An 8 bit Serial Communication module Chip Design Using Synopsys tools and ASIC Design Flow Methodology

by

Anvesh Munugala

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Electrical Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

May, 2018

An 8 bit Serial Communication module Chip Design Using Synopsys tools and ASIC Design Flow Methodology

Anvesh Munugala

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

Anvesh Munugala, Student Date

Approvals:

Dr. Frank X. Li, Thesis Advisor Date

Dr. Eric W. MacDonald, Thesis Co-Advisor Date

Dr. Jalal Jalali, Committee Member Date

Dr. Salvatore A. Sanders, Dean of Graduate Studies Date

ACKNOWLEDGMENTS

I would like to express my gratitude to my thesis supervisor Dr. Frank X. Li and Dr. Eric W. MacDonald for providing exceptional support in every possible way to successfully complete my thesis. I thank Dr. Jalal Jalali for giving me an opportunity to work for Department of Electrical Engineering as Graduate Research Assistant. I express my heartfelt thanks to Ronald Valenzuela for supporting me and for providing the reference documents in implementing the ASIC methodology. I am grateful to University of Utah for providing me access to the standard cell libraries and technology files. I would like to thank MOSIS for fabricating the design as part of the educational run since I worked as teaching assistant for ASIC design course. I am blessed to have my parents and my elder brother without their support, this achievement would have never been possible. I thank them for being with me during all times and for helping me out to pursue my master's degree.

ABSTRACT

The aim of this project is to design a small FPGA chip with 0.5 μ methodology. The standard cell-based ASIC implementation of 8-bit serial communication module using the Synopsys EAD tools is presented. The design was also fabricated by MOSIS. By using Synopsys Design Compiler, IC compiler and Custom Compiler flow, simulation results are presented with the verification of logical and physical design features such as area, timing, any hold violations, Design Rule Check (DRC). This design methodology describes the RTL to GDS implementation of 8-bit serial communication module using Synopsys tools. The proposed design has 10 Input and Output ports, 54 registers. The design has passed all the Timing checks, Functional verification, Metal density requirements, Physical design verification and Layout vs Schematic. The baud rate generator which mimics a clock signal is supplied to the receiver module. The desired baud rate can be chosen from the baud rate generator. The serial communication protocol can be used for reading a sensor's bit stream data.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Integrated circuit (IC) design is a combination of numerous electronic devices on a die, made normally from silicon wafer. The electronic circuits are built in many stacks of layers such as polysilicon, aluminum and silicon dioxide predominately. The fabricated IC's cannot be reprogrammed again to change its application type, this type of IC development is called Application Specific Integrated Circuit (ASIC). The work associated toward ASIC design began in the early 80s. Chip design has seen tremendous advancement in recent decades. ASIC is specific to an application and widely used in space, military, and many commercial industries.

1.1 Different types of ASICs:

i. Standard Cell ASIC design.

ii. Full custom ASIC design.

iii. Gate Array ASIC design.

1.1.1 Standard Cell ASIC design:

In Standard Cell based ASIC design, it uses pre-designed logic cells like Inverters, OR gates, AND gates, multiplexers, NOR gates, NAND gates and flip-flops. In this method of design, the placement and routing is done by the Synopsys IC Compiler tool. The Standard Cell based ASIC design saves time and effort, reduces risk, therefore cost effective. It also allows optimizing the speed and area of each standard cell. This project primarily focuses on standard cell-based ASIC design methodology. The standard cell-based design

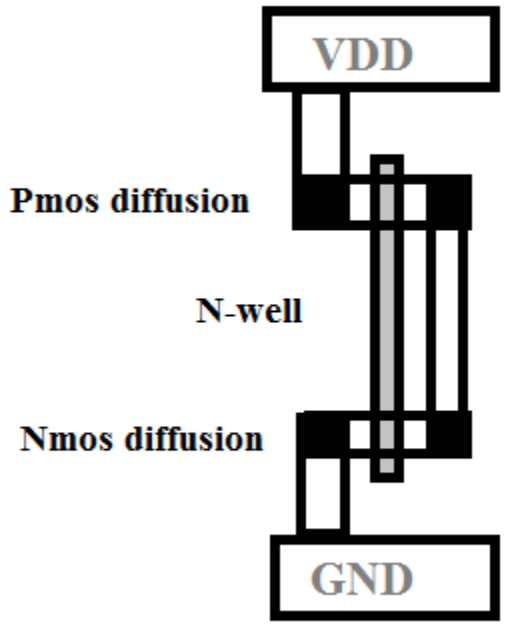


Figure 1 Standard cell (Inverter)

1.1.2 Full Custom ASIC design:

A full custom ASIC design is based on a particular application right from the beginning. It is because of their pre-specified design and functionality, the manufacturing is done with all pre-defined photolithographic layers. The full custom ASIC is basically designed as a single and specific product for any particular application. For instance, while designing a Microprocessor, full custom IC design can be adapted. Usually the design and manufacturing of full custom IC design is expensive. This process normally takes 8 weeks of time.

1.1.3 Gate Array ASIC design:

In Gate Array ASIC design, ICs are partially fabricated with rows of resistors and transistors but are not connected to each other. The connection pathways in the IC are done by adhering the top metal layers. These gate arrays are built using “basic cells”, where each cell from the array has a standard sized die that contains few transistors and resistors. Reduced time and low cost is the main advantage of Gate Array ASIC design compared to standard cell and full custom designs.

Gate Arrays are classified into three types: Structured Gate Arrays, Channeless Gate Arrays and Channeled Gate Arrays. Channeless Gate Array has no predefined space between the rows of cells. In Channeled Gate Array, the available space for interconnect between the rows of cells are fixed in terms of height. Structured Gate Array has combination of both the features of standard cell and gate array ASICs.

1.1.4 Field Programmable Gate Arrays:

Field programmed gate arrays are very reliable and cost effective option for ASIC design. The fabless job and re-programmability of the design is what makes it as best option. The FPGA configuration is generally specified using a HDL (Hardware Descriptive Language) usually in Verilog or VHDL language. These devices contain an array of programmable logic blocks that help building the logic blocks. The FPGA device used for verifying the design functionality is Xilinx Spartan 6 FPGA.

1.2 Electronic Design Automation (EDA):

EDA has been playing major role in growth and success of the semiconductor industry. Design challenges have increased for every new process generation. The growing need and demand in today's market has brought aggressive challenges and constraints like time and cost of production. All of these challenges cannot be met by just following the custom design methodology which requires a lot of resources and time consumption. Instead of relying on custom design methodology, an automated RTL to GDS design flow can be adopted which ensures that the design meets the time to market.

Synopsys and Cadence are the two major available vendors in the market who develop EDA tools. In this project, I made use of Synopsys University program (Synopsys educational package) to setup the ASIC design methodology.

MOSIS (Metal Oxide Semiconductor Implementation Service) provides chip design tools and related services that enable universities, government agencies, research institutes and businesses to prototype chips efficiently and cost-effectively. The MOSIS Instructional program gives free fabrication of integrated circuits designed by undergraduate students from an accredited university and the MOSIS run happens every year in the months of April, August and December.

CHAPTER 2: LITERATURE REVIEW

2.1 Background:

The initial ASICs made use of gate array technology. The initial successful commercial application was the gate array based design. Customization of these ICs occurred by changing the metal interconnect mask. Gate arrays had complex designs that included some thousand gates. The later version of ICs became more generalized with different base dies customized by using both polysilicon layers and metal.

The ASIC industry has traditionally been driven by the design flow which has a strict partition between the physical design and the logical design. (Static timing analysis is shown since it has largely displaced simulation-based timing verification). The fabless design is responsible for the Hardware Descriptive Language (HDL) design and synthesis, passing a netlist to the foundry for physical design. The success of this approach was based on the predictability of timing after placement and routing, that is interconnect loading has relatively low effect on the timing.

2.2 Serial Data Communication:

Serial data communication is a process of sending data one bit at a time through a communication channel or data bus between any two digital devices. Serial communication is generally used for computer networks, control system and to read any sensor's data. Serial communication module mainly contains transmitter, receiver and baud rate generator. This type of communication does not require any system clock for

transmitting the data bits. Baud Rate generator produces the clock for transmission of data bits. The standard baud rates are 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 data bits per second. Desired baud rate can be achieved by using a clock division factor from the system clock. The speed of communication is directly proportional to the baud rate. Higher the baud rate, faster the speed of communication whereas the division factor is inversely proportional to the baud rate.

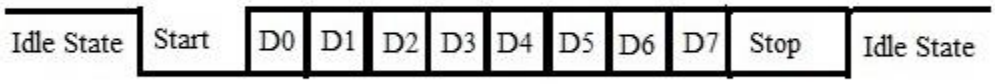


Figure 2 Serial data communication

Data frame contains 8-bit data, 1 start bit and 1 stop bit. The early serial communication module was not dependent on speed and distance of transmission. As the growing needs of today's life, speed and error free communication is very much necessary. Since the technology has changed dramatically and wireless communication has paved the path for faster transmission and yet reliable, serial data communication cannot meet the present trend. So to meet the present market, a simplest, reliable and compact design can satisfy the cost of production.

2.3 Traditional designs:

The serial communication modules usually had 8 input ports to transmit a single byte and 8 output ports to receive. The past designs did not have any state machine controllers that send the 8-bit data accurately. This has led to huge communication errors. In the current trend, the dependency on reliability and area is an important factor. Therefore, a faster communication and yet simple and reliable designs is in need today.

2.4 Scope of the proposed design:

The 0.5 µ standard cell technology is used for implementing the ASIC design. This technology implementation can be used for space applications, Analog to Digital converter, and Mixed signal applications. The proposed design has improvements in reduction of Input and output ports with much simpler design that uses 54 registers/Flipflops. The total cell area is 148176 sq µ and the clock arrival time had slack of 3.91 ns before the required data time. In this proposed thesis, the main system clock is 12MHZ and the time to transfer each bit is 6.64 us.

2.5 Applications of the UART module:

The serial communication protocol has many applications which is mainly used for receiving the bit stream from the sensor and send it to the processor at a desired baud rate frequency. The baud rate can be selected from the list of available frequencies depending on the system main clock.

Some applications of the serial communication module are listed below:

a) Reading the digital bit stream from Analog to Digital converter and storing the measured values.

b) Serves as a serial communication buffer between master (microcontroller) and slave (sensor) devices.

c) Bluetooth modules and wireless communication devices

d) GPS navigation systems

e) Computer bus lines

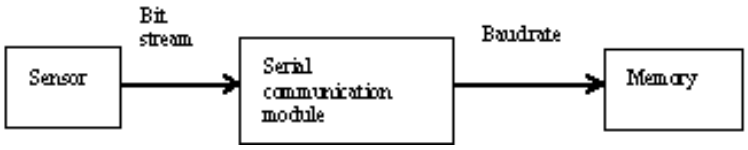


Figure 3 Application of serial communication module

CHAPTER 3: HDL DESIGN

The first goal of ASIC development is to come up with the following specifications for any particular design.

- Functionality
- Logic Design (HDL Coding)
- Performance analysis
- Physical Design
- Fabrication technology and design techniques

After we have the above specifications for a particular application, next step is to follow a design methodology in implementing the design for tape-out.

3.1 Specification and HDL coding:

The architecture specifications of the design are about the partitioning and functionality of the IC into distinct blocks. The electrical specifications describe the relation between the parts of design in terms of timing information. The next step involves the implementation of these pre-defined specifications. To overcome this problem and for an efficient outcome, HDL (hardware description languages) were developed. The functionality of the design is done using the HDL code. Verilog/VHDL are the two HDLs which are in use today. The both programming languages perform the similar function but vary in syntax, each have their own disadvantages and advantages.

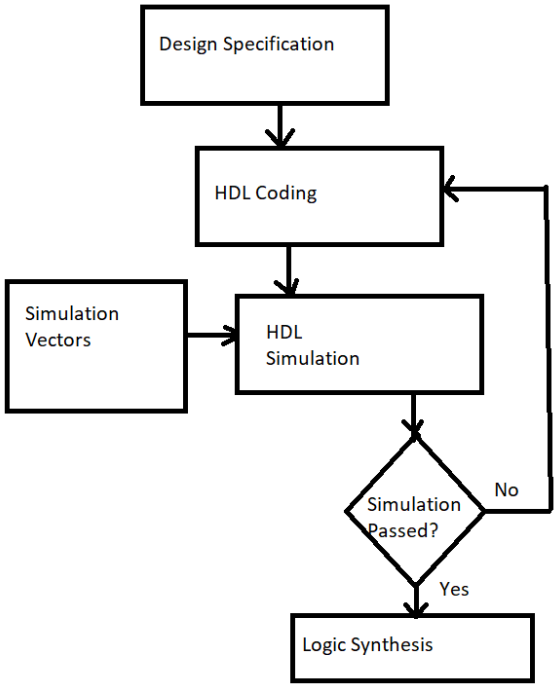


Figure 4 HDL design flow

The design can be divided into three levels: Behavioral, Register Transfer Level (RTL) and Structural. Synthesizing a design is a must and it starts with define the timing constraints for each block of the design.

3.2 Understanding HDL design flow for FPGAs:

ASIC designs are first implemented on FPGAs using HDL programming language. However, the synthesis flow for FPGA is little different from the ASIC design flow which has area, timing and power analysis. FPGA implementation and ASIC implementation is similar whereas in FPGA implementation, all the process is fabless and reprogrammable which best suits for optimization of the design.

The design flow includes these following steps:

a) Create a HDL file either in Verilog/VHDL

b) Create a new FPGA project and add your HDL file

c) View Register Transfer Level (RTL) schematic

d) Synthesize the design

e) Perform the functional simulation of the design

f) Create a Floorplan for the design, which is optional

g) Place and route of the design

h) Perform timing simulation of the design

The design flow for FPGA implementation is illustrated in Figure 5. FPGA implementation is the most reliable and fabless process to verify the functionality of our design. A successful implementation of functioning design is a good sign of progress and to verify the design functionality.

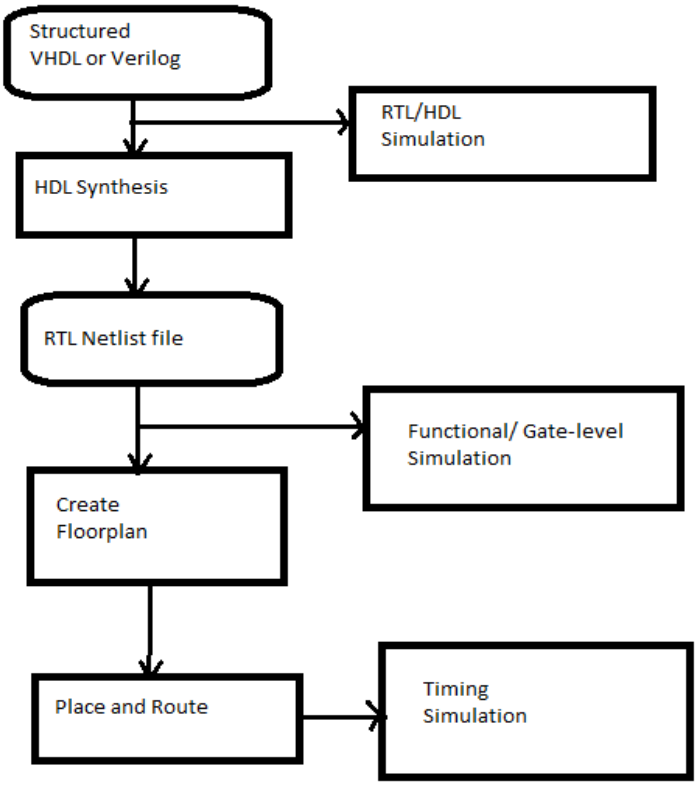


Figure 5 FPGA synthesis design flow

3.3 Logic design:

The design of serial communication or Universal Asynchronous Receiver Transmitter (UART) module was done in Verilog code. The proposed design has 10 Input and Output ports, 54 registers and flip-flops. The design contains transmitter, receiver and baud rate generator. Figure 6 shows the RTL (Register Transistor Level) view of serial communication module.

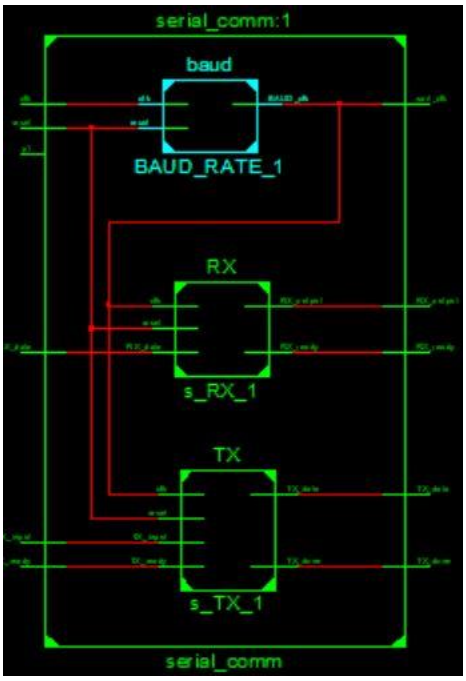


Figure 6 Serial communication module RTL view

As shown in figure 6, the serial communication module consists of following inputs and output ports:

Inputs:

a. clk (system clock)

b. reset

c. TX_input (1 bit)

d. RX_data (1 bit)

e. TX_ready (1 bit)

Outputs:

f. TX_data (1 bit)

g. TX_done

h. RX_output (1 bit)

i. RX_ready (1 bit)

j. uart_clk (baud generator output)

The RX_output, RX_ready, uart_clk wire connects with the input and output ports. The transmission rate is lowered from the 12 MHZ system clock speed to baud rate speed which is 9600 bits per second. The reset port of the uart module will clear all the input and output values on every negative edge of reset signal. The uart_clk port will give the output from the baud generator which will be used for transmission and reception of data.

3.3.1 Transmitter module:

The inputs of the transmitter are TX_input, clk (baud clock), reset and the outputs of the transmitter are TX_data and TX_done. The registers of the transmitter module are as follows: Load_SR, TX_ready, TX_done, Delay_High, Input_data (8 bit), DelayEnable, ShiftEnable, TX_DS (data select), TX_data, nextstate, state, count, rst_count, enable_count, TX_shiftReg [7:0].

The transmitter of uart module is responsible for sending the 8 data bits one at a time. The Input_data [7:0] which is an 8 bit register, loads the data bits one bit per clock cycle through TX_input port. This Input_data [7:0] register holds the first 8 bits which needs to be transmitted.

The initial change in transition of reset signal sets the value in all registers to zero. The reset clears the previous values in all the registers from the next edge of reset signal. As the TX_ready goes high, the transmission begins.

The proposed transmitter module consists of a state machine, a counter and a shift register. There are 4 states in this state modal.

At beginning of the state machine, all the register values are set to zero. The first state, tx_state0 is idle state that waits for the TX_ready. After the TX_ready is active, state changes to tx_state1.

Tx_state1 sends a start bit and sets Load_SR to 1. The tx_state1 is put on hold for 8 clock pulses to load the input data bit by bit. At the end of 8 clock pulses, state changes to tx_state2.

Tx_state2 is to load the shift register with 8 bit input data. As the shiftEnable is pulled high, Input_data [0:7] value is assigned to TX_shiftReg [7:0]. Now, till the count is equal to 8 bit data length, the counter keeps incrementing for every clock pulse. After the count is equal to 8-bit data length, the state is changed to Tx_state3. In Tx_state2, as the TX_DS is set to tx_shiftRegbit, TX_shiftReg[0] value is sent out to the receiver through TX_data one bit at a time.

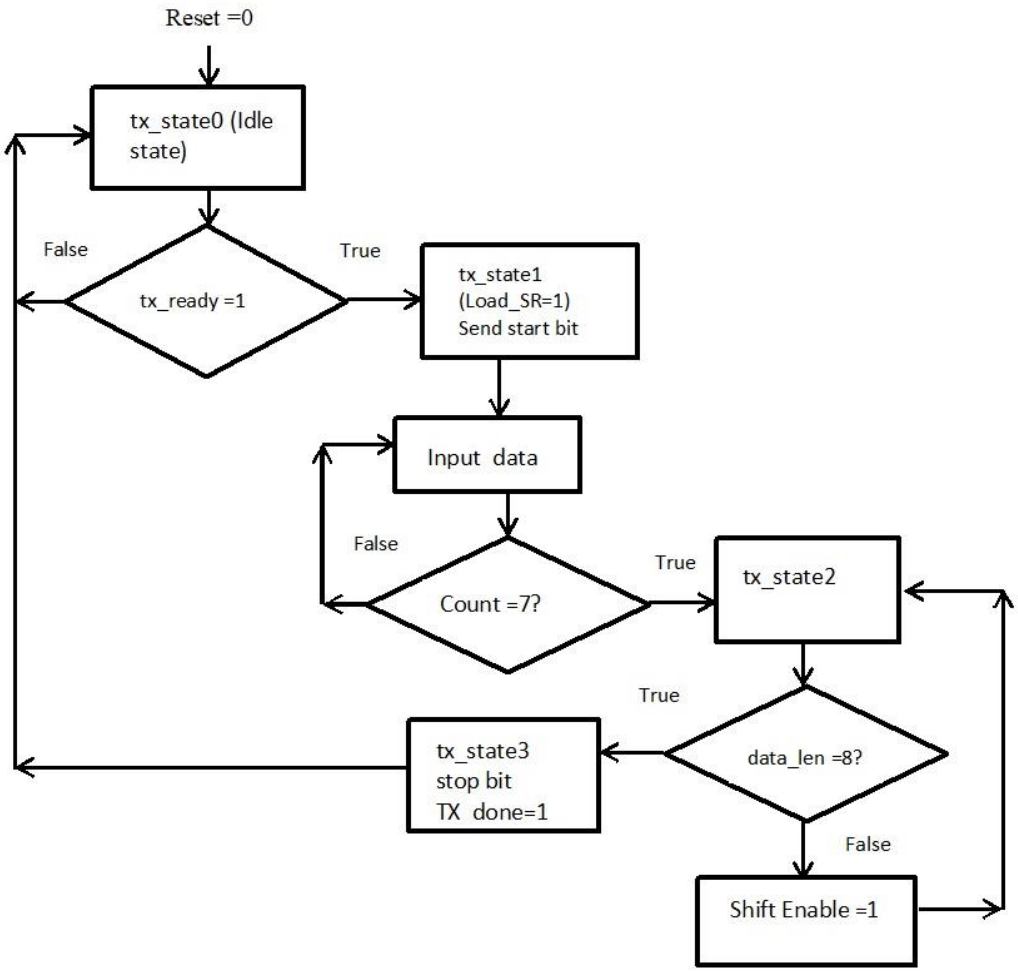


Figure 7 Transmitter state machine

Tx_state3 is the final state in the state machine. In tx_state3, a stop bit is sent to the receiver indicating that the transmission of the 8 bit data is done.

TX_done changes to high indicating that the next 8 bit data can be sent. Now the next state is tx_state0.

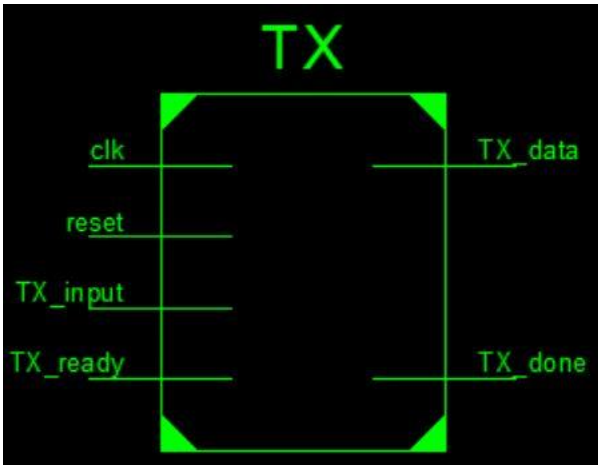


Figure 8 Transmitter module top view

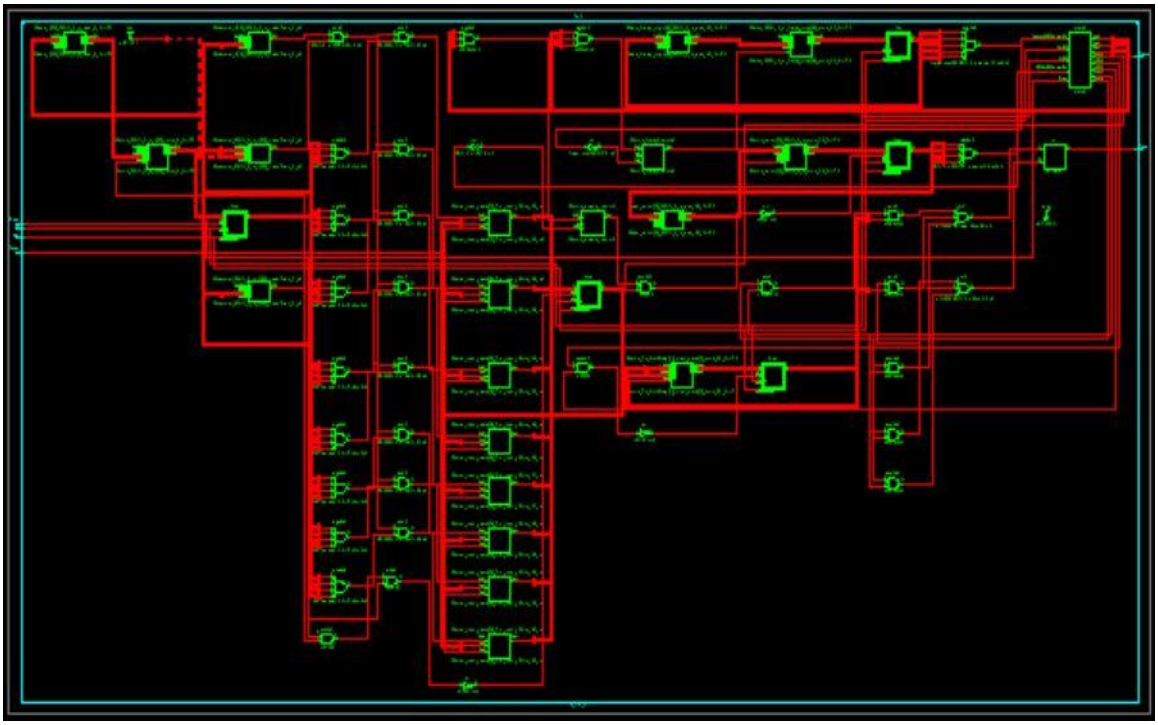


Figure 9 Transmitter module RTL view

3.3.2 Receiver module:

The inputs of the receiver are RX_data, clk(sensor clock), reset and the outputs of the receiver are RX_output and RX_ready. The registers of the receiver module are as follows: RX_ready, Delay_High, parse_data (8 bit), Delay_reset, ShiftEnable, RX_data, ack, nextstate, state, rxd_Cntr, rst_count, enable_count.

The proposed receiver module consists of a state machine, two counters and a shift register. There are 4 states in this state modal.

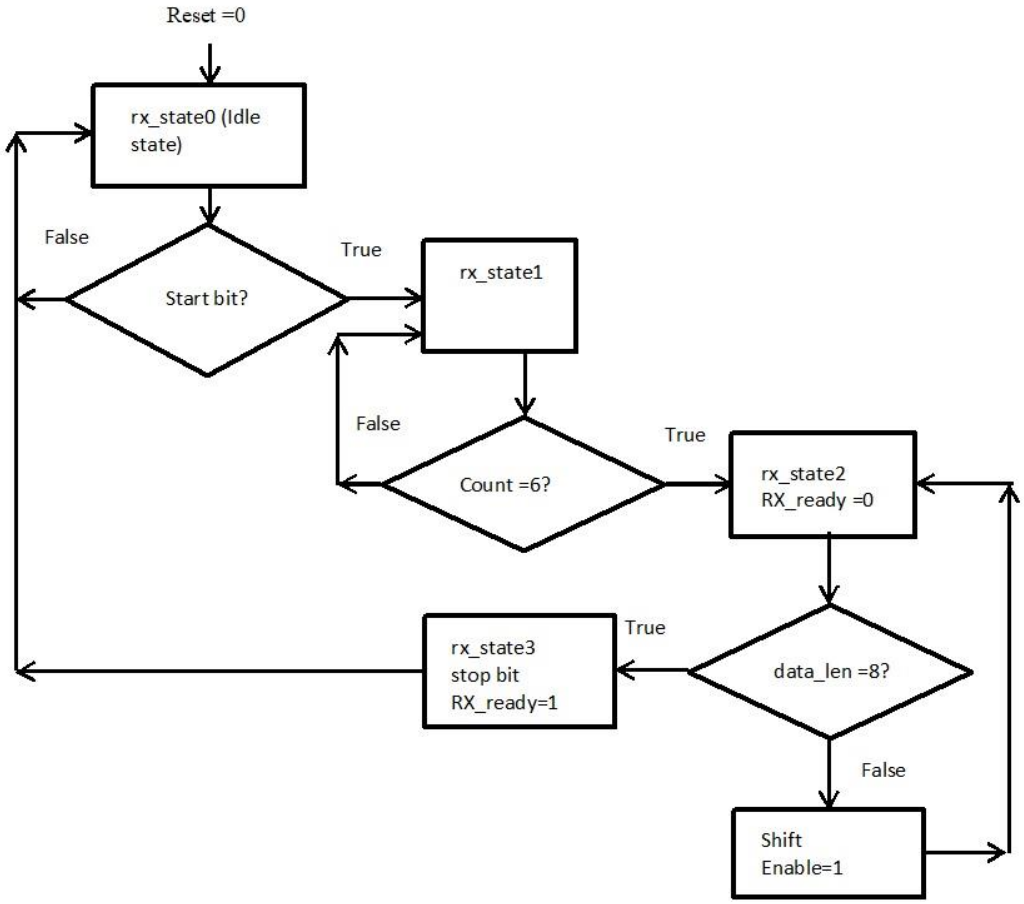


Figure 10 Receiver State Machine

Rx_state0 is idle state and waits for a start bit. After the start bit received is assigned to ack register. At this point RX_ready is high indicating the transmitter to send the bytes. After the start bit is acknowledged by the receiver, state changes to rx_state1.

The rx_state1 waits for 8 clock pulses to allow the transmitter to load the input data bit by bit. After the 8 clock pulses, RX_ready is dropped low indicating the transmitter to stop loading the input data. The state changes to rx_state2.

The rx_state2 stays in the same state till the count value is equal to the data length (8). After the count value is equal to the data length, state changes to rx_state3.

While the counter is incrementing, 'shiftEnable' is high. This enables the parse_data[7:0] register to load the received byte, one bit at a time. Parse_data [7:0] stores the received byte. The received byte is sent out through RX_output, one bit at a time.

Rx_state3 pulls the RX_ready to high, indicating the transmitter to send the next byte and state changes to rx_state0.

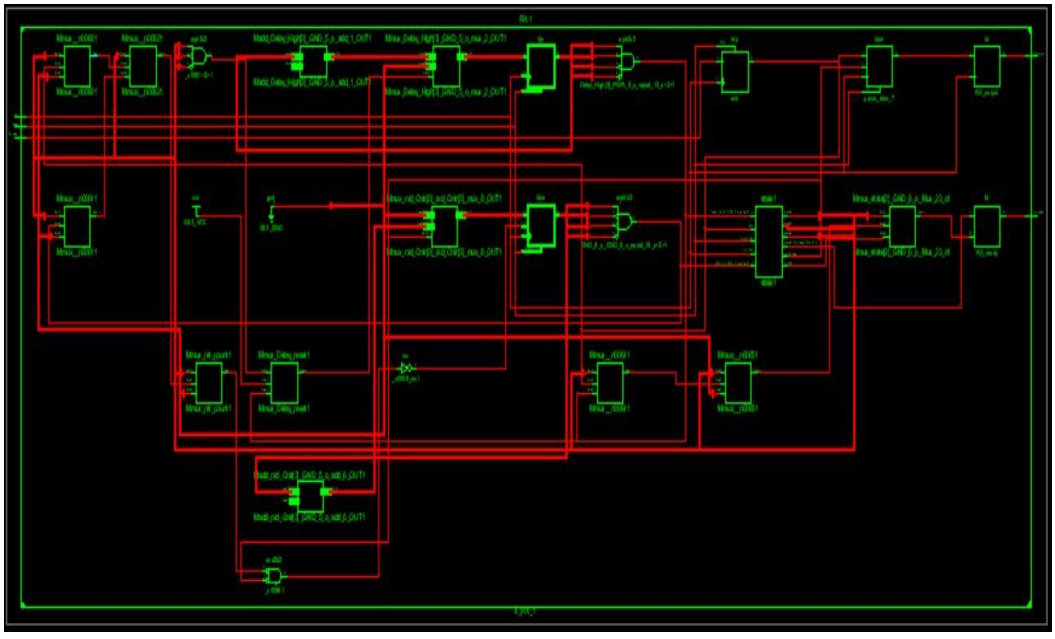


Figure 11 Receiver RTL view

3.3.3 Baud generator:

The baud generator module is responsible to generate a pulse wave at desired baud rate. This module consists of system clock, reset as inputs, baud_clk as output, cycles [8:0] and baud_clk as the registers. The value of cycles is (system_clock)/(baudrate*16*2). The desired baud rate can be chosen depending on the receiver or transmitter device clock speed. If the transmitted data is at 9600 baud rate, the receiver must also receive at 9600 baud rate to avoid communication errors.

The system_clock depends on the clock oscillator that we use and 9600 is the baud rate. Considering the system_clock is 12 MHZ, for every 39 cycles the pulsed wave is generated. So for every 78 cycles that is 6.64 us, a single clock pulse which has '1' and '0' states is generated out of this baud generator through baud_clk. This baud clock is given as clock input to the TX and RX module.

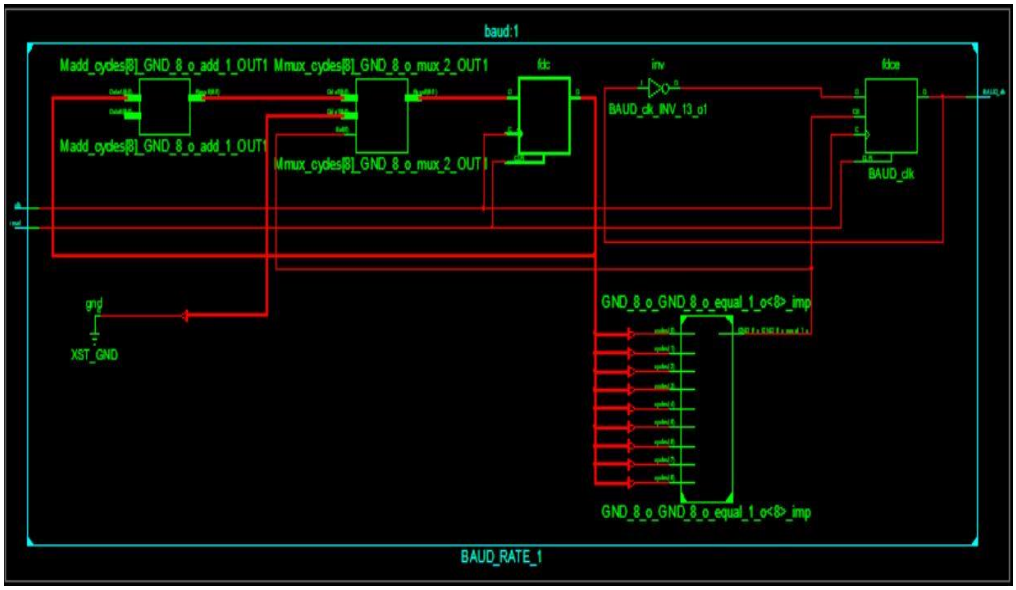


Figure 12 Baud generator RTL view

CHAPTER 4: HDL SIMULATION RESULTS DISCUSSION

This chapter discusses about the HDL simulation results and gives the Xilinx ISE synthesis report of the schematic.

4.1 HDL Synthesis:

The main job of synthesizer is to convert HDL program into a gate-level netlist such as AND, OR, Inverter, MUX. By default, Xilinx ISE uses XST (Xilinx Synthesis Technology) synthesizer. There are other synthesizers like Quartus, Vivado that can be used for respective FPGA vendor boards.

4.2 HDL Synthesis report:

TX Summary:

3 Adder/ Subtractors, 57 D-type flip-flops, 1 Latch, 3 Comparators,

14 Multiplexers,1 Finite State Machine.

RX Summary:

2 Adder/Subtractors, 18 D-type flip-flops, 1 Latch, 10 Multiplexers, 1 Finite State Machine.

Baud Summary:

1 Adder/Subtractor,10 D-type flip-flops, 1 Multiplexer.

Table 1 HDL Synthesis Report

Macro Statistics	**No.**
# Adders/Subtractors	6
32-bit adder	1
4-bit adder	3
4-bit adder	1
9-bit adder	1
# Registers	12
1-bit register	3
32-bit register	1
4-bit register	3
5-bit register	1
8-bit register	3
9-bit register	1
# Latches	2
1-bit latch	2
# Comparators	3
32-bit comparator greater	1
32-bit comparator less/equal	2
# Multiplexers	25
1-bit 2-to-1 multiplexer	18
32-bit 2-to-1 multiplexer	1
4-bit 2-to-1 multiplexer	3
5-bit 2-to-1 multiplexer	1
8-bit 2-to-1 multiplexer	1
9-bit 2-to-1 multiplexer	1
# FSMs	2
Flip-Flops	54

4.3 Transmitter module simulation:

The transmitter is responsible to send 8 data bits, one bit at a time. The proposed design has only one input port (TX_input) to send the transmit data. Input data is a register which loads the data, bit by bit. As shown in figure 13, the input data register loads the 8 data bits and stops to load after the first byte is ready. TX_shiftReg is a shift register that stores the byte to transmit at later point. TX_DS is a data select line which has start bit '00' loaded initially. At this point, state is tx_state1.

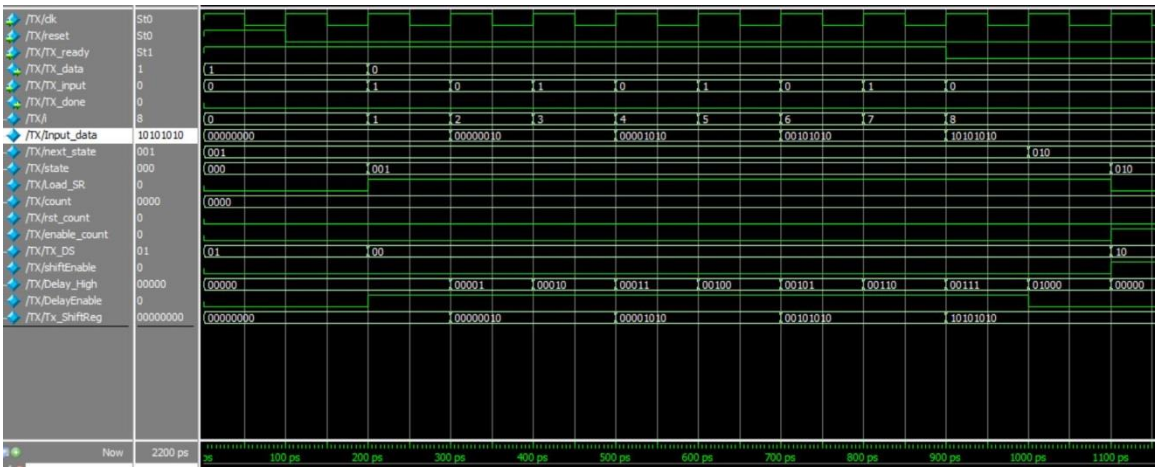


Figure 13 Load the input data

After the first byte is ready, TX_shiftReg sends the data out, one bit at a time through TX_data output port. During this process, the shift register is active and TX_DS changes to tx_ShiftRegbit '10'. As shown in figure 14, the TX_data sends the first byte of input data for every clock pulse. TX_shiftReg register is loaded with '0' at the MSB to clear the data after it transmits the input data.

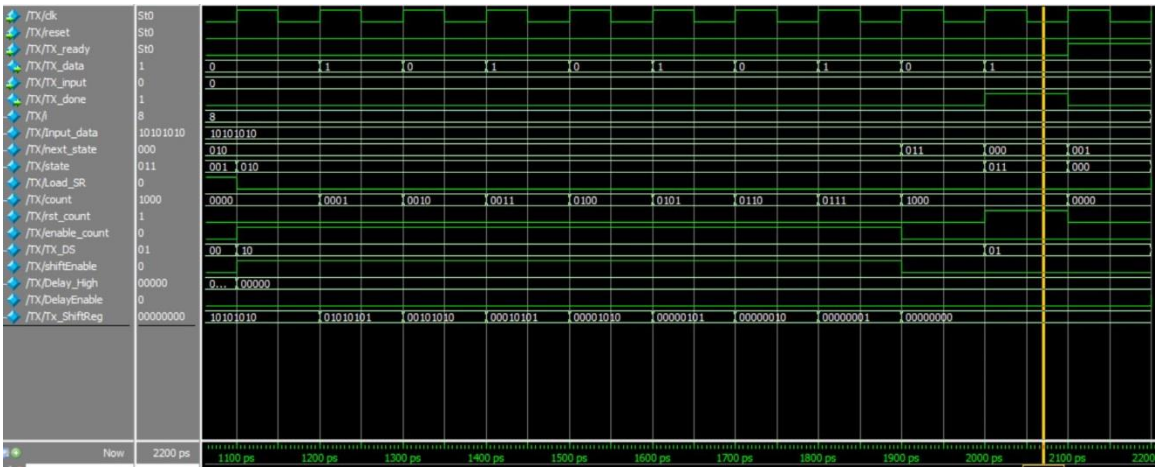


Figure 14 Transmit data

After the first byte is sent to the receiver through TX_data, the TX_done bit changes to high indicating that transmission is done. As shown in figure 14, the TX_ready is pulled high to load the next byte of input data to be sent out.

4.4 Receiver module simulation:

The receiver is responsible to receive 8 data bits, one bit at a time. The proposed design has only one output port (RX_output) to send out the received data. Parse data is a register which stores the data, bit by bit. The receiver waits till 8 clock pulses after receiving a start bit, since the transmitter prepares to send the first byte of information. At this point, the RX_ready is high.

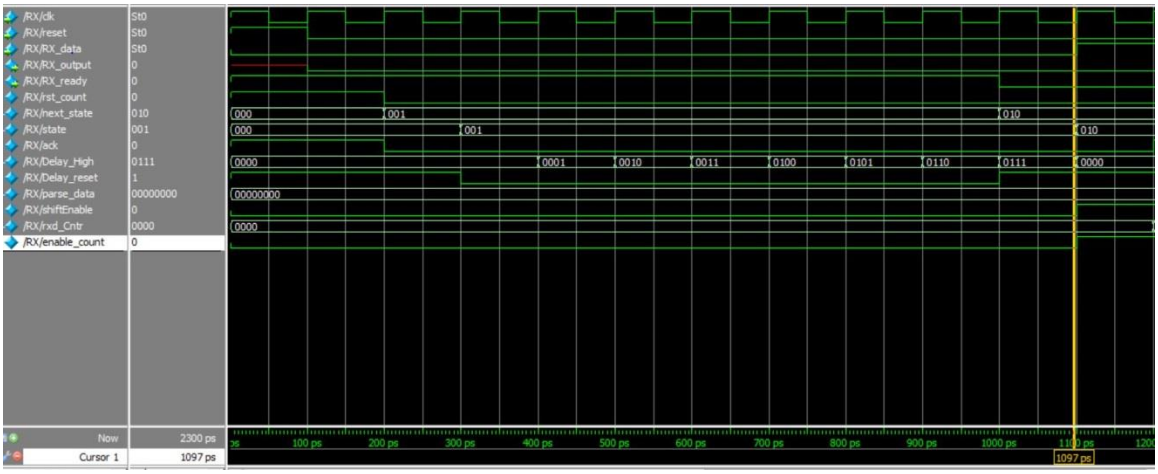


Figure 15 Receiver waiting for transmitter

After the 8 clock pulses, the receiver gets the transmitted byte one bit at a time through RX_data. This data is stored in parse_data register and the MSB (parse_data[7]) bit is sent out through the RX_output port as shown in figure 16. At this point RX_ready is low indicating transmitter to stop loading the input data.

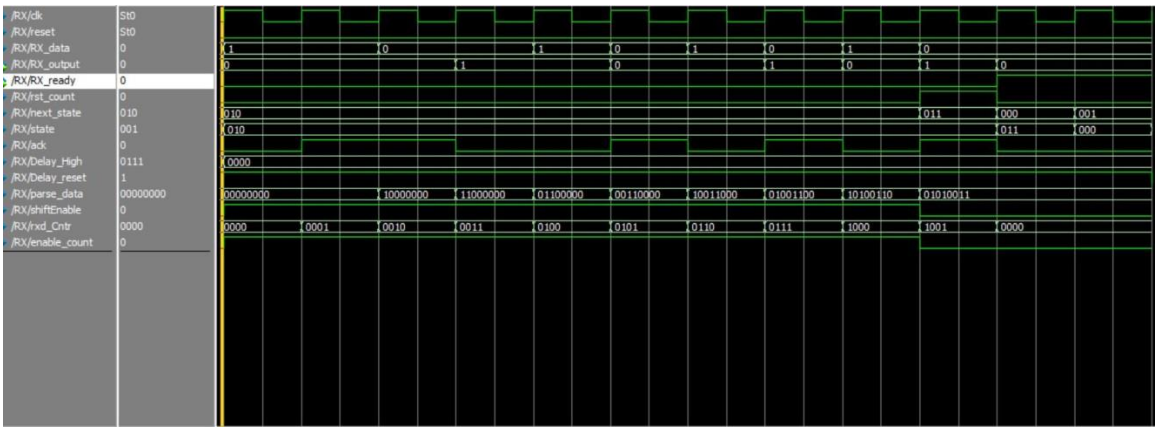


Figure 16 Receiver output data

After the receiver has sent the received byte, RX_ready goes high indicating the transmitter to send the next byte of the data.

4.5 Baud generator simulation:

The baud generator is an asynchronous clock for the UART module. This particular module generates the clock pulses at 9600 baud rate. As shown in figure 17, the baud generator generates a pulsed wave at every 39 cycles since the system clock is 12MHZ.

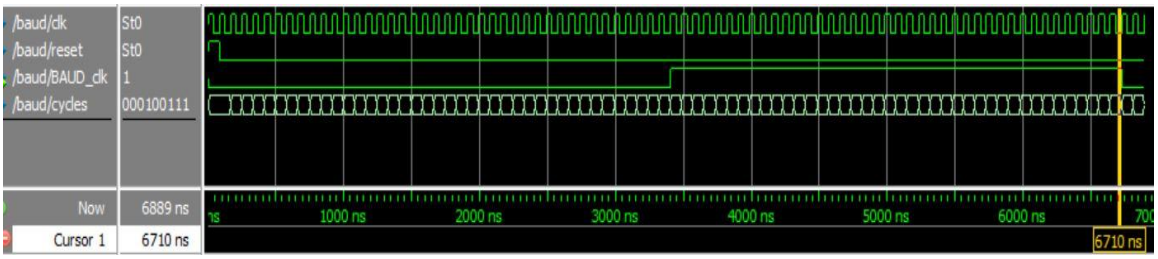


Figure 17 Baud generator simulation

4.6 Serial communication module simulation:

As shown in figure 18, the clk is the system clock and uart_clk is the clock output from the baud generator module. TX_data is an output port that sends the TX_input data. RX_data receives the transmitted data from TX_data and sends the received byte one bit at a time through RX_output which is an output port.

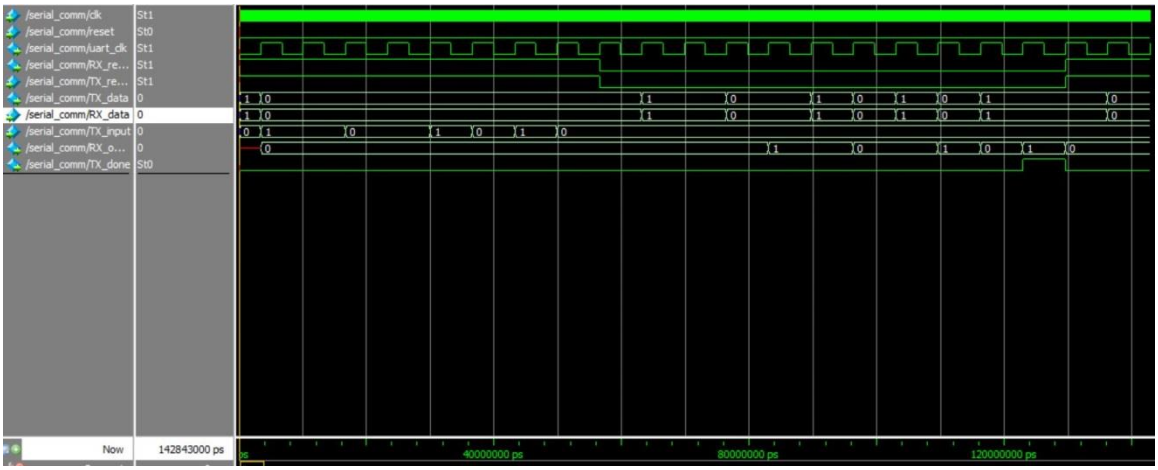


Figure 18 Serial communication module simulation

CHAPTER 5: ASIC IMPLEMENTATION

5.1 Technology and Library files:

The ASIC design methodology that is adapted for building 0.5 µ ASIC technology is described in this chapter. I have used the UTAH standard cell library provided by University of UTAH. The dc_shell (Synopsys Design Compiler) and icc_shell (Synopsys IC Compiler) script files that were utilized to perform the ASIC implementation in this chapter are provided as a reference by the University of Utah.

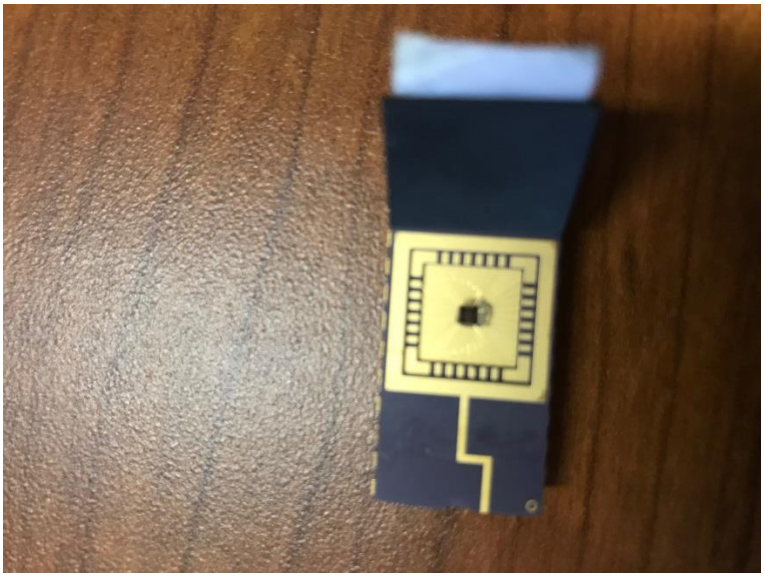


Figure 19 MOSIS fabricated chip

To assess the impact of layout related issues on power and clock features, we need to understand the different design steps that shape the layout and optimize its performance. Design planning was not necessary in earlier generations of process technologies due to two reasons: (1) the size of the design in terms of the number of gates

was reasonable, and (2) the performance targets were modest. In current and future process technologies, two forces made design planning a necessity: the exponential growth of the number of transistors that can be packed on a die, and the aggressive and tight design constraints and market forces.

5.2 ASIC design flow

The ASIC design flow involves in the following steps:

1) Functional specification of the design
2) HDL coding in VHDL/Verilog
3) RTL/behavioral or functional simulation of the HDL using Modelsim
4) FPGA synthesis using Xilinx ISE
5) Check synthesis reports
6) FPGA implementation
7) Synthesis using Synopsys Design Compiler (dc_shell)
8) Check Timing report
9) Check Area report
10) Check power report
11) Check netlist file
12) ASIC implementation using Synopsys IC compiler (icc_shell)
13) Floorplan
14) Power and I/O fill
15) Placement of cells
16) Clock tree insertion
17) Perform timing analysis

18) Routing of cells

19) Perform timing analysis

20) Check for any hold violations in quality of results (qor) report

21) Dummy metal filling (Poly, Metal 1, 2 and 3) to fill the empty spaces

22) Formal verification (Layout VS Schematic)

23) Error browser for any short errors

24) Perform Design Rule Check (DRC) using Synopsys custom compiler

25) Fix any DRC errors within the (260, 260, 1240, 1240) bounding area

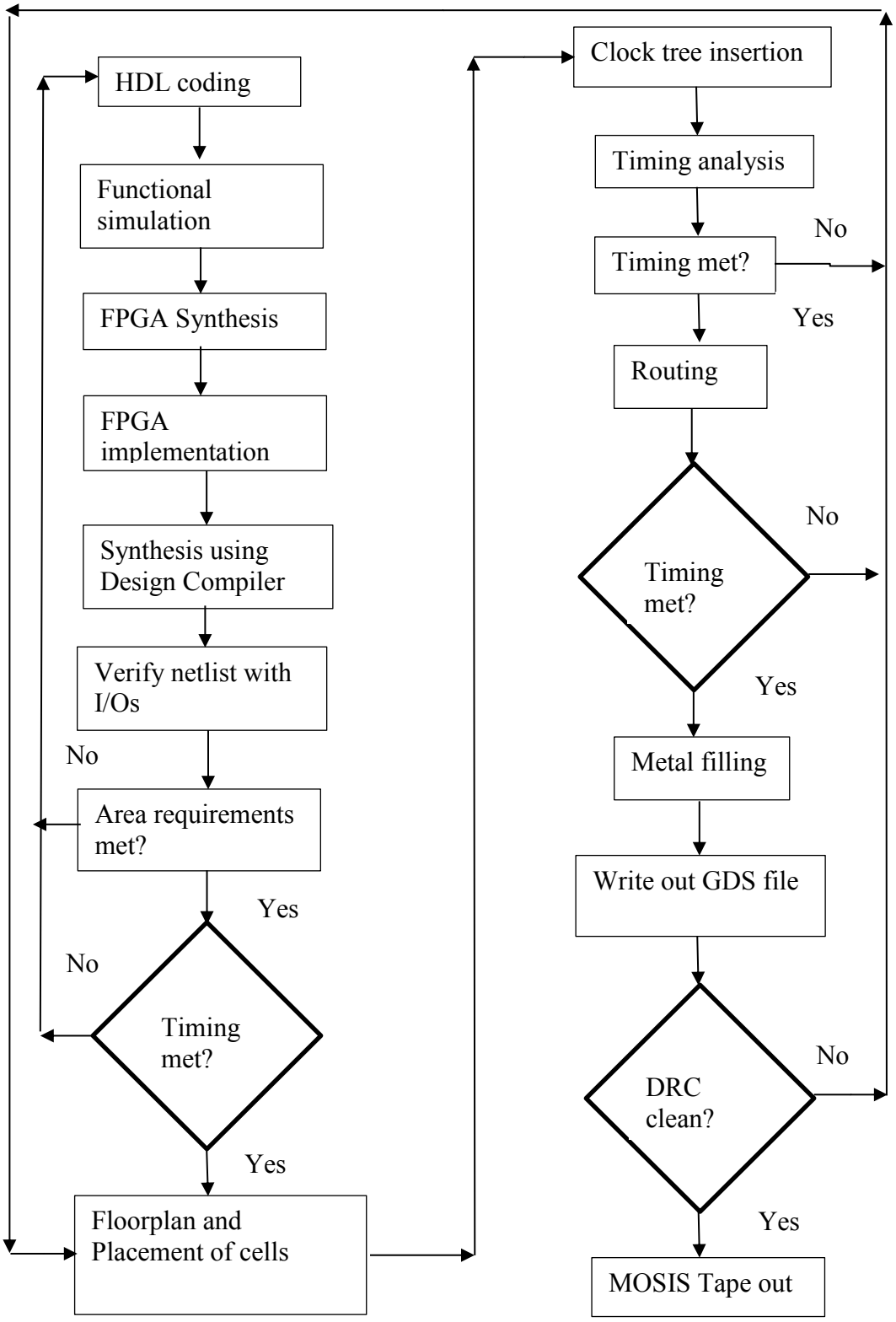
26) Export the GDS file

27) Check to see if metal densities are met with the minimum requirements

28) If any metal density is not met, add metal manually

29) Submit the design to MOSIS for fabrication

Figure 20 ASIC design flow



5.3 Synthesis using Synopsys Design Compiler tool:

This section shows the synthesis reports obtained from the Design Compiler tool. After running the dc_shell with the serial communication design, the following reports can be generated.

```
****************************************
Report : area
Design : serial_comm_with_io
Version: M-2016.12-SP5-3
Date   : Sun Apr 22 20:35:38 2018
****************************************

Library(s) Used:

    c5n_utah_std_v5_t27 (File:
/home/amunugala@ysu.local/ON.PDK/samples/logic/c5n_utah_std_v5
_t27.db)
    io (File:
/home/amunugala@ysu.local/ON.PDK/samples/logic/io.db)

Number of ports:                           11
Number of nets:                           286
Number of cells:                          281
Number of combinational cells:            214
Number of sequential cells:                57
Number of macros/black boxes:              10
Number of buf/inv:                         52
Number of references:                      21

Combinational area:              70704.000000
Buf/Inv area:                    11304.000000
Noncombinational area:           77472.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                148176.000000
```

Figure 21 Area report

Figure 21 shows the area report of the serial communication module. As shown in figure, the total cell area is 148176 square µ.

Figure 22 shows the timing report of the serial communication module. As shown in the figure, the clock timing is verified against the timing constraints. The start point is main_clock and the end point is uart_clk. There should be (MET) beside the slack in order to say that the timing is met.

```
  Point                                                   Incr       Path
  ------------------------------------------------------------------------
------------
  clock main_clock (rise edge)                             0.00     0.00
  clock network delay (ideal)                              0.00       0.00
  I_serial_comm/BAUD_RATE_1/BAUD_clk_reg/CLK (DCX1)0.00    0.00       0.00 r
  I_serial_comm/BAUD_RATE_1/BAUD_clk_reg/Q (DCX1)1.15      1.15       1.15 f
  U330/Y (INVX1)                                           0.32       1.47 r
  U345/Y (INVX2)                                           0.67       2.14 f
  u_uart_clk_pad/pad (pad_bidirhe)                         1.13       3.27 f
  uart_clk (out)                                           0.00       3.27 f
  data arrival time                                                   3.27

  clock main_clock (rise edge)                            10.00      10.00
  clock network delay (ideal)                              0.00      10.00
  output external delay                                   -3.00       7.00
  data required time                                                  7.00
  ------------------------------------------------------------------------
------------
  data required time                                                    7.00
  data arrival time                                                    -3.27
  ------------------------------------------------------------------------
------------
  slack (MET)                                                         3.73
```

Figure 22 Timing report

```
module serial_comm_with_io ( clk, reset, TX_ready, TX_input,
RX_data, Port11,
        uart_clk, TX_data, TX_done, RX_output, RX_ready );
  input clk, reset, TX_ready, TX_input, RX_data, Port11;
  output uart_clk, TX_data, TX_done, RX_output, RX_ready;
  wire   n2, n4, n6, n8, n10, n14, n16, n18, n20, n22,
         \I_serial_comm/s_TX_1/next_state[2] ,
\I_serial_comm/s_TX_1/N196 ,
         \I_serial_comm/s_TX_1/N195 , \I_serial_comm/s_TX_1/N194
,
         \I_serial_comm/s_TX_1/N193 , \I_serial_comm/s_TX_1/N192
,
         \I_serial_comm/s_RX_1/N55 , \I_serial_comm/s_RX_
1/next_state[2] ,
         \I_serial_comm/s_RX_1/N18 , \I_serial_comm/s_RX_1/N17 ,
         \I_serial_comm/s_RX_1/N16 , \I_serial_comm/s_RX_1/N15 ,
         \I_serial_comm/s_RX_1/parse_data[7] ,
\I_serial_comm/BAUD_RATE_1/N22 ,
         \I_serial_comm/BAUD_RATE_1/N21 ,
\I_serial_comm/BAUD_RATE_1/N20 ,
         \I_serial_comm/BAUD_RATE_1/N19 ,
\I_serial_comm/BAUD_RATE_1/N18 ,
         \I_serial_comm/BAUD_RATE_1/N17 ,
\I_serial_comm/BAUD_RATE_1/N16 ,
         \I_serial_comm/BAUD_RATE_1/N15 ,
\I_serial_comm/BAUD_RATE_1/N14 , n24,
```

Figure 23 Netlist file

5.4 ASIC implementation using Synopsys IC Compiler tool:

IC compiler tool also called icc_shell is responsible for floorplan, placement of cells, clock insertion, routing, cell filling, metal fill, writing out GDS file.

Figure 24 shows the floorplan. Floorplan is a process to choose different layers on the die. Some of the layers are contact, via, glass, metal 1, metal 2, metal 3, poly, active. Length of the die used is 1500 μm. The horizontal lines in figure 23 are the different layers. Figure 25 shows the I/O pads. I/O pads are the square blocks on the four sides of the layout. The vertical straight lines are VDD and GND.

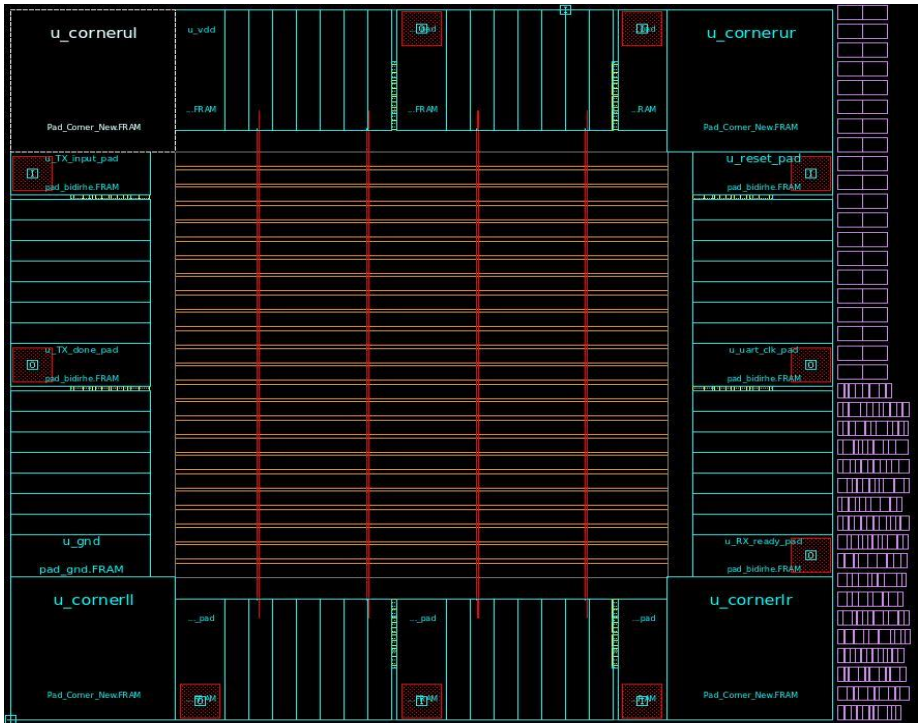


Figure 24 Floorplan

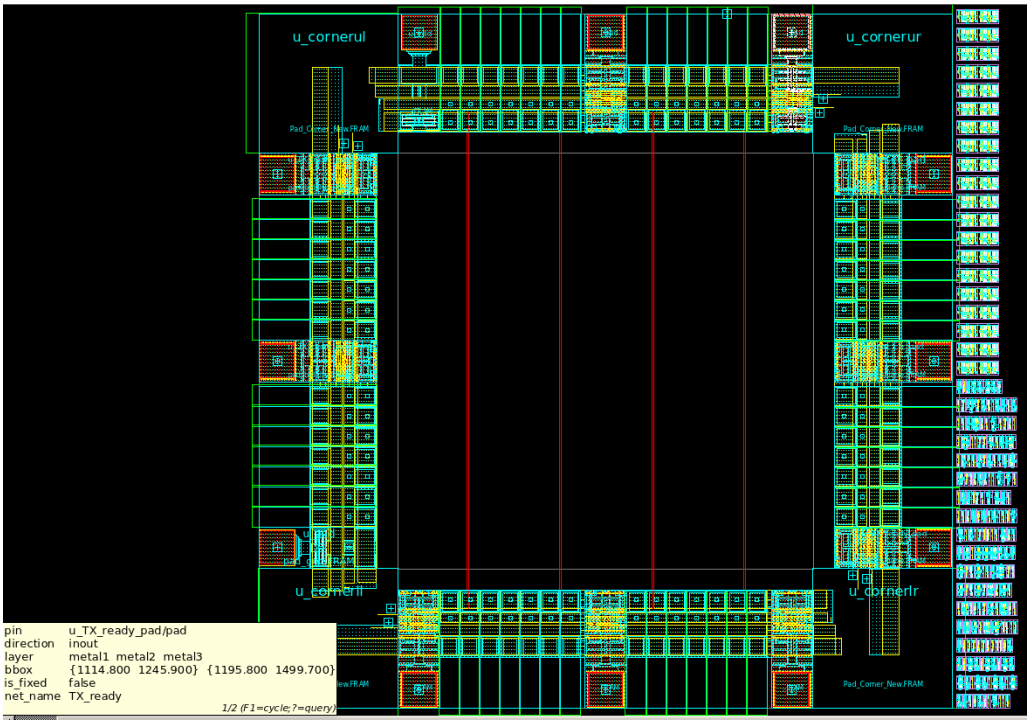


Figure 25 Pad fill

The placement of the cells is shown in figure 26. The IC compiler is responsible for the placement of cells. The cells are placed in the center part of the die inside the bounding box (260, 260, 1240, 1240).

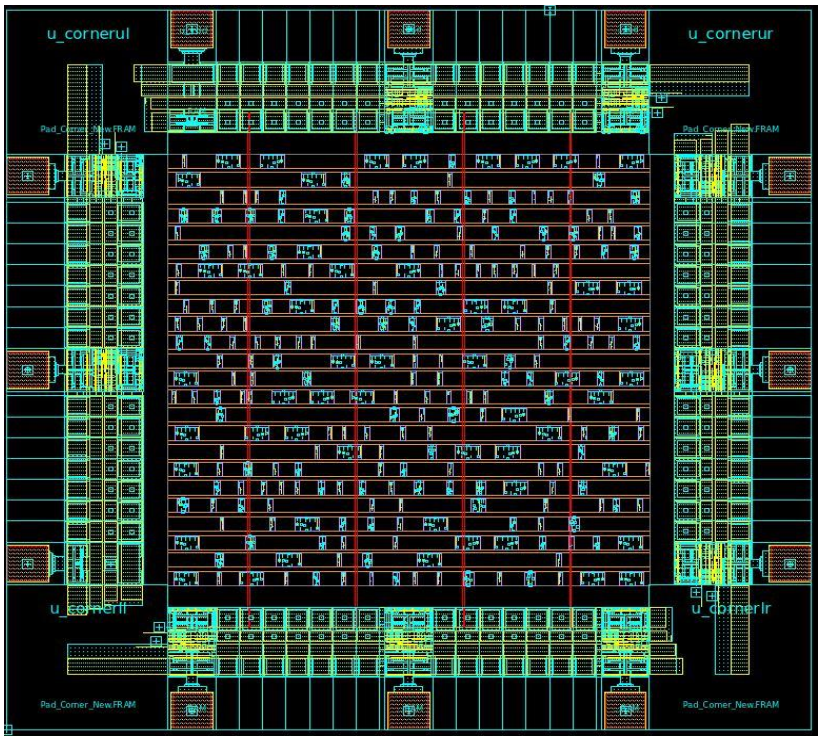


Figure 26 Placement of cells

After placement of cells, a clock tree must be inserted to supply the clock signal to all the modules. Clock tree is responsible to supply the clock to all the modules. Clock tree is shown in figure 27. After the clock tree insertion, the timing check is performed to make sure the timing is met. The clock tree report is shown in figure 28. There should be MET beside the slack to say that the timing is met.

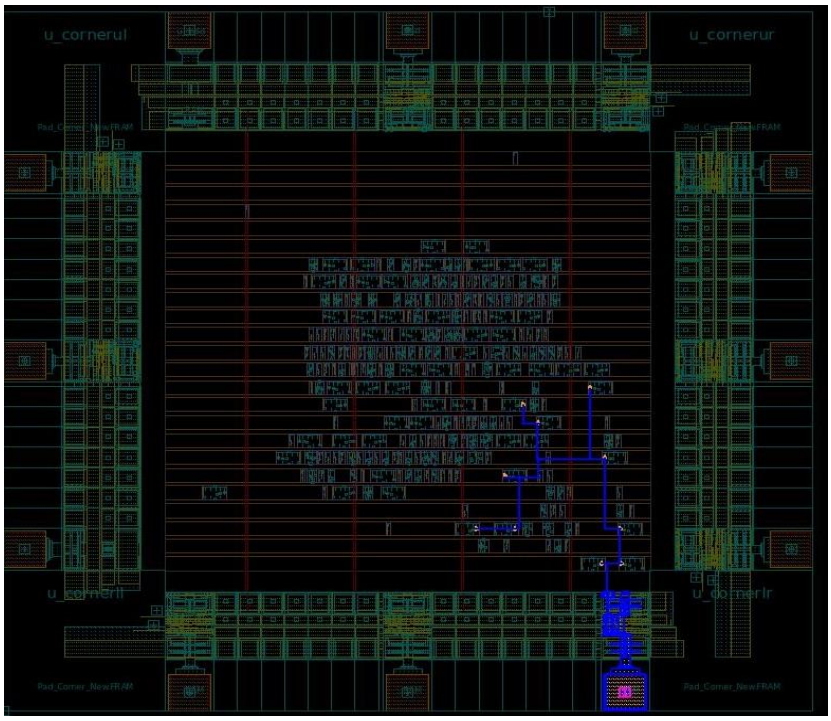


Figure 27 Clock-tree insertion

Point	Incr	Path
clock main_clock (rise edge)	0.00	0.00
clock network delay (propagated)	0.37	0.37
I_serial_comm/BAUD_RATE_1/BAUD_clk_reg/CLK (DCX1)	0.00	0.37 r
I_serial_comm/BAUD_RATE_1/BAUD_clk_reg/Q (DCX1)	0.91	1.28 f
U535/Y (BUFX2)	0.71 *	2.00 f
u_uart_clk_pad/pad (pad_bidirhe)	1.09 *	3.09 f
uart_clk (out)	0.00 *	3.09 f
data arrival time		3.09
clock main_clock (rise edge)	10.00	10.00
clock network delay (ideal)	0.00	10.00
output external delay	-3.00	7.00
data required time		7.00
data required time		7.00
data arrival time		-3.09
slack (MET)		3.91

Figure 28 Timing after clock insertion

The clock tree summary report is shown in figure 29. The skew should be less than 0.2 ns. The report shows that skew obtained is 0.0071ns.

```
======================= Clock Tree Summary ========================
Clock                Sinks     CTBuffers ClkCells   Skew      LongestPath TotalDRC   BufferArea
-------------------------------------------------------------------------------------------------
main_clock           10        0         1          0.0071    0.3729      0             0.0000
```

Figure 29 Skew report

After the placement and clock tree insertion, the layout now has clock connected to all the cells.

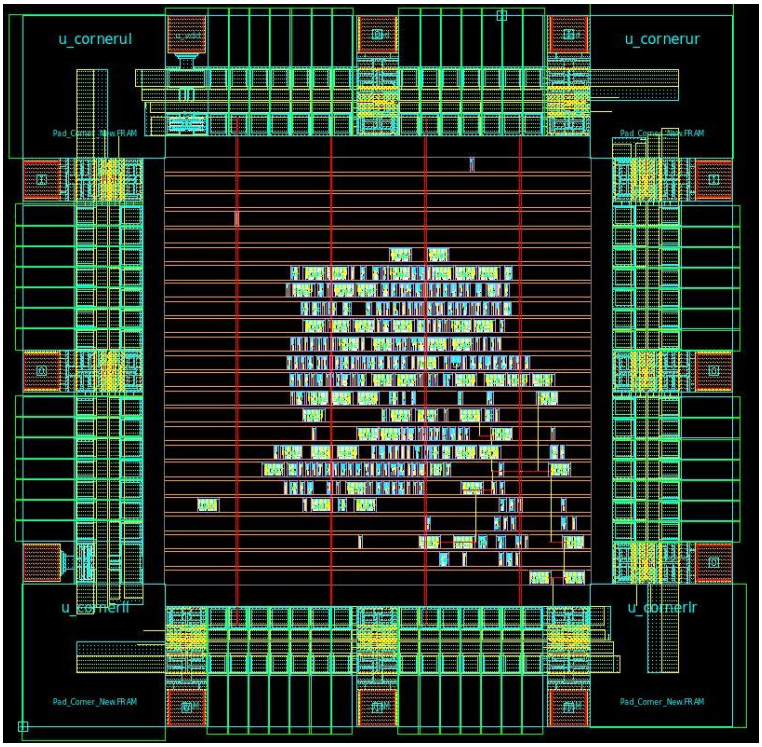


Figure 30 After Placement and clock insertion

The cells should be routed (connected). Routing is a process to connect the cells based on the logic design. Figure 31 shows the routing of cells (lines seen in between the cells). The routing and placement of cells are both inter linked to each other. The cells are placed near to the respective I/O ports to improve the timing of the whole circuitry.

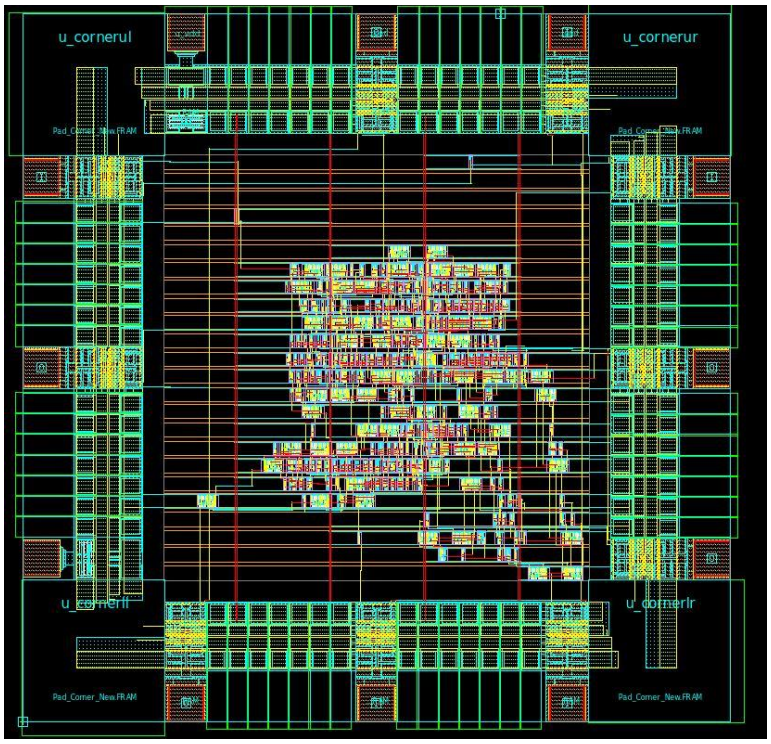


Figure 31 Routing

The cell view of the Input_data register is shown in figure 32. The small square blocks seen are the contacts. These cells are connected through nets.

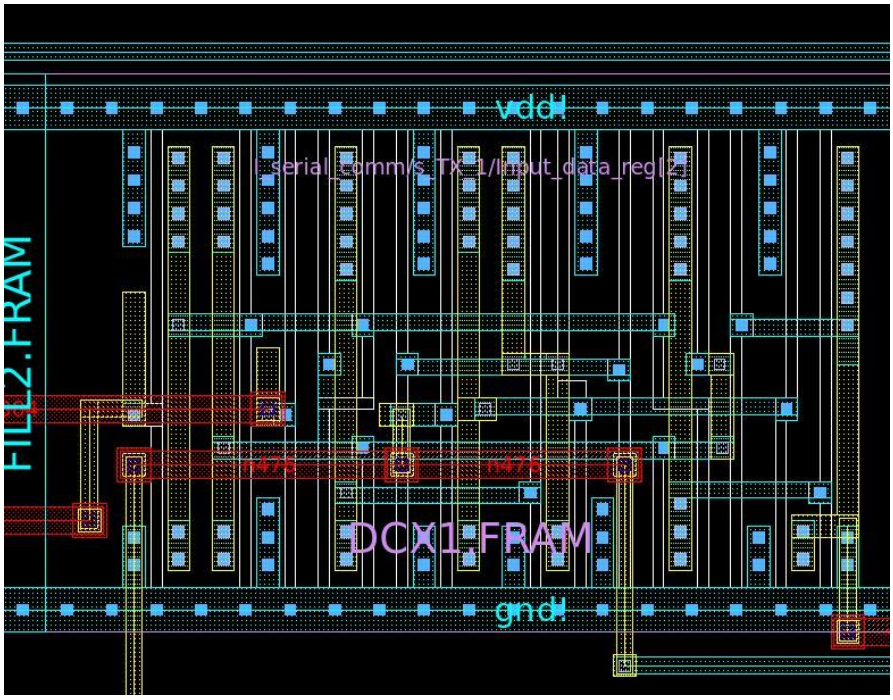


Figure 32 Cell view

The QOR (Quality of Results) report is shown in figure 33. This report is responsible to identify any hold violations. A hold violation is caused when the input signal is changing very frequently after the positive edge of clock signal.

```
Timing Path Group 'main_clock'
-----------------------------------
Levels of Logic:                2.00
Critical Path Length:           2.69
Critical Path Slack:            3.94
Critical Path Clk Period:      10.00
Total Negative Slack:           0.00
No. of Violating Paths:         0.00
Worst Hold Violation:           0.00
Total Hold Violation:           0.00
No. of Hold Violations:         0.00
-----------------------------------
```

Figure 33 QOR report

```
Point                                                    Incr       Path
-----------------------------------------------------------------------------
clock main_clock (rise edge)                             0.00       0.00
clock network delay (propagated)                         0.37       0.37
I_serial_comm/BAUD_RATE_1/BAUD_clk_reg/CLK (DCX1)        0.00       0.37 r
I_serial_comm/BAUD_RATE_1/BAUD_clk_reg/Q (DCX1)          0.90       1.27 f
U535/Y (BUFX2)                                           0.68 c     1.95 f
u_uart_clk_pad/pad (pad_bidirhe)                         1.10 c     3.06 f
uart_clk (out)                                           0.00 *     3.06 f
data arrival time                                                   3.06

clock main_clock (rise edge)                            10.00      10.00
clock network delay (ideal)                              0.00      10.00
output external delay                                   -3.00       7.00
data required time                                                  7.00
-----------------------------------------------------------------------------
data required time                                                  7.00
data arrival time                                                  -3.06
-----------------------------------------------------------------------------
slack (MET)                                                         3.94
```

Figure 34 Timing after routing

After the routing of cells, metal filling is required to fill the empty spaces with dummy metal. Metal layers are classified into metal 1, metal 2 and metal 3.

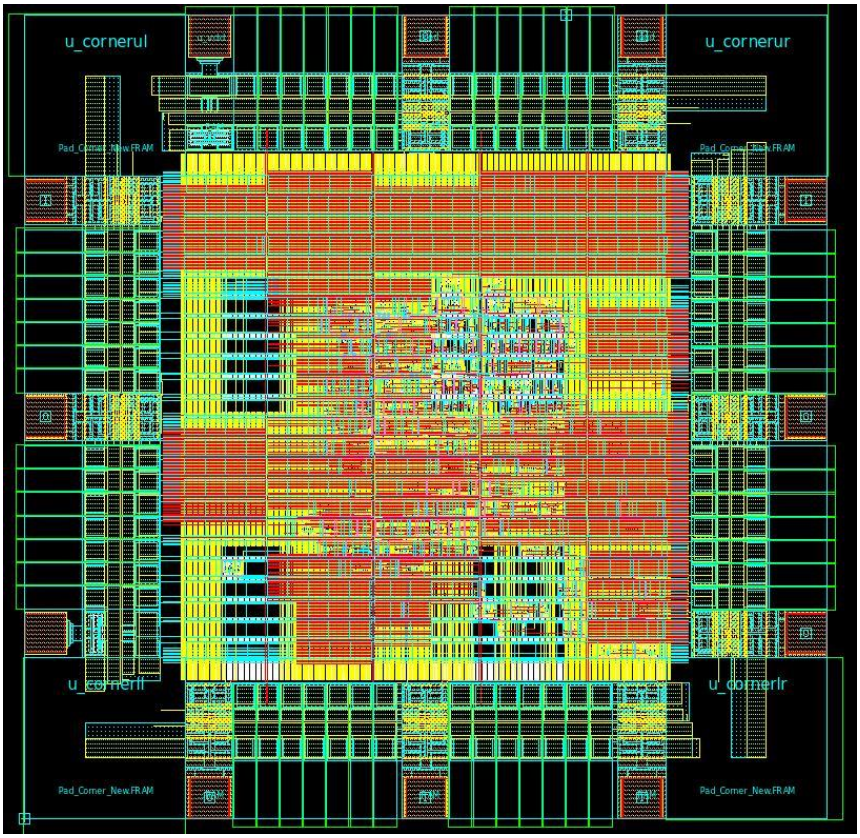


Figure 35 Metal Fill

After the metal fill, the layout vs schematic check is performed. This verification identifies any short nets, open nets and floating nets. The floating ports are fine because the inputs and outputs are not connected. As shown in figure 36, the LVS report says no short nets.

```
** Total Floating ports are 15.
** Total Floating Nets are 0.
** Total SHORT Nets are 0.
** Total OPEN Nets are 0.
** Total Electrical Equivalent Error are 0.
** Total Must Joint Error are 0.

-- LVS END : --
```

Figure 36 Layout Vs Schematic

5.5 DRC and metal density check of the design:

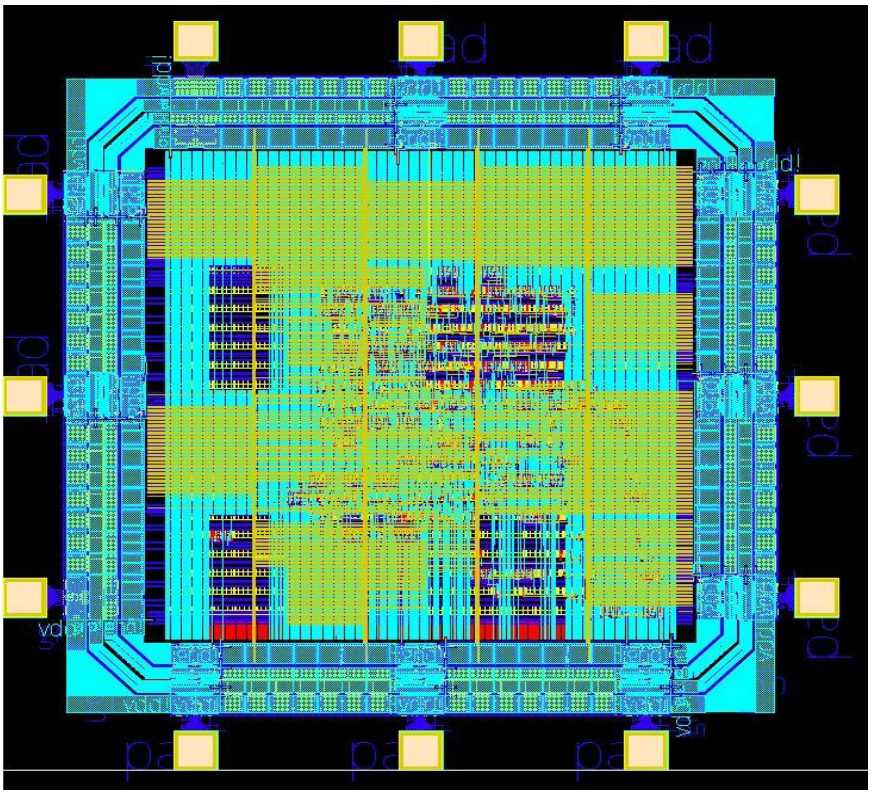


Figure 37 Chip after layout finish

Figure 38 shows the poly layer. The empty spaces should generally be filled with poly by the IC Compiler. These empty spaces should be filled with poly layer. This can be done by using rectangular tool to draw rectangular blocks in the empty spaces.

Figure 39 shows the die after adding the poly manually.

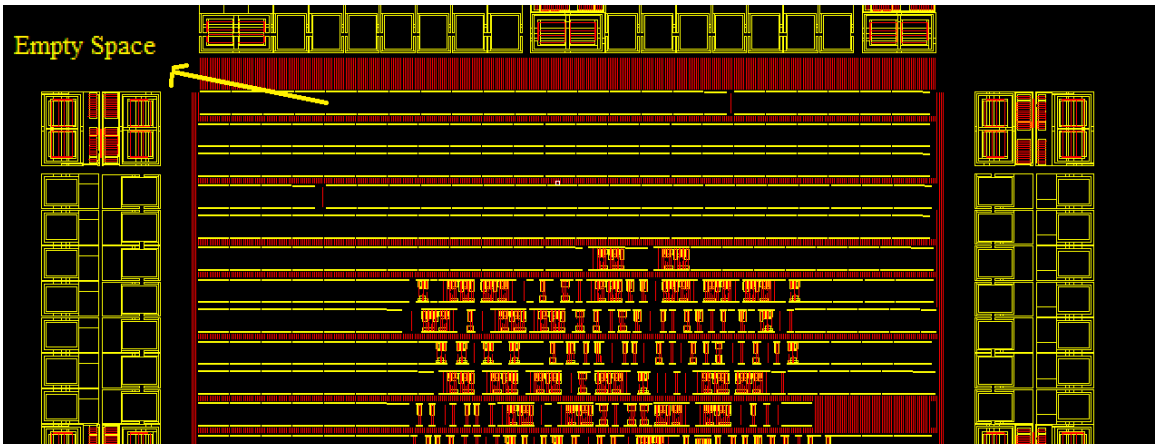


Figure 38 Empty space between layers

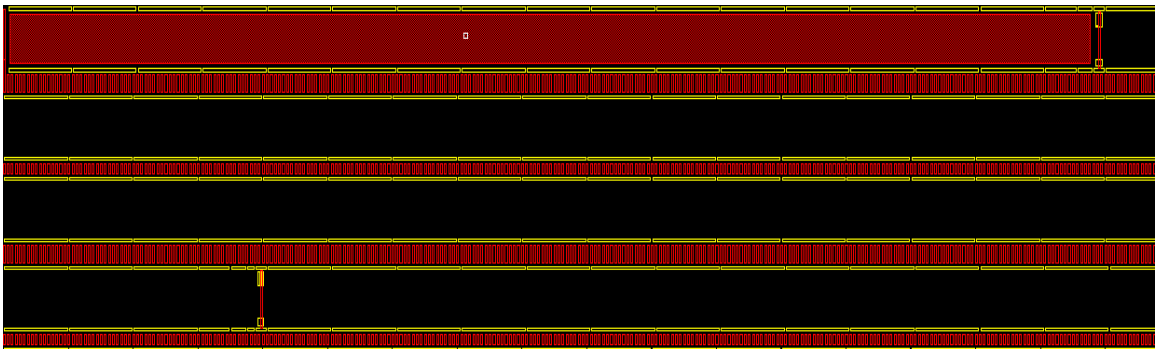


Figure 39 After adding Poly

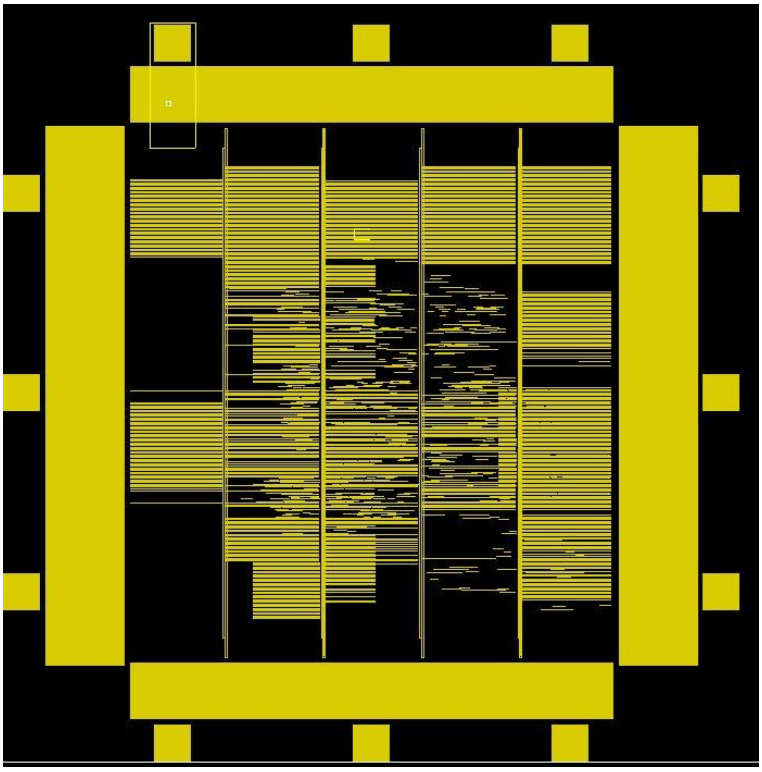


Figure 40 Metal3 drawing

Figure 40 shows the metal 3 drawing, the rectangular blocks (yellow) on the 4 sides of the die is metal3 layer.

Figure 41 shows the DRC (Design Rule Check) setup, the DRC check was performed only within the (260, 260, 1240, 1240) bounding box. Figure 42 shows the DRC report which has returned 0 errors.

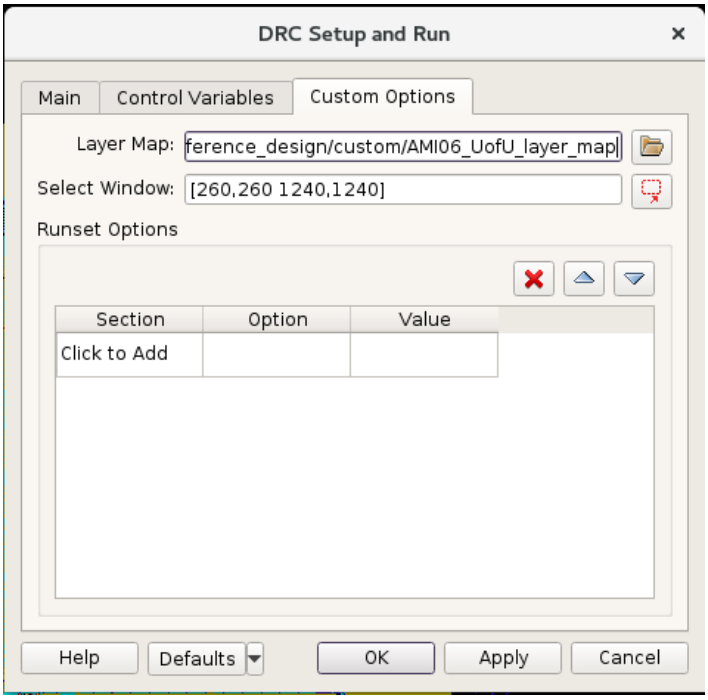


Figure 41 DRC setup

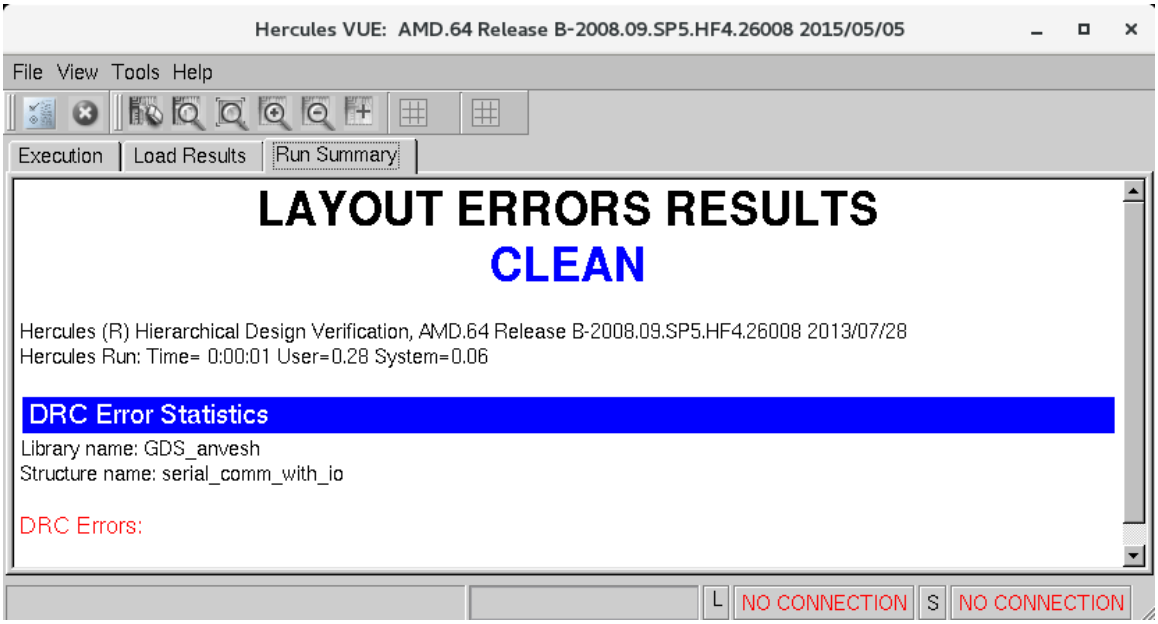


Figure 42 DRC report

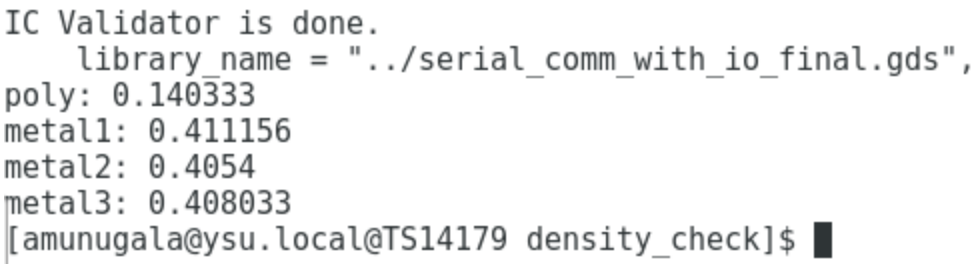


Figure 43 Metal density check

As shown in figure 43, the metal density check results obtained are poly: 14%, metal1: 41%, metal2: 40%, metal3: 40%. The minimum metal density requirements (poly: 12%, metal1, metal2 and metal3: 30-80%) are met.

CHAPTER 6: CONCLUSION AND FUTURE STUDIES

6.1 Conclusion:

The functionality of 8-bit serial data communication module is successfully simulated and verified on Spartan6 FPGA. The ASIC implementation of an 8-bit serial communication module was completed, and the design was successfully submitted to the MOSIS for fabrication on 1st April 2018. The expected delivery of the fabricated chip is in the month of June.

6.2 Verification:

The timing report, area report, short ports, netlist data, Layout vs schematic, design rule check (DRC), any hold violations, any functionality errors, metal density check are performed and documented in the ASIC implementation chapter.

6.3 Future work:

The further studies can be performed on testing the fabricated ASIC for laboratory results. The future work may be focused on implementation of 16-bit synchronous serial communication module with two-way communication and possibly that can handle multiple slaves at same time such as an I2C framework.

REFERENCES

[1] ASIC Implementation of Universal Asynchronous Receiver and Transmitter using 45nm Technology B.Venkataramana1 , P. Srikanth Reddy2 , P. Kishore Kumar3 , K Sivasankaran

[2] He Chun-zhi, Xia Yin-shui and Wang Lum-yao, "A Universal Asynchronous Receiver Transmitter Design," Natural Science Foundation of china(No.61041001)2011, pp.691-694

[3] Y. Fang Yi-yuan and Chen Xue-jun, "Design and Simulation of UART Serial Communication Module Based on VHDL," International Workshop on Intel Sys and App.(ISA 2011),May2011

[4] Garima Bandhawarkar and Iti Agarwal and Shweta Gaba, "Synthesis and Implementation of UART using VHDL codes, " IEEE International Symposium on Computer, Consumer and Control,2012, pp.1-3

[5] Asynchronous Interface, Implementation of Complete ASIC Design Flow by Raviteja Podila, CALIFORNIA STATE UNIVERSITY NORTHRIDGE

[6] Himanshu Bhatnagar, “Advanced ASIC chip Synthesis Using Synopsys Design Compiler, Physical Compiler and Primetime” 2004

[7] VLSI Circuit Design Methodology Demystified: A Conceptual Taxonomy, Liming Xiu

[8] ASIC Design and Implementation of UART with DFT logic for Built-in Self-test using Verilog HDL Author: Kistipati Karthik Reddy ; Jeeru Dinesh Reddy

[9] Synopsys Design Compiler user guide version N-2017.09

[10] Synopsys IC Compiler user guide version L-2016.03-SP2

[11] Digital VLSI chip design with cadence and Synopsys CAD tools {http://www.cs.utah.edu/~elb/cadbook/}

APPENDIX

Serial communication module:

```
//Serial communication module
`include "parameters.v"
module serial_comm     (clk,reset,uart_clk,TX_data,TX_ready,
                                TX_input,TX_done,RX_data,RX_output,RX_ready);
input       clk; input       reset;
output      uart_clk;
//Receiver
input      RX_data;
output    RX_output;
output     RX_ready;
//Transmitter
output    TX_data;
input    TX_input;
input    TX_ready;
output   TX_done;

// Instantiation of the TX, RX and baud modules
TX s_TX_1
(.clk(uart_clk),.reset(reset),.TX_data(TX_data),.TX_ready(TX_ready),.TX_input(TX_inp
ut),.TX_done(TX_done));
RX s_RX_1
(.clk(uart_clk),.reset(reset),.RX_data(RX_data),.RX_output(RX_output),.RX_ready(RX_
ready));
baud BAUD_RATE_1(.clk(clk),.reset(reset),.BAUD_clk(uart_clk));
```

```
endmodule
```

Transmitter module:

```
//Transmitter module
`include "parameters.v"
module TX(clk,reset,TX_ready, TX_data,TX_input,TX_done);

input      clk;  input      reset;  input   TX_ready;
output     TX_data; input          TX_input;
output     TX_done; integer i;
reg TX_data;
reg          TX_done;
reg       [7:0]  Input_data;
reg    [2:0]  next_state, state;
reg          Load_SR;
reg    [3:0]  count;
reg          rst_count;
reg          enable_count;
reg    [1:0]  TX_DS;
reg          shiftEnable;
reg    [4:0]  Delay_High;
reg          DelayEnable;
reg    [7:0]  Tx_ShiftReg;

//Loading of input data into an 8 bit register
always @(posedge clk or posedge reset)
```

```
if(reset) begin Input_data = 8'b0;
i=0; end
else begin
if(TX_ready)begin
if(i < 8) begin
        Input_data[i] = TX_input;
        i = i+1;
        end
        else i=0;
end
end
//Shift register
always @(posedge clk or posedge reset)
  if (reset) Tx_ShiftReg <= 0;
  else  if (Load_SR)  Tx_ShiftReg <= Input_data;
  else if (shiftEnable) begin  Tx_ShiftReg[6:0] <= Tx_ShiftReg[7:1];
  Tx_ShiftReg[7]   <= 1'b0;
  end
  else Tx_ShiftReg      <= Tx_ShiftReg;

//Counter for data length
always @(posedge clk or posedge reset)
  if (reset) count <= 0;
  else if (rst_count) count <= 0;
  else if (enable_count) count <= count + 1;
```

```
//Data select bit generator
always @(Tx_ShiftReg or TX_DS)
  case (TX_DS)
     `tx_strtbit: TX_data = 1'b0;
     `tx_stpbit:  TX_data = 1'b1;
     `tx_ShiftRegbit: TX_data = Tx_ShiftReg[0];
  endcase

//Delay counter
always @(posedge clk or posedge reset)
  if (reset) Delay_High <= 0;
  else if (DelayEnable) Delay_High <= Delay_High + 1;
  else Delay_High <= 0;

always @(posedge clk or posedge reset)
  if (reset) state <= `tx_state0;
  else state <= next_state;

//State machine
always @(state or Delay_High or count or TX_ready)
begin
  rst_count  = 1'b0;
  enable_count = 1'b0;
  TX_DS   = `tx_stpbit;
  TX_done  = 1'b0;
  next_state  = state;
```

```
  Load_SR  = 1'b0;
  DelayEnable = 1'b0;
  shiftEnable = 1'b0;
case (state)
 `tx_state0: begin
     if (TX_ready)begin
       next_state = `tx_state1;
     end
          else      next_state = `tx_state0;
     end
 `tx_state1: begin
     TX_DS   = `tx_strtbit;
          Load_SR  = 1'b1;
     if (Delay_High == 4'h7)
       next_state = `tx_state2;
     else         begin
          next_state = `tx_state1;
       DelayEnable = 1'b1;
     end
   end
    `tx_state2: begin
          TX_DS   = `tx_ShiftRegbit;
      if (count > `data_length) begin next_state = `tx_state3;
     TX_DS   = `tx_stpbit;       end
      else begin
          TX_DS   = `tx_ShiftRegbit;
```

```
      next_state = `tx_state2;
      shiftEnable = 1'b1;
       enable_count = 1'b1;
       end
       end
    `tx_state3: begin
      next_state   = `tx_state0;
      rst_count   = 1'b1;
      TX_done = 1'b1;
      end
   endcase
end
endmodule
```

Receiver module:

```
//Receiver module
`include "parameters.v"
module RX(clk,reset,RX_data,RX_output,RX_ready);

input clk;       input reset;
input RX_data;
output RX_output;
output RX_ready;

reg   rst_count;
reg   RX_ready;
```

```
reg   [2:0] next_state, state;
reg   ack;
reg   [3:0] Delay_High;
reg    Delay_reset;
reg   [7:0] parse_data;
reg   shiftEnable;
reg   [3:0] rxd_Cntr;
reg   enable_count;

reg   RX_output;

//sending the received bit
always @(posedge clk or posedge reset)
          RX_output = parse_data[7];

//acknowledge transmitter data
always @(posedge clk or posedge reset)
  if (reset)
    ack <= 1;
  else
    ack <= RX_data;

//Delay counter
always @(posedge clk or posedge reset)
  if (reset) Delay_High <= 0;
  else if (Delay_reset) Delay_High <= 0;
```

```
  else Delay_High <= Delay_High + 1;

//Shift register
always @(posedge clk or posedge reset)
  if (reset) parse_data <= 0;
  else if(shiftEnable) begin
     parse_data[6:0] <= parse_data[7:1];
     parse_data[7]   <= ack;
  end

//Counter for data length
always @(posedge clk or posedge reset)
  if (reset) rxd_Cntr <= 0;
  else if (enable_count) rxd_Cntr <= rxd_Cntr + 1;
  else if (rst_count) rxd_Cntr <= 0;

always @(posedge clk or posedge reset)
  if (reset) state <= `rx_state0;
  else state <= next_state;

//state machine
always @(state or ack or Delay_High or rxd_Cntr)
begin
  next_state  = state;
  Delay_reset = 1'b1;
  shiftEnable = 1'b0;
```

```
enable_count  = 1'b0;
rst_count = 1'b0;

case (state)
  `rx_state0: begin
    if (!ack) begin
        next_state = `rx_state1;
        end
    else begin
      next_state = `rx_state0;
      rst_count  = 1'b1;
      RX_ready = 1'b1;
    end
  end
  `rx_state1: begin
        if (Delay_High == 4'h6) begin
          RX_ready = 1'b0;
         next_state = `rx_state2;
        end
        else begin
      next_state = `rx_state1;
          Delay_reset = 1'b0;
        end
        end
  `rx_state2: begin
        if (rxd_Cntr > `data_length) begin
```

```
          next_state = `rx_state3;
           rst_count = 1'b1;         end
          else begin
        shiftEnable = 1'b1;
           enable_count = 1'b1;
         next_state = `rx_state2;        end
           end
           `rx_state3: begin
        next_state = `rx_state0;
        RX_ready = 1'b1;           end
  endcase
end
endmodule
```

Baud generator:

```
//Baud generator
`include "parameters.v"
module baud(clk,reset,BAUD_clk);
input clk;
input reset;
output BAUD_clk;
reg    [8:0]  cycles;
reg    BAUD_clk;
//Counter
always @(posedge clk or posedge reset)
  if (reset) begin    cycles  <= 0;    BAUD_clk <= 0;  end
```

```
else if (`cycles == cycles) begin    cycles  <= 0;
   BAUD_clk <= ~BAUD_clk;
  end
else begin
   cycles  <= cycles + 1;
   BAUD_clk <= BAUD_clk; end
endmodule
```

Design Compiler script:

```
set design_name "serial_comm"
set lib_path "~/ON.PDK/samples/logic/"
set syn_path  "/usr/synopsys/Synopsys_tools/Design_Compiler/syn/M-2016.12-SP5-3/libraries/syn"
set search_path [list ./ "$lib_path"
"/home/amunugala@ysu.local/UofU_SYNS_v1_2/UTFSM_libraries/MW_UTAH"
"$syn_path" ../rtl]

set target_library [list  c5n_utah_std_v5_t27.db ]
set synthetic_library [list dw_foundation.sldb standard.sldb]
set link_library [concat $target_library $synthetic_library io.db]

# These cells have two outputs which causes a DRC error in ICC
# INVX1 input cap shoulde be around 14fF
set_attribute [get_lib_pins */TIEHI/Y] max_capacitance 0.2
set_attribute [get_lib_pins */TIELO/Y] max_capacitance 0.2

set_dont_use c5n_utah_std_v5_t27/DCBX1
set_dont_use c5n_utah_std_v5_t27/DCBNX1

if {[shell_is_in_topographical_mode]} {
   source scripts/topo.tcl
}

# Work library
define_design_lib WORK -path ./SYN/WORK

set_app_var hdlin_enable_upf_compatible_naming true
# Add design file list here
```

```
read_sverilog {${design_name}.v }
current_design ${design_name}

# Create a die scope with IOs
source scripts/add_pads.tcl
create_iowrap "${design_name}_with_io" ${design_name}

#Load Timing Constraints
source input/cons.tcl

set_operating_condition typical

if {[shell_is_in_topographical_mode]} {
   #Load Floorplan
   extract_physical_constraints input/fp.def
}


# Power Vectors
read_saif -input  ../rtl/cordic.saif -instance ${design_name} /dut

# Design checks
check_design > reports/check_design.rpt
# Run Synthesis
#compile_ultra -no_autoungroup
compile_ultra
sizeof_collection [get_cells * -hier -filter ref_name=~TIE*]
optimize_netlist -area

report_timing  > reports/${design_name}.timing.rpt
report_power   > reports/${design_name}.power.rpt
report_qor     > reports/${design_name}.qor.rpt
report_area    > reports/${design_name}.area.rpt

# Write Outputs
write -hier -format ddc -output results/${design_name}.ddc
write -hier -format verilog -output results/${design_name}.v
write_sdc results/${design_name}.sdc
write_sdf  results/${design_name}.sdf
```

ICC compiler script:

```
# Add your design name here
set design_name "serial_comm"
set module_name "${design_name}_with_io"
```

```
source -echo scripts/setup.tcl

proc stage_report { stage_name folder } {
  redirect -file "${folder}/${stage_name}.report_qor.rpt"  { report_qor }
  redirect -file "${folder}/${stage_name}.report_timing.rpt"  { report_timing }
  redirect -file "${folder}/${stage_name}.report_power.rpt" { report_power }
}

set_attribute [get_lib_pins */TIEHI/Y] max_capacitance 0.2
set_attribute [get_lib_pins */TIELO/Y] max_capacitance 0.2

# Verify librery consistency
set_check_library_options -logic_vs_phy
check_library

################################################################################
#######
# Source Input Data Base
# - Binary Flow (DDC)
################################################################################
########
# Load Mapped Netlist
import_design ../syn/results/${design_name}.ddc -format ddc

# Append IO Cells
# Need to run this script twice
#      once with is_pad.tcl commented out.
#      second time with newly created file copied into is_pad.tcl from run directory
source scripts/create_pads.tcl
source is_pad.tcl

################################################################################
#######
# Design Planning
# Floorplan:
# - When short on routing layers adjust row_core_ratio to provide space
#   between site rows.
################################################################################
########

create_floorplan \
   -control_type "width_and_height" \
   -core_height 900 \
   -core_width 900 \
   -row_core_ratio "0.8" \
   -no_double_back \
```

```
    -start_first_row \
    -left_io2core  -30 \
    -right_io2core -30 \
    -bottom_io2core -30 \
    -top_io2core -30

insert_pad_filler -cell "pad_space43_2 "

# Now we load the fp since is
# {0.000 0.000} {1500.000 1497.000}
#read_def ../syn/input/fp.def

derive_pg_connection -power_net {vdd!} -ground_net {gnd!}  -create_ports top

create_pad_rings -nets {vdd! gnd!}

# Strap numbers is a trade off between congestion and IR Drop
create_power_straps  \
    -direction vertical \
    -num_placement_strap 4 \
    -start_at 450  \
    -increment_x_or_y 200 \
    -nets  {vdd! gnd!}  \
    -width 2.700 \
    -layer metal3

write_def -rows_tracks_gcells -pins -macro -output ../syn/input/fp.def
get_attribute [get_die_area] bbox

## Checkpoint 1 Floor Planning
#start_gui
return

##############################################################################
#######
#
# Placement
#  - Optimize for congestion
##############################################################################
########
#place_opt -effort high -congestion
place_opt -effort high -area_recovery

stage_report place_opt reports
save_mw_cel -as ${design_name}_postPlaceOPT
## Checkpoint 2 Placement
```

```
#start_gui
return

##############################################################################
#######
# Clock Tree Synthesis
##############################################################################
########
clock_opt -no_clock_route -only_cts
report_clock_tree -summary


# Enable hold fixing, since CTS is synthesized (attribute is stored in MW).
set_fix_hold [all_clocks]
extract_rc

clock_opt -no_clock_route -only_psyn \
   -area_recovery -power


route_zrt_group -all_clock_nets
report_clock_tree -summary

# Post CTS Netlist optimization
stage_report clock_opt reports
save_mw_cel -as ${design_name}_postCTS

report_clock_tree
report_timing

## Checkpoint 3  - Clock Tree
#start_gui
return

# Route PG rails for std cells (CTS/PSYNOPT)
derive_pg_connection -power_net {vdd!} -ground_net {gnd!} -verbose
preroute_standard_cells -nets {vdd! gnd!} -connect horizontal
get_attribute [get_die_area] bbox


##############################################################################
#######
# Routing
#  - Optimize for congestion
##############################################################################
########
```

```
route_opt -initial_route_only
report_qor

set_app_var routeopt_drc_over_timing true
route_opt -effort high -incremental -only_design_rule

# Do we still see DRC?
focal_opt -drc_nets all
# Account for new buffers inserted
derive_pg_connection -power_net {vdd!} -ground_net {gnd!} -verbose
preroute_standard_cells -nets {vdd! gnd!} -connect horizontal
route_zrt_detail -incremental true

# Checkpoint 2 - Routed design
save_mw_cel -as ${design_name}_postRouteOpt
stage_report route_opt reports

## Checkpoint 4  - Routing
#start_gui
return


##########################################################################
#######
# Chip Finish
#  - Insert filler cells (Provide NWELL & rail continuity)
#  - Fix Antenna violations gate driving nets (%RouteAre/%GateArea)
#  - Fill small gaps with NWELL
##########################################################################
########
# Std Cell Fillers
insert_stdcell_filler -cell_with_metal "FILL8 FILL4 FILL2 FILL"  -connect_to_power
"vdd!" -connect_to_ground "gnd!"
derive_pg_connection -power_net {vdd!} -ground_net {gnd!} -verbose
preroute_standard_cells -nets {vdd! gnd!} -connect horizontal
# Antenna Constraints
set_route_zrt_detail_options -default_gate_size 1.8 -default_port_external_gate_size 1.8
define_antenna_rule $mw_design_library -mode 1 -diode_mode 0 -metal_ratio 1000 -
cut_ratio 0
route_zrt_detail -incremental true
verify_zrt_route -antenna true

## Checkpoint 5  - Fill
#start_gui
return
```

```
# Fill to meet density rule
#insert_metal_filler  -out self  -fill_poly  -timing_driven  -to_metal 3
insert_metal_filler  -fill_poly -bounding_box { { 260 260 } { 1240 1240 } }  -out self  -
timing_driven  -to_metal 3
# Requires ICV
report_metal_density

## Checkpoint 6  - Density checks and fix
#start_gui
return

# Layer fill
#insert_well_filler -layer NWELL -ignore_PRboundary
# This one makes the well to align preventing DRC
set_parameter -name wellFillerAlignWithCell -value 1 -module apl
insert_well_filler  -layer {NWELL} -ignore_PRboundary -higher_edge min -lower_edge
max

save_mw_cel -as ${design_name}_postChipFinish

## Checkpoint 6  - Metal Density
#start_gui
return

##########################################################################
#######
# Signoff
# - Run LVS Checks on routing layers (Fix LVS before proceeding to DRC check)
# - Call Hercules for DRC:
#   - Checking routing layers only
#   - Exclude IO area (Use die dimension as reference)
##########################################################################
########
verify_pg_nets
verify_lvs -max_error 2000 -check_open_locator -check_short_locator
verify_lvs
save_mw_cel -as ${design_name}_postLVS
#start_gui
return
```