ROUTER DESIGN

MASTERS DEGREE

IN

COMPUTER ENGINEERING

Submitted by

KUMAR SAI REDDY GUNTAKA 202616615

Under the Supervision of
Orod Haghighiara

CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

December, 2022

Abstract

In this project I created FIFO memory modules with synchronizers and an FSM that transports internal signal data packets to create a top level router module that is capable of transferring data packets from one host to another. and a register module for storing data packets Here, I am planning to develop and test each module separately before building a router top module that allows data packets to transit from source to destination.

Keywords: router; FIFO; FSM; synchronizer; Register; Data Packets;

Academic Dishonesty

I hereby attest that this lab report is entirely my own work.I have not copied either code or text from anyone, nor have I allowed or will I allow anyone to copy my work.

Name (printed) KUMAR SAI REDDY GUNTAKA

Name (signed) KUMAR SAI REDDY GUNTAKA Date 12/18/2022

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 About

Routing is the act of transporting a data packet from its origin to its destination, allowing messages to move from one computer to another and finally reach the destination system. A router is a networking device that transmits data packets from one computer network to another. It is linked to two or more data lines from several networks (as opposed to a network switch, which connects data lines from one single network).

1.2 Cisco Packet Tracker

A router is a networking device that sends data packets from one computer network to another. Routers are responsible for traffic routing on the Internet. Data packets are used to represent data transferred over the internet, such as a web page or an email. A router is connected to two or more data connections from distinct IP networks. The router scans the network address information in the packet header to identify the eventual destination when a data packet arrives on one of the lines.

1.3 Router packet

Packet format: A packet consists of three parts: a header, a payload, and parity, each of which is eight bits wide, and the length of the payload can be increased from one to 63 bytes.

1.3.1 Header:

The packet header has two fields: DA and length. DA: the packet's destination address is two bits long. Based on the destination address of the

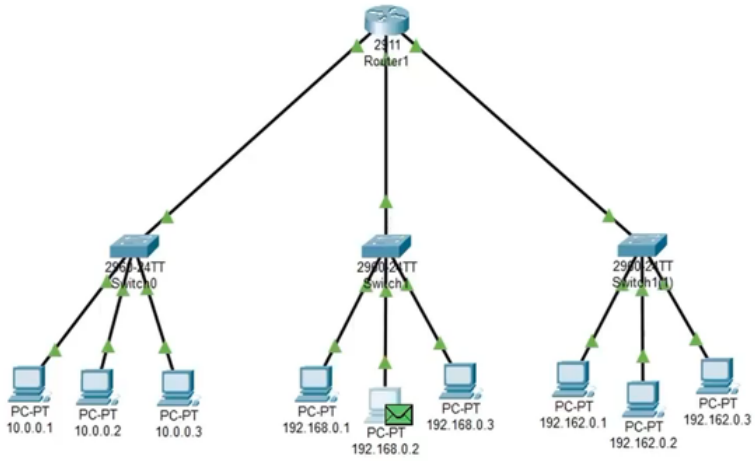

Figure 1.1: Data Transmission Using Router

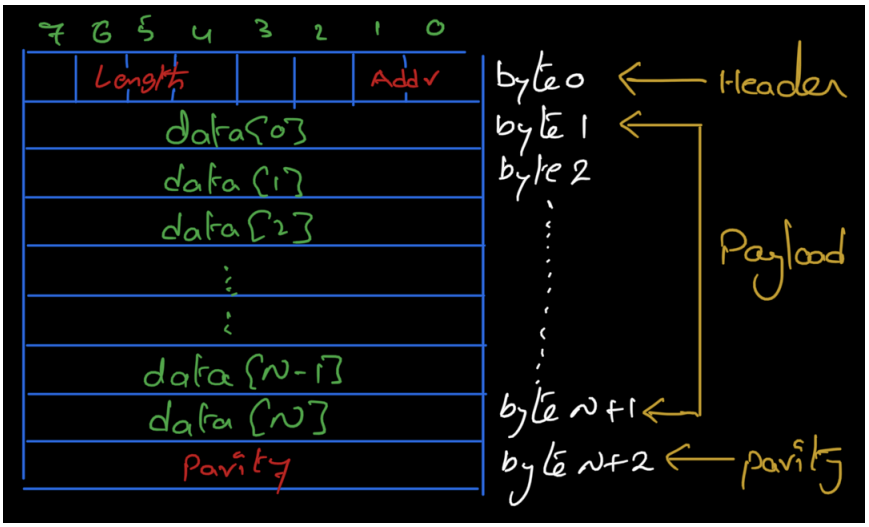

Figure 1.2: Data Packet Format

packets, the router routes them to the appropriate ports. Each output port has a two-bit port address. If the packet's destination address matches the port address, the router sends the packet to the output port. The address "3" is incorrect.

Length: The data length is 6 bits. It defines the number of bytes in the data. A packet can have a data size of 1 byte or a maximum of 63 bytes.

1.3.2 Payload:

payload is the data information. Data should be in terms of the bytes.

1.3.3 Parity:

This field contains the security check of the packet. It is calculated as bitwise parity over the header and payload bytes of the packet.

1.4 Input Router Protocol:

The input protocol is as follows: Except the low reset, all input signals are active high and synced to the clock's falling edge. Because the router is sensitivity to the clock's rising edge. As a result, driving input signals on the falling edge assures setup and hold time. This clocking block may be used to drive the signals on the clock's positive edge, avoiding metastability.

