RTL Implementation of SoC Bridge: AMBA AHB-APB bridge

CND111 – Final Project

By

Abeer Tarek | v23009912

Maria George | v23010119

Al-Moemenbellah Allam | v23009973

Supervisor/s

Dr. Khaled Salah
Eng. Muhammed El-Shafeay

Contents

I. Introduction

Arm's Acorn RISC Machine, developed by Arm Holdings, is a family of reduced instruction set computing (RISC) architectures for processors, licensed to companies for diverse product designs. RISC processors, like those based on Arm's architecture, require fewer transistors, improving cost, power, and heat efficiency. This makes them suitable for various devices, from portable electronics to servers and supercomputers.

The advanced microcontroller bus architecture, AMBA, is an open standard for on-chip communication. It’s widely used in high-preformance embedded systems as it defines protocols, interfaces, and connectivity standards for efficiently communicating processors, memories, and various peripherals on an IC.

The AMBA has two system busses; the high-perfomance bus (AHB), which defines interfaces between high-performance components as memory controllers, processors, and a DSP. The second bus system is the advanced perophcral bus (APB) which is designed for low-power, low-complexity, and low-cost interfaces. It mainly connects the low-bandwidth and low-power peripherals.

AMBA also includes a bridge linking the AHB and APB buses. Bridges are standard interfaces between buses, enabling communication between IPs connected to different buses in a standardized manner. In the following sections, the AHB, and APB will be discussed in more details.

1- AHB

The AMBA AHB bus defines an interface for high-performance components including masters, interconnects, and slaves. The most important features implemented by the AHB include the burst transfer of data, the single clock-edge operations, the non-tristate implementation, and wide data bus configurations (from 64 to 1024 bits), those are the features required by high clock frequency and high-performance devices.

To bridge between AHB and APB, an AHB slave is used. External memory interfaces and internal memory devices as well as high-performance peripherals are the most common AHB slaves. The AHB master and slaves connect together using address decoder to select an appropriate slave, and a slave-to-master multiplexer to conncet the output of the slave back to the master. The AHB master is responsible for initiating the read and write operations by providing the suitable adresses and control information. Figure 1 shows the AHB block diagram , while figures 2, and 3 show the slave and master diagrams respectively. The slave respond to the transfers initiated by the master, and itsignals back the ouput (completion or extension of bus transfer) as well as the success or failure of the bus back to the master.

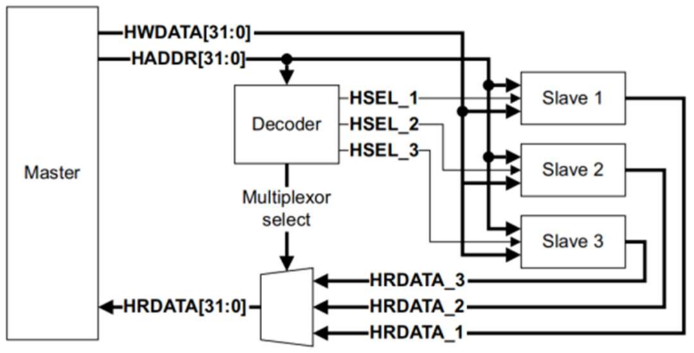


Figure 1. AHB block diagram.

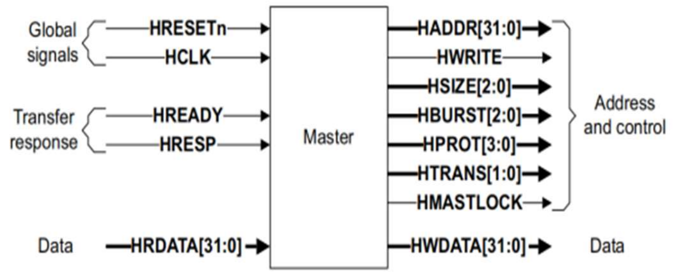


Figure 2. AHB Master block.

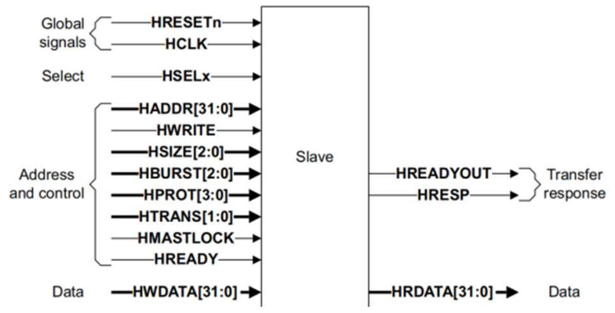


Figure 3. AHB Slave block.

2- APB

As mentioned above, the APB interface is optimized for low-power peripherals. This protocol is not pipelined, and hence can be used to connect low-bandwidth devices that donot require the AXI (Advanced Extensible Interface) protocol's high performance. The duration of APB varies, it takes three HCLK cycles for a read, and two cycles for a write in the EASY. Moreover, the APB acesses are limited to one word in width, therefore writing an 8-bit portion of a 32-bit APB register is not feasible. APB peripherals do not need a PCLK input as the APB access is timed with an enable signal generated by the AHB to APB bridge interface. This makes APB peripherals low power consumption parts, because they are only strobed when accessed.

