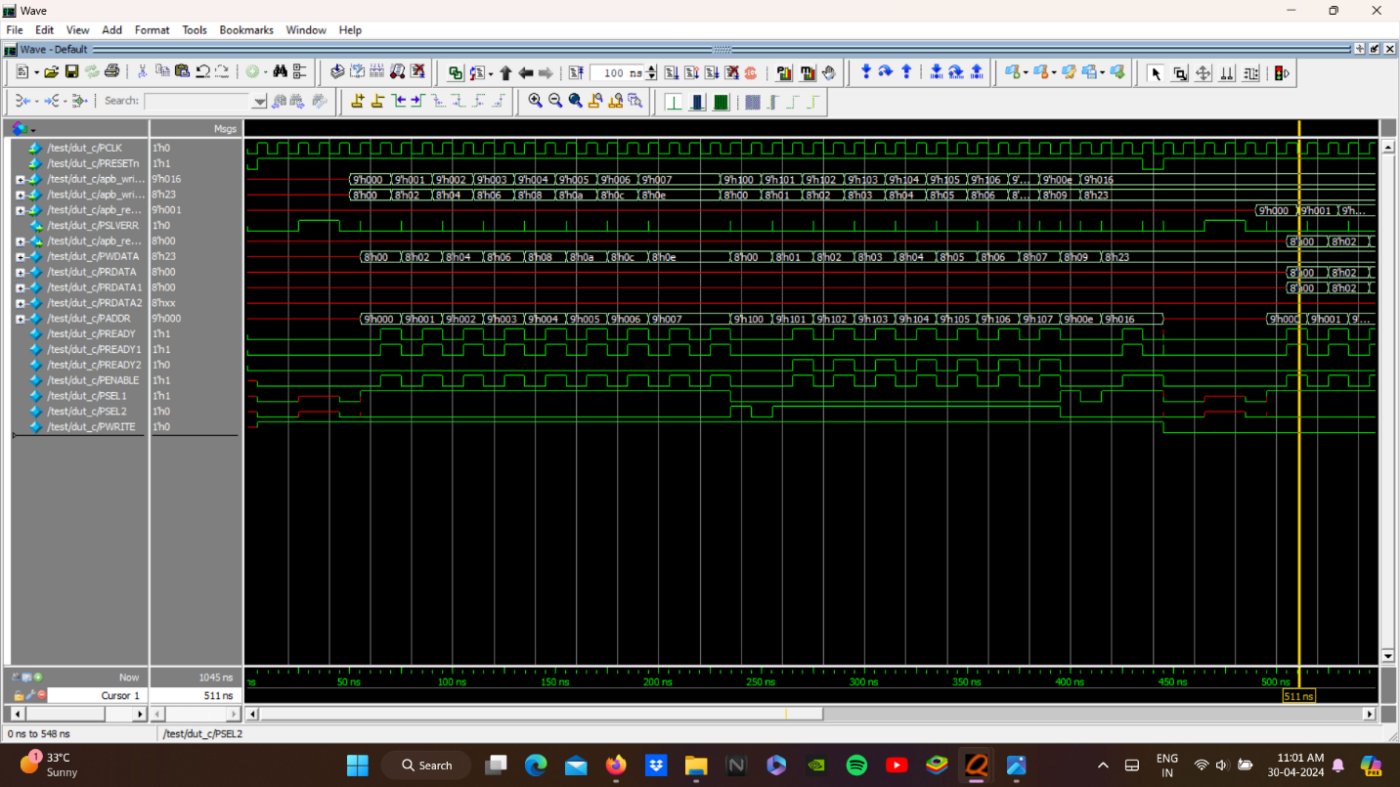
**RESULTS**

**Simulation Results**

****

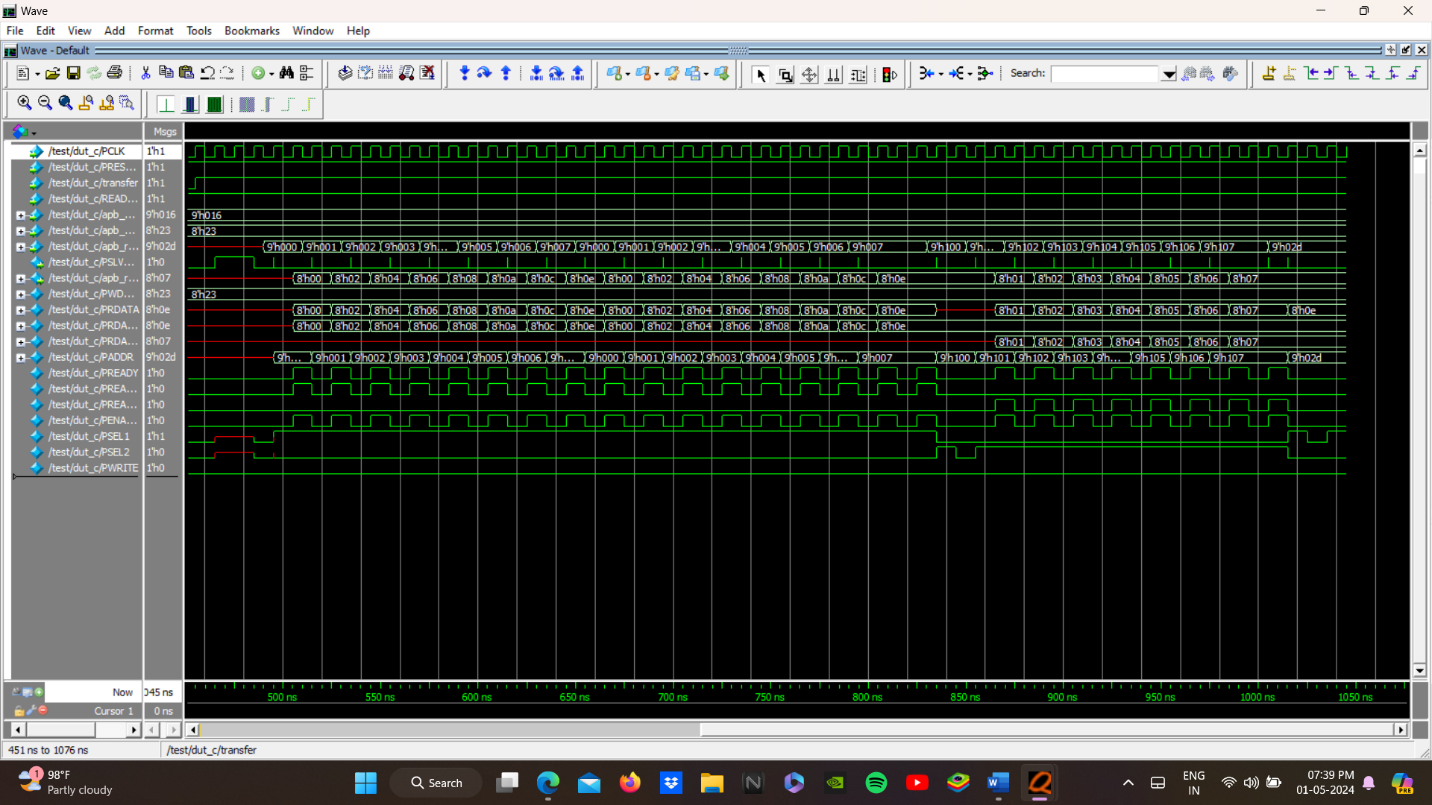
**Fig 6.1**

**6.1 Write operation**

****

**Fig 6.2**

**6.2 Read Operation**

****

**Fig 6.3**

**CONCLUSION**

The development of the synthesizable APB Bridge in system verilog HDL was done. The PENABLE mechanism was implemented for making it the low-power consuming system. The functional verification of the bridge was done by driving various testcases to the design for testing the features. The multimaster and multislave AHB to APB bridge is one of the future scopes.