When the header byte is driven onto the input data bus, the packet valid signal is asserted on the same clock edge. Since the header byte provides the address, this the router to which output channel the packet should be routed to (data out 0, data out 1, data out 2). For each new falling edge of the clock, the next byte of payload after the header byte shall be pushed on the input data bus.

After the last payload byte has been driven, the packet valid signal must be de-asserted and the packet parity must be driven on the next falling edge of the clock. This indicates packet completion. When a busy signal is detected, it should not drive any bytes but instead keep the last driven values. When the 'busy' signal is asserted, any incoming byte of data is dropped. When a packet parity mismatch is discovered, the "err" signal is asserted.

1.5 Output Router Protocol:

The output protocol is as follows: All output signals are active high and synced to the clock's rising edge. Each data out X output port (data out 0, data out 1, data out 2) is internally buffered by a 16X9 FIFO. When valid data occurs on the vld out X(data out 0,data out 1,data out 2) output bus, the router asserts the vld out X (vld out 0,vld out !,vld out 2) signal. This is a signal sent to the receiver's client indicating that data is accessible on

a certain output data bus. The packet receiver will then wait until it has enough room to keep the packet's bytes before responding with the read enb X(read enb 0.read enb 1,read enb 2) signal. On the falling clock edge, the read enb X(read enb 0,read enb '1 or read enb 2) input signal can be asserted, allowing data from the data out X(data out 0,data out 1,data out 2) bus to be read. The read enb X (read enb 0, read out 1, or read out 2) must be asserted within 30 clock cycles of the vld out X (vld out 0, vld out 1, or vld out 2) being asserted, otherwise the FIFO will be reset. When a packet's header byte is lost owing to a time-out event, the data out X (data out 0, data out 1, or data out 2) bus will be tri-stated (high Z).

CHAPTER 2

Methodology

2.1 Modules Used

From these chapters I am going to discuss about the each module used in the design:

1.FSM Module

2.Register

3.Synchronizer

4.FIFO module.

2.2 FSM Module

The router architecture's monitoring and controlling unit is the FSM. In this design, Moore's FSM is employed.

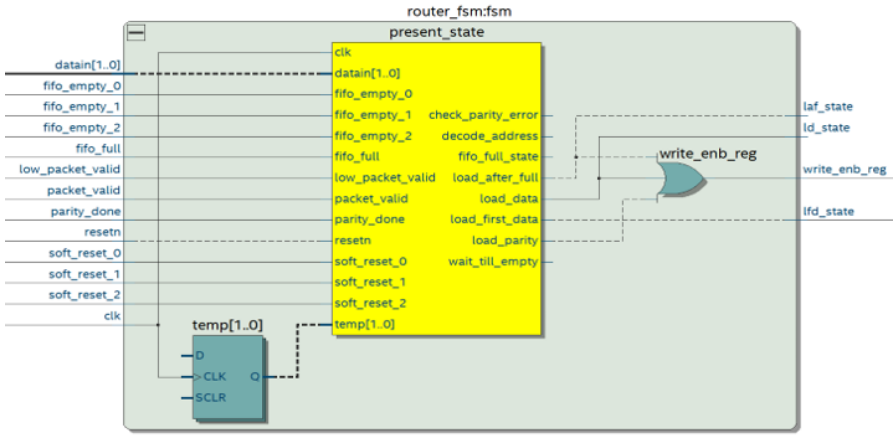

Figure 2.1: RTL design for FSM module[**berlin**]

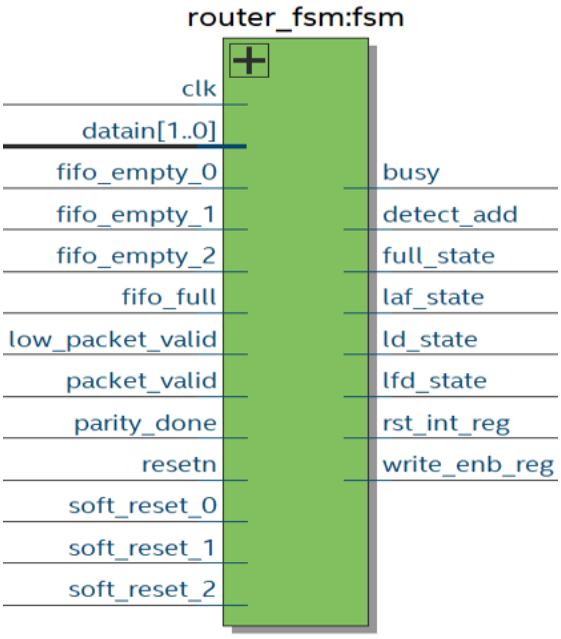

Figure 2.2: FSM Module[**berlin**]

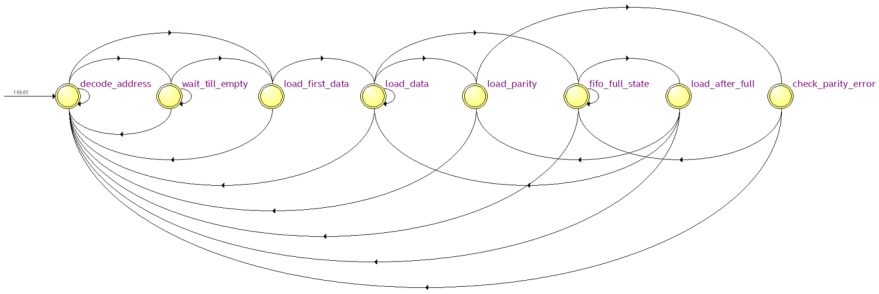

Figure 2.3: State machine[**berlin**]

	Name	load_after_full	fifo_full_state	load_parity	load_data	load_first_data	wait_till_empty	decode_address	check_parity_error
1	decode_address	0	0	0	0	0	0	0	0
2	wait_till_empty	0	0	0	0	0	1	1	0
3	load_first_data	0	0	0	0	1	0	1	0
4	load_data	0	0	0	1	0	0	1	0
5	load_parity	0	0	1	0	0	0	1	0
6	fifo_full_state	0	1	0	0	0	0	1	0
7	load_after_full	1	0	0	0	0	0	1	0
8	check_parity_error	0	0	0	0	0	0	1	1

Figure 2.4: State Table[**berlin**]

2.2.1 state 1: Decode Address

1.This is the initial reset state.

