Ministry of Electronics and Information Technology
Government of India

ELECTRONICS INDIA

Digital India Internship Scheme

DIGITAL INDIA INTERNSHIP SCHEME

Broad Area:R&D
Sub Area:R&D in Electronics

INTERNSHIP REPORT

Verilog Design of Advanced Microcontroller Bus Architecture & exploration of front-end design using FPGA

11th June 2018 to 1st August 2018

By:

Aditya Tejaswi

Cochin University of Science and Technology

Acknowledgement

I would like to express my sincere gratitude towards Secretary (MeitY), Shri Arvind Kumar (GC and scientist 'G'), Smt. Sunita Verma (Scientist 'F') & Shri Nishit Gupta (Scientist 'C'), Ministry of Electronics & Information Technology, Govt. of India who gave me the opportunity to work in R & D in Electronics group for the project "Verilog design of Advanced Microcontroller Bus Architecture & exploration of front-end design using FPGA."

Shri Arvind Kumar
GC(R&D in Electronics) & Scientist 'G'

Smt. Sunita Verma
(Scientist 'F')

Shri Nishit Gupta
(Scientist 'C')

Contents

1. Introduction.....2
   1.1 Need for HDLs.....2
   1.2 Verilog.....3
2. Verilog vs VHDL.....4
3. EDA Playground.....5
4. Verilog problem statements and their codes.....6
   4.1 Problem 1 Statement.....6
      4.1.1 Code.....7
      4.1.2 Testbench.....9
      4.1.3 Output.....10
   4.2 Problem 2 Statement.....11
      4.2.1 Code.....12
      4.2.2 Testbench.....13
      4.2.3 Output.....14
   4.3 Problem 3 Statement.....15
      4.3.1 Code.....16
      4.3.2 Testbench.....18
      4.3.3 Output.....19
5. Advanced Microcontroller Bus Architecture.....20
   5.1 AMBA Code.....23
   5.2 Testbench.....39
   5.3 Output.....40
6. FPGAs.....42
7. Nexys4 DDR.....43
   7.1 Xilinx Vivado.....44
8. Snake game implementation using Nexys4 DDR and Vivado.....46
9. References.....50

1.Introduction

Hardware description language (HDL)[1] is a specialized computer language used to program electronic and digital logic circuits. The structure, operation and design of the circuits are programmable using HDL. HDL includes a textual description consisting of operators, expressions, statements, inputs and outputs. Instead of generating a computer executable file, the HDL compilers provide a gate map. The gate map obtained is then downloaded to the programming device to check the operations of the desired circuit. The language helps to describe any digital circuit in the form of structural, behavioral and gate level and it is found to be an excellent programming language for FPGAs and CPLDs.

The three common HDLs are Verilog, VHDL, and SystemC. Of these, SystemC is the newest. The HDLs will allow fast design and better verification. In most of the industries, Verilog and VHDL are common. Verilog, one of the main Hardware Description Language standardized as IEEE 1364 is used for designing all types of circuits. It consists of modules and the language allows Behavioral, Dataflow and Structural Description. VHDL (Very High Speed Integrated Circuit Hardware Description Language) is standardized by IEEE1164. The design is composed of entities consisting of multiple architectures. SystemC is a language that consist a set of C++classes and macros. It allows electronic system level and transaction modeling.

1.1 Need for HDLs

The Moore's Law in the year 1970 has brought a drastic change in the field of IC technology. This change has made the developers to bring out complex digital and electronic circuits. But the problem was the absence of a better programming language allowing hardware and software codesign. Complex digital circuit designs require more time for development, synthesis, simulation and debugging. The arrival of HDLs has helped to solve this problem by allowing each module to be worked by a separate team.

All the goals like power, throughput, latency (delay), test coverage, functionality and area consumption required for a design can be known by using HDL. As a result, the designer can make the necessary engineering tradeoffs and can develop the design in a better and efficient way. Simple syntax, expressions, statements, concurrent and sequential programming is also necessary while describing the electronics circuits. All these features can be obtained by using a hardware description language. Now while comparing HDL and C languages, the major difference is that HDL provides the timing information of a design.

1.2 Verilog[2]

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits.

The advent of hardware verification languages such as OpenVera, and Verisity's e language encouraged the development of Superlog by Co-Design Automation Inc. (acquired by Synopsys). The foundations of Superlog and Vera were donated to Accellera, which later became the IEEE standard P1800-2005: SystemVerilog.

SystemVerilog is a superset of Verilog-2005, with many new features and capabilities to aid design verification and design modeling. As of 2009, the SystemVerilog and Verilog language standards were merged into SystemVerilog 2009 (IEEE Standard 1800-2009). The current version is IEEE standard 1800-2017.

2.Verilog vs VHDL

VHDL [3] stands for VHSIC Hardware Description Language and VHSIC stands for Very High Speed Integrated Circuit. So on the whole VHDL actually stands for Very High Speed Integrated Circuit Hardware Description Language. One of the key features of VHDL is that it is a strongly typed language, which means that each data type (integer, character, or etc.) has been predefined by the language itself. All values or variables defined in this language must be described by one of the data types.

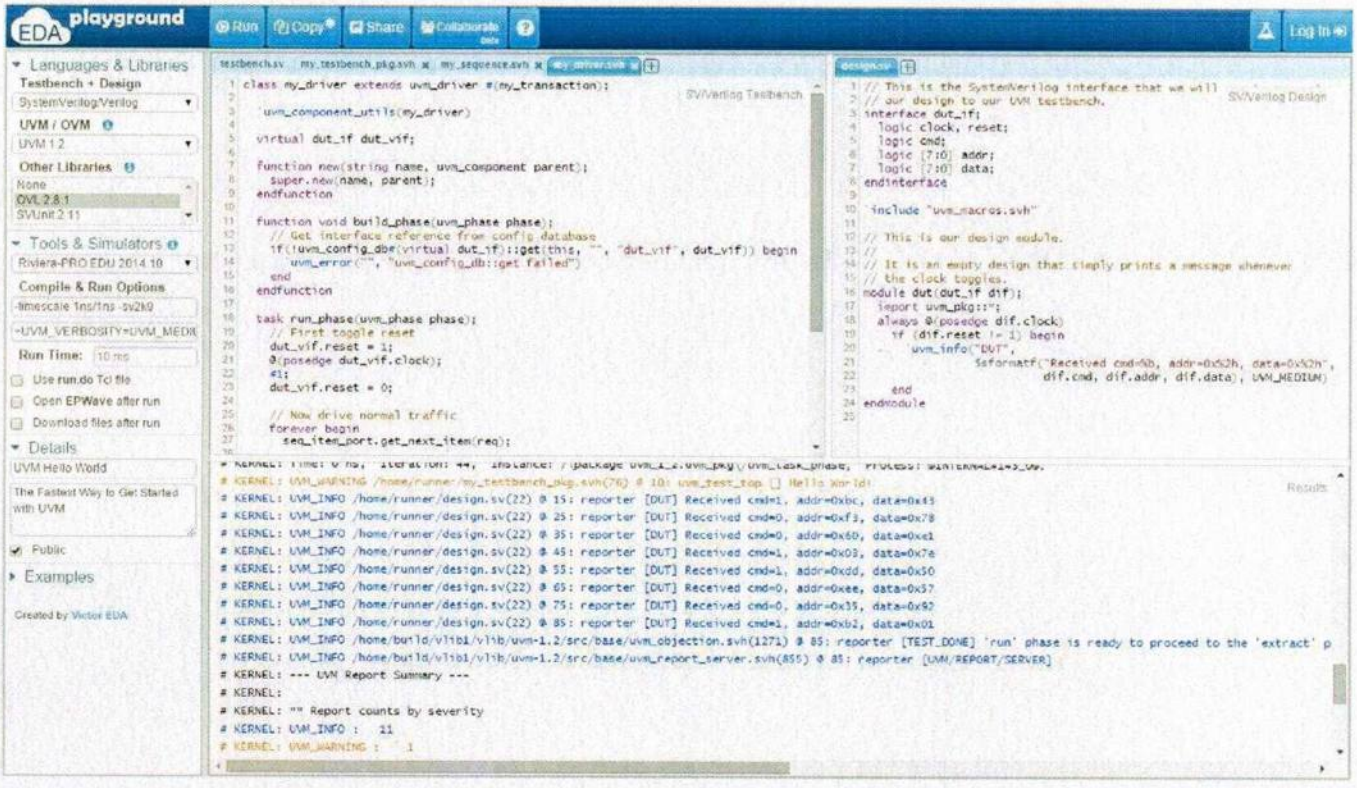
VHDL is more verbose than Verilog and it also has a non-C like syntax. With VHDL, you have a higher chance of writing more lines of code. VHDL can also just seem more natural to use at times. When you're coding a program with VHDL, it can seem to flow better. But maybe that's just my personal opinion.

In Verilog, the language is more compact, as the Verilog language is more of a hardware modeling language. You will end up typing few lines of code and it draws similarities to the C language. Verilog has a better grasp on hardware modeling, but has a lower level of programming constructs. Verilog is not as verbose as VHDL so that's why it's more compact. All in all, Verilog is pretty different from VHDL. There are some similarities, but they are overshadowed by their differences.

3.EDA Playground[4]

EDA Playground gives engineers immediate hands-on exposure to simulating SystemVerilog, Verilog, VHDL, C++/SystemC, and other HDLs. All you need is a web browser. The goal is to accelerate learning of design/testbench development with easier code sharing and simpler access to EDA tools and libraries.

- With a simple click, run your code and see console output in real time.
- View waves for your simulation using EPWave browser-based wave viewer.
- Save your code snippets ("Playgrounds").
- Share your code and simulation results with a web link. Perfect for web forum discussions or emails. Great for asking questions or sharing your knowledge.
- Quickly try something out
    - Try out a language feature with a small example.
    - Try out a library that you're thinking of using.



All the simulations for the problem statements were done on EDA Playground and the output was taken from the EPWave waveforms generated.

4.Verilog Problem Statements and their Codes

4.1 Problem 1.

1. Producer

module producer(input producer_iclk, output logic [3:0] producer_data); endmodule

- Producer will produce the data starting from 1 into the FIFO (using produce_data port)

2. Consumer

module consumer(input consumer_iclk, output bit consumer_req, input logic[3:0] consumer_data); endmodule

- Consumer will issue a request (using consumer_req port) to FIFO to consume data from it and receive data (if available) from FIFO on consumer_data port.

