CpE 186 Computer Hardware Design
Professor: Dr. Jing Pang

Project 1 Report
Design of 8KB PCIe TLP Replay Buffer

Names: Alexis Ho, Andrew Peterson, Jayven Khoonsirivong,
Jeb Chua, Mario Palacios
California State University, Sacramento

Table of Contents

Introduction

PCIe's when compared to a PCI-X work more like a network and being a network, it must pass information along to make things operate properly differently than a PCI-X. The way PCIe's handle transaction from a CPU is by first generating a Write Packet, to transmit it over a bus which in return creates a Read Packet. The packets that are being created can be referenced as a Transaction Layer Packet (TLP), because it occurs in the Transaction Layer of the PCIe. The information contained in the TLP can be simplified to, "write this data to this address." However once the Read Packet is created it must respond with a Completion Packet. This is where the LCRC and Replay Buffer come in and help the PCIe. A LCRC is a function that acts as an error detector and will ensure the contents of a TLP gets across a single PCIe link. The goal of a Replay Buffer is to hold each TLP until the receiving device sends an implicit or explicit acknowledgement. In this lab we will going through majority of these steps, in order to get a more hands-on experience on how data is transferred within a computer. While also analyzing how important each step is in ensuring the correct data is being transmitted.

Transaction Layer Packet (TLP)

Design Purpose

The goal for this part of the lab is to manually create TLP packets. As explained previously, Transaction Layer Packets, occur in the Transaction Layer of the PCIe. Where a Write Packet is first created by the Requester and contains data and an address to which will be sent to the completer, that will then respond with a Read Packet. The completer will decode the data and respond with a Completion Packet. The packet is containing a value that is decoded and must match with the Requester, completing the Transaction Layer of the PCIe.

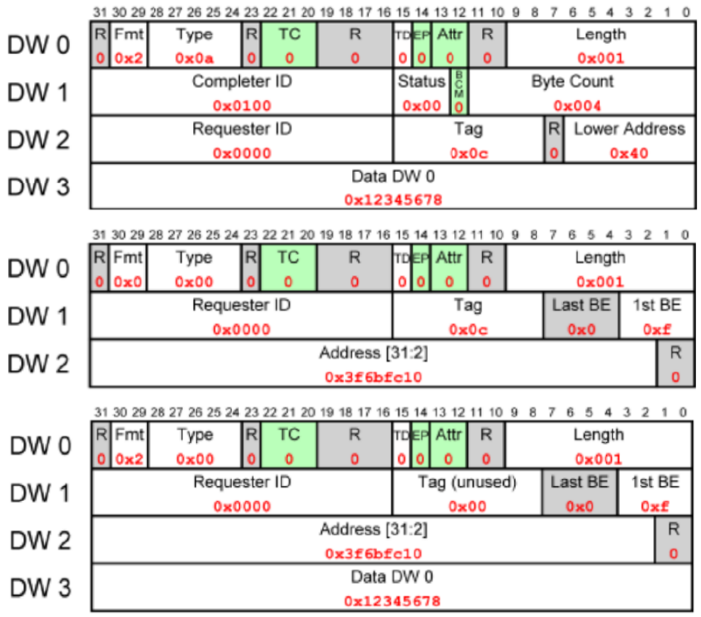

Figure 1. TLP Packet Examples for Write, Read, and Completion

Engineering Data

TLP Write Packet 1

	31	30 29 28	27 26 25 24	23 22 21 20 19 18 17 16	15	14	13 12 11 10	9 8 7 6 5 4 3 2 1 0
DW 0 0	R 0x2	Fmt 0x00	Type 0x00	R 0	TD 0	EP 0	Attr 0	R 0	Length 0x001
DW 1	Requester ID 0x0000				Tag(unused) 0x00		Last BE 0x0	1st BE 0xf
DW 2	Address [31:2] 0x11111111							R 0
DW 3	DATA DW 0 0xba5eba11							