3- Timing diagrams

A- AHB

A basic transfer in AHB has two phases: one for address, and the other for data. The address phase by default lasts for only the **HCLK** cycle if not extended. The data phase can have several cycles controlled by the **HREADY** signal. The direction of transfer is controlled by the **HWRITE** signal as started before. Figures 6, and 7 show simple read and write transfers with no wait states. In this basic transfer, the address and control signals drive onto the bus at the **HCLK** rising edge. They are sampled on the next edge, and the **HREADYout** response is sampled at the third rising edge.

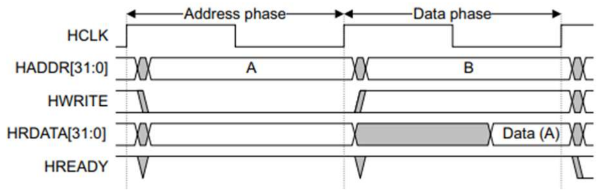


Figure 4. Read Transfer with no waits.

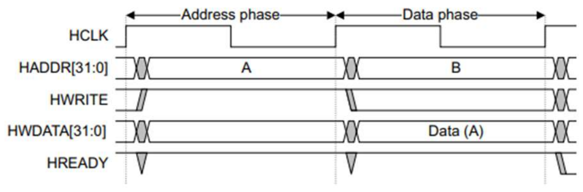


Figure 5. Write Transfer with no waits.

Wait states can be inserted to any transfer to extend the completion of the transfer. Each Subordinate has an **HREADYOUT** signal that it drives during the data phase of a transfer. The interconnect combines the **HREADYOUT** signals from all Subordinates to generate a single **HREADY** signal that is used to control the overall progress. Following is the timing of the read and write transfers with wait states inserted.

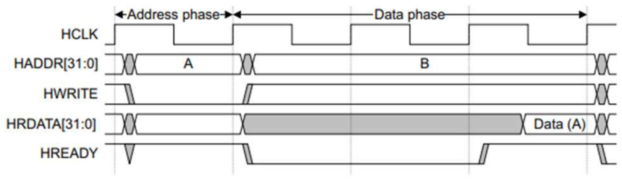


Figure 6. Read transfer with waits.

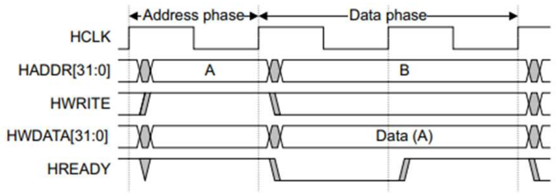


Figure 7. Write transfer with waits.

B- APB

Write Transfer

All signals are driven by the **PCLK**. The write transfer has two phases: setup and access phases. In the setup phase, the select signal is activated, so the **PADDR, PWRITE,** and **PWDATA** are valid. During the access phase, any control signals must remain stable, the enable and ready signals are asserted indicating the readiness of accepting the data. At the end of the transfer, the enable and select signals should be deactivated unless there's another transfer to the same peripheral. The following figures show the timing diagrams of the write transfers with and without wait states.

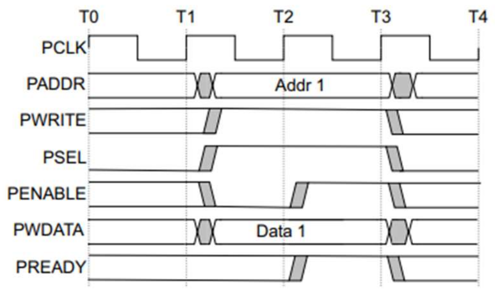


Figure 8. Write Transfer with no waits.

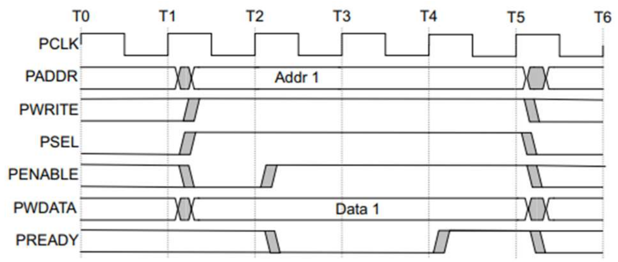


Figure 9. Write Transfer with waits.

Read Transfer

All signals are driven by the **PCLK** rising edge. To extend the transfer, the **PREADY** signal is driven to low. The following figures show the read transfer with and without wait states.

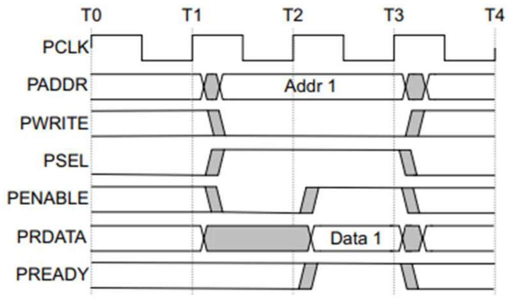


Figure 10. Read Transfer with no waits.