3. FIFO

module fifo(input logic[3:0] producer_data, input bit, consumer_req, output logic[3:0] consumer_data); endmodule

- FIFO is an array of only 10 elements.
- FIFO will receive the data from producer on produce_data port till it gets full and then it calls $finish
- FIFO will receive request from consumer on consumer_req port. FIFO will respond back to Consumer with data using consumer_data port (if data is available). If FIFO is empty, it will return 0 on consumer_data port.

4. TOP

module top(); endmodule

- All modules are instantiated in top module.
- Clock of 5ns time period and 10ns time period is passed to producer & consumer respectively.
- All modules/ sub-module print messages of data generated/ consumed.

Code :

**********Verilog Code**********

```
module producer(input producer_iclk, output producer_data);
 reg [3:0] producer_data;
 int p;

 initial p=0;

 always @(posedge producer_iclk)
  begin
   producer_data = p++;
  end
endmodule

module consumer(consumer_iclk, consumer_req, consumer_data);
 input consumer_iclk;
 input reg[3:0] consumer_data;
 output reg[3:0] consumer_req;

 integer m;
 initial m = 10;

 always @(posedge consumer_iclk)
  begin
    consumer_req = m;
    m--;
  end
endmodule
```

```
module fifo1 (producer_data, consumer_req, consumer_data);
 reg [3:0] fifo [10:1];
 input wire[3:0] producer_data;
 input wire[3:0]consumer_req;
 output reg[3:0] consumer_data;

 integer r;
 initial r = 0;

 always @(producer_data)
  begin
  fifo[r++] = producer_data;
   $display("Data in fifo %h is %h ", r, producer_data);
   if(r>10)
    begin
     $display("No space left in FIFO");
     $finish;
    end
  end

 always @(consumer_req)
  begin
   if(fifo[consumer_req] == 4'b000)
   consumer_data = 0;
   else if(fifo[consumer_req] != 4'b000)
    consumer_data = fifo[consumer_req];
   $display("Consumer data = %h", fifo[consumer_req]);
  end
endmodule
```

**********Testbench**********

```
module top;
 wire [3:0] prod_dat;
 wire [3:0] cons_dat;
 wire [3:0] cons_req;
 reg prod_clk;
 reg cons_clk;

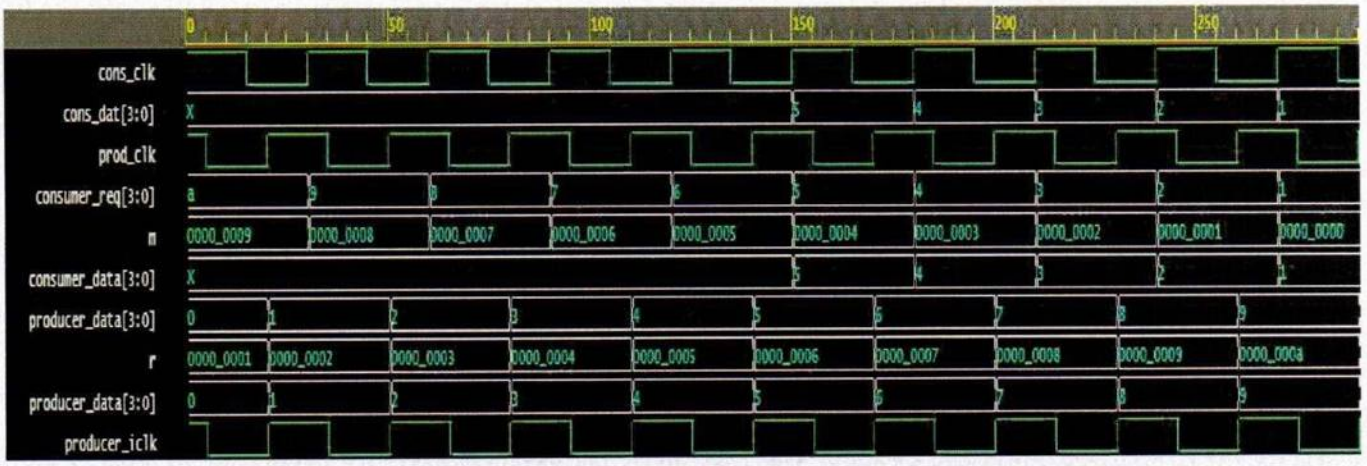
 initial
  begin
   $dumpfile("p1.vcd");
   $dumpvars;
                prod_clk = 1;
        cons_clk = 1;
    #750 $finish;
  end

 always
  begin
  #5 prod_clk = ~prod_clk;
        #10 cons_clk = ~cons_clk;
  end

 producer p1 (.producer_iclk(prod_clk), .producer_data(prod_dat));
 consumer c1 (.consumer_iclk(cons_clk), .consumer_req(cons_req), .consumer_data(cons_dat));
 fifo1 f1 (.producer_data(prod_dat), .consumer_req(cons_req), .consumer_data(cons_dat));

endmodule
```

**********Output**********


0
50
100
150
200
250
cons_clk
cons_dat[3:0]
prod_clk
consumer_req[3:0]
n
consumer_data[3:0]
producer_data[3:0]
r
producer_data[3:0]
producer_iclk
0000_0009
0000_0008
0000_0007
0000_0006
0000_0005
0000_0004
0000_0003
0000_0002
0000_0001
0000_0000
0000_0001
0000_0002
0000_0003
0000_0004
0000_0005
0000_0006
0000_0007
0000_0008
0000_0009
0000_000a

4.2 Problem 2.

1. Interface

interface intf (input master_iclk, input slave_iclk); endinterface

- interface will declare modports for master & slave
- in modport for master, it will import task produce()
- in modport for slave, it will export task produce()

2. master

module master(intf itf); endmodule

- master will produce the data at each clock edge starting from 1 into the FIFO (using the task produce()) and will block in case there is no space in FIFO to produce

3. slave

module slave(intf itf); endmodule

- Slave will consume the date from its FIFO at each clock edge and will block in case there is no data in FIFO to consume
- slave will export a function (through modport declared inside interface) which will be called by master for writing into FIFO of slave

4. TOP

module top(); endmodule

- All modules & interface is instantiated in top module.
- All modules/ sub-module print messages of data generated/ consumed.
- Run simulation for 500ns with:

o Clock of 10ns time period and5ns time period for master & slave respectively.
o Clock of 5ns time period and 10ns time period for master & slave respectively.

Code :

**********Verilog Code**********

```
interface intf(input master_iclk,input slave_iclk);
 modport master(input master_iclk, import task produce(int data));
 modport slave(input slave_iclk, export task produce(int data));
endinterface

module master(intf itf);
  reg [3:0] master_data;
       initial master_data = 0;
   always @(posedge itf.master_iclk)
        begin
          itf.produce(master_data++);
       $display("@time=%t, master_data=%d", $time, master_data);
       end
endmodule

  module slave(intf itf);
  reg [3:0] master_index;
    reg [3:0] slave_index;
  reg [3:0] slave_data;
    reg [3:0] fifo [9:0];
  event slave_event;
    initial master_index =0;
    initial slave_index =0;

  task itf.produce (input int data);
    if (master_index>=10)
```

```
    begin
     ->slave_event;
    end
   else
    fifo [master_index++] = data;
  endtask

   always @ (slave_event)
  begin
   slave_data = fifo[10 - (slave_index++)];
   $display("@time -%t, slave_data=%d", $time, slave_data);
   if(slave_index>=10)
    $display("No more data to consume");
  end
 endmodule
```

**********Testbench**********

```
module top();

 logic master_iclk = 0, slave_iclk = 0;
 //bit consumer_req;
 logic [3:0] slave_data;
 logic [3:0] master_data;

 intf asd(.*);
 master mast(asd.master);
 slave slav(asd.slave);

 initial begin
  $dumpfile("problem2.vcd");
```

```
 $dumpvars;
end

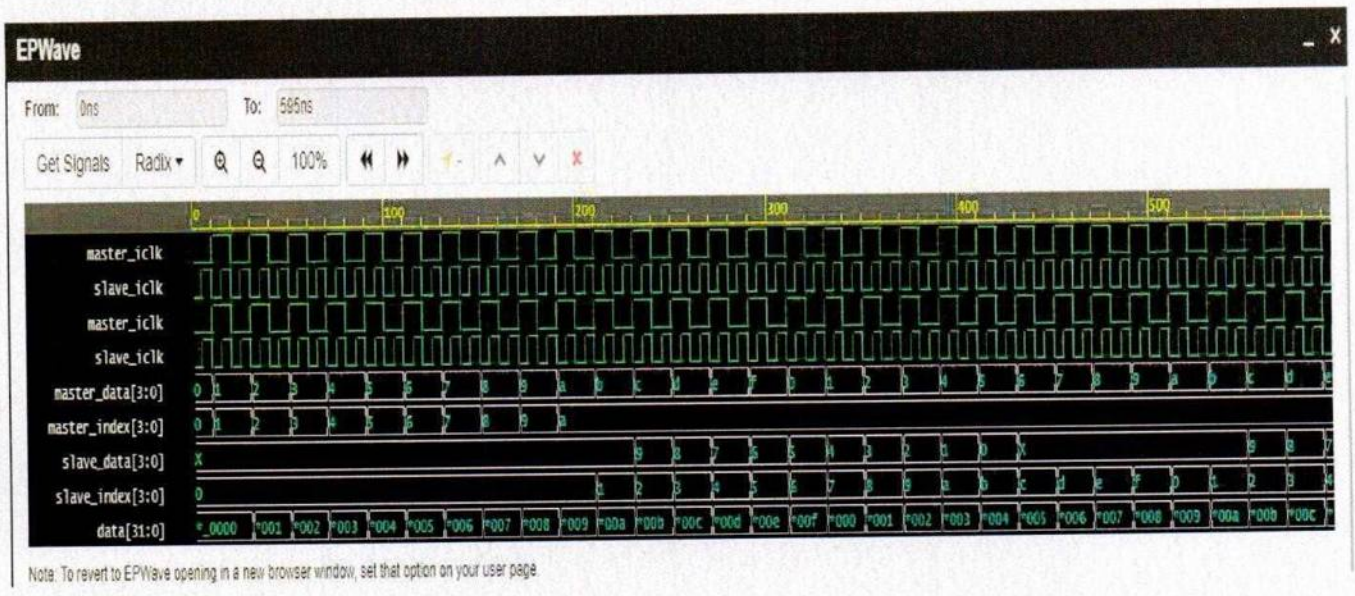
always  begin
    #10 master_iclk = ~ master_iclk;
end

always  begin
    #5 slave_iclk = ~ slave_iclk;
end

always begin
  #500 $finish;
end

endmodule
```

**********Output**********



4.3 Problem 3.

1. Interface

interface intf (input master_iclk, input slave_iclk); endinterface

- interface will declare modports for master & slave
- in modport for master, it will import task produce()
- in modport for slave, it will export task produce()

2. master

module master(intf itf); endmodule

- master will produce non-stop data at each clock edge starting from 1to 15 into the FIFO (using the task produce()) (it should not block in case there is no space in FIFO to produce) and thus it should finish calling 15 produce() calls in 15 clock cycles. Subsequently, it will wait for all 15 responses from slave and then call $finish
- master will export a task named response (through modport declared inside interface) which will be called by slave for providing the status of produce operation.