**REFERENCES**

[1] N.Venkateswara Rao, PV Chandrika, Abhishek Kumar and Sowmya Reddy,“ Design of AMBA based AHB2APB protocol for efficient utilization of AHB and APB”, International Research Journal of Engineering and Technology (IRJET), Volume 07, Issue 03, pp. 23950072, 2020.

[2] Vaishnavi R.K, Bindu.S and Sheik Chandbasha, “Design and Verificat ion of APB Protocol by using System Verilog and Universal Verificat ion Methodology”, Internat ional Research Journal of Engineering and Technology (IRJET), Volume 06, Issue 06, pp. 23950072, 2019.

[3] Kommiriset t i Bheema Raju and Bala Krishna Konda, “Design and Verificat ion of AMBA APB Protocol”, Int. Journal of Engineering Research and Applicat ion, Volume 7, Issue 1, pp.87-90, 2017.

[4] M. Kiran Kumar, Amrita Sajja and Dr. Fazal Noorbasha, “Design and FPGA Implementation of AMBA APB Bridge with Clock Skew Minimization Technique”, IOSR Journal of VLSI and Signal Processing (IOSR-JVSP), Volume 7, Issue 3, pp. 42-45, 2017.

[5] Kiran Rawat, Kanika Sahni and Sujata Pandey, “RTL Implementat ion for AMBA ASB APB Protocol at System on Chip level”, IEEE International Conference on Signal Processing and Integrated Networks (SPIN), 2015.

[6] Roopa.M, Vani.R.M and P.V.Hunagund, “Design of Low Bandwidth Peripherals Using High Performance Bus Architecture”, ELSEVIER, International Conference Communication Technology and System Design, 2011.

[7] Chenghai Ma, Zhijun Liu and Xiaoyue Ma, “Design and Implementation of APB Bridge based on AMBA 4.0”, IEEE, International Conference on Consumer Electronics, Communications and Networks (CECNet), 2011.

[8] Muhammad Hafeez and Azilah Saparon, “IP Core of Serial Peripheral Interface (SPI) with AMBA APB Interface”, IEEE 9th Symposium on Computer Applications & Industrial Electronics (ISCAIE), 2019.

[9] Manu B.N and Prabhavathi P, “Design and Implementation of AMBA ASB APB Bridge”, IEEE International Conference on Fuzzy Theory and Its Application, pp. 2377-5823, 2013.

[10] Ashutosh Gupta, Kiran Rawat, Sujata Pandey, Pradeep Kumar, Saket Kumar and H.P.Singh, “Physical Design Implementation of 32-bit AMBA ASB APB module with improved performance”, IEEE International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), 2016.

[11] Zdravko Panjkov, et.al., “OCP2XI Bridge: An OCP to AXI Protocol Bridge”, Springer International Publishing Switzerland, pp. 179–190, 2014.

[12] Shankar, Dipti Girdhar and Neeraj Kr. Shukla, “Design and Verification of AMBA APB Protocol”, International Journal of Computer Applications, Volume 95, No.21, June 2014.

[13] Karin Avnit, Arcot Sowmya and Jorgen Peddersen, “ACS: Automat ic Converter Synthesis for SoC Bus Protocols”, Springer-Verlag Berlin Heidelberg, pp. 343–348, 2010.

[14] Wang Hang Suan, Asral Bahari Jambek, Mohd Nazrin Bin Md Isa, Azizi bin Harun, Shaiful Nizam Bin Mohya and Zulfiqar Ali Bin Abd. Aziz, “Design and implementation of AMBA bridge protocol in System-on- Chip design”, Indonesian Journal of Electrical Engineering and Computer Science, Volume 14, No. 2, pp. 788-795, May 2019.

[15] Shivani Malhotra and Neelam Rup Prakash, “UVM-Based Verification IP of AMBA AXI Protocol Showing Multiple Transact ions and Data Interconnect”, Springer Nature Singapore Pte Ltd, 2019.