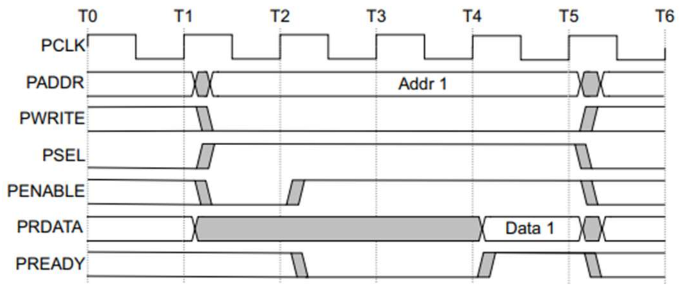


Figure 11. Read Transfer with waits.

4- AHB-APB Bridge

As mentioned earlier, the bridge is an AHB slave connecting the high speed peripherals with the low-power devices. The APB receives the read and write transfers from the AHB bridge. As can be seen from the timing diagrams, the AHB has a pipelined structure where the data and address of the transaction are sent on different clock cycles, while for the APB, bothe the data and the address have to be driven at both cycles. The Bridge will therefore have to facilitate communication between these two protocols by making the AHB waituntil both the address and the data are available for the APB. When the AHB is required to wait for the APB, wait states are added from thebridge by controlling the **HREADYout** signal.

To ensure there's no data loss during data transfers between the high bandwidth processor's and other peripherals on the APB, the interface between the AHB and APB should first buffer the adress, controls, and data from the AHB before driving the APB peripherals and returning data back along with response signals to the APB. The AHB-APB interface works with any combination of clock frequency and phase. The data transfers from AHB to APB during the write cycle, and from the APB to AHB during the read cycle.

II. Architecture Design

Figure 4 shows the main architecture of the AMBA bridge. The bridge have 3 important modules; the AHB slave bus interface, the APB output generation, and the APB transfer state machine.

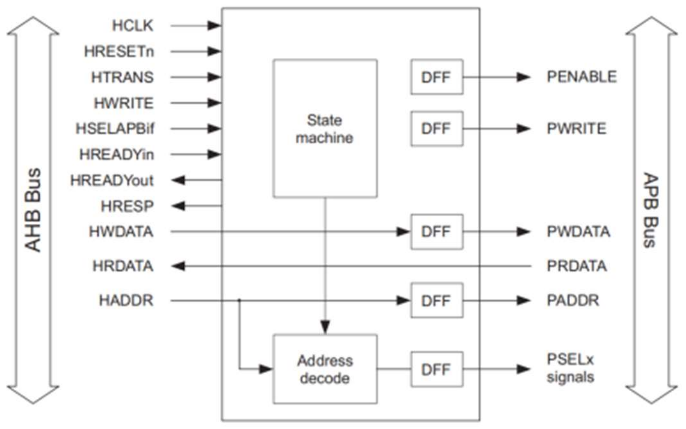


Figure 12. AHB-APB interface block diagram.

Table 1 shows the description of the important system signals.

Table 1. System signals description.

Signal	Direction	Description
HCLK	Input	Bus clock (times all bus transfers)
HRESETn	Input	Global reset for the bus and system
HADDR[31:0]	Input	System adress bus
HTRANS[1:0]	Input	Type of transfer (sequential, non-sequential, idle, busy)
HWRITE	Input	Transfer direction (1 for write, 0 for read)
HWDATA[31:0]	Input	Write data bus (transfers data from the master to the bus slaves during the write operation)
HRDATA[31:0]	Output	Read data bus (bus slaves to the master)
HREADYin/HREADYout	Inout	Flag indicating the transfer is done or not. If low, the transfer is extended.
HRESP[1:0]	Output	Transfer response/status (usually okay)
PRDATA[31:0]	Input	Peripheral read data bus
PWDATA[31:0]	Output	Peripheral write data bus
PENABLE	Output	Peripheral enable
PSELx	Output	Peripheral slave select

5- AHB Slave

This module incorporates the transfer detection logic, address and control registers for proper data transfer. Both the current and registered adsresses are required by this module to manage the transfer properly since the read and write operations occur at different timings. The current and previous addresses are multiplexed.

6- State Machine

Figure 5 shows the APB transfer machine that drive the APB functions based on the AHB inputs.

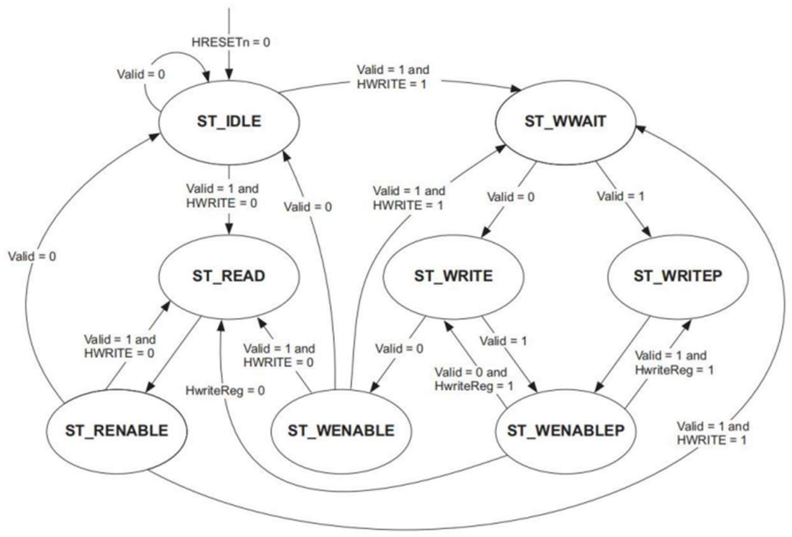


Figure 13. APB state machine.