3. slave

module slave(intf itf); endmodule

- Slave will consume the date from its FIFO at each clock edge and will send the response to producer once the data is successfully put in FIFO. It will block in case there is no data in FIFO to consume.
- slave will export a task (through modport declared inside interface) which will be called by master for writing into FIFO of slave

4. TOP

module top(); endmodule

- All modules & interface is instantiated in top module.
- All modules/ sub-module print messages of data generated/ consumed.

Code :

```
**********************************Verilog Code**********************************
interface intf(input master_iclk, input slave_iclk);
 modport master(input master_iclk, import task produce(int data), export task response(int resp));
          modport slave(input slave_iclk, import task response(int resp), export task produce(int data));

endinterface

module master(intf itf);
 reg [3:0] master_data, slave_data;
  initial master_data = 0;

 task itf.response(input int resp);
  if(resp == 10)
   begin
    $display("Slave data full");
   end
  if(resp == 15)
   begin
    $display("All 15 cycles finished");
    $finish;
   end
 endtask

  always @(posedge itf.master_iclk)
   begin
    itf.produce(master_data++);
```

```
    end
endmodule

module slave(intf itf);
 reg [3:0] master_index,slave_index, slave_data, fifo[9:0];
 event master_event, slave_event;
 int resp; initial resp = 0;
 initial begin master_index=0; slave_index = 0; end

 task itf.produce(input int data);
  if(master_index>=15)
   begin
    @slave_event;
   end
  fifo[master_index++] = data;
  ->master_event;
 endtask

 always @(posedge itf.slave_iclk)
  begin
   if((master_index ==0) || (slave_index >= master_index))
    begin
     @master_event;
    end
   slave_data = fifo[slave_index++];
   resp ++;
   $display("response = %d",resp);
   itf.response(resp);
   ->slave_event;
   $display("@time=%t, slave_data=%d", $time, slave_data);
  end
endmodule
```

**********Testbench**********

```
module top();

 logic master_iclk   0, slave_iclk = 0;
 logic [3:0] slave_data;
 logic [3:0] master_data;

 intf asd(.*);
 master mast(asd.master);
 slave slav(asd.slave);

 initial begin
  $dumpfile("problem2.vcd");
  $dumpvars;
 end

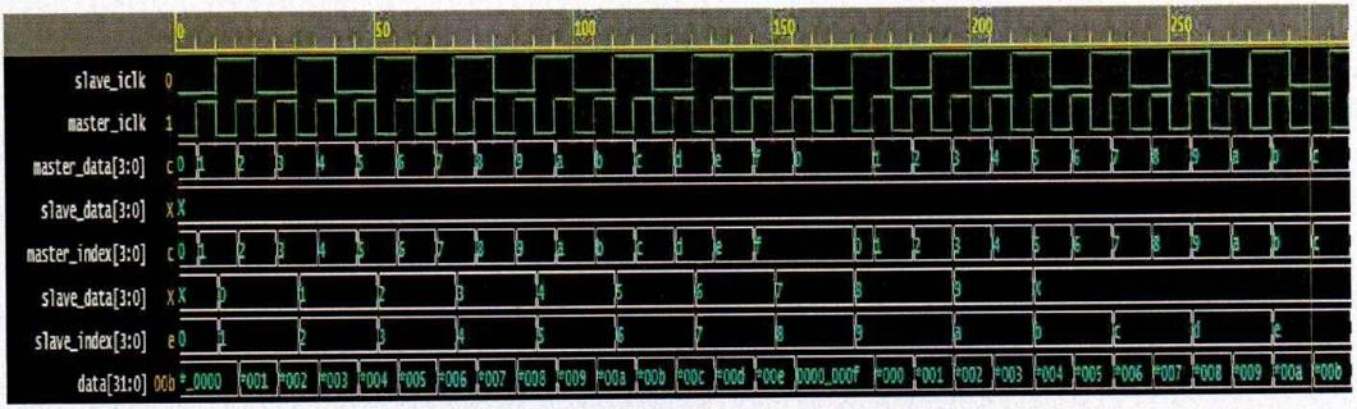

 always  begin
     #5 master_iclk = ~ master_iclk;
 end

 always  begin
     #10 slave_iclk = ~ slave_iclk;
 end

 always begin
  #1000 $finish;
 end
```

endmodule

**********Output**********



5.Advanced Microcontroller Bus Architecture (AMBA)

AHB-Lite Protocol

AMBA AHB-Lite addresses the requirements of high-performance synthesizable designs. It is a bus interface that supports a single bus master and provides high-bandwidth operation. AHB-Lite implements the features required for high-performance, high clock frequency systems including:

- Burst transfers
- Single-clock edge operation
- Non-tristate implementation
- Wide data bus configurations, 64, 128, 256, 512, and 1024 bits.

The most common AHB-Lite slaves are internal memory devices, external memory interfaces, and high bandwidth peripherals. Although low-bandwidth peripherals can be included as AHB-Lite slaves, for system performance reasons they typically reside on the AMBA *Advanced Peripheral Bus* (APB). Bridging between this higher level of bus and APB is done using an AHB-Lite slave, known as an APB bridge.

Figure 1-1 shows a single master AHB-Lite system design with one AHB-Lite master and three AHB-Lite slaves. The bus interconnect logic consists of one address decoder and a slave-to-master multiplexor. The decoder monitors the address from the master so that the appropriate slave is selected and the multiplexor routes the corresponding slave output data back to the master.

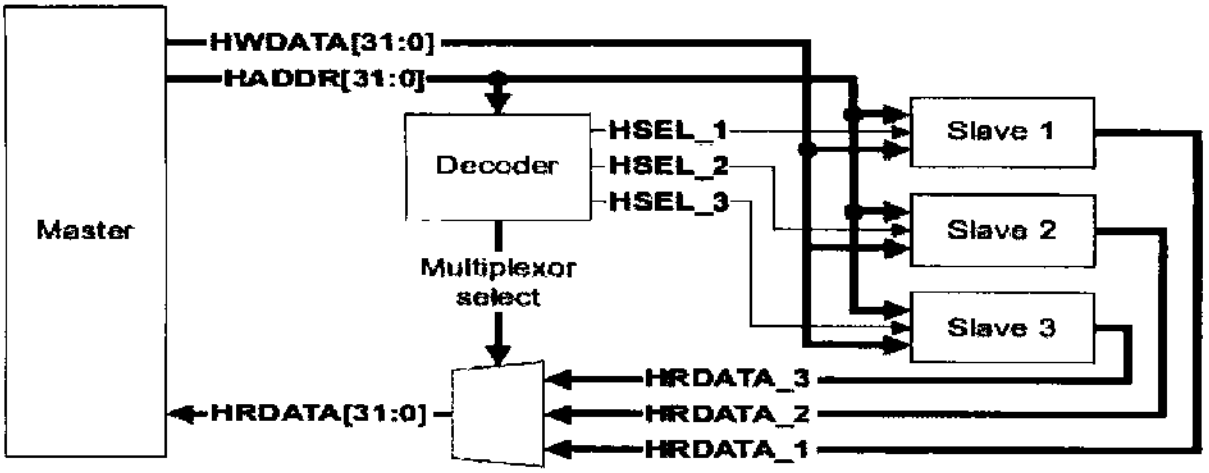


Figure 1-1 AHB-Lite block diagram

The main component types of an AHB-Lite system are described in:

- *Master*
- *Slave*
- *Decoder*
- *Multiplexor*

Master:

An AHB-Lite master provides address and control information to initiate read and write operations.

Slave:

An AHB-Lite slave responds to transfers initiated by masters in the system. The slave uses the **HSELx** select signal from the decoder to control when it responds to a bus transfer. The slave signals back to the master:

- The success
- Failure
- Or waiting of the data transfer.

Decoder:

This component decodes the address of each transfer and provides a select signal for the slave that is involved in the transfer. It also provides a control signal to the multiplexor. A single centralized decoder is required in all AHB-Lite implementations that use two or more slaves.

Multiplexor:

A slave-to-master multiplexor is required to multiplex the read data bus and response signals from the slaves to the master. The decoder provides control for the multiplexor. A single centralized multiplexor is required in all AHB-Lite implementations that use two or more slaves.

Operation:

The master starts a transfer by driving the address and control signals. These signals provide information about the address, direction, width of the transfer, and indicate if the transfer forms part of a burst. Transfers can be:

- Single
- Incrementing bursts that do not wrap at address boundaries
- Wrapping bursts that wrap at particular address boundaries.

The write data bus moves data from the master to a slave, and the read data bus moves data from a slave to the master. Every transfer consists of:

1. **Address phase** one address and control cycle
2. **Data phase** one or more cycles for the data.

A slave cannot request that the address phase is extended and therefore all slaves must be capable of sampling the address during this time. However, a slave can request that the master extends the data phase by using **HREADY**. This signal when LOW, causes wait states to be inserted into the transfer and enables the slave to have extra time to provide or sample data. The slave uses **HRESP** to indicate the success or failure of a transfer.