TLP Write Packet 2

	31	30 29 28	27 26 25 24	23 22 21 20 19 18 17 16	15	14	13 12 11 10	9 8 7 6 5 4 3 2 1 0
DW 0 0	R 0x2	Fmt 0x00	Type 0x00	R 0	TD 0	EP 0	Attr 0	R 0	Length 0x001
DW 1	Requester ID 0x0000				Tag(unused) 0x00		Last BE 0x0	1st BE 0xf
DW 2	Address [31:2] 0x12345678							R 0
DW 3	DATA DW 0 0x012af707							

TLP Read Packet 1

	31	30 29 28	27 26 25 24	23 22 21 20 19 18 17 16	15	14	13 12 11 10	9 8 7 6 5 4 3 2 1 0
DW 0 0	R 0	Fmt 0x0	Type 0x00	R 0	TD 0	EP 0	Attr 0	R 0	Length 0x001
DW 1	Requester ID 0x0000				Tag 0x0c		Last BE 0x0	1st BE 0xf
DW 2	Address [31:2] 0x11111111							R 0

TLP Read Packet 2

	31	30 29 28	27 26 25 24	23 22 21 20 19 18 17 16	15	14	13 12 11 10	9 8 7 6 5 4 3 2 1 0
DW 0 0	R 0	Fmt 0x0	Type 0x00	R 0	TD 0	EP 0	Attr 0	R 0	Length 0x001
DW 1	Requester ID 0x0000				Tag 0x0c		Last BE 0x0	1st BE 0xf
DW 2	Address [31:2] 0x12345678							R 0

TLP Completion Packet 1

	BYTE 0								BYTE 1								BYTE 2								BYTE 3							
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DW 0 0	R 0x2			Fmt 0x0a			Type 0		R 0		TC 0			R 0			TD 0	EP 0	Attr 0		R 0				Length 0x001							
DW 1	Completed ID 0x0100																Status 0x00				BCM 0	Byte Count 0x004										
DW 2	Requester ID 0x0000																Tag 0x0c				R 0	Lower Address 0x44										
DW 3	DATA DW 0 0xba5eba11																															

TLP Completion Packet 2

	BYTE 0								BYTE 1								BYTE 2								BYTE 3							
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DW 0 0	R 0x2			Fmt 0x0a			Type 0		R 0		TC 0			R 0			TD 0	EP 0	Attr 0		R 0				Length 0x001							
DW 1	Completed ID 0x0100																Status 0x00				BCM 0	Byte Count 0x004										
DW 2	Requester ID 0x0000																Tag 0x0c				R 0	Lower Address 0xE0										
DW 3	DATA DW 0 0x012af707																															

Results Discussion

Understanding the formatting for the packets was crucial in order to manually create them. However, after careful reading of the book and having Dr. Pang explain during class it was not so difficult. This section also helped understand the Transaction Layer of PCIe architecture and how packets are assembled and passed for communication.

Link Cyclic Redundancy Code (LCRC)

Design Purpose

The goal for this part of the lab is to create the LCRC bits for each TLP packet. These bits play an important role, because it acts as an error detector. Each packet is given a unique incremental Sequence Number, that will be used to ensure if a TLP was successfully received in the order they were sent. This makes it possible to detect missing TLPs at the Receiver's Data Link Layer.