2.In this state, the signal detect add is asserted, which is utilized to detect an incoming packet. It is also used as a header byte to latch the first byte.

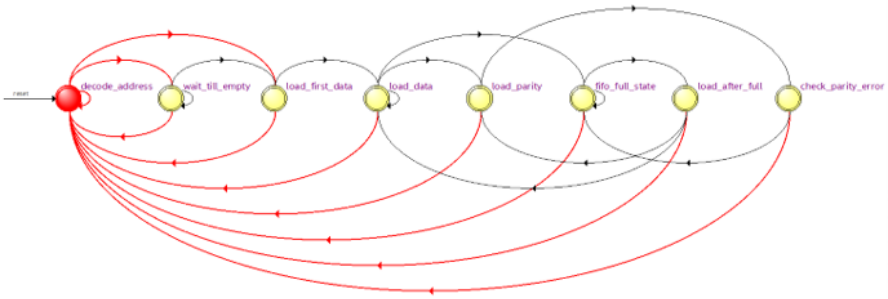

Figure 2.5: FSM for state 1[**berlin**]

```
decode_address:   // decode address state
begin
    if((packet_valid && (datain==2'b00) && fifo_empty_0)||
    (packet_valid && (datain==2'b01) && fifo_empty_1)||
    (packet_valid && (datain==2'b10) && fifo_empty_2))

            next_state<=load_first_data;   //lfd_state

    else if((packet_valid && (datain==2'b00) && !fifo_empty_0)||
    (packet_valid && (datain==2'b01) && !fifo_empty_1)||
    (packet_valid && (datain==2'b10) && !fifo_empty_2))
            next_state<=wait_till_empty;  //wait till empty state

    else
        next_state<=decode_address;    // same state
end
```

Figure 2.6: code for state1[**berlin**]

2.2.2 State 2: Wait Till End

Busy signal is made high and write enable reg signal is made low.

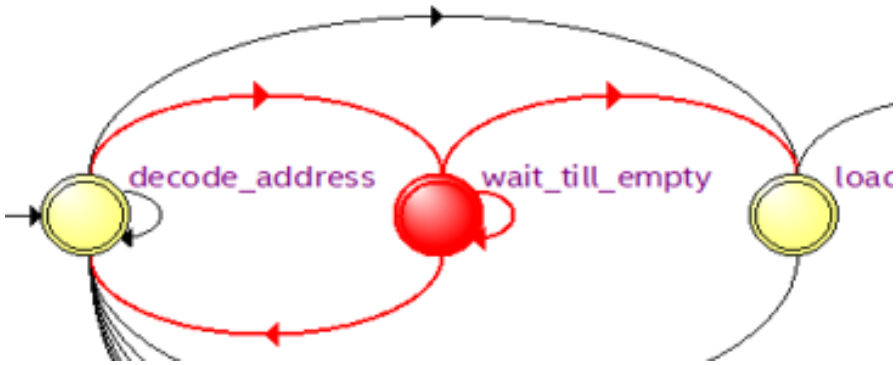

Figure 2.7: state2 FSM[**berlin**]

```
wait_till_empty:          //wait till empty state
begin
    if((fifo_empty_0 && (temp==2'b00))||
    (fifo_empty_1 && (temp==2'b01))||
    (fifo_empty_2 && (temp==2'b10)))
    //fifo is empty and were using same fifo
            next_state<=load_first_data;

        else
            next_state<=wait_till_empty;
    end
```

Figure 2.8: code for state2[**berlin**]

2.2.3 State 3: Load First Data

1. In this state, the signal load first data state is asserted, which is used to load the first data byte into the FIFO.

2. In this condition, signal busy is additionally asserted so that the already latched header byte does not change to a new value for the current packet.

3. In the following clock cycle, this state is unconditionally changed to LOAD DATA.

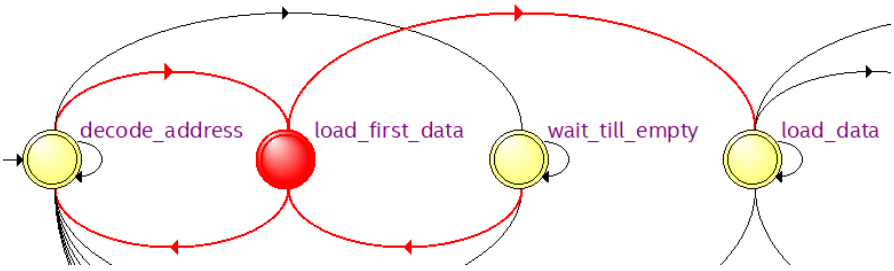

Figure 2.9: state3 FSM[**berlin**]

```
load_first_data:          // load first data state
begin
    next_state<=load_data;
end
```

Figure 2.10: code for state2[**berlin**]

2.2.4 State 4: Load Data

1.The signal load data state is asserted in this state, which is used to load the payload data into the FIFO.