5.1 Architecture Code for Single Master and Slave with Linear and Wrapping Bursts:

**********Verilog Code**********

```
module master(clk, haddr,hready,hburst,hsize,htrans, hwrite, hwdata_out,hrdata_in);
 //////////////////////////////////////
 input clk;
 output reg [7:0] haddr;
 input hready;
 output reg [2:0] hburst;
 output reg [1:0] hsize;
 output reg [2:0] htrans;
 output reg hwrite;
 output reg [7:0] hwdata_out;
 input wire [7:0] hrdata_in;
 reg [9:0] master_memory [63:0];
 reg [1:0] r2;
 reg [2:0] wrap_size;
 reg r1;
 reg [7:0] prev_addr;
 reg [((2) - 1):0] hlf2_addr2;
 reg [((3) - 1):0] hlf2_addr3;
 reg [((4) - 1):0] hlf2_addr4;
 reg [((5) - 1):0] hlf2_addr5;
 reg [((6) - 1):0] hlf2_addr6;
 reg [(7 - (2)):0] hlf1_addr2;
 reg [(7 - (3)):0] hlf1_addr3;
 reg [(7 - (4)):0] hlf1_addr4;
 reg [(7 - (5)):0] hlf1_addr5;
 reg [(7 - (6)):0] hlf1_addr6;
 event master_event;
 event hwrite_event;
```

```
event hsize_check_event;
event haddr_event;
event wrap_address_event;
event r2_event;
event wrap_size_product_event: //Wrap size product event
int y;
int k;
int wrap_bits;
int burst_size;
int mast_indx;
int size;
int size_product;//product of hsize case and hburst case

initial
 begin
 y = 4;
 mast_indx = 0;
 burst_size = 4;
 hwrite = 1;
 end

always @(posedge clk)
 begin
  ->r2_event;
 end

always @(r2_event)
 begin
  r1 = hready;
  r2 = r2<<1;
  r2[0] = r1;
 end
```

```
always @(r2)
 begin
  if(r2==3)
   begin
    if(y == burst_size)
     begin
      hwrite = $urandom%2;
      $display("Write generated is %h at time %t", hwrite, $time);
       ->master_event;
     end
   end
 end

always @(master_event)
 fork
  hburst = $urandom%8;
  $display("Burst genereated is %h at time %t", hburst, $time);
  haddr = $urandom%9;
  hsize = $urandom%3;
  $display("Size genereated is %h at time %t", hsize, $time);
  ->hsize_check_event;
  ->haddr_event;
  ->hwrite_event;
  ->wrap_size_product_event;
  ->wrap_address_event;
 join

always @(hburst)
 begin
case (hburst)
 0 : burst_size = 1;
```

```
  2 : burst_size = 4;
  3 : burst_size = 4;
  4 : burst_size = 8;
  5 : burst_size = 8;
  6 : burst_size = 16;
  7 : burst_size = 16;
 default : burst_size = 1;
endcase
 end

 always @(hsize_check_event)
 begin
  case (hsize)
 0 : size = 1;
 1 : size = 2;
 2 : size = 4;
 default : size = 1;
endcase
 end

always @(wrap_size_product_event)
 begin
  size_product = burst_size*size;
  case(size_product)
   4 : wrap_bits = 2;
   8 : wrap_bits = 3;
   16 : wrap_bits = 4;
   32 : wrap_bits = 5;
   64 : wrap_bits = 6;
  endcase
end
```

```
always @(haddr_event)
 begin
  k = 0;
  if (hburst == 0 || hburst == 3 || hburst == 5 || hburst == 7)
   begin
  while( k <= ((burst_size) - 1))
   begin
    if(hready)
     begin
      prev_addr = haddr;
      prev_addr = prev_addr + size;
      haddr = prev_addr;
      $display("Address generated is %h at time %t", haddr, $time);
      if (k==0)
       begin
        htrans = 3;
       end
      else
       begin
        htrans = 4;
       end
      k++;
     end
    #10;
   end
  end
end


always @(wrap_address_event)
 begin
  k=0;
  if (hburst == 2 || hburst == 4 || hburst == 6)
```

```
            begin
            case (wrap_bits)
             2 : begin
hlf2_addr2 = haddr[((2) - 1) : 0];//Right half of address
hlf1_addr2 = haddr[7 : 2];//Left half of address
            while( k <= ((burst_size) - 1))
                        begin
                        if (hready)
                                begin
                        hlf2_addr2 = hlf2_addr2 + size;
             haddr = {hlf1_addr2, hlf2_addr2};
                        $display("WRAPPING address generated is %h at time %t", haddr, $time);
                          if (k==0)
                                        begin
                                         htrans = 3;
                                        end
                                else
                                        begin
                                         htrans = 4;
                                        end
                         k++;
                        end
                        #10;
                end
                end
3 : begin
            if(hburst == 2 && size == 2)
                        begin
            hlf2_addr3 = haddr[((3) - 1) : 0];//Right half of address
        hlf1_addr3 = haddr[7 : 3];//Left half of address
            while( k <= ((burst_size) - 1))
                        begin
```

```
                    if (hready)
                            begin
                    hlf2_addr3 = hlf2_addr3 + size;
            haddr = {hlf1_addr3, hlf2_addr3};
                    $display("WRAPPING address generated is %h at time %t", haddr, $time);
                     if (k    0)
                                    begin
                                     htrans = 3;
                                    end
                            else
                                    begin
                                     htrans = 4;
                                    end
                      k++;
                    end
                    #10;
                end
                end
if(hburst == 4 && size == 1)
                begin
          hlf2_addr3 = haddr[((3) - 1) : 0];//Right half of address
    hlf1_addr3 = haddr[7 : 3];//Left half of address
          while( k <= ((burst_size) - 1))
                    begin
                    if (hready)
                            begin
                    hlf2_addr3 = hlf2_addr3 + size;
           haddr = {hlf1_addr3, hlf2_addr3};
                    $display("WRAPPING address generated is %h at time %t", haddr, $time);
                            if (k==0)
                                    begin
                                     htrans = 3;
```

```
                                        end
                                else
                                        begin
                                         htrans = 4;
                                        end
                                  k++;
                                end
                                #10;
                        end
                        end
 end
4 : begin
 if(hburst == 2 && size == 4)
                        begin
     hlf2_addr4 = haddr[((4) - 1) : 0];//Right half of address
     hlf1_addr4 = haddr[7 : 4];//Left half of address
            while( k <= ((burst_size) - 1))
                                begin
                                if (hready)
                                        begin
                                hlf2_addr4 = hlf2_addr4 + size;
            haddr = {hlf1_addr4, hlf2_addr4};
                                $display("WRAPPING address generated is %h at time %t", haddr, $time);
                                        if (k==0)
                                                begin
                                                 htrans = 3;
                                                end
                                        else
                                                begin
                                                 htrans = 4;
                                                end
                                        k++;
```

```
                    end
                    #10;
                end
                end
if(hburst == 4 && size     2)
                begin
    hlf2_addr4 = haddr[((4) - 1) : 0];//Right half of address
    hlf1_addr4 = haddr[7 : 4];//Left half of address
        while( k <= ((burst_size) - 1))
                    begin
                    if (hready)
                        begin
                    hlf2_addr4 = hlf2_addr4 + size;
            haddr = {hlf1_addr4, hlf2_addr4};
                    $display("WRAPPING address generated is %h at time %t", haddr, $time);
                     if (k==0)
                            begin
                             htrans = 3;
                            end
                        else
                            begin
                             htrans = 4;
                            end
                        k++;
                    end
                    #10;
                end
                end
if(hburst == 6 && size == 1)
                begin
    hlf2_addr4 = haddr[((4) - 1) : 0];//Right half of address
    hlf1_addr4 = haddr[7 : 4];//Left half of address
```

```
            while( k <= ((burst_size) - 1))
                                begin
                                if (hready)
                                        begin
                                hlf2_addr4 = hlf2_addr4 + size;
             haddr = {hlf1_addr4, hlf2_addr4};
                                $display("WRAPPING address generated is %h at time %t", haddr, $time);
                                 if (k==0)
                                                begin
                                                 htrans = 3;
                                                end
                                        else
                                                begin
                                                 htrans = 4;
                                                end
                                 k++;
                                end
                                #10;
                        end
                        end
 end
5 : begin
 if(hburst == 4 && size == 4)
                        begin
     hlf2_addr5 = haddr[((5) - 1) : 0];//Right half of address
     hlf1_addr5 = haddr[7 : 5];//Left half of address
            while( k <= ((burst_size) - 1))
                                begin
                                if (hready)
                                        begin
                                hlf2_addr5 = hlf2_addr5 + size;
             haddr = {hlf1_addr5, hlf2_addr5};
```

```
                $display("WRAPPING address generated is %h at time %t", haddr, $time);
                if (k==0)
                        begin
                         htrans = 3;
                        end
                else
                        begin
                         htrans = 4;
                        end
                 k+=;
                end
                #10;
        end
        end
if(hburst == 6 && size == 2)
        begin
   hlf2_addr5 = haddr[((5) - 1) : 0];//Right half of address
   hlf1_addr5 = haddr[7 : 5];//Left half of address
      while( k <= ((burst_size) - 1))
                begin
                if (hready)
                        begin
                hlf2_addr5 = hlf2_addr5 + size;
     haddr = {hlf1_addr5, hlf2_addr5};
                $display("WRAPPING address generated is %h at time %t", haddr, $time);
                 if (k==0)
                        begin
                         htrans = 3;
                        end
                else
                        begin
                         htrans = 4;
```

```
                                                end
                                        k++;
                                        end
                                        #10;
                                end
                                end
        end
      6 : begin
        hlf2_addr6 = haddr[((6) - 1) : 0];//Right half of address
        hlf1_addr6    haddr[7 : 6];//Left half of address
                        while( k <= ((burst_size) - 1))
                                        begin
                                        if (hready)
                                                begin
                                        hlf2_addr6 = hlf2_addr6 + size;
                         haddr = {hlf1_addr6, hlf2_addr6};
                                        $display("WRAPPING address generated is %h at time %t", haddr, $time);
                                        if (k==0)
                                                        begin
                                                         htrans = 3;
                                                        end
                                                else
                                                        begin
                                                         htrans = 4;
                                                        end
                                        k++;
                                        end
                                        #10;
                                end
                                end
      endcase
                        end
```

```
 end


always @(hwrite_event)
 begin
  if(hwrite)
   begin
    y = 0;
    while( y <= ((burst_size) - 1))
     begin
      #10;
      if(hready)
       begin
        hwdata_out = $urandom%31;
        $display("Write data is %h at time %t", hwdata_out, $time);
        y++;
       end
     end
   end
  else
   begin
       y = 0;
    while( y <= ((burst_size) - 1))
     begin
      #10;
      if(hready)
       begin
                                    master_memory[mast_indx] = hrdata_in;
        $display("Data entered in master memory at address %h is %h at time %t",mast_indx,hrdata_in, $time);
                mast_indx++;
        y++;
        if(mast_indx>63)
                        begin
```

```
                    $display("Master memory is full");
                    $finish;
                    end
            end
          end
        end
      end

endmodule

module slave(clk, haddr,hready,hburst,hsize,htrans, hwrite, hwdata_in,hrdata_out);

 input clk;
 input reg [7:0] haddr;
 output reg hready;
 input wire [2:0] hburst;
 input wire [1:0] hsize;
 input wire [2:0] htrans;
 input wire hwrite;
 input wire [7:0] hwdata_in;
 output reg [7:0] hrdata_out;
 reg [9:0] slave_memory [63:0];
 int burst_size;
 reg [9:0] sprev_addr;
 reg [9:0] sprev_addr2;
 int e;
 int t;
 parameter n = 10;

 event hburst_check_event;
 event hburst_force_event;
 event hwrite_and_hready_force_event;
```

```
event address_generate_event;
event hsize_check_event;
event read_and_write_event;

initial
 begin
  hready = 1;
  sprev_addr2 = haddr;
 end

initial
 begin
  for( t= 0; t<= 63; t++)
   begin
    slave_memory[t] = $urandom%31;
   end
 end

always @(posedge clk)
 begin
  hready = $urandom%2;
 end

always @(hburst)
 fork
                    ->hburst_check_event;
        ->hsize_check_event;
        ->read_and_write_event:
 join

always @(hburst_check_event)
 begin
```

```
case (hburst)
 0 : burst_size = 1;
 3 : burst_size = 4;
 5 : burst_size = 8;
 7 : burst_size = 16;
default : burst_size = 1;
endcase
 end



always @(haddr)
   begin
    if (hwrite)
         begin
         sprev_addr = sprev_addr2;
        $display("Address is %h at time %t", sprev_addr, $time);
                              wait (hwdata_in)
                              begin
              $display("Data is %h", hwdata_in);
              slave_memory[sprev_addr] = hwdata_in;
      $display("Data entered in slave memory at address %h is %h at time %t", sprev_addr, hwdata_in, $time);
              sprev_addr2 = haddr;
         end
         end
    else
     begin
      if((~hwrite))
                begin
                        t = 0;
          while( t <  ((burst_size) - 1))
                                begin
                                if(hready)
```

```
                                        begin
                    hrdata_out = slave_memory[haddr];
                    $display("Read data is %h at time %t", hrdata_out, $time);
                                                t++;
                                        end
                                end
                                end
                        end
            end

endmodule
```

**********Testbench**********

```
module top;
 reg clk;
 wire [7:0] haddr;
 wire hwrite;
 wire [7:0] hwdata;
 wire [7:0] hrdata;
 wire hready;
 wire [2:0] hburst;
 wire [1:0] hsize;
 wire [2:0] htrans;

 initial
  begin
   $dumpfile("gtyhd.vcd");
   $dumpvars;
  clk = 1;
  end
```

```
always

  #5 clk = ~clk;

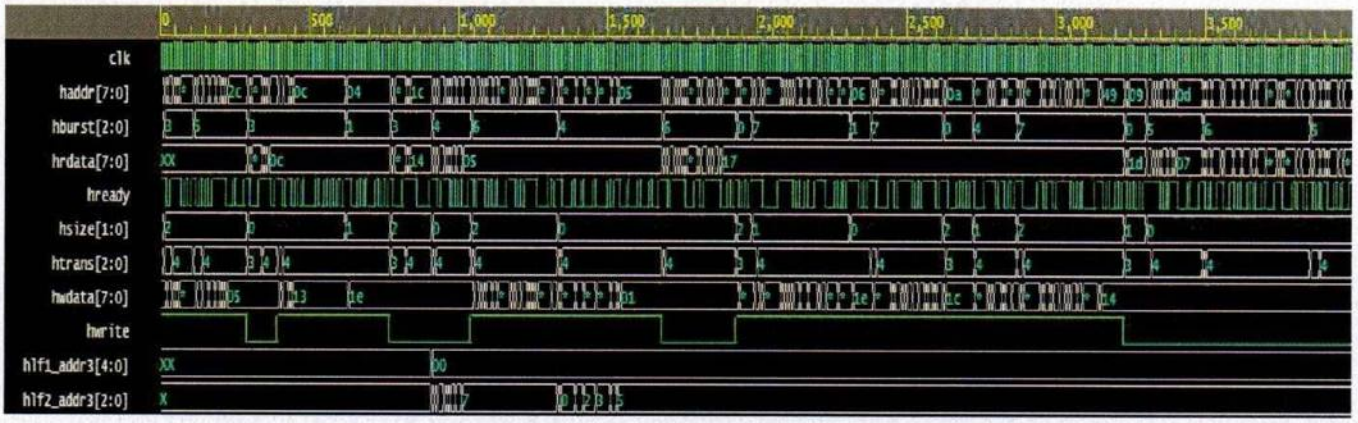

  master m1(.clk(clk), .haddr(haddr), .hwrite(hwrite), .hwdata_out(hwdata), .hready(hready),.hrdata_in(hrdata), .hburst(hburst),
.hsize(hsize), .htrans(htrans));


  slave s1(.clk(clk), .haddr(haddr), .hwrite(hwrite), .hwdata_in(hwdata), .hready(hready),.hrdata_out(hrdata), .hburst(hburst), .hsize(hsize),
.htrans(htrans));

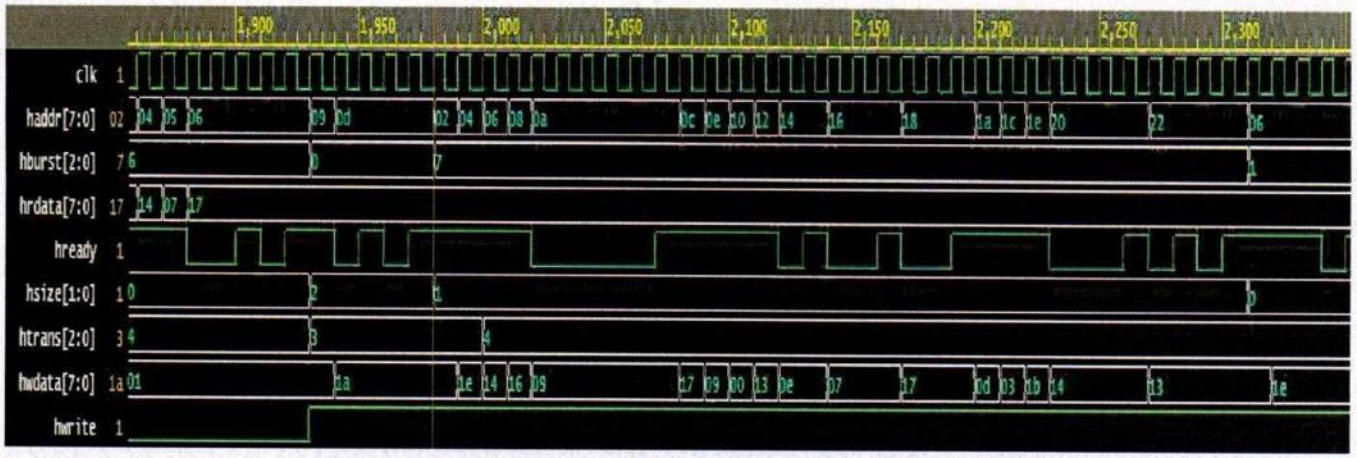

endmodule
```

**********Output**********

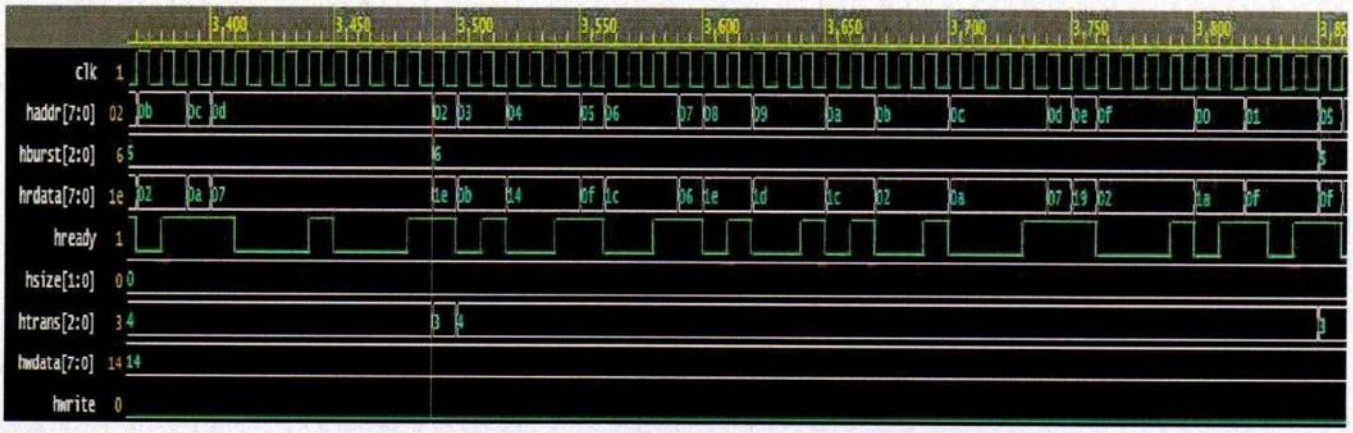
1. Full Output



2. Zoomed Output for Linear Burst



3. Zoomed Output for Wrapping Burst



6.Field Programmable Gate Arrays (FPGAs)

A **field-programmable gate array (FPGA)** is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC). (Circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare.)

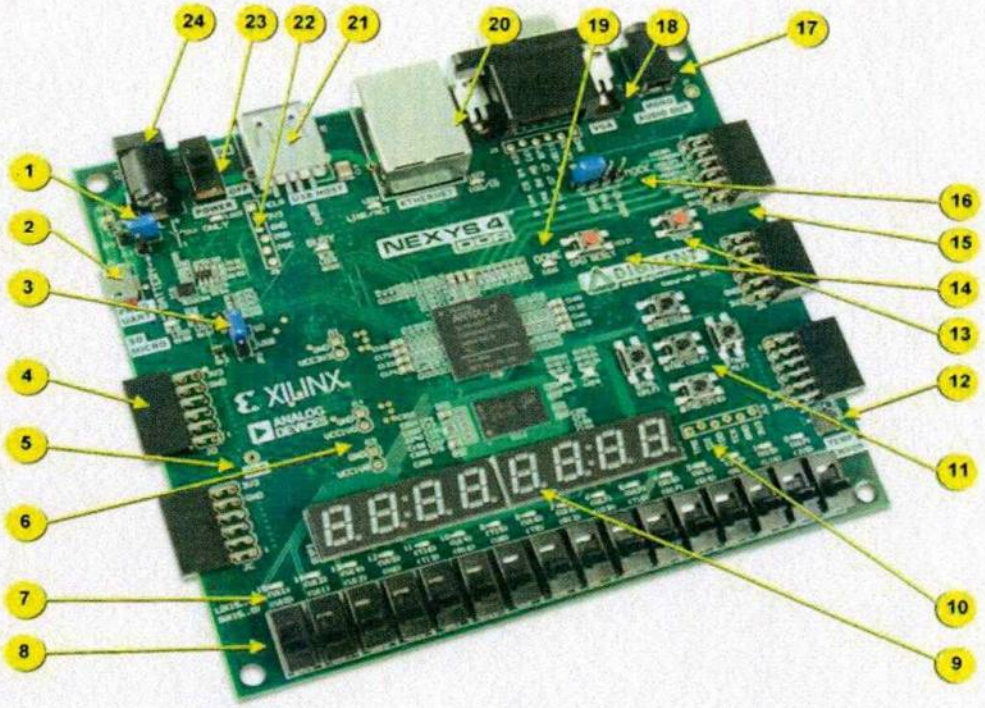
FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

Contemporary field-programmable gate arrays (FPGAs) have large resources of logic gates and RAM blocks to implement complex digital computations. As FPGA designs employ very fast I/O rates and bidirectional data buses, it becomes a challenge to verify correct timing of valid data within setup time and hold time. Floor planning enables resource allocation within FPGAs to meet these time constraints. FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, partial re-configuration of a portion of the design[2] and the low non-recurring engineering costs relative to an ASIC design (notwithstanding the generally higher unit cost), offer advantages for many applications.

7.Nexys4 DDR FPGA Board[5]

The Nexys 4 DDR board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx®. With its large, high-capacity FPGA (Xilinx part number XC7A100T-1CSG324C), generous external memories, and collection of USB, Ethernet, and other ports, the Nexys4 DDR can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys4 DDR to be used for a wide range of designs without needing any other components.

The Nexys4 DDR is compatible with Xilinx's new high-performance Vivado® Design Suite as well as the ISE® toolset, which includes ChipScope™ and EDK. Xilinx offers free WebPACK™ versions of these toolsets, so designs can be implemented at no additional cost. The Nexys4 DDR is not supported by the Digilent Adept Utility.



Callout	Component Description	Callout	Component Description
1	Power select jumper and battery header	13	FPGA configuration reset button
2	Shared UART/ JTAG USB port	14	CPU reset button (for soft cores)
3	External configuration jumper (SD / USB)	15	Analog signal Pmod port (XADC)
4	Pmod port(s)	16	Programming mode jumper
5	Microphone	17	Audio connector
6	Power supply test point(s)	18	VGA connector
7	LEDs (16)	19	FPGA programming done LED
8	Slide switches	20	Ethernet connector
9	Eight digit 7-seg display	21	USB host connector
10	JTAG port for (optional) external cable	22	PIC24 programming port (factory use)
11	Five pushbuttons	23	Power switch
12	Temperature sensor	24	Power jack

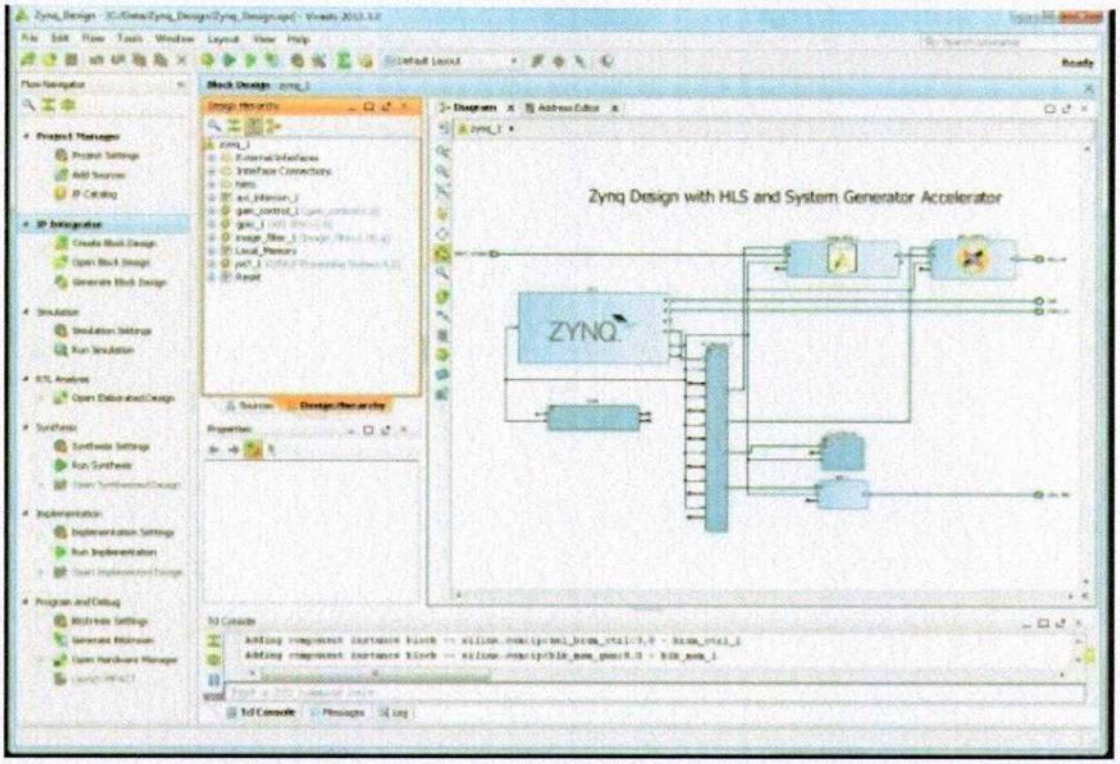
7.1Xilinx Vivado[6]

Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISEwith additional features for system on a chip development and high-level synthesis. Vivado represents a ground-up rewrite and re-thinking of the entire design flow (compared to ISE), and has been described by reviewers as "well-conceived, tightly integrated, blazing fast, scalable, maintainable, and intuitive".

Vivado enables developers to synthesize (compile) their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Vivado is a design environment for FPGA products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors.

Vivado was introduced in April 2012,[] and is an integrated design environment (IDE) with system-to-IC level tools built on a shared scalable data model and a common debug environment. Vivado includes electronic system level (ESL) design tools for synthesizing and verifying C-based algorithmic IP; standards based packaging of both algorithmic and RTL IP for reuse; standards based IP stitching and systems integration of all types of

system building blocks; and the verification of blocks and systems. A free version WebPACK Edition of Vivado provides designers with a limited version of the design environment.



As of 2018, Xilinx recommends Vivado Design Suite for new designs with Ultrascale, Ultrascale+, Virtex-7, Kintex-7, Artix-7, and Zynq-7000.

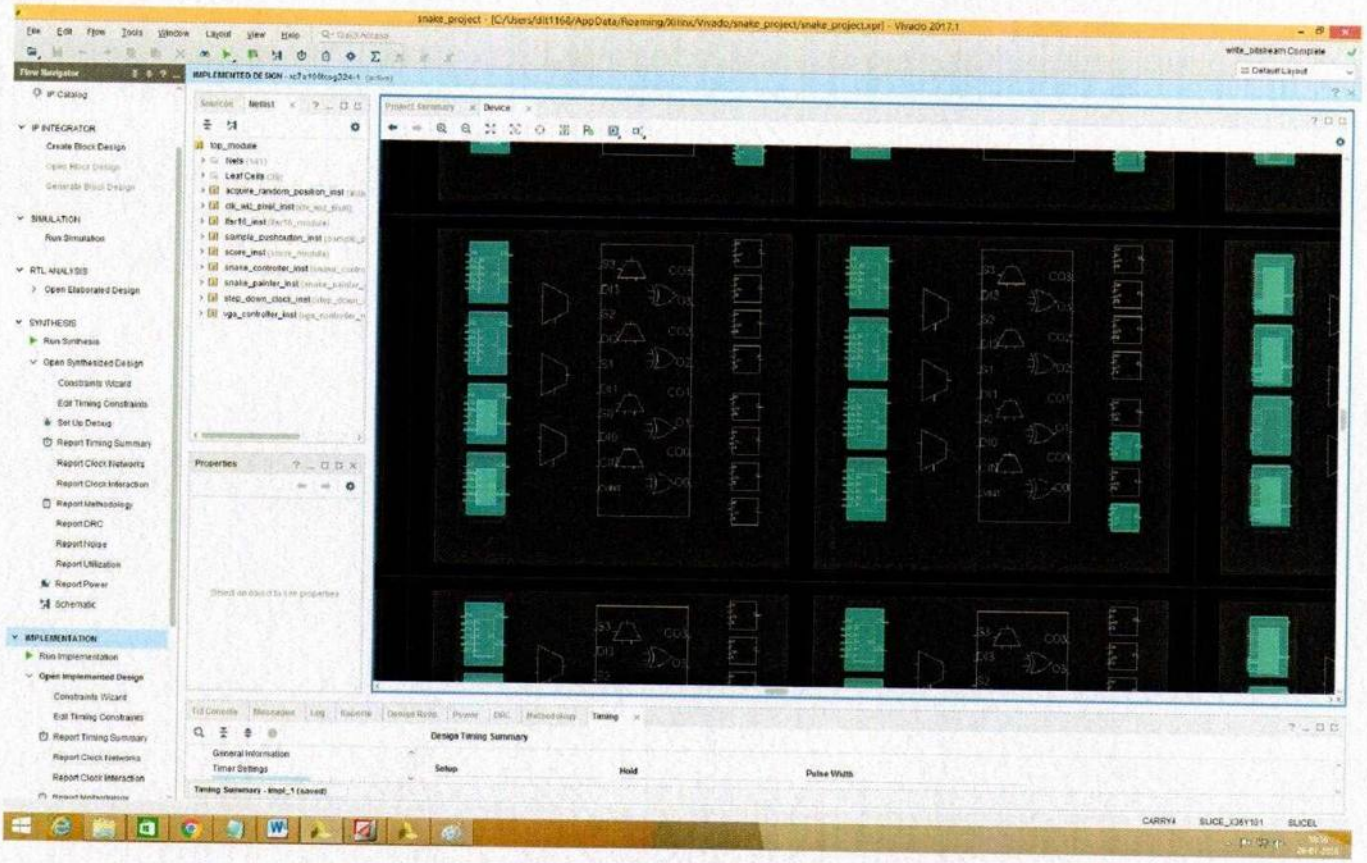
Vivado supports newer high capacity devices, and speeds the design of programmable logic and I/O. Vivado provides faster integration and implementation for programmable systems into devices with 3D stacked silicon interconnect technology, ARM processing systems, analog mixed signal (AMS), and many semiconductor intellectual property (IP) cores.

Vivado is targeted at Xilinx's larger FPGAs, and is slowly replacing Xilinx ISE as their mainline tool chain. As of 2014, Vivado covers Xilinx's mid-scale and large FPGAs, and ISE covered the mid-scale and smaller FPGAs and all CPLDs.

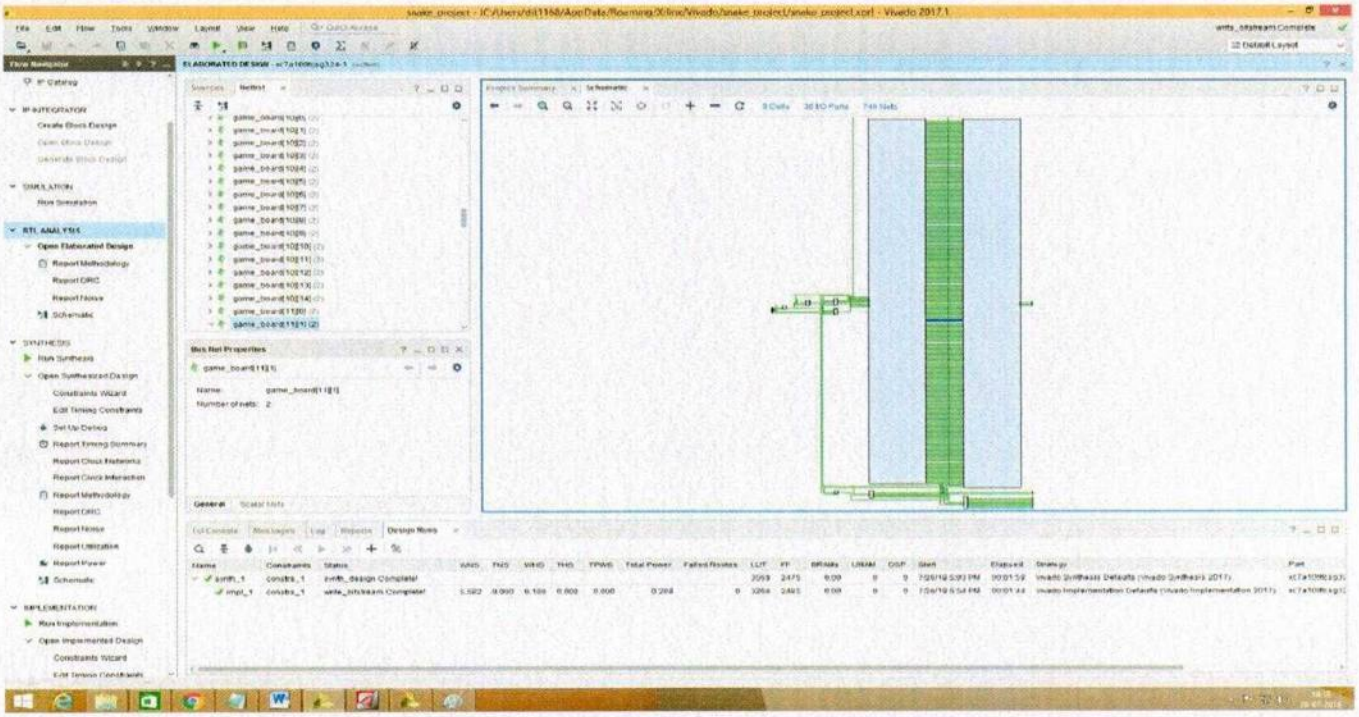
8.Snake Game Implementation using Nexys4 DDR and Vivado

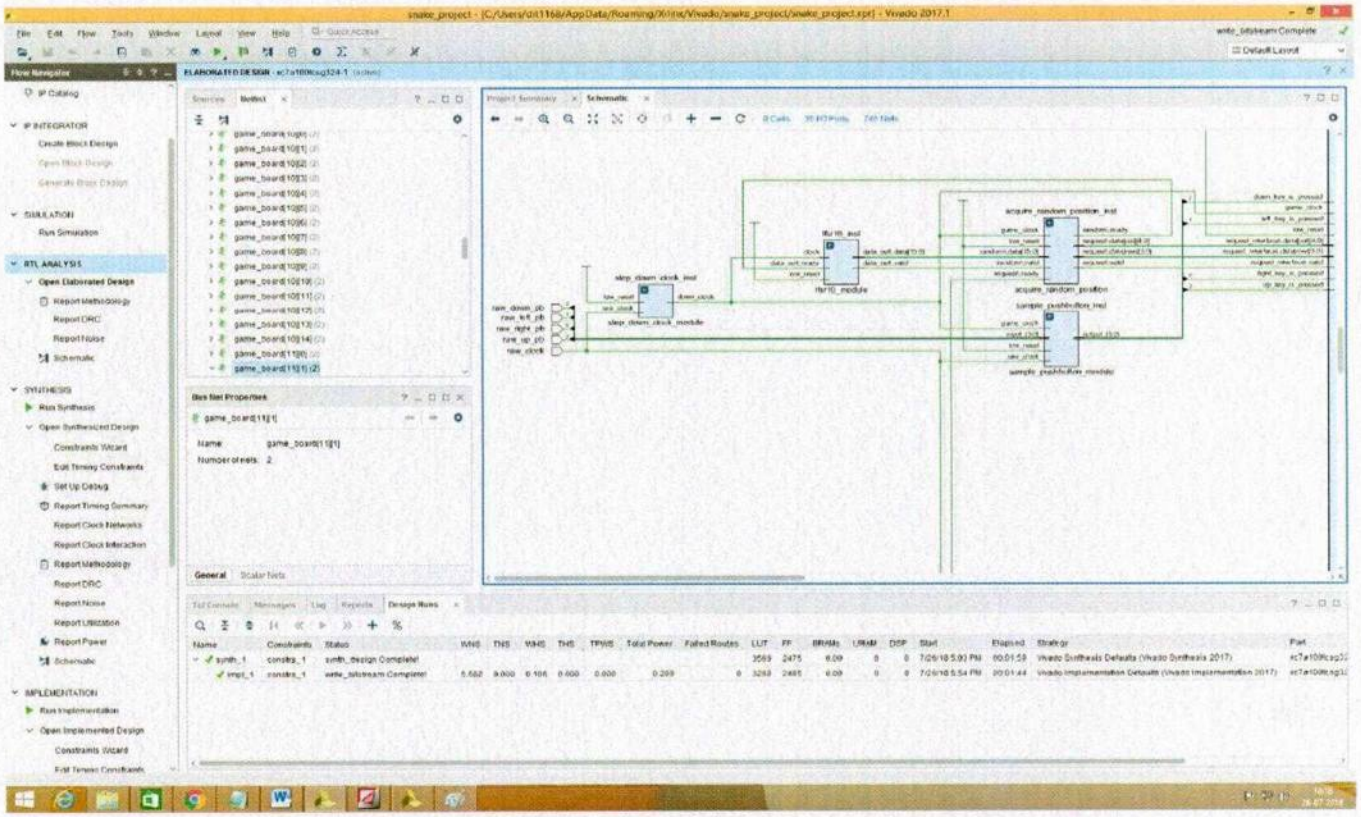
A game of snake eating a pixel was implemented using the board. The length and speed of the snake increases as it eats the green pixel.

1. Implemented Design

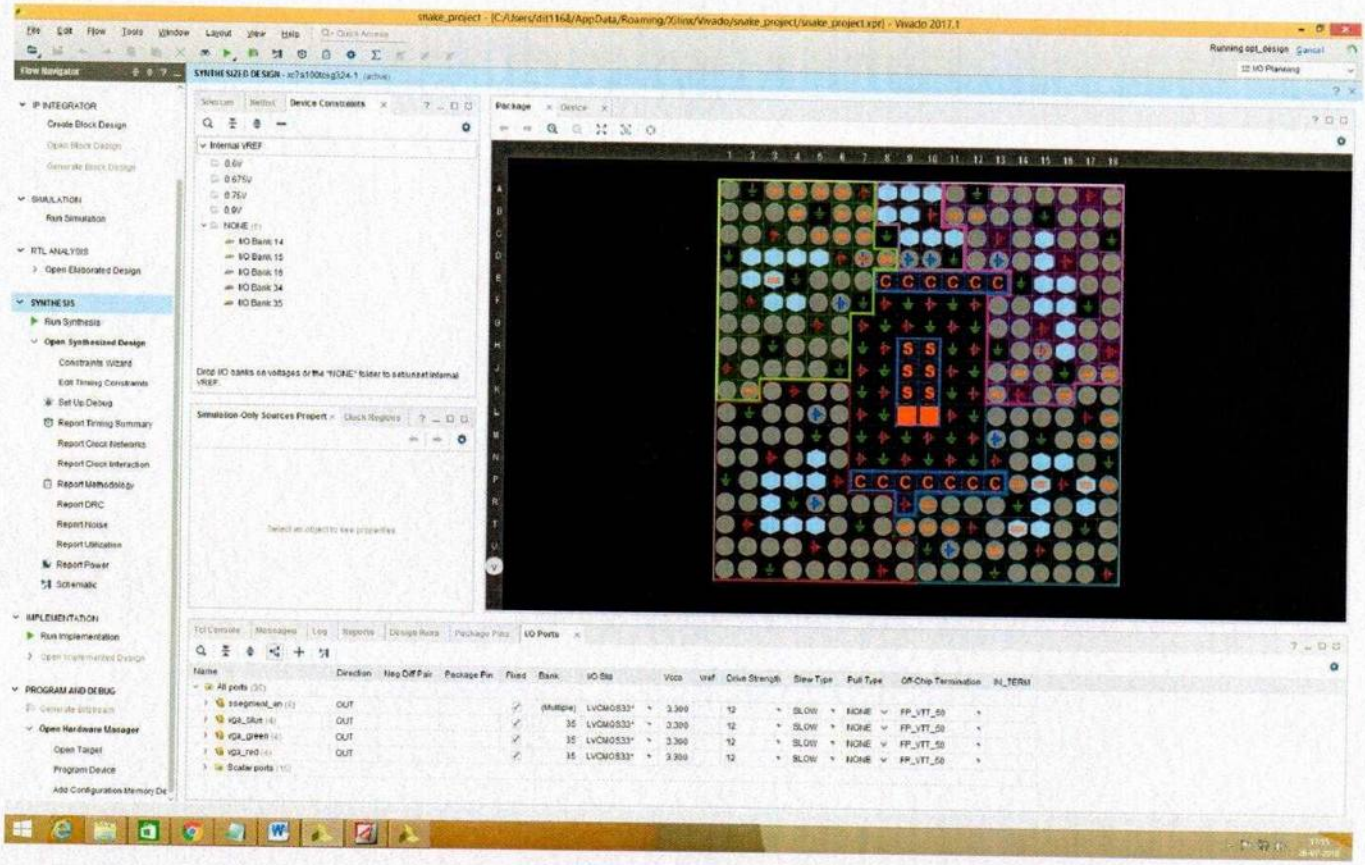


2. RTL Designs

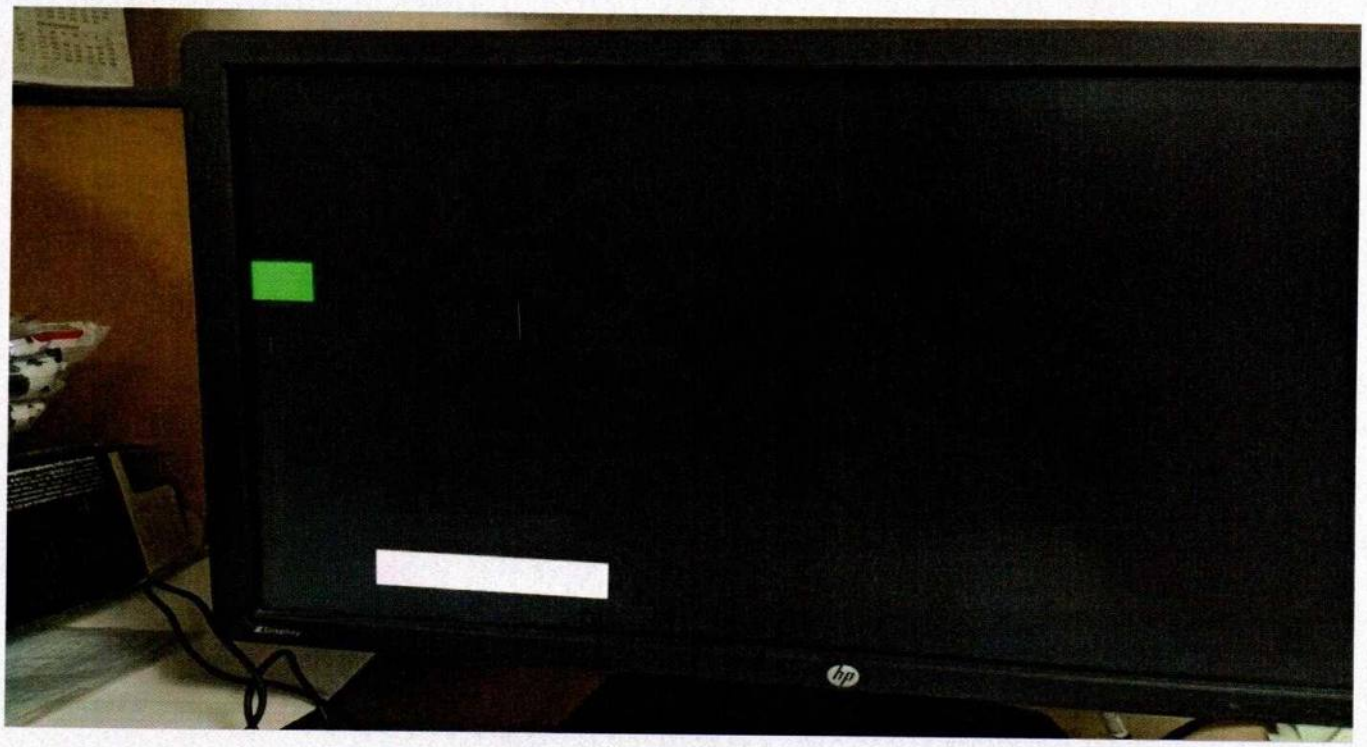




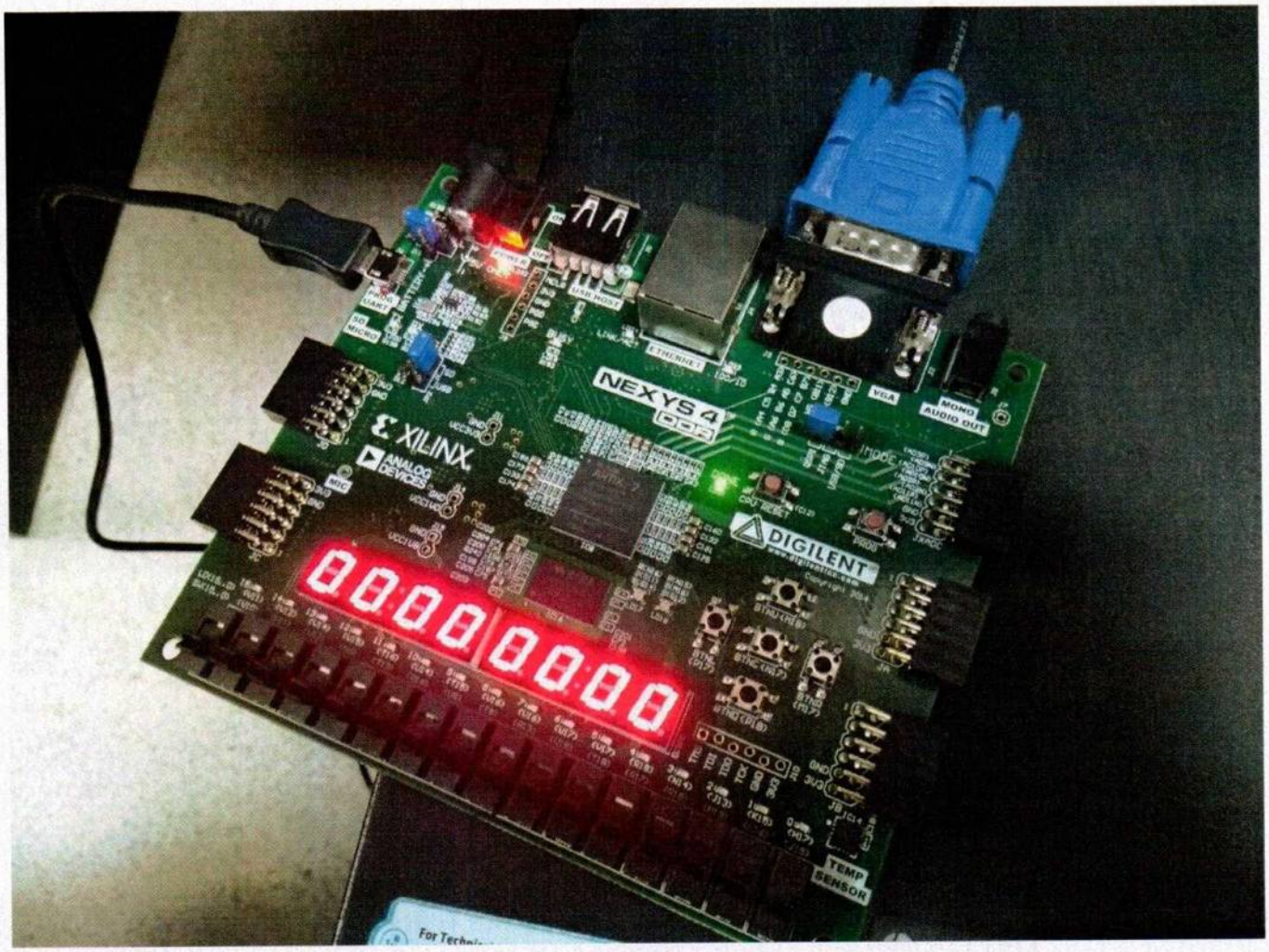
3. Synthesis



OUTPUT:



CONNECTION


NEXYS 4
XILINX
ANALOG DEVICES
DIGILENT
VGA
MONO AUDIO OUT
TEMP SENSOR

9.References

[1]. https://www.mepits.com/tutorial/143/vlsi/hardware-description-language

[2]. https://en.wikipedia.org/wiki/Verilog

[3].https://blog.digilentinc.com/battle-over-the-fpga-vhdl-vs-verilog-who-is-the-true-champ/

[4].http://eda-playground.readthedocs.io/en/latest/intro.html

[5].https://reference.digilentinc.com/reference/programmable-logic/nexys-4/reference-manual

[6]. https://en.wikipedia.org/wiki/Xilinx_Vivado

1. ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf

2. [J._Bhasker]_A_Verilog_HDL_Primer.pdf