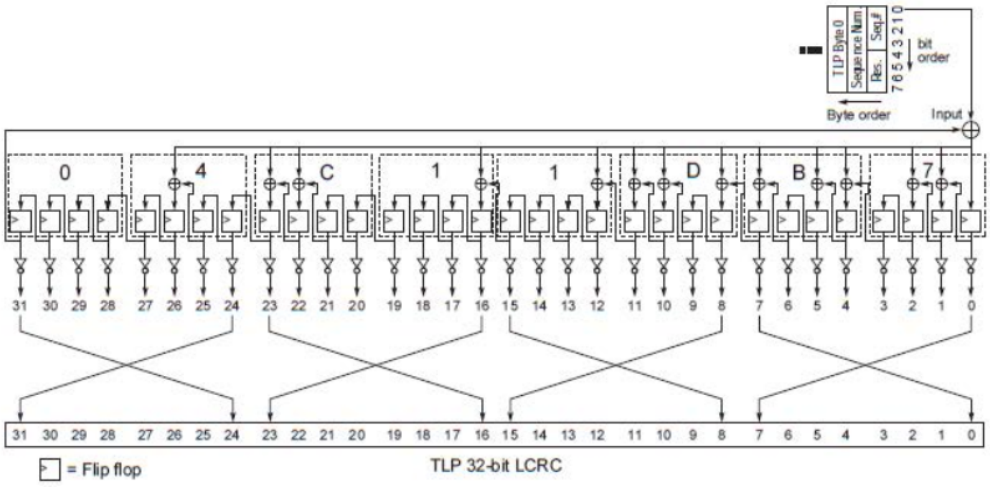

Figure 2. Link Cyclic Redundancy Cycle (*LCRC*) Diagram

Engineering Data

```
 0 input = 0000 output = xxxxxxxx
 5 input = 0000 output = 00000000
10 input = 12ab output = 00000000
15 input = 12ab output = ea0f7f5f
20 input = fe87 output = ea0f7f5f
25 input = fe87 output = 1f88e91a
```

Verilog Design

```
`timescale 1ns / 1ps
module LCRC(clk, reset, in, out);
    input clk, reset;
    input [15:0] in;
    output reg [31:0] out;

    integer byteCount, bitCount;

    reg [15:0] tmp;
    reg [7:0] byte;
    reg [31:0] current;
    reg [31:0] previous;
    reg xorBit;

    always @(posedge clk)
    begin
        if(reset)
        begin
            out = 0;
            previous[31:0] <= 32'b0000_0100_1100_0001_0001_1101_1011_0111;
            byteCount = 2;
            bitCount = 1;
            byte = in[7:0];
        end
        else
        begin
            byteCount = 2;
            bitCount = 1;
            byte = in[7:0];
            for(byteCount = 2; byteCount > 0; byteCount = byteCount - 1)
            begin
                for(bitCount = 1; bitCount <= 8; bitCount = bitCount + 1)
```

```
            begin
                xorBit = previous[31] ^ byte[bitCount - 1];

                current[0] = xorBit;
                current[1] = previous[0] ^ xorBit;
                current[2] = previous[1] ^ xorBit;
                current[3] = previous[2];
                current[4] = previous[3] ^ xorBit;
                current[5] = previous[4] ^ xorBit;
                current[6] = previous[5];
                current[7] = previous[6] ^ xorBit;

                current[8] = previous[7] ^ xorBit;
                current[9] = previous[8];
                current[10] = previous[9] ^ xorBit;
                current[11] = previous[10] ^ xorBit;
                current[12] = previous[11] ^ xorBit;
                current[13] = previous[12];
                current[14] = previous[13];
                current[15] = previous[14];

                current[16] = previous[15] ^ xorBit;
                current[17] = previous[16];
                current[18] = previous[17];
                current[19] = previous[18];
                current[20] = previous[19];
                current[21] = previous[20];
                current[22] = previous[21] ^ xorBit;
                current[23] = previous[22] ^ xorBit;

                current[24] = previous[23];
                current[25] = previous[24];
                current[26] = previous[25] ^ xorBit;
                current[27] = previous[26];
                current[28] = previous[27];
                current[29] = previous[28];
                current[30] = previous[29];
                current[31] = previous[30];

                //output arrangment
                out[31] = current[24];
                out[30] = current[25];
                out[29] = current[26];
                out[28] = current[27];
                out[27] = current[28];
                out[26] = current[29];
                out[25] = current[30];
                out[24] = current[31];
```

```
                    out[23] = current[16];
                    out[22] = current[17];
                    out[21] = current[18];
                    out[20] = current[19];
                    out[19] = current[20];
                    out[18] = current[21];
                    out[17] = current[22];
                    out[16] = current[23];


                    out[15] = current[8];
                    out[14] = current[9];
                    out[13] = current[10];
                    out[12] = current[11];
                    out[11] = current[12];
                    out[10] = current[13];
                    out[9] = current[14];
                    out[8] = current[15];


                    out[7] = current[0];
                    out[6] = current[1];
                    out[5] = current[2];
                    out[4] = current[3];
                    out[3] = current[4];
                    out[2] = current[5];
                    out[1] = current[6];
                    out[0] = current[7];

                    previous = current;
                end
                tmp = (in >> ( (byteCount - 1) * 8));
                byte = tmp[7:0];
            end
        end
    end
endmodule
```