2.In this condition, signal busy is de asserted so that the ROUTER may accept fresh data from the input source every clock cycle, and signal write enable reg is asserted to write the Packet information (Header+Payload+Parity) to the specified FIFO.

3.When pkt valid is low, this state transitions to LOAD PARITY, and when FIFO is full, it transitions to FIFO FULL STATE.

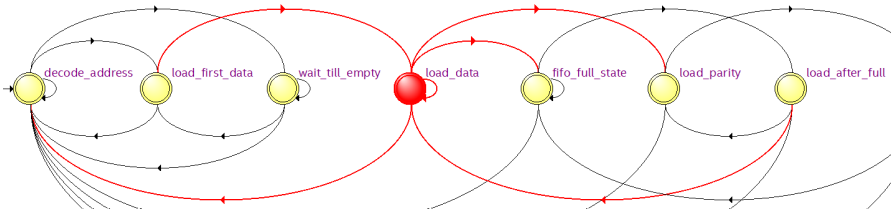

Figure 2.11: state4 FSM[**berlin**]

```
load_data:                              //load data
begin
    if(fifo_full==1'b1)
            next_state<=fifo_full_state;
    else
            begin
                if (!fifo_full && !packet_valid)
                    next_state<=load_parity;
                else
                    next_state<=load_data;
            end
end
```

Figure 2.12: code for state2[**berlin**]

2.2.5 State 5: FIFO Full State

1.The busy signal is set to high, but the write enable reg signal is set to low.

2.The signal full state is asserted, indicating that the FIFO is full.

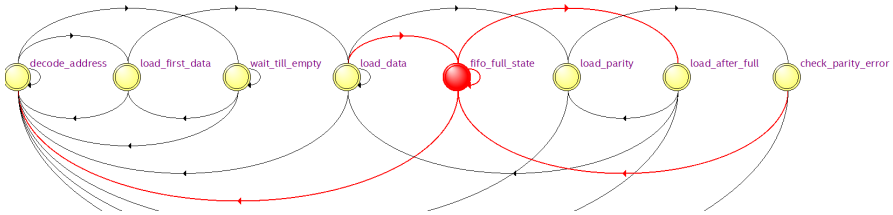

Figure 2.13: state5 FSM[**berlin**]

```
fifo_full_state:              //fifo full state
begin
    if(fifo_full==0)
        next_state<=load_after_full;
    else
        next_state<=fifo_full_state;
end
```

Figure 2.14: code for state2[**berlin**]

2.2.6 State 6: Load parity

1.The final byte, which is the parity byte, is latched in this condition.

2.It always returns to the state CHECK PARITY ERROR. Signal busy is asserted, indicating that the ROUTER is not accepting fresh data.

3.write enable reg is set to high to latch the parity byte to the FIFO.

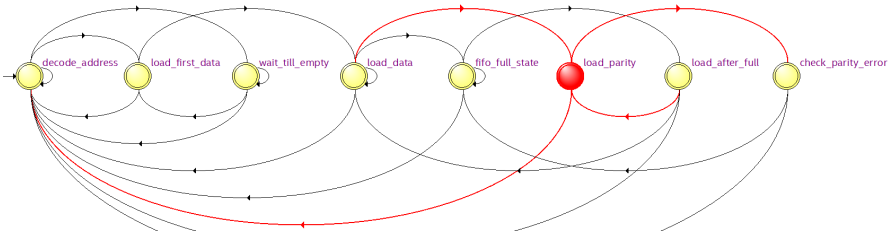

Figure 2.15: state6 FSM[**berlin**]

```
load_parity:                  // load parity state
begin
    next_state<=check_parity_error;
end
```

Figure 2.16: code for state2[**berlin**]

2.2.7 State 7: Load After Full

1.The load after full state signal is asserted in this state, which is used to latch the data following FIFO FULL STATE.

2.The signals busy and write enb reg are asserted.

3.It looks for the parity done signal, and if it is high, it indicates that the LOAD PARITY state has been identified, and it proceeds to the DECODE ADDRESS stage.

4.If low packet valid is set to true, the state changes to LOAD PARITY; otherwise, it returns to LOAD DATA.

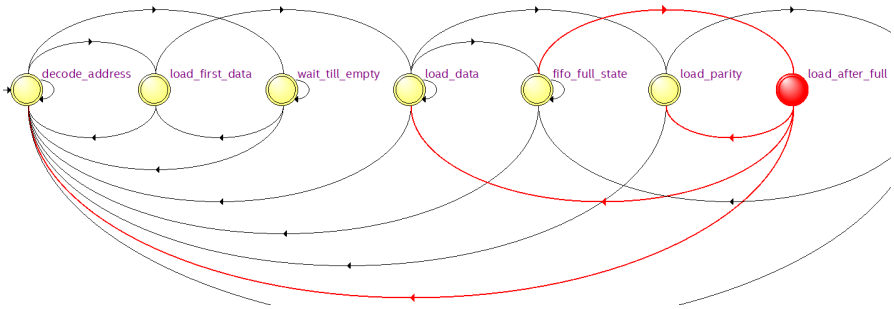

Figure 2.17: state7 FSM[**berlin**]

```
load_after_full:            // load after full state
begin
    if(!parity_done && low_packet_valid)
        next_state<=load_parity;
    else if(!parity_done && !low_packet_valid)
        next_state<=load_data;

    else
        begin
            if(parity_done==1'b1)
                next_state<=decode_address;
            else
                next_state<=load_after_full;
        end

end
```

Figure 2.18: code for state2[**berlin**]

2.2.8 State 8: Check Parity Error

1.The reset int reg signal is created in this condition, and it is used to reset the low packet valid signal.

2.When the FIFO is not full, this state changes to DECODE ADDRESS, and when it is full, it changes to FIFO FULL STATE. In this stage, busy is emphasized.

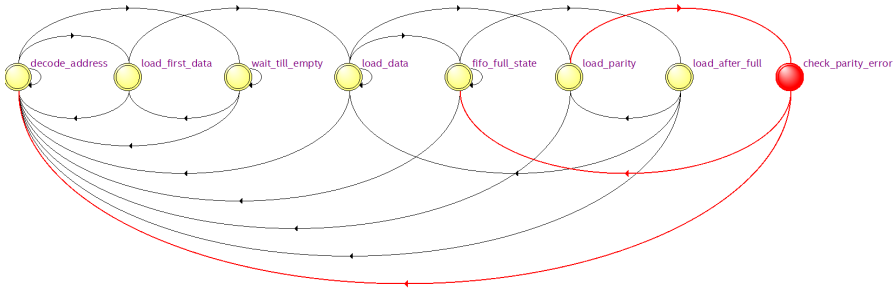

Figure 2.19: state8 FSM[**berlin**]

```
check_parity_error:          // check parity error
begin
    if(!fifo_full)
        next_state<=decode_address;
    else
        next_state<=fifo_full_state;
end
```

Figure 2.20: code for state2[**berlin**]

	Source State	Destination State	
1	check_parity_error	decode_address	(!fi
2	check_parity_error	fifo_full_state	(fif
3	decode_address	decode_address	(!d
4	decode_address	load_first_data	(!te
5	decode_address	wait_till_empty	(!te
6	fifo_full_state	decode_address	(!te
7	fifo_full_state	fifo_full_state	(fif
8	fifo_full_state	load_after_full	(!fi
9	load_after_full	decode_address	(!p
10	load_after_full	load_data	(!p
11	load_after_full	load_parity	(!p
12	load_data	decode_address	(!te
13	load_data	fifo_full_state	(fif
14	load_data	load_data	(pa
15	load_data	load_parity	(!p
16	load_first_data	decode_address	(!te
17	load_first_data	load_data	(!te
18	load_parity	check_parity_error	(!te
19	load_parity	decode_address	(!te
20	wait_till_empty	decode_address	(!te
21	wait_till_empty	load_first_data	(!te
22	wait_till_empty	wait_till_empty	(!te

Figure 2.21: Sources and Destinations[**berlin**]

CHAPTER 3

Register Module

3.1 Functionality:

When the relevant FIFO is filled, this module provides an internal register named data reg to retain the data packet. And when the FSM's load first data state signal asserts, the data in data reg is transferred to the appropriate FIFO. When the signal resetn is low, the signals dout, parity done, and low pkt valid are also low. When load data state is high and fifo full and pkt valid are low, the signal parity done is high. If the pkt valid signal is not asserted during the LOAD DATA state, the data is not loaded. The value of low packet valid is increased to indicate that the loading packet is invalid.

This module's registers are all latched on the rising edge of the clock. 1. If resetn is low, the signals output, error, parity done, and low pkt valid are also low.

2. Under the following situations, the signal parity done is high:

3. When the load data state signal is high and the signals (fifo full and pkt valid) are low.

4. When both the load full state and the low pkt valid signals are high, and the previous value of parity done is low.

5. The signal reset int reg is used to reset the low pkt valid signal.

6. To reset the parity done signal, utilize the detect add signal.

7. When load data state is high and pkt valid is low, the signal low pkt valid is high. Low packet valid indicates that the current state's pkt valid has been de-asserted.

8. When the detect add and pkt valid signals are high, the first data byte, i.e., the header, is latched within an internal register. When load full data state is set to high, this data is latched to the output dout.

9. If the ld state signal is high and the fifo full signal is low, signal data in, i.e. payload, is latched to dout.

10. When ld state and fifo full are both high, the signal data in is latched to an internal register. When laf state is set to high, this data is latched to output dout.

11. Internal parity is calculated using Full state.

12. Internal parity is stored in another internal register for parity matching.

Last Payload Data: The err is only calculated after the packet parity has been loaded and becomes high if the packet parity does not match the internal parity.

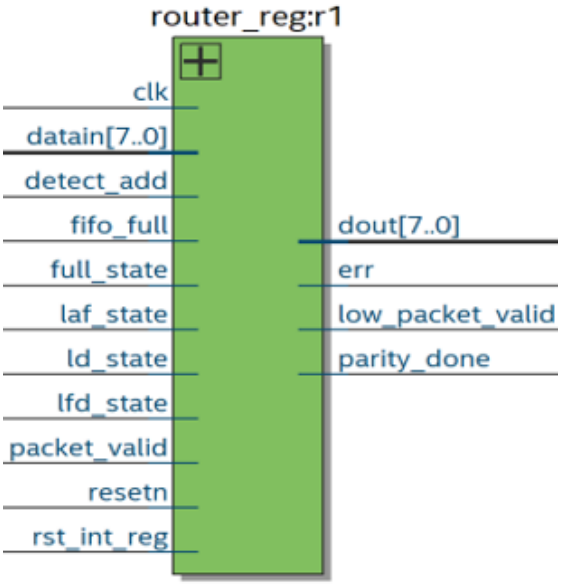

Figure 3.1: Register RTL_1[**berlin**]

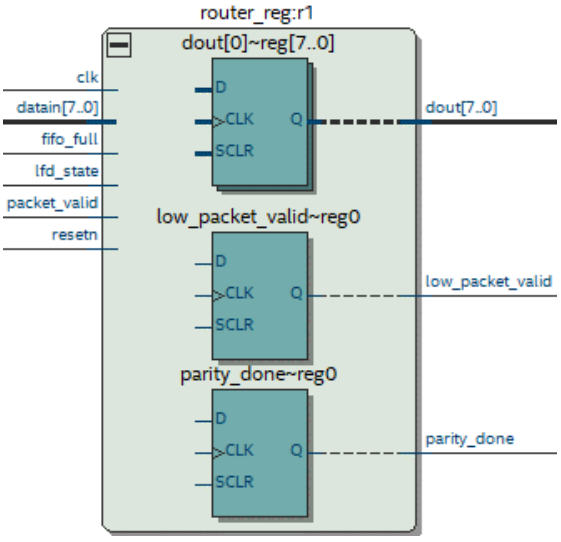

Figure 3.2: Register RTL_2[**berlin**]

```
//parity done
always@(posedge clk)
    begin
        if(!resetn)
            begin
                parity_done<=1'b0;
            end

        else
            begin
                if(ld_state && !fifo_full && !packet_valid)
                    parity_done<=1'b1;
                else if(laf_state && low_packet_valid && !parity_done)
                    parity_done<=1'b1;
                else
                    begin
                        if(detect_add)
                            parity_done<=1'b0;
                    end
            end
    end
```

Figure 3.3: Parity Done[**berlin**]

```
//low_packet valid
always@(posedge clk)
    begin
        if(!resetn)
            low_packet_valid<=1'b0;
        else
            begin
                if(rst_int_reg)
                    low_packet_valid<=1'b0;
                if(ld_state==1'b1 && packet_valid==1'b0)
                    low_packet_valid<=1'b1;
            end
    end
```

Figure 3.4: Low packet Valid[**berlin**]

```
//dout
always@(posedge clk)

    begin
        if(!resetn)
            dout<=8'b0;
        else
        begin
            if(detect_add && packet_valid)
                hold_header_byte<=datain;
            else if(lfd_state)
                dout<=hold_header_byte;
            else if(ld_state && !fifo_full)
                dout<=datain;
            else if(ld_state && fifo_full)
                fifo_full_state_byte<=datain;
            else
                begin
                    if(laf_state)
                        dout<=fifo_full_state_byte;
                end
        end
    end
```

Figure 3.5: dout[**berlin**]

```
// internal parity
always@(posedge clk)
    begin
        if(!resetn)
            internal_parity<=8'b0;
        else if(lfd_state)
            internal_parity<=internal_parity ^ hold_header_byte;
        else if(ld_state && packet_valid && !full_state)
            internal_parity<=internal_parity ^ datain;
        else
            begin
                if (detect_add)
                    internal_parity<=8'b0;
            end
    end
```

Figure 3.6: internal parity[**berlin**]

```
//error and packet_
always@(posedge clk)
    begin
        if(!resetn)
            packet_parity_byte<=8'b0;
        else
            begin
                if(!packet_valid && ld_state)
                    packet_parity_byte<=datain;
            end
    end
```

Figure 3.7: error and parity[**berlin**]

```
//error
always@(posedge clk)
    begin
        if(!resetn)
            err<=1'b0;
        else
            begin
                if(parity_done)
                begin
                    if(internal_parity!=packet_parity_byte)
                        err<=1'b1;
                    else
                        err<=1'b0;
                end
            end
    end
```

Figure 3.8: error[**berlin**]

3.2 Simulation

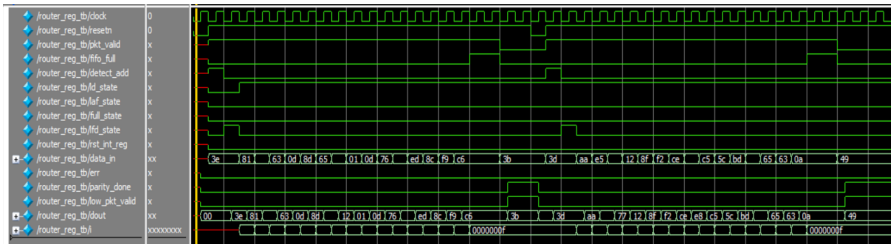

Figure 3.9: register simulation result[**berlin**]

CHAPTER 4

Synchronizer

4.1 Functionality

This sub-block validates data packets and generates the soft reset signal to reset the router FSM and FIFO. In addition, it sends a write enable signal to one of the FIFOs.

The detect add and data in signals are used to choose a FIFO until the packet routing for the specified FIFO is completed. The signal fifo full is asserted based on the full status of fifo 0, FIFO 1, or FIFO 2. If data in = 2b00, fifo full=full 0.

If data in=2b01, fifo full=full 1. If data in=2b10, fifo full=full 2 else fifo full=0

The signal vld out x is created dependent on the FIFO's empty status, as seen below: empty 0 == vld out 0 empty 1 == vld out 1 empty 2 == vld out 2 The write enable reg signal is used to create the write enable signal for the specified FIFO's write operation. Each FIFO has three internal reset signals (soft reset 0, soft reset 1, and soft reset 2). If read enable X (read enable 0,read out 1,read out 2) is not asserted within 30 clock cycles of vld out X (vld out 0,vld out 1, or vld out 2) being asserted, the appropriate internal reset signals go high.

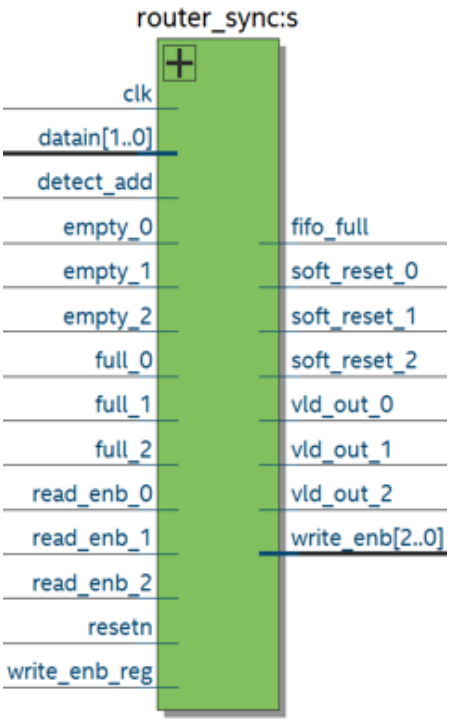

Figure 4.1: synchronizer module[**berlin**]

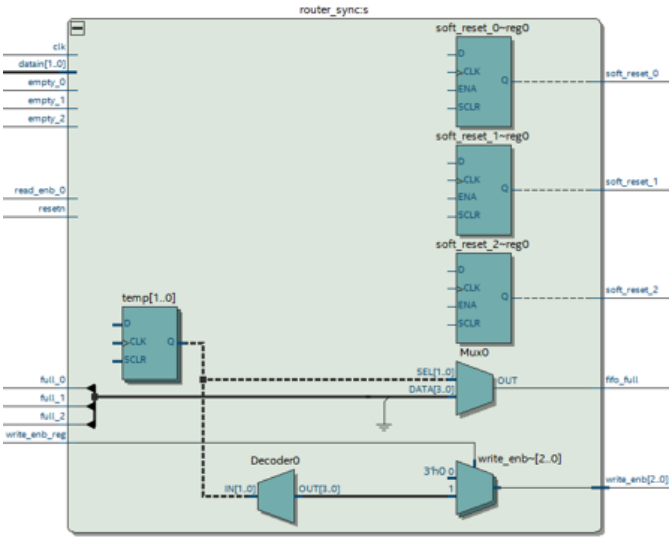

Figure 4.2: synchronizer rtl module[**berlin**]

```
//for fifo full
always@(*)
    begin
        case(temp)
            2'b00: fifo_full=full_0;
            2'b01: fifo_full=full_1;
            2'b10: fifo_full=full_2;
            default fifo_full=0;
        endcase
    end
''
```

Figure 4.3: synchronizer rtl module[**berlin**]

CHAPTER 5

FIFO Module

5.1 Functionality

The clock is used to synchronize the FIFO, and an active low resetn signal is used to reset the FIFO. Soft reset is an active high signal generated by the SYNCHRONIZE that resets the FIFO during the Router's time-out condition. Full and empty are the outputs that signal that the FIFO memory is full and that there is no data in the FIFO.

When the channel is busy, a memory of size 16X4 bits is generated to hold the incoming packet. During write and read operations, the write ptr and read ptr signals are created to point to the memory location. Full, empty, data out, and memory are all set to 0 upon reset (resetn=0).

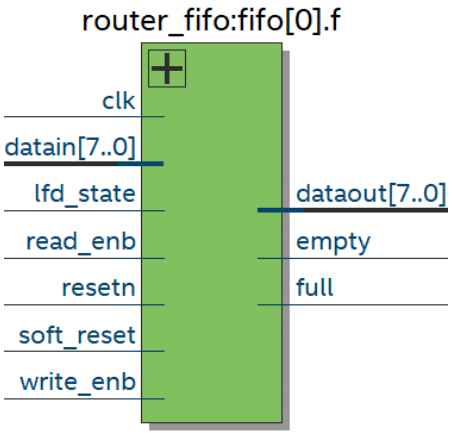

Figure 5.1: FIFO module[**berlin**]

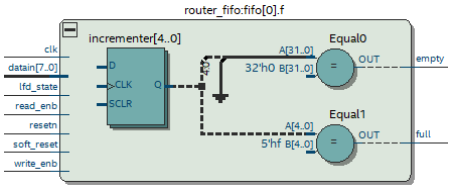

Figure 5.2: FIFO RTL module[**berlin**]

5.1.1 Write Operation

When write enable is high, the signal data in is sampled at the rising edge of the clock's edge. To avoid an over run scenario, write operations are only performed when the FIFO is not full.

5.1.2 Read Operation

When read enable is high, data is read from data out at the rising edge of the clock. To avoid under run conditions, read operations are only performed when the FIFO is not empty. When a header byte is read during a read operation, an internal counter is loaded with the payload length of the packet plus "1" (parity byte) and begins decreasing every clock until it reaches 0. The counter remains at zero until it is reloaded with a fresh packet payload length.

Full=0, empty=1 during the time out condition. Data out is driven to HIGH impedance state in two scenarios: When the FIFO m/m is fully read (header+payload+parity). Under the Router's time out condition.