- ST_IDLE: no operations are being performed.
- ST_READ: access phase of a reading transaction.
- ST_RENABLE: Enable phase of a reading transaction.
- ST_WWAIT: A wait state is inserted to allow the bridge to obtain both the address and the data of the write transaction.
- ST_WRITE: the access phase of a write transaction with more writes pending.
- ST:WENABLE: the Enable phase of a write transaction, with no more writes pending.
- ST_WRITEP: the access phase of a write transaction with more writes pending.
- ST:WENABLEP: the Enable phase of a write transaction, with more writes pending.

7- AHB Output Generation

The following table summarizes the activation of each output at the AHB bus.

Output	Activation
HRDATA	Driven by PRDATA directly
HREADYout	Driven by a registered signal to improve the output timings when wait states are generated.
HRESP	Always low as long as no split, error, or retry signals are generated by the APB.

8- APB Output Generation

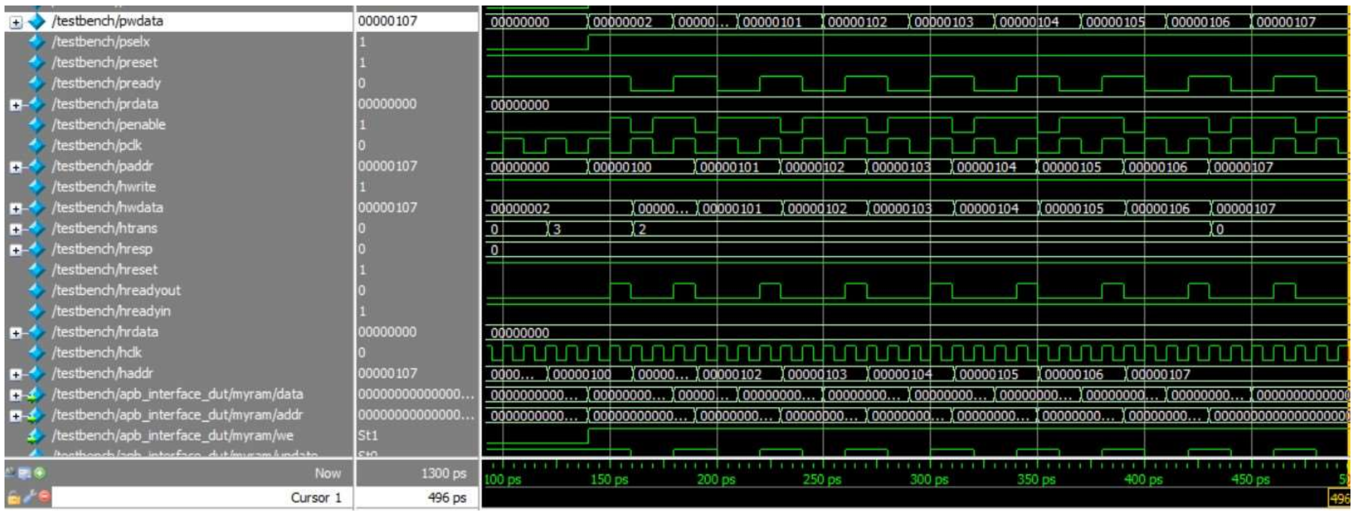
The APB outputs are generated from the state machine. The following table summarizes the activation of each output at the APB bus.

Output	Activation
PWDATA	Enabled only during a write transfer
PENABLE	HIGH only during one of three enable states, in the last cycle of an APB transfer.
PSELx	These outputs are decoded from the current transfer address. Valid only during the read, write and enable states, and LOW otherwise
PADDR	A registered version of the currently selected address input (HADDR or the address register). It only changes when the read and write states are entered at the start of the APB transfer.
PWRITE	HIGH during a write transfer, and only changes when a new APB transfer is started.

III. Verification:

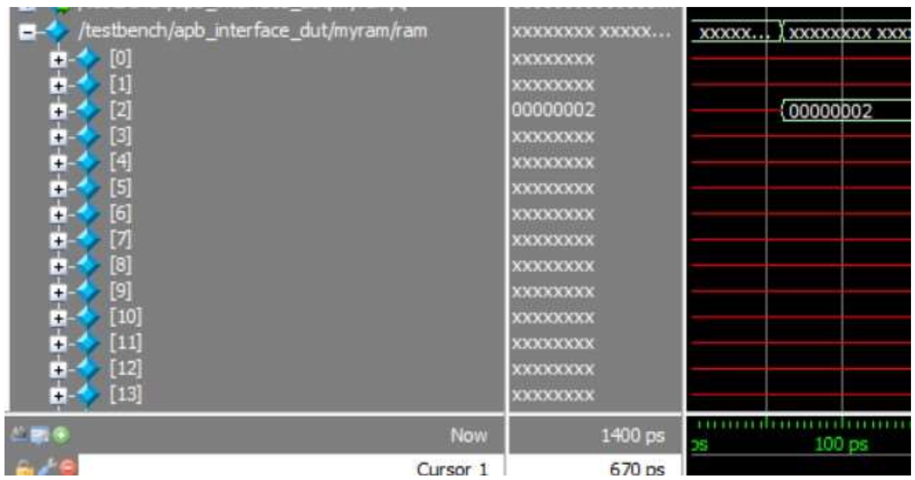
To test the behavior of the bridge, we performed 4 operations, a single write at address 0x002, 8 burst writes from addresses 0x100 to 0x107, a single read at address 0x002, and 8 burst reads from addresses 0x100 to 0x107. The written data during the write operation is the same as the memory address to ensure that the correct data went to the correct address.