Verilog Testbench

```
`include "LCRC.v"
module LCRC_tb;

reg [15:0] in;
reg clk, reset;
```

```
wire [31:0] out;


initial
    $monitor ($time, " input = %h output = %h", in, out);


LCRC LCRC_instant(.clk(clk), .reset(reset), .in(in), .out(out));

initial begin
    clk = 0;
    forever #5 clk <= ~clk;
end

initial begin

         reset = 1; in=16'b0;
    #10; reset = 0; in=16'b0001_0010_1010_1011;  //enters 12ABhex, outputs
EA0F_7F5Fhex
    #10; in=16'b1111_1110_1000_0111;             //enters FE87hex without resetting,
outputs 1F88_E91Ahex
        #10     $finish;

end
endmodule
```

Results Discussion

The LCRC works perfectly and allows for a new series of bits to be able to compare with the incoming TLP packet. We can see on the simulation results it can turn a 16bit input and turn it into a new 32bit sequence number.

Replay Buffer

Design Purpose

The goal of this part of the lab, is to create a fully functioning Replay Buffer, that will hold each TLP that is transmitted until it is positively acknowledged or negatively acknowledged by the receiving device. This process is also known as the ACK/NAK protocol. Where once it is positively acknowledged (ACK) it will be removed from the Replay Buffer allowing making room for more TLPs. However, if it is negatively acknowledged (NAK) then that TLP and any other TLPs transmitted after will be retransmitted.