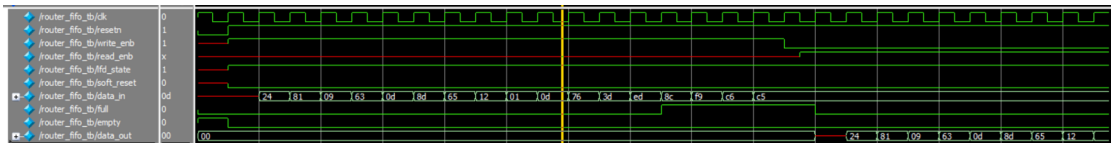

Figure 5.3: FIFO output[**berlin**]

CHAPTER 6

Top Module

6.1 Schematic

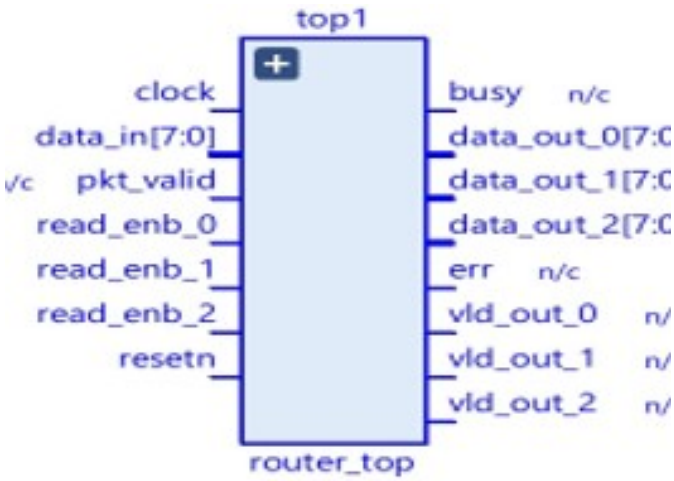

Figure 6.1: Top module[**berlin**]

The diagram below depicts how the modules are linked to one another.

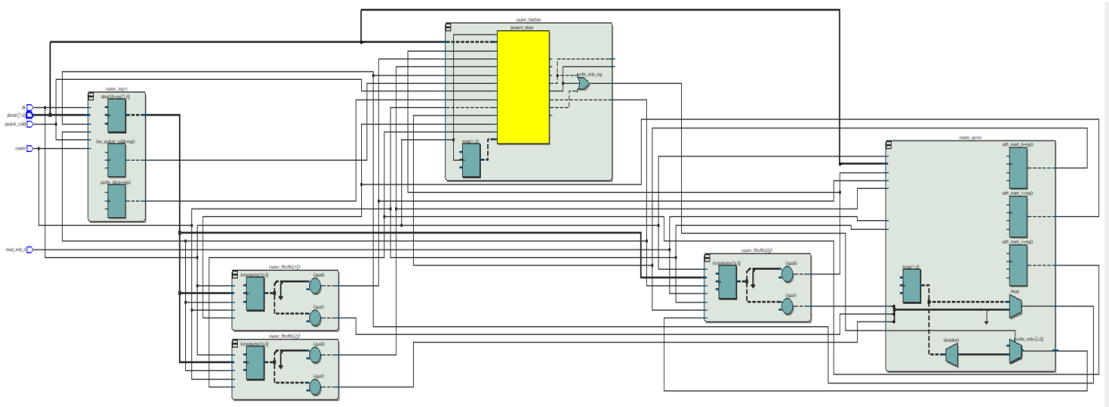

Figure 6.2: Top module schematic[**berlin**]

6.1.1 Outputs:

The read enable 2 is enabled in this case, and we are reading data from data out 2 from data in.

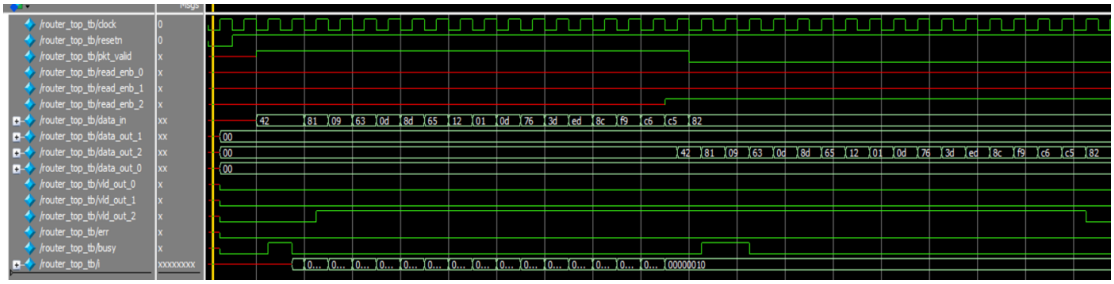

Figure 6.3: output data read from data out 2[**berlin**]

6.2 Conclusion:

The design is verified successfully each module. Resolved few issues like payload length, read write operations, and sending data to data out 0 and 1 when the read enable is high.

6.3 Appendix:

This drive contains simulations, verilog files and the projects created.
https://drive.google.com/drive/folders/1LW0jQLibtfExRbLUEWKKyp3ARA9wGFWU?usp

CHAPTER 7

References:

[1] Priyanka Rakshe and Pankaj Prajapati," "Router1x3 Protocol design with Virtual cut through Mechanism for Network on Chip (NoC)", Journal of Emerging Technologies and Innovative Research

[2] Ramya. C.B, Chethan. C.M, Praveen. S.G, Rakshith S, "Design and Implementation of Spartan 6 Series FPGA Board and Application of PID Controller for Robots", International Journal of Recent Advances in Engineering Technology (IJRAET) Volume-3, Issue -5, 2015

[3] Router 1X3 – RTL Design

[4] https://www.khanacademy.org/computing/computers-and-internet

[5] Neso Academy: https://www.youtube.com/watch?v=xwaqSZHK8eM