The figure below shows the pwdata signal writes data into the APB:

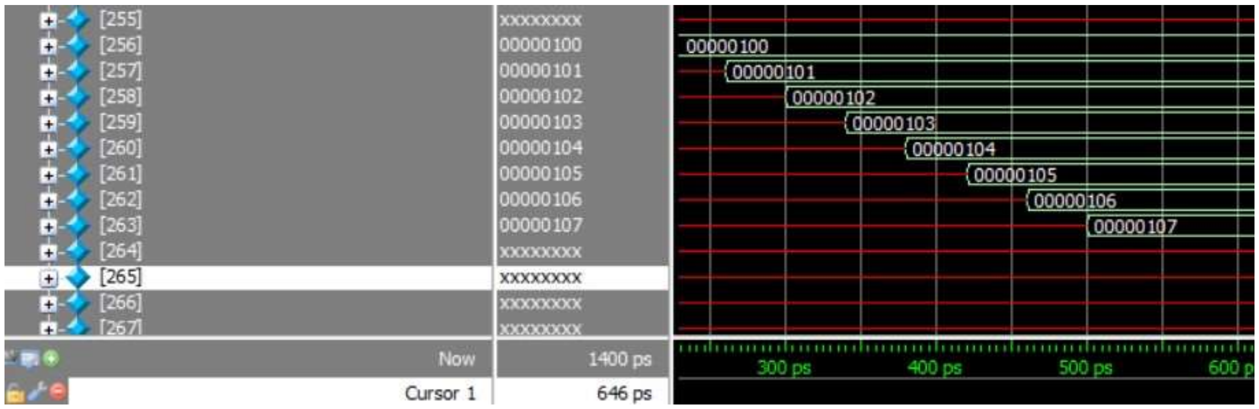


```
/testbench/pwdata                              00000107
/testbench/pselx                               1
/testbench/preset                              1
/testbench/pready                              0
/testbench/prdata                              00000000
/testbench/penable                             1
/testbench/pclk                                0
/testbench/paddr                               00000107
/testbench/hwrite                              1
/testbench/hwdata                              00000107
/testbench/htrans                              0
/testbench/hresp                               0
/testbench/hreset                              1
/testbench/hreadyout                           0
/testbench/hreadyin                            1
/testbench/hrdata                              00000000
/testbench/hclk                                0
/testbench/haddr                               00000107
/testbench/apb_interface_dut/myram/data        00000000000000...
/testbench/apb_interface_dut/myram/addr        00000000000000...
/testbench/apb_interface_dut/myram/we          St1
Now                                            1300 ps
Cursor 1                                       496 ps
```

We can also see below hoe the data was written into the APB device storage memory:



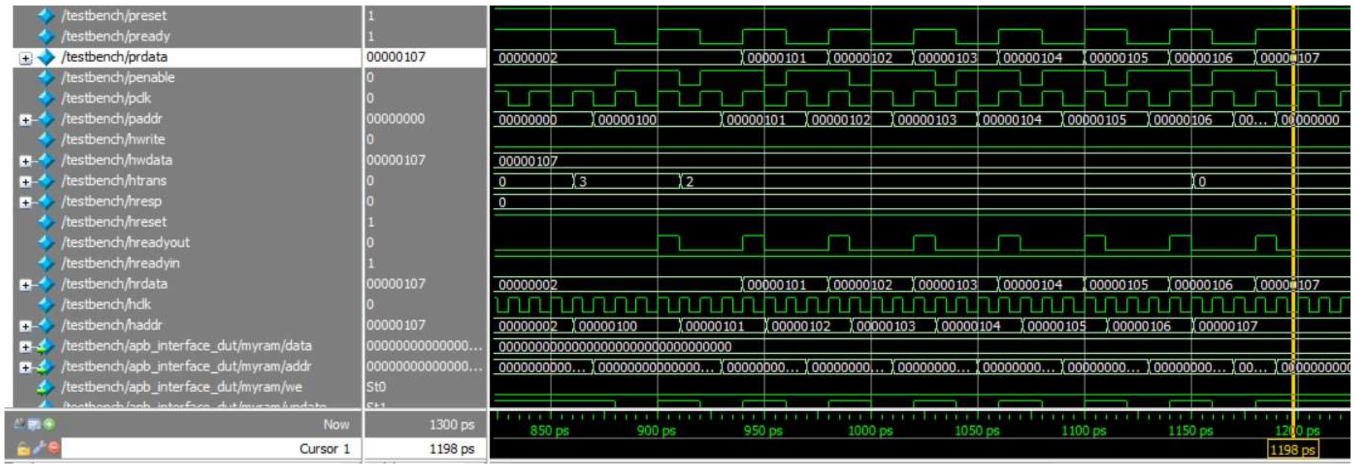
```
/testbench/apb_interface_dut/myram/ram    xxxxxxxx xxxxx...
  [0]                                     xxxxxxxx
  [1]                                     xxxxxxxx
  [2]                                     00000002
  [3]                                     xxxxxxxx
  [4]                                     xxxxxxxx
  [5]                                     xxxxxxxx
  [6]                                     xxxxxxxx
  [7]                                     xxxxxxxx
  [8]                                     xxxxxxxx
  [9]                                     xxxxxxxx
  [10]                                    xxxxxxxx
  [11]                                    xxxxxxxx
  [12]                                    xxxxxxxx
  [13]                                    xxxxxxxx
Now                                       1400 ps
Cursor 1                                  670 ps
```



```
  [255]                                   xxxxxxxx
  [256]                                   00000100
  [257]                                   00000101
  [258]                                   00000102
  [259]                                   00000103
  [260]                                   00000104
  [261]                                   00000105
  [262]                                   00000106
  [263]                                   00000107
  [264]                                   xxxxxxxx
  [265]                                   xxxxxxxx
  [266]                                   xxxxxxxx
  [267]                                   xxxxxxxx
Now                                       1400 ps
Cursor 1                                  646 ps
```

We can see how the correct data went to its correct address, note that 0x100 = 0d256.

The figure below shows how the data is retrieved on the prdata signal during the read operations:



IV. RTL Implementation

1- Bridge:

The brige consists of the state machine module and and AHB slave module to interface with the AHB bus:

```
module bridge_top(input [31:0] HADDR,
    input [31:0] HWDATA,
    input HWRITE,
    input HCLK,
    input HRESET,
    input [1:0] HTRANS,
    input HREADY,
    input [31:0] prdata,
    output [31:0] paddr,
    output  pwrite,
    output [31:0] pwdata,
    output penable,
    output pselx,
    output hready_out,
    output [1:0] hresp,
    output [31:0] hrdata,
    input pready
    );

wire VALID;
wire HWRITE_REG;
wire TEMP_SELX;

//PIPELINING REGISTERERS
wire [31:0] HADDR_1,HADDR_2,HWDATA_1,HWDATA_2;
//INSTATNTIATING AHB_SLAVE
```

```
AHB_SLAVE
ahb_slave(HADDR,HWDATA,HWRITE,HCLK,HRESET,HTRANS,HREADY,VALID,HADDR_1,HADDR_2,HWDATA_1,HWDATA_2,HWRIT
E_REG,TEMP_SELX);
//INSTATNTIATING FSM CONTROLLER
apb_fsm
fsm_controller(VALID,HADDR_1,HADDR_2,HWDATA_1,HWDATA_2,HWRITE_REG,TEMP_SELX,HCLK,HRESET,HWRITE,prdata
,paddr,pwrite,
  pwdata,penable,pselx,hready_out,hresp,hrdata,pready);

endmodule
```

a. State Machine

```
module apb_fsm(
    input valid,
    input [31:0] haddr_1,
    input [31:0] haddr_2,
    input [31:0] hwdata_1,
    input [31:0] hwdata_2,
    input hwrite_reg,
    input temp_selx,
    input hclk,
    input hreset,
    input hwrite,
    input [31:0] prdata,
    output reg [31:0] paddr,
    output reg pwrite,
    output reg [31:0] pwdata,
    output reg penable,
    output reg pselx,
    output reg hready_out,
    output [1:0] hresp,
    output [31:0] hrdata,
    input pready
    );

//STATES OF FSM
parameter ST_IDLE    =3'b000,
          ST_WWAIT   =3'b001,
         ST_READ    =3'b010,
         ST_WRITE   =3'b011,
            ST_WRITEP  =3'b100,
            ST_RENABLE =3'b101,
         ST_WENABLE =3'b110,
            ST_WENABLEP=3'b111;

//PRESENT STATE AND NEXT STATE REGISTERS
reg [2:0] present_state,next_state;
reg [31:0] addr;


//PRESENT STATE LOGIC
always@(posedge hclk or negedge hreset)
  begin
    if(~hreset)
      present_state <= ST_IDLE;
    else
```

```
        present_state <= next_state;
    end


  //NEXT STATE LOGIC
  always@(*)
    begin:ns_block
      next_state <= ST_IDLE;//default state
      case(present_state)
         ST_IDLE:
           begin
             if(valid == 1 && hwrite == 0)
                      next_state <= ST_READ;
                    else if(valid && hwrite)
                      next_state <= ST_WWAIT;
                    else
                      next_state <= ST_IDLE;
               end

         ST_WWAIT:
           begin
             if(valid)
                     next_state <= ST_WRITEP;
                else
                     next_state <= ST_WRITE;
              end

        ST_READ: next_state <= ST_RENABLE;

        ST_WRITE:
          begin
            if(valid)
                     next_state <= ST_WENABLEP;
               else
                     next_state <= ST_WENABLE;
             end

        ST_WRITEP: next_state <= ST_WENABLEP;

        ST_RENABLE:
          begin
      if(~pready)
      next_state<=ST_RENABLE;
      else begin
             if(valid == 0)
                      next_state <= ST_IDLE;
                if(valid == 1 && hwrite == 0)
                      next_state <= ST_READ;
                else if(valid && hwrite)
                      next_state <= ST_WWAIT;
           end
      end

       ST_WENABLE:
         begin

      if(~pready)
      next_state<=ST_WENABLE;
      else begin
```

```
              if(valid == 0)
                   next_state <= ST_IDLE;
                 if(valid == 1 && hwrite == 0)
                   next_state <= ST_READ;
                 else if(valid && hwrite)
                   next_state <= ST_WWAIT;
            end
         end
         ST_WENABLEP:
           begin
         if(~pready)
         next_state<=ST_WENABLEP;
         else begin
             if(hwrite_reg ==0)
                   next_state <= ST_READ;
                else if(hwrite_reg == 1 && valid == 0)
                  next_state <= ST_WRITE;
                else if(hwrite_reg == 1 && valid == 1)
                  next_state <= ST_WRITEP;
                 end
         end
       endcase
     end


     //SIGNAL VALUES, OUTPUT LOGIC (COMBINATIONAL)
     always@(*)
       begin
         paddr= 0;
         pwdata= 0;
         pwrite= 0;
         penable = 0;
         pselx = 0;
         hready_out = 0;
         case(present_state)
           ST_IDLE : hready_out= 0;
           ST_WWAIT : hready_out = 0;
           ST_READ :
             begin
               paddr= haddr_1;
                  pselx= temp_selx;
                  hready_out = 0;
             end
           ST_RENABLE :
             begin
               penable = 1;
                  hready_out = (next_state!=ST_RENABLE);
                  paddr = haddr_2;
                  pselx = temp_selx;
             end

           ST_WRITE :
             begin
               paddr = haddr_1;
                  hready_out = 0;
                  pselx= temp_selx;
                  pwdata= hwdata_1;
                  pwrite= 1;
              end
```

```
                ST_WENABLE :
                  begin
                     paddr = haddr_1;
                        hready_out = (next_state!=ST_WENABLE);
                        pselx= temp_selx;
                        pwdata= hwdata_1;
                        pwrite= 1;
                        penable = 1;
                   end

               ST_WRITEP :
                 begin
                   paddr= haddr_2;
                      addr = paddr;
                      pselx= temp_selx;
                      pwdata = hwdata_1;
                      pwrite = 1;
                 end

              ST_WENABLEP :
                begin
                  paddr= addr;
                 hready_out = (next_state!=ST_WENABLEP);
                     pselx= temp_selx;
                     pwdata = hwdata_2;
                     pwrite= 1;
                     penable = 1;
                end

            endcase
          end


          assign hrdata = prdata;
          assign hresp  = 0;

          endmodule
```

b. AHB Slave interface

```
module AHB_SLAVE(
        input [31:0] HADDR,
        input [31:0] HWDATA,
        input HWRITE,
        input HCLK,
        input HRESET,
        input [1:0] HTRANS,
        input HREADY,
        output reg VALID,
        output reg [31:0] HADDR_1,HADDR_2,HWDATA_1,HWDATA_2,
        output reg HWRITE_REG,
        output TEMP_SELX
        );
```

```
//DATA TRANSITION TYPE
parameter IDLE = 2'b00, BUSY = 2'b01, NON_SEQ = 2'b10, SEQ = 2'b11;

//PIPELINING ADDRESS, DATA AND HWRITE
always@(posedge HCLK or negedge HRESET)
  begin:pipeline_block
      if(~HRESET)//Asynchronous Negative Reset
        begin : reset_block

        HADDR_1 <= 0;
        HADDR_2 <= 0;
        HWDATA_1 <= 0;
        HWDATA_2 <= 0;
        HWRITE_REG <= 0;
      end

      else
        begin

       HADDR_1 <= HADDR;
       HADDR_2 <= HADDR_1;
       HWDATA_1 <= HWDATA;
       HWDATA_2 <= HWDATA_1;
       HWRITE_REG <= HWRITE;
       end
   end


//which slave to select?, since we only haveone slave, we will only have one wire which is
always chosen.
//for more slaves, add more wires and set only the chosen slave as 1.
assign  TEMP_SELX = 1'b1;


//VALID SIGNAL LOGIC
always@(*)
  begin : valid_logic
    VALID = 1'b0;
    if(HRESET)
        if((HTRANS != IDLE&& HTRANS != BUSY))
            VALID = 1'b1;
           else
           VALID = 1'b0;
  end
endmodule
```

For testing, some systemverilog modules were added to interface with the bridge

2- AHB Master

```
// AHB MASTER
module ahb_master(input hclk,hreset,hreadyout,
                    input [31:0] hrdata,input [1:0] hresp,
                    output reg [31:0] haddr,hwdata,
                    output reg hwrite,hreadyin,
                    output reg [1:0] htrans);


always@(negedge hreset)begin
   hwdata = 0;
        hwrite = 0;
   hreadyin = 0;
   htrans = 0;

end


//HTRANS
`define IDLE 2'b00
`define BUSY 2'b01
`define SEQ 2'b10
`define NONSEQ 2'b11

  integer i;



//defining routines to be used by the testbench.

//SINGLE WRITE
  task single_write;

      @(posedge hclk);
      #1;
      haddr = 32'h0000_0002;
      hwrite = 1;
      hreadyin = 1;
      htrans = 2'b10;
      @(posedge hclk);
      #1;
      hwdata =  32'h0000_0002;
      htrans = 2'b00;

  endtask




//8 BURST WRITES
 task burst_write();
       @(posedge hclk);
        #1;
        hreadyin = 1;
        hwrite = 1;
        haddr = 32'h0000_0100;
```

```
                htrans = `NONSEQ;

        for(i=0;i<7;i++)
                      begin
                        wait(hreadyout);

                        @(posedge hclk);
                        #1;

                        hwdata = haddr;
                        haddr = haddr + 1'b1;
                        htrans = `SEQ;
                      end
                    wait(hreadyout);
                    @(posedge hclk);
                    #1;
                    hwdata = haddr;
                    htrans = `IDLE;


endtask






//SINGLE READ
  task single_read;

      @(posedge hclk);
      #1;
      haddr = 32'h0000_0002;
      hwrite = 0;
      hreadyin = 1;
      htrans = 2'b10;
      @(posedge hclk);
      #1;
      htrans = 2'd00;

  endtask



//8 burst reads
task burst_read();
       @(posedge hclk);
      #1;
      hreadyin = 1;
      hwrite = 0;
      haddr = 32'h0000_0100;
      htrans = `NONSEQ;
```

```
   for(i=0;i<7;i++)
                begin
                  wait(hreadyout);
                  @(posedge hclk);
                  #1;
                  haddr = haddr + 1'b1;
                  htrans = `SEQ;
                end
               htrans = `IDLE;
endtask



endmodule
```

3- APB_device with storage memory

```
//APB INTERFACE
module apb_interface(pclk,preset, penable,pwrite,
                     paddr,pwdata,
                     pselx,prdata,
                      pready);

               input pclk,preset;
               input penable,pwrite;
                     input [31:0] paddr,pwdata;
                     input pselx;
               output reg pready;
               output reg[31:0] prdata;



//storing whether we have passed the select stage or not
reg  selected = 0;

reg updateram = 0;

//storage memory of the peripheral
ram myram(pwdata,paddr,pwrite,updateram,prdata);


always@(posedge pclk,negedge preset)begin

   if(~preset)begin
   pready = 0;
   prdata = 0;

   end else begin

   if(selected==1'b1)
          begin
```

```
              if(penable)
              begin
              selected <= 1'b0;
              updateram<=1'b1;
              pready<=1'b1;
              end

         end

         else  begin
        if(pselx) begin
        selected <= 1'b1;
        updateram<=1'b0;
        pready<=1'b0;
        end

        end
        end
   end


   endmodule



   //storage memory of the peripheral
   module  ram(
      input      [31:0] data,
      input     [31:0] addr,
      input       we,update,
      output reg [31:0] q);


   // the size of the memory is made smaller than the possible addresses for the
   sake of simulation
      reg        [31:0] ram[400];
      always @ (posedge update)
        begin
           if (we)
              ram[addr] <= data;

           q <=   ram[addr];          // output data - output
        end
   endmodule
```

4- Testbench

```
   module testbench();
     reg hclk,hreset,pclk,preset;
     wire hreadyout;
```

```
  wire [31:0] prdata;
  wire [1:0] hresp;
  wire [31:0] hrdata;
  wire [31:0] haddr,hwdata;
  wire hwrite,hreadyin;
  wire [1:0] htrans;

  wire penable,pwrite;
  wire pselx;
  wire [31:0] paddr,pwdata;
  wire pready;

  assign hrdata = prdata;

 //AHB MASTER INSTANTIATION
  ahb_master
ahb_master_dut(hclk,hreset,hreadyout,hrdata,hresp,haddr,hwdata,hwrite,hreadyin,ht
rans);

 //BRIDGE TOP INSTANTIATION
 bridge_top
top_bridge_dut(haddr,hwdata,hwrite,hclk,hreset,htrans,hreadyin,prdata,paddr,pwrit
e,pwdata,penable,pselx,hreadyout,hresp,hrdata,pready);
 //APB INTERFACE INSTANTIATION
  apb_interface
apb_interface_dut(pclk,preset,penable,pwrite,paddr,pwdata,pselx,prdata,pready);

//CLOCK GENERATION
  initial
    begin
      hclk = 1;
      forever #5 hclk = ~ hclk;
  end

  initial
   begin
   pclk = 1;
   forever #10 pclk = ~ pclk; //the pclk period must be an integer multiple of
the hclk period, and must be at least twice the value.
   end

//RESET GENERATION
  initial
    begin
      hreset = 0;
      #4;
      hreset =1;
    end

  initial
    begin
      preset = 0;
      #4;
```

```
        preset =1;
      end




//test routines
  initial
    begin

     ahb_master_dut.single_write();

   #100;

   ahb_master_dut.burst_write();

   #300;


      ahb_master_dut.single_read();

   #100;

      ahb_master_dut.burst_read();

      @(posedge hclk);
      #300 $finish;
    end

endmodule
```

V. Summary

In this project, we have developed an AMBA AHB to APB bridge module which communicates the pipelined AHB communication protocol to the unpipelined APB protocol. The bridge also facilitates wait signals for the AHB due to the lower frequency of the APB module. The bridge can correctly communicate single and burst read/write operations. The figure below shows the generated netlist of the bridge module using Quartus prime pro: