

MINI PROJECT IN VLSI DESIGN

(Subject Code: EC-383)



GUIDE: PROF. SUSHIL KUMAR PANDEY

Report submission

by

K A GAGANASHREE (201EC228)

MEGHNA UPPULURI (201EC237)

DEPARTMENT OF ELECTRONICS & COMMUNICATION
ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA, SURATHKAL
SRINIVASNAGAR 575025 KARNATAKA INDIA

July 2022

TABLE OF CONTENTS

S.NO	TITLE	PAGE NO.
1	Motivation	2
2	Objective	2
3	Introduction	2
4	Results	3
5	Implementation in FPGA	6
6	Designing Custom IP	10
7	Final Results and Outcomes	13
	APPENDIX	13

1. MOTIVATION:

We wanted to explore the scope of the hardware implementation of RTL Code. Since we had a basic understanding of how synthesizable code is generated in Vivado, working with FPGA (Field Programmable Gate Array) would help us learn how RTL code written in Verilog is integrated on the FPGA.

2. OBJECTIVE:

Implementation of Verilog Integrator for Lorentz Equations using Nexys 4 DDR FPGA Development Kit.

3. INTRODUCTION:

The Lorenz model is a three-dimensional independent differential system derived from the truncation of an expansion of the equations of convection (stress-free top/bottom plates and periodic lateral boundary conditions).

x_t = amplitude of first horizontal harmonic of the vertical velocity

y_t = amplitude of the corresponding temperature fluctuations

z_t = uniform correction to the temperature field

σ = Prandtl number

ρ = Rayleigh number

β = horizontal wave vector

It is represented by three coupled ordinary differential equations, given below

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \beta z\end{aligned}$$

We will numerically integrate these differential equations using the [Euler method](#), the simplest of all numerical integration techniques. The Euler method linearises the values of the state variables x , y and z at the current timestep to approximate the importance of the state variables at the next time step. That is:

$$x(k+1) = x(k) + dt \cdot \left(\frac{dx}{dt} \right) \Big|_{x=x(k)}$$

$$y(k+1) = y(k) + dt \cdot \left(\frac{dy}{dt} \right) \Big|_{y=y(k)}$$

$$z(k+1) = z(k) + dt \cdot \left(\frac{dz}{dt} \right) \Big|_{z=z(k)}$$

Starting this iteration with initial parameters taken as:

$$\sigma = 10$$

$$\beta = 8/3$$

$$\rho = 28$$

$$dt = 1/256$$

$$x(0) = -1$$

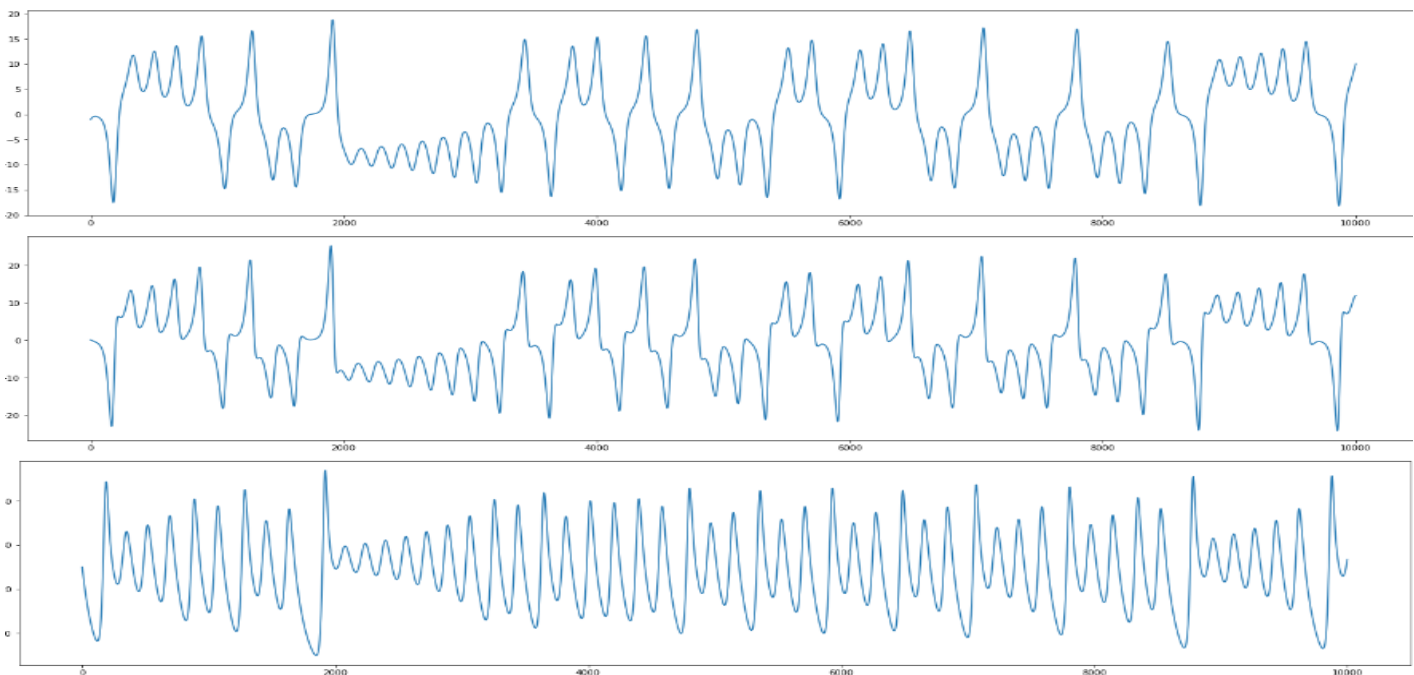
$$y(0) = 0.1$$

$$z(0) = 25$$

4. RESULTS:

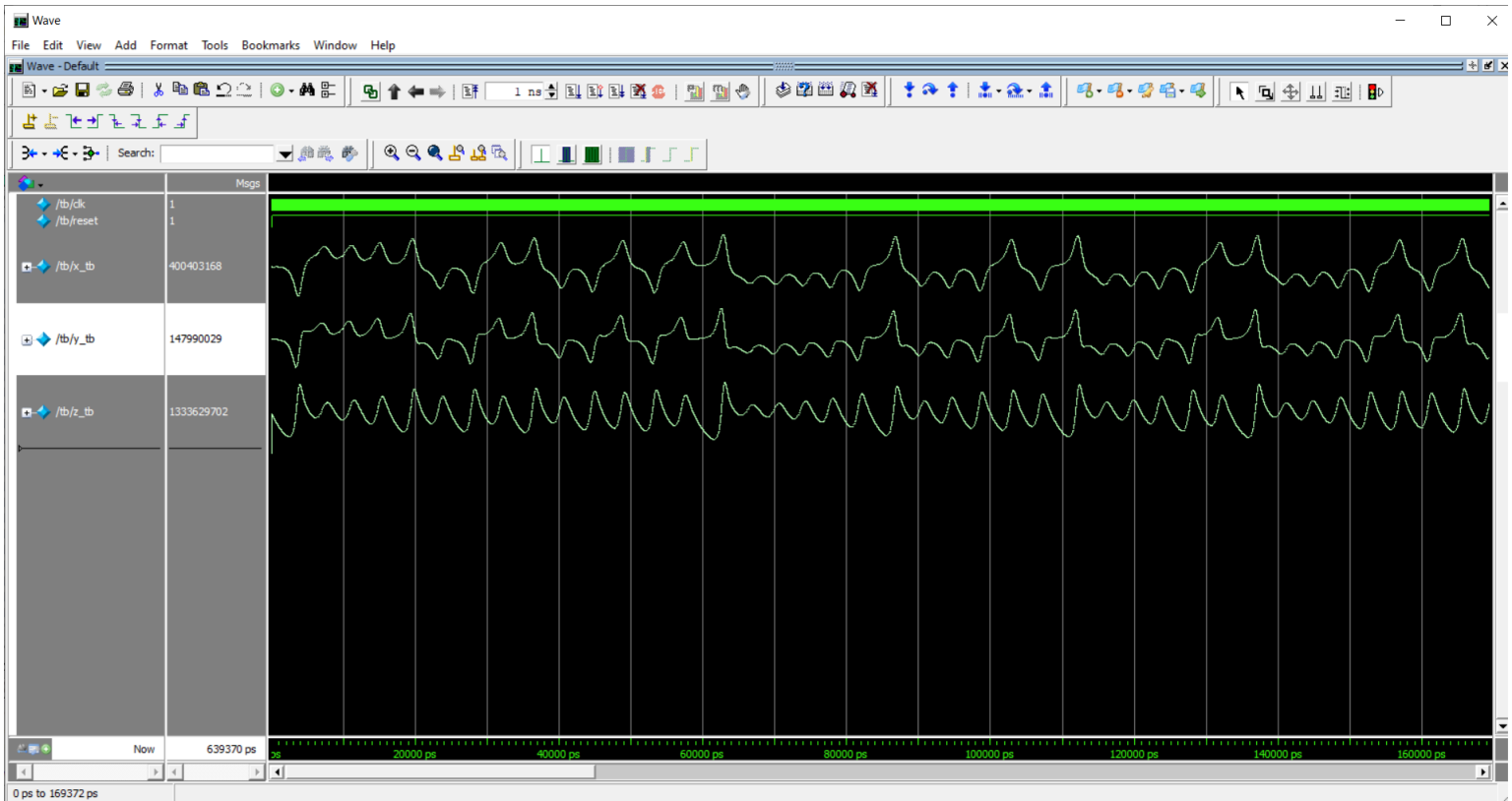
[Python Code](#)

Waveform:



Implementation in Verilog: [Verilog Code](#)

Waveform:



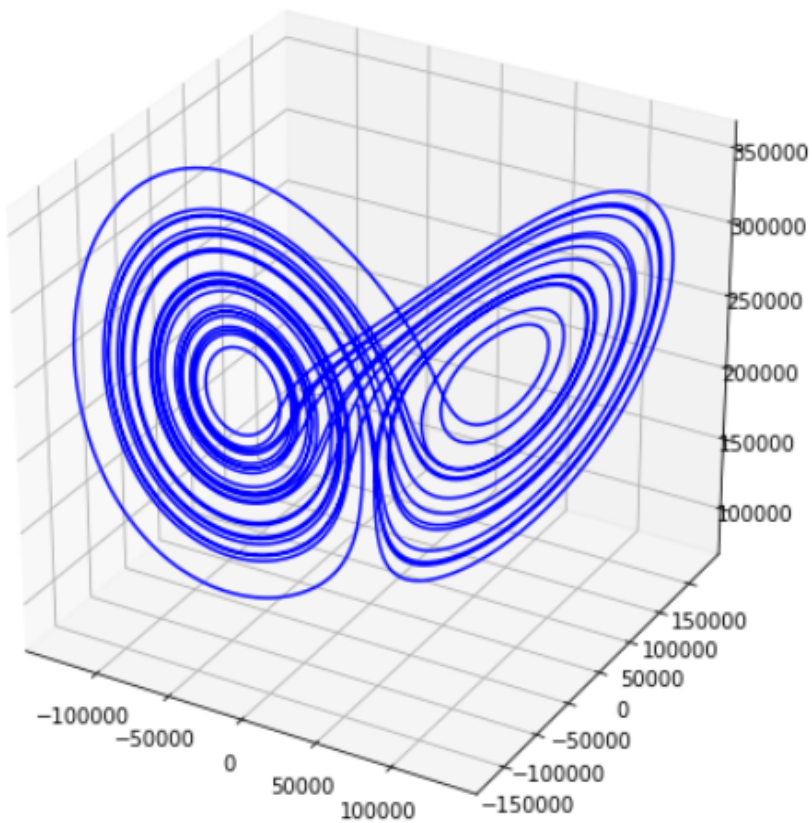
When the vectors points are extracted as a list of approximate 10000 vector points from the source code in Modelsim and plotted using Python

[Python Code](#)

Glimpse of the extracted list (All values in Signed Decimal

	1	2	X	Y	Z
1	10	+2	-33554432	104857	838860800
2	50	+2	-32239622	-288768	832238525
3	70	+2	-30991532	-690306	825670019
4	90	+2	-29807892	-1098384	819154771
5	110	+2	-28686422	-1511790	812692274
6	130	+2	-27624912	-1929447	806282029
7	150	+2	-26621182	-2350413	799923541
8	170	+2	-25673112	-2773869	793616326
9	190	+2	-24778612	-3199116	787359906
10	210	+2	-23935662	-3625556	781153811
11	230	+2	-23143392	-4052600	774997580

Plot:



5. IMPLEMENTATION IN FPGA

AXI4 Protocol:

Advanced eXtensible Interface 4 (AXI4) is a family of buses defined as part of the fourth generation of the ARM Advanced Microcontroller Bus Architecture (AMBA) standard. AXI was first introduced with the third generation of AMBA, AXI3, in 1996.

The AMBA specification defines 3 AXI4 protocols:

- AXI4: A high-performance memory-mapped data and address interface. Capable of Burst access to memory-mapped devices.
- AXI4-Lite: A subset of AXI, lacking burst access capability. Has a simpler interface than the full AXI4 interface.
- AXI4-Stream: A fast unidirectional protocol for transferring data from master to slave.

Clock and Reset

Any AXI component has two global signals: the clock **ACLK** and an active-low asynchronous reset **ARESETN**. All AXI4 signals are sampled on the rising edge of the clock and all signal changes must occur after the rising edge.

Handshake Process

All five transaction channels use the same VALID/READY handshake process to transfer addresses, data, and control information. This two-way flow control mechanism means both the master and slave can control the rate at which the information moves between master and slave. The information source generates the **VALID** signal to indicate when the address, data or control information is available. The information destination generates the **READY** signal to indicate that it can accept the information. The handshake completes if both **VALID** and **READY** signals in a channel are asserted during a rising clock edge.

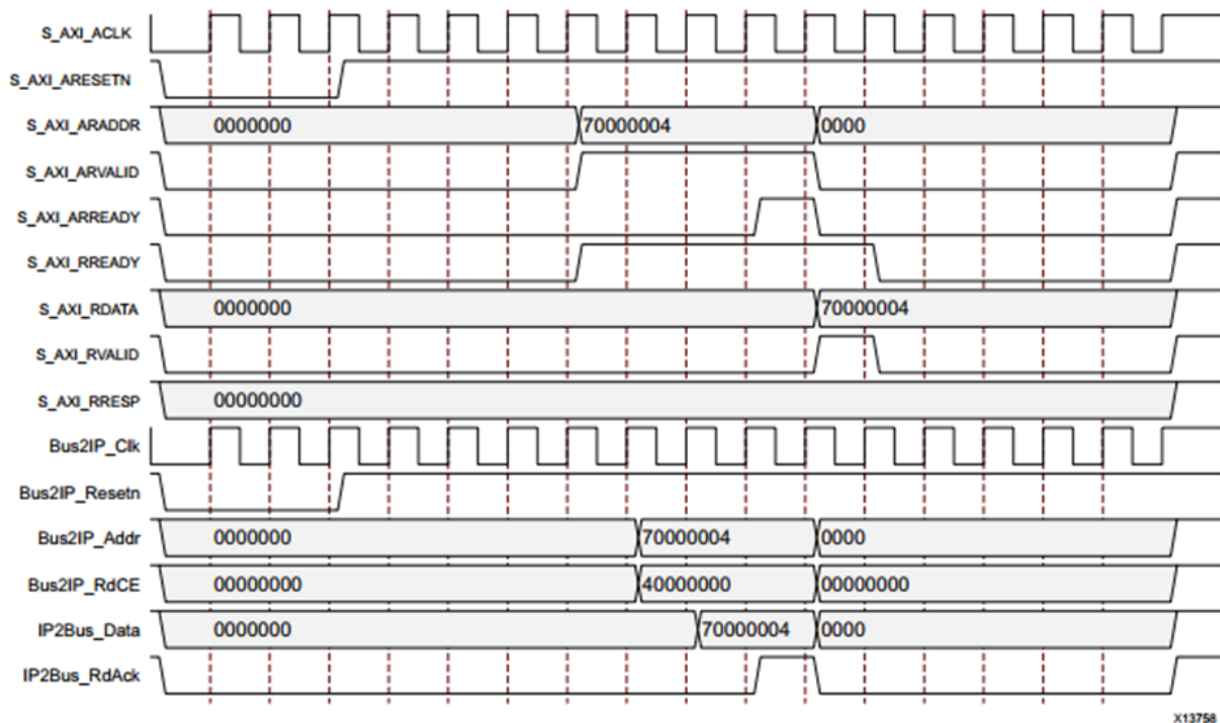
AXI4-Lite Read Transaction

Below, the sequence for an AXI4-Lite read is shown:

A description of the events in figure 3 follows:

1. The Master puts an address on the Read Address channel as well as asserting ARVALID, indicating the address is valid, and RREADY, indicating the master is ready to receive data from the slave.
2. The Slave asserts ARREADY, indicating that it is ready to receive the address on the bus.

3. Since both ARVALID and ARREADY are asserted, on the next rising clock edge the handshake occurs, after this the master and slave deassert ARVALID and the ARREADY, respectively. (At this point, the slave has received the requested address).
4. The Slave puts the requested data on the Read Data channel and asserts RVALID, indicating the data in the channel is valid. The slave can also put a response on RRESP, though this does not occur here.
5. Since both RREADY and RVALID are asserted, the next rising clock edge completes the transaction. RREADY and RVALID can now be deasserted.



AXI4-Lite Read Transaction.

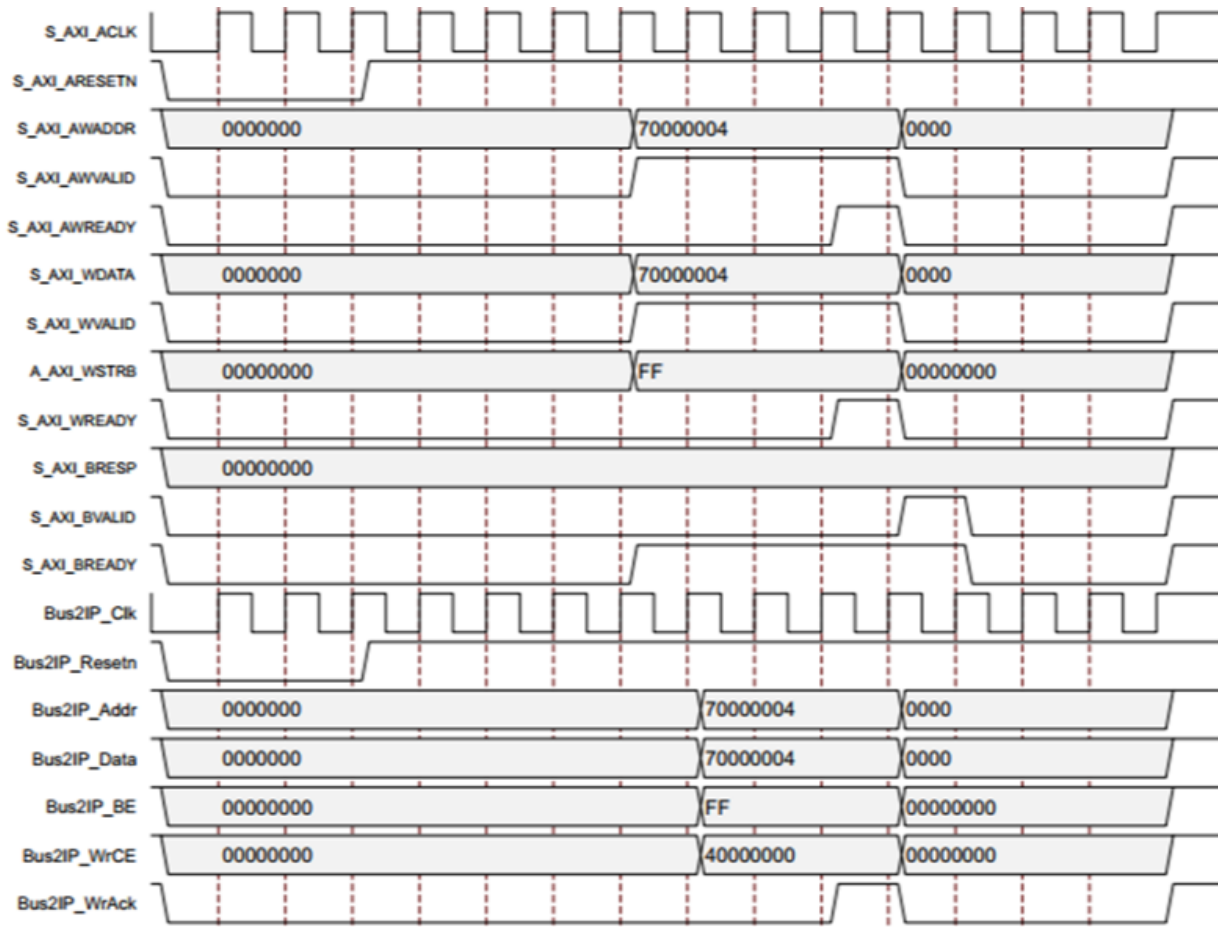
AXI4-Lite Write Transaction

Below, the sequence for an AXI4-Lite write is shown:

A description of the events in figure 4 follows:

1. The Master puts an address on the Write Address channel and data on the Write data channel. At the same time, it asserts AWVALID and WVALID indicating the address and data on the respective channels are valid. BREADY is also asserted by the Master, indicating it is ready to receive a response.
2. The Slave asserts AWREADY and WREADY on the Write Address and Write Data channels, respectively.

3. Since Valid and Ready signals are present on both the Write Address and Write Data channels, the handshakes on those channels occur and the associated Valid and Ready signals can be deasserted. (After both handshakes occur, the slave has the write address and data)
4. The Slave asserts BVALID, indicating there is a valid response on the Write response channel. (in this case, the response is 2'b00, that being 'OKAY').
5. The next rising clock edge completes the transaction, with both the Ready and Valid signals on the write response channel high.

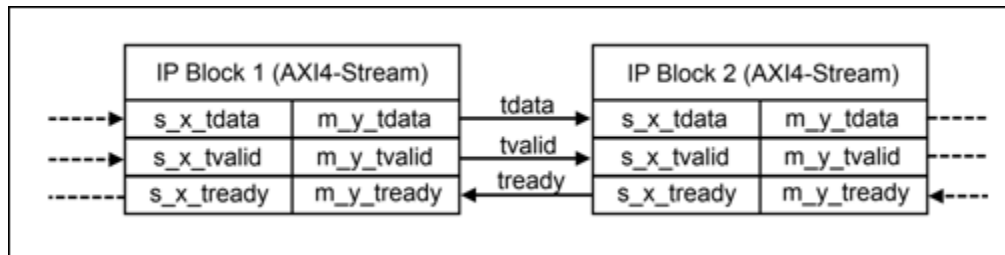


AXI4-Lite Write Transaction

Note: The Handshakes on the Write Address and Write Data channel do not necessarily occur simultaneously (as they do in the shown transaction). However, the AXI4 specification states that both must occur before the slave can send a written response. Both Write Address and Write Data handshakes can occur independently or simultaneously and no order is enforced, only that both must occur to complete the transaction.

AXI to AXI Interconnect

The following diagram shows a conceptual configuration for the interconnection of two AXI4-Stream IP blocks.



The signals that flow downstream serve a similar purpose to those signals in the four-wire handshaking protocol. They inform downstream blocks of the validity of the data coming from the upstream block. For example, assuming two simple blocks with input signal **x** and output signal **y**, the **m_y_tvalid** output of *IP Block 1* is TRUE whenever its **m_y_tdata** output contains valid **y** data.

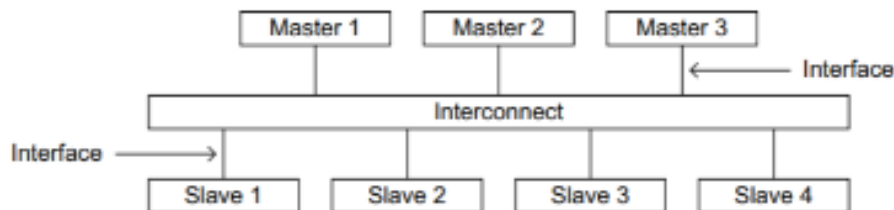


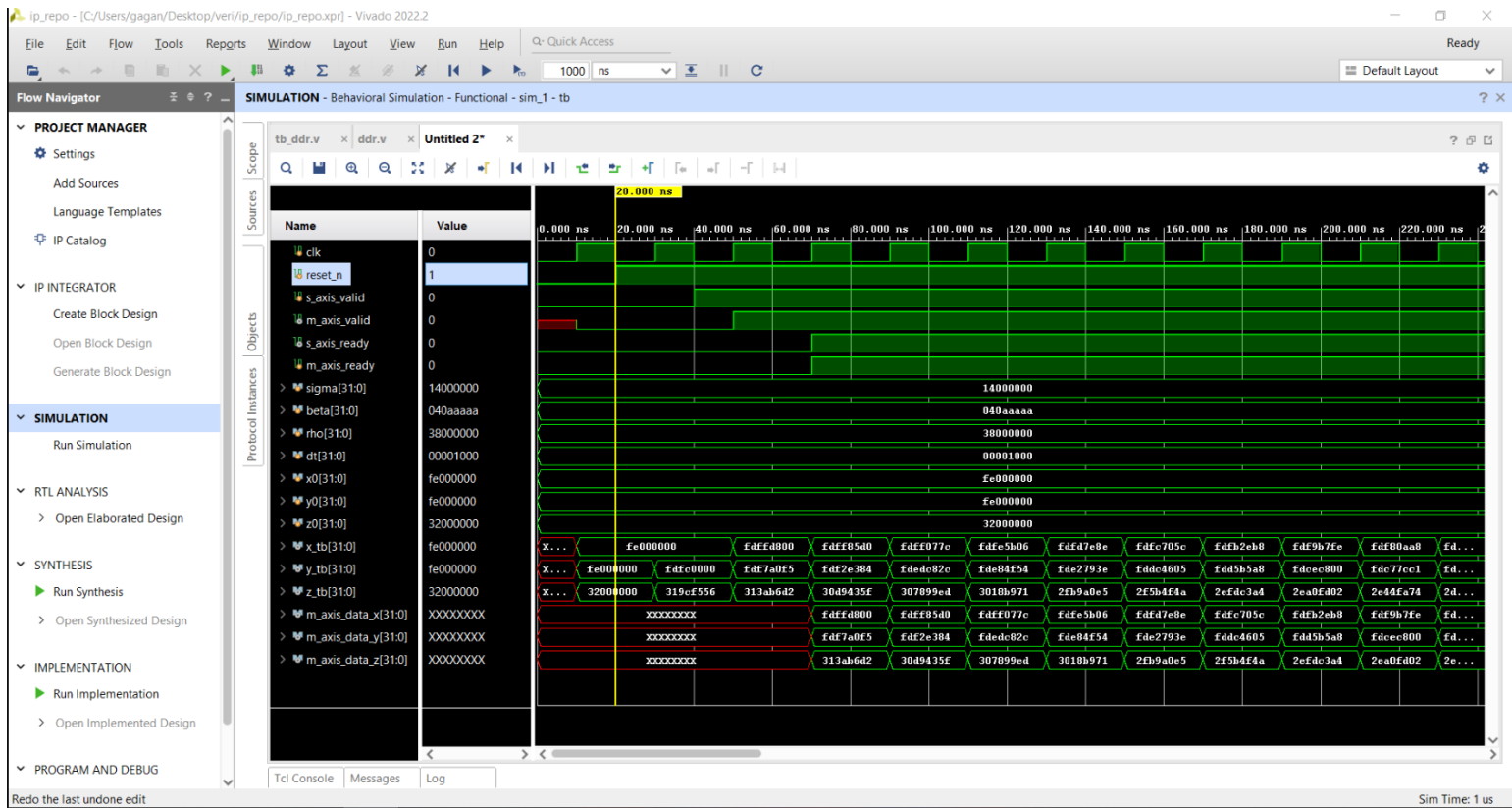
Figure A1-3 Interface and interconnect

A typical system consists of a number of master and slave devices connected together through some form of interconnect, as figure A1-3 shows:

- Between a master and the interconnect
- Between a slave and the interconnect
- Between a master and a slave

To get a much better understanding of how the Master-Slave Protocol works we implemented a two-way handshake mechanism. The Slave would process the inputs and write back to the Master once m_axis_ready, and s_axis_valid are high. ([Code](#) is provided in *Appendix*)

Waveform:



6. Designing Custom IP

For our Custom IP, we will be integrating AXI4-Lite(Advanced Extensible Interface) which will be the interface for MicroBlaze Soft Core Processor which we will run on FPGA.

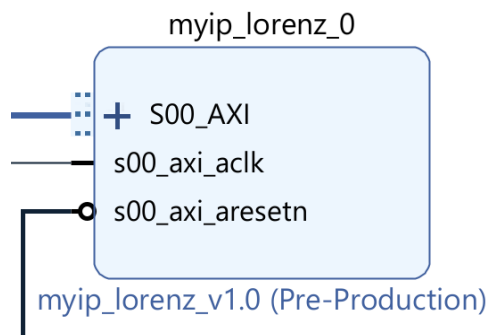


MicroBlaze is a soft microprocessor core designed for Xilinx field-programmable gate arrays (FPGA). As a soft-core processor, MicroBlaze is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs.

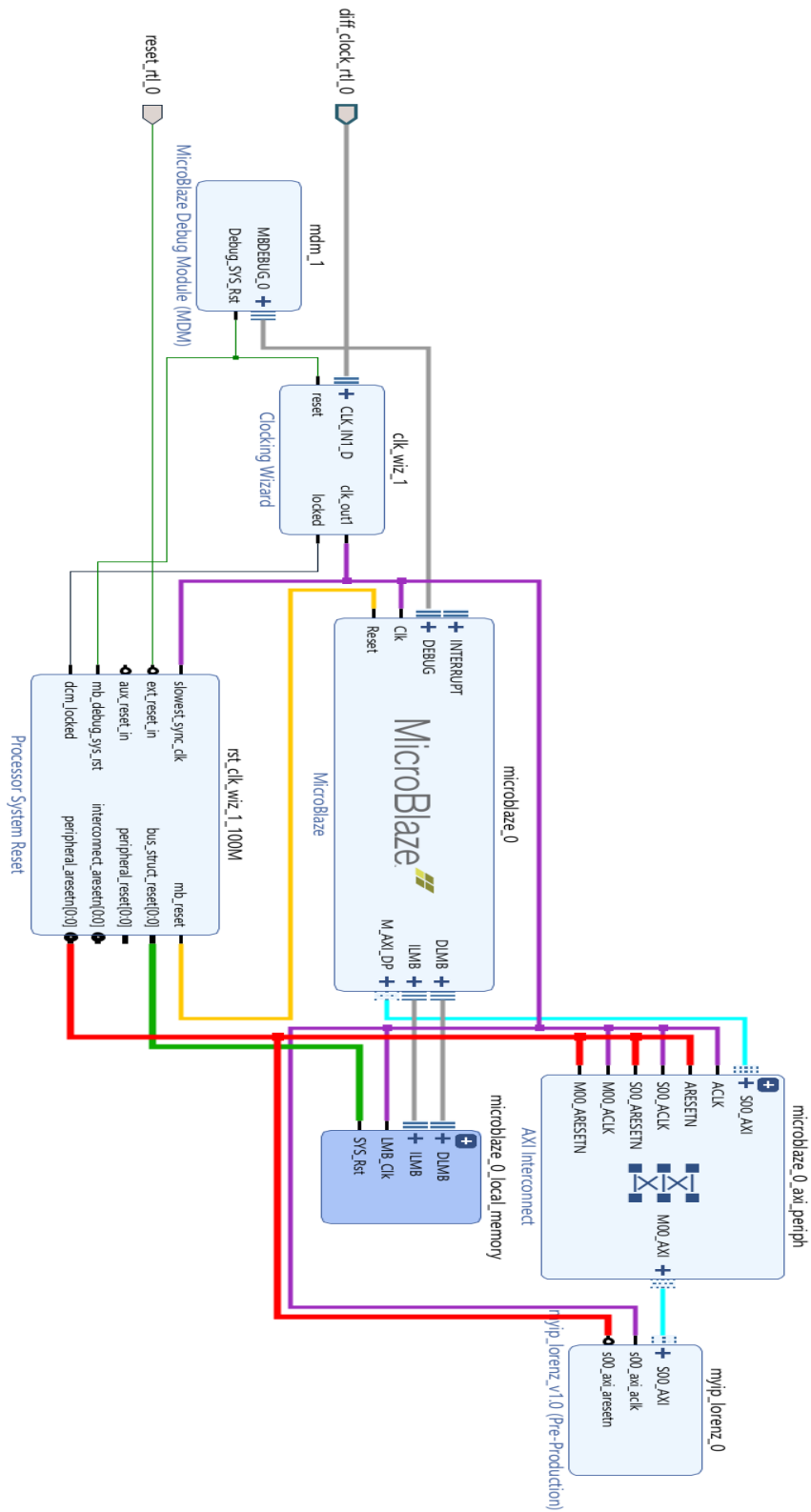
We will use Vivado IP Integrator to configure and build the hardware specification of their embedded system (processor core, memory controller, I/O peripherals, etc.) The IP Integrator converts the designer's block design into a synthesizable RTL description (Verilog or VHDL) and automates the implementation of the embedded system (from RTL to the bitstream file.) For the MicroBlaze core, Vivado generates an encrypted (non-human-readable) netlist.

Then the **SDK(Software Development Kit)** handles the software that will execute on the embedded system. Powered by the GNU toolchain (GNU Compiler Collection, GNU Debugger), the SDK enables programmers to write, compile, and debug C/C++ applications for their embedded systems. Xilinx's tools provide the possibility of running software in simulation or using a suitable FPGA board to download and execute on the actual system.

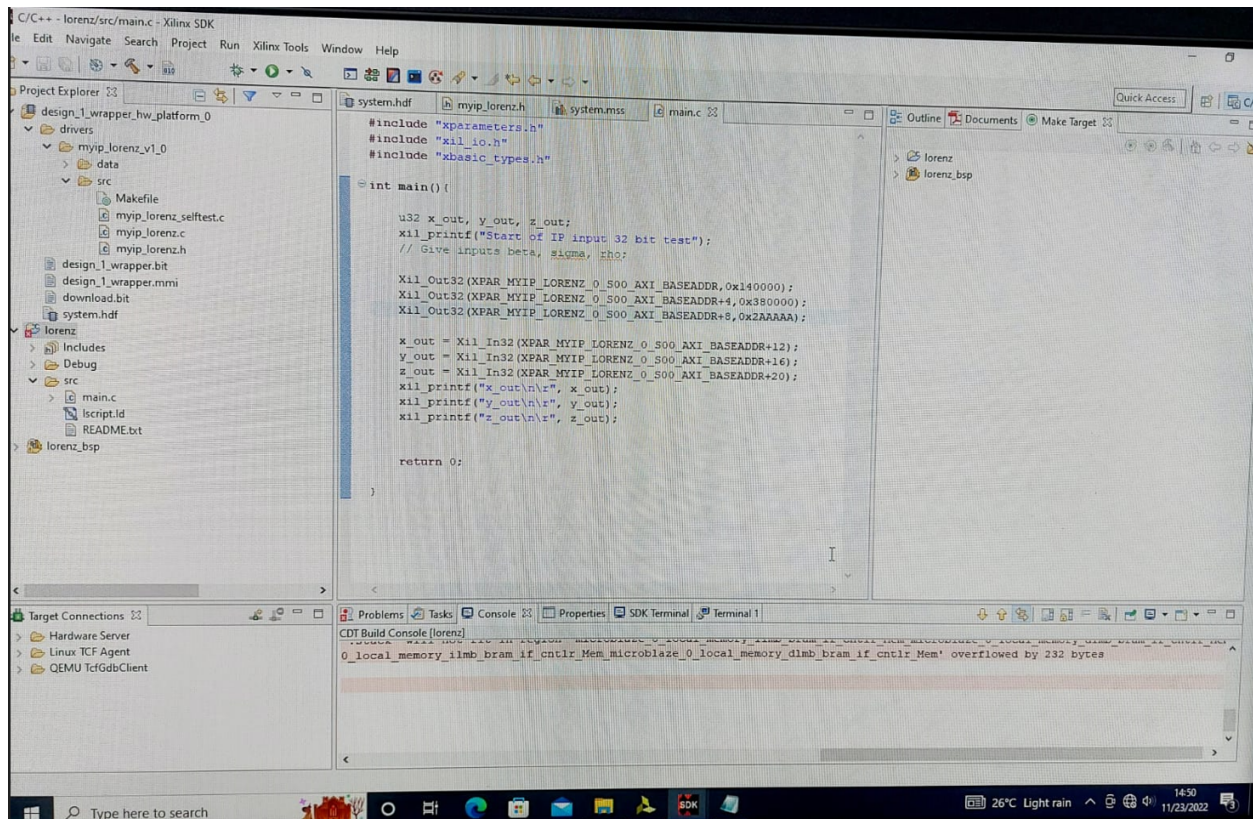
Custom AXI-Lite IP Graphic User Interface GUI



Custom IP integrated MicroBlaze Soft Core with AXI-Lite Interface:



SDK results, we got overflow :)



7. FINAL RESULTS AND OUTCOMES:

- Implemented Verilog Code with Testbench
- Generated Custom IP for AXI4-Lite Interface
- Integrated the Custom IP with MicroBlaze Soft Core and generated the Bit Stream File Successfully

APPENDIX

[Main codes added in GitHub Repository](#)

Python Test Code:

```
import matplotlib.pyplot as plt
import numpy as np
dt = (1./256)
x = [-1.]
```

```

y = [0.1]
z = [25.]
sigma = 10.0
beta = 8./3.
rho = 28.0
def dx(sigma, x, y):
    return sigma*(y-x)
def dy(rho, x, y, z):
    return x*(rho-z)-y
def dz(beta, x, y, z):
    return x*y - beta*z
for i in range(10000):
    x.extend([x[i] + dt*dx(sigma, x[i], y[i])])
    y.extend([y[i] + dt*dy(rho, x[i], y[i], z[i])])
    z.extend([z[i] + dt*dz(beta, x[i], y[i], z[i])])
plt.rcParams["figure.figsize"] = (25,5.5)
plt.plot(range(10001),x)
plt.show()
plt.plot(range(10001),y)
plt.show()
plt.plot(range(10001),z)
plt.show()

```

Verilog Code

```

// 32 bit parameter
// For some reason code is working in modelsim and giving overflow in vivado
for x and y values
// overflow is not happening for z value
module ddr #(parameter N = 32) (
    input clk, reset,
    input signed [N-1:0] sigma, beta, rho, dt, x0, y0,z0,
    output signed [N-1:0] x, y, z
);

wire signed [N-1:0] dtx, dty, dtz;
wire signed [N-1:0] s1_out, s2_out, s3_out, s4_out;

assign dtx = x >>> 8;
assign dty = y >>> 8;
assign dtz = z >>> 8;

//instantiate signed_mult modules

```

```

    // s1_out = sigma*(y-x)*dt
    // s2_out = (x*(rho-z)*dt
    // s3_out - s4_out = (x*y-beta*z)*dt
signed_mult s1 (.a(dty-dtx), .b(sigma), .out(s1_out));
signed_mult s2 (.a(dtx), .b(rho - z), .out(s2_out));
signed_mult xy (.a(x), .b(dty), .out(s3_out));
signed_mult bz (.a(dtz), .b(beta), .out(s4_out));

//instantiate integrator module

integrator int1(x, s1_out, x0, clk, reset);
integrator int2(y, (s2_out-dty), y0, clk, reset);
integrator int3(z, (s3_out-s4_out), z0, clk, reset);

endmodule

module integrator(out,funcnt,InitialOut,clk,reset);
    output signed [31:0] out;          //the state variable V
    input signed [31:0] funcnt;        //the dV/dt function
    input clk, reset;
    input signed [31:0] InitialOut;    //the initial state variable V

    wire signed [31:0] out, vlnew ;
    reg signed [31:0] v1 ;

    always @ (posedge clk)
    begin
        if (reset==0) //reset
            v1 <= InitialOut ; //
        else
            v1 <= vlnew ;
    end
    assign vlnew = v1 + funcnt ;
    assign out = v1 ;
endmodule

///// signed mult of 7.25 format 2'comp//////////
module signed_mult (out, a, b);
    output signed [31:0] out;
    input signed[31:0] a;
    input signed[31:0] b;
    // intermediate full bit length
    wire signed[63:0]mult_out;
    assign mult_out = a * b;
    // select bits for 7.25 fixed point

```



```

        assign out = {mult_out[63], mult_out[55:25]};
    endmodule

```

Testbench:

```

module tb();

    reg clk, reset;

    // reg [31:0] index;
    wire signed [31:0] x_tb, y_tb, z_tb;

    ddr DUT (
        .clk(clk),
        .reset(reset),
        .dt({7'h0, 25'h01000}),
        .x0({7'h7f, 25'd0}),
        .y0({7'h0, 25'h19999}),
        .z0({7'd25, 25'd0}),
        .beta({7'd2, 25'haaaaa}),
        .sigma({7'd10, 25'd0}),
        .rho({7'd28, 25'd0}),
        .x(x_tb),
        .y(y_tb),
        .z(z_tb)
    );

    //Initialize clocks and index
    initial begin
        // $dumpfile("dump.vcd");
        // $dumpvars(1);
        clk = 1'b0;
        reset = 1'b0;
        end

    //Toggle the clocks
    always begin
        #10
        clk = !clk;
    end

    //Intialize and drive signals
    initial begin
        #10 reset = 1'b0;
        #30 reset = 1'b1;
    end

endmodule

```

Python Code for vector points taken from ModelSim

```

x = []
y = []
z = []

with open('/content/vectors_pts.txt', 'r') as f:
    for line in f:
        first, second, x_comp, y_comp, z_comp = line.split()
        x.append(int(x_comp)/4096)
        y.append(int(y_comp)/4096)
        z.append(int(z_comp)/4096)

import numpy as np

x = np.array(x)
y = np.array(y)
z = np.array(z)
print(x)
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(8,8))
ax = plt.axes(projection = '3d')
ax.plot3D(x,y,z, 'blue')

```

Master -Slave protocol

Verilog Code:

```

module lorenz #(parameter DATA_WIDTH = 32) (s_axis_valid, s_axis_ready,
m_axis_valid,clk, reset_n, sigma, beta, rho, dt, x0, y0, z0, x, y, z,
m_axis_data_x, m_axis_data_y, m_axis_data_z,m_axis_ready);
input clk;

    input reset_n;

    //AXI4-S slave i/f which is accepting the data

    input s_axis_valid;

    input signed [DATA_WIDTH-1:0] sigma, beta, rho, dt, x0, y0, z0;

    output signed [DATA_WIDTH-1:0] x, y, z;

    output s_axis_ready;

    //AXI4-S master i/f which sends out data

```

```

output reg m_axis_valid;

output reg signed [DATA_WIDTH-1:0] m_axis_data_x, m_axis_data_y,
m_axis_data_z;

input m_axis_ready;


wire signed [DATA_WIDTH-1:0] dtx, dty, dtz;

wire signed [DATA_WIDTH-1:0] s1_out, s2_out, s3_out, s4_out;


assign dtx = x >>> 8;
assign dty = y >>> 8;
assign dtz = z >>> 8;


//instantiate signed_mult modules
// s1_out = sigma*(y-x)*dt
// s2_out = (x*(rho-z)*dt
// s3_out - s4_out = (x*y-beta*z)*dt
signed_mult s1 (.a(dty-dtx), .b(sigma), .out(s1_out));
signed_mult s2 (.a(dtx), .b(rho - z), .out(s2_out));
signed_mult xy (.a(dtx), .b(y), .out(s3_out));
signed_mult bz (.a(dtz), .b(beta), .out(s4_out));


//instantiate integrator module

integrator int1(x, s1_out, x0, clk, reset_n);
integrator int2(y, (s2_out-dty), y0, clk, reset_n);
integrator int3(z, (s3_out-s4_out), z0, clk, reset_n);

```

```

assign s_axis_ready = m_axis_ready;

always @(posedge clk) begin
    if(s_axis_valid & s_axis_ready) begin
        m_axis_data_x <= x;
        m_axis_data_y <= y;
        m_axis_data_z <= z;

    end
end

always @(posedge clk) begin
    m_axis_valid <= s_axis_valid ;
end

endmodule

```

```

module integrator(out,funcnt,InitialOut,clk,reset_n);

    output signed [31:0] out;           //the state variable V
    input signed [31:0] funcnt;        //the dV/dt function
    input clk, reset_n;
    input signed [31:0] InitialOut;    //the initial state variable V

    wire signed [31:0] out, vlnew ;
    reg signed [31:0] v1 ;

    always @ (posedge clk)
begin

```

```

        if (reset_n==0) //reset
            v1 <= InitialOut ; //
        else
            v1 <= vlnew ;
    end

    assign vlnew = v1 + funct ;

    assign out = v1 ;
endmodule

//// signed mult of 7.20 format 2'comp//////////
module signed_mult (out, a, b);
    output      signed  [31:0]    out;
    input signed[31:0]    a;
    input signed[31:0]    b;
    // intermediate full bit length
    wire  signed[63:0]mult_out;
    assign mult_out = a * b;
    // select bits for 7.20 fixed point
    assign out = {mult_out[63], mult_out[55:25]};
endmodule

```

Testbench:

```

`timescale 1ns/1ps

module tb();

    reg clk, reset_n;

    reg s_axis_valid;

```

```

    reg [31:0] sigma, beta, rho, dt, x0, y0, z0;

    reg m_axis_ready;

    // reg [31:0] index;

    wire signed [31:0] x_tb, y_tb, z_tb;

    wire s_axis_ready;

    wire m_axis_valid;

    wire [31:0] m_axis_data_x, m_axis_data_y, m_axis_data_z;

    lorenz DUT (
        s_axis_valid, s_axis_ready, m_axis_valid, clk, reset_n, sigma, beta, rho, dt,
        x0, y0, z0, x_tb, y_tb, z_tb,

        m_axis_data_x, m_axis_data_y, m_axis_data_z, m_axis_ready
    );

    //Initialize clocks and index

    initial begin

        // $dumpfile("dump.vcd");

        // $dumpvars(1);

        clk = 1'b0;

        reset_n = 1'b0;

        dt = {7'h0, 25'h01000};

        x0 = {7'h7f, 25'd0};

        y0 = {7'h7f, 25'd0};

        z0 = {7'd25, 25'd0};

        beta = {7'd2, 25'haaaaa};

        sigma = {7'd10, 25'd0};

```

```

rho = {7'd28, 25'd0};

s_axis_valid = 0;
m_axis_ready = 0;
    end

//Toggle the clocks
always begin
    forever clk = #10 ~clk;
end


//Intialize and drive signals
initial begin
    #20 reset_n = 1'b1;
    #20 s_axis_valid = 1;
    #30 m_axis_ready = 1;
end

endmodule

```

REFERENCES:

- Colin Sparrow, An Introduction to Lorenz equations, VOL CAS-30, No. 8, August 1983
- Chiranjeet Kumar, A Design on AMBA AXI4-Lite Interconnect Protocol, Vol. 6, Issue 6, June 2017
- [AMBA AXI and ACE Protocol Specification \](#)
- Nexys4-DDR FPGA Reference Manual, 2014
- Hardware ODE solver with HPS control, 2022
- [AXI4-Lite Interface](#)

- [Building a basic AXI Master](#)
-  Vivado 2015.2 CUSTOM IP PART I - Creating and Packaging Your IP Vivado