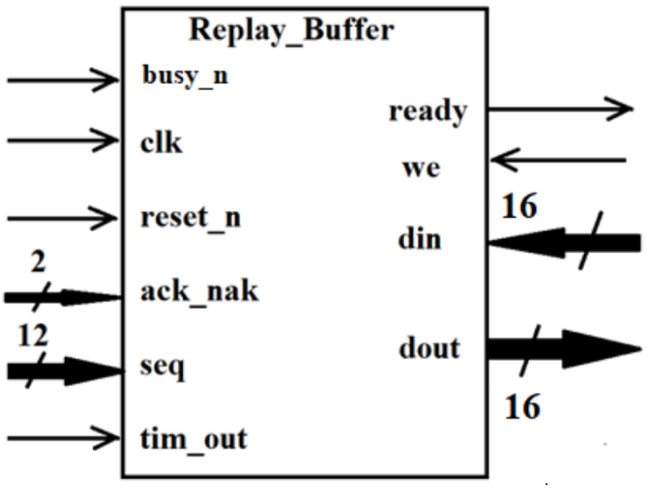

Figure 3. Replay Buffer Diagram

Engineering Data

Verilog Design

```
`include "fsm.v"
`include "replayBuff.v"
module topLevel (input clk, reset_n, busy_n, tim_out, we,
                 input [1:0] ack_nak,
                 input [15:0] din,
                 input [11:0] seq,
                 output ready,
                 output [15:0] dout);

    fsm f1(.reset_n(reset_n), .clk(clk), .ack_nak(ack_nak), .tim_out(tim_out),
.busy_n(busy_n), .we(we));
    replayBuff r1(.clk(clk), .reset_n(reset_n), .tim_out(tim_out), .we(we),
.ack_nak(ack_nak), .seq(seq), .din(din), .ready(ready), .dout(dout));
endmodule
```

```
module fsm(reset_n, clk, ack_nak, tim_out, busy_n, we);
    input reset_n, clk, tim_out, busy_n, we;
    input [1:0] ack_nak;

    reg ready;

    reg[2:0] cs, ns;
    parameter s0 = 3'b000, s1 = 3'b001, s2 = 3'b010,
              s3 = 3'b011, s4 = 3'b100;
```

```
    always@(cs or tim_out or we or ack_nak or busy_n)
    begin
        case(cs)
            s0:
                if (busy_n) ns = s1;
                else ns = s0;
            s1:
                if(we) ns = s2;
                else ns = s1;
            s2:
                if (tim_out) ns = s4;
                else ns = s1;
            s3:
                if (ack_nak == 2'b10) ns = s4;
                else ns = s1;
            s4:
                if (ack_nak == 2'b01) ns = s3;
                else ns = s1;
            default: ns = s0;
        endcase
    end

    always@(posedge clk or negedge reset_n)
    begin
        if(reset_n) cs <= s0;
        else cs <= ns;
    end

endmodule


module replayBuff (input clk, reset_n, tim_out, we,
                   input [1:0] ack_nak,
                   input [11:0] seq,
                   input [15:0] din,
                   output reg ready,
                   output reg [15:0] dout);

    reg [15:0] packetHolder [4095:0];
    integer cnt_Wr, cnt_Rd;

    always@(posedge clk or negedge reset_n)
    begin
        if(!reset_n)
        begin
            dout <= 16'b0;
```

```
            cnt_Rd = 0; cnt_Wr = 0;
        end
        else if(tim_out) // checking for ack_nak timeout
        begin
            dout = 16'b0;
        end
        else begin
            if(ack_nak == 01 || ack_nak == 10)  // checking on ack_nak is working
            begin
                if(!we && !(cnt_Wr == 0)) // should read after ack
                begin
                    dout = packetHolder[seq[9:0]];
                    cnt_Rd = cnt_Rd + 1;
                    if(cnt_Wr == 4095) cnt_Wr = 0;
                end
                else dout = 0;
            end
            else begin  // no ack for DLLP
                if(we && !(cnt_Wr == 4095)) // overwriting
                begin
                    packetHolder[seq[9:0]] = din;
                    cnt_Wr = cnt_Wr + 1;
                    if(cnt_Rd == 4095) cnt_Rd = 0;
                end
                else dout = 16'b0;
            end
        end
    end

    always@(posedge clk or negedge reset_n)
    begin
        if(!reset_n || tim_out) ready = 1;
        if (cnt_Rd == 4095) ready = 0;
    end
endmodule
```

Verilog Testbench

```
`include "topLevel.v"
module topLevel_tb;
    reg clk, reset_n, busy_n, tim_out, we;
    reg [1:0] ack_nak;
    reg [15:0] din;
    reg [11:0] seq;
    wire ready;
    wire [15:0] dout;
```

```
    initial
        $vcdpluson;

    topLevel t1 (.clk(clk), .reset_n(reset_n), .busy_n(busy_n), .tim_out(tim_out),
.we(we), .ack_nak(ack_nak), .din(din), .seq(seq), .ready(ready), .dout(dout));

    always@(posedge clk)
    begin
        $display("      Clock       = %b    Reset       = %b", clk, reset_n);
        $display("      Data In     = %h    Sequence    = %h", din, seq);
        $display("      Ack/Nak     = %b    Timeout     = %b", ack_nak, tim_out);
        $display("      Busy        = %b    Reday       = %b", busy_n, ready);
        $display("            Write Enable      = %b", we);
        $display("            Data Out          = %h\n\n", dout);
    end

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial
    begin
             reset_n = 1; busy_n = 0; tim_out = 0; we = 0;
             din = 16'h0001; seq = 12'h001; ack_nak = 2'b00;
        #100; reset_n = 0;
        #10; we = 1; busy_n = 1;
        #10; ack_nak = 2'b10; we = 0;
        #100; reset_n = 1;
        #10; we = 1;
        #10; tim_out = 1;
        #10; $finish;
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(1);
    end
endmodule
```

Results Discussion

The code did not work, and I believe it has to do something to do with our replay buffer. It seems that it is not taking in the bits it needs to hold.

Conclusion

This project over all was a difficult one because it requires a lot of communication not only with the team but with each other components. The LCRC component of the project was completed to our best ability. Nonetheless we did learn vast amount on how the TLP packets are formed and how they are to communicate with devices, and how important the Replay Buffer is in order to keep things going smoothly. There a lot of errors along the way, but in the end this is